5

Adapting Iterative-Improvement Heuristics for Scheduling File-Sharing Tasks on Heterogeneous Platforms

Kamer Kaya[1], Bora Uçar[2], and Cevdet Aykanat[3]

[1] Department of Computer Engineering, Bilkent University, Ankara, Turkey
kamer@cs.bilkent.edu.tr
[2] CERFACS, 42 Av. Gaspard Coriolis, Toulouse, Cedex 1, 31057 France
ubora@cerfacs.fr
[3] Department of Computer Engineering, Bilkent University, Ankara, Turkey
aykanat@cs.bilkent.edu.tr

Summary. We consider the problem of scheduling an application on a computing system consisting of heterogeneous processors and one or more file repositories. The application consists of a large number of file-sharing, otherwise independent tasks. The files initially reside on the repositories. The interconnection network is heterogeneous. We focus on two disjoint problem cases. In the first case, there is only one file repository which is called as the master processor. In the second case, there are two or more repositories, each holding a distinct set of files. The problem is to assign the tasks to the processors, to schedule the file transfers from the repositories, and to order the executions of tasks on each processor in such a way that the turnaround time is minimized.

This chapter surveys several solution techniques; but the stress is on our two recent works [22, 23]. At the first glance, iterative-improvement-based heuristics do not seem to be suitable for the aforementioned scheduling problems. This is because their immediate application suggests iteratively improving a complete schedule, and hence building and exploring a complex neighborhood around the current schedule. Such complex neighborhood structures usually render the heuristics time-consuming and make them stuck to a part of the search space. However, in both of the our recent works, we show that these issues can be solved by using a three-phase approach: initial task assignment, refinement, and execution ordering. The main thrust of these two works is that iterative-improve-based heuristics can efficiently deliver effective solutions, implying that iterative-improve-based heuristics can provide highly competitive solutions to the similar scheduling problems.

Keywords: Scheduling File-Sharing Tasks, Iterative-Improvement Heuristics, Heterogeneous Platforms, Neighborhood exploration.

5.1 Introduction

Task scheduling in heterogeneous systems is an important problem for today's computational Grid environments [9], as heterogeneous systems become more

F. Xhafa, A. Abraham (Eds.): Meta. for Sched. in Distri. Comp. Envi., SCI 146, pp. 121–151, 2008.
springerlink.com

and more prevalent. There are important Grid applications [10] which are typically composed of a large number of independent but file-sharing tasks. Therefore, the problem of scheduling a large number of independent but file-sharing tasks on heterogeneous platforms has recently attracted much attention, see for example [12, 13, 17, 18, 19, 20, 22, 23, 25] and the references therein. By file sharing, we mean that a file may be requested by a number of tasks. The computing system consists of heterogeneous processors and one or more repositories that store input files. The files are not replicated, i.e., if there are two or more repositories, each one stores a distinct set of files. The repositories are decoupled from the processors. The processors and the repositories are connected through a heterogeneous interconnection network. The problem 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.

Once the tasks are assigned to the processors, the files should be transferred from the repositories to the processors. A task execution can start only after its input files are delivered to the respective processor. Once a file is transferred to a processor, it can be used by all tasks assigned to the same processor without any additional cost. Since the interconnection network is heterogeneous, the costs of transferring a certain file between different source and destination pairs are not necessarily equal. We assume the one-port communication model in which 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.

Task scheduling for heterogeneous environments is harder than task scheduling for homogeneous ones, since in a heterogeneous environment, different tasks which need the same files might have different favorite processors. Therefore, it may not be feasible to assign them to the same processor on the grounds of efficient 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 while decreasing the file transfer amount.

The application and computing models and the objective function which characterize the scheduling problems at hand are introduced formally in Sect. 5.2. In Sect. 5.3, we discuss the single repository case and review the heuristics from the works [12,13,17,20,22]. Then, in Sect. 5.4, we discuss the multiple repository case and review the heuristics from the works [18, 19, 23, 25].

5.2 Framework

5.2.1 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 files(t); these files should be delivered to the processor that will execute the task t. We extend the operator

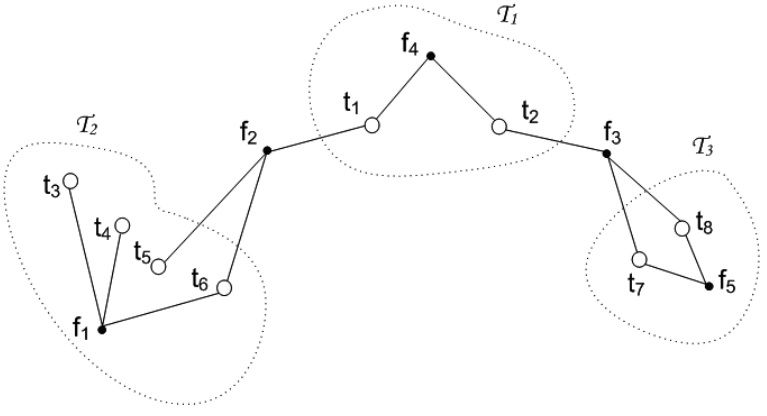

Fig. 5.1. Hypergraph model $\mathcal{H}_{\mathcal{A}} = (\mathcal{T}, \mathcal{F})$ for an application with a set of 8 tasks $\mathcal{T} = \{1, 2, \ldots, 8\}$ and a set of 5 files $\mathcal{F} = \{1, 2, \ldots, 5\}$. Vertices are shown with empty circles and correspond to the tasks; nets are shown 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.

files$(\cdot)$ to a subset of tasks $\mathcal{S} \subseteq \mathcal{T}$ such that files$(\mathcal{S}) = \bigcup_{t \in \mathcal{S}}$ 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., $|\mathcal{A}| = \sum_{t \in \mathcal{T}} |\text{files}(t)|$.

It seems natural to use a hypergraph $\mathcal{H}_{\mathcal{A}} = (\mathcal{T}, \mathcal{F})$ to model the application $\mathcal{A} = (\mathcal{T}, \mathcal{F})$, see [22, 23]. 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. 5.1 contains an example hypergraph model.

5.2.2 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 this chapter 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 distinct repositories r and s. We use 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 curves encompassing the vertices in Fig. 5.1 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 $\mathcal{T}_1$.

