ITERATIVE-IMPROVEMENT-BASED HEURISTICS FOR ADAPTIVE SCHEDULING OF TASKS SHARING FILES ON HETEROGENEOUS MASTER-SLAVE ENVIRONMENTS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Kamer Kaya

August, 2004

ABSTRACT

ITERATIVE-IMPROVEMENT-BASED HEURISTICS FOR ADAPTIVE SCHEDULING OF TASKS SHARING FILES ON HETEROGENEOUS MASTER-SLAVE ENVIRONMENTS

Kamer Kaya
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
August, 2004

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 to use iterative-improvement based heuristics for refining the task assignments according to two novel objective functions. Unlike the actual scheduling 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.

Keywords: Task scheduling, file-sharing tasks, heterogeneous master-slave platforms, iterative-improvement.

ÖZET

YİNELEMELİ İYİLEŞTİRME TABANLI BULUŞSAL YÖNTEMLERİN ORTAK DOSYA KULLANAN GÖREVLERİN TÜRDEŞ OLMAYAN İSTEMCİ SUNUCU ORTAMLARINDA UYARLAMALI ZAMANLAMASINDA KULLANIMI

Kamer Kaya
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Ağustos, 2004

Ortak dosya kullanan bağımsız görevlerin türdeş olmayan istemci sunucu ortamlarında zamanlanması son zamanlarda Grid çevrelerinde önemli uygulamalarda kullanılmaktadır. Bu problem için daha önce önerilen buluşsal yöntemlerin hepsi ardışık zamanlayıcı bir yaklaşıma ve görevlerin anlık bitiş zamanlarını kullanan açgözlü bir karar verme mekanizmasına dayanmaktadır. Gözlemlerimize göre, bu tür bir karar verme mekanizması anlık bitiş zamanlarının ortak dosya kullanımı bilgisinin genel yapısnı çıkartma yetersizliğinden dolayı bu bilgiyi etkili ve verimli bir şekilde kullanamamaktadır. Bu tür görevlerin zamanlanması probleminin çözümlenmesi için başlangıç görev ataması, atamaların geliştirilmesi ve görev yürütme sıralaması safhalarını içeren 3 safhalı yeni bir yaklaşım önerdik. Atama geliştirme safhası için uygulamalar birer hiperçizge olarak, görev atama problemi ise hiperçizge bölümleme benzeri bir problem gibi modellenmiştir. Önerilen yinelemeli iyileştirme tabanlı buluşsal yöntemlerin etkin ve verimli bir biçimde kullanımı geliştirilen hedef fonksiyonların düzgünlük özelliğine bağlıdır. Bu etkin ve verimli kullanım gerçek zamanlama maliyeti olan paralel yürütme zamanında bulunmayan düzgünlük özelliğine sahip yeni iki hedef fonksiyon önerilerek sağlanmıştır. Yapay olarak yaratılan uygulamalar ve türdeş olmayan sunucu istemci ortamları ile gerçekleştirilen deneylerin sonuçları önerilen 3 safhalı yaklaşımın açgözlü ardışık zamanlayıcı yaklaşıma göre daha başarılı oldugunu göstermiştir.

Anahtar sözcükler: Görev zamanlama, ortak dosya kullanan görevler, türdeş olmayan istemci sunucu ortamları, yinelemeli iyileştirme.

To my family

Acknowledgement

I would like to express my gratitude to my supervisor Prof. Cevdet Aykanat for his guidance, suggestions and invaluable encouragement throughout the development of this thesis. Also, I would like to thank to Prof. Altay Güvenir and Prof. Özgür Ulusoy for reading and commenting on the thesis.

I owe special thanks to Bora Uçar for providing infinite moral support. Also, I would like to thank for his endless patience when my help requests were intolerable.

I would like to express very special thanks to Serkan Bayraktar, Ata Türk and Berkant Barla Cambazoğlu for their supporting friendship and instructive comments.

Finally, I would like to thank my mother and father for their everlasting support. Without their support, I could not have come so far to express my gratitude to anyone in this page.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Grid [16, 19] is a distributed computing infrastructure designed for resource sharing and problem solving on a global scale. The existence of such a distributing infrastructure is important because the excessive need of computing power for today's advanced science and engineering problems emerges the need of a decentralized system which can be used by dynamic and multi-institutional sets of individuals or institutions. The sharing of Grid resources must be controlled by the resource providers and customers to specify what resources are shared, who can share a resource, who can use a shared resource and how a shared resource is used. A set of users specified with such rules is called a virtual organization(VO) in the literature. The aggregated resources like computing power, bandwidth, storage and memory form a huge environment with enormous potential, however, efficient use of this potential needs careful planning for resource management to improve the problem solving phase. This challenge arises the need of advanced tools for resource monitoring, discovery and selection process as well as advanced algorithms for assignment/scheduling of the submitted tasks [15].

In this work, we investigate the scheduling of independent but file-sharing tasks on heterogeneous master-slave environments. This scheduling framework has been recently studied in [7, 9, 10, 17, 18] for adaptive scheduling of parameter-sweep-like applications in Grid environments. Such applications arise within the *Application Level Scheduling* (AppLeS) project [7]. In this framework, the input files for tasks are initially stored in the master processor, and slave processors, which are responsible for task executions, have different network access bandwidths and different computing powers. Although tasks are independent, i.e. there are no inter-task communications, they may need same files to start their computation, so there is still an interaction between them. A file must be transferred to a slave if one or more of the assigned tasks need this file. If two or more tasks, which need the same file, are assigned to the same slave, the file will be transferred only once. Obviously, assigning such two or more tasks to the same slave may be useful, since this decreases number so the amount of file transfers. The objective of the scheduling problem is to find a schedule that minimizes the parallel execution time of the target applications 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 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 from 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 environments and it is an important problem for today's computational Grid [15] which contains highly heterogeneous environments. In a heterogeneous environment, highly interacting tasks with lots of shared files 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 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 and this work is a consequence of two previous works by Casanova et al. [9, 10] and Giersch et al. [17, 18]. Casanova et al. [9, 10] extended three heuristics, namely *MinMin*, *MaxMin* and *Sufferage*, which were initially proposed in [25] 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. [17, 18] proposed several different heuristics. Their heuristics run several hundred times faster than existing ones and their scheduling performance are nearly equal to the ones in [9].

All of the existing heuristics are greedy and constructive, that is, they directly construct the schedule with consequent scheduling decisions of the unscheduled tasks one at a time without using any other information other than a greedy decision criterion which depends on the momentary completion time values of tasks. We claim that this greedy decision criterion cannot use the file sharing information effectively because of the scheduling decisions which depend on the momentary completion time values of unscheduled tasks. The consequences of this inefficiency may produce lots of file transfers and this can considerably increase the execution time of the schedule especially when file sizes are large. Even the file transfer times are comparable to task execution times, balancing computational loads of the processors can be hard since all existing scheduling heuristics tend to schedule a task on an overloaded processor because the file transfers already scheduled to a processor make the respective processor more favorable for next scheduling decisions. Such problems are closely related to the parallel execution time of a schedule and can decrease the performance of a scheduling heuristic drastically. These flaws of the existing constructive scheduling approach will be explained in Chapter 3.

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 and use iterative-improvement based heuristics for refining the task assignments according to these 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 scheduling 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 execution time of a schedule. The former one corresponds to a pessimistic view while the latter one corresponds an optimistic view for the execution scheme of a schedule.

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 experiments vary with respect to computation-to-communication ratio of the applications. When this ratio is big, the tasks of the applications in the experiment can be thought to consume more time for computation when compared to file transfer times. Obviously when this ratio is small, file transfer times consume more time than the task executions. The detailed explanation of this ratio can be found in Chapter 8.

We will use the terms basic and clustered master-slave frameworks for the two simulated master-slave environments. The basic one contains a master processor and several slave processors as mentioned before. A clustered master-slave framework is similar to the basic one. In this framework, multiple slave processors construct a cluster and a cluster takes the advantage of existence of a file storage unit which stores all of the transferred files needed by a task, which is scheduled on a processor in the respective cluster. So, if two or more tasks, which need the same file, are assigned to the processors in the same cluster, this file will be transferred to the file storage unit of the respective cluster only once. These platforms will be explained in detail in Chapter 2. For the sake of clarity, we introduce the proposed scheduling approach for the basic-master slave platform, then we give the modifications to the proposed approach for the clustered platform.

