Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

**DIPLOMOVÁ PRÁCE**

Jan Vyšohlíd

# Rozvrhování v distribuovaných systémech

Katedra softwarového inženýrství

Vedoucí diplomové práce: doc. Ing. Jan Janeček, CSc.
Studijní program: Informatika, softwarové systémy

2010

Rád bych na tomto místě poděkoval doc. Ing. Janu Janečkovi, CSc. za připomínky, rady a náměty vedoucí ke zlepšení kvality práce a za jeho čas potřebný k vedení této práce. Mé další poděkování patří především Ing. Peteru Macejkovi za ochotu a trpělivost, s jakou mi odpovídal na mé dotazy, za připomínky, rady a náměty vedoucí ke zlepšení kvality práce, za poskytnutí testovacích serverů a aplikace, kterou jsem dále rozšiřoval, za čas strávený konzultacemi se mnou a celkovou pomoc při psaní této práce. Dále děkuji všem citovaným autorům za jejich díla, z nichž jsem mohl čerpat, a v neposlední řadě děkuji také rodičům a přítelkyni za podporu během psaní práce. Bez všech těchto lidí by práce nedostala svou nynější podobu.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 06.12.2010 Jan Vyšohlíd

# Contents

Název práce: Rozvrhování v distribuovaných systémech
Autor: Jan Vyšohlíd
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: doc. Ing. Jan Janeček, CSc.
e-mail vedoucího: janecek@fel.cvut.cz

Abstrakt: Předložená práce studuje metody rozvrhování v heterogenních distribuovaných systémech. Nejprve jsou uvedeny některé teoretické poznatky, které kromě vlastní teorie z oblasti rozvrhování obsahují také potřebné znalosti z teorie grafů a složitosti. Dále jsou představeny metody statického rozvrhování a nejznámější algoritmy řešící tento problém, po nichž následují základy dynamického rozvrhování a klasifikace používaných metod. V hlavní části práce jsou navrženy algoritmy, které respektují přidaná omezení. Tyto algoritmy jsou testovány pomocí přiložené aplikace a porovnány navzájem nebo s ostatními algoritmy, které většinou přidané podmínky na systém nekladou. Součástí práce je rovněž zmíněná aplikace a dokumentace k této aplikaci.

Klíčová slova: statické rozvrhování, dynamické rozvrhování, heterogenní distribuované systémy, přidaná omezení

Title: Scheduling in distributed systems
Author: Jan Vyšohlíd
Department: Department of Software Engineering
Supervisor: doc. Ing. Jan Janeček, CSc.
Supervisor's e-mail address: janecek@fel.cvut.cz

Abstract: The present work studies methods of scheduling in heterogeneous distributed systems. First there are introduced some theoretical basics which contain not only the scheduling theory itself but also the graph theory and the computational complexity theory. After that, compile-time scheduling methods and some well-known algorithms solving the problem are presented, followed by real-time scheduling basics and by classification of used methods. In the main part of the work there are proposed algorithms which respect additional restrictions. These algorithms are tested via the enclosed application and compared either to each other or to another algorithms which mostly don't respect additional restrictions. The mentioned application and the documentation for this application are a part of this work as well.

Keywords: compile-time scheduling, real-time scheduling, heterogeneous distributed systems, additional restrictions

# Chapter 1

# Introduction

This chapter introduces the task scheduling problem in heterogeneous distributed systems studied in the present work, main objectives of the thesis and chapters organization for better orientation.

## 1.1 Overview

Heterogeneous distributed system is a computing platform in which a number of machines with different parameters (such as processor performance) are interconnected with a network, where the network speed between each couple of machines can differ. These platforms are mostly utilized to execute computationally intensive applications. Single parallel architecture or homogeneous distributed system can be used to execute this kind of applications, too. But in some cases, distributed system have been shown to produce higher performance for lower cost than a single large machine. On the other hand, homogeneous distributed system needn't correspond with a real situation because the computing system generally consists of different machines interconnected with networks working on different speeds.

So homogeneous distributed system is a special case of more general heterogeneous distributed system. However [7], the performance of a parallel application on distributed system is highly dependent on scheduling of application tasks to machines of which the system consists. Scheduling can be qualified as crucial issue in distributed systems. The main objective of the scheduling algorithm is to assign tasks to machines and order their executions so that precedence requirements are satisfied and minimum overall completion time is achieved. When the structure of the parallel application (its task execution times, task dependencies and size of communicated data) is known a priori, the application can be represented by the directed acyclic graph (DAG) in which the nodes represent application tasks and the edges represent inter-task data dependencies. Each node is labeled by the computation cost of the task and each edge is labeled by the communication cost. In this case scheduling can be accomplished during the compile-time, so compile-time (also called static) scheduling algorithms are used. Otherwise, real-time (also called dynamic) scheduling methods must be applied.

Finding an optimal solution for the scheduling problem has been proven to be NP-complete, so heuristics are used to find a sub-optimal schedule rather than parsing all possible schedules. Because of the key importance to get high performance and low complexity, various heuristics were proposed in the literature.

## 1.2 Objectives

In this thesis studying the scheduling problem in heterogeneous distributed systems there are three main objectives:

- Analyze mechanisms applied to optimal task scheduling in heterogeneous distributed systems.
- Develop one or more methods which respect given time limits in multiple tasks systems (i.e., needn't require schedule length minimalization) and those tasks distribution (i.e., processors functional and performance abilities) or location (i.e., given entry task and exit task) restrictions.
- Compare proposed methods with some other methods that mostly do not require additional time limits and configuration restrictions.

## 1.3 Organization

The thesis is organized as follows: the second chapter presents basic definitions and theorems related to the problem of compile-time scheduling in heterogeneous distributed systems. In the background of scheduling there is found not only the scheduling theory itself but also the graph theory and the computational complexity theory. The application model and the computation model are introduced. The compile-time task scheduling problem is characterized, its complexity for optimal solution is explained and heuristics as sub-optimal solution are mentioned.

The third chapter presents compile-time scheduling algorithms, where the application can be represented by the directed acyclic graph (DAG). These algorithms are rather heuristics because they give a sub-optimal solution for the scheduling problem. Finding an optimal solution for the scheduling problem has been proven to be NP-complete (see the second chapter), so they can be used to find a sub-optimal schedule rather than parsing all possible schedules. They are classified in the beginning of the chapter. Finally, their performance and complexity characteristics are introduced and commented.

The fourth chapter presents real-time scheduling basics and real-time scheduling algorithms classification.

The fifth chapter presents evaluation method and comparison metrics which are used to compare performance of heterogeneous scheduling algorithms.

The sixth and seventh chapter presents the developed methods as given in the previous section and compare it with some other methods that mostly do not require additional time limits and configuration restrictions.

# Chapter 2

# Theoretical basics

This chapter presents basic definitions and theorems related to the problem of compile-time scheduling in heterogeneous distributed systems. In the background of scheduling there is found not only the scheduling theory itself but also the graph theory and the computational complexity theory. The application model and the computation model are introduced. The compile-time task scheduling problem is characterized, its complexity for optimal solution is explained and heuristics as sub-optimal solution are mentioned. Following sections involve basics of scheduling theoretical background.

## 2.1 Application model

This section presents the application model. Also some basic definitions from the graph theory are introduced. These are graph, graph model and task graph definitions [15]. Finally an entry and exit task definition is cited [16].

**Definition 2.1 (Graph)** *A graph is a pair $(V, E)$, where $V$ and $E$ are finite sets. An element $v$ of $V$ is called vertex and an element $e$ of $E$ is called edge. An edge is a pair of vertices $(u, v), u, v \in V$, and by convention the notation $e_{u,v}$ is used for an edge between the vertices $u$ and $v$.*
*In a directed graph, an edge $e_{u,v}$ has a distinguished direction, from vertex $u$ to vertex $v$, hence $e_{u,v} \neq e_{v,u}$, and such an edge shall be referred to as a directed edge.*

**Definition 2.2 (Graph model)** *In a graph theoretic abstraction, a program consists of two kinds of activity – computation and communication. The computation is associated with the vertices of a graph and the communication with its edges. A vertex is called node and the computation associated with it task. A task can range from an atomic instruction/operation (i.e., an instruction that cannot be divided into smaller instructions) to threads or compound statements such as loops, basic blocks, and sequences of these. All instructions or operations of one task are executed in sequential order; there is no parallelism within a task. A node is at any time involved in only one-activity – computation or communication.*

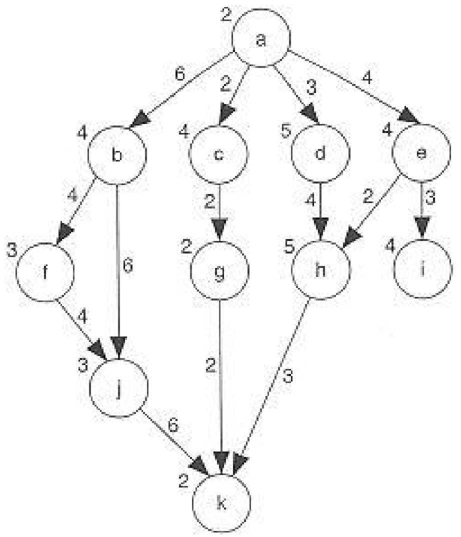


**Figure 2.1** *The task graph for a fictitious program [15]. Nodes are named by letters a-k; node and edge weights are noted beside them.*

