Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 67 (2007) 271–285
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories ☆

Kamer Kaya, Bora Uçar, Cevdet Aykanat*

Department of Computer Engineering, Bilkent University, 06800 Ankara, Turkey

Received 3 August 2005; received in revised form 15 October 2006; accepted 21 November 2006
Available online 19 January 2007

Abstract

We consider the problem of scheduling an application on a computing system consisting of heterogeneous processors and data repositories. The application consists of a large number of file-sharing otherwise independent tasks. The files initially reside on the repositories. The processors and the repositories are connected through a heterogeneous interconnection network. Our aim is to assign the tasks to the processors, to schedule the file transfers from the repositories, and to schedule the executions of tasks on each processor in such a way that the turnaround time is minimized. We propose a heuristic composed of three phases: initial task assignment, task assignment refinement, and execution ordering. We experimentally compare the proposed heuristics with three well-known heuristics on a large number of problem instances. The proposed heuristic runs considerably faster than the existing heuristics and obtains 10–14% better turnaround times than the best of the three existing heuristics.

Keywords: Scheduling; Heterogeneous computing systems; Grid computing

1. Introduction

In this work, we propose heuristic algorithms for scheduling an application on a heterogeneous computing system. The application is composed of a large number of independent but file-sharing tasks. The computing system consists of heterogeneous processors and data repositories that store input files. The repositories are decoupled from the processors. The processors are connected to the repositories through a heterogeneous interconnection network. Our aim is to schedule the task executions on processors and to schedule the input file transfers in such a way that the turnaround time, i.e., the completion time of the application is minimized. Recently, similar scheduling frameworks have been studied [14,15,20–23,26,28] for the Grid environments.

By file sharing, we mean that an input file may be requested by a number of tasks. Files are assumed to be stored in data repositories without replication, e.g., each file is stored in a particular repository. Once the tasks are assigned to the processors, the files should be transfered from the repositories to the processors. A task execution can start after its input files are delivered to the respective processor. We assume that once a file is transfered to a processor, it can be used by all tasks assigned to the same processor without any additional cost. The network interconnecting the repositories and the processors is heterogeneous, i.e., the costs of transferring a certain file between different source and destination pairs are not necessarily equal. We work under the one-port communication model, i.e., a data repository or a processor can, respectively, send or receive at most one file at a given time. In order to minimize the turnaround time, the scheduler must decide the task-to-processor assignment, the order of file transfers, and the order of task executions on each processor.

We review some recent works and state the contributions of the current manuscript in Section 2. The application and computing models which characterize the target scheduling problem are discussed in Section 3. The proposed scheduling heuristic

☆ This work is partially supported by the Scientific and Technical Research Council of Turkey under projects 106E069 and 105E065, and by the European Commission FP6 project SEE-GRID-2 with contract no. 031775.
* Corresponding author. Fax: +90 312 266 4047.
E-mail addresses: kamer@cs.bilkent.edu.tr (K. Kaya), ubora@cs.bilkent.edu.tr (B. Uçar), aykanat@cs.bilkent.edu.tr (C. Aykanat).

0743-7315/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2006.11.004

is presented in Section 4. Section 5 contains experimental assessment of the proposed heuristics.

2. Related work and contributions

This work is based on the works of Casanova et al. [14,15], Giersch et al. [20–23], Kaya and Aykanat [26], and Khanna et al. [28]. Below, we review these works and highlight our contributions.

2.1. Single repository: master–slave platforms

Casanova et al. [14,15] consider task scheduling for *AppLeS Parameter Sweep Template* (APST) applications [10]. APST applications have large number of tasks which share input data files. The scheduling problem is to assign the tasks to heterogeneous processors, to schedule the file transfers from a single master, and to schedule the executions of the tasks in each processor to minimize the turnaround time. They propose three well-known heuristics *MinMin*, *MaxMin* and *Sufferage*. These three heuristics are extensions of the methods proposed in [32] for scheduling independent tasks. For a given set of tasks to be scheduled, these three heuristics compute the minimum completion time (MCT) of each task over the processors to find the best processor for each task. Then, the execution of a task on a processor—selected according to a task selection policy—and the file transfers of the task to the respective processor are scheduled. Upon scheduling a task and the associated file transfers, these heuristics continue with the remaining tasks. For the *MinMin* heuristic, the task selection policy is to choose the task with the minimum MCT. For the *MaxMin* heuristic, the policy is to choose the task with the maximum MCT. Task selection operation for the *Sufferage* heuristic requires computing not only the MCT but also the second minimum completion time (SMCT) for each task. The difference between the SMCT and MCT values is defined as the *sufferage* value of the task when it is not assigned to its best processor. For the *Sufferage* heuristic, the task selection policy is to choose the task with the maximum sufferage value.

Giersch et al. [20,23] propose another family of heuristics for the same scheduling problem. These heuristics reduce the time complexity of the aforementioned heuristics by an order of magnitude without degrading the solution qualities. The proposed heuristics create a list of tasks for each processor and sort these lists according to an objective function. Giersch et al. propose six objective functions which are formulated by different combinations of the computation and communication costs. The computation cost is defined as the time required for the execution of a task on a processor, and the communication cost is defined as the time required for transferring the files required by a task to a processor. The *communication* and *computation* objective functions are defined as the communication and computation costs, respectively; the *duration* objective function is defined as the sum of computation and communication costs; the *payoff* objective function is defined as the ratio of the computation cost to the communication cost; the *advance* objective function is defined as the difference between the computation and communication costs. For the communication, computation, and duration objective functions, the tasks are sorted in the increasing order of their objective values. For the advance and payoff objective functions, the tasks are sorted in the decreasing order of their objective values. The sixth objective function applies Johnson's algorithm (see [20]) in which the tasks whose communication costs are smaller than their computation costs are scheduled in the increasing order of communication costs, and the other tasks are scheduled in the decreasing order of computation costs. After creating and sorting a list of tasks for each processor, the tasks are scheduled one by one. The completion time of the first task in each processor's list is computed and a task–processor pair with the MCT is selected as a part of the schedule. After marking the scheduled task, the proposed heuristics commence on scheduling the remaining tasks. Note that by computing only one completion time value for each processor during selecting a task–processor pair, these heuristics become an order of magnitude faster than the ones given in [14,15].

All of the aforementioned heuristics are constructive and are based on greedy choices that depend on momentary completion times of the tasks. Kaya and Aykanat [26] attack the same scheduling problem using a different paradigm. In [26], a hypergraph model is used to represent the tasks, files, and their interactions, and a hypergraph-partitioning-like formulation is devised for the problem. A three-phase approach which involves initial task assignment, refinement, and execution ordering phases is proposed. In the task assignment phase, the heuristics proposed in [20,23] are used to obtain an initial assignment of tasks to processors. Kaya and Aykanat observe that the turnaround time cannot be determined from task-to-processor assignments. However, by proposing lower and upper bounds on the turnaround time, they relate the turnaround time to the task-to-processor assignments. Iterative-improvement-based heuristics are proposed for refining the task assignments by improving the lower and upper bounds. In the execution ordering phase, the order of task executions and hence file transfers are determined by preserving the refined task-to-processor assignment.

2.2. Multiple repositories

Different from the above works, Giersch et al. [21,22] deal with a more general scheduling problem. Instead of a master–slave system, they assume a fully decentralized system composed of servers linked through an interconnection network. Each server acts both as a file repository and as a computing node consisting of a cluster of heterogeneous processors. Files are initially assumed to be stored at one or more repositories. In addition to the objectives stated above for the master–slave system, the scheduler has to decide how to route the files from repositories to other servers. The paper [21] establishes NP-completeness results for this instance of the scheduling problem and proposes several practical heuristics. The proposed heuristics include extensions of the *MinMin* heuristic, *Sufferage* heuristic, and the heuristics presented in the previous works of the authors [20,23].

Khanna et al. [28] deal with a scheduling problem for a slightly different computing system. They assume a decoupled system consisting of processors and storage nodes (repositories) connected in a local area network. As in the works discussed above, the application consists of file-sharing otherwise independent tasks. They assume that the computation time of a task is a linear function of the total size of the requested files, and hence the expected execution time of a task can be calculated as a constant multiple of the total size of the requested files. This execution time model incorporates the local disk access costs in addition to the file transfer and processing costs. Under these assumptions, the problem addressed in [28] can be specified as scheduling file-sharing tasks on a set of homogeneous processors connected to a set of storage nodes through a uniform (homogeneous) network. Khanna et al. also use a hypergraph to model the application. They propose a two-stage strategy for scheduling task executions and file transfers. In the first stage, they partition the tasks into groups—one group to be assigned to a processor—using a hypergraph partitioning tool. In the second stage, they order the tasks in each group and file transfers from the storage nodes. Due to the homogeneous processors and network assumptions, hypergraph partitioning objective and constraint correspond, respectively, to minimizing total volume of file transfers (excluding local access) and maintaining a balance on the loads (including I/O) of the processors. Khanna et al. report better performance than some existing heuristics, including *MinMin*, *MaxMin*, and *Sufferage*, on two real world applications.