The rest of the thesis is organized as follows: The details of the scheduling framework are presented in Chapter 2. Chapter 3 discusses the structure and flaws of the existing constructive scheduling heuristics. The background material on the hypergraph partitioning problem and iterative-improvement heuristics is given in Chapter 4. Chapter 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 Chapter 6. Chapter 7 briefly mentions the modifications needed for adapting the proposed approach to the clustered master-slave framework. The experimental evaluation of the proposed scheduling approach is presented in Chapter 8. And finally Chapter 9 concludes the thesis.

Chapter 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 which depend on the subsets of a 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., $|\mathcal{A}| = \sum_{t_i \in \mathcal{T}} |files(t_i)|$.

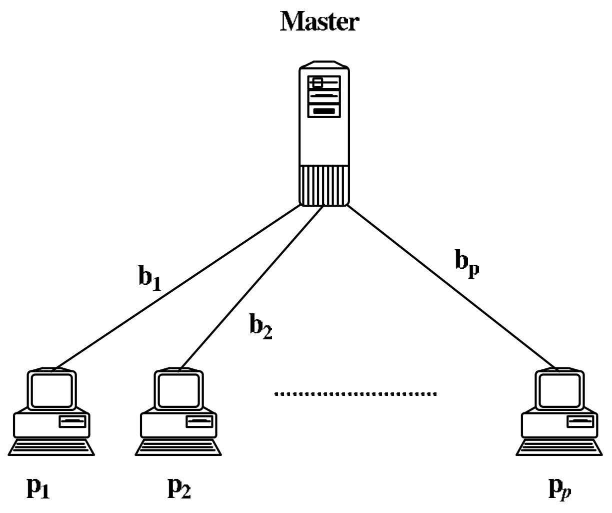


Figure 2.1: Basic master-slave communication network adapted from [17, 18]

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 exist 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. 2.1 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. 2.1, 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 on the same processor. Note that Fig. 2.1 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.

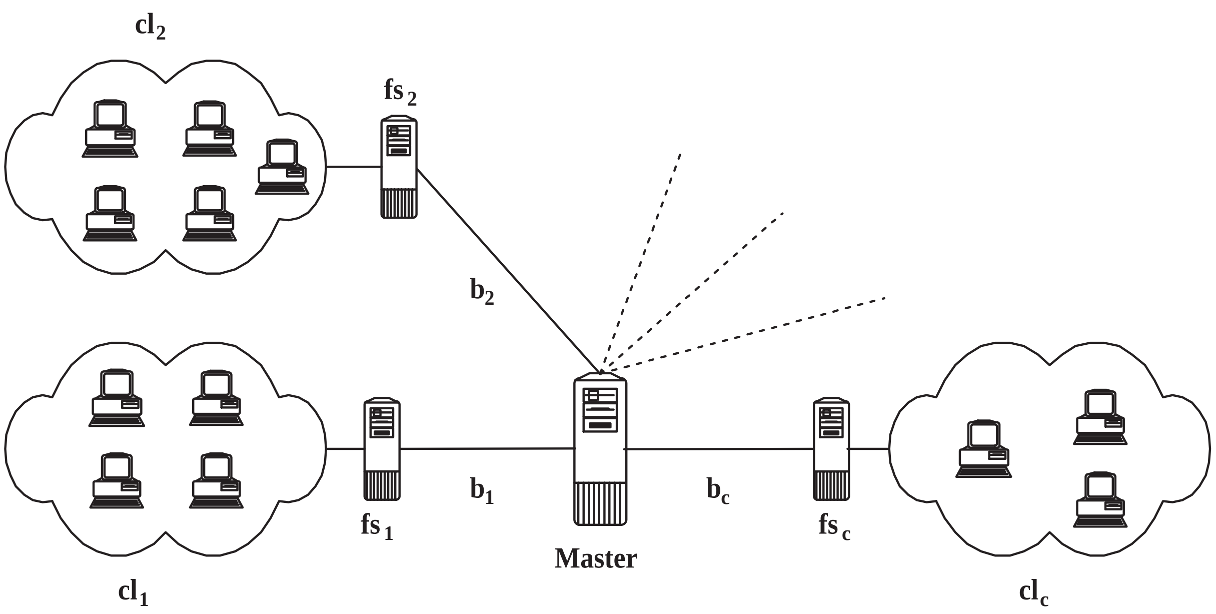


Figure 2.2: Clustered master-slave communication network adapted from [9, 10]

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 transferred files to cluster cl_i until the end of the schedule. Fig. 2.2 displays the main features of this framework. The network heterogeneity is modeled by assigning different bandwidth values to the links between the server and file storage units of the clusters. The intra-cluster communication costs and congestion due to the local file transfers from a file storage unit is not considered, because intra-cluster 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 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 $n \times p$ ETC (Expected Time to Compute) matrix. The ETC matrix can be *consistent* or *inconsistent* in terms of the relation between execution times of different tasks [3]. An ETC matrix is said to be consistent if a processor can execute some task t_i faster than another processor then this relation must be true for all other tasks. If there is no such relation between execution times then the ETC matrix is said to be inconsistent. A further categorization of ETC matrices is a partially-consistent ETC matrix [8, 25] which is a special type of inconsistent ETC matrices. In each partially-consistent ETC matrix there is a consistent sub-matrix with a structured relation between execution times of tasks and processors. 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]. Detailed explanation of execution time estimation process can be found in Chapter 8.

2.2.1 Cost Model

The cost of a schedule is the parallel execution time of the application in the computing environment. The schedule can be considered as a time-line 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 intra-processor task executions in such a way that the parallel execution time of the application is minimized.

For the sake of 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 same processor p_ℓ, a file needed by both t_i and t_j is transferred to p_ℓ only once.

Chapter 3

Existing Scheduling Heuristics

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

3.1 Structure

Fig. 3.1 shows the structure of the heuristics used by Casanova et al. [9, 10]. In Fig. 3.1, 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 of unscheduled tasks cannot be initialized before the file transfers of 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 3.1. As seen in Fig. 3.1, 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. [9, 10], Giersch et al. [17, 18] proposed several different heuristics. These heuristics have better time complexity and their solution quality

```
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 scheduling cost $f(CT(t_i, p_1), \ldots, CT(t_i, p_p))$ for task $t_i$
6:    Choose task $t_{i_b}$ with the "best" scheduling cost
7:    Pick the best processor $p_{\ell_b}$ for $t_{i_b}$ with minimum completion time
8:    Schedule $t_{i_b}$ on $p_{\ell_b}$, schedule its file transfers and execution as soon as possible
9:    Mark $t_{i_b}$ as scheduled
```

Figure 3.1: Structure of heuristics by Casanova et al. [9, 10]

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

Table 3.1: Definitions for the heuristics proposed by Casanova et al. [9, 10]

is comparable with those of the previous heuristics. Fig. 3.2 shows the structure of these heuristics. Table 3.1 displays the objective functions proposed by Giersch et al. [17, 18] 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. [17, 18] are also explained in Table 3.1. As seen in Fig. 3.2, 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 takes $\theta(pn + |\mathcal{A}|)$ 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 $O(pn \log n + pn|\mathcal{A}|)$.