**Definition 2.3 (Task graph (DAG))** *A task graph (see Figure 2.1 for example) is a directed acyclic graph $G = (V, E, w, c)$ representing a program $\mathcal{P}$ according to Definition 2.2. The nodes in $V$ represent the tasks of $\mathcal{P}$ and the edges in $E$ the communications between the tasks. An edge $e_{i,j} \in E$ from node $n_i$ to $n_j, n_i, n_j \in V$, represents the communication from node $n_i$ to node $n_j$. The positive weight $w(n)$ associated with node $n \in V$ represents its computation cost and the nonnegative weight $c(e)$ associated with edge $e \in E$ represents its communication cost.*

The application model can be defined as shown in Figure 2.2 [7]. The application is represented by a directed acyclic graph $G = (V, E, w, c)$, where:

- $V$ is the set of nodes. Each node $v_i \in V$ represents an application task, which is a sequence of instructions that must be executed serially on the same machine.
- $E$ is the set of communication edges. The directed edge $e_{i,j}$ joins $v_i$ and $v_j$, where $v_i$ is called the parent and $v_j$ is called the child. This also implies that $v_j$ cannot start until $v_i$ finishes and sends its data to $v_j$.
- $w$ is the computation cost function of the node $n \in V$. It can be represented by a $v \times p$ computation costs matrix $W$ in which

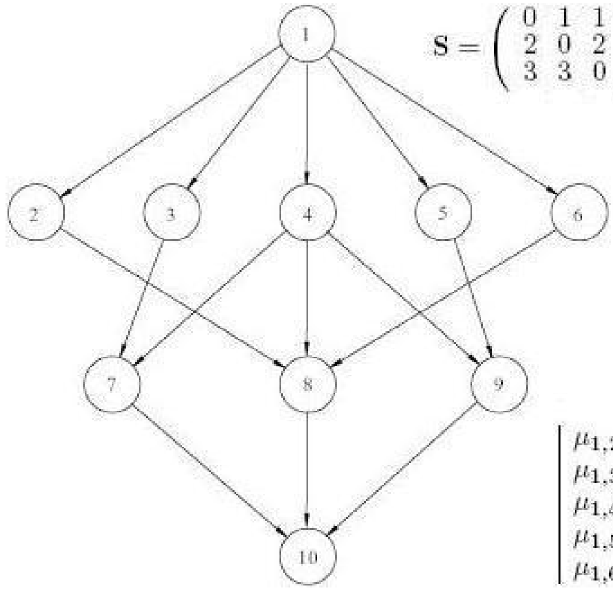
  $$W_{i,q} = t_i \cdot d_q,$$

  where $t_i$ is the time to execute task $v_i$ on machine $p_0$ (unit machine) and $d_q$ is the constant multiplier reflecting how machine $p_q$ differs from $p_0$.
- $c$ is the communication cost function of the edge $e \in E$. It can be represented by the set of communication costs $C$, where $c_{i,j} \in C$ is the communication cost carried by edge $e_{i,j}$.

**Definition 2.4 (Entry and exit task)** *In a given task graph, a task without any parent is called an entry task and a task without any child is called an exit task. Some of the task scheduling algorithms may require single-entry and single-exit task graphs. If there is more than one entry (exit) task, they are connected to a zero-cost pseudo entry (exit) task edges, which does not affect the schedule.*

## 2.2 Computing model

This section presents the computing model. Some basic definitions from the scheduling theory are introduced. These are fully connected network, target parallel system with heterogeneous processors, execution time, computation and communication costs, path length and critical path definitions [15].



$$\mathbf{S} = \begin{pmatrix} 0 & 1 & 1 \\ 2 & 0 & 2 \\ 3 & 3 & 0 \end{pmatrix} \quad \mathbf{R} = \begin{pmatrix} 0 & 2 & 4 \\ 4 & 0 & 6 \\ 5 & 3 & 0 \end{pmatrix}$$

$$\mathbf{W} = \begin{pmatrix} 20 & 15 & 5 \\ 16 & 12 & 4 \\ 20 & 15 & 5 \\ 12 & 9 & 3 \\ 16 & 12 & 4 \\ 16 & 12 & 4 \\ 12 & 9 & 3 \\ 20 & 15 & 5 \\ 4 & 3 & 1 \\ 16 & 12 & 4 \end{pmatrix}$$

| | | | | | |
|---|---|---|---|---|---|
| $\mu_{1,2}$ | 4 | $\mu_{2,8}$ | 4 | $\mu_{5,9}$ | 2 |
| $\mu_{1,3}$ | 2 | $\mu_{3,7}$ | 5 | $\mu_{6,8}$ | 3 |
| $\mu_{1,4}$ | 1 | $\mu_{4,7}$ | 3 | $\mu_{7,10}$ | 4 |
| $\mu_{1,5}$ | 2 | $\mu_{4,8}$ | 6 | $\mu_{8,10}$ | 2 |
| $\mu_{1,6}$ | 3 | $\mu_{4,9}$ | 5 | $\mu_{9,10}$ | 3 |

**Figure 2.2** *Application model and computation model matrices of heterogeneous computing systems [7].*

**Definition 2.5 (Fully connected network)** *A static network in which every processor has a direct link to any other processor is called fully connected. It has the nice property that it is nonblocking; that is, the communication of two processors does not block the connection of any other two processors in the network.*

**Definition 2.6 (Target parallel system – heterogeneous processors)** *A target parallel system $M_{hetero} = (P, \omega)$ consists of a set of processors $P$, whose heterogeneity, in terms of processing speed, is described by the execution time function $\omega$. The processors are connected by a communication network. This system has the following properties:*

1. *Dedicated System. The parallel system is dedicated to the execution of the scheduled task graph. No other program or task is executed on the system while the scheduled task graph is executed.*

2. *Dedicated Processor. A processor $p_q \in P$ can execute only one task at a time and the execution is not preemptive.*

3. *Cost-Free Local Communication. The cost of communication between tasks executed on the same processor, local communication, is negligible and therefore considered zero. This assumption is based on the observation that for many parallel systems remote communication (i.e., interprocessor communication) is one or more orders of magnitude more expensive than local communication (i.e., intraprocessor communication).*

4. *Communication Subsystem. Interprocessor communication is performed by a dedicated communication subsystem. The processors are not involved in communication.*

5. *Concurrent Communication. Interprocessor communication in the system is performed concurrently; there is no contention for communication resources.*

6. *Fully Connected. The communication network is fully connected. Every processor can communicate directly with every other processor via a dedicated communication link.*

**Definition 2.7 (Execution time)** *Let $G = (V, E, w, c)$ be a task graph and $M_{hetero} = (P, \omega)$ a heterogeneous parallel system. The execution time of $n \in V$ is the function $\omega : V \times P \rightarrow \mathbb{Q}^+$.*

**Definition 2.8 (Computation and communication costs)** *$M_{hetero} = (P, \omega)$ is a heterogeneous parallel system. The computation and communication costs of a task graph $G = (V, E, w, c)$ expressed as weights of the nodes and edges, respectively, are defined as follows:*

- *$w : V \rightarrow \mathbb{Q}^+$ is the computation cost function of the node $n \in V$. The computation cost $w(n)$ of node $n$ is the average time the task represented by $n$ occupies a processor of $P$ for its execution.*

- *$c : E \rightarrow \mathbb{Q}_0^+$ is the communication cost function of the edge $e \in E$. The communication cost $c(e)$ of edge $e$ is the time the communication represented by $e$ takes from an origin processor in $P$ until it completely arrives at a different destination processor in $P$. In other words, $c(e)$ is the communication delay between sending the first data item until receiving the last.*

The heterogeneous computing model as shown in Figure 2.2 [7] is a set $P$ of $p$ machines with different capabilities (heterogeneous processors) connected in a fully connected topology. As given below, this assumption is only for simulation simplicity and the communication costs can be simple modified to take into account the network topology. It is also assumed that:

- Any machine (processor) can execute the task and communicate with other machines at the same time.

- Once a machine (processor) has started task execution, it continues without interruption and on completing the execution it immediately sends the output data to all children tasks in parallel.

The communication costs per transferred byte between any two machines are stored in matrix $R$ of size $p \times p$. The communication startup costs between any two machines (for each line separately) are given in matrix $S$ of size $p \times p$. The communication cost of edge $e_{i,j}$ for transferring $\mu$ bytes of data from task $v_i$ (scheduled on $p_m$) to task $v_j$ (scheduled on $p_n$) is defined as

$$c_{i,j} = S_{m,n} + R_{m,n} \cdot \mu_{i,j},$$

where:

- $S_{m,n}$ is $p_m$ to $p_n$ communication startup cost (in seconds),
- $\mu_{i,j}$ is the amount of data transmitted from task $v_i$ to task $v_j$ (in bytes),
- $R_{m,n}$ is the communication cost per transferred byte from $p_m$ to $p_n$ (in seconds per byte).

As written in [8], the assumption of fully connected machines is only for simulation simplicity. If the network topology will be taking into account, the communication cost given above can be simple modified to take into account the network topology.