The authors of [12,13,17,20,22,23] assume the one-port communication 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 [5,7,30] and it is prevalent in the scheduling for Grid computing literature, however, alternatives exist (see [4,11]). 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 other tasks. The file transfer operations take place only between a repository and a processor. The congestion in the communication network during the file transfers is ignored. 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 [17,20] when $R = 1$. Such complete graph models are used to abstract wide-area networking infrastructures [11]. The network heterogeneity is modeled 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. The heuristics in the literature generally use

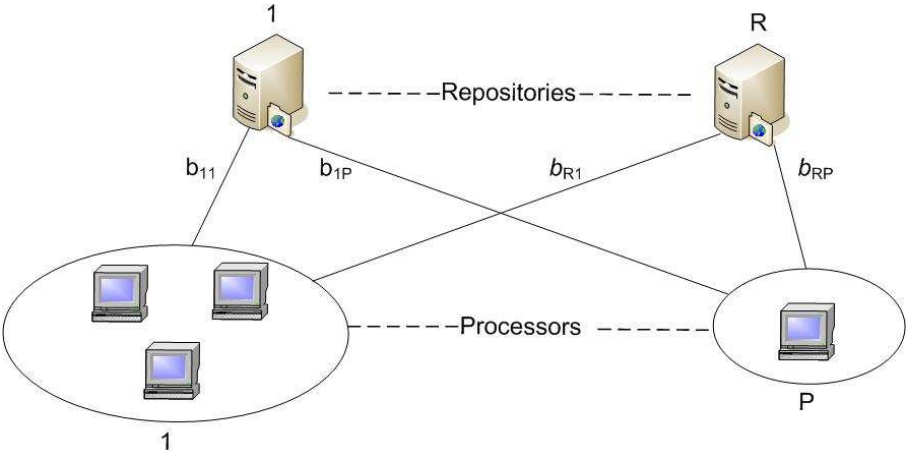

Fig. 5.2. Computing system

the linear cost model [6,11] for file transfers, i.e., transferring the file f from the repository r to the processor p takes $w(f)/b_{rp}$ time units. Fig. 5.2 displays the essential properties of the computing system described.

The task and processor heterogeneity are modeled 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 the other tasks. The *inconsistent* ETC matrices have no such special structure. In general, the inconsistent ETC matrices are more realistic for heterogeneous computing environments, since they can model a variety of computing systems and applications that arise in Grid environments.

5.2.3 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. Scheduling file-sharing tasks on heterogeneous systems with $R = 1$ repository is NP complete [17]. The NP completeness of the multiple repositories case, i.e., $R > 1$ case, follows easily.

5.3 Scheduling File-Sharing Tasks with Single Repository

In this section, we survey the heuristics proposed for the scheduling problem on heterogeneous systems with $R = 1$ repository, e.g., heterogeneous master-slave environments where the master processor stores all files. This framework has been studied in [10,12,13,17,20,22] for adaptive scheduling of parameter-sweep-like applications in Grid environments. Such applications arise in the *Application Level Scheduling* (AppLeS) project [10].

For the single-repository case, Casanova et al. [12,13] extend three heuristics, namely *MinMin*, *MaxMin* and *Sufferage*, which are initially proposed in [28] for scheduling independent tasks. They use these extended heuristics in the *AppLeS Parameter Sweep Template* (APST) project [10]. They also proposed a new heuristic *XSufferage* exclusively for APST. After this work, Giersch et al. [17,20] proposed several different heuristics which reduce the time complexity while preserving the quality of schedules.

The heuristics in [12,13,17,20] are based on the greedy choices that depend on the momentary completion time values of tasks. Kaya and Aykanat claim that this greedy decision criterion cannot use the file sharing information effectively,

since the completion time values are not sufficient to extract the global view of the interaction among the tasks [22]. Instead of a direct construction of schedules, Kaya and Aykanat propose a three-phase scheduling approach which involves initial task assignment, refinement and execution ordering phases.

Kaya and Aykanat argue in [22] that an iterative-improvement-based method which uses task reassignments to improve the actual length of the schedule, i.e., the turnaround time, have a global perturbation on the given schedule. However, the effectiveness and efficiency of the iterative-improvement-based heuristics, which are widely and successfully used for hypergraph partitioning, depend on the perturbations being local [2]. When the perturbations are local, the objective functions become *smooth* over the search space, and the iterative-improvement-based heuristics explore a relatively large part of the search space in relatively small time.

In the refinement phase of the proposed three phase approach, Kaya and Aykanat use two novel smooth objective functions in a hypergraph-partitioning-like formulation to refine task-to-processor assignments. The first objective function represents an upper bound while the second one represents a lower bound for the turnaround time of a schedule by considering only the task-to-processor assignments. The first and the second objective functions relate, respectively, to a pessimistic and an optimistic view of the execution time of an application. In the rest of this section, we will investigate the heuristics in detail.

The notation described in Sect. 5.2 is slightly modified for the master-slave case. In this section, we will omit the notation for the repositories since in this framework there is a single repository. As an example, the bandwidth of a processor p will be denoted as b_p instead of b_{rp}. Similarly, for a file f the notation store(f) is not used.

5.3.1 Greedy Constructive Scheduling Heuristics

Algorithm 5.1 shows the structure of the heuristics used by Casanova et al. [12, 13]. In Alg. 5.1, the completion time $CT(t, p)$ of task t on processor p is computed

Algorithm 5.1. Structure of heuristics by Casanova et al. [12, 13]

```
1: while there remains a task to schedule do
2:    for each unscheduled task t do
3:       for each processor p do
4:          Evaluate completion time CT(t,p) of t on p
5:       end for
6:       Evaluate schedule cost g(CT(t,p_1),...,CT(t,p_P)) for t
7:    end for
8:    Choose task t_b with the "best" schedule cost
9:    Pick the best processor p_b for t_b with min. completion time
10:   Schedule t_b on p_b and its file transfers
11:   Mark t_b as scheduled
12: end while
```

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 g and the meaning of the "best" characterize these heuristics as shown in Table 5.1. As seen in Alg. 5.1, computing the completion times for all task-processor pairs takes $O(TP + P|\mathcal{A}|)$ time for each scheduling decision. As this decision is made once for each task, the total time complexity of these heuristics is $O(T^2P + TP|\mathcal{A}|)$.