2.3. Contributions

In this work, we deal with a scheduling problem for a distributed system similar to that considered in [28]. We assume a system consisting of decoupled heterogeneous processors and repositories linked through a heterogeneous interconnection network. We extend the three-phase approach of Kaya and Aykanat [26] to this more involved computing system. In the initial task assignment phase, we use the heuristics proposed by Giersch et al. [20,23] as is done in [26]. For the refinement phase, we define a new lower bound to establish a relation between a task-to-processor assignment and the turnaround time. Kaya and Aykanat propose to use a loose upper bound on the turnaround time. In this work, we define a new objective function which is likely to induce an upper bound. We also present appropriate gain functions for the proposed bounds. The existence of multiple repositories makes the execution ordering phase more complicated. In a master–slave system, when a task is determined to be scheduled, all of the necessary file transfers associated with the task are also scheduled one after another. That is, the master is never left idle. However, in the target scheduling problem repositories or processors can be idle, because of the one-port communication model. Due to the existence of idle times, a file transfer for a task can be scheduled between already scheduled transfers. This is an issue for the extensions of *MinMin*, *Sufferage*, and for those presented in [20,23]. The works [21,28] try to schedule the file transfers by inserting them at the idle times. We also use this approach for scheduling the file transfers.

The proposed scheduling heuristic is static in the sense that the schedule is determined before the execution of the application and is kept fixed throughout the execution. However, large and non-dedicated computing systems may require adapting the schedule to run-time changes such as processor or link failures, increases in the workload and network traffic. For these issues, see [11,19,32]. See [33,34] for different perspectives on the application of master–slave computing. For many facets of scheduling for Grid environments, see [9,12,29].

3. Framework

3.1. Application model

The application is defined as a two tuple $\mathcal{A} = (\mathcal{T}, \mathcal{F})$, where $\mathcal{T} = \{1, 2, \ldots, T\}$ denotes the set of T tasks, and $\mathcal{F} = \{1, 2, \ldots, F\}$ denotes the set of F input files. Each task t depends on a subset of files denoted by $\text{files}(t)$; these files should be delivered to the processor that will execute the task t. We extend the operator $\text{files}(\cdot)$ to a subset of tasks $\mathcal{S} \subseteq \mathcal{T}$ such that $\text{files}(\mathcal{S}) = \bigcup_{t \in \mathcal{S}} \text{files}(t)$ denotes the set of files that the set $\mathcal{S}$ of tasks depend on. Apart from sharing the input files, there are no dependencies and interactions among the tasks. The size of a file f is denoted by $w(f)$. We extend the operator $w(\cdot)$ to a subset $\mathcal{E} \subseteq \mathcal{F}$ of files such that $w(\mathcal{E})$ denotes the total size of the files in $\mathcal{E}$, i.e., $w(\mathcal{E}) = \sum_{f \in \mathcal{E}} w(f)$. We use $|\mathcal{A}|$ to denote the total number of file requests in the application, i.e., $|\mathcal{A}| = \sum_{t \in \mathcal{T}} |\text{files}(t)|$.

As in [26], we use a hypergraph $\mathcal{H}_{\mathcal{A}} = (\mathcal{T}, \mathcal{F})$ to model the application $\mathcal{A} = (\mathcal{T}, \mathcal{F})$. Recall that a hypergraph is defined as a set of vertices and a set of hyperedges (nets) each of which contains a subset of vertices [8]. We use $\mathcal{T}$ and $\mathcal{F}$ to denote, respectively, the vertex and net sets of the hypergraph. In this setting, the net corresponding to the file f contains the vertices that correspond to the tasks depending on f. Fig. 1 contains an example hypergraph model.

3.2. Computing model

The tasks are to be executed on a heterogeneous system consisting of a set $\mathcal{P} = \{1, 2, \ldots, P\}$ of P computing resources, and a set $\mathcal{R} = \{1, 2, \ldots, R\}$ of R repositories. Each computing resource can be any computing system ranging from a single processor workstation to a parallel computer. Throughout the paper we use processor to refer to any type of computing resource. The set of files stored on a repository r is denoted as $\mathcal{F}(r)$. We assume that the files are not duplicated, i.e., $\mathcal{F}(r) \cap \mathcal{F}(s) = \emptyset$ for $r \neq s$. We use $\text{store}(f)$ to denote the repository which holds the file f.

We use $\Pi = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_P\}$ to denote a partition on the vertices of the hypergraph $\mathcal{H}_{\mathcal{A}}$ and hence an assignment of the tasks to the processors. In other words, we denote the set of tasks assigned to processor p as $\mathcal{T}_p$. Given a task assignment, we use Λ_f to denote the set of processors to which the file f is to be transfered, i.e., $\Lambda_f = \{p \mid f \in \text{files}(\mathcal{T}_p)\}$. The three dashed

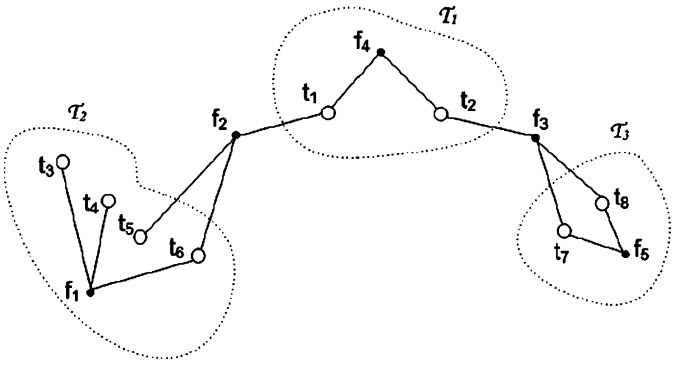

Fig. 1. Hypergraph model $\mathcal{H}_{\mathcal{A}} = (\mathcal{T}, \mathcal{F})$ for an application with a set of eight tasks $\mathcal{T} = \{1, 2, \ldots, 8\}$ and a set of five files $\mathcal{F} = \{1, 2, \ldots, 5\}$. Vertices are shown with empty circles and correspond to the tasks; hyperedges (nets) are shown with filled circles and correspond to the files. File requests are shown with lines connecting vertices and nets. For example, task t_6 needs files f_1 and f_2 and hence vertex t_6 is in the nets f_1 and f_2. A 3-way partition on the vertices of the hypergraph is shown with dashed curves encompassing the vertices.

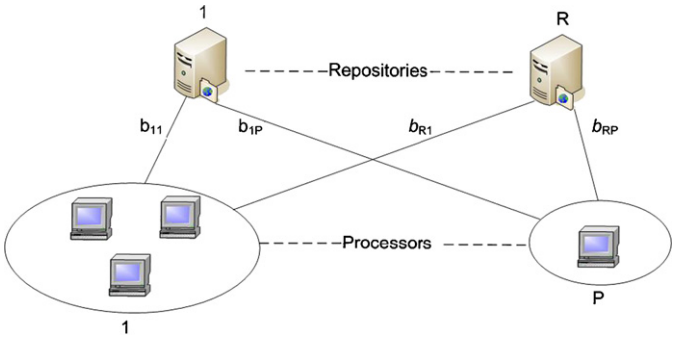

Fig. 2. Computing system.

curves encompassing the vertices in Fig. 1 show a partition on the vertices of the hypergraph, and hence an assignment of tasks to processors. For example, the tasks t_1 and t_2 are assigned to the processor 1 since the vertices t_1 and t_2 are in $\mathcal{T}_1$.

We assume the one-port model for the file transfers from the repositories to the processors. In this model, a processor can receive at most one file, and a repository can send at most one file at a given time. This model is deemed to be realistic [5,7,35] and it is prevalent in the scheduling for Grid computing literature, however, alternatives exist (see [4,13]). Task executions and file transfers can overlap at a processor. That is, a processor can execute a task while it is downloading a file for another task. The file transfer operations take place only between a repository and a processor. We disregard the congestion in the communication network during the file transfers. In other words, each processor is assumed to be connected to all repositories through direct communication links. Note that the resulting topology is a complete bipartite graph ($K_{P\times R}$). Computing platforms of this topology are called heterogeneous fork-graphs [20,23] when $R = 1$. Such complete graph models are used to abstract wide-area networking infrastructures [13]. The network heterogeneity is modelled by assigning different bandwidth values to the links between the repositories and the processors. We use b_{rp} to represent the bandwidth from the repository r to the processor p. We use the linear cost model [6,13] for file transfers, i.e., transferring the file f from the repository r to the processor p takes $\frac{w(f)}{b_{rp}}$ time units. Fig. 2 displays the essential properties of the computing system.