**Definition 2.9 (Path length)** *Let $G = (V, E, w, c)$ be a task graph. The length of a path $L$ in $G$ is the sum of the weights of its nodes and edges:*

$$len(L) = \sum_{n \in L, V} w(n) + \sum_{e \in L, E} c(e).$$

**Definition 2.10 (Critical path)** *Let $G = (V, E, w, c)$ be a task graph. A critical path $CP$ of $G$ is a longest path in $G$ (from the entry task to the exit task):*

$$len(CP) = \max_{L \in G} \{len(L)\}.$$

Clearly, there might be more than one critical path as several paths can have the same maximum length. The critical path tasks are all tasks of the critical path $CP$ and they are also called *Critical tasks*. In order to be able to compute the attributes needed to find the critical tasks the average computation and communication costs are utilized. The average computation cost and the average communication cost can be computed as

$$\overline{w_i} = \sum_{j=1}^{p} \frac{w_{i,j}}{p}, \qquad \overline{c_{i,j}} = \overline{S} + \overline{R} \cdot \mu_{i,j},$$

where $\overline{S}$ is the average communication startup cost over all machines computed as

$$\overline{S} = \sum_{q=1}^{p} \frac{S_q}{p}$$

and $\overline{R}$ is the average communication cost per transferred byte over all machines computed as

$$\overline{R} = \sum_{i=1}^{p} \sum_{j=1}^{p} \frac{R_{i,j}}{(p(p-1))}.$$

## 2.3 Scheduling problem

This section presents the scheduling problem. Some more definitions from the scheduling theory and a theorem build on NP-completeness theoretical knowledge are introduced. These are processor allocation, schedule, node finish time, schedule length and scheduling problem definitions. Finally the NP-completeness theorem is cited and commented [15].

**Definition 2.11 (Processor allocation)** *Processor allocation $\mathcal{A}$ of the task graph $G = (V, E, w, c)$ on a finite set $P$ of processors is the processor allocation function proc: $V \rightarrow P$ of the nodes of $G$ to the processors of $P$.*

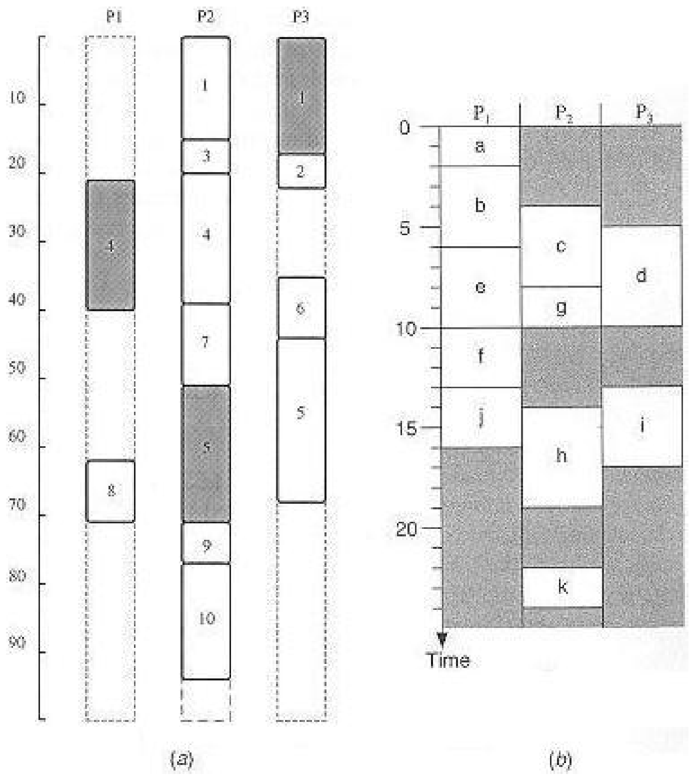


**Figure 2.3** *(a) Scheduling of Figure 2.2 application graph on three processors preserving communication heterogeneity (makespan = 94) [8]; (b) the Gantt chart of a schedule for the task graph of Figure 2.1 on three processors [15].*

**Definition 2.12 (Schedule)** *A schedule $S$ of the task graph $G = (V, E, w, c)$ on a finite set $P$ of processors is the function pair $(t_s, proc)$, where*

- *$t_s : V \rightarrow \mathbb{Q}_0^+$ is the start time function of the nodes of $G$. It can be computed recursively by traversing the task graph downward, starting from the entry task $n_{entry}$, as follows [7]:*

$$t_s(n_i) = \max_{n_j \in pred(n_i)} \{t_s(n_j) + w_j + c_{j,i}\},$$

  *where $pred(n_i)$ is the set of immediate predecessors of task $n_i$. For the entry task $n_{entry}$, the start time function value is equal to $t_s(n_{entry}) = 0$.*

- *proc: $V \rightarrow P$ is the processor allocation function of the nodes of $G$ to the processors of $P$.*

*The node $n \in V$ is scheduled to start execution at $t_s(n)$ on processor $proc(n) = p_q, p_q \in P$, which is denoted by $t_s(n, p_q)$; hence,*

$$t_s(n, p_q) \Leftrightarrow t_s(n), \qquad proc(n) = p_q, \qquad p_q \in P.$$

**Definition 2.13 (Node finish time)** *Let $S$ be a schedule for task graph $G = (V, E, w, c)$ on a heterogeneous parallel system $M_{hetero} = (P, \omega)$. The finish time of node $n$ on processor $p_q \in P$ is*

$$t_f(n, p_q) = t_s(n, p_q) + \omega(n, p_q).$$

**Definition 2.14 (Schedule length – called makespan)** *Let $S$ be a schedule for task graph $G$ = ($V$, $E$, $w$, $c$) on a heterogeneous parallel system $M_{hetero} = (P, \omega)$. The schedule length of $S$ is*

$$sl(S) = \max_{n \in V,\ p_q \in P} \{t_f(n, p_q)\} - \min_{n \in V,\ p_q \in P} \{t_s(n, p_q)\}.$$

**Definition 2.15 (Scheduling problem)** *Let $G = (V, E, w, c)$ be a task graph and $M_{hetero} = (P, \omega)$ a heterogeneous parallel system. The scheduling problem is to determine a feasible schedule $S$ of minimal length $sl$ for $G$ on $P$.*

**Theorem 2.16 (NP-completeness)** *Let $G = (V, E, w, c)$ be a task graph and $M_{hetero} = (P, \omega)$ a heterogeneous parallel system. The decision problem $H-Sched(G, M_{hetero})$ associated with the scheduling problem is as follows. Is there a schedule $S$ for $G$ on $M_{hetero}$ with length $sl(S) \leq T, T \in \mathbb{Q}^+$? $H-Sched(G, M_{hetero})$ is NP-complete.*

So as given in the theorem above, finding an optimal solution for the scheduling problem has been proven to be NP-complete. Therefore, heuristics can be used to find a sub-optimal schedule rather than parsing all possible schedules. These heuristics are classified in next two chapters and some of them are presented simultaneously as compile-time and real-time scheduling algorithms.

# Chapter 3

# Compile-time scheduling

This chapter presents some of well-known compile-time heterogeneous scheduling algorithms, where the application can be represented by the directed acyclic graph (DAG). Finding an optimal solution for the scheduling problem has been proven to be NP-complete (see the second chapter). These algorithms are therefore rather heuristics because they can be used to find a sub-optimal schedule rather than parsing all possible schedules. Static scheduling heuristics are classified in the beginning of the chapter. Performance and complexity characteristics are commented for each of these heuristics.

## 3.1 Algorithms classification

This section classifies static scheduling algorithms and brings a short description of each of their groups. According to [16], compile-time scheduling algorithms can be classified into two main groups, heuristic based algorithms and guided random search based algorithms. The former can be further classified into three subgroups: list scheduling heuristics, clustering heuristics and task duplication heuristics (see Figure 3.1). Following subsections present them more closely.

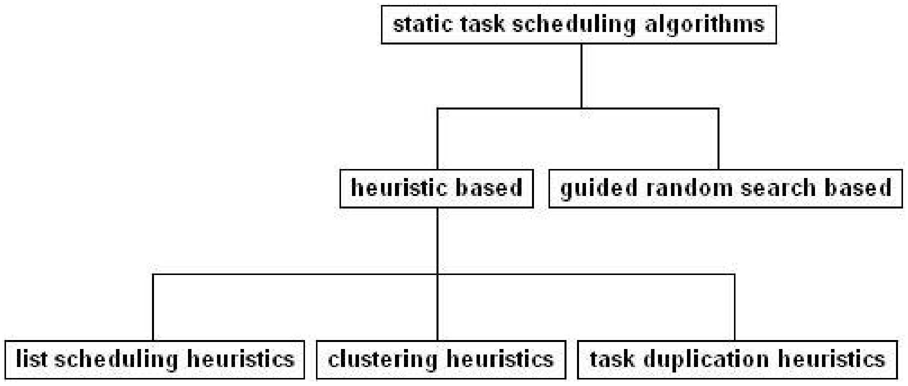

static task scheduling algorithms
heuristic based
guided random search based
list scheduling heuristics
clustering heuristics
task duplication heuristics

**Figure 3.1** *Classification of compile-time scheduling algorithms.*

From [7], static scheduling algorithms basically try to select a task (called a ready task) to be ready for machine assignment in each scheduling step. The task can be selected according to certain priority attributes. These priority attributes can be computed for all tasks before starting the machine assignment (static) or they can be recomputed to select the next ready task after assigning the current task to a machine (dynamic). Three different ways are used for machine assignment for a ready task: non-insertion based, where the first machine that is ready to execute the current ready task is selected, insertion based, where the idle time slots left by the previous scheduled tasks are examined in addition to the machines ready time, and duplication based, where certain or all parents are redundantly executed on some selected machines to minimize the start time of the current ready task.