Table 5.1. Definitions for the heuristics proposed by Casanova et al. [12, 13]

Heuristics	Function g	best
MinMin	minimum of all $CT(t,p)$ values	minimum
MaxMin	minimum of all $CT(t,p)$ values	maximum
Sufferage	difference between 2nd minimum and minimum of all $CT(t,p)$ values	maximum

After Casanova et al. [12, 13], Giersch et al. [17, 20] proposed several different heuristics. These heuristics have better time complexity and their solution quality is comparable with those of the previous heuristics. Algorithm 5.2 shows the structure of these heuristics. Table 5.2 displays the objective functions proposed by Giersch et al. [17, 20] for a task-processor pair (t, p) based on the computation time $\text{Comp}(t, p) = x_{tp}$ and communication time $\text{Comm}(t, p) = w(\text{files}(t))/b_p$ values of the task t when it is executed on the processor p. The additional policies

Algorithm 5.2. Structure of heuristics by Giersch et al. [17, 20]

```
1: for each processor p do
2:    for each task t do
3:       Evaluate OBJECTIVE(t, p)
4:    end for
5:    Build the list L(p) of the tasks sorted according
      according to the value of OBJECTIVE(t, p)
6: end for
7: while there remains a task to schedule do
8:    for each processor p do
9:       Let t be the first unscheduled task in L(p)
10:      Evaluate completion time CT(t, p) of t at p
11:   end for
12:   Pick a task-processor pair (t_b,p_b) with
      minimum completion time
13:   Schedule t_b on p_b and its file transfers
14:   Mark t_b as scheduled
15: end while
```

Table 5.2. Definitions for the heuristics proposed by Giersch et al. [17, 20]

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

Additional Policy	Explanation
Readiness	Selects a ready task for a processor if one exists. A task is called ready for processor p if the transfers of all input files of the task to p are previously scheduled.
Shared	While calculating $w(\text{files}(t))$, 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.

readiness, *shared* and *locality* proposed by Giersch et al. [17, 20] are also explained in Table 5.2. As seen in Alg. 5.2, the heuristics construct a task list for each processor. These lists are 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., $w(\text{files}(t))$ values, in $\Theta(|\mathcal{A}|)$ time in a preprocessing step. In this way, the objective value computations for all task-processor pairs take $\Theta(TP + |\mathcal{A}|)$ time, so the construction of all sorted lists takes $O(TP \log T + |\mathcal{A}|)$ time. The while loop for scheduling tasks in step 5 takes $O(TP|\mathcal{A}|)$ time. Therefore, the overall time complexity becomes $O(TP \log T + TP|\mathcal{A}|)$.

Flaws of the Greedy Heuristics

The task-processor pair selection according to the momentary completion time values is the greedy decision criterion commonly used in all existing constructive heuristics. Kaya and Aykanat show that this criterion suffers from ineffective use of information about file sharing among the tasks [22]. This flaw is likely to increase with the increasing amount of file sharing and can incur extra file transfers in the resulting schedule. Since the amount of the total file transfers from the server is a bottleneck under the one-port communication model, extra

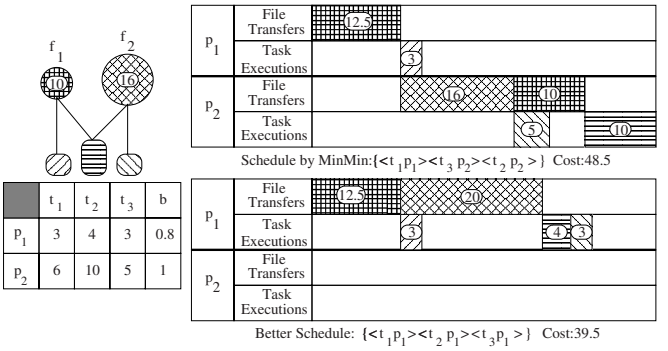

Fig. 5.3. A flaw of the greedy constructive approach for communication-intensive tasks

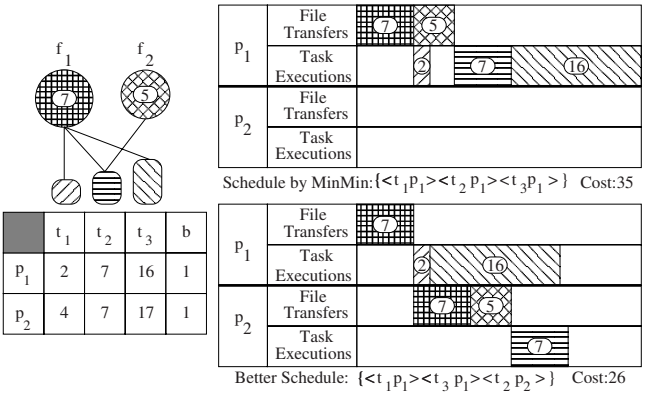

Fig. 5.4. Another flaw of the greedy constructive approach

file transfers can deteriorate the quality of the schedule. This effect is amplified for communication-intensive tasks where the cost of file transfers is considerably higher than the cost of task executions.

Fig. 5.3 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. 5.3.

Although extra file transfers constitute crucial bottleneck, it is stated in [22] that 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 being already favorable, they have lots of file transfers already scheduled.

Fig. 5.4 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.

5.3.2 Iterative-Improvement-Based Scheduling Heuristics

In [22], Kaya and Aykanat propose an iterative-improvement-based heuristic for scheduling file-sharing tasks on a heterogeneous framework with a single repository. They propose a three-phase scheduling approach which involves initial task assignment, refinement and execution ordering phases. For the refinement phase, they model the target application as a hypergraph and with a hypergraph-partitioning-like formulation, they propose 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.

Before a detailed analysis of the heuristics in [22], we first give the background material on hypergraph partitioning and iterative-improvement heuristics which are exploited in the scheduling approach.

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 [8]. Every net n in $\mathcal{N}$ is a subset of vertices, i.e. $n \subseteq \mathcal{V}$. The vertices in a net n are called its pins. The set of nets that contain vertex v is denoted as $nets(v)$. 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_n of a net n is the set of parts that n connects and the connectivity $\lambda_n = |\Lambda_n|$ of n 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 (HP) 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 hypergraph partitioning [26]. The partitioning objective in this metric is the minimization of CutSize(Π) which is given as:

$$\text{CutSize}(\Pi) = \sum_{n \in \mathcal{N}} w(n)(\lambda_n - 1), \tag{5.1}$$

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

$$(W_{max} - W_{avg})/W_{avg} \leq \epsilon, \tag{5.2}$$

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.

Iterative-Improvement Heuristics

The refinement heuristics proposed by Kaya et al. [22, 23] are based on the iterative-improvement heuristics introduced by Kernighan-Lin (KL) [24] and Fidducia-Mattheyses (FM) [16] for graph/hypergraph partitioning. Both KL and FM are move-based approaches with the neighborhood operator of swapping a pair of vertices between parts and shifting a vertex from one part to another, respectively. These heuristics have been widely used for graph/hypergraph partitioning by the VLSI [26] and scientific computing [3,14,15,21,32] communities because of their effectiveness with good-quality results and efficiency with short run times.

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 are proposed for the generalization of the approach to the K-way refinement [31].