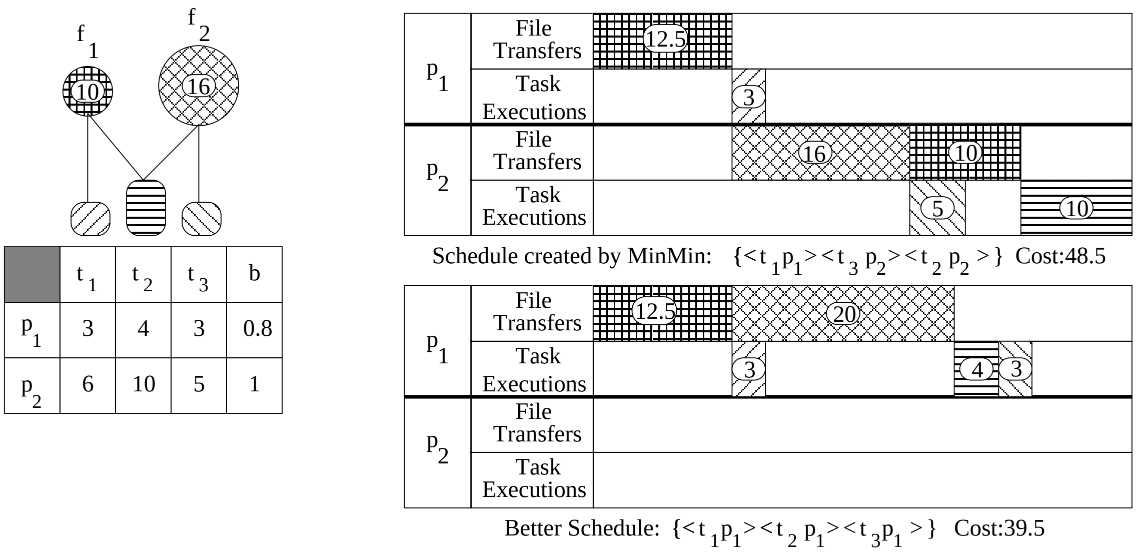
```
1: for each processor $p_\ell$ do
2:    for each task $t_i$ do
3:       Evaluate $OBJECTIVE(t_i, p_\ell)$
4:    Build the list $L(p_\ell)$ of the tasks sorted according
      to the value of $OBJECTIVE(t_i, p_\ell)$
5: while there remains a task to schedule do
6:    for each processor $p_\ell$ do
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 $p_{\ell_b}$, schedule its file transfers and execution as soon as possible
11:   Mark $t_{i_b}$ as scheduled
```

Figure 3.2: Structure of heuristics by Giersch et al. [17, 18]

Heuristic	Objective Function	Task Selection Order w.r.t. Objective Function
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 $w(files(t_i))$, a scaled version of 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 amount of file transfers, locality tries to avoids assigning a task to a processor if some of the files used by the task were already scheduled to be transferred to another processor.

Table 3.2: Definitions for the heuristics proposed by Giersch et al. [17, 18]



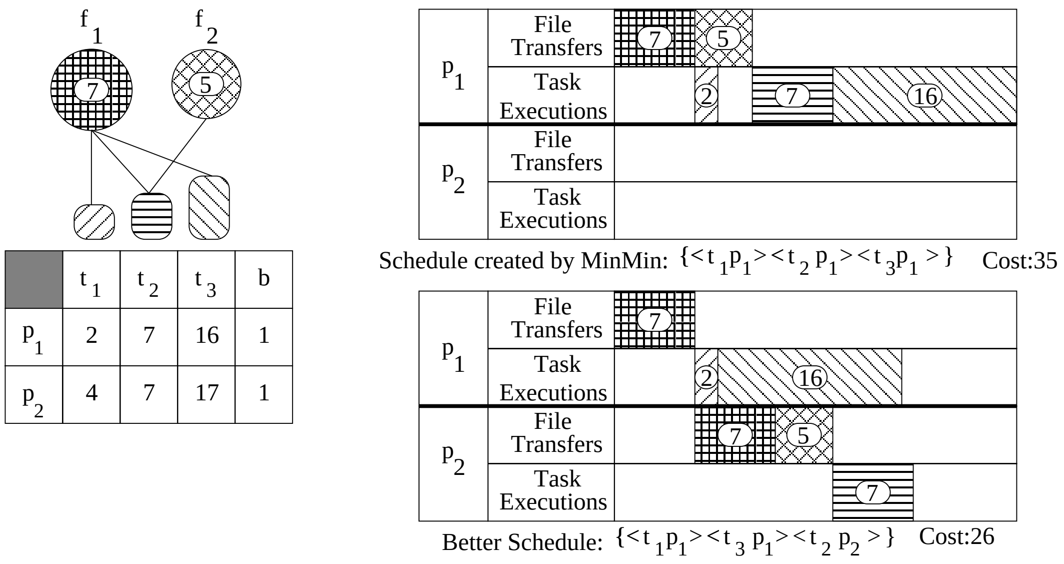
	t_1	t_2	t_3	b
p_1	3	4	3	0.8
p_2	6	10	5	1

Figure 3.3: A flaw of the greedy constructive approach for communication-intensive tasks

3.2 Flaws

The task-processor pair selection according to the momentary completion time values is the greedy decision criterion 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 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 model, extra file transfers can deteriorate the quality of the schedule, especially for communication-intensive tasks. We say a task is communication-intensive, if file transfer time for the task dominates its execution time. A similar problem can be seen in Fig. 3.3 for the *MinMin* heuristic.

Fig. 3.3 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 guarantees an extra transfer of file f_1 since an extra transfer of file f_1 or file f_2 for the processor p_2 or p_1 will be needed, respectively. The completion time of t_2 is smaller when it is scheduled on p_2 so *MinMin* creates a schedule with 48.5



f_1
f_2
7
5
p_1
File Transfers
Task Executions
p_2
Schedule created by MinMin: $\{<t_1 p_1><t_2 p_1><t_3 p_1>\}$ Cost:35
Better Schedule: $\{<t_1 p_1><t_3 p_1><t_2 p_2>\}$ Cost:26
2
16

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

Figure 3.4: Another flaw of the greedy constructive approach

time units of execution and an extra file transfer of f_1. However, there is another schedule without this extra file transfer and with better execution time as shown in Fig. 3.3. The scheduling performance of heuristic *MinMin* considerably worsen because of the extra file transfer it contains since file transfer times dominate task execution times of tasks due to the large file sizes.

Although extra file transfers constitute crucial bottleneck because of the single-port assumption and single master model, 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 not be able to resolve this problem. Although computational balance can be obtained by assigning the unscheduled tasks to the processors with lower computational loads, 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. Because of these effects, if former scheduling decisions lead the greedy heuristic to a schedule with a computational imbalance, it is not trivial to resolve this problem as expected. A similar problem can be seen in Fig. 3.4 for the *MinMin* heuristic.

Fig. 3.4 illustrates a sample application with three tasks and three 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. So, when *MinMin* tries to schedule t_3 on the empty processor p_2, an extra transfer of f_1 is required and this cannot be accomplished before the $19th$ time unit due to the previous file transfers. Since p_2 is also slightly slower than p_1, if t_3 is scheduled to p_2, the execution time of the schedule increases considerably. So *MinMin* avoids extra file transfers and generates a schedule with 35 time units because of scheduling all tasks onto the fastest processor. At last, *MinMin* schedules t_3 on the overloaded processor p_1, however, there is a much better schedule that utilizes both processors and has a lower parallel execution time of 26 units as shown in Fig. 3.4. This schedule is optimal but another optimal schedule can also be achieved by scheduling t_3 on p_1, t_2 on p_2 and t_1 on p_1, respectively.

Having such problems like the ones in examples given above may be reasonable for a greedy constructive approach in the sense of optimality since the target task scheduling problem is NP-Hard [17, 18]. Neither the minimum total file transfer time, nor the perfect computational load balance are necessary and sufficient for the optimality of a schedule. However, it is obvious that these costs usually tend to be bottleneck for schedules. especially at the extreme values of computation-communication ratio. We claim that a greedy constructive scheduling heuristic cannot minimize these conflicting costs easily because of the local information usage and we propose a refinement heuristic to resolve these flaws.

Chapter 4

Hypergraph Partitioning

In this chapter, 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 give $\mathcal{V}$. 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. If λ_k is equal to one, i.e., $\Lambda_k = \mathcal{V}_\ell$ then

n_k is said to be internal to the part $\mathcal{V}_\ell$. 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 *weighted connectivity*-1 metric is frequently used in VLSI circuit partitioning [23] and scientific computing [4, 11, 27]. In this metric, the contribution of each net n_k to the objective function is $w(n_k)(\lambda_k - 1)$, where $w(n_k)$ denotes the weight of net n_k. So, the contribution of the internal nets to the objective function is zero. The partitioning objective in this metric is the minimization of $CutSize(\Pi)$ which is given as:

$$CutSize(\Pi) = \sum_{n_k \in \mathcal{N}} w(n_k)(\lambda_k - 1). \tag{4.1}$$

The partitioning constraint is to maintain a balance on the part weights, i.e.,

$$(W_{max} - W_{avg})/W_{avg} \leq \epsilon, \tag{4.2}$$

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.