The task and processor heterogeneity are modelled by incorporating different execution costs for each task on different processors. The execution-time values of the tasks are stored in a $T \times P$ expected time to compute (ETC) matrix. We use x_{tp} to denote the execution time of the task t on the processor p. The ETC matrices are classified into two categories [1]. In the *consistent* ETC matrices, there is a special structure which implies that if a machine has a lower execution time than another machine for some task, then the same is true for other tasks. The *inconsistent* ETC matrices have no such special structure. In this work, we adopt inconsistent ETC matrices as they can model a variety of computing systems and applications that arise in Grid environments.

3.3. Objective function

The cost of a schedule is the turnaround time, i.e., the length of the time interval whose start and end points are defined by the start of the first file transfer operation and the completion of the last task execution, respectively. Therefore, the objective of the scheduling problem is to assign the tasks to processors, to determine the order in which the files are transfered from the repositories to the processors, and to determine the task execution order on each processor in order to minimize the turnaround time. Given that scheduling file-sharing tasks on heterogeneous master–slave systems is NP complete [20], the NP completeness of the target scheduling problem follows easily.

4. Proposed scheduling heuristic

We propose a heuristic consisting of three phases. In the first phase, we find five initial task-to-processor assignments using the objective functions proposed in [20,23]. In the second phase, we refine these five assignments independently. These first two phases determine five task-to-processor assignments. In the third phase, the order of task executions and file transfers are determined for each of the five task-to-processor assignments found in the second phase and the best schedule is returned as the solution.

The reason behind constructing and refining task-to-processor assignments rather than complete schedules is given by Kaya and Aykanat [26] as follows. A refinement method which uses task reassignments to improve the turnaround time would have a global perturbation on the given schedule. This is because removing a task from a processor and scheduling it among the previously scheduled tasks of other processors affects the initialization and completion times of executions of all waiting tasks [26]. Such a global effect leads to a huge neighborhood definition. Exploring the search space under a huge neighborhood definition is time consuming. More im-

portantly, an iterative-improvement method will most likely stuck to a part of the search space and will hardly find paths to improve the turnaround time. Therefore, a task reassignment-based refinement method focusing on the turnaround time of a schedule does not seem effective (see [2, Section 3]). With this reasoning, Kaya and Aykanat propose the refinement of task-to-processor assignments to pave the way for efficient schedules. We follow the same approach here.

4.1. Phase 1: initial task assignment

In this phase, we use the previously proposed constructive heuristics [20,23] to find an initial task-to-processor assignment. These heuristics were originally developed for creating a schedule for a set of file-sharing tasks on heterogeneous master–slave systems. We have modified these heuristics to handle multiple data repositories and implemented them in such a way that their outputs are task-to-processor assignments rather than complete schedules.

Recall from Section 2.1 that the heuristics proposed by Giersch et al. [20,23] create a list of tasks for each processor sorted with respect to one of the five (except Johnson's algorithm) objective functions. Then, a task–processor pair with the MCT is selected among the first tasks in the processors' lists. Upon selecting the task–processor pair, the execution of the task on the selected processor and the transfer of the files needed by the task are scheduled. These steps are repeated on the reduced lists of the processors until all of the tasks are scheduled.

Determining the MCT of a task on a processor requires scheduling the associated file transfers. This scheduling problem is known as the communication scheduling with respect to a history problem, and it is NP-complete [21]. The intricacy of the problem stems from the fact that both the repositories and the processors may have idle times in their time lines, because of the one-port communication model. A solution method should schedule the remaining file transfers wisely to fill or reduce these idle times.

Since scheduling the file transfer operations is NP-complete, we resort to the heuristics similar to the insertion scheduling methods used in [21]. We maintain a list of free time intervals for each repository and processor. The last interval starts from the end of the last file transfer operation and extends to infinity. When the transfer of a file f is to be scheduled from a repository r to a processor p, we intersect the free-time intervals of r and p and find an interval $[i_s, i_f]$ that can accommodate $w(f)/b_{rp}$ time units, and mark the interval $[i_s, i_s + w(f)/b_{rp}]$ in both lists as used. We follow the first fit approach, i.e., allocate the earliest time interval in which f can be transfered from r to p.

Time complexity: Selecting a task–processor pair requires P completion time calculations. For each $f \in \text{files}(t)$, we intersect the interval lists of the processor p and the repository that stores f. Each intersection takes at most $\frac{FP}{R} + F$ time, since a repository's interval list can contain at most $\frac{FP}{R}$ intervals, and a processor's interval list can contain at most F intervals. Since the intervals are in sorted order, the intersection can be computed in time linearly proportional to the cardinalities of the interval lists. Therefore, completion time calculation for a task t and a processor p requires $\sum_{f\in\text{files}(t)} O(\frac{FP}{R})$ time. Hence, the time complexity of initial task assignment phase is

$$\sum_{t=1}^{T}\sum_{p=1}^{P}\sum_{f\in\text{files}(t)} O\left(\frac{FP}{R}\right) = \sum_{t=1}^{T}\sum_{f\in\text{files}(t)} O\left(P\frac{FP}{R}\right) = O\left(P^2\frac{F}{R}|\mathcal{A}|\right).$$

4.2. Phase 2: refining task assignments

In this phase, we refine the task-to-processor assignments obtained in the first phase by using the approach proposed by Kaya and Aykanat [26]. Their approach is to define a lower bound and an upper bound on the turnaround time under a given task-to-processor assignment, and then to try to close the gap between these bounds. A crucial point in this approach is that the perturbations in the solutions (task-to-processor assignments) are local under the task reassignment operation and therefore the objective functions (e.g., the lower and upper bounds) are smooth over the search space.

4.2.1. Bounds on the turnaround time

Given a task-to-processor assignment Π, we identify three different cost components that are associated with the turnaround time. These are

- CompTime(Π): Parallel task execution time. In particular, this is the maximum task execution time spent by a single processor.
- UploadTime(Π): File transfer cost from the repositories' perspective. In particular, this is the maximum file transfer time spent by a single repository.
- DownloadTime(Π): File transfer cost from the processors' perspective. In particular, this is the maximum file download time spent by a single processor.

Since the assignment Π is clear from the context, we drop Π in the following text. We now formulate these cost components. Recall that $\mathcal{T}_p$ and x_{tp} denote the set of tasks assigned to the processor p and the execution cost of task t on processor p. The total execution time of the processor p is

$$X_p = \sum_{t\in\mathcal{T}_p} x_{tp}.$$

Hence, the parallel execution time is

$$\text{CompTime} = \max_p \{X_p\}. \quad (1)$$

Suppose that the file f is stored in the repository r, i.e., $\text{store}(f) = r$. Recall that Λ_f denotes the set of processors to which file f is to be uploaded. The time spent by the repository r on transferring the file f is

$$\text{Upload}(f) = w(f)\sum_{p\in\Lambda_f}\frac{1}{b_{rp}}.$$

For each repository r, we define the total upload time U_r as the summation of Upload(f) costs over all files stored in r, i.e.,

$$U_r = \sum_{f \in \mathcal{F}(r)} \text{Upload}(f).$$

Since the files can be transfered in parallel, with an optimistic view, the maximum upload time spent by a single repository is

$$\text{UploadTime} = \max_r \{U_r\}. \quad (2)$$

The time spent by the processor p on downloading the file f is

$$\text{Download}(f, p) = \frac{w(f)}{b_{\text{store}(f),p}}.$$

Recall that $\text{files}(\mathcal{T}_p) = \bigcup_{t \in \mathcal{T}_p} \text{files}(t)$ is the set of files to be transfered to processor p. For each processor p, we define the total download time D_p as the summation of Download(f, p) costs over all files that are needed by the tasks assigned to the processor p, i.e.,

$$D_p = \sum_{f \in \text{files}(\mathcal{T}_p)} \text{Download}(f, p).$$

Since the files can be downloaded in parallel, with an optimistic view, the maximum download time spent by a single processor is

$$\text{DownloadTime} = \max_p \{D_p\}. \quad (3)$$

Although the three cost components given in Eqs. (1)–(3) do not represent the turnaround time, they are closely related to it. By using these components, we can define lower and upper bounds on the turnaround time. First, observe that the turnaround time cannot be less than any of these components. Therefore, a lower bound on the turnaround time is

$$\text{LBTime} = \max\{\text{CompTime}, \text{UploadTime}, \text{DownloadTime}\}. \quad (4)$$

Furthermore, these components can be used to define an upper bound. A trivial upper bound is

$$\text{UBTime} = \sum_{f \in \mathcal{F}} \text{Upload}(f) + \text{CompTime}.$$

This bound is too pessimistic to be useful; it states that task executions start after all files have been transfered to the processors, where there are no concurrent file transfers. We think that it is hard to define a tighter upper bound that is smooth over the search space generated by task reassignments. Therefore, we define an objective function which is estimated to be an upper bound. By assuming concurrent transfers, we obtain

$$\text{EstUBTime} = \max\{\text{UploadTime}, \text{DownloadTime}\} + \text{CompTime}, \quad (5)$$

which is likely to be an upper bound on the turnaround time. Note that this is an estimation, since it is not guaranteed to be an upper bound. This objective function is a combination of the aforementioned optimistic and pessimistic views. It expects full parallelism among the file transfers and no overlap among the task executions and file transfers.