Iterative-Improvement-Based Refinement Approach

Both 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, Kaya and Aykanat [22] consider the task scheduling problem as involving two consecutive processes: task assignment process which determines the task-to-processor assignment, and execution-ordering process which determines the order of inter- and intra-processor 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 execution-ordering process.

HP Models for Task Assignment in Heterogeneous Environments:

Kaya and Aykanat use the hypergraph model $\mathcal{H}_A = (\mathcal{T}, \mathcal{F})$ described in Sect. 5.2.1 to represent the interaction among the tasks of the target application $\mathcal{A} = (\mathcal{T}, \mathcal{F})$. Recall that 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 the heuristics using the problem-specific notation of Sect. 5.2 instead of hypergraph-specific notation, as much as possible, for clarity of presentation. For example, we will use files(t) instead of $nets(t)$. The size of a file f is the weight of the corresponding net. Recall also from Sect. 5.2.2 that a P-way vertex partition $\Pi = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_P\}$ of $\mathcal{H}_A$ is decoded as inducing a task-to-processor assignment for a target schedule. That is, all tasks in a part $\mathcal{T}_p$ will be executed by processor p in the target schedule.

Successful hypergraph partitioning formulations have been recently proposed for solving the task-to-processor assignment problem arising in the parallelization of several applications on homogeneous platforms [3,14,15,32]. 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 (5.1) and the load balancing constraint given in (5.2) 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, Kaya and Aykanat 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 with connectivity λ_n in Π. Let f_n be the corresponding file for n. is clear that $\lambda_n - 1$ denotes the number of additional transfers of file f_n incurred by Π. Hence $w(f_n)(\lambda_n - 1)$ represents the additional transfer volume, whereas $w(f_n)\lambda_n$ denotes the total transfer volume for file f_n. 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:

$$\text{CommVol}(\Pi) = \sum_{f_n \in \mathcal{F}} w(f_n)\lambda_n. \tag{5.3}$$

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

Equation (5.2) can also be used to represent the total file transfer time if the network is homogeneous by normalizing file sizes with respect to the bandwidth value. That is, minimization of the total file transfer volume and the total file transfer time are 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_n of a net n in which different parts connected by n make equal contribution to λ_n. Since we want to formulate the total file transfer time as the real communication cost and bandwidth values of the links are different, Kaya and Aykanat define a *heterogeneous connectivity* λ'_f of a file f as:

$$\lambda'_f = \sum_{p \in \Lambda_f} \frac{1}{b_p}, \tag{5.4}$$

where Λ_f denotes the set of processors that have at least one task needing f as input. Then the total communication time, i.e., the total file transfer time, for the single-repository case can be defined as:

$$\text{CommTime}(\Pi) = \sum_{f_k \in \mathcal{F}} w(f_k)\lambda'_k. \tag{5.5}$$

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,

$$\text{CompTime}(\Pi) = \max_p \left(\sum_{t \in \mathcal{T}_p} x_{tp} \right). \tag{5.6}$$

Since the assignment Π is clear from the context, we drop Π while referring to CompTime and CommTime in the following text. 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 (5.2) – can be considered as a lower bound for CompTime if a vertex weight represents a computational cost. Similarly, W_{max} can be considered as CompTime which is the exact parallel computational cost of the partition. Therefore in homogeneous environments, the load balancing constraint given in (5.2) can be used for minimizing CompTime. 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. Therefore, CompTime should be explicitly included in the assignment objective function as well as CommTime.

By using CompTime and CommTime, Kaya and Aykanat propose two novel 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. It is 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

$$\text{UBTime} = \text{CommTime} + \text{CompTime}. \tag{5.7}$$

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 Sect. 5.2, a processor can execute a task while that or another processor is transferring a file from the server, i.e., 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:

$$\text{LBTime} = \max\{\text{CommTime}, \text{CompTime}\}. \tag{5.8}$$

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 non-overlapping cases at the very beginning and the 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 end with a task execution on the 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 non-overlapping intervals are 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. The resulting task-to-processor assignments can be used to obtain schedules with better turnaround times. Instead of one objective as in the hypergraph partitioning problem, we have two assignment objectives and there are various options to improve them. The details of the iterative-improvement-based approach are given in the following subsection.

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, Kaya and Aykanat propose to use an alternating refinement scheme in which refinement according to one objective function follows the refinement according to the other one in a repeated pattern. The refinement of a task-to-processor assignment Π according to UBTime or LBTime is referred to here as *UB-Refinement* or *LB-Refinement* stage, respectively.

Kaya and Aykanat state that 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, the authors propose to modify the move selection policy of 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 [22], the proposed two-level gain computation scheme is used in the LB-Refinement stage. The rationale behind this decision is explained as follows: First, the variations in the task-move gains are expected to be larger in UBTime compared to LBTime. Second, UBTime is a relatively loose bound compared to LBTime. Therefore, 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, they also recommend to start the alternating refinement sequence with UB-Refinement stage.

In [22], both 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. Algorithms 5.3 and 5.4 show the general structures of UB- and LB-Refinement stages, respectively. In these

Algorithm 5.3. 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 in this random order do
4:       leaveGain ← UB-ComputeLeaveGain(t)
5:       if leaveGain > 0 then
6:          p_b ← UB-SelectBestMove(t, leaveGain)
7:          if p_b is not equal to Map(t) then
8:             UpdateGlobalData(t, p_b)
9:             Map(t) ← p_b
10:         end if
11:      end if
12:   end for
13: end while
```

Algorithm 5.4. 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 in this random order do
4:       {commLeaveGain, compLeaveGain} ←textbfLB-ComputeLeaveGain(t)
5:       if (CommCost(Π) > CompCost(Π) and commLeaveGain > 0) or
            (CompCost(Π) > CommCost(Π) and compLeaveGain > 0)  then
6:          {p_b, bestCommGain, bestCompGain} ←
               LB-SelectBestMove(t, commLeaveGain, compLeaveGain)
7:          if p_b is not equal to Map(t) then
8:             UpdateGlobalData(t, p_b)
9:             CommCost(Π) ← CommCost(Π) − bestCommGain
10:            CompCost(Π) ← CompCost(Π) − bestCompGain
11:            Map(t) ← p_b
12:         end if
13:      end if
14:   end for
15: end while
```

figures, $Map(t)$ denotes the processor to which task t is currently assigned. For a more detailed structure of the refinement phase, we refer the reader to [22].

