Iterative-Improvement-Based Heuristics for Adaptive Scheduling of Tasks Sharing Files on Heterogeneous Master-Slave Environments

Kamer Kaya and Cevdet Aykanat, *Member*, *IEEE Computer Society*

Abstract—The scheduling of independent but file-sharing tasks on heterogeneous master-slave platforms has recently found important applications in Grid environments. The scheduling heuristics recently proposed for this problem are all constructive in nature and based on a common greedy criterion which depends on the momentary completion time values of the tasks. We show that this greedy decision criterion has shortcomings in exploiting the file-sharing interaction among tasks since completion time values are inadequate to extract the global view of this interaction. We propose a three-phase scheduling approach which involves initial task assignment, refinement, and execution ordering phases. For the refinement phase, we model the target application as a hypergraph and, with an elegant hypergraph-partitioning-like formulation, we propose using iterative-improvement-based heuristics for refining the task assignments according to two novel objective functions. Unlike the turnaround time, which is the actual schedule cost, the smoothness of proposed objective functions enables the use of iterative-improvement-based heuristics successfully since their effectiveness and efficiency depend on the smoothness of the objective function. Experimental results on a wide range of synthetically generated heterogeneous master-slave frameworks show that the proposed three-phase scheduling approach performs much better than the greedy constructive approach.

Index Terms—Scheduling, file-sharing tasks, heterogeneous master-slave platform, grid computing, iterative improvement.

1 Introduction

In this work, we investigate the scheduling of independent but file-sharing tasks on heterogeneous master-slave environments. This framework has recently been studied in [7], [8], [9], [15], [16] for adaptive scheduling of parameter-sweep-like applications in Grid environments. Such applications arise in the *Application Level Scheduling* (AppLeS) project [7]. In this framework, input files, which can be requested by multiple tasks, are initially stored in the master processor and slave processors have different network access bandwidths and computing powers. The objective is to find a schedule that minimizes the turnaround time of the target application on the given master-slave platform.

In Grid systems, the environment variables such as the execution times of tasks on heterogeneous processors and the bandwidth values of the network dynamically change due, respectively, to the loads of the processors and the congestion in the network. Since creating a good schedule depends on the quality of the information used, the system state must be monitored by an information agent to enable the generation of better schedules for the execution of the target application. Such an agent can estimate the network bandwidths and task-execution times by previous executions, machine benchmark values, or information provided by users. These estimations can be useful to create an adaptive scheduling tool. In our model, we assume that task-execution times and network bandwidth values remain constant during each schedule period, however, the dynamic nature of the processors and network is assumed to be modeled by using up-to-date values for these environment variables obtained by an information agent before each schedule generation period.

Task scheduling in such heterogeneous environments is harder than scheduling in homogeneous ones and it is an important problem for today's computational Grid [14] which contains highly heterogeneous environments. In a heterogeneous environment, highly interacting tasks which need the same files as inputs might have different favorite processors so that it may not be feasible to assign them to the same processor because of appropriate resource utilization. Even if such tasks may have the same favorite processor, that processor might have relatively low bandwidth so that assigning these tasks to that processor can increase the file transfer time, although this decision decreases the file transfer amount.

Several heuristics were recently proposed for the target framework. Casanova et al. [8], [9] extended three heuristics, namely, *MinMin*, *MaxMin*, and *Sufferage*, which were initially proposed in [21] for scheduling independent tasks. They used these extended heuristics in the *AppLeS Parameter Sweep Template* (APST) project [7]. They also proposed a new heuristic *XSufferage* exclusively for APST. After this work, Giersch et al. [15], [16] proposed several different heuristics which reduce the time complexity while preserving the quality of schedules. All these scheduling heuristics are based on the greedy choices that depend on the

• *The authors are with the Computer Engineering Department, Bilkent University, TR 06800, Ankara, Turkey.*
E-mail: {kamer, aykanat}@cs.bilkent.edu.tr.

Manuscript received 31 July 2004; revised 28 Jan. 2005; accepted 25 June 2005; published online 26 June 2006.
Recommended for acceptance by K.-J. Lin.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0189-0704.

momentary completion time values of tasks. We show that this greedy decision criterion cannot use the file sharing information effectively since completion time values are not sufficient to extract the global view of the interaction among the tasks.

Instead of the direct construction of schedules, we propose a three-phase scheduling approach which involves initial task assignment, refinement, and execution ordering phases. For the refinement phase, we propose an elegant hypergraph-partitioning-like formulation with two novel smooth objective functions. The effectiveness and efficiency of the iterative-improvement heuristics, which are widely and successfully used for hypergraph partitioning, depend on the smoothness of the objective functions they improve, but the actual schedule cost does not have this property. Fortunately, the smoothness of proposed objective functions enables the use of these heuristics for refining the task assignments successfully. The first assignment objective function represents an upper bound, while the second one represents a lower bound for the turnaround time of a schedule. The former one corresponds to a pessimistic view, while the latter one corresponds to an optimistic view for the execution scheme. Experimental results on a wide range of synthetically generated heterogeneous master-slave frameworks show that the proposed three-phase scheduling approach performs much better than the existing greedy constructive heuristics.

The rest of the paper is organized as follows: The details of the scheduling framework are presented in Section 2. Section 3 discusses the structure and flaws of the existing constructive heuristics. The background material on hypergraph partitioning problem and iterative-improvement heuristics is given in Section 4. Section 5 presents and discusses the models and methodologies used in the proposed refinement phase. Our implementation scheme and the complexity analysis for the proposed three-phase approach are given in Section 6. Section 7 briefly mentions the modifications needed for adapting the proposed approach to the clustered master-slave framework. The experimental evaluation of the proposed approach is presented in Section 8. Section 9 concludes the paper.

2 Framework

Here, we briefly summarize the target scheduling framework that consists of a class of applications, a computing platform, and a cost model.

2.1 Application Model

The target application is represented as a two tuple $\mathcal{A} = (\mathcal{T}, \mathcal{F})$. Here, $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ denotes the set of n independent tasks, each of which needs a subset of the set $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$ of m files as inputs. There is no data dependency or interprocess communication between the tasks. The only reason for an interaction among the tasks is the existence of files that are inputs to several tasks. Files can have different sizes; the size of a file f_k is denoted as $w(f_k)$. The set of files used by a task t_i is denoted as $files(t_i)$ and the total size of the files in $files(t_i)$ is denoted as $w(files(t_i))$, i.e., $w(files(t_i)) = \sum_{f_k \in files(t_i)} w(f_k)$. Finally, $|\mathcal{A}|$ denotes the total number of file requests in the application $\mathcal{A}$, i.e., $|\mathcal{A}| = \sum_{t_i \in \mathcal{T}} |files(t_i)|$.

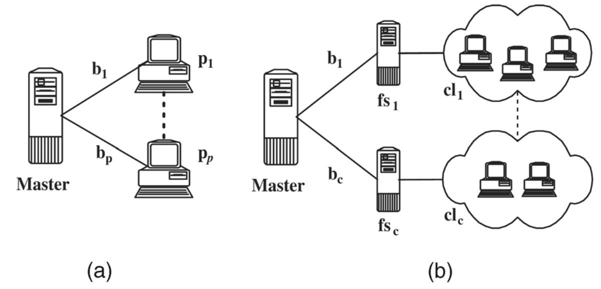


Fig. 1. Master-slave communication networks: (a) basic (adapted from [15], [16]) and (b) clustered (adapted from [8], [9]).

2.2 Heterogeneous Computing Model

The target computing platform is a heterogeneous system based on the well-known master-slave paradigm [5]. In this paradigm, there exists a master/server as a repository for all files and a set $\mathcal{P} = \{p_1, p_2, \ldots, p_p\}$ of p slaves/processors. Each processor can be any computing system from a single processor workstation to a high-performance parallel architecture. Fig. 1a represents the communication topology of the network as a graph.

The single-port communication model is assumed for the file transfers from the server to processors. In this model, only one processor can download a file from the server and only one file can be transferred by a processor at a time. The network heterogeneity is modeled by assigning different bandwidth values to the links between the server and processors. In Fig. 1a, the b_ℓ value, associated with the edge between the server and processor p_ℓ, represents the bandwidth from the server to processor p_ℓ, for $\ell = 1, \ldots, p$. Task executions and file transfers can overlap on a processor. That is, a processor can execute a task while it is downloading a file needed for another task that is scheduled for execution on the same processor. Note that Fig. 1a does not contain any communication links between processors in order to point out that the framework does not encapsulate the possibility of file exchange between processors instead of downloading from the server.

A clustered master-slave platform is also considered as the target computing environment. The clustered platform differs from the above-mentioned basic one in the following aspects: Each processor node of the basic master-slave platform effectively becomes a cluster of processors, which is served by a local file storage unit for that cluster. That is, we have a set $\mathcal{CL} = \{cl_1, cl_2, \ldots, cl_c\}$ of c clusters and a set $\mathcal{FS} = \{fs_1, fs_2, \ldots, fs_c\}$ of c local file storage units, where fs_i is the file storage unit of cluster cl_i. fs_i is responsible for storing the files that are transferred to cluster cl_i, until the end of the schedule. Fig. 1b displays the main features of this framework. The network heterogeneity is modeled by assigning different bandwidth values to the links between the server and the file storage units of the clusters. The intracluster communication costs due to the local file transfers from a file storage unit are not considered because intracluster file transfers are assumed to be much faster than the file transfers from the server.