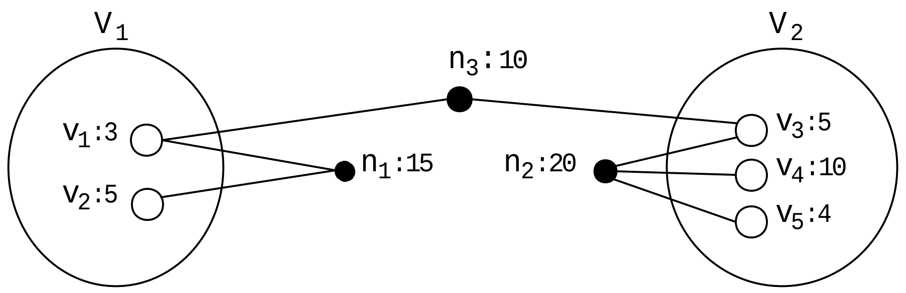


Figure 4.1: A sample partition

Fig. 4.1 illustrates a partition of an hypergraph with five vertices and three nets. Nets n_1 and n_2 are internal to the parts $\mathcal{V}_1$ and $\mathcal{V}_2$ respectively so their contributions to the objective function, i.e., $CutSize(\Pi)$, are zero. The connectivity of net n_3, λ_3, is two, so its contribution to the objective function is $10 \times (2 - 1) = 10$. The part weights are 8 and 19 for parts $\mathcal{V}_1$ and $\mathcal{V}_2$, respectively.

4.2 Iterative-Improvement Based Heuristics

The refinement heuristics proposed in this work are based on the iterative-improvement heuristics introduced by Kernighan-Lin (KL) [22] and Fidducia-Mattheyses (FM) [14]. Both KL and FM heuristics 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 VLSI [23, 2, 20, 21] and scientific computing [4, 11, 27, 12] 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 were proposed for the generalization of the approach to K-way refinement [26].

Chapter 5

Proposed Refinement Approach

Both effectiveness and efficiency of FM-based heuristics depend on "the smoothness" of the objective functions 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 operation, computing the gain, which is the change in the execution time, is a time consuming work and its time complexity is as high as computing the parallel execution time of a given schedule.

In order to alleviate the above problem, we consider the task scheduling problem as involving two consecutive processes: task assignment process which determines the task-to-processor assignments, 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 parallel execution time of a schedule. This refined task-to-processor assignment can then be used to generate better schedules during execution-ordering process.

5.1 Hypergraph Partitioning Models for Task Assignment in Heterogeneous Environments

We propose to use 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 Chapter 2 instead ofspecific 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. For a task-to-processor assignment Π, $Map(t_i)$ denotes the processor p_ℓ where $t_i \in \mathcal{T}_\ell$.

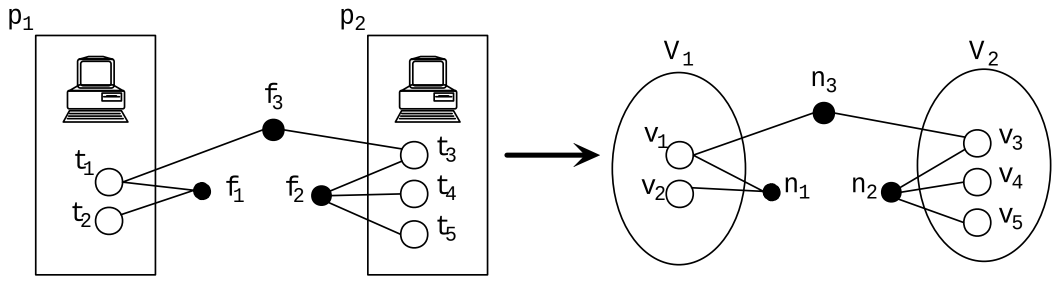


Figure 5.1: Representation of a task-to-processor assignment as a hypergraph partition

As proposed in [4, 11, 27], the general task-to-processor assignment problem can be effectively formulated as a hypergraph partitioning problem for homogeneous parallel environments. Fig. 5.1 illustrates a simple task assignment of five tasks to two processors as a 2-way partition of corresponding hypergraph. 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 Eq. 4.1 and the load balancing constraint given in Eq. 4.2 can be used effectively and efficiently for the refinement after modeling tasks and files as a hypergraph. However, the heterogeneity of the environment brings difficulties to the formulation of the task assignment problem. For this purpose, 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 *weighted connectivity* metric is the correct metric, rather than the *weighted connectivity*-1 metric, for encoding the total file transfer volume in a given task-to-processor assignment as shown below:

$$CommVol(\Pi) = \sum_{f_k \in \mathcal{F}} w(f_k)\lambda_k. \tag{5.1}$$

As an example, in Fig. 5.1, files f_1 and f_2 will be transferred to processors p_1 and p_2, respectively. So, they contribute to the communication cost of this task-to-processor assignment, however, they do not contribute to the objective function of the hypergraph partitioning problem if *weighted connectivity*-1 metric is used because they are internal nets. It is obvious 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.

Eq. 5.1 can also be used to represent total file transfer time if the network is homogeneous by normalizing file sizes with respect to the bandwidth value. In that case, minimization of total file transfer volume or total file transfer time are equivalent. However, master-slave platforms contain heterogeneous networks with different server-to-processor bandwidths. This forces us to modify the conventional definition of the connectivity of a net n_k in which different parts connected by n_k make equal contribution to the connectivity of n_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* for files in the task assignment problem for heterogeneous environments. The *heterogeneous connectivity* $\lambda_k^{'}$ of a file f_k as:

$$\lambda_k^{'} = \sum_{p_\ell \in \Lambda_k} \frac{1}{b_\ell}, \tag{5.2}$$

where Λ_k denotes the set of processors that has at least one task needing f_k as input to start its execution. Then total communication time, i.e., total file transfer time, can be defined as:

$$CommTime(\Pi) = \sum_{f_k \in \mathcal{F}} w(f_k)\lambda_k'. \tag{5.3}$$

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,

$$CompTime(\Pi) = \max_{\ell} \left(\sum_{t_i \in \mathcal{T}_\ell} c_{i\ell} \right). \tag{5.4}$$

The processor heterogeneity creates difficulties in modeling the computational cost of a task-to-processor assignment Π. The balancing constraint in hypergraph partitioning problem correctly encodes the computational load balance of a parallel system with homogeneous processors. In homogeneous environments, the average part weight (W_{avg} in Eq. 4.2) 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 Eq. 4.2 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 constant 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 parallel execution 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 execution times of all possible schedules that can be derived from a given task-to-processor assignment Π are bounded above by

$$UBTime(\Pi) = CommTime(\Pi) + CompTime(\Pi). \tag{5.5}$$

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 execution time of a schedule. As mentioned in Chapter 2, a processor can execute a task while that or other 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 execution times of all possible schedules that can be derived from a given task-to-processor assignment Π are bounded below by

$$LBTime(\Pi) = max\{CommTime(\Pi), CompTime(\Pi)\}. \tag{5.6}$$

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. It is clear that these are the only time intervals that excludes the possibility of an overlap between computation and communication. The length of these nonoverlapping intervals are negligible with respect to the total execution time of a schedule due to the large number of tasks.

These two assignment objectives are closely related to the execution time of a schedule and their minimization can generate good task-to-processor assignments which can be used to obtain schedules with better execution 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 parallel execution time, the aggregate objective should be closing the gap between these two objective functions while minimizing both of them. For this purpose, we 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 $UBTime(\Pi)$ or $LBTime(\Pi)$ is referred to here as *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 aggregate 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 to modify the move selection policy of FM-based approach accordingly in the LB-Refinement stage and/or in the UB-Refinement stage. We do not recommend to apply this modification to both LB- and UB-Refinement stages as this scheme is expected to restrict the effectiveness of the overall scheme.

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 to apply the proposed two-level gain computation scheme either to the 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 $UBTime(\Pi)$ compared to $LBTime(\Pi)$. Second, $UBTime(\Pi)$ is a relatively loose bound compared to $LBTime(\Pi)$. 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 to start the alternating refinement sequence with UB-Refinement stage. Our experimental results given in Chapter 8 verify our expectations.