The Three-Phase Approach

In the first phase, initial task-to-processor assignments are derived from the schedules created by some of the existing constructive scheduling heuristics. Kaya and Aykanat 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. They use the heuristics proposed by Giersch et al. [17,20] because of their short execution times. The additional policies are not used, but all of the five heuristics, each

having a different objective function, are used since their relative performances vary with the characteristics of applications, e.g., with the number of tasks and files, the average execution time of the tasks, and the average transfer time of the files. 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 the end, the best schedule in terms of the turnaround time is taken as the schedule for the target application.

After the initial task assignment phase, these task assignments are refined with respect to the UBTime(Π) and LBTime(Π), the two proposed objective functions. The authors state 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. 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, a constant number of alternating sequences of UB- and LB-Refinement stages is allowed in the implementation.

In the execution ordering phase, each task-to-processor assignment Π obtained in the refinement phase is preserved while determining the inter- and intra-processor ordering of the task executions. Note that CommTime, CompTime and hence the improved values of both objective functions remain the same as determined in the refinement phase. The structure of the execution ordering heuristic is similar to the scheduling heuristics proposed by Giersch et al. [17, 20]. However, the 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 is run five times by using each one of the five objective functions proposed by Giersch et al. [17, 20] and the best schedule is selected for this Π.

We omit the details of the subroutines used in LB- and UB-Refinement stages. Slightly different versions of some of them will be explained in detail for the general framework with multiple repositories. The time complexity of the iterative-improvement-based scheduling heuristic is $O(TP\log T + TP|\mathcal{A}|)$. A detailed explanation of the heuristics and complexity analysis can be found in [22].

Experimental Analysis

Kaya and Aykanat give various experimental results for the assessment of the proposed iterative-improvement-based approach. To make the section self-contained, we give the details of the experimental framework in [22] and restate some important results to show the effectiveness of the proposed heuristic. A detailed and complete list of the experiments conducted to analyze the performance of the heuristics can be found in [22].

Kaya and Aykanat demonstrate the performance of the proposed heuristic in comparison with the existing constructive heuristics. They simulate a total of 250 applications, each consisting of $T = 2000$ tasks and $F = 2000$ 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

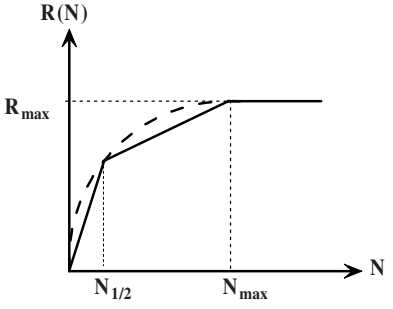

Fig. 5.5. Piecewise linear approximation for task-execution time estimation

200 Gbytes. The experiments vary with the computation-to-communication ratio $\rho = Comp_{avg}/Comm_{avg}$ of the target application, where $Comp_{avg} = (1/P)\sum_{t\in\mathcal{T}}\sum_{p\in\mathcal{P}} x_{tp}$, and $Comm_{avg} = (1/b_{avg})\sum_{i=1}^{n} w(files(t_i))$. Note that b_{avg}=$(1/P)\sum_{p\in\mathcal{P}} b_p$ denote the average server-to-processor bandwidth. They show results with five different ratios ρ = 10.0, 5.0, 1.0, 0.2, and 0.1, where for each ρ value there are 50 randomly created applications; thus totaling 250 applications. These choices of ρ characterize a range of applications containing including computation intensive ($\rho = 10$) and communication intensive ($\rho = 0.1$) ones.

Kaya and Aykanat use the GridG topology generator [27] for creating a heterogeneous master-slave platform with P=32 processors. The network contains communication links with bandwidth values varying between 20 Mbit/s and 1 Gbit/s.

The Top500 supercomputer list maintained by Dongarra et al. [29] is used to generate the task execution times. Since the Top500 list depends on the LINPACK benchmark, the individual tasks are instances of the same problem approximately incurring $(2/3)N^3$ floating point operations for an instance size N. The benchmark values R_{max}, N_{max} and $N_{1/2}$, provided in [29] for each supercomputer, are used to make realistic approximations (inconsistent ETC matrices) 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 greater than or equal to N_{max}. Here, $N_{1/2}$ represents the instance size for which half of the R_{max} is achieved. For specific ρ value, the instance sizes for the tasks are uniformly distributed on an interval which is selected judiciously to achieve ρ. Therefore, 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. 5.5. The execution time of a task t with instance size N on a processor p is estimated as x_{tp}=$(2/3)N^3/R_p(N)$.

Table 5.3 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 individually in the initial task assignment phase as follows: For each heuristic used, the amount of decrease achieved in both 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 5.3, close to one time-unit (between 0.91 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 5.3. Effectiveness of the objective functions

Heuristic in	Min		Max		Avg	
the first phase	UB	LB	UB	LB	UB	LB
Communication	0.703	0.955	1.879	0.996	1.281	0.980
Computation	0.331	0.928	1.718	0.993	0.989	0.966
Duration	0.570	0.905	1.647	0.997	1.049	0.964
Payoff	0.746	0.988	1.790	1.000	1.291	0.994
Advance	0.747	0.975	1.470	0.999	1.378	0.992
MinMin	0.266	0.923	1.759	0.986	0.923	0.958
Sufferage	0.160	0.993	1.951	0.999	1.128	0.995

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

Table 5.4. Relative performances of the heuristics for the single-repository case

Heuristic	Cost	Execution Time
Iterative-Improvement-Based Heu.	1.000	46.5
Sufferage	1.251	606.9
MinMin	1.303	655.6
Computation+Readiness	1.415	3.9
Communication+Shared+Readiness	1.418	1.3
Computation+Shared	1.426	1.1
Computation	1.435	3.6
Advance+Shared+Readiness	1.439	4.6
Communication+Readiness	1.455	1.3
Communication	1.468	1.0

Table shows the averages of the relative performances of good heuristics normalized with respect to the best/fastest heuristic for each scheduling instance.

Table 5.4 summarizes the results of the experiments conducted to compare the performance of the proposed iterative-improvement-based approach with the best greedy constructive heuristics. The last column of the table also shows the relative runtime performances of these heuristics. For each scheduling instance, the relative runtime performance of every heuristic is calculated by dividing the execution time of the heuristic to that of the fastest heuristic. As seen in Table 5.4, the iterative-improvement-based heuristic performs significantly better than all existing heuristics on the average. For example, *Sufferage*, which is the second best heuristics for the single-repository case, produces 25.1% worse schedules than the iterative-improvement-based heuristic on the average.

5.3.3 An Extension: Clustered Platform

In [12, 20, 22], a slightly different version of the basic platform, a clustered platform, is also considered as the target computing environment. The clustered platform also has a single-repository but 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. 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 intra-cluster communication costs due to the local file transfers from a file storage unit are not considered, because intra-cluster file transfers are assumed to be much faster than the file transfers from the server.