For both master-slave platforms, the task and processor heterogeneity are modeled by incorporating different execution times for each task on different processors. We

```
1: while there remains a task to schedule do
2:    for each unscheduled task $t_i$ do
3:       for each processor $p_\ell$ do
4:          Evaluate completion time $CT(t_i, p_\ell)$ of $t_i$ on $p_\ell$
5:       Evaluate schedule cost $f(CT(t_i, p_1), \ldots, CT(t_i, p_p))$ for $t_i$
6:    Choose task $t_{i_b}$ with the "best" schedule cost
7:    Pick the best processor $p_{\ell_b}$ for $t_{i_b}$ with min. completion time
8:    Schedule $t_{i_b}$ on $p_{\ell_b}$ and its file transfers
9:    Mark $t_{i_b}$ as scheduled
```

Fig. 2. Structure of heuristics by Casanova et al. [8], [9].

TABLE 1
Definitions for the Heuristics Proposed by Casanova et al. [8], [9]

Heuristics	Function f	best
MinMin	minimum of all $CT(t_i, p_\ell)$ values	minimum
MaxMin	minimum of all $CT(t_i, p_\ell)$ values	maximum
Sufferage	difference between 2nd minimum and minimum of all $CT(t_i, p_\ell)$ values	maximum

use $c_{i\ell}$ to denote the execution time of task t_i on a processor p_ℓ. The estimated execution-time values of the tasks are stored in an $n \times p$ expected time to compute the (ETC) matrix. The ETC matrix can be *consistent* or *inconsistent* in terms of the relation between execution times of different tasks [3]. In a consistent ETC matrix, if a processor executes a task t_i faster than another processor, then it executes all other tasks faster than that processor. If there is no such relation between execution times, then the ETC matrix is said to be inconsistent. We believe that an inconsistent ETC matrix is a better model for the Grid system since Grid contains very heterogeneous computing resources with different task execution characteristics [1].

2.3 Cost Model

The cost of a schedule is the turnaround time, which is the parallel execution time of the application on the computing environment. The schedule can be considered as a timeline which starts with the first file transfer from the server and ends with the completion of the last task execution. So, the objective of the target scheduling problem is to assign the tasks of the target application to suitable processors and order the inter and intraprocessor task executions in such a way that the turnaround time is minimized.

For clarity, we give the important definitions and assumptions only for the basic master-slave platform. These concepts can be easily modified for the clustered master-slave platform. The time spent for the transfer of file f_k from the server to processor p_ℓ is $w(f_k)/b_\ell$. A task t_i becomes ready for execution on a processor p_ℓ after all its input files are transferred by the processor from the server. The transferred files are assumed to be stored by the processors until the end of the schedule, so, for a pair of tasks t_i and t_j assigned to the same processor p_ℓ, a file needed by both t_i and t_j is transferred to p_ℓ only once.

3 Existing Constructive Scheduling Heuristics

In this section, we first summarize the structure of existing constructive scheduling heuristics and then discuss their flaws.

3.1 Structure

Fig. 2 shows the structure of the heuristics used by Casanova et al. [8], [9]. In Fig. 2, the completion time $CT(t_i, p_\ell)$ of task t_i on processor p_ℓ is computed by taking the previously scheduled tasks into account. That is, the file transfers for unscheduled tasks cannot be initialized before the file transfers for scheduled tasks and the executions of unscheduled tasks on a candidate processor cannot be initialized before the completion of the scheduled tasks on the same processor. The scheduling objective function f and the meaning of the "best" characterize these heuristics as shown in Table 1. As seen in Fig. 2, computing the completion times for all task-processor pairs takes $O(pn + p|\mathcal{A}|)$ time for each scheduling decision. As this decision is made once for each task, the total time complexity of these heuristics is $O(pn^2 + pn|\mathcal{A}|)$.

After Casanova et al. [8], [9], Giersch et al. [15], [16] proposed several different heuristics. These heuristics have better time complexity and their solution quality is comparable with those of the previous heuristics. Fig. 3 shows the structure of these heuristics. Table 2 displays the objective functions proposed by Giersch et al. [15], [16] for a task-processor pair (t_i, p_ℓ) based on the computation time $Comp(t_i, p_\ell) = c_{i\ell}$ and communication time $Comm(t_i, p_\ell) = w(files(t_i))/b_\ell$ values of t_i when it is executed on p_ℓ. The additional policies *readiness*, *shared*, and *locality* proposed by Giersch et al. [15], [16] are also explained in Table 2. As seen in Fig. 3, the heuristics construct a task list for each processor, which is sorted with respect to various objective values in Step 4. For an efficient implementation, we compute the total file sizes for all tasks, i.e., $w(files(t_i))$ values, in $\Theta(|\mathcal{A}|)$ time in a preprocessing step. In this way, the objective value computations for all task-processor pairs take $\Theta(pn + |\mathcal{A}|)$ time, so the construction of all sorted lists takes $O(pn \log n + |\mathcal{A}|)$ time. The while loop for scheduling tasks in Step 5 takes $O(pn|\mathcal{A}|)$ time. So, the overall time complexity becomes $O(pn \log n + pn|\mathcal{A}|)$.

3.2 Flaws

The task-processor pair selection according to the momentary completion time values is the greedy decision criterion

```
1: for each processor $p_\ell$ do
2:    for each task $t_i$ do
3:       Evaluate $OBJECTIVE(t_i, p_\ell)$
4:    Build the list $L(p_\ell)$ of the tasks sorted according
      to the value of $OBJECTIVE(t_i, p_\ell)$
5: while there remains a task to schedule do
6:    for each processor $p_\ell$ do
7:       Let $t_i$ be the first unscheduled task in $L(p_\ell)$
8:       Evaluate completion time $CT(t_i, p_\ell)$ of $t_i$ at $p_\ell$
9:    Pick a task-processor pair $(t_{i_b}, p_{\ell_b})$ with
      minimum completion time
10:   Schedule $t_{i_b}$ on $p_{\ell_b}$ and its file transfers
11:   Mark $t_{i_b}$ as scheduled
```

Fig. 3. Structure of heuristics by Giersch et al. [15], [16].

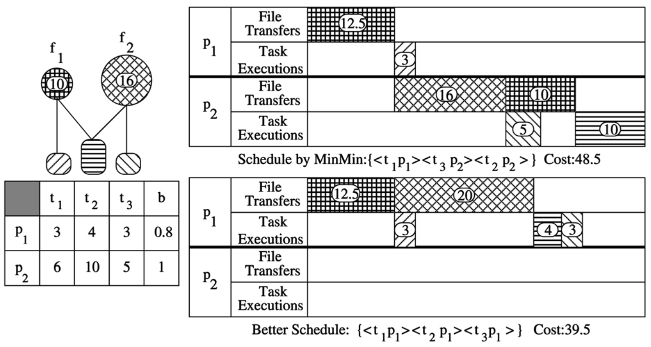
TABLE 2
Definitions for the Heuristics Proposed by Giersch et al. [15], [16]

Heuristic	Objective Function	Task Selection Order w.r.t. Objective Func.
Computation	$Comp(t_i, p_\ell)$	increasing
Communication	$Comm(t_i, p_\ell)$	increasing
Duration	$Comp(t_i, p_\ell) + Comm(t_i, p_\ell)$	increasing
Payoff	$Comp(t_i, p_\ell) \ / \ Comm(t_i, p_\ell)$	decreasing
Advance	$Comp(t_i, p_\ell) - Comm(t_i, p_\ell)$	decreasing

Additional Policy	Explanation
Readiness	Selects a ready task for a processor if one exists. A task is called ready for processor p_ℓ if the transfers of all input files of the task to p_ℓ are previously scheduled.
Shared	While calculating $w(files(t_i))$, scaled versions of file sizes are used. The scaled size of a file is calculated by dividing its original size to the number of tasks that need this file as an input. This policy is redundant with the Computation objective function
Locality	To reduce the file transfer amount, locality tries to avoid assigning a task to a processor if some files used by the task were already scheduled to be transferred to another processor.

commonly used in all existing constructive heuristics. This criterion suffers from ineffective use of information about file sharing among the tasks. This flaw is likely to increase with the increasing amount of file sharing and can incur extra file transfers in the resulting schedule. Since total file transfer amount from the server is a bottleneck under the single-port communication model, extra file transfers can deteriorate the quality of the schedule, especially for communication-intensive tasks. We say a task is communication-intensive if the file transfer time for the task dominates its execution time.

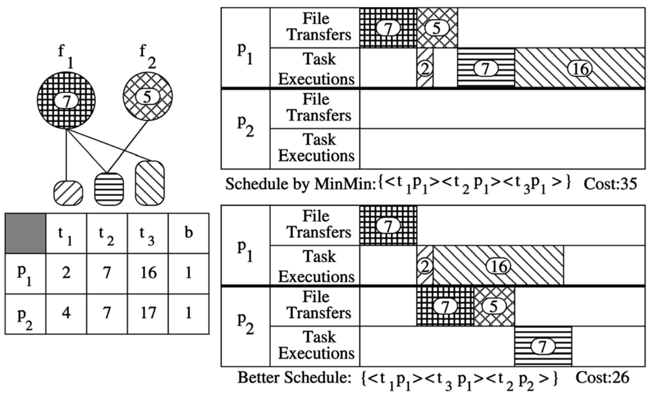
Fig. 4 displays a sample communication-intensive application with three tasks and two large files. As seen in the figure, *MinMin* schedules t_3 on p_2 after scheduling t_1 on p_1, ignoring the fact that t_2 needs both files. This greedy choice incurs an extra transfer of file f_1. However, there is another schedule without this extra file transfer and with much less turnaround time, as shown in Fig. 4.



	t_1	t_2	t_3	b
p_1	3	4	3	0.8
p_2	6	10	5	1

f1
f2
10
16
File Transfers
Task Executions
12.5
3
16
10
5
10
Schedule by MinMin:{<t1p1><t3p2><t2p2>} Cost:48.5
12.5
20
3
4
3
Better Schedule: {<t1p1><t2p1><t3p1>} Cost:39.5

Fig. 4. A flaw of the greedy constructive approach for communication-intensive tasks.



	t_1	t_2	t_3	b
p_1	2	7	16	1
p_2	4	7	17	1

f1
f2
7
5
File Transfers
Task Executions
7
5
2
7
16
Schedule by MinMin:{<t1p1><t2p1><t3p1>} Cost:35
7
2
16
7
5
7
Better Schedule: {<t1p1><t3p1><t2p2>} Cost:26

Fig. 5. Another flaw of the greedy constructive approach.