Here, we describe the implementation scheme which adopts the two-level gain computation scheme in only LB-Refinement stage for the sake of presenting the use of both the conventional and proposed gain computation schemes. 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 selected 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. Figs. 5.2 and 5.3 show the general structures of UB- and LB-Refinement stages, respectively.

```
UB-Refinement(Π)
1: while a stopping criterion is 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:             $Map(t_i) \leftarrow p_{\ell_b}$
```

Figure 5.2: Structure of UB-Refinement stage

For the sake of run-time efficiency of move gain computations, a task move is considered as a two-step process: task leaves the source processor, which the task is assigned to, 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 sub gains. The first sub gain can be obtained in case of a decrease in $CompTime(\Pi)$ due to a leave from a processor with maximum computational load. The second sub gain 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 is the counterpart of the critical net concept used in hypergraph partitioning.

```
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:                 $Map(t_i) \leftarrow p_{\ell_b}$
```

Figure 5.3: Structure of LB-Refinement stage

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 in case of the assignment of t_i 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 with respect to the primary objective function is negative, it is impossible to obtain a positive gain in total since an arrival can add only a loss to the total move gain. In Figs. 5.2 and 5.3, ℓ_b denotes the index of the best destination 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, i.e., times, of processors. 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. 5.4 displays the pseudocode for the global update operations common to both UB- and LB-Refinement stages.

```
**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 $\ell_2$
```

Figure 5.4: Global update operations for UB- and LB-Refinement stages

Figs. 5.5 and 5.6 displays the algorithms used for leave gain and arrival loss computations in UB-Refinement stage. Recall that, conventional gain computation scheme is adopted in this stage. The conventional scheme can be considered as the proposed two-level gain computation scheme with $UBTime(\Pi)$ is both the primary and secondary objective function. So the good moves are defined the ones with positive gain, in which both the gain from total file transfer amount and maximum computational load are included. The best move is the one with maximum gain among good moves. A positive leave gain can be obtained if t_i needs at least one critical file for its processor, or t_i was previously assigned to processor p_{ℓ_1}. Otherwise, the move gain computations are unnecessary since the results are guaranteed to be negative or zero and because of the definition of good moves, UB-Refinement stage omits these computations as mentioned before. 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 in Fig. 5.6 for computing arrival losses for t_i.

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** $leaveGain$
```

Figure 5.5: UB-Refinement heuristics: leave gain computation for task t_i

```
**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 $p_{\ell_b}$
```

Figure 5.6: UB-Refinement heuristics: 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 $\{commLeaveGain, compLeaveGain\}$
```

Figure 5.7: LB-Refinement heuristics: leave gain computation for task t_i

```
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 $\{p_{\ell_b}, bestCommGain, bestCompGain\}$
```

Figure 5.8: LB-Refinement heuristics: arrival loss computations and best processor selection for task t_i

Figs. 5.7 and 5.8 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 total file transfer time, otherwise it tries to minimize maximum computational load. So, the task-to-processor assignment is adaptively refined with respect to the primary assignment objective function in which the relation between the values of $CompTime(\Pi)$ and $CommTime(\Pi)$ is important. 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 which gives minimum degradation to the secondary objective function $UBTime(\Pi)$.

Chapter 6

Implementation Choices

The proposed scheduling heuristic involves three phases: *initial task assignment*, *refinement* and *execution ordering*. In this chapter, 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 instead of using a heuristic which directly gives an initial task-to-processor assignment, because the proposed refinement heuristics are developed by taking the flaws of existing constructive scheduling heuristics into account. For this purpose, we use heuristics proposed by Giersch et al. [17, 18] because of their short runtimes. 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 respect to 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 terms of the parallel execution time is taken as the schedule for the target application.

6.2 Refinement Phase

Experiments show that the main improvement in the parallel execution 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 5 passes in the UB- and LB-Refinement stages. Likewise, the main improvement in the execution 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 3 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 intra-processor ordering of the task executions in this phase. Note that $CommTime(\Pi)$, $CompTime(\Pi)$ and hence the improved values of the objective functions remain the same as determined in the second phase. Fig. 6.1 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. [17, 18]. 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. 6.1 is run five times by using each one of the five objective functions proposed by Giersch et al. [17, 18] and the best schedule is selected for this Π.

6.4 Overall Complexity Analysis

As the heuristics proposed by Giersch et al. [18] are used in the initial task assignment phase, the time complexity of the first phase is $O(pn \log n + pn|\mathcal{A}|)$.

```
**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 remain 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 $p_\ell$, schedule its file transfers and execution as soon as possible
9:    Mark $t_i$ as scheduled
```

Figure 6.1: Execution ordering phase

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 Figs. 5.6 and 5.8. The update operations within a pass take $O(p|\mathcal{A}|)$ time because of the $O(p)$-time 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 time complexity, so the proposed three-phase scheduling approach takes $O(pn \log n + pn|\mathcal{A}|)$.

Chapter 7

Heuristics for the Clustered Framework

The topology of the clustered master-slave platform is similar to the topology of a basic master-slave platform as explained in Chapter 2. We modified existing constructive scheduling heuristics we used and proposed a three-phase heuristic for the clustered platform while preserving the main ideas of heuristics. This chapter explains the modifications needed for both the existing and the proposed scheduling heuristics.

7.1 Existing Constructive Scheduling Heuristics

In addition to the heuristics given in Table 3.1, Casanova et al. [9] also proposed 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_i from $CT(t_i, p_\ell)$ values. 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 to 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. [9] and Giersch et al. [18] become $O(pn^2 + cn|\mathcal{A}|)$ and $O(pn \log n + cn|\mathcal{A}|)$, respectively.

For adapting the readiness policy [18] 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, 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 the connectivity set, i.e., Λ_k, of each file f_k contains clusters instead of processors. So the definition of the heterogeneous connectivity of a net f_k changes to

$$\lambda_k^{'} = \sum_{cl_i \in \Lambda_k} 1/b_i, \tag{7.1}$$

which can be used in Eq. 5.3 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 a processor in cluster cl_i.

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

Figure 7.1: Clustered platform: Global update operations for UB- and LB-Refinement stages

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.

Fig. 7.1 shows the pseudocode of the global update operations modified for the clustered platform. Figs. 7.2 and 7.3 display the modified algorithms used for leave gain and arrival loss computations in UB-Refinement stage, respectively. Similarly, Figs. 7.4 and 7.5 display the modified algorithms used for leave gain and arrival loss computations in LB-Refinement stage, respectively.

UB-ComputeLeaveGainCL(t_i)

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

Figure 7.2: UB-Refinement heuristics for the clustered platform: leave gain computation for task t_i

UB-SelectBestMoveCL$(t_i, leaveGain)$

```
1: $p_{\ell_b} \leftarrow Map(t_i)$
2: $c\ell_{j_b} \leftarrow$ cluster of $p_{\ell_b}$
3: for each cluster $c\ell_j$ do
4:    $clusterLoss(c\ell_j) \leftarrow w(files(t_i))/b_j$
5: for all files $f_k \in files(t_i)$ do
6:    for each cluster $c\ell_j$ in $\Lambda_k$ do
7:       $clusterLoss(c\ell_j) \leftarrow clusterLoss(c\ell_j) - (w(f_k)/b_j)$
8: for each cluster $c\ell_j$ do
9:    for each candidate processor $p_\ell$ in $c\ell_j$ do
10:      $arrivalLoss(p_\ell) \leftarrow clusterLoss(c\ell_j)$
11: $bestMoveGain \leftarrow 0$
12: for each candidate processor $p_\ell$ do
13:    if $(Load(p_\ell) + c_{i\ell}) > leaveMaxLoad$ then
14:       $arrivalLoss(p_\ell) \leftarrow arrivalLoss(p_\ell) + Load(p_\ell) + c_{i\ell} - leaveMaxLoad$
15:    $moveGain \leftarrow leaveGain - arrivalLoss(p_\ell)$
16:    if $moveGain > bestMoveGain$ then
17:       $bestMoveGain \leftarrow moveGain$
18:       $p_{\ell_b} \leftarrow p_\ell$
19: return $p_{\ell_b}$
```

Figure 7.3: UB-Refinement heuristics for the clustered platform: arrival loss computations and best processor selection for task t_i

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

Figure 7.4: LB-Refinement heuristics for the clustered platform: leave gain computation for task t_i

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

Figure 7.5: LB-Refinement heuristics for the clustered platform: arrival loss computations and best processor selection for task t_i

Chapter 8

Experimental Results

We tested the performance of the proposed scheduling heuristic in comparison with the existing constructive heuristics by running 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=2000 tasks and m=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 200 Gbytes.