Table 5.5. Relative performances of the heuristics for the single-repository case: clustered platform

Heuristic	Cost	Execution Time
Iterative-Improvement-Based Heu.	1.000	22.4
XSufferage	1.164	280.8
MinMin	1.193	263.0
Sufferage	1.236	263.5
Computation+Readiness	1.270	3.6
Computation	1.275	3.5
Duration+Readiness	1.358	3.7
Duration	1.370	3.7
Communication+Shared	1.445	1.0
Communication+Shared+Readiness	1.446	1.1

Table shows the averages of the relative performances of every heuristic normalized with respect to the best/fastest heuristic for each scheduling instance.

The greedy constructive heuristics [12, 13, 17, 20] and the iterative-improvement-based heuristic [22] can be easily extended for the clustered platform. In addition to the heuristics given in Table 5.1, Casanova et al. [12] also propose a new heuristic called *XSufferage* for the clustered master-slave platforms. Unlike other three scheduling heuristics, *XSufferage* computes cluster-based minimum completion times for each task t from $CT(t, p)$ values. The function g is defined as the difference between the second minimum and the minimum of these minimum completion times and "best" is defined as maximum. For the case of the heuristics by Giersch et al. [17, 20], to adapt the readiness policy, a task is called ready for a cluster if all of its input files are available at that cluster. Similarly for adapting the locality policy, 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. Experiments in [22] show that the iterative-improvement-based approach performs better than all other heuristics. The results are summarized in Table 5.5.

5.4 Scheduling with Multiple Repositories

For the multiple-repository case, Giersch et al. [18, 19] 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. This system is slightly different from the framework given in Sect. 5.2. In [18, 19], files can be replicated and they are initially assumed to be stored at one or more repositories. In addition to the objectives stated above for the single-repository case, the scheduler has to decide how to route the files from repositories to other servers. The paper [18] 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 [17, 20]. The structure of the extended *MinMin*, *MaxMin*, and *Sufferage* and the heuristics by Giersch et al. [17, 20] is similar to Algs. 5.1 and 5.2. We refer the reader to [18, 19, 23] for a detailed explanation and analysis of the extended heuristics.

Khanna et al. [25] 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 [25] 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.

Kaya et al. [23] extend the approach in [22] for the multi-repository case and propose a similar three phase heuristic for scheduling file-sharing tasks on a heterogeneous network with multiple data repositories. They state that, the objective functions given in Sect. 5.3.2 cannot be used for the general framework because of the existence of distributed repositories. We will give the details of the new objective functions in this section. Kaya et al. also implement the *MinMin*, *MaxMin*, and *Sufferage* heuristics [12, 13] and compare the performances of the greedy constructive and iterative-improvement-based approach.

5.4.1 Iterative-Improvement-Based Scheduling Heuristics

Since we are dealing with heterogeneous environments, existing hypergraph partitioning techniques and iterative-improvement-based approaches that are used by Khanna et al. [25] are not applicable. Therefore, Kaya et al. adopt the techniques proposed in [22] and reviewed in the previous section. The objective functions proposed in [22] cannot be used when the files are stored in multiple repositories. Hence, new smooth objective functions are required to design iterative-improvement-based heuristics on heterogeneous environments with multiple repositories. Here, we give the details of the heuristics proposed in [23].

Iterative-Improvement-Based Refinement Approach

Objective Functions for Scheduling with Multiple Repositories

In an attempt to obtain bounds on the turnaround time, Kaya et al. [23] make the following observations. The computational cost, CompTime, for the single-repository case given in (5.6) is applicable as is in the multiple repositories case. However, the communication cost, CommTime, for the single repository case given in (5.5) is not applicable to the multiple repository case. Kaya et al. identify two other cost components that are associated with the turnaround time and that can be used instead of the CommTime. These are:

- 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. 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

$$\text{Upload}(f) = w(f) \sum_{p \in \Lambda_f} \frac{1}{b_{rp}} \; . \tag{5.9}$$

For each repository r, the total upload time U_r is defined as the summation of $\text{Upload}(f)$ costs over all files stored in r, i.e.,

$$U_r = \sum_{f \in \mathcal{F}(r)} \text{Upload}(f) \; . \tag{5.10}$$

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

$$\text{UploadTime} = \max_r \{U_r\} \; . \tag{5.11}$$

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

$$\text{Download}(f, p) = \frac{w(f)}{b_{\text{store}(f),p}} \; . \tag{5.12}$$

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

$$D_p = \sum_{f \in \text{files}(\mathcal{T}_p)} \text{Download}(f, p) \; . \tag{5.13}$$

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

$$\text{DownloadTime} = \max_p \{D_p\} \; . \tag{5.14}$$

Although the three cost components given in (5.6), (5.11), and (5.14) do not represent the turnaround time, they are closely related to it. By using these components, Kaya et al. 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

$$\text{LBTimeEX} = \max \{\text{CompTime}, \text{UploadTime}, \text{DownloadTime}\} \; . \tag{5.15}$$

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

$$\text{UBTimeEX} = \sum_{f \in \mathcal{F}} \text{Upload}(f) + \text{CompTime} \; . \tag{5.16}$$

Kaya et al. state that 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 and it is hard to define a tighter upper bound that is smooth over the search space generated by task reassignments. Therefore, they define an objective function which is estimated to be an upper bound. By assuming concurrent transfers, they obtain

$$\text{EstUBTime} = \max\{\text{UploadTime}, \text{DownloadTime}\} + \text{CompTime} \; , \tag{5.17}$$

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.

Structure of the Refinement Heuristics

Similar to [22], for the multiple-repository case, we have two different objective functions, LBTimeEX and EstUBTime. As in [22], Kaya et al. [23] choose a similar approach and use an FM [16] based refinement heuristic to close the gap between these two bounds while minimizing both of them. For this purpose, they use an alternating refinement scheme in which first LBTimeEX 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, Kaya et al. use the two-level gain approach proposed in [22] which modifies the gain concept as described in Sect. 5.3.2. They adopt this modification in improving the LBTimeEX as the primary objective and refine EstUBTime without the two-level gain approach. Similar to [22], the authors state that this latter scheme gives more freedom in EstUBTime refinement and provides the future LBTimeEX refinements with a larger search space to explore.

The objective functions LBTimeEX 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 transferring 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.

Algorithm 5.5 displays the LB-RefinementEX 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., LBTimeEX $= C_1$. 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 (5.17). 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 LBTimeEX 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.