4.2.2. Improving the bounds

We propose an iterative-improvement heuristic to improve the LBTime and EstUBTime objective functions. The heuristic is based on Fidducia–Mattheyses (FM) [18] refinement heuristic used for graph and hypergraph partitioning [3,16,24,25,27,30,37]. The FM algorithm, starting from an initial partition, performs a number of passes until it finds a locally optimal partition, where each pass contains a sequence of vertex moves. The fundamental idea is the notion of *gain*, which is the decrease in the cost of a partition achieved by moving a vertex to another part. The FM algorithm was initially proposed for refining 2-way graph/hypergraph partitions [18] and then extended for refining multiway graph/hypergraph partitions [36].

In general, vertex movements are performed starting from the one with the maximum gain. Meanwhile, moves with negative gain values are performed tentatively to enable limited hill climbing. However, maintaining the necessary data structures for the multiway refinement is a time consuming operation [36,37]. Therefore, the FM passes for the multiway refinement are usually performed as follows. The vertices are visited according to a random order, and the best moves of the vertices with positive gain are performed [25]. Here, the best move of a vertex is defined as the one with the maximum gain.

Recall that we have two different objective functions, LBTime and EstUBTime. As in our previous work [26], our aim is to close the gap between these two bounds while minimizing both of them. For this purpose, we use an alternating refinement scheme in which first LBTime and then EstUBTime are improved repeatedly until there exists no improvement in these two bounds.

Since we have two bounds to improve, a task reassignment which improves one of these functions may worsen the other one. To solve this problem, we use the two-level gain approach proposed in [26] which modifies the gain concept as follows. The two-level gain scheme determines the best reassignment associated with a task by considering one of the bounds as the primary objective while considering the other bound as the secondary one. For each task, among the reassignments with positive gains in the primary objective, we choose the one with the maximum gain in the secondary objective. We adopt this modification in improving the LBTime as the primary objective. Since EstUBTime is only an estimation, we refine it without the two-level gain approach. This latter scheme gives more freedom in EstUBTime refinement and provides the future LBTime refinements with a larger search space to explore.

The objective functions LBTime and EstUBTime depend highly on the communication cost incurred by the file transfers. If a file f is required to be transfered to a processor p for only one task, reassigning that task from p to another processor will save the cost of transfering f to p. We call such files as *critical* to the processor p and maintain a list of such critical file and processor pairs. The critical file concept corresponds to the critical net concept in hypergraph partitioning.

LBTime-Refinement(Π)
1: $\langle C_1, C_2, C_3\rangle \leftarrow$ DefBounds{UploadTime(Π), DownloadTime(Π), CompTime(Π)}
2: **while** $C_1 \geq C_2$ **and** $C_1 \geq C_3$ **do**
3: Create a random visit order of the tasks associated with C_1
4: **for** each task t in this random order **do**
5: $\langle gain, q\rangle \leftarrow$**LB-ComputeGain**$(t, C_1, C_2)$
6: **if** $gain > 0$ **then**
7: **UpdateGlobalData**(t, q)
8: $Assign(t) \leftarrow q$
9: **if** $C_1 < C_2$ **or** $C_1 < C_3$ **then**
10: **return**
11: **if** bottleneck repository or processor is changed **then**
12: **goto** 2

Fig. 3. Algorithm for improving LBTime objective.

LBTime-refinement: Fig. 3 displays the LBTime-refinement heuristic. The heuristic first finds the values of the variables C_1, C_2, and C_3 that are used to refer to the three cost components. The variable C_1 refers to the maximum of UploadTime, DownloadTime, and CompTime, i.e., LBTime $= C_1$. The variable C_2 refers to the cost component which in conjunction with C_1 defines EstUBTime $= C_1 + C_2$, e.g., if C_1 is CompTime, C_2 will be the maximum of UploadTime and DownloadTime, otherwise it will be CompTime (see Eq. (5)). Effectively, C_1 becomes the primary objective, and $C_1 + C_2$ becomes the secondary one. The heuristics run until the cost component that defines LBTime changes. If the largest cost component C_1 is the UploadTime, then a randomly permuted list of tasks that request files from the bottleneck repository is constructed. Otherwise, a randomly permuted list of tasks that are assigned to the bottleneck processor is constructed. For the sake of run time efficiency, the visit orders are constructed using only the tasks that are associated with the bottleneck repositories and processors.

The procedure LB−ComputeGain(t, C_1, C_2) computes the reassignment gains associated with task t and returns the reassignment with positive gain in the primary objective C_1 and the maximum gain in the secondary objective $C_1 + C_2$. If such a reassignment is found, the task is reassigned from its current owner $p = Assign(t)$ to a new processor q.

The gain computations for the cost components are performed as follows. Let $X(2)$ denote the execution time of the processor with the second maximum task execution time. Then, the gain of reassigning the task t from a bottleneck processor p to processor q is

$$g_{\text{comp}}(t, p, q) = \min \left\{ \begin{array}{c} x_{tp} \\ X_p - X(2) \\ X_p - (X_q + x_{tq}) \end{array} \right\}, \tag{6}$$

according to the objective CompTime. The first argument out of the three, x_{tp}, corresponds to the case in which the processor p remains to be the bottleneck processor after the reassignment. The second argument $X_p - X(2)$ corresponds to the case in which $X_q < X(2)$ and the second bottleneck processor before the reassignment becomes the bottleneck processor afterwards. The third argument $X_p - (X_q + x_{tq})$ handles the cases in which processor q becomes the bottleneck processor after the reassignment.

Let $D(2)$ denote the download cost on the processor with the second maximum file download time. Then, the gain of reassigning task t from a bottleneck processor p to processor q is

$$g_{\text{download}}(t, p, q) = \min \left\{ \begin{array}{c} \displaystyle\sum_{f \in \text{critical}(\text{files}(t), p)} \frac{w(f)}{b_{\text{store}(f), p}} \\ D_p - D(2) \\ \displaystyle D_p - \left(D_q + \sum_{f \in \text{notNeeded}(\text{files}(t), q)} \frac{w(f)}{b_{\text{store}(f), q}} \right) \end{array} \right\}, \tag{7}$$

according to the objective DownloadTime. The first argument corresponds to the case in which the processor p remains to be the bottleneck processor after the reassignment. In this argument, the set critical(files(t), p) contains the files that are needed by task t and are critical to the processor p before the reassignment. The second argument $D_p - D(2)$ corresponds to the case in which $D_q < D(2)$ and the second bottleneck processor before the reassignment becomes the bottleneck processor afterwards. The third argument handles the cases in which processor q becomes the bottleneck processor after the reassignment. In this argument, the set notNeeded(files(t), q) contains those files of task t that are not needed by any task in $\mathcal{T}_q$ before the reassignment. Note that the set of files notNeeded(files(t), q) become critical to processor q after the reassignment.