Although extra file transfers constitute crucial bottleneck, they can also be necessary for efficient utilization of computational resources, especially when tasks have comparable computation and communication times. However, if initial scheduling decisions create a computational imbalance, the following greedy decisions may aggravate this problem. The processors that are computationally overloaded due to the previous scheduling decisions are likely to be more favorable for future task assignments since, in addition to already being favorable, they have lots of file transfers already scheduled.

Fig. 5 illustrates a sample application with three tasks and two small files. As seen in the figure, *MinMin* schedules t_2 on p_1 after scheduling t_1 on p_1 because of the cost of the extra transfer of file f_1 in case of scheduling t_2 on p_2. However, *MinMin* ignores the fact that scheduling t_3 on p_1 does not require any extra file transfer. After faster processor p_1 is overloaded by these two scheduling decisions, it becomes more favorable since both f_1 and f_2 are already transferred to p_1. Finally, *MinMin* schedules t_3 on the overloaded processor p_1 because of the extra transfer of file f_1 required for the other choice of scheduling t_3 on the empty processor p_2. However, there is a much better schedule that utilizes both processors, as shown in Fig. 5.

4 Hypergraph Partitioning and Iterative-Improvement Heuristics

In this section, we present the background material on hypergraph partitioning and iterative-improvement heuristics which are exploited in our proposed scheduling approach.

4.1 Hypergraph Partitioning Problem

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among these vertices [6]. Every net n_k in $\mathcal{N}$ is a subset of vertices, i.e., $n_k \subseteq \mathcal{V}$. The vertices in a net n_k are called its pins. The set of nets that contain vertex v_i is denoted as $nets(v_i)$. The total number of pins denotes the size of the hypergraph. Weights can be associated with vertices and nets. Graph is a special instance of hypergraph such that each net has exactly two pins.

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is a K-way vertex partition of $\mathcal{H}$ if each part $\mathcal{V}_k$ is nonempty, parts are pairwise disjoint, and the union of parts gives $\mathcal{V}$. In Π, a net is said to connect a part if it has at least one pin in that part. The connectivity set Λ_k of a net n_k is the set of parts that n_k connects and the connectivity $\lambda_k = |\Lambda_k|$ of n_k is the number of parts it connects. In Π, the weight of a part is the sum of the weights of the vertices in that part.

The K-way hypergraph partitioning problem is defined as finding a K-way vertex partition that optimizes a given objective function while preserving a given partitioning constraint. The *connectivity*-1 metric is frequently used in VLSI circuit partitioning [19] and scientific computing [4], [10], [23]. The partitioning objective in this metric is the minimization of $CutSize(\Pi)$, which is given as:

$$CutSize(\Pi) = \sum_{n_k \in \mathcal{N}} w(n_k)(\lambda_k - 1), \tag{1}$$

where $w(n_k)$ denotes the weight of net n_k. The partitioning constraint is to maintain a balance on the part weights, i.e.,

$$(W_{max} - W_{avg})/W_{avg} \leq \epsilon, \tag{2}$$

where W_{max} is the weight of the part with the maximum weight, W_{avg} is the average part weight, and ϵ is a predetermined imbalance ratio.

4.2 Iterative-Improvement Heuristics

The refinement heuristics proposed in this work are based on the iterative-improvement heuristics introduced by Kernighan-Lin (KL) [18] and Fiducia-Mattheyses (FM) [13] for graph/hypergraph partitioning. Both KL and FM are move-based approaches with the neighborhood operator of swapping a pair of vertices between parts or shifting a vertex from one part to another, respectively. These heuristics have been widely used for graph/hypergraph partitioning by the VLSI [19] and scientific computing [4], [10], [11], [17], [23] communities because of their effectiveness with good-quality results and efficiency with short runtimes.

The FM algorithm, starting from an initial bipartition, 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 bipartition by moving a vertex to the other part. Several FM variants were proposed for the generalization of the approach to K-way refinement [22].

5 Proposed Refinement Approach

Both the effectiveness and efficiency of FM-based heuristics depend on "the smoothness" of the objective function over the neighborhood structure [2], i.e., the neighborhood operator should be small and local. However, a direct generalization of FM-based heuristics to the task scheduling problem suffers from disturbing this smoothness criterion. Removing a task from a processor and scheduling it among previously scheduled tasks of another processor incurs a global perturbation in the schedule because previously scheduled tasks affect the initialization and completion times of executions of the waiting tasks. Due to this global effect of a task move, computing the gain, which is the change in the turnaround time, is a time consuming work and its time complexity is as high as computing the turnaround time of a given schedule.

In order to alleviate the above problem, we consider the task scheduling problem as involving two consecutive processes: the task assignment process which determines the task-to-processor assignment and the execution-ordering process which determines the order of inter and intraprocessor task executions. This view enables the use of FM-based heuristics effectively and efficiently in the task-assignment process by proposing smooth assignment objective functions that are closely related to the turnaround time of a schedule. This refined task-to-processor assignment can then be used to generate better schedules during the execution-ordering process.

5.1 Hypergraph Partitioning Models for Task Assignment in Heterogeneous Environments

We propose using a hypergraph $\mathcal{H}_A = (\mathcal{T}, \mathcal{F})$ to represent the interaction among tasks in the target application $\mathcal{A} = (\mathcal{T}, \mathcal{F})$. In this model, the vertices of the hypergraph represent the tasks and the nets represent the files. The pins of a net correspond to the tasks that use the respective file. Because of this natural correspondence between a target application and a hypergraph, we describe our algorithms using the problem-specific notation of Section 2 instead of hypergraph-specific notation, as much as possible, for clarity of presentation. For example, we will use $files(t_i)$ instead of $nets(t_i)$. The size of a file is the weight of the corresponding net. A p-way vertex partition $\Pi = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_p\}$ of $\mathcal{H}_A$ can be decoded as inducing a task-to-processor assignment for a target schedule. That is, all tasks in a part $\mathcal{T}_\ell$ will be executed by processor p_ℓ in the target schedule.

Successful hypergraph partitioning formulations have recently been proposed for solving the task-to-processor assignment problem arising in the parallelization of several applications on homogeneous platforms [4], [10], [11], [23]. If the master-slave platform is homogeneous, i.e., processors are identical and server-to-processor bandwidth values are equal, the partitioning objective given in (1) and the load balancing constraint given in (2) can be used effectively and efficiently for the refinement. However, the heterogeneity of the environment brings difficulties to the formulation of the task assignment problem. For this reason, we propose new assignment objectives, which can be generalized as partitioning objectives of the hypergraph partitioning problem for heterogeneous environments.

In a given task-to-processor assignment Π, each file will be transferred at least once since it is used by at least one task. Consider a cut net n_k with connectivity λ_k in Π. It is clear that $\lambda_k - 1$ denotes the number of additional transfers of file f_k incurred by Π. Hence, $w(f_k)(\lambda_k - 1)$ represents the additional transfer volume, whereas $w(f_k)\lambda_k$ denotes the total transfer volume for file f_k. That is, the *connectivity* metric is the correct metric, rather than the *connectivity*-1 metric, for encoding the total file transfer volume in a given task-to-processor assignment, as shown below:

$$CommVol(\Pi) = \sum_{f_k \in \mathcal{F}} w(f_k)\lambda_k. \tag{3}$$

Note that minimizing $CommVol(\Pi)$ is equal to minimizing $CutSize(\Pi)$ since $CommVol(\Pi) = CutSize(\Pi) + \sum_{f_k \in \mathcal{F}} w(f_k)$ and the second term is only a constant factor.

If the network is homogeneous, (3) can also be used to represent the total transfer time by normalizing file sizes with respect to the bandwidth values. That is, minimization of total file transfer volume and total file transfer time is equivalent in the homogeneous case. To encapsulate the network heterogeneity of the target master-slave platform, we need to modify the conventional definition of the connectivity λ_k of a net n_k in which different parts connected by n_k make equal contribution to λ_k. Since we want total file transfer time as the real communication cost and bandwidth values of the links are different, we define a *heterogeneous connectivity* λ'_k of a file f_k as:

$$\lambda'_k = \sum_{p_\ell \in \Lambda_k} \frac{1}{b_\ell}, \tag{4}$$

where Λ_k denotes the set of processors that have at least one task needing f_k as input.