Algorithm 5.5. LB-RefinementEX(Π)

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

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

$$g_{comp}(t, p, q) = \min \left\{ \begin{array}{c} x_{tp} \\ X_p - X(2) \\ X_p - (X_q + x_{tq}) \end{array} \right\} , \tag{5.18}$$

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

$$g_{download}(t,p,q) = \min \left\{ \begin{array}{c} \sum\limits_{f \in \text{critical}(\text{files}(t),p)} \frac{w(f)}{b_{\text{store}(f),p}} \\ D_p - D(2) \\ D_p - \left(D_q + \sum\limits_{f \in \text{notNeed}(\text{files}(t),q)} \frac{w(f)}{b_{\text{store}(f),q}} \right) \end{array} \right\}, \quad (5.19)$$

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$(\text{files}(t), p)$ 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 notNeed$(\text{files}(t), q)$ 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 notNeed$(\text{files}(t), q)$ 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

$$g_{upload}(t,p,q) = U(1) - \max_r \left\{ \begin{array}{ll} U_r & - \sum\limits_{f \in \text{critical}(\text{files}(t) \cap \mathcal{F}(r),p)} \frac{w(f)}{b_{rp}} \\ & + \sum\limits_{f \in \text{notNeed}(\text{files}(t) \cap \mathcal{F}(r),q)} \frac{w(f)}{b_{rq}} \end{array} \right\}, \quad (5.20)$$

according to the objective UploadTime. Here, $U(1)$ gives the bottleneck value before the reassignment. The $\max_r\{\cdot\}$ corresponds to the bottleneck value upon realizing the reassignment. The set $\text{files}(t) \cap \mathcal{F}(r)$ 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 notNeed$(\text{files}(t), q)$.

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 LBTimeEX and EstUBTime. It also maintains the identities