### 3.1.1 List scheduling

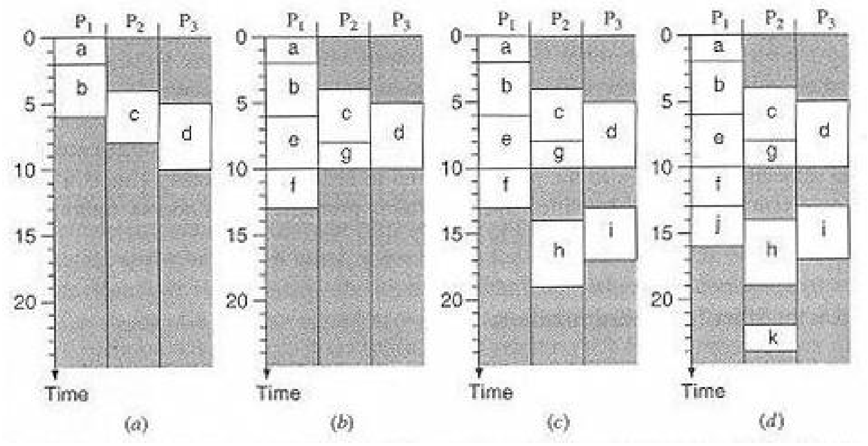


**Figure 3.2** *Example of simple list scheduling with start time minimization for the task graph of Figure 2.1: (a) to (c) are snapshots of partial schedules; (d) shows the final schedule [15].*

As written in [16], a list scheduling heuristic (see Figure 3.2) maintains a list of all tasks of a given graph according to their priorities. It has two phases: the task prioritizing (or task selection) phase for selecting the highest priority ready task and the processor selection phase for selecting a suitable processor that minimizes a predefined cost function (which can be the execution start time). Some of the examples are heterogeneous earliest finish time (HEFT) [16], critical path on a processor (CPOP) [16], fast load balance (FLB) [13], dynamic level scheduling (DLS) [14], mapping heuristic (MH) [5] or levelized-min time (LMT) [6]. Most of the list scheduling algorithms are for a bounded number of fully connected processors. List scheduling heuristics are generally more practical and provide better performance results at a lower scheduling time than the other groups.

#### Node priorities

Static priorities [15] are based on the characteristics of the task graph. In the simplest case, the priority metric itself establishes a precedence order among the nodes. It is then sufficient to order the nodes according to their priorities with any sort algorithm, for example Mergesort, which has a complexity of $O(V \times logV)$. Figure 3.3 shows the algorithm to create node list.

```
Assign a priority to each $n \in V$.
Put source nodes $n \in V$: pred($n$) = 0 into priority queue $Q$.
while $Q \neq 0$ do // $Q$ only contains free nodes
    Let $n$ be the node of $Q$ with highest priority; remove $n$ from $Q$.
    Append $n$ to list $L$.
    Put $\{n_i \in$ succ($n$): pred($n_i$) $\subseteq L\}$ into $Q$.
endwhile
```

**Figure 3.3** *Create node list algorithm.*

Level or critical path based priority schemes turn into dynamic priorities [15] when they are recalculated in each scheduling step based on the current partial schedule. In general, recalculating the levels or the critical path in each step of scheduling multiplies the costs for the level or critical path determination by a factor of $|V|$. However, an efficient implementation might only update the levels for those nodes that are affected by the scheduling of a node.

## Graph attributes

Except the earliest start time ($EST$) and the earliest finish time ($EFT$) defined in previous chapter as $t_s$ and $t_f$, the upward rank or the downward rank are used in heterogeneous list scheduling algorithms [16] as priority attributes. The upward rank of a task $n_i$ is recursively defined by

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{i,j}} + rank_u(n_j)),$$

where $succ(n_i)$ is the set of immediate successors of task $n_i$, $\overline{c_{i,j}}$ is the average communication cost of edge $(i, j)$ and $\overline{w_i}$ is the average computation cost of task $n_i$. Since the rank is computed recursively by traversing the task graph upward, starting from the exit task, it is called upward rank. For the exit task $n_{exit}$, the upward rank value is equal to

$$rank_u(n_{exit}) = \overline{w_{exit}}.$$

Basically, $rank_u(n_i)$ is the length of the critical path from task $n_i$ to the exit task, including the computation cost of task $n_i$. There are algorithms in the literature which compute the rank value using computation costs only, which is called static upward rank ($rank_u^s$). Similarly, the downward rank of a task $n_i$ is recursively defined by

$$rank_d(n_i) = \max_{n_j \in pred(n_i)} \{rank_d(n_j) + \overline{w_j} + \overline{c_{j,i}}\},$$

where $pred(n_i)$ is the set of immediate predecessors of task $n_i$. The downward ranks are computed recursively by traversing the task graph downward starting from the entry task of the graph. For the entry task $n_{entry}$, the downward rank value is equal to zero. Basically, $rank_d(n_i)$ is the longest distance from the entry task to task $n_i$, excluding the computation cost of the task itself.

### 3.1.2 Clustering

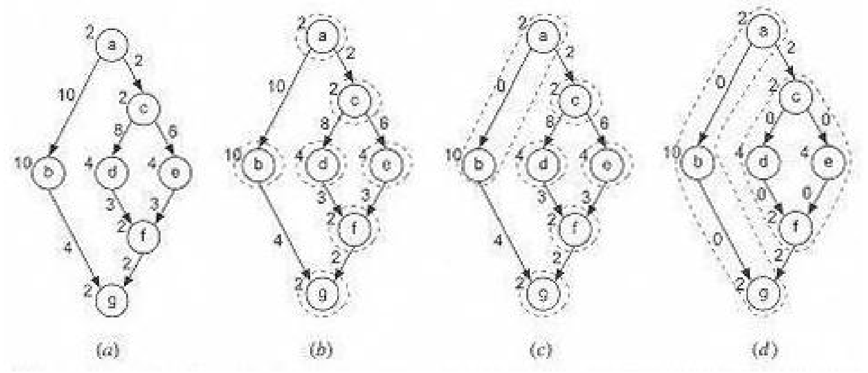


**Figure 3.4** *Clustering of a simple task graph: (a) simple task graph for clustering algorithms; (b) initial clustering; (c) clustering after clusters of nodes a and b have been merged; (d) clustering with only two clusters [15].*

According to [16], an algorithm in this group maps the task in a given graph to an unlimited number of clusters (see Figure 3.4 and Figure 3.5). At each step, the selected tasks for clustering can be any task, not necessarily a ready task. Each iteration refines the previous clustering by merging some clusters. If two tasks are assigned to the same cluster, they will be executed on the same processor. A clustering heuristic require additional steps to generate a final schedule: a cluster merging step for merging the clusters so that the remaining number of clusters equals the number of processors, a cluster mapping step for mapping the clusters on the available processors, and a task ordering step for ordering the mapped task within each processor [10]. More formal definition of clustering as stated in [15] follows.

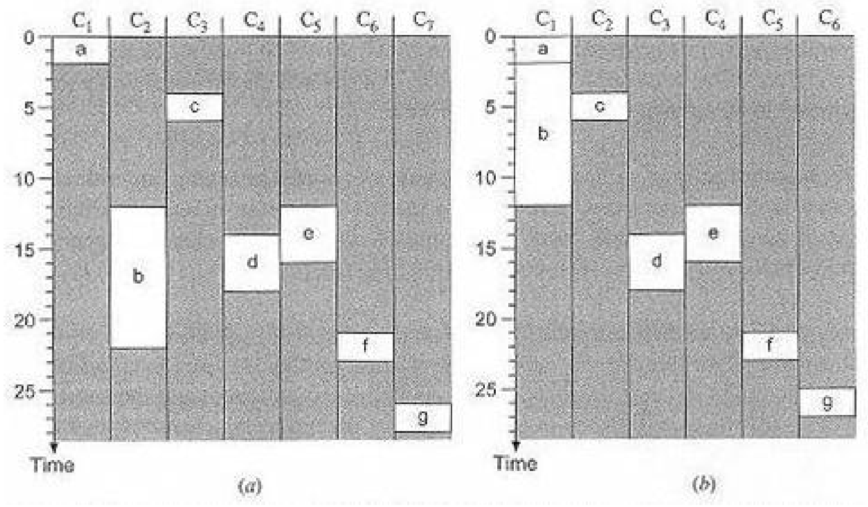


**Figure 3.5** *Implicit schedules of (a) the initial clustering as shown in Figure 3.4(b) and (b) the clustering in Figure 3.4(c) [15].*

**Definition 3.1 (Clustering)** *Let $G = (V, E, w, c)$ be a task graph. A clustering $C$ is a schedule of $G$ on an implicit parallel system $P$ with an "unlimited" number of processors; that is, $|P| = |V|$. The processors $p \in P$ are called clusters.*

## 3.1.3 Task duplication

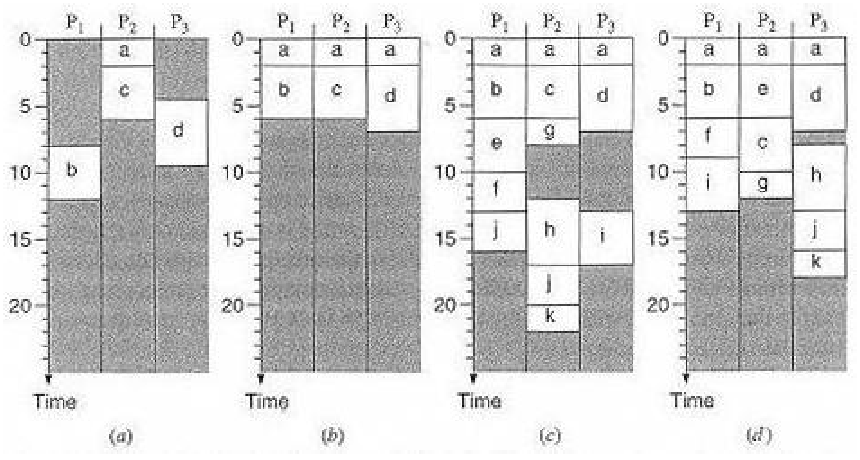


**Figure 3.6** *(a), (b) partial and (c), (d) full schedules for the task graph of Figure 2.1; the schedules of (b), (c) and (d) use task duplication [15].*