Then, total communication time, i.e., total file transfer time, can be defined as:

$$CommTime(\Pi) = \sum_{f_k \in \mathcal{F}} w(f_k)\lambda'_k. \tag{5}$$

The computational cost of a task-to-processor assignment Π to the environment is the load of the maximally loaded processor since computations are done in parallel. That is,

$$CompTime(\Pi) = \max_{\ell} \left(\sum_{t_i \in \mathcal{T}_\ell} c_{i\ell} \right). \tag{6}$$

The processor heterogeneity creates difficulties in modeling the computational cost of a task-to-processor assignment Π. In homogeneous environments, the average part weight (W_{avg} in (2)) can be considered as a lower bound for $CompTime(\Pi)$ if a vertex weight represents a computational cost to its part. Similarly, W_{max} can be considered as $CompTime(\Pi)$, which is the exact parallel computational cost of the partition. So, in homogeneous environments, the load balancing constraint given in (2) can be used for minimizing $CompTime(\Pi)$. However, in heterogeneous environments, since the same task incurs different computational costs to different processors, a lower bound for parallel computational cost of Π cannot be treated as a balancing constraint as in the hypergraph partitioning formulation for homogeneous environments. So, we should rather include $CompTime(\Pi)$ explicitly in the assignment objective function as well as $CommTime(\Pi)$.

Here, we propose two novel assignment objective functions. The first one represents an upper bound for the turnaround time of a schedule with a pessimistic view that assumes no overlap between communication and computation. We call it a pessimistic view since it excludes the possibility of communication-computation overlap between different processors as well as on the same processor. For example, a schedule in which all task executions commence only after the completion of all file transfers from the server constitutes a typical schedule for this pessimistic view. Under this pessimistic view, the turnaround times of all possible schedules that can be derived from a given task-to-processor assignment Π are bounded above by

$$UBTime(\Pi) = CommTime(\Pi) + CompTime(\Pi). \tag{7}$$

Note that this upper bound is independent of the order of task executions for a given task-to-processor assignment Π.

The second assignment objective function represents a lower bound for the turnaround time of a schedule. As mentioned in Section 2, a processor can execute a task while it or another processor is transferring a file from the server, so computation and communication can overlap. Even with an optimistic view that assumes complete overlap between communication and computation, the turnaround times of all possible schedules that can be derived from a given task-to-processor assignment Π are bounded below by

$$LBTime(\Pi) = max\{CommTime(\Pi), CompTime(\Pi)\}. \tag{8}$$

Note that this lower bound is also independent of the order of task executions for a given task-to-processor assignment Π. This bound is unreachable because of the nonoverlapping cases at the very beginning and end of a schedule. A schedule must begin with a file transfer and the respective task execution cannot be initialized until the completion of this file transfer. A schedule must also end with a task execution on its bottleneck processor. All file transfers from the server to all processors should be completed before the completion of the execution of this task. The length of these nonoverlapping intervals is negligible compared to the turnaround time of a schedule due to the large number of tasks.

These two assignment objectives are closely related to the turnaround time of a schedule and their minimization can generate good task-to-processor assignments which can be used to obtain schedules with better turnaround times. Instead of one objective, as in hypergraph partitioning problem, we have two assignment objectives and there are various options to improve them. The details of our approach are given in the following section.

5.2 Structure of the Refinement Heuristics

It is clear that the effectiveness of the refinement phase depends on considering both objective functions simultaneously. Since the objective functions represent upper and lower bounds for the turnaround time, the overall objective should be closing the gap between these two objective functions while minimizing both of them. For this purpose, we propose using an alternating refinement scheme in which refinement according to one objective function follows refinement according to the other one in a repeated pattern. The refinement of a task-to-processor assignment Π according to *UBTime*(Π) or $LBTime(\Pi)$ is referred to here as the *UB-Refinement* or *LB-Refinement* stage, respectively.

In the alternating scheme, using FM-based heuristics separately and independently for the minimization of the respective objective function is only a partial remedy for satisfying the overall objective. While choosing the best move according to one objective function, the effect of the move according to the other one should also be considered indirectly since the minimization of one objective function may degrade the value of the other one. For this purpose, we propose modifying the move selection policy of the FM-based approach accordingly in the LB-Refinement stage and/or in the UB-Refinement stage.

In the general FM-based approach, the *best move* associated with a task corresponds to reassigning the task to another processor that incurs maximum decrease in the respective objective function. In the proposed modification, a two-level gain scheme is applied to determine the best move associated with a task through considering the respective objective function as the primary one while considering the other objective function as the secondary one. For the first level, a *good move* concept is introduced which selects the moves that decrease the primary objective function. In the second level, the best move associated with that vertex is selected among these good moves that incurs the minimum increase to the secondary objective function.

In this work, we recommend applying the proposed two-level gain computation scheme either to both refinement stages or only to the LB-Refinement stage. The reasons for the latter choice are as follows: First, the variations in the task-move gains are expected to be larger in *UBTime*(Π) compared to *LBTime*(Π). Second, *UBTime*(Π) is a relatively loose bound compared to *LBTime*(Π). So, providing more freedom in the minimization of the loose upper bound while incorporating the constraint to the minimization of the relatively tight lower bound is expected to be more effective for reducing the gap between these two bounds. Based on these two reasons, we also recommend starting the alternating refinement sequence with the UB-Refinement stage. Our experimental results given in Section 8 verify our expectations.

Here, we describe the implementation scheme which adopts the two-level gain computation scheme in only the LB-Refinement stage for the sake of presenting the use of both the conventional and proposed gain computation schemes. Both the UB and LB-Refinement stages contain multiple FM-like passes. In each pass, all tasks are visited in random order. The best move associated with each visited task is computed according to the adopted gain computation scheme and this move is realized if it incurs a positive gain according to the respective objective function. Note that each task is visited exactly once in a pass and these passes are repeated until a stopping criterion is met. Fig. 6 shows the general structures of the UB and LB-Refinement stages, respectively. In this figure, $Map(t_i)$ denotes the processor to which task t_i is currently assigned.

For the sake of runtime efficiency of move gain computations, a task move is considered as a two-step process: A task leaves the source processor to which it is assigned and arrives at the destination processor as a reassignment. So, the move gain can be considered as the *leave gain* minus *arrival loss*. The leave gain of a task t_i may include two subgains. The first subgain can be obtained in case of a decrease in $CompTime(\Pi)$ due to a leave from a processor with maximum computational load. The second subgain can be obtained in case of a decrease in $CommTime(\Pi)$ due to the existence of some files that are needed by t_i and critical to the source processor. We say a file is *critical* to a processor if it is an input to a single task assigned to that processor. This critical file concept corresponds to the critical net concept used in hypergraph partitioning.

```
UB-Refinement(Π)
1:  while a stopping criterion is not met do
2:      Create a random visit order of tasks
3:      for each task $t_i$ in this random order do
4:          $leaveGain \leftarrow$ UB-ComputeLeaveGain($t_i$)
5:          if $leaveGain > 0$ then
6:              $p_{\ell_b} \leftarrow$ UB-SelectBestMove($t_i, leaveGain$)
7:              if $p_{\ell_b}$ is not equal to $Map(t_i)$ then
8:                  UpdateGlobalData($t_i, p_{\ell_b}$)
9:                  $Map(t_i) \leftarrow p_{\ell_b}$
LB-Refinement(Π)
1:  while a stopping criterion is not met do
2:      Create a random visit order of tasks
3:      for each task $t_i$ in this random order do
4:          $\{commLeaveGain, compLeaveGain\} \leftarrow$
                LB-ComputeLeaveGain($t_i$)
5:          if ($CommCost(\Pi) > CompCost(\Pi)$ and
                                $commLeaveGain > 0$) or
               ($CompCost(\Pi) > CommCost(\Pi)$ and
                                $compLeaveGain > 0$) then
6:              $\{p_{\ell_b}, bestCommGain, bestCompGain\} \leftarrow$
                    LB-SelectBestMove($t_i, commLeaveGain,$
                                      $compLeaveGain$)
7:              if $p_{\ell_b}$ is not equal to $Map(t_i)$ then
8:                  UpdateGlobalData($t_i, p_{\ell_b}$)
9:                  $CommCost(\Pi) \leftarrow CommCost(\Pi) - bestCommGain$
10:                 $CompCost(\Pi) \leftarrow CompCost(\Pi) - bestCompGain$
11:                 $Map(t_i) \leftarrow p_{\ell_b}$
```

Fig. 6. Structure of the UB and LB-refinement stages.

After computing the leave gain of task t_i, an arrival loss value is computed for each destination processor p_ℓ. This value represents the increase in the objective function when t_i is assigned to p_ℓ. Such a loss can occur due to the increase in $CommTime(\Pi)$ and/or $CompTime(\Pi)$. Clearly, if the leave gain of t_i is negative, it is impossible to obtain a positive gain in total since an arrival cannot increase the total move gain. In Fig. 6, ℓ_b denotes the index of the best processor selected for the move of the visited task.

The main data structures needed for the implementations are as follows: δ is a 2D file-to-processor counter array, where $\delta(f_k, p_\ell)$ denotes the number of tasks that need file f_k and are assigned to processor p_ℓ. Note that, if $\delta(f_k, p_\ell) = 1$, then f_k is critical to p_ℓ. $Load$ is a 1D array used to maintain the computational loads of processors in terms of time units. Map is a 1D array used to represent task-to-processor assignment. A linked-list Λ_k is used for each file f_k to maintain the set of processors that need f_k. ℓ_1 and ℓ_2 are used to maintain the indices of the processors with the maximum and second maximum computational loads, respectively. Fig. 7 displays the pseudocode for the global update operations common to both UB and LB-Refinement stages.