Algorithm 5.6. EstUB-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:         end if
12:         if bottleneck repository or processor is changed then
13:            goto 2
14:         end if
15:      end if
16:   end for
17: end while
```

of the repositories and the processors that attain the maximum and the second maximum load in terms of the three cost components.

The EstUB-Refinement heuristic is similar to the LB-RefinementEX 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, $g_{comp}(t,p,q) + g_{download}(t,p,q)$ 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, $g_{comp}(t,p,q)$ and $g_{download}(t,p,q)$ are computed according to (5.18) and (5.19), respectively. The best reassignment of a task is realized if the total gain is non-negative. 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 EstUB-Refinement procedure.

A refinement pass consists of an LB-RefinementEX and an EstUB-Refinement. Similar to [22], Kaya et al. [23] choose to apply three refinement passes and run the LB-RefinementEX and EstUB-Refinement heuristics with at most five iterations of the while loops (see line 2 of Algs. 5.5 and 5.6).

Three-Phase Approach

Kaya et al. use the same three-phase approach described in Sect. 5.3.2. In the first phase, they use the greedy constructive heuristics [17, 20] to find an initial task-to-processor assignment. They modify these heuristics to handle multiple data repositories and implement them in such a way that their outputs are task-to-processor assignments rather than complete schedules. After refining these

task assignments in the second phase by using the above-mentioned refinement approach, they use the modified versions of the greedy constructive heuristics to find the inter- and intra-processor ordering of task executions. For more details, we refer the reader to [23].

Experimental Analysis

Kaya et al. give various experimental results for the assessment of the proposed heuristics and objective functions. Here we restate some important results to show the effectiveness of the proposed heuristic. A detailed and complete list of the experiments conducted to analyze the performance of the heuristics can be found in [23].

The experimental setting of [23] is similar to the one in [22]. The authors generate five sets of applications, where the computation-to-communication ratio, ρ, is different for each application set. For each application set, they generate 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]. The applications contains $T = 3000$ tasks and $F = 3000$ files where file sizes were random integers ranging from 50 Mbytes to 70 Gbytes.

Similar to [22], to create a heterogeneous system, they use the GridG network topology generator [27]. The generated framework has $P = 32$ processors and up to nine data repositories. The bandwidths of the communication links between repositories and processors were in between 10 Mbit/s and 1 Gbit/s.

Table 5.6 summarizes the results of the experiments conducted to validate the relation between the objective functions proposed for refining task assignments and the turnaround time. That is, these values display the amount of improvements needed in LBTimeEX and EstUBTime to attain one unit of improvement in the turnaround time. As seen in Table 5.6, close to one unit (between 0.97 and 1.29) of improvements are needed in LBTimeEX, 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

Table 5.6. Effectiveness of the Objective Functions

Heuristic in	Min		Max		Avg	
the first phase	EstUB	LB-EX	EstUB	LB-EX	EstUB	LB-EX
Communication	1.880	1.004	3.151	1.191	2.348	1.071
Computation	1.275	0.973	2.097	1.051	1.682	1.005
Duration	1.699	0.989	2.994	1.273	2.166	1.086
Payoff	1.949	1.031	3.443	1.160	2.484	1.083
Advance	1.951	1.036	4.423	1.287	2.871	1.161

The amount of improvements in LBTimeEX *and* EstUBTime *objective values required to obtain one unit of improvement in the turnaround time, i.e.,* Δ(LBTimeEX)/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.*

Table 5.7. Relative performances of the heuristics for the multiple-repository case

Heuristic	Cost
Iterative-Improvement-Based Heu.	1.005
Sufferage	1.108
MinMin	1.133
MaxMin	1.459

Table shows the averages of the relative performances of every heuristic normalized with respect to the best/fastest heuristic for each scheduling instance.

we consider. It is clear that, different and tighter estimations would increase the efficiency of the proposed heuristic in terms of the turnaround time.

Kaya et al. also perform some experiments to compare the performance of their approach with some greedy constructive scheduling heuristics. They also modify the *MinMin*, *MaxMin* and *Sufferage* to compare their approach with these heuristics. The experimental results show that *Sufferage* is the best among the three heuristics. The proposed iterative-improvement-based heuristic has been reported to be considerably faster than the Sufferage heuristic (approximately 90 times) while obtaining 10% improvement on the average in the turnaround times. Table 5.7 summarizes the results of these experiments.

5.5 Conclusions

This chapter surveys the heuristics for scheduling file-sharing tasks on heterogeneous environments and shows a generic approach to adapt the iterative-improvement-based refinement heuristics to the task scheduling problem.

In this approach, the task scheduling problem is considered as involving two consecutive processes: task assignment which determines the task-to-processor assignments, and execution ordering which determines the order of inter- and intra-processor task executions. This approach enables 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 is then used to generate a better schedule during execution ordering process.

The presented heuristics are static in the sense that the schedule is determined before the program execution begins. Real-life environments with large and non-dedicated computing platforms may require dynamic scheduling to adapt to the run-time changes such as increases in the workload, processor failures, and link failures. The refinement heuristics seem to be viable to adapt the original schedule to the run-time changes. However, dynamic scheduling methods interact with other system components such as process migration mechanism whose costs should be considered in the refinement heuristics.

The presented heuristics can be used in population-based heuristics. First, the solutions found by the heuristics can be used to generate an initial generation. Note that the randomized part of these heuristics is the creation of the visit orders

during the refinement phases. Therefore, we believe that this approach can create initial generations of very small sizes. Second, the refinement heuristics can be used to improve the current individuals before producing the next generation. In other words, the refinement heuristics can be used to let the individuals mature for a while (for a number of refinement phases) before producing the next generation. We have confidence that this approach will deliver promising solutions.

References

1. Ali, S., Siegel, H.J., Maheswaran, M., Hensgen, D., Ali, S.: Task execution time modeling for heterogeneous computing systems. In: Raghavendra, C. (ed.) Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000), Cancun, Mexico, May 2000, pp. 185–199. IEEE, Los Alamitos (2000)
2. Alpert, C.J., Kahng, A.B.: Recent directions in netlist partitioning: A survey. Integration, The VLSI Journal 19(1-2), 1–81 (1995)
3. Aykanat, C., Pınar, A., Çatalyürek, Ü.V.: Permuting sparse rectangular matrices into block-diagonal form. SIAM Journal on Scientific Computing 25(6), 1860–1879 (2004)
4. Banino, C., Beaumont, O., Carter, L., Ferrante, J., Legrand, A., Robert, Y.: Scheduling strategies for master-slave tasking on heterogeneous processor platforms. IEEE Transactions Parallel and Distributed Systems 15(4), 319–330 (2004)
5. Beaumont, O., Boudet, V., Robert, Y.: A realistic model and an efficient heuristic for scheduling with heterogeneous processors. Technical Report RR-2001-37, LIP, ENS Lyon, France (September 2001)
6. Beaumont, O., Legrand, A., Marchal, L., Robert, Y.: Steady-state scheduling on heterogeneous clusters. International Journal of Foundations of Computer Science 16(2), 163–194 (2005)
7. Beaumont, O., Marchal, L., Robert, Y.: Broadcast trees for heterogeneous platforms. Technical Report RR-2004-46, LIP, ENS Lyon, France (November 2004)
8. Berge, C.: Hypergraphs. North Holland, Amsterdam (1989)
9. Berman, F.: High-performance schedulers. In: Foster, I., Kesselman, C. (eds.) The Grid: Blueprint for a new computing infrastructure, ch. 12, pp. 279–309. Morgan Kaufmann, San Francisco (1999)
10. Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S.M., Hayes, J., Obertelli, G., Schopf, J.M., Shao, G., Smallen, S., Spring, N.T., Su, A., Zagorodnov, D.: Adaptive computing on the Grid using AppLeS. IEEE Transactions on Parallel and Distributed Systems 14(4), 369–382 (2003)
11. Casanova, H.: Network modeling issues for Grid application scheduling. International Journal of Foundations of Computer Science 16(2), 145–162 (2005)
12. Casanova, H., Legrand, A., Zagorodnov, D., Berman, F.: Heuristics for parameter sweep applications in Grid environments. In: Proc. Ninth Heterogeneous Computing Workshop, pp. 349–363. IEEE Computer Society Press, Los Alamitos (2000)
13. Casanova, H., Obertelli, G., Berman, F., Wolski, R.: 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 Press, Los Alamitos (2000)
14. Çatalyürek, Ü.V., Aykanat, C.: A hypergraph model for mapping repeated sparse matrix-vector product computations onto multicomputers. In: Proceedings of The Second International Conference on High Performance Computing, HiPC 1995, Goa, India (1995)

15. Çatalyürek, Ü.V., Aykanat, C.: Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. IEEE Transactions Parallel and Distributed Systems 10(7), 673–693 (1999)
16. Fidducia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: 19th ACM/IEEE Design Automation Conference, pp. 175–181 (1982)
17. Giersch, A., Robert, Y., Vivien, F.: Scheduling tasks sharing files on heterogeneous clusters. Technical Report RR-2003-28, LIP, ENS Lyon, France (May 2003)
18. Giersch, A., Robert, Y., Vivien, F.: Scheduling tasks sharing files from distributed repositories. Technical Report RR-2004-04, LIP, ENS Lyon, France (February 2004)
19. Giersch, A., Robert, Y., Vivien, F.: Scheduling tasks sharing files from distributed repositories. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 246–253. Springer, Heidelberg (2004)
20. Giersch, A., Robert, Y., Vivien, F.: 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, Los Alamitos (2004)
21. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing 48(1), 96–129 (1998)
22. Kaya, K., Aykanat, C.: Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogeneous master-slave platforms. IEEE Transactions on Parallel and Distributed Systems 17(8), 883–896 (2006)
23. Kaya, K., Uçar, B., Aykanat, C.: Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories. Journal of Parallel and Distributed Computing 67, 271–285 (2007)
24. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. The Bell System Technical Journal 49(2), 291–307 (1970)
25. Khanna, G., Vydyanathan, N., Kurc, T., Çatalyürek, Ü.V., Wyckoff, P., Saltz, J., Sadayappan, P.: A hypergraph partitioning based approach for scheduling of tasks with batch-shared I/O. In: Proceedings of Cluster Computing and Grid (2005)
26. Lengauer, T.: Combinatorial Algorithms for Integrated Circuit Layout. Wiley–Teubner, Chichester (1990)
27. Lu, D., Dinda, P.A.: GridG: Generating realistic computational grids. SIGMETRICS Perform. Eval. Rev. 30(4), 33–40 (2003)
28. Maheswaran, M., Ali, S., Siegel, H.J., Hensgen, D., Freund, R.: Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. Journal of Parallel and Distributed Computing 59(2), 107–131 (1999)
29. Meuer, H.W., Dongarra, J.J., Strohmaier, E.: TOP500 Supercomputer Sites. In: Proceedings of the IEEE/ACM Supercomputing Conference, SC 2003, 22th edn., Phoenix, USA (2003)
30. Saif, T., Parashar, M.: Understanding the behavior and performance of non-blocking communications in MPI. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 173–182. Springer, Heidelberg (2004)
31. Sanchis, L.A.: Multiple-way network partitioning. IEEE Transactions on Computers 38(1), 62–81 (1989)
32. Uçar, B., Aykanat, C.: Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. SIAM Journal on Scientific Computing 25(6), 1837–1859 (2004)