The experiments vary with the computation-to-communication ratio r of an application, where $r = AvgCompTime/AvgCommTime$. Here $AvgCompTime = (1/p)\sum_{i=1}^{n}\sum_{\ell=1}^{p} c_{i\ell}$ and $AvgCommTime = (1/b_{avg})\sum_{i=1}^{n} w(files(t_i))$. 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. The r value represents the ratio of the processor usage and network usage requirements of an application on a master-slave environment. We experimented the heuristics with 5 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 parallel execution 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 [24] for creating a heterogeneous master-slave platform with p=32 processors as follows: We created a Grid topology with 32 hosts and 9 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 5 clusters, where 4 clusters contain 8 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.

8.2 Task execution time estimation

As we mentioned before the quality of a schedule depends on the information used, so task execution times are important for the schedule generation process. This information can be estimated from machine benchmarks, users or previous executions. Modeling such information is also an important issue for simulating realistic heterogeneous environments. With a realistic modeling, the relative performances of various scheduling/mapping heuristics for different levels of task and processor heterogeneity can be efficiently computed. A related work by Ali et al. about modeling task execution times in heterogeneous computing environments can be found in [1].

In this work, we used machine benchmark values to generate realistic heterogeneous ETC matrices. We used the Top500 supercomputer list, maintained by Dongarra et al. [13], to estimate the task execution times as follows: We randomly chose our processors from mid-rank 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 [13]. The benchmark values R_{max}, N_{max} and $N_{1/2}$, provided in [13] 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 computation-to-communication ratio. 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. 8.1. The execution time of a task t_i with instance size N on a processor p_ℓ was estimated as $c_{i\ell}=(2/3)N^3/R_\ell(N)$. A sample ETC matrix created by this method is given in Table 8.1.

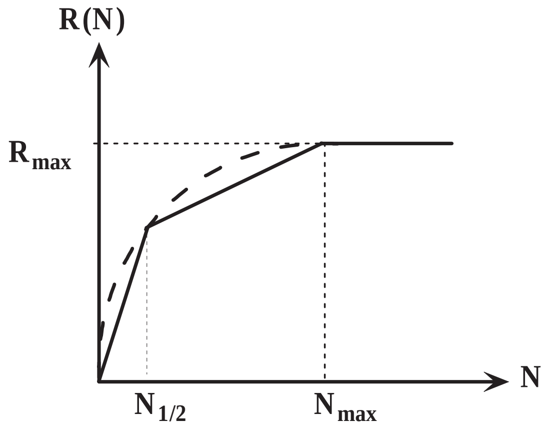


Figure 8.1: Piecewise linear approximation

Tasks	Processors						
	p_1	p_2	p_3	p_4	p_5	p_6	p_7
t_1	1214	1084	1435	1353	1280	745	818
t_2	376	335	445	419	397	456	500
t_3	543	485	643	605	573	659	722
t_4	139	124	164	160	146	168	185
t_5	1382	1233	828	1540	1456	849	932
t_6	5820	10262	6912	12795	6144	7084	7776
t_7	716	638	846	797	754	867	950
t_8	13	14	12	24	10	14	13
t_9	5581	9841	6627	12271	5891	6793	7456

Table 8.1: Sample ETC Matrix

8.3 Results

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

Table 8.2: Effects of the Implementation Choices in the Refinement Phase

Table 8.2 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 3rd and 7th rows, as well as the 4th and 8th 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 3rd row corresponds to the proposed scheme which uses the UB-LB ordering scheme with the two-level gain computation scheme adopted only in the LB-Refinement stage. The proposed iterative-improvement scheduling heuristic will be referred to as IIS here and hereafter.

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
MaxMin	1.794	0.999	1.645	0.998	0.976	0.999	1.458	1.000	1.033	1.000

Table 8.3: Effectiveness of the Proposed Assignment Objective Functions

Table 8.3 summarizes the results of the experiments conducted to validate the relation between the proposed assignment objective functions and the actual scheduling cost which is the parallel execution 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 scheduling 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 scheduling 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 8.3, 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 8.4 summarizes the results of the experiments conducted to validate the restriction on the number of alternating sequences of UB- and LB-Refinement

Basic master-slave platform						
Refinmenent Count	r: Computation-to-communication ratio					
	10	5	1	0.2	0.1	Avg
Ref. 0	1.375	1.374	1.336	1.300	1.150	1.307
Ref. 1	1.083	1.035	1.054	1.012	1.000	1.037
Ref. 2	1.038	1.021	1.030	1.004	1.000	1.019
Ref. 3	1.023	1.016	1.022	1.002	1.000	1.012
Ref. 4	1.016	1.011	1.016	1.002	1.000	1.009
Ref. 5	1.011	1.009	1.011	1.001	1.000	1.006
Ref. 6	1.009	1.006	1.008	1.002	1.000	1.005
Ref. 7	1.006	1.004	1.005	1.001	1.000	1.003
Ref. 8	1.003	1.002	1.003	1.001	1.000	1.002
Ref. 9	1.002	1.000	1.001	1.001	1.000	1.001
Ref. 10	1.000	1.000	1.000	1.000	1.000	1.000

Clustered master-slave platform						
Refinement Count	r: Computation-to-communication ratio					
	10	5	1	0.2	0.1	Avg
Ref. 0	1.327	1.425	1.358	1.040	1.013	1.233
Ref. 1	1.038	1.032	1.000	1.000	1.000	1.014
Ref. 2	1.023	1.012	1.000	1.000	1.000	1.007
Ref. 3	1.011	1.008	1.000	1.000	1.000	1.004
Ref. 4	1.006	1.006	1.000	1.000	1.000	1.002
Ref. 5	1.004	1.005	1.000	1.000	1.000	1.002
Ref. 6	1.003	1.003	1.000	1.000	1.000	1.001
Ref. 7	1.002	1.004	1.000	1.000	1.000	1.001
Ref. 8	1.000	1.002	1.000	1.000	1.000	1.000
Ref. 9	1.000	1.002	1.000	1.000	1.000	1.000
Ref. 10	1.000	1.000	1.000	1.000	1.000	1.000

Table 8.4: Effects of the Number of Alternating Sequences in a UB- and LB-Refinement Stage

stages. For this experiment, the number of alternating sequences is restricted to 10 and the values in the table are derived by computing the relative performances of the proposed heuristic when it is used with different number of alternating sequences. For computing relative performances, the best version of the proposed heuristic is selected as 1 for each application. Then the average of 50 relative performance values for 50 applications is computed for each r value. As seen in Table 8.4, the most significant part of the improvement on the execution time can be obtained with only one execution of UB- and LB-Refinement phases. Although the last ones may still improve the execution time, the effect of them is negligible compared to the effect of first few alternating sequences.

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
MaxMin	1.215	1.031	0.370	1.252	1.025	0.287	1.325	1.237	0.237
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
MaxMin	1.439	1.389	0.241	1.447	1.590	0.000	1.153	1.158	0.000
IIS	1.135	1.007	0.304	1.243	1.000	0.264	1.041	1.000	0.040

Table 8.5: Effectiveness of the Refinement

Table 8.5 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 *MinMin*, *Sufferage*, *XSufferage* 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 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 8.5, 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.