Fig. 8 displays the algorithms used for leave gain and arrival loss computations in the UB-Refinement stage. Recall that the conventional gain computation scheme is adopted in this stage. As seen in Fig. 8, both gains due to a decrease in total file transfer time and maximum computational load are added to the leave gain of task t_i because they are both included in the objective function. While computing the gain due to the decrease in $CompTime(\Pi)$, the maximum computational load in case of removal of t_i from processor $Map(t_i)$ is calculated and saved in a variable called $leaveMaxLoad$. This information will be used for computing arrival losses for t_i.

```
UpdateGlobalData($t_i, p_{\ell_b}$)
1:  $p_\ell \leftarrow Map(t_i)$
2:  for all $f_k \in files(t_i)$ do
3:      $\delta(f_k, p_\ell) \leftarrow \delta(f_k, p_\ell) - 1$
4:      if $\delta(f_k, p_\ell) = 0$ then
5:          $\Lambda_k \leftarrow \Lambda_k - \{p_\ell\}$
6:      $\delta(f_k, p_{\ell_b}) \leftarrow \delta(f_k, p_{\ell_b}) + 1$
7:      if $\delta(f_k, p_{\ell_b}) = 1$ then
8:          $\Lambda_k \leftarrow \Lambda_k \cup \{p_{\ell_b}\}$
9:  $Load(p_{\ell_b}) \leftarrow Load(p_{\ell_b}) + c_{i\ell_b}$
10: $Load(p_\ell) \leftarrow Load(p_\ell) - c_{i\ell}$
11: Update $\ell_1$ and $\ell_2$
```

Fig. 7. Global update operations for the UB and LB-refinement stages.

Fig. 9 displays the algorithms used for leave gain and arrival loss computations in LB-Refinement stage. Recall that the proposed two-level gain computation scheme is adopted in this stage. If $CommTime(\Pi) > CompTime(\Pi)$, then LB-Refinement tries to minimize the total file transfer time, otherwise it tries to minimize the maximum computational load. In this stage, the good moves are the ones with a positive gain for the primary objective function $LBTime(\Pi)$ and the best move is the one that gives minimum degradation to the secondary objective function *UBTime* (Π).

```
UB-ComputeLeaveGain($t_i$)
1:  $p_\ell \leftarrow Map(t_i)$
2:  $leaveGain \leftarrow 0$
3:  for all $f_k \in files(t_i)$ do
4:      if $\delta(f_k, p_\ell) = 1$ then
5:          $leaveGain \leftarrow leaveGain + w(f_k)/b_\ell$
6:  if $p_\ell = p_{\ell_1}$ then
7:      if $(Load(p_\ell) - c_{i\ell}) > Load(p_{\ell_2})$ then
8:          $leaveGain \leftarrow leaveGain + c_{i\ell}$
9:          $leaveMaxLoad \leftarrow Load(p_\ell) - c_{i\ell}$
10:     else
11:         $leaveGain \leftarrow leaveGain + (Load(p_\ell) - Load(p_{\ell_2}))$
12:         $leaveMaxLoad \leftarrow Load(p_{\ell_2})$
13: else
14:     $leaveMaxLoad \leftarrow Load(p_{\ell_1})$
15: return $leaveGain$

UB-SelectBestMove($t_i, leaveGain$)
1:  $p_{\ell_b} \leftarrow Map(t_i)$
2:  for each candidate processor $p_\ell$ do
3:      $arrivalLoss(p_\ell) \leftarrow w(files(t_i))/b_\ell$
4:  for all files $f_k \in files(t_i)$ do
5:      for each candidate processor $p_\ell$ in $\Lambda_k$ do
6:          $arrivalLoss(p_\ell) \leftarrow arrivalLoss(p_\ell) - w(f_k)/b_\ell$
7:  $bestMoveGain \leftarrow 0$
8:  for each candidate processor $p_\ell$ do
9:      if $(Load(p_\ell) + c_{i\ell}) > leaveMaxLoad$ then
10:         $arrivalLoss(p_\ell) \leftarrow arrivalLoss(p_\ell)$
                        $+ (Load(p_\ell) + c_{i\ell}) - leaveMaxLoad$
11:     $moveGain \leftarrow leaveGain - arrivalLoss(p_\ell)$
12:     if $moveGain > bestMoveGain$ then
13:         $bestMoveGain \leftarrow moveGain$
14:         $p_{\ell_b} \leftarrow p_\ell$
15: return $p_{\ell_b}$
```

Fig. 8. UB-refinement heuristics: leave gain computation for task t_i; arrival loss computations and best processor selection for task t_i.

```
LB-ComputeLeaveGain($t_i$)
1:  $p_\ell \leftarrow Map(t_i)$
2:  $commLeaveGain \leftarrow 0$
3:  $compLeaveGain \leftarrow 0$
4:  for all $f_k \in files(t_i)$ do
5:      if $\delta(f_k, p_\ell) = 1$ then
6:          $commLeaveGain \leftarrow commLeaveGain + w(f_k)/b_\ell$
7:  if $p_\ell = p_{\ell_1}$ then
8:      if $(Load(p_\ell) - c_{i\ell}) > Load(p_{\ell_2})$ then
9:          $compLeaveGain \leftarrow c_{i\ell}$
10:         $leaveMaxLoad \leftarrow Load(p_\ell) - c_{i\ell}$
11:     else
12:         $compLeaveGain \leftarrow Load(p_\ell) - Load(p_{\ell_2})$
13:         $leaveMaxLoad \leftarrow Load(p_{\ell_2})$
14: else
15:     $leaveMaxLoad \leftarrow Load(p_{\ell_1})$
16: return $\{commLeaveGain, compLeaveGain\}$

LB-SelectBestMove($t_i, commLeaveGain, compLeaveGain$)
1:  $p_{\ell_b} \leftarrow Map(t_i)$
2:  for each candidate processor $p_\ell$ do
3:      $arrivalCommLoss(p_\ell) \leftarrow w(files(t_i))/b_\ell$
4:  for all files $f_k \in files(t_i)$ do
5:      for each candidate processor $p_\ell$ in $\Lambda_k$ do
6:          $arrivalCommLoss(p_\ell) \leftarrow arrivalCommLoss(p_\ell)$
                                        $- w(f_k)/b_\ell$
7:  $bestUBDamage \leftarrow -\infty$
8:  for each candidate processor $p_\ell$ do
9:      $arrivalCompLoss \leftarrow 0$
10:     if $(Load(p_\ell) + c_{i\ell}) > leaveMaxLoad$ then
11:         $arrivalCompLoss \leftarrow (Load(p_\ell) + c_{i\ell}) - leaveMaxLoad$
12:     $commGain \leftarrow commLeaveGain - arrivalCommLoss(p_\ell)$
13:     $compGain \leftarrow compLeaveGain - arrivalCompLoss$
14:     if $CommTime(\Pi) > CompTime(\Pi)$ then
15:         $moveGain \leftarrow commGain$
16:     else if $CommTime(\Pi) < CompTime(\Pi)$ then
17:         $moveGain \leftarrow compGain$
18:     if $moveGain > 0$ then
19:         if $commGain + compGain > bestUBDamage$ then
20:             $bestUBDamage \leftarrow commGain + compGain$
21:             $p_{\ell_b} \leftarrow p_\ell$
22:             $bestCommGain \leftarrow commGain$
23:             $bestCompGain \leftarrow compGain$
24: return $\{p_{\ell_b}, bestCommGain, bestCompGain\}$
```

Fig. 9. LB-refinement heuristics: leave gain computation for task t_i; arrival loss computations and best processor selection for task t_i.

6 Implementation Choices

The proposed scheduling heuristic involves three phases: *initial task assignment*, *refinement*, and *execution ordering*. In this section, we briefly describe each phase and give a complexity analysis for the overall approach.

6.1 Initial Task Assignment Phase

In this phase, initial task-to-processor assignments are derived from the schedules created by some of the existing constructive scheduling heuristics. We prefer this approach to a direct task-to-processor assignment heuristic because the proposed refinement heuristics are developed by taking the flaws of existing constructive scheduling heuristics into account. For this purpose, we use the heuristics proposed by Giersch et al. [15], [16] because of their short runtimes. The additional policies are not used, but all five of the heuristics, each having a different objective function, are used since their relative performances vary with the computation-to-communication ratio characteristics of applications. Each one of the five initial task-to-processor assignments obtained in this way is fed to the next two phases to obtain five schedules. At last, the best schedule in

```
ExecutionOrdering($\Pi$)
1: for each processor $p_\ell$ do
2:     for each task $t_i$ assigned to $p_\ell$ in $\Pi$ do
3:         Evaluate $OBJECTIVE\,(t_i, p_\ell)$
4:     Build the list $L(p_\ell)$ of the tasks that are assigned to $p_\ell$
       sorted according to the value of $OBJECTIVE\,(t_i, p_\ell)$
5: while there remains a task to schedule do
6:     Select the processor $p_\ell$ with maximum load
7:     Let $t_i$ be the first unscheduled task in $L(p_\ell)$
8:     Schedule $t_i$ on $p_\ell$ and its file transfers
9:     Mark $t_i$ as scheduled
```

Fig. 10. Execution ordering phase.

terms of the turnaround time is taken as the schedule for the target application.

6.2 Refinement Phase

Experiments show that the main improvement in the turnaround time of a schedule can be obtained within only a few passes, whereas the following passes incur negligible improvement. Because of this reason, we allow at most five passes in the UB and LB-Refinement stages. Likewise, the main improvement in the turnaround time of a schedule can be obtained within the first two alternating sequences of UB and LB-Refinement stages, whereas the following alternating sequences incur negligible improvement. For this reason, we allow at most three alternating sequences of UB and LB-Refinement stages.