Let $U(1)$ denote the upload cost of the repository with the maximum file upload time. Then, the gain of reassigning task t from the processor p to processor q is

$$g_{\text{upload}}(t, p, q) = U(1) - \max_r \left\{ \begin{array}{l} U_r - \displaystyle\sum_{f \in \text{critical}(\text{files}(t) \cap \mathcal{F}(r), p)} \frac{w(f)}{b_{rp}} \\ \quad + \displaystyle\sum_{f \in \text{notNeeded}(\text{files}(t) \cap \mathcal{F}(r), q)} \frac{w(f)}{b_{rq}} \end{array} \right\}, \tag{8}$$

according to the objective UploadTime. Here, $U(1)$ gives the bottleneck value before the reassignment. The $\max_r\{\cdot\}$ corre-

```
EstUBTime-Refinement(Π)
 1: ⟨C1, C2, C3⟩ ← DefBounds{UploadTime(Π), DownloadTime(Π), CompTime(Π)}
 2: while C1 ≥ C3 and C2 ≥ C3 do
 3:   Create a random visit order of the tasks
 4:   for each task t in this random order do
 5:     ⟨gain, q⟩ ←EstUB-ComputeGain(t, C1, C2)
 6:     if gain > 0 then
 7:       UpdateGlobalData(t, q)
 8:       Assign(t) ← q
 9:       if C1 < C3 or C2 < C3 then
10:         return
11:       if bottleneck repository or processor is changed then
12:         goto 2
```

Fig. 4. Algorithm for improving EstUBTime objective.

sponds to the bottleneck value upon realizing the reassignment. The set files$(t) \cap \mathcal{F}(r)$ contains those files that are needed by task t and are stored in repository r. Reassigning task t changes the upload times of the repositories in which files(t) are stored. The first summation corresponds to the decrease in the upload time of the repository r due to relieving r of transferring the critical files of t to processor p. The second summation corresponds to the increase in the upload time of the repository r due to the files in the set notNeeded(files$(t), q)$.

The procedure UpdateGlobalData(t, q) computes the new loads of the repositories and the processors, and it keeps track of the changes in the cost components that define LBTime and EstUBTime. It also maintains the identities of the repositories and the processors that attain the maximum and the second maximum load in terms of the three cost components.

EstUBTime-refinement: The EstUBTime-refinement heuristic (see Fig. 4) is similar to the LBTime-refinement heuristic with a few differences. This procedure visits all tasks in a random order and computes the reassignment gains of the tasks as the total gain obtained for the cost components that define EstUBTime. For example, $g_{\text{comp}}(t, p, q) + g_{\text{download}}(t, p, q)$ is computed as the gain of reassigning the task t from processor p to processor q, if EstUBTime is defined by CompTime and DownloadTime. Here, $g_{\text{comp}}(t, p, q)$ and $g_{\text{download}}(t, p, q)$ are computed according to Eqs. (6) and (7), respectively. The best reassignment of a task is realized if the total gain is nonnegative. The procedure adapts itself to the cost components that define the EstUBTime and discards the tasks that are not associated with the bottleneck cost components. Observe that the CompTime is always one of the bottleneck cost components and the other may change because of the tasks reassignments throughout the execution of the EstUBTime-Refinement procedure.

Time complexity: A refinement pass consists of an LBTime-refinement and an EstUBTime-refinement. We apply three refinement passes. We run the LBTime-refinement and EstUBTime-refinement heuristics with at most five iterations of the while loops (see line 2 of Figs. 3 and 4). Therefore, the time complexity of the refinement pass is linearly proportional to the time complexities of LBTime-refinement and EstUBTime-refinement (although with a relatively high constant). Both in LBTime-refinement and EstUBTime-refinement, computing the best gain for a task t takes $O(P|\text{files}(t)|)$ time, and updating global data takes $O(|\text{files}(t)| + P + R)$ time. Therefore, the overall time complexity of an LBTime-refinement or EstUBTime-refinement pass is

$$\sum_{t \in \mathcal{T}} O(P|\text{files}(t)| + |\text{files}(t)| + P + R) = O(P|\mathcal{A}| + T(P + R)).$$

4.3. Phase 3: determining task execution orderings

In this phase, we generate a complete solution for the scheduling problem. All of the five task-to-processor assignments obtained in the second phase are kept intact while determining the schedules of the file transfers and the order of task executions on each processor. Note that CompTime, UploadTime, and DownloadTime and hence the values of the LBTime and EstUBTime computed in the second phase remain as is (for each task-to-processor assignment).

We again utilize the structure of the heuristics proposed by Giersch et al. [20,23]. We have modified those heuristics in order to keep the task-to-processor assignments fixed while determining the order of the task executions and file transfers. Fig. 5 displays the execution ordering heuristic for a given task-to-processor assignment Π. The structure of the execution ordering heuristic is similar to the scheduling heuristics proposed in [20,23,26].

For each processor, we build a list of tasks sorted according to the objective values OBJECTIVE(t, p) computed using the five functions proposed in [20,23] and discussed in Section 2.1. Then, we extract the first task in the list of the processor which is determined according to the processor selection rule. We then schedule the file transfers associated with the current task using the heuristic discussed in Section 4.1 and determine the task's earliest start-up time. When the transfer of file f is scheduled, the earliest start-up time of the tasks that need f are updated. When all file transfers are scheduled and hence the earliest start-up times of the tasks are determined, we schedule the executions of the tasks according to the earliest start-up times on each processor. Note that using five different objective functions for each of the five task-to-processor assignments

ExecutionOrdering(Π)
1: **for** each processor p **do**
2: **for** each task t assigned to p in Π **do**
3: Evaluate OBJECTIVE(t, p)
4: Build the list $L(p)$ of the tasks that are assigned to processor p sorted according to the value of OBJECTIVE(t, p)
5: **while** there remains a file transfer to schedule **do**
6: Select the processor p with respect to processor selection rule
7: Let t be the first task in $L(p)$
8: **for** each yet to be scheduled file f in files(t) **do**
9: schedule f using insertion scheduling
10: **for** each task in $L(p)$ that depends on file f **do**
11: update earliest start-up time
12: Mark f as transfered to p
13: **for** each processor p **do**
14: Schedule the tasks assigned to p according to the increasing order of the earliest start-up times

Fig. 5. Algorithm for scheduling file transfers and determining execution orders.

found in the second phase enables us to return the best out of 25 complete schedules as a solution.

We propose two rules for selecting a processor in line 6 of Fig. 5 to schedule a file transfer. The first rule is to select the processor which has the largest execution time defined in terms of the tasks that have files to be scheduled. The second one is to select the processor which requires the largest file download time defined in terms of those yet to be scheduled. To maximize the probability of overlap between file transfers and task executions, it is better to produce a schedule with a small number of idle processors throughout the execution of the application. Therefore, applying the first rule is more promising, because it tries to balance the remaining loads of the processors. The second rule greedily delivers files to the processor which has the largest file download time. To apply these selection rules, we maintain two variables for each processor which initially hold the total file download time and the total execution time of the respective processor. For each scheduled file, we update these two variables appropriately to apply the processor selection rule in line 6 of Fig. 5.

Time complexity: The first four initialization steps take $O(T + |\mathcal{A}| + T \log T)$ time. The while loop runs for T iterations. At each iteration, the files of a task are scheduled. As in Section 4.1, scheduling the files of a task t takes $\sum_{f \in \text{files}(t)} O\left(\frac{FP}{R}\right)$ time. The for loop in line 10 of Fig. 5 appends an additive term which sums up to $O(P|\mathcal{A}|)$ in the overall execution. Therefore, the complexity of the execution ordering phase is

$$\sum_{t=1}^{T} \sum_{f \in \text{files}(t)} O\left(\frac{FP}{R}\right) = O\left(\frac{FP}{R}|\mathcal{A}|\right).$$

5. Simulations

We generate synthetic applications and computing platforms to create a large number of scheduling problems. We have been careful in generating the instances to be representative of real life applications and platforms. Below, we first describe how we went about creating the instances. Then, we investigate the performance of the proposed heuristics on these instances by computing the turnaround time by means of simulations.

5.1. Generating application instances

We generated applications with $T = 3000$ tasks and $F = 3000$ files. The file sizes were random integers ranging from 50 Mbytes to 70 Gbytes. Following our previous work [26], we generated the task execution times as follows. We randomly chose mid-rank supercomputers from the Top500 list [17]. As the Top500 list uses the LINPACK benchmark, we assume that the individual tasks are instances of the same problem consuming approximately $(2/3)n^3$ floating point operations for an instance of size n. The benchmark values $Flops_{\max}$, $n_{\max}$ and $n_{1/2}$ of the selected supercomputers were used to generate realistic task execution times. Here, $Flops_{\max}$ of a processor denotes the maximum processor performance, in terms of FLOPS, that can be achieved for a task of size greater than or equal to $n_{\max}$, and $n_{1/2}$ represents the problem size for which half of the $Flops_{\max}$ is achieved. Therefore, the performance variation of a processor p for a task of size n can be approximated with a piecewise linear function $Flops_p(n)$ as shown in Fig. 6. Thus, the estimated execution time of a task t with instance size n on a processor p can be computed as $x_{tp} = (2/3)n^3/Flops_p(n)$. For each task, the size n was randomly chosen from the range [10 000–35 000].