Tables 8.6 and 8.7 summarize 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 [18] and all 4 heuristics given in [9] were implemented. Tables 8.6 and 8.7 displays the relative scheduling performances of the heuristics ranked according to the their average performances. The last column of the tables also shows the 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 Tables 8.6 and 8.7, 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% and 16.4% worse schedules than IIS on the average, respectively. This relative performance gap is much higher for computation-intensive applications so that IIS produces at least 30% better

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	45.692
Sufferage	1.313	1.350	1.156	1.247	1.191	1.251	608.751
MinMin	1.433	1.496	1.186	1.259	1.140	1.303	647.672
Computation Readiness	1.457	1.566	1.399	1.454	1.197	1.415	3.920
Communication Shared Readiness	1.402	1.467	1.384	1.489	1.346	1.418	1.272
Communication Shared	1.402	1.507	1.389	1.483	1.347	1.426	1.014
Computation	1.490	1.590	1.349	1.427	1.319	1.435	3.596
Advance Shared Readiness	1.666	1.781	1.346	1.281	1.120	1.439	4.576
Communication Readiness	1.396	1.464	1.397	1.559	1.460	1.455	1.268
Communication	1.402	1.502	1.403	1.562	1.472	1.468	1.006
Advance Shared	1.713	1.852	1.480	1.331	1.141	1.503	4.340
Payoff Readiness	1.876	1.887	1.363	1.331	1.189	1.529	4.483
Payoff Shared Readiness	1.908	1.916	1.390	1.314	1.185	1.542	4.645
Duration Shared Readiness	1.442	1.599	1.498	1.626	1.565	1.546	4.594
Advance Shared Readiness Locality	1.756	1.833	1.404	1.463	1.395	1.570	4.734
Communication Shared Readiness Locality	1.497	1.569	1.418	1.665	1.719	1.574	1.440
Communication Shared Locality	1.496	1.582	1.424	1.660	1.720	1.576	1.183
MaxMin	1.601	1.638	1.437	1.626	1.635	1.587	607.605
Advance Shared Locality	1.775	1.856	1.448	1.469	1.395	1.589	4.498
Duration Shared	1.529	1.658	1.465	1.729	1.692	1.615	4.316
Duration Readiness	1.379	1.543	1.564	1.864	1.724	1.615	4.353
Payoff	1.962	1.951	1.445	1.415	1.305	1.616	4.254
Communication Readiness Locality	1.487	1.573	1.427	1.770	1.863	1.624	1.423
Communication Locality	1.488	1.589	1.437	1.772	1.866	1.630	1.171
Payoff Readiness Locality	1.889	1.925	1.408	1.507	1.490	1.644	4.642
Payoff Shared	2.005	1.970	1.498	1.430	1.318	1.644	4.429
Duration	1.439	1.567	1.545	1.888	1.798	1.647	4.170
Computation Readiness Locality	1.530	1.650	1.561	1.816	1.746	1.661	4.097
Computation Locality	1.512	1.660	1.534	1.825	1.800	1.666	3.788
Payoff Shared Readiness Locality	1.932	1.946	1.442	1.511	1.510	1.668	4.854
Payoff Locality	1.919	1.941	1.463	1.556	1.566	1.689	4.387
Duration Readiness Locality	1.440	1.610	1.674	1.953	1.908	1.717	4.556
Duration Shared Readiness Locality	1.508	1.662	1.627	1.930	1.894	1.724	4.778
Payoff Shared Locality	1.972	1.957	1.520	1.589	1.593	1.726	4.603
Duration Locality	1.464	1.627	1.610	1.973	1.977	1.730	4.291
Duration Shared Locality	1.542	1.681	1.587	1.989	1.969	1.753	4.491
Advance Readiness	2.537	2.744	2.067	1.264	1.087	1.940	4.391
Advance Readiness Locality	2.453	2.662	1.939	1.453	1.429	1.987	4.551
Advance Locality	2.529	2.716	2.028	1.460	1.438	2.034	4.319
Advance	2.785	2.944	2.314	1.296	1.116	2.091	4.185

Table 8.6: Relative Performances of Heuristics: Basic Master-Slave Platform

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	22.445
XSufferage	1.347	1.348	1.117	1.004	1.002	1.163	280.840
MinMin	1.331	1.333	1.187	1.067	1.048	1.193	263.044
Sufferage	1.357	1.458	1.254	1.072	1.049	1.238	263.561
Computation Readiness	1.531	1.550	1.252	1.010	1.008	1.270	3.607
Computation	1.433	1.479	1.398	1.047	1.018	1.275	3.545
Duration Readiness	1.480	1.523	1.421	1.219	1.151	1.359	3.734
Duration	1.387	1.484	1.566	1.252	1.160	1.370	3.755
Communication Shared	1.586	1.661	1.567	1.253	1.163	1.446	1.033
Communication Shared Readiness	1.589	1.666	1.563	1.253	1.163	1.447	1.069
Communication Readiness	1.578	1.655	1.580	1.271	1.174	1.452	1.033
Communication	1.577	1.653	1.590	1.271	1.174	1.453	1.018
MaxMin	1.706	1.830	1.584	1.158	1.018	1.459	244.850
Duration Shared Readiness	1.561	1.641	1.711	1.678	1.665	1.651	3.875
Duration Readiness Locality	1.712	2.025	1.945	1.403	1.313	1.680	3.874
Communication Shared Locality	1.776	1.935	1.766	1.546	1.434	1.692	1.145
Communication Shared Readiness Locality	1.773	1.949	1.774	1.543	1.427	1.693	1.198
Payoff Readiness	1.779	2.060	1.912	1.391	1.371	1.703	3.876
Communication Readiness Locality	1.786	2.015	1.806	1.518	1.446	1.714	1.138
Communication Locality	1.794	2.026	1.821	1.521	1.449	1.722	1.105
Duration Shared	1.511	1.674	1.964	1.766	1.703	1.723	3.855
Payoff Shared Readiness	1.812	2.108	1.941	1.414	1.352	1.726	3.927
Duration Locality	1.791	2.185	2.076	1.427	1.319	1.760	3.824
Payoff	1.825	2.132	2.026	1.465	1.439	1.777	3.895
Payoff Shared	1.858	2.182	2.074	1.493	1.417	1.805	3.908
Computation Readiness Locality	1.943	2.248	2.046	1.558	1.567	1.873	3.733
Advance Shared Readiness	1.654	1.908	2.220	1.949	1.694	1.885	3.921
Duration Shared Readiness Locality	1.886	2.170	2.182	1.702	1.628	1.914	3.951
Computation Locality	1.966	2.342	2.155	1.568	1.573	1.921	3.679
Payoff Readiness Locality	1.990	2.627	2.289	1.580	1.571	2.011	3.995
Duration Shared Locality	1.936	2.315	2.417	1.755	1.646	2.014	3.911
Payoff Shared Readiness Locality	2.021	2.658	2.324	1.590	1.573	2.033	4.011
Advance Shared	1.735	2.066	2.542	2.091	1.744	2.036	3.908
Payoff Locality	2.041	2.704	2.360	1.590	1.581	2.055	3.965
Payoff Shared Locality	2.083	2.730	2.394	1.596	1.582	2.077	3.982
Advance Readiness	1.883	2.436	3.491	1.506	1.327	2.129	3.859
Advance Shared Readiness Locality	1.975	2.578	2.602	1.876	1.849	2.176	4.038
Advance Readiness Locality	2.018	2.646	3.101	1.719	1.591	2.215	3.942
Advance	2.008	2.639	3.625	1.522	1.332	2.225	3.860
Advance Shared Locality	2.030	2.676	2.754	1.914	1.860	2.247	3.991
Advance Locality	2.092	2.812	3.457	1.728	1.587	2.335	3.915

Table 8.7: Relative Performances of Heuristics: Clustered Master-Slave Platform

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 these scheduling instances.

The above findings are in concordance with the experimental results given in [18], 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.

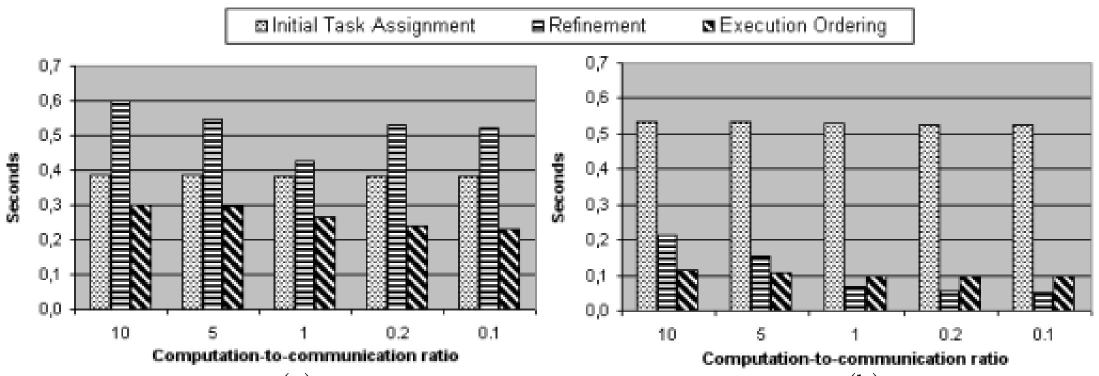
As seen in Tables 8.6 and 8.7, the performance gap between IIS and existing heuristics decreases in scheduling communication-intensive applications ($r = 0.2$ and 0.1) on the clustered master-slave platform. Although not seen in the tables, a similar pattern is also observed in the basic platform for 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 scheduling heuristics. This bottleneck incurs 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 performance of all existing heuristics become very close to each other for these scheduling instances as also stated in [18].