6.3 Execution Ordering Phase

Each task-to-processor assignment Π obtained in the second phase is preserved while determining the inter and intraprocessor ordering of the task executions in this phase. Note that $CommTime(\Pi)$, $CompTime(\Pi)$ and, hence, the improved values of both objective functions remain the same as determined in the second phase. Fig. 10 shows the structure of the execution ordering heuristic used in this phase. As seen in the figure, the structure of the execution ordering heuristic is similar to the scheduling heuristics proposed by Giersch et al. [15], [16]. However, the proposed execution ordering heuristic is asymptotically faster since the same task-to-processor assignment Π is used during the course of the heuristic. For each Π, the execution ordering heuristic in Fig. 10 is run five times by using each one of the five objective functions proposed by Giersch et al. [15], [16] and the best schedule is selected for this Π.

6.4 Overall Complexity Analysis

As the heuristics proposed by Giersch et al. [15] are used in the initial task assignment phase, the time complexity of the first phase is $O(pn\log n + pn|\mathcal{A}|)$. In the refinement phase, each task is visited exactly once in each pass of the UB and LB-Refinement stages. Each vertex visit involves a leave gain and p arrival loss computations. The leave gain computations in each pass take $\Theta(|\mathcal{A}|)$ time since each file request of all tasks must be checked for being a critical file request or not. The arrival loss computations in each pass take $O(p|\mathcal{A}|)$ time because of the doubly-nested for loop at Steps 4-6 of best move selection heuristics in Fig. 8 and Fig. 9. The update operations within a pass take $O(p|\mathcal{A}|)$ time because of the $O(p)$ cost of removing processor ids from the connectivity sets (i.e., Λ linked lists) of files. As constant number of passes are involved in the refinement phase, the overall complexity of the second phase is $O(p|\mathcal{A}|)$.

In the execution ordering phase, computing all objective values takes $\Theta(n + |\mathcal{A}|)$ time, constructing sorted processor lists takes $O(n\log n)$ time, and finally ordering task executions takes $O(pn + |\mathcal{A}|)$ time. So, the overall time complexity of the third phase is $O(n\log n + |\mathcal{A}| + pn)$.

The time complexity of the initial task assignment phase dominates the overall complexity, so the proposed three-phase scheduling approach takes $O(pn\log n + pn|\mathcal{A}|)$ time.

7 Modifications for the Clustered Framework

In this section, we briefly explain the modifications needed for adapting both the existing and the proposed scheduling heuristics to the clustered master-slave framework.

7.1 Existing Constructive Scheduling Heuristics

In addition to the heuristics given in Table 1, Casanova et al. [8] also proposed a new heuristic called *XSufferage* for the clustered master-slave platforms. Unlike the other three scheduling heuristics, *XSufferage* computes cluster-based minimum completion times for each task t_i from $CT(t_i, p_\ell)$ values. The scheduling objective function f is the difference between the second minimum and the minimum of these minimum completion times and "best" is defined as maximum.

The communication related calculations for a task, such as objective values and file transfer completion times, need not be performed for all processors because these values are the same for all processors in a cluster. It is sufficient to perform these calculations for each cluster and this reduces the time complexity of the existing scheduling heuristics by replacing the term $pn|\mathcal{A}|$ with $cn|\mathcal{A}|$. Thus, the overall complexities of the heuristics proposed by Casanova et al. [8] and Giersch et al. [15] become $O(pn^2 + cn|\mathcal{A}|)$ and $O(pn\log n + cn|\mathcal{A}|)$, respectively.

For adapting the readiness policy [15] to the clustered platform, a task is called ready for a cluster if all of the input files of the task are available at that cluster. Similarly, for adapting the locality policy, the assignment of a task to a processor of a cluster is avoided if some of the input files of that task were already transferred to another cluster.

7.2 Proposed Scheduling Heuristic

The existence of local file storage units changes the hypergraph model slightly. Instead of processors, clusters are defined as parts in the original hypergraph partitioning problem so that the connectivity set Λ_k of each file f_k contains clusters instead of processors. So, the definition of the heterogeneous connectivity λ'_k of a net f_k becomes $\lambda'_k = \sum_{cl_i \in \Lambda_k} 1/b_i$, which can be used in (5) to compute the total communication time.

There are also some modifications needed in the definitions and global data used. We say a file is *critical* to a cluster if it is an input to a single task assigned to a processor in that cluster. As global data, we use $\delta(f_k, cl_i)$ to keep the number of tasks that use file f_k and are assigned to any processor in cluster cl_i.

The time complexity of the initial task assignment phase becomes $O(pn\log n + cn|\mathcal{A}|)$. In the refinement phase, the cost of leave gain computations remains the same, but the time complexity of the arrival loss computations and update operations become $O(c|\mathcal{A}|)$. So, the time complexity of the refinement phase reduces from $O(p|\mathcal{A}|)$ to $O(c|\mathcal{A}|)$. The complexity of the execution ordering phase remains the same, so the total complexity of the proposed scheduling heuristic is $O(pn\log n + cn|\mathcal{A}|)$ for clustered master-slave platform.

8 Experimental Results

We tested the performance of the proposed scheduling heuristic in comparison with the existing constructive heuristics by running a large number of experiments on synthetically generated heterogeneous master-slave platforms. The proposed and existing heuristics were implemented in C language on a Linux platform. All experiments were performed on a PC equipped with a 2.4 GHz Intel Pentium-IV processor and 2 Gbytes RAM. A total of 250 applications were created, each consisting of $n = 2,000$ tasks and $m = 2,000$ files. Each task in an application uses a random number of files between 1 and 10. The file sizes are randomly selected to vary between 100 Mbytes and 200 Gbytes.

The experiments vary with the computation-to-communication ratio $r = Comp_{avg}/Comm_{avg}$ of the target application. Here, $Comp_{avg} = (1/p)\sum_{i=1}^{n}\sum_{\ell=1}^{p} c_{i\ell}$ and $Comm_{avg} = (1/b_{avg})\sum_{i=1}^{n} w(files(t_i))$. Note that $b_{avg} = (1/p)\sum_{\ell=1}^{p} b_\ell$ and $b_{avg} = (1/c)\sum_{\ell=1}^{c} b_\ell$ denote the average server-to-processor or server-to-cluster bandwidth in the basic and clustered master-slave platforms, respectively. We experimented with the heuristics of five different r values from 10 to 0.1 as $r = 10, 5, 1, 0.2, 0.1$. For each r value, 50 randomly created applications were scheduled by all heuristics. For each scheduling instance, the relative performance of every heuristic was calculated by dividing the turnaround time of the schedule it generates to that of the best schedule. Then, the average of these relative performances for all 50 applications was displayed in the following tables as the performance of the respective heuristic for a specific r ratio.

8.1 Heterogeneous Master-Slave Platform Creation

We used the GridG topology generator [20] for creating a heterogeneous master-slave platform with $p = 32$ processors as follows: We created a Grid topology with 32 hosts and nine routers. One of the routers was randomly selected as the server. The resulting topology contains 82 communication links with bandwidth values varying between 20 Mbit/s and 1 Gbit/s. Each server-to-processor bandwidth value is selected as the bandwidth value of the fastest path from the server, where the slowest link along a path determines the bandwidth value of that path. The clustered master-slave platform is created in a similar way. It contains a total of 48 processors in five clusters, where four clusters contain eight processors each and the remaining cluster contains 16 processors. The bandwidth value of a cluster is computed as the average of the bandwidth values of the processors it contains.

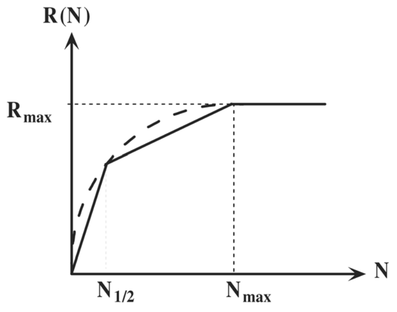


Fig. 11. Piecewise linear approximation for task-execution time estimation.

8.2 Task Execution Time Estimation

We used the Top500 supercomputer list, maintained by Dongarra et al. [12], to estimate the task execution times as follows: We randomly chose our processors from midrank supercomputers, i.e., the ones ranked between the first and second hundred, with sufficient mutual performance variation. As the Top500 list depends on the LINPACK benchmark, we assumed that the individual tasks are instances of the same problem approximately incurring $(2/3)N^3$ floating-point operations for an instance size N as in [12]. The benchmark values R_{max}, N_{max}, and $N_{1/2}$ provided in [12] for each supercomputer were exploited to make realistic approximations for task execution times in a heterogeneous Grid system. Here, R_{max} denotes the maximum processor performance in terms of FLOPS that can be achieved for a task with an instance size $\geq N_{max}$. $N_{1/2}$ represents the instance size for which half of the R_{max} is achieved. Each task has a problem size selected from a uniformly distributed interval. This interval was selected judiciously to achieve a specific r value. So, the performance variation of a task with instance size N can be represented approximately with a piecewise linear function $R(N)$, as shown in Fig. 11. The execution time of a task t_i with instance size N on a processor p_ℓ was estimated as $c_{i\ell} = (2/3)N^3/R_\ell(N)$.