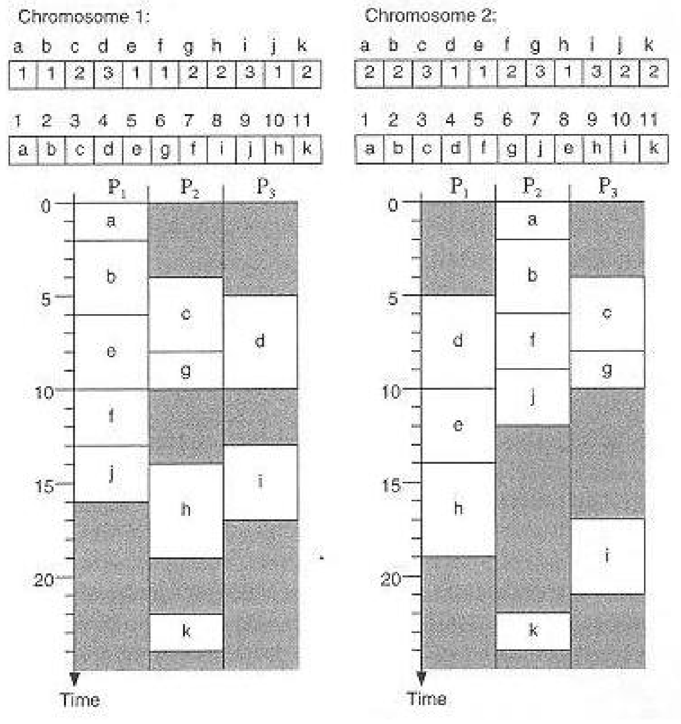


**Figure 3.7** *Two examples of chromosomes (top) that encode both processor allocation and node list for the scheduling of the Figure 2.1 sample task graph on three processors and the schedules they represent (bottom) [15].*

As given in [16], the idea behind duplication based scheduling algorithms (see Figure 3.6) is to schedule a task graph by mapping some of its tasks redundantly, which reduces the interprocess communication overhead. Duplication based algorithms differ according to the selection strategy of the tasks for duplication. The algorithms in this group are usually for an unbounded number of processors and they have much higher complexity values than the algorithms in the other groups.

### 3.1.4 Genetic algorithms

According to [16], guided random search techniques (or randomized search techniques) use random choice to guide themselves through the problem space, which is not the same as performing merely random walks as in the random search methods. These techniques combine the knowledge gained from previous search results with some randomizing features to generate new results. Genetic algorithms (GAs) are the most popular and widely used techniques for several flavors of the task scheduling problem (see Figure 3.7). GAs generate good quality of output schedules; however, their scheduling times are usually much higher than the heuristic based techniques [1]. Additionally, several control parameters in a genetic algorithm should be determined appropriately. The optimal set of control parameters used for scheduling a task graph may not give the best results for another task graph. In addition to GAs, simulated annealing and local search method are the other methods in this group.

## 3.2 Scheduling algorithms

This section presents some of well-known compile-time heterogeneous scheduling algorithms, where the application can be represented by the directed acyclic graph (DAG). Finding an optimal solution for the scheduling problem has been proven to be NP-complete (see the second chapter). These algorithms are therefore rather heuristics because they can be used to find a sub-optimal schedule rather than parsing all possible schedules. Heterogeneous earliest finish time (HEFT) [16], critical path on a processor (CPOP) [16], fast load balance (FLB) [13] (see Figure 3.11 for example schedules of these three algorithms), dynamic level scheduling (DLS) [14], mapping heuristic (MH) [5] and levelized-min time (LMT) [6] algorithms are introduced in the following subsections.

### 3.2.1 HEFT algorithm

According to [16], the HEFT (Heterogeneous earliest finish time) algorithm (see Figure 3.8) is an application scheduling algorithm for a bounded number of heterogeneous processors, which has two major phases: a task prioritizing phase for computing the priorities of all tasks and a processor selection phase for selecting the tasks in the order of their priorities and scheduling each selected task on its "best" processor, which minimizes the task's finish time.

#### Task prioritizing phase

This phase requires the priority of each task to be set with the upward rank value, $rank_u$, which is based on mean computation and mean communication costs. The task list is generated by sorting the tasks by decreasing order of $rank_u$, where the tie-breaking is done randomly. There can be alternative policies for tie-breaking, such as selecting the task whose immediate successor task(s) has higher upward ranks. Since these alternate policies increase the time complexity, we prefer a random selection strategy. It can be easily shown that the decreasing order of $rank_u$ values provides a topological order of tasks, which is a linear order that preserve the precedence constraints.

```
Set the computation costs of tasks and communication costs of edges with mean values.
Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task.
Sort the tasks in a scheduling list by nonincreasing order of $rank_u$ values.
while there are unscheduled tasks in the list do
    Select the first task, $n_i$, from the list for scheduling.
    for each processor $p_k$ in the processor set ($p_k \in \mathbb{Q}$) do
        Compute $EFT(n_i, p_k)$ value using the insertion based scheduling policy.
    Assign task $n_i$ to the processor $p_j$ that minimizes EFT of task $n_i$.
endwhile
```

**Figure 3.8** *The HEFT algorithm [16].*

### Processor selection phase

For most of the task scheduling algorithms, the earliest available time of a processor $p_j$ for a task execution is the time when $p_j$ completes the execution of its last assigned task. However, the HEFT algorithm has an insertion based policy which considers the possible insertion of a task in an earliest idle time slot between two already scheduled tasks on a processor. The length of an idle time slot, i.e., the difference between execution start time and finish time of two tasks that were consecutively scheduled on the same processor, should be at least capable of computation cost of the task to be scheduled. Additionally, scheduling on this idle time slot should preserve precedence constraints.

In the HEFT algorithm, the search of an appropriate idle time slot of a task $n_i$ on a processor $p_j$ starts at the time equal to the ready time of $n_i$ on $p_j$, i.e., the time when all input data of $n_i$ that were sent by $n_i$'s immediate predecessor tasks have arrived at processor $p_j$. The search continues until finding the first idle time slot that is capable of holding the computation cost of task $n_i$. The HEFT algorithm has an $O(e \times q)$ time complexity for $e$ edges and $q$ processors. For a dense graph when the number of edges is proportional to $O(v^2)$, where $v$ is the number of tasks, the time complexity is on the order of $O(v^2 \times q)$.

## 3.2.2 CPOP algorithm

From [16], although the CPOP (Critical path on a processor) algorithm (see Figure 3.9) has the task prioritizing and processor selection phases as in the HEFT algorithm, it uses a different attribute for setting the task priorities and a different strategy for determining the "best" processor for each selected task.

### Task prioritizing phase

In this phase, upward rank ($rank_u$) and downward rank ($rank_d$) values for all tasks are computed using mean computation and mean communication costs. The

CPOP algorithm uses the critical path of a given application graph. The length of this path, $|CP|$, is the sum of the computation costs of the tasks on the path and intertask communication costs along the path. The sum of computation costs on the critical path of a graph is basically the lower bound for the schedule lengths generated by the task scheduling algorithm.

The priority of each task is assigned with the summation of upward and downward ranks. The critical path length is equal to the entry task's priority. Initially, the entry task is the selected task and marked as a critical path task. An immediate successor (of the selected task) that has the highest priority value is selected and it is marked as a critical path task. This process is repeated until the exit node is reached. For tie-breaking, the first immediate successor which has the highest priority is selected.

We maintain a priority queue (with the key of $rank_u + rank_d$) to contain all ready tasks at any given instant. A binary heap was used to implement the priority queue, which has the time complexity of $O(logv)$ for insertion and deletion of a task and $O(1)$ for retrieving the task with the highest priority. At each step, the task with the highest $rank_u + rank_d$ value is selected from the priority queue.

```
Set the computation costs of tasks and communication costs of edges with mean values.
Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task.
Compute $rank_d$ for all tasks by traversing graph downward, starting from the entry task.
Compute $priority(n_i) = rank_d(n_i) + rank_u(n_i)$ for each task $n_i$ in the graph.
$|CP| = priority(n_{entry})$, where $n_{entry}$ is the entry task.
$SET_{CP} = \{n_{entry}\}$, where $SET_{CP}$ is the set of tasks on the critical path.
$n_k \leftarrow n_{entry}$.
while $n_k$ is not the exit task do
    Select $n_j$ where $((n_j \in succ(n_k))$ and $(priority(n_j) == |CP|))$.
    $SET_{CP} = SET_{CP} \cup \{n_j\}$.
    $n_k \leftarrow n_j$.
endwhile
Select the critical path processor $(p_{CP})$ which minimizes $\sum_{n_i \in SET_{CP}} w_{i,j}, \forall p_j \in \mathbb{Q}$.
Initialize the priority queue with the entry task.
while there is an unscheduled task in the priority queue do
    Select the highest priority task $n_i$ from priority queue.
    if $n_i \in SET_{CP}$ then
        Assign the task $n_i$ on $p_{CP}$.
    else
        Assign the task $n_i$ to the processor $p_j$ which minimizes the $EFT(n_i, p_j)$.
    Update the priority queue with the successors of $n_i$, if they become ready tasks.
endwhile
```

**Figure 3.9** *The CPOP algorithm [16].*

## Processor selection phase

The critical path processor, $p_{CP}$, is the one that minimizes the cumulative computation costs of the tasks on the critical path. If the selected task is on the critical path, then it is scheduled on the critical path processor; otherwise it is assigned to a processor which minimizes the earliest execution finish time of the task. Both cases consider an insertion based scheduling policy. The time complexity of the CPOP algorithm is equal to $O(e \times q)$ for $e$ edges and $q$ processors. For a dense graph when the number of edges is proportional to $O(v^2)$, where $v$ is the number of tasks, the time complexity is on the order of $O(v^2 \times q)$.