As seen in the last columns of Tables 8.6 and 8.7, IIS is an order of magnitude faster than the successful but slow heuristics [9], whereas it is an order of magnitude slower than the fast heuristics [18]. IIS produces approximately 25–30% better schedules while being 13–14 times faster than *MinMin* and *Sufferage* in the basic master-slave platform. Similarly, IIS produces approximately 16–24% better schedules while being 11–12 times faster than *MinMin*, *Sufferage* and *XSufferage* in the clustered master-slave platform.

Fig. 8.2 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 is taking 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 Chapters 6.4 and 7. Comparing Fig. 8.2 and Tables 8.6 and 8.7 show 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 the time spent for the improvement of the objective functions is directly proportional to the improvement in the actual scheduling cost, the parallel execution time of a schedule. This experimental finding also strengthens our claim about the direct relation between the proposed objective functions and the actual scheduling cost. Table 8.8 shows the execution times of the phases in detail.

As mentioned before, when the r value is decreasing the time spent for transferring files from the server becomes a serious bottleneck for the execution time of the schedule. This bottleneck makes the task scheduling problem more trivial. As an example, if task execution times are negligible compared to file transfer times, the trivial solution will be the assignment of all tasks to the processor with the highest bandwidth. Even the the hypergraph created from the application has a disconnected structure and network is homogeneous, the quality of such a solution will still be sufficient because of the single port assumption (A better solution in that case will incur the assignment of the components of the hypergraph to different processors). Because of the same reason, obtaining near optimal solutions in the clustered master-slave platform is easier than obtaining them in the basic one. The clustered structure of this platform makes possible the assignment of the tasks to different processors by transferring each file for these tasks only once. If the task scheduling problem become more trivial because of the reasons given above, the improvement by the refinement process also decreases as seen in the tables. In that case the solutions created by the existing heuristics would be expected to be better so the refinement process can be easily saturated by less vertex moves as seen in Fig. 8.2 and Table 8.8.

Table 8.8 also shows times spent for the UB- and LB-Refinement stages. As seen in the table, the time spent for the UB-Refinement is greater than the time spent for the LB-Refinement. This result validates our expectation about the variations in the task-move gains for the $UBTime(\Pi)$ and $LBTime(\Pi)$. Since the variations are larger in $UBTime(\Pi)$ the number of possible vertex moves is also larger. In our experiments, we realized that UB-Refinement usually stops because of the 5-pass restriction, whereas LB-Refinement because of the nonexistence of a task move with a positive gain.



(a) (b)

Figure 8.2: Execution times of the phases of the IIS heuristic in seconds: (a) basic master-slave platform, (b) clustered master-slave platform

Basic master-slave platform

Phase	r: Computation-to-communication ratio				
	10	5	1	0.2	0.1
Initial Task Assignment	0.387	0.401	0.381	0.379	0.376
UB-Refinement	0.534	0.576	0.380	0.383	0.367
LB-Refinement	0.053	0.036	0.061	0.171	0.160
Total Refinement	0.587	0.612	0.441	0.553	0.527
Execution Ordering	0.298	0.307	0.265	0.238	0.230
Total Time	1.272	1.320	1.087	1.170	1.133

Clustered master-slave platform

Phase	r: Computation-to-communication ratio				
	10	5	1	0.2	0.1
Initial Task Assignment	0.533	0.532	0.527	0.524	0.524
UB-Refinement	0.181	0.132	0.055	0.046	0.044
LB-Refinement	0.032	0.023	0.014	0.009	0.007
Refinement	0.213	0.155	0.069	0.054	0.051
Execution Ordering	0.113	0.108	0.097	0.097	0.097
Total Time	0.859	0.795	0.693	0.676	0673

Table 8.8: Execution Times of Phases in Seconds

Chapter 9

Conclusion

We investigated the problem of scheduling independent but file-sharing tasks onto heterogeneous master-slave platforms. All existing heuristics are constructive in nature and based on a common greedy criterion which decides according to completion time values of the tasks. We showed that this greedy decision criterion has a shortcoming in using the file-sharing interaction among tasks since momentary completion time values are inadequate to obtain an overall view of this interaction. In order to alleviate this problem, we investigated the feasibility of using iterative-improvement heuristics. However, the actual cost of a schedule does not satisfy the smoothness property required for the effective and efficient use of these heuristics. For this reason, 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 intra-processor 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 execution ordering process. We implemented a scheduling heuristic based on the proposed approach and tested its performance in comparison with 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.

There are various applications with different task types and task interactions. Besides, there are various computing platforms in Grid. In this work, parameter sweep applications and master-slave platforms were considered for the scheduling problem. Another heuristic designed for another type of the task scheduling problem may still suffer because of the structure of its solution procedure like the existing greedy constructive heuristics designed for scheduling tasks sharing files. Even this is not the case, iterative-improvement heuristics can still be used to improve the scheduling performance of various heuristics. Adaptation of the iterative-improvement heuristics to the task scheduling problem will be different for different applications and computing platforms since each application structure or computing platform has its own characteristics. Future work may explore the adaptation strategies of these heuristics and their ideas to the task scheduling problem.

Bibliography

[1] S. Ali, H. J. Siegel, M. Maheswaran, and D. Hensgen. Task execution time modeling for heterogeneous computing systems. In *9.th Heterogeneous Computing Workshop*, May 2000.

[2] C. J. Alpert, J. H. Huang, and A. B. Kahng. Multilevel circuit partitioning. In *Proceedings of the 34th annual conference on Design automation conference*, pages 530–533. ACM Press, 1997.

[3] R. Armstrong. Investigation of effect of different run time distributions on smartnet performance. Master's thesis, Department Of Computer Science, Naval Postgraduate School, 1997.

[4] C. Aykanat, A. Pınar, and U. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM Journal of Scientific Computing*, 25(6):1860–1879, 2004.

[5] O. Beaumont, A. Legrand, and Y. Robert. The master-slave paradigm with heterogeneous processors. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):897–908, 2003.

[6] C. Berge. *Hypergraphs.* North Holland, Amsterdam, 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 Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[8] T. D. Braun, H. J. Siegel, N. Beck, L. Blni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, R. F. Freund, and D. Hensgen. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *8.th Heterogeneous Computing Workshop*, May 1999.

[9] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for parameter sweep applications in grid environments. In *Proc. Ninth Heterogeneous Computing Workshop*, pages 349–363. IEEE Computer Society Press, 2000.

[10] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The apples parameter sweep template: User-level middleware for the grid. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 60. IEEE Computer Society, 2000.

[11] U. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions Parallel and Distributed Systems*, 10(7):673–693, 1999.

[12] U. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0.* Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey, 1999.

[13] J. Dongarra, H. Meuer, and E. Strohmaier. TOP500 Supercomputer Sites, 22nd edition. In *Proceedings of the Supercomputing Conference (SC2003), Phoenix, Arizona, USA*, 2003.

[14] C. M. Fidducia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *19.th ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[15] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*, chapter F. Berman. High Performance Schedulers, pages 279–309. Morgan-Kaufmann, 1999.

[16] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[17] 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.

[18] A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. In *PDP'2004, 12th Euromico Workshop on Parallel Distributed and Network-based Processing.* IEEE Computer Society Press, 2004.

[19] I.Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann, 1999.

[20] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[21] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 343–348. ACM Press, 1999.

[22] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[23] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley–Teubner, Chichester, UK, 1990.

[24] D. Lu and P. A. Dinda. Gridg: Generating realistic computational grids. *SIGMETRICS Perform. Eval. Rev.*, 30(4):33–40, 2003.

[25] 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. *Journal of Parallel and Distributed Computing*, 59(2):107–131, 1999.

[26] L. A. Sanchis. Multiple-way network parititoning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[27] B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal of Scientific Computing*, 25(6):1837–1859, 2004.