8.3 Results

Table 3 shows the effects of the proposed two-level gain computation scheme and the refinement order of the alternating scheme on the overall scheduling performance. As seen in the table, the two-level gain computation scheme leads to better scheduling performance with the same ordering in the alternating scheme. As expected, the UB-LB ordering leads to better scheduling performance than the LB-UB ordering in the alternating scheme. Comparison of the third and seventh rows, as well as the fourth and eighth rows, shows that adopting the two-level gain computation scheme only in the LB-Refinement stage suffices to achieve the same performance with that of adopting it in both stages. Note that the third row corresponds to the proposed scheme. The proposed iterative-improvement scheduling heuristic will be referred to as IIS here and hereafter.

Table 4 summarizes the results of the experiments conducted to validate the relation between the proposed assignment objective functions and the actual schedule cost, which is the turnaround time of a schedule. The values in the table are derived by using scheduling heuristics

TABLE 3
Effects of the Implementation Choices in the Refinement Phase

Basic master-slave framework									
Parameters									
#	2 level gain		Order	r: Computation-to-communication ratio					
	UB	LB		10	5	1	0.2	0.1	AVG
1	No	No	UB-LB	1.113	1.039	1.016	1.004	1.002	1.035
2	No	No	LB-UB	1.116	1.065	1.020	1.153	1.033	1.077
3	No	Yes	UB-LB	1.002	1.003	1.005	1.004	1.001	1.003
4	No	Yes	LB-UB	1.020	1.013	1.012	1.152	1.034	1.046
5	Yes	No	UB-LB	1.107	1.039	1.016	1.003	1.001	1.033
6	Yes	No	LB-UB	1.106	1.058	1.021	1.154	1.033	1.074
7	Yes	Yes	UB-LB	1.005	1.003	1.005	1.002	1.001	1.003
8	Yes	Yes	LB-UB	1.022	1.013	1.012	1.153	1.033	1.047

UB and LB denote the upper bound and lower bound refinement stages and the order denotes the refinement sequence. The table shows the averages of the relative performances of every implementation choice normalized with respect to the best schedule generated for each scheduling instance.

individually in the initial task assignment phase as follows: For each heuristic used, the amount of decrease achieved in both the *UBTime* and *LBTime* during the refinement phase are normalized with respect to the amount of the resulting decrease in the actual schedule cost. That is, these values display the amount of improvements needed in *UBTime* and *LBTime*, simultaneously to attain one time unit of improvement in the actual schedule cost. Note that performance results are also given for *MinMin* and *Sufferage*, which are not adopted in IIS, in the last two rows of the table. As seen in Table 4, close to one time-unit (between 0.92 and 1.00) of improvements are needed in *LBTime*, which is a rather tight bound, whereas a large variation (between 0.16 and 1.95) can be seen for the improvements needed in *UBTime* which is a loose bound.

Table 5 displays the results of the experiments conducted to justify the use of cheap scheduling heuristics *Communication*, *Computation*, *Advance*, *Duration*, and *Payoff* in the initial task assignment phase instead of the expensive but more successful heuristics *MinMin*, *Sufferage*, and *XSufferage*. In the table, the "No" column represents the relative performances of the expensive heuristics and IIS without refinement. In this case, IIS reduces to selecting the best schedule out of the five schedules generated by the cheap heuristics. The "Yes" column represents the relative

TABLE 4
Effectiveness of the Proposed Assignment Objective Functions

Basic master-slave framework										
Heuristic in first phase	r: Computation-to-communication ratio									
	10		5		1		0.2		0.1	
	UB	LB	UB	LB	UB	LB	UB	LB	UB	LB
Communication	1.879	0.967	1.865	0.956	0.955	0.989	1.001	0.996	0.703	0.996
Computation	1.718	0.928	1.606	0.946	0.852	0.972	0.331	0.992	0.441	0.993
Duration	1.647	0.905	1.503	0.941	0.570	0.983	0.657	0.996	0.868	0.997
Payoff	1.790	0.988	1.728	0.990	1.124	0.995	1.066	1.000	0.746	0.999
Advance	1.470	0.994	1.449	0.996	1.378	0.999	1.460	0.994	0.747	0.975
MinMin	1.759	0.923	1.697	0.947	0.349	0.950	0.266	0.986	0.545	0.985
Sufferage	1.945	0.993	1.951	0.993	1.274	0.993	0.160	0.998	0.309	0.999

The table shows the improvements in the UB and LB stages required to obtain one unit improvement in the execution time.

TABLE 5
Effectiveness of the Refinement

Basic master-slave platform									
Heuristic in first phase	r: Computation-to-communication ratio								
	5			1			0.2		
	Refinement			Refinement			Refinement		
	No	Yes	Ratio	No	Yes	Ratio	No	Yes	Ratio
MinMin	1.109	1.011	0.324	1.033	1.037	0.125	1.026	1.049	0.167
Sufferage	1.000	1.009	0.252	1.006	1.046	0.095	1.016	1.080	0.133
IIS	1.106	1.002	0.328	1.168	1.000	0.254	1.052	1.000	0.224

Clustered master-slave platform									
Heuristic in first phase	r: Computation-to-communication ratio								
	5			1			0.2		
	Refinement			Refinement			Refinement		
	No	Yes	Ratio	No	Yes	Ratio	No	Yes	Ratio
MinMin	1.051	1.109	0.174	1.078	1.176	0.008	1.063	1.065	0.001
Sufferage	1.135	1.144	0.204	1.138	1.249	0.004	1.069	1.073	0.000
XSufferage	1.057	1.124	0.166	1.015	1.109	0.007	1.001	1.004	0.000
IIS	1.135	1.007	0.304	1.243	1.000	0.264	1.041	1.000	0.040

The table shows the averages of the relative performances of every heuristic normalized with respect to the best schedule generated for each scheduling instance.

performances of these heuristics when they are used in the initial task assignment phase of the proposed three-phase scheduling approach. In the table, the refinement ratio is the ratio of the improvement obtained by applying the refinement and execution ordering phases to the initial schedule generated by each heuristic. Note that IIS corresponds to the actual proposed heuristic in this case.

As seen in Table 5, choosing the best result of the cheap heuristics does not suffice to obtain a better performance than a single run of the expensive *MinMin* and *Sufferage* heuristics. However, as also seen in the table, much higher improvement ratios are obtained in the refinement of the cheap heuristics in IIS compared to those of the expensive heuristics. As a result, IIS outperforms the refined version of *MinMin*, *Sufferage*, and *XSufferage* as seen in the "Yes"columns. These experimental findings confirm our rationale behind using the cheap scheduling heuristics in the initial task assignment phase.

Table 6 summarizes the results of the experiments conducted to compare the performance of the proposed IIS heuristic with the existing constructive heuristics. Besides IIS, 36 heuristics given in [15] and all four heuristics given in [8] were implemented. Table 6 displays the relative scheduling performances of the 10 best scheduling heuristics ranked according to the their average performances. The last column of the table also shows the relative runtime performances of these 10 heuristics. For each scheduling instance, the relative runtime performance of every heuristic was calculated by dividing the execution time of the heuristic to that of the fastest heuristic.

As seen in Table 6, the proposed IIS heuristic performs significantly better than all existing heuristics on the average. For example, *Sufferage* and *XSufferage*, which are the second best heuristics for the basic and clustered master-slave platforms, produce 25.1 percent and 16.4 percent worse schedules than IIS on the average, respectively. This relative performance gap is much higher for computa-

TABLE 6
Relative Performances of 10 Best Heuristics

Basic master-slave platform							
Heuristic	r: Computation-to-communication ratio						Exec.
	10	5	1	0.2	0.1	Avg	time
IIS	1.000	1.000	1.000	1.000	1.000	1.000	46.5
Sufferage	1.313	1.350	1.156	1.247	1.191	1.251	606.9
MinMin	1.433	1.496	1.186	1.259	1.140	1.303	655.6
Comp+R	1.457	1.566	1.399	1.454	1.197	1.415	3.9
Comm+S+R	1.402	1.467	1.384	1.489	1.346	1.418	1.3
Comm+S	1.402	1.507	1.389	1.483	1.347	1.426	1.1
Comp	1.490	1.590	1.349	1.427	1.319	1.435	3.6
Advance+S+R	1.666	1.781	1.346	1.281	1.120	1.439	4.6
Comm+R	1.396	1.464	1.397	1.559	1.460	1.455	1.3
Comm	1.402	1.502	1.403	1.562	1.472	1.468	1.0

Clustered master-slave platform							
Heuristic	r: Computation-to-communication ratio						Exec.
	10	5	1	0.2	0.1	Avg	time
IIS	1.000	1.000	1.000	1.000	1.000	1.000	22.4
XSufferage	1.347	1.348	1.117	1.004	1.002	1.164	280.8
MinMin	1.331	1.336	1.187	1.066	1.048	1.193	263.0
Sufferage	1.357	1.448	1.254	1.072	1.049	1.236	263.5
Comp+R	1.531	1.547	1.252	1.010	1.008	1.270	3.6
Comp	1.433	1.479	1.398	1.049	1.018	1.275	3.5
Duration+R	1.480	1.518	1.421	1.217	1.153	1.358	3.7
Duration	1.385	1.480	1.566	1.253	1.164	1.370	3.7
Comm+S	1.582	1.658	1.567	1.254	1.164	1.445	1.0
Comm+S+R	1.585	1.663	1.563	1.254	1.164	1.446	1.1

The table shows the averages of the relative performances of every heuristic normalized with respect to the best/fastest heuristic for each scheduling instance. Comp: Computation, Comm: Communication, S: Shared, and R: Readiness.

tion-intensive applications so that IIS produces at least 30 percent better schedules than all other heuristics for $r = 10$ and 5. In fact, IIS is always the best heuristic for all scheduling instances except the communication-intensive ones in the clustered master-slave platform with $r = 0.2$ and 0.1. That is, IIS achieves the actual relative performance exactly equal to 1 except for those scheduling instances.