We generated three groups of problem instances in which the number of files requested by a task was in the range [1–5] or [1–10] or [1–20]. We use f1–5, f1–10, and f1–20 to denote these three choices. These choices enable us to generate problem instances in which the levels of file sharing among the tasks are low, medium, and high. Besides, large number of files requested by a task increases the probability of a task requesting files from a number of repositories. This in turn results in harder problem instances. Initial distribution of files to repositories is

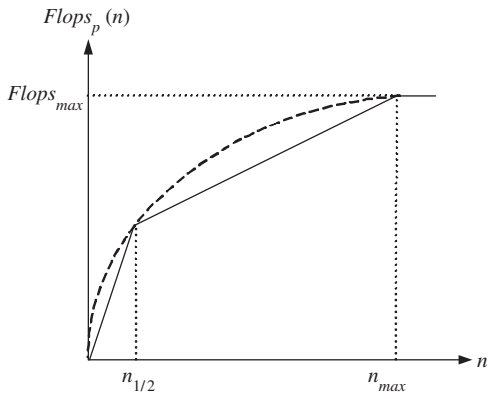

Fig. 6. The piecewise linear function to estimate task execution times on a processor p [26].

built randomly; repositories store almost equal number of files regardless of the stored files' sizes.

Communication-to-computation ratio ρ of an application is defined as the ratio of the average communication cost to the average computation cost, i.e., $\rho = c_{\text{avg}}/x_{\text{avg}}$, where

$$c_{\text{avg}} = (1/R)\sum_{t=1}^{T}\sum_{r=1}^{R}(1/b_r)w\,(\text{files}(t)\cap\mathcal{F}(r))\,,$$

$$x_{\text{avg}} = (1/P)\sum_{t=1}^{T}\sum_{p=1}^{P}x_{tp}.$$

Here, b_r represents the average bandwidth from the repository r to the processors, i.e., $b_r = \frac{1}{P}\sum_p b_{rp}$. We scaled file sizes to obtain five different ratios $\rho = 0.1$, 0.2, 1.0, 5.0, and 10.0. These choices characterize a range of applications starting from computation intensive ($\rho = 0.1$) to communication intensive ($\rho = 10.0$) ones.

5.2. Generating computing system

We used GridG network topology generator [31] for creating a heterogeneous system with $P = 32$ processors and up to nine data repositories. Although GridG generates large and realistic network topologies that follow the power laws of Internet topology, we used only a portion of the GridG's output to design the computing systems. For a given number of repositories R, we randomly chose R of the nine routers as the repositories. Then, for each repository–processor pair, we assumed a direct connection with a bandwidth equal to the lowest bandwidth of the path whose slowest link has the highest bandwidth among the alternatives. The bandwidths of the communication links between repositories and processors were in between 10 Mbit/s and 1 Gbit/s.

5.3. Results

Table 1 summarizes the results of the experiments conducted to validate the relation between the objective functions proposed for refining task assignments and the turnaround time. The values in the table were derived by using the scheduling heuristics individually in the initial task assignment phase as follows: for each heuristic used, the amount of decrease achieved in both LBTime and EstUBTime during the refinement phase were normalized with respect to the amount of the resulting decrease in the turnaround time. That is, these values display the amount of improvements needed in LBTime and EstUBTime to attain one unit of improvement in the turnaround time. As seen in Table 1, close to one unit (between 0.97 and 1.29) of improvements are needed in LBTime, whereas the required improvement in EstUBTime is in between 1.28 and 4.42. This shows that the EstUBTime is not a tight upper bound on the turnaround time in the problem instances that we consider. We think that it is hard to come up with a tighter upper bound without considering the order of file transfers and task executions. However, different and tighter estimations would increase the efficiency of the proposed heuristic in terms of the turnaround time.

Recall from Sections 4.1 and 4.3 that we use the five objective functions of Giersch et al. [20,23] in the initial task assignment and execution phases. That is, we return the best out of 25 different schedules. We observed that the returned schedules are mostly constructed by using the objective functions *computation*, *duration*, and *communication* in the initial assignment phase. For $\rho = 10.0$, almost half of the returned solutions are constructed by using *duration*, and for $\rho = 0.1$ almost one-third of the returned solutions are constructed by using *computation*. Additionally, we found the *payoff* and *communication* functions to be superior to the others in the execution ordering phase. For $\rho = 0.1$, the number of solutions obtained by using *communication* in the execution ordering phase is larger than those obtained by using the other functions. For the other ρ values, the number of solutions obtained by using *payoff* in the execution ordering phase is larger than those obtained by using the other functions. In the light of above observations, we recommend the use of *duration* objective function in the initial assignment phase, and the use of *payoff* objective function in the execution ordering phase when ρ is large. For small values of ρ, we recommend the use of *computation* objective function in the initial assignment phase, and the use of *communication* objective function in the execution ordering phase.

Recall from Section 4.3 that we have two rules to select a processor to schedule a file transfer in the execution ordering phase of the proposed heuristic. The first rule—selecting the processor which has the largest execution cost defined in terms of the tasks that have file transfers to be scheduled—led to better schedules than the second rule for all ρ values in almost all instances.

5.3.1. Performance

We implemented extensions of the MinMin, MaxMin, and Sufferage heuristics for the scheduling problem at hand. The proposed iterative-improvement-based scheduling heuristic for the distributed repositories is referred to here as MIIS. Tables 2–4 show the results of the experiments conducted to compare the performance of MIIS with those of MinMin, MaxMin, and Sufferage. For each scheduling instance consisting of a ρ and R

Table 1
The amount of improvements in LBTime and EstUBTime objective values required to obtain one unit of improvement in the turnaround time, i.e., Δ(LBTime)/Δ(TurnaroundTime) and Δ(EstUBTime)/Δ(TurnaroundTime), respectively. Here, Δ(Obj) is the difference between Obj values after the first and the third phases of the proposed heuristic

Heuristic used in Phase 1	LBTime			EstUBTime		
	f1–5	f1–10	f1–20	f1–5	f1–10	f1–20
Duration	0.989	0.997	1.273	1.807	1.699	2.994
Payoff	1.031	1.059	1.160	1.949	2.061	3.443
Advance	1.036	1.162	1.287	1.951	2.238	4.423
Communication	1.004	1.018	1.191	2.014	1.880	3.151
Computation	0.992	0.973	1.051	1.673	1.275	2.097

Table 2
Averages of the relative performances of the heuristics normalized with respect to the best solutions for the scheduling instance with tasks requesting 1–5 files

ρ	R	MinMin	MaxMin	Sufferage	MIIS
0.1	1	1.216	1.341	1.103	1.004
	2	1.141	1.454	1.027	1.011
	3	1.114	1.548	1.005	1.016
	4	1.097	1.605	1.041	1.000
	5	1.094	1.674	1.036	1.002
	6	1.085	1.735	1.051	1.001
	7	1.079	1.789	1.056	1.001
	8	1.070	1.758	1.075	1.000
0.2	1	1.383	1.558	1.224	1.000
	2	1.346	1.788	1.241	1.000
	3	1.262	1.838	1.148	1.000
	4	1.217	1.924	1.172	1.000
	5	1.177	2.028	1.150	1.000
	6	1.153	2.083	1.153	1.000
	7	1.130	2.087	1.133	1.000
	8	1.117	2.079	1.127	1.000
1.0	1	1.147	1.487	1.122	1.000
	2	1.162	1.616	1.094	1.000
	3	1.109	1.584	1.045	1.000
	4	1.117	1.662	1.049	1.003
	5	1.099	1.636	1.043	1.000
	6	1.092	1.678	1.045	1.000
	7	1.095	1.668	1.052	1.000
	8	1.078	1.636	1.088	1.000
5.0	1	1.129	1.581	1.161	1.000
	2	1.173	1.518	1.021	1.026
	3	1.125	1.455	1.015	1.022
	4	1.097	1.488	1.030	1.005
	5	1.088	1.441	1.067	1.003
	6	1.077	1.452	1.075	1.000
	7	1.083	1.456	1.112	1.000
	8	1.090	1.491	1.136	1.000
10.0	1	1.153	1.560	1.185	1.010
	2	1.136	1.419	1.000	1.047
	3	1.120	1.393	1.026	1.009
	4	1.093	1.432	1.057	1.001
	5	1.084	1.428	1.070	1.000
	6	1.090	1.505	1.108	1.000
	7	1.080	1.451	1.093	1.000
	8	1.091	1.431	1.153	1.000