### 3.2.3 FLB algorithm

As given in [7], the early version of the FLB (Fast load balance) algorithm was presented for homogeneous computing systems. The algorithm is a modified version of the well-known earliest time first (ETF) algorithm. The latest versions of the algorithm are modifications to handle heterogeneous computing systems.

```
Initialize the ready tasks list with the entry task.
while the ready tasks list is not empty do
    Compute the minimum task finish time for each task $n_i$ in the ready tasks list.
    Select the (task $n_i$, machine $p_j$) pair that minimizes the minimum task finish time.
    Assign the task $n_i$ to the corresponding selected machine $p_j$.
    Update the ready tasks list with the tasks, if they become ready tasks.
endwhile
```

**Figure 3.10** *The FLB-f algorithm [13].*

The FLB algorithm [13] utilizes a list called the ready list, which contains all ready tasks to be scheduled at each step. A ready task is defined as a task that has all its parents scheduled. In each step, the minimum task finish time for each ready task in the ready list is computed in a set of two machines for each ready task and the (task, machine) pair that minimizes the minimum task finish time is selected. The machine set for each ready task contains two machines: the first idle machine and the machine where the last message was received. The presented version of the FLB algorithm is called FLB-f (see Figure 3.10), where all machines are examined for each ready task. The complexity of the FLB-f algorithm is $O(v^3 \times q)$, where $v$ is the number of tasks and $q$ is the number of processors.

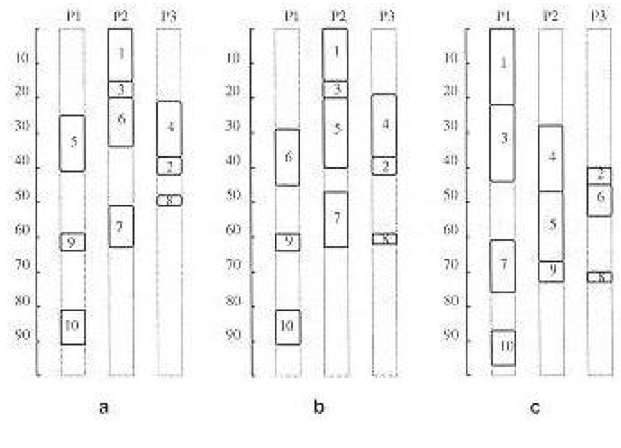


**Figure 3.11** *Scheduling of the task graph in Figure 2.2 with (a) FLB-f (makespan = 91), (b) HEFT (makespan = 91) and CPOP (makespan = 97) [8].*

### 3.2.4 Other algorithms

#### Dynamic level scheduling (DLS)

According to [16], at each step, the dynamic level scheduling (DLS) algorithm [14] selects the (ready node, available processor) pair that maximizes the value of the dynamic level which is equal to $DL(n_i, p_j) = rank_u^s(n_i) - EST(n_i, p_j)$. The computation cost of a task is the median value of the computation costs of the task on the processors. In this algorithm, upward rank calculation does not consider the communication costs. For heterogeneous environments, a new term is added for the difference between the task's median execution time on all processors and its execution time on the current processor. The general DSL algorithm has an $O(v^3 \times q)$ time complexity, where $v$ is the number of tasks and $q$ is the number of processors.

### Mapping heuristic (MH)

From [16], in the mapping heuristic (MH) algorithm [5], the computation cost of a task on a processor is computed by the number of instructions to be executed in the task divided by the speed of the processor. However, in setting the computation costs of tasks and the communication costs of edges before scheduling, similar processing elements (i.e., homogeneous processors) are assumed; the heterogeneity comes into the picture during the scheduling process.

This algorithm uses static upward ranks to assign priorities. (The authors also experimented by adding the communication delay to the rank values.) In this algorithm, the ready time of a processor for a task is the time when the processor has finished its last assigned task and is ready to execute a new one. The MH algorithm does not schedule a task to an idle time slot that is between two tasks already scheduled. The time complexity, when contention is considered, is equal to $O(v^2 \times q^3)$ for $v$ tasks and $q$ processors; otherwise, it is equal to $O(v^2 \times q)$.

### Levelized-min time (LMT)

As written in [16], the levelized-min time (LMT) algorithm [6] is a two-phase algorithm. The first phase groups the tasks that can be executed in parallel using the level attribute. The second phase assigns each task to the fastest available processor. A task in a lower level has higher priority than a task in a higher level. Within the same level, the task with the highest computation cost has the highest priority. Each task is assigned to a processor that minimizes the sum of the task's computation cost and the total communication costs with tasks in the previous levels. For a fully connected graph, the time complexity is $O(v^2 \times q^2)$ when there are $v$ tasks and $q$ processors.

# Chapter 4

# Real-time scheduling

This chapter presents real-time scheduling basics and methods classification. According to [2], for uniprocessor systems, the problem of ensuring that deadline constraints are met has been widely studied, so effective scheduling algorithms that take into account the many complexities that arise in real systems are well understood. By contrast, understanding the trade-offs involved in scheduling independent, periodic real-time tasks on multiprocessor systems is still weak.

## 4.1 Introduction

Research on real-time scheduling has largely focused on the problem of scheduling of recurring processes. The periodic task model of Liu and Layland is the simplest model of a recurring process [11, 12]. From [2], in this model, a task $T$ is characterized by two parameters: a worst-case execution requirement $e$ and a period $p$. Such a task is invoked at each non-negative integer multiple of $p$. Task invocations are also called job releases or job arrivals. Each invocation requires at most $e$ units of processor time and must complete its execution within $p$ time units. A collection of periodic tasks is referred to as a periodic task system and is denoted $\tau$.

We say that a task system $\tau$ is schedulable by an algorithm $A$ if $A$ ensures that the timing constraints of all tasks in $\tau$ are met. $\tau$ is said to be feasible under a class $C$ of scheduling algorithms if $\tau$ is schedulable by some algorithm $A \in C$. An algorithm $A$ is said to be optimal with respect to class $C$ if $A \in C$ and $A$ correctly schedules every task system that is feasible under $C$. When the class $C$ is not specified, it should be assumed to include all possible scheduling algorithms.

Liu and Layland proved [12] that for a set of $n$ periodic tasks with unique periods, a feasible schedule that will always meet deadlines exists if the CPU utilization $U$ is below a specific bound (depending on the number of tasks) which is defined as

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1),$$

where $C_i$ is the computation time, $T_i$ is the release period (with deadline one period later), and $n$ is the number of processes to be scheduled. When the number of processes tends towards infinity this expression will tend towards

$$\lim_{n\to\infty} n(\sqrt[n]{2}-1) = \ln 2.$$

As written in [3], besides representations of task schedules such as timing diagrams (see Figure 4.1 (b)) or Gannt charts (see Figure 4.1 (a)), an elegant, dynamic representation of tasks exists in a multiprocessor system called the scheduling game board (see Figure 4.2 and Figure 4.3). This dynamic representation graphically shows the statuses (remaining computation time and laxity) of each task at a given instant.

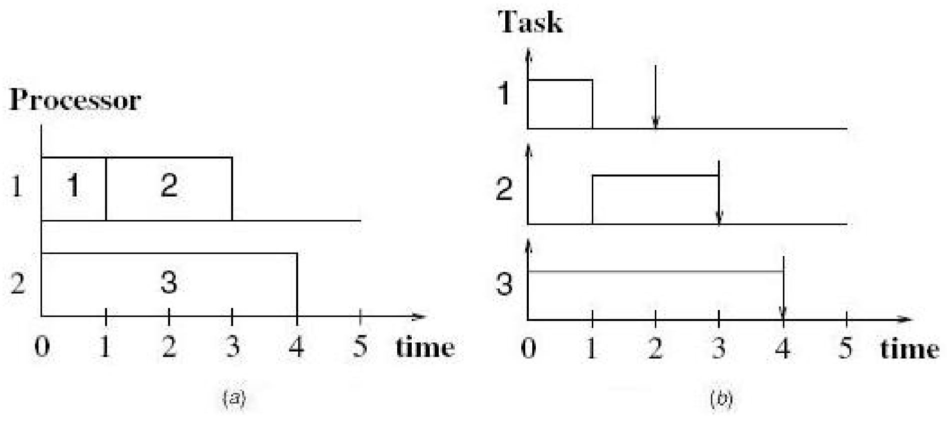
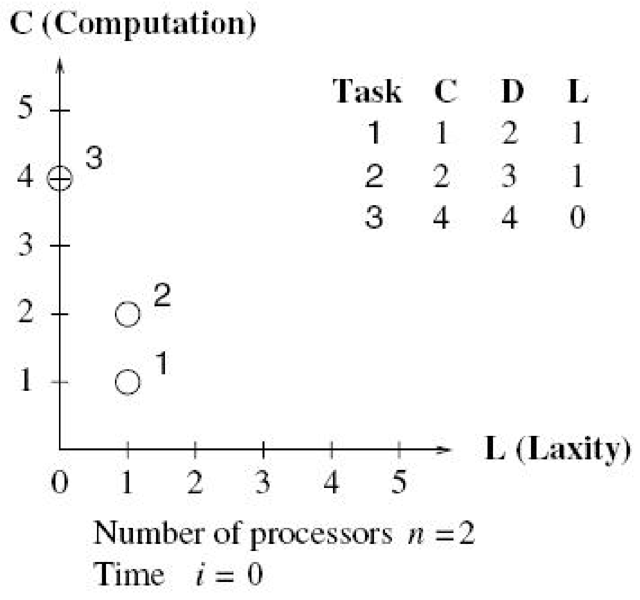


**Figure 4.1** *(a) Gannt chart, (b) Timing diagram [3].*



| Task | C | D | L |
|---|---|---|---|
| 1 | 1 | 2 | 1 |
| 2 | 2 | 3 | 1 |
| 3 | 4 | 4 | 0 |

**Figure 4.2** *Scheduling game board [3].*

## 4.2 Algorithms classification

This section presents several views how to classify real-time algorithms on multiprocessors. As given in [2], traditionally, there have been two approaches for scheduling periodic task systems on multiprocessors:

- global scheduling (see subsection 4.2.1)
- partitioning (see subsection 4.2.2)

In addition to the above approaches, a new "middle" approach is considered in which each job is assigned to a single processor, while a task is allowed to migrate. In other words, inter-processor task migration is permitted only at job boundaries. An active job must be defined before another view of algorithms classification will be stated [2].