The above findings are in concordance with the experimental results given in [15], which state that the scheduling performances of the existing heuristics become far from optimal when the r value increases. Although the experimental framework in this work differs in the generation of the experimental data and calculation of the r value, experimental results in both works can be interpreted as to point out the sensitivity of the computation-intensive applications to the greedy constructive structure of the existing scheduling heuristics.

As seen in Table 6, the performance gap between IIS and existing heuristics decreases in scheduling communication-intensive applications ($r = 0.2$ and 0.1) on the clustered master-slave platform. Although not seen in the table, a similar pattern is also observed in the basic platform for much smaller r values ($r = 0.01$). This common behavior can be attributed to the fact that communication from the master becomes a serious bottleneck for all heuristics with decreasing r. This bottleneck occurs earlier in the clustered platform since the number of file storage units, which can be considered as p in the basic platform, is much smaller in the clustered platform. In fact, the performances of all existing heuristics become very close to each other for these scheduling instances, as also stated in [15].

As seen in the last column of Table 6, IIS is an order of magnitude faster than the successful but slow heuristics [8], whereas it is an order of magnitude slower than the fast heuristics [15]. IIS produces approximately 25-30 percent better schedules while being 13-14 times faster than *MinMin* and *Sufferage* in the basic master-slave platform. Similarly, IIS produces approximately 16-24 percent better schedules while being 11-12 times faster than *MinMin*, *Sufferage*, and *XSufferage* in the clustered master-slave platform.

Fig. 12 displays the dissection of the execution time of the IIS heuristic into phases. For the basic master-slave framework, all phases take comparable time while the refinement phase takes more time than the others. On the other hand, the initial task assignment phase dominates the total execution time for the clustered master-slave framework. These experimental findings are in accordance with the complexity analysis given in Sections 6.4 and 7. Comparing Fig. 12 and Table 6 shows that, while r is changing from 10 to 0.1, the refinement time is correlated with the amount of the performance improvement of IIS with respect to the second best scheduling heuristic. This correlation indicates that more time spent in the refinement phase is likely to incur more improvement in the resulting schedule. This experimental finding also strengthens our claim about the

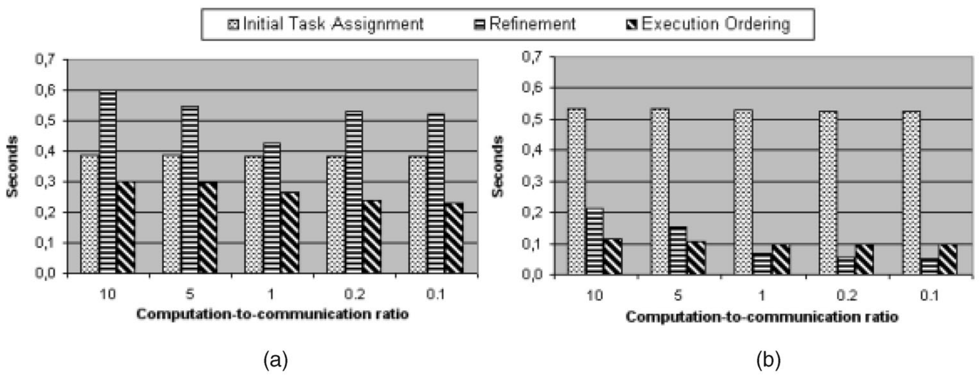


Fig. 12. Execution times of the phases of the IIS heuristic in seconds: (a) basic master-slave platform and (b) clustered master-slave platform.

direct relation between the proposed objective functions and the actual schedule cost.

9 Conclusion

We investigated the problem of scheduling independent but file-sharing tasks on heterogeneous master-slave platforms. We considered the task scheduling problem as involving two consecutive processes: task assignment, which determines the task-to-processor assignments, and execution ordering, which determines the order of inter and intraprocessor task executions. This approach enabled the use of iterative-improvement heuristics effectively and efficiently in the task assignment process by proposing smooth assignment objective functions that are closely related to the cost of a schedule. This refined task-to-processor assignment was then used to generate a better schedule during the execution ordering process. We implemented a scheduling heuristic based on the proposed approach and tested its performance in comparison to the existing constructive heuristics by running large number of experiments on synthetically generated heterogeneous master-slave platforms. Our scheduling heuristic outperformed the existing constructive heuristics in all of the experiments, thus verifying the validity of the proposed approach. The proposed hypergraph-partitioning-like model together with the two objective functions can also be used to map unstructured computations to heterogeneous parallel systems.

With recent advances in optical networking technology, server-to-cluster and intracluster file transfer times are expected to be comparable in the very near future. This advancement will open a new research direction for scheduling in clustered master-slave frameworks since existing and our approaches rely on the assumption that intracluster file transfer times are negligible. A possible adaptation of our approach to this problem might be applying the proposed hypergraph model to the server-to-clusters and intraclusters scheduling problem and subproblems separately in a hierarchical manner.

Acknowledgments

This work is partially supported by the Scientific and Technical Research Council of Turkey (TUBITAK) under project EEEAG-103E028 and by the European Commission FP6 project SEE-GRID with contract no 002356.

References

[1] S. Ali, H.J. Siegel, M. Maheswaran, and D. Hensgen, "Task Execution Time Modeling for Heterogeneous Computing Systems," *Proc. IEEE Ninth Heterogeneous Computing Workshop,* May 2000.

[2] C.J. Alpert, J.H. Huang, and A.B. Kahng, "Multilevel Circuit Partitioning," *Proc. ACM 34th Ann. Conf. Design Automation,* pp. 530-533, 1997.

[3] R. Armstrong, "Investigation of Effect of Different Run Time Distributions on Smartnet Performance," MS thesis, Dept. of Computer Science, Naval Postgraduate School, 1997.

[4] C. Aykanat, A. Pinar, and Ü.V. Çatalyürek, "Permuting Sparse Rectangular Matrices into Block-Diagonal Form," *SIAM J. Scientific Computing,* vol. 25, no. 6, pp. 1860-1879, 2004.

[5] O. Beaumont, A. Legrand, and Y. Robert, "The Master-Slave Paradigm with Heterogeneous Processors," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 9, pp. 897-908, Sept. 2003.

[6] C. Berge, *Hypergraphs.* Amsterdam: North Holland, 1989.

[7] 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, and D. Zagorodnov, "Adaptive Computing on the Grid Using AppLeS," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 4, pp. 369-382, Apr. 2003.

[8] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for Parameter Sweep Applications in Grid Environments," *Proc. Ninth IEEE Heterogeneous Computing Workshop,* pp. 349-363, 2000.

[9] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, "The Apples Parameter Sweep Template: User-Level Middleware for the Grid," *Proc. IEEE/ACM Supercomputing Conf. (SC '00),* p. 60, 2000.

[10] Ü.V. Çatalyürek and C. Aykanat, "Hypergraph-Partitioning Based Decomposition for Parallel Sparse-Matrix Vector Multiplication," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 7, pp. 673-693, July 1999.

[11] Ü.V. Çatalyürek and C. Aykanat, "Hypergraph Model for Mapping Repeated Sparse-Matrix Vector Product Computations onto Multicomputers," *Proc. Second Int'l Conf. High Performance Computing,* pp. 27-30, Dec. 1995.

[12] J.J. Dongarra, H.W. Meuer, and E. Strohmaier, "TOP 500 Supercomputer Sites, 22nd Edition," *Proc. IEEE/ACM Supercomputing Conf. (SC '03),* 2003.

[13] C.M. Fidducia and R.M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. ACM/IEEE 19th Design Automation Conf.,* pp. 175-181, 1982.

[14] "High Performance Schedulers," *The Grid: Blueprint for a New Computing Infrastructure,* I. Foster and C. Kesselman, eds., pp. 279-309,Morgan-Kaufmann, 1999.

[15] A. Giersch, Y. Robert, and F. Vivien, "Scheduling Tasks Sharing Files on Heterogeneous Master-Slave Platforms," *Proc. 12th IEEE Euromico Workshop Parallel Distributed and Network-Based Processing (PDP '04),* 2004.

[16] A. Giersch, Y. Robert, and F. Vivien, "Scheduling Tasks Sharing Files on Heterogeneous Clusters," Technical Report RR-2003-28, LIP, ENS Lyon, France, May 2003.

[17] 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.

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

[19] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout.* Chichester, U.K.: Wiley-Teubner, 1990.

[20] D. Lu and P.A. Dinda, "GridG: Generating Realistic Computational Grids," *ACM SIGMETRICS Performance Evaluation Rev.,* vol. 30, no. 4, pp. 33-40, 2003.

[21] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R. Freund, "Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems," *J. Parallel and Distributed Computing,* vol. 59, no. 2, pp. 107-131, 1999.

[22] L.A. Sanchis, "Multiple-Way Network Partitioning," *IEEE Trans. Computers,* vol. 38, no. 1, pp. 62-81, Jan. 1989.

[23] B. Uçar and C. Aykanat, "Encapsulating Multiple Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiplies," *SIAM J. Scientific Computing,* vol. 25, no. 6, pp. 1837-1859, 2004.

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

Cevdet Aykanat received the BS and MS degrees from the 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 35 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 was recently appointed as a member of IFIP Working Group 10.3 (Concurrent Systems) and the INTAS Council of Scientists.