Table 3
Averages of the relative performances of the heuristics normalized with respect to the best solutions for the scheduling instance with tasks requesting 1–10 files

ρ	R	MinMin	MaxMin	Sufferage	MIIS
0.1	1	1.200	1.300	1.055	1.018
	2	1.160	1.324	1.025	1.006
	3	1.115	1.405	1.007	1.022
	4	1.091	1.441	1.011	1.007
	5	1.079	1.449	1.037	1.002
	6	1.089	1.520	1.037	1.000
	7	1.090	1.537	1.064	1.000
	8	1.077	1.536	1.067	1.000
0.2	1	1.258	1.400	1.208	1.000
	2	1.271	1.516	1.141	1.000
	3	1.247	1.544	1.166	1.000
	4	1.234	1.624	1.162	1.000
	5	1.188	1.635	1.164	1.000
	6	1.175	1.680	1.157	1.000
	7	1.149	1.652	1.136	1.000
	8	1.133	1.652	1.146	1.000
1.0	1	1.144	1.416	1.135	1.000
	2	1.162	1.505	1.153	1.000
	3	1.181	1.573	1.149	1.000
	4	1.138	1.494	1.092	1.000
	5	1.141	1.467	1.122	1.000
	6	1.123	1.421	1.092	1.000
	7	1.104	1.393	1.067	1.000
	8	1.086	1.378	1.128	1.000
5.0	1	1.216	1.528	1.224	1.000
	2	1.132	1.521	1.076	1.011
	3	1.169	1.440	1.083	1.000
	4	1.177	1.400	1.087	1.003
	5	1.134	1.342	1.187	1.000
	6	1.120	1.351	1.216	1.000
	7	1.102	1.305	1.213	1.000
	8	1.108	1.296	1.370	1.000
10.0	1	1.187	1.490	1.219	1.000
	2	1.091	1.401	1.049	1.001
	3	1.188	1.499	1.131	1.000
	4	1.155	1.382	1.136	1.000
	5	1.141	1.352	1.100	1.001
	6	1.121	1.341	1.260	1.000
	7	1.116	1.312	1.301	1.000
	8	1.102	1.285	1.271	1.000

pair, we generated 10 different problems. We then computed the relative scheduling performance of every heuristic by dividing the obtained turnaround time by the best turnaround time found for the same problem instance. The averages of these 10 relative scheduling performances are given in Tables 2–4.

As seen in these tables, the proposed MIIS heuristic obtains better results than the MinMin, MaxMin, and Sufferage heuristics in almost all problem instances. On the average, MIIS performs 10% better than Sufferage—the best of the existing heuristics—in terms of turnaround time. We observed that the MaxMin heuristic is consistently worse than the MinMin and Sufferage heuristics. This is in concordance with the previously reported results [20,26]. The relative performance of the MinMin heuristic improves by the increasing number of data repositories. The relative performance of the Sufferage heuristic gets better at the mid-range values of the number of data repositories. In particular, it achieves its best performance at $R = 3$ or 4. Our experiments show that when the tasks request 1–10 files, the MIIS heuristic performs much better than the others, e.g., 14% better than the MinMin and Sufferage heuristics.

5.3.2. Running time

The proposed and existing heuristics were implemented in C. Experiments were performed on a PC equipped with a 2.4 GHz Intel Pentium-IV processor and 2 Gbytes RAM running Linux kernel 2.4.20. The Sufferage heuristic runs for 77 min to find

Table 4
Averages of the relative performances of the heuristics normalized with respect to the best solutions for the scheduling instance with tasks requesting 1–20 files

ρ	R	MinMin	MaxMin	Sufferage	MIIS
0.1	1	1.167	1.279	1.067	1.024
	2	1.117	1.258	1.011	1.017
	3	1.104	1.280	1.013	1.010
	4	1.083	1.285	1.010	1.004
	5	1.073	1.296	1.015	1.004
	6	1.078	1.322	1.044	1.000
	7	1.077	1.333	1.042	1.000
	8	1.070	1.312	1.054	1.000
0.2	1	1.191	1.241	1.122	1.000
	2	1.170	1.280	1.074	1.000
	3	1.167	1.297	1.067	1.000
	4	1.170	1.359	1.105	1.000
	5	1.186	1.391	1.125	1.000
	6	1.204	1.458	1.192	1.000
	7	1.202	1.450	1.191	1.000
	8	1.181	1.402	1.166	1.000
1.0	1	1.100	1.292	1.172	1.000
	2	1.144	1.353	1.175	1.000
	3	1.198	1.425	1.145	1.000
	4	1.176	1.412	1.135	1.000
	5	1.141	1.349	1.118	1.000
	6	1.074	1.261	1.042	1.012
	7	1.063	1.245	1.051	1.001
	8	1.051	1.224	1.059	1.001
5.0	1	1.151	1.400	1.219	1.000
	2	1.089	1.448	1.061	1.028
	3	1.093	1.348	1.044	1.037
	4	1.105	1.277	1.045	1.063
	5	1.097	1.251	1.052	1.006
	6	1.085	1.209	1.064	1.009
	7	1.080	1.185	1.133	1.000
	8	1.084	1.199	1.162	1.000
10.0	1	1.155	1.420	1.182	1.000
	2	1.051	1.398	1.054	1.010
	3	1.098	1.312	1.018	1.027
	4	1.104	1.218	1.009	1.082
	5	1.106	1.220	1.045	1.007
	6	1.085	1.196	1.107	1.000
	7	1.090	1.181	1.261	1.000
	8	1.092	1.193	1.263	1.000

a solution to the problem instance with $T = 3000$ tasks, $F = 3000$ files, f1–10, $P = 32$ processors, $R = 4$ repositories, and $\rho = 1.0$.

Table 5 shows the running times of the MIIS heuristic for the problem instances with 1, 4, and 8 repositories and 32 processors. As seen from the table, the running time of the proposed MIIS heuristic is much less than that of the Sufferage heuristic, e.g., almost 90 times faster than Sufferage for the problem instance given above. The running time of MIIS is expected to increase with the increasing number of repositories. This expectation, however, does not hold for some of the problem instances. This may stem from the running time differences in the refinement heuristics. The running time of the proposed MIIS heuristic increases with the increasing number of files-per-task, since the running time complexities of the heuristics proposed for the three phases of MIIS contain the term $|\mathcal{A}|$.

6. Conclusion

We have proposed a three phase heuristic for scheduling file-sharing tasks on a computing system consisting of heterogeneous processors and data repositories connected through a heterogeneous interconnection network. The proposed heuristic is built upon our previous work on scheduling file-sharing tasks on heterogeneous master–slave systems [26]. We have clearly stated the problems that arise because of the existence of dis-

Table 5
The running times of MIIS (in s)

ρ	Num. files per task	Number of repositories		
		$R = 1$	$R = 4$	$R = 8$
0.1	f1–5	14.61	27.94	24.14
	f1–10	19.75	56.41	52.15
	f1–20	28.29	134.99	141.69
0.2	f1–5	14.32	25.34	23.35
	f1–10	20.06	55.07	51.34
	f1–20	28.34	135.91	130.27
1.0	f1–5	14.84	30.39	23.62
	f1–10	16.75	52.06	50.35
	f1–20	22.34	111.84	129.62
5.0	f1–5	11.96	28.07	23.23
	f1–10	10.24	50.29	47.84
	f1–20	14.36	99.81	128.56
10.0	f1–5	11.13	28.62	22.78
	f1–10	10.55	50.23	48.38
	f1–20	12.95	100.25	127.58

tributed repositories. We have also implemented the MinMin, MaxMin, and Sufferage heuristics [14,15] in order to assess the performance of the proposed scheduling heuristic. We have found Sufferage to be the best among these three heuristics. The proposed heuristic has been reported to be considerably faster than the Sufferage heuristic while obtaining 10–14% improvements in the turnaround times.

In the target scheduling problem, inconsistent ETC matrices were used to incorporate heterogeneity both in tasks and processor capacities. However, there exist some problem variants in which the linear cost model is used to incorporate heterogeneity only in processor capacities. In the linear cost model, a computing resource p is associated with a weight s_p which represents the number of time-steps required to process one unit of computation. Additionally, the costs of the tasks are specified in units of computation. In other words, processing a task of size L on the computing resource p requires $L \cdot s_p$ time units. We believe that clustering methods and hence multilevel refinement heuristics will be viable for both reducing the run-time complexity and improving the solution quality under the linear cost model for the computations.

Acknowledgment

Part of the experimental tests were carried out at the TUBITAK ULAKBIM High Performance Computing Center.