**Definition 4.1 (Active job)** *A job is said to be active at time instant $t$ in a given schedule if (i) it has arrived at or prior to time $t$, (ii) its deadline occurs after time $t$, and (iii) it has not yet completed execution.*

Because scheduling algorithms typically execute upon the same processor(s) as the task system being scheduled, according to [2], it is important for such algorithms to be relatively simple and efficient. Most known real-time scheduling algorithms are work-conserving and operate as follows: at each instant, a priority is associated with each active job, and the highest-priority jobs that are eligible to execute are selected for execution upon the available processors.

In work-conserving algorithms, a processor is never left idle while an active job exists (unless migration constraints prevent the task from executing on the idle processor). Because the runtime overheads of such algorithms tend to be less than those of non-work-conserving algorithms, scheduling algorithms that make scheduling decisions on-line tend to be work-conserving. A taxonomy is presented that ranks scheduling schemes along the following two dimensions:

- the complexity of the priority scheme (see subsection 4.2.3)
- the degree of migration allowed (see subsection 4.2.4)

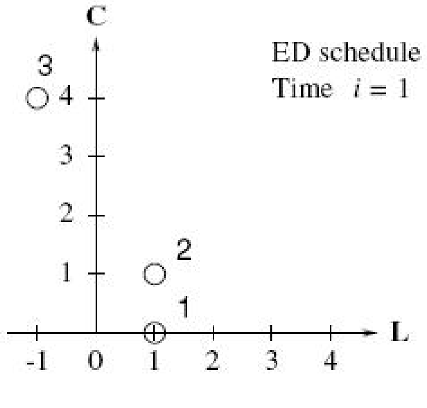


**Figure 4.3** *Game board showing deadline miss [3].*

### 4.2.1 Global scheduling

From [2], in global scheduling, all eligible tasks are stored in a single priority-ordered queue and the global scheduler selects for execution the highest priority tasks from this queue. Unfortunately, using this approach with optimal uniprocessor scheduling algorithms, such as RM (Rate monotonic, see Figure 4.4 for schedule example) and EDF (Earliest deadline first, see Figure 4.5 for schedule example), may result in arbitrarily low processor utilization in multiprocessor systems [4]. However, as given in [2], the research on proportionate fair (Pfair) scheduling has shown considerable promise in that it has produced the only known optimal method for scheduling periodic tasks on multiprocessors.

### 4.2.2 Partitioning

According to [2], in partitioning, each task is assigned to a single processor, on which each of its jobs will execute, and processors are scheduled independently. The main advantage of partitioning approaches is that they reduce a multiprocessor scheduling problem to a set of uniprocessor ones. Unfortunately, partitioning has two negative consequences. First, finding an optimal assignment of tasks to processors is a bin-packing problem, which is NP-hard in the strong sense. Therefore, tasks are usually partitioned using non-optimal heuristics. Second, task systems exist that are schedulable if and only if tasks are not partitioned.

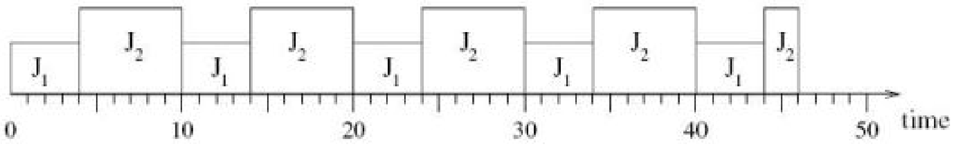


**Figure 4.4** *RM schedule [3].*

### 4.2.3 Priority-based classification

As written in [2], in differentiating among scheduling algorithms according to the complexity of the priority scheme, three categories are considered:

1. Static priorities – A unique priority is associated with each task, and all jobs generated by a task have the priority associated with that task. Thus, if task $T_1$ has higher priority than task $T_2$, then whenever both have active jobs, $T_1$'s job will have priority over $T_2$'s job. An example of a scheduling algorithm in this class is the RM (Rate monotonic, see Figure 4.4 for schedule example) or the DM (Deadline monotonic) algorithm.

2. Job-level dynamic priorities (dynamic but fixed within a job) – For every pair of jobs $J_i$ and $J_j$ , if $J_i$ has higher priority than $J_j$ at some instant in time, then $J_i$ always has higher priority than $J_j$. An example of a scheduling algorithm that is in this class, but not the previous class, is the EDF (Earliest deadline first, see Figure 4.5 for schedule example) algorithm.

3. Unrestricted dynamic priorities (fully dynamic) – No restrictions are placed on the priorities that may be assigned to jobs, and the relative priority of two jobs may change at any time. An example scheduling algorithm that is in this class, but not the previous two classes, is the LLF (Least laxity first) algorithm.

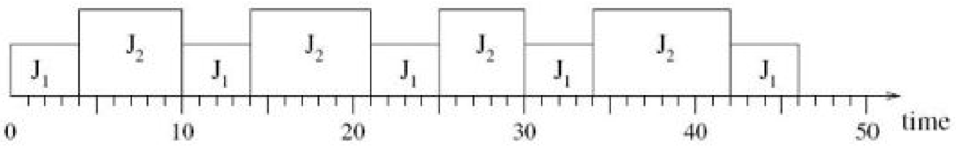


**Figure 4.5** *EDF schedule [3].*

### 4.2.4 Migration-based classification

From [2], interprocessor migration has traditionally been forbidden in real-time systems for the following reasons:

- In many systems, the cost associated with each migration (i.e., the cost of transferring a job's context from one processor to another) can be prohibitive.
- Until recently, traditional real-time scheduling theory lacked the techniques, tools, and results to permit a detailed analysis of systems that allow migration. Hence, partitioning has been the preferred approach due largely to the non-existence of viable alternative approaches.

In differentiating among multiprocessor scheduling algorithms according to the degree of migration allowed, we consider the following three categories:

1. No migration (i.e., task partitioning) – In partitioned scheduling algorithms, the set of tasks is partitioned into as many disjoint subsets as there are processors available, and each such subset is associated with a unique processor. All jobs generated by the tasks in a subset must execute only upon the corresponding processor.

2. Restricted migration (migration allowed, but only at job boundaries, i.e., dynamic partitioning at the job level) – In this category of scheduling algorithms, each job must execute entirely upon a single processor. However, different jobs of the same task may execute upon different processors. Thus, the runtime context of each job needs to be maintained upon only one processor. However, the task-level context may be migrated.

3. Full (unrestricted) migration (i.e., jobs are also allowed to migrate) – No restrictions are placed upon interprocessor migration.

By definition, unrestricted dynamic-priority algorithms are a generalization of job-level dynamic-priority algorithms, which are in turn a generalization of static-priority algorithms. In the multiprocessor systems, unrestricted dynamic-priority scheduling algorithms are strictly more powerful than job-level dynamic-priority algorithms.

# Chapter 5

# Evaluation methods

This chapter presents evaluation methods which are used to compare performance of heterogeneous scheduling algorithms. Testing benchmarks and comparison metrics are introduced from [7] in the following sections.

## 5.1 Testing benchmarks

This section presents some testing benchmarks for heterogeneous scheduling algorithms and brings a short description for each of them. It is very convenient to use random generated graphs in measuring the performance of the tested algorithms. The graphs are generated randomly with various numbers of tasks, various numbers of communication edges and various computation and communication costs. Randomly generated graphs can cover a wide range of applications. It is also convenient to use real application graphs to examine the behavior of the tested algorithms in some real problems.

Five application graphs are used as a testing benchmark to compare performance of heterogeneous scheduling algorithms. These graphs are: random generated graph with different characteristics, Gaussian elimination, Laplace equation, molecular dynamic code and fork-join. Heterogeneous scheduling algorithms should be able to deal with each machine capability and the speed of each communication link in the system. For this reason, the application graph generators are implemented to generate tasks with different computation and communication costs, depending on the machine and communication links used.

### 5.1.1 Random generated graph

In order to be able to generate a random application graph with different characteristics, a random graph generator is implemented, depending on several input parameters. These parameters are:

- number of tasks in a graph ($v$),
- graph levels (GL),

- computation to communication ratio (CCR), which is defined as the ratio of the average communication cost to the average computation cost.

The expected execution cost of each task and the expected communication cost are generated randomly, preserving the CCR. Graphs with a single entry task and a single exit task are considered.