References

[1] S. Ali, H.J. Siegel, M. Maheswaran, D. Hensgen, S. Ali, Task execution time modeling for heterogeneous computing systems, in: C. Raghavendra (Ed.), Proceedings of the Nineth Heterogeneous Computing Workshop (HCW 2000), Cancun, Mexico, IEEE, May 2000, pp. 185–199.

[2] C.J. Alpert, A.B. Kahng, Recent directions in netlist partitioning: a survey, Integration, The VLSI J. 19 (1–2) (August 1995) 1–81.

[3] C. Aykanat, A. Pınar, Ü.V. Çatalyürek, Permuting sparse rectangular matrices into block-diagonal form, SIAM J. Sci. Comput. 25 (6) (2004) 1860–1879.

[4] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, Y. Robert, Scheduling strategies for master–slave tasking on heterogeneous processor platforms, IEEE Trans. Parallel and Distributed Systems 15 (4) (2004) 319–330.

[5] O. Beaumont, V. Boudet, Y. Robert, A realistic model and an efficient heuristic for scheduling with heterogeneous processors, Technical Report RR-2001-37, LIP, ENS Lyon, France, September 2001.

[6] O. Beaumont, A. Legrand, L. Marchal, Y. Robert, Steady-state scheduling on heterogeneous clusters, Internat. J. Foundations Comput. Sci. 16 (2) (2005) 163–194.

[7] O. Beaumont, L. Marchal, Y. Robert, Broadcast trees for heterogeneous platforms, Technical Report RR-2004-46, LIP, ENS Lyon, France, November 2004.

[8] C. Berge, Hypergraphs, North-Holland, Amsterdam, 1989.

[9] F. Berman, High-performance schedulers, in: I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, Los Altos, CA, 1999, pp. 279–309, (Chapter 12).

[10] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S.M. Figueira, J. Hayes, G. Obertelli, J.M. Schopf, G. Shao, S. Smallen, N.T. Spring, A. Su, D. Zagorodnov, Adaptive computing on the Grid using AppLeS, IEEE Trans. Parallel Distrib. Systems 14 (4) (2003) 369–382.

[11] C. Boeres, A. Lima, V.E.F. Rebello, Hybrid task scheduling: Integrating static and dynamic heuristics, in: Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing, IEEE, November 2003, pp. 199–206.

[12] R. Buyya, D. Abramson, J. Giddy, H. Stockinger, Economic models for resource management and scheduling in Grid computing, Concurrency and Comput.: Pract. Experience 14 (13–15) (2002) 1507–1542.

[13] H. Casanova, Network modeling issues for Grid application scheduling, Internat. J. Found. Comput. Sci. 16 (2) (2005) 145–162.

[14] H. Casanova, A. Legrand, D. Zagorodnov, F. Berman, Heuristics for parameter sweep applications in Grid environments, in: Proceedings of Nineth Heterogeneous Computing Workshop, IEEE Computer Society Press, Silver Spring, MD, 2000, pp. 349–363.

[15] H. Casanova, G. Obertelli, F. Berman, R. Wolski, The AppLeS parameter sweep template: user-level middleware for the Grid, in: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing CDROM, IEEE Computer Society, Silver Spring, MD, 2000, p. 60.

[16] Ü.V. Çatalyürek, C. Aykanat, Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication, IEEE Trans. Parallel Distrib. Systems 10 (7) (1999) 673–693.
[17] J.J. Dongarra, H.W. Meuer, E. Strohmaier, TOP500 Supercomputer Sites, 22nd edition, in: Proceedings of the Supercomputing Conference (SC2003), Phoenix, Arizona, USA, 2003.
[18] C.M. Fidducia, R.M. Mattheyses, A linear-time heuristic for improving network partitions, in: 19th ACM/IEEE Design Automation Conference, 1982, pp. 175–181.
[19] B. Folliot, P. Sens, Load sharing and fault tolerance manager, in: R. Buyya (Ed.), High Performance Cluster Computing, vol. 1. Architectures and Systems, Prentice-Hall, Englewood Cliffs, NJ, 1999, pp. 534–552, (Chapter 22).
[20] A. Giersch, Y. Robert, F. Vivien, Scheduling tasks sharing files on heterogeneous clusters. Technical Report RR-2003-28, LIP, ENS Lyon, France, May 2003.
[21] A. Giersch, Y. Robert, F. Vivien, Scheduling tasks sharing files from distributed repositories. Technical Report RR-2004-04, LIP, ENS Lyon, France, February 2004.
[22] A. Giersch, Y. Robert, F. Vivien, Scheduling tasks sharing files from distributed depositories, Euro-Par-2004: International Conference on Parallel Processing, Lecture Notes in Computer Science, vol. 3149, Springer, Berlin, 2004, pp. 246–253.
[23] A. Giersch, Y. Robert, F. Vivien, Scheduling tasks sharing files on heterogeneous master–slave platforms, in: PDP'2004, 12th Euromicro Workshop on Parallel Distributed and Network-based Processing, IEEE Computer Society Press, Silver Spring, MD, 2004.
[24] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM), San Diego, California, ACM Press, New York, 1995, p. 28.
[25] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, J. Parallel Distrib. Comput. 48 (1) (1998) 96–129.
[26] K. Kaya, C. Aykanat, Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogeneous master–slave platforms, IEEE Trans. Parallel Distrib. Systems 17 (8) (2006) 883–896.
[27] B.W. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, The Bell System Tech. J. 49 (2) (1970) 291–307.
[28] G. Khanna, N. Vydyanathan, T. Kurc, Ü. V. Çatalyürek, P. Wyckoff, J. Saltz, P. Sadayappan, A hypergraph partitioning based approach for scheduling of tasks with batch-shared I/O. in: Proceedings of Cluster Computing and Grid, 2005.
[29] K. Krauter, R. Buyya, M. Maheswaran, A taxonomy and survey of grid resource management systems for distributed computing, Software: Practice and Experience 32 (2) (2002) 135–164.
[30] T. Lengauer, Combinatorial Algorithms for Integrated Circuit Layout, Wiley-Teubner, Chichester, UK, 1990.
[31] D. Lu, P.A. Dinda, GridG: Generating realistic computational grids, SIGMETRICS Perform. Eval. Rev. 30 (4) (2003) 33–40.
[32] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, R. Freund, Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems, J. Parallel Distrib. Comput. 59 (2) (1999) 107–131.
[33] S. Sahni, Scheduling master–slave multiprocessor systems, IEEE Trans. Comput. 45 (10) (1996) 1195–1199.
[34] S. Sahni, G. Vairaktarakis, The master–slave paradigm in parallel computer and industrial settings, J. Global Optim. 9 (3–4) (1996) 357–377.
[35] T. Saif, M. Parashar, Understanding the behavior and performance of non-blocking communications in MPI, Lecture Notes in Computer Science, vol. 3149, Springer, Berlin, 2004, pp. 173–182.
[36] L.A. Sanchis, Multiple-way network partitioning, IEEE Trans. Comput. 38 (1) (1989) 62–81.
[37] B. Uçar, C. Aykanat, Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix–vector multiplies, SIAM J. Sci. Comput. 25 (6) (2004) 1837–1859.

Kamer Kaya graduated from Bilkent University, Turkey in 2004 with a MSc degree in Computer Engineering where he is currently a PhD candidate. His research deals with cryptography, parallel computing and algorithms.

Bora Uçar received the PhD degrees (2005) in Computer Engineering from Bilkent University, Ankara, Turkey. His research interests are combinatorial scientific computing and high performance computing.

Cevdet Aykanat received the BS and MS degrees from Middle East Technical University, Ankara, Turkey, both in electrical engineering, and the PhD degree from Ohio State University, Columbus, in electrical and computer engineering. He was a Fulbright scholar during his PhD studies. He worked at the Intel Supercomputer Systems Division, Beaverton, Oregon, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Ankara, Turkey, where he is currently a professor. His research interests mainly include parallel computing, parallel scientific computing and its combinatorial aspects, parallel computer graphics applications, parallel data mining, graph and hypergraph-partitioning, load balancing, neural network algorithms, high performance information retrieval systems, parallel and distributed web crawling, parallel and distributed databases, and grid computing. He has (co)authored over 40 technical papers published in academic journals indexed in SCI. He is the recipient of the 1995 Young Investigator Award of The Scientific and Technical Research Council of Turkey. He is a member of the ACM and the IEEE Computer Society. He has been recently appointed as a member of IFIP Working Group 10.3 (Concurrent Systems) and EU-INTAS Council of Scientists.