### 5.1.2 Gaussian elimination

Gaussian elimination is a well known problem, where the structure of the graph is predefined as shown in Figure 5.1. The number of tasks $v$ and the number of graph levels GL depend on the matrix size $m$. The graph total number of tasks is equal to:

$$\frac{m^2 + m - 2}{2}.$$

The computation cost of each task and the communication costs are generated randomly, preserving the selected CCR.

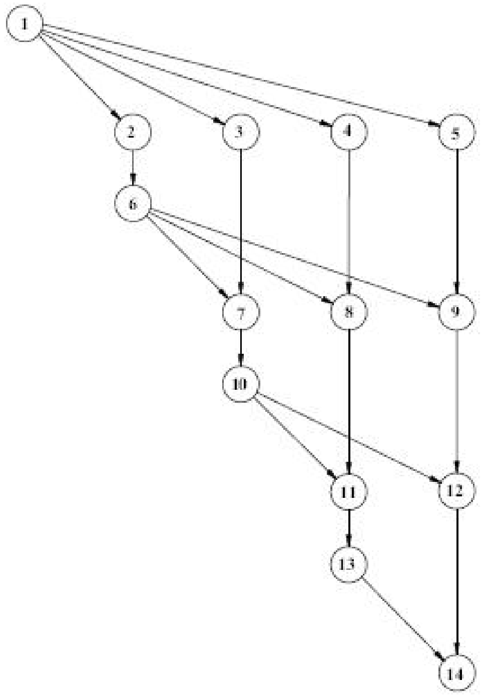


**Figure 5.1** *Gaussian elimination application graph [7].*

### 5.1.3 Laplace equation

The Laplace equation is also a well known problem where the structure of the graph is predefined as shown in Figure 5.2 The number of tasks $v$ is predefined and the number of graph levels GL depends on the number of tasks $v$. The computation cost of each task and the communication costs are generated randomly, preserving the selected CCR.

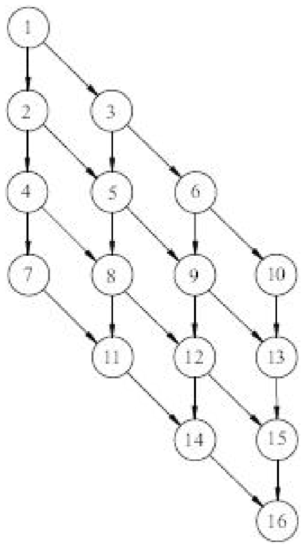


**Figure 5.2** *Laplace equation application graph [7].*

### 5.1.4 Fork-join

Fork-join is a simple application, where the entry task forks its output to $O(v)$ tasks and the output of these $O(v)$ tasks are joined to one task, as shown in Figure 5.3. For each $v$, the number of graph levels GL is selected to preserve the fork and join structure. The computation cost of each task and the communication costs are generated randomly, preserving the selected CCR.

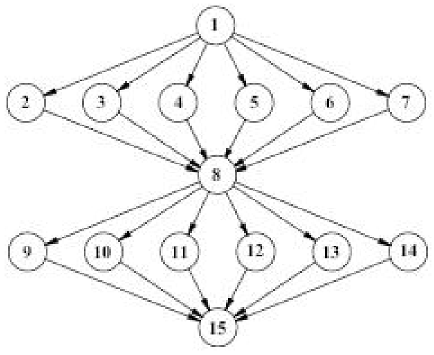


**Figure 5.3** *Fork-join application graph [7].*

### 5.1.5 Molecular dynamic code

Molecular dynamic code is an irregular application, since it has a fixed number of tasks ($v = 41$) and a known graph structure, as shown in Figure 5.4. The computation cost of each task and the communication costs are generated randomly, preserving the selected CCR.

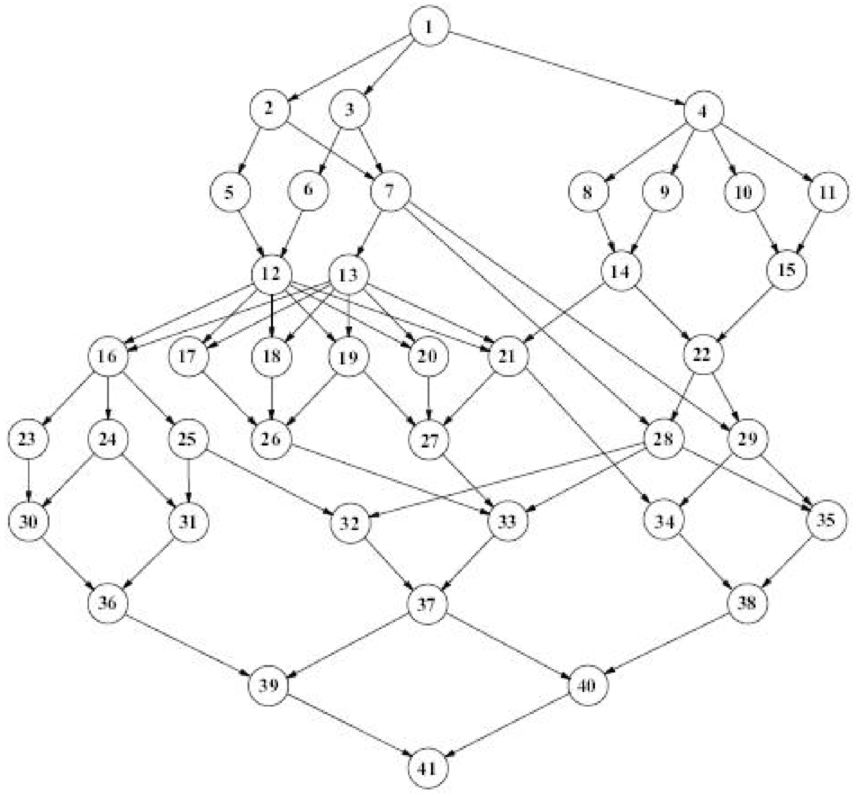


**Figure 5.4** *Molecular dynamic code application graph [7].*

## 5.2 Comparison metrics

This section introduces comparison metrics for heterogeneous scheduling algorithms. According to [7], the performance comparisons can be done based on three comparison metrics: makespan, average schedule length ratio (SLR) and quality of schedule. For each application, the performance comparisons are done with respect to: graph size, number of available machines and computation to communication ratio (CCR). The average communication cost is utilized for homogeneous communication to be able to compare the performance with the other algorithms.

### 5.2.1 Makespan

The makespan (or the schedule length) is the main performance measure and it is defined as:

$$makespan = FT(v_{exit}),$$

where $FT(v_{exit})$ is the finish time of the scheduled exit task.

### 5.2.2 Schedule length ratio

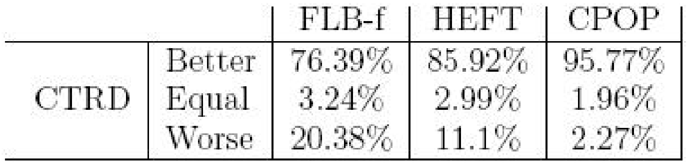
The main performance measure is the makespan. Since a large set of application graphs with different characteristics is used, it is necessary to normalize the schedule length to the lower bound, which is called the schedule length ratio (SLR). The SLR is defined as:

$$SLR = \frac{makespan}{\sum_{v_i \in CT} min_{p_j \in P}\{w_{i,j}\}}.$$

The denominator is the sum of the minimum computation costs of the critical tasks.

### 5.2.3 Quality of schedule

The percentage number of times that an algorithm produced a better, equal or worse schedule is compared to some other algorithms. A table of such percentage comparisons is used in publications as the standard format (see Figure 5.5 for an example). Sometimes a standard table isn't enough for more complicated comparisons. Some kind of graphs can be used in these cases. See [9] for examples of standard tables as well as mentioned graphs.



| | | FLB-f | HEFT | CPOP |
|---|---|---|---|---|
| | Better | 76.39% | 85.92% | 95.77% |
| CTRD | Equal | 3.24% | 2.99% | 1.96% |
| | Worse | 20.38% | 11.1% | 2.27% |

**Figure 5.5** *Percentage comparisons table - schedule quality of random generated graphs with respect to graph size [7].*

# Chapter 6

# Time restrictions

This chapter brings time restrictions into the scheduling problem. The problem of time restrictions is described first and it's also explained what's respecting of such kind of restrictions good for and why it's useful to deal with this problem. After that, the analysis of possible models and methods to solve the problem follows, so the reasons for why to extend some existing models and methods, why to develop new algorithms or which methods should be chosen to extend them are given. Finally, the method is proposed, analyzed and experimentally tested.

## 6.1 Problem description

This section describes the problem of time restrictions. Examples are given to illustrate the usage in real systems.

The only point of view to the scheduling in distributed systems has been for many years or even for many decades the classical scheduling as presented in the chapter 3 of this work where graph models are defined, a fully connected topology is assumed and the main aspect of the schedule quality is mostly its length minimalization. The world is somehow idealized in this approach because it doesn't respect any requirements, problems or restrictions of real systems. People researching this area have used to this approach and some totally different approach could mean something like a small revolution for them.

However, the reality differs from the classical view and it's not ideal in many ways. This difference brings some requirements, problems or restrictions into scheduling systems. Both machines and networks have the real physical characteristics and they aren't homogeneous in today's systems as well as a fully connected network topology can't be assumed. Moreover, both machines and networks become quicker, so it needn't be necessary to take attributes like communication startup cost into account because it tends to zero. This attribute will be discussed further in the next section.

Time restrictions, i.e. intervals when one or more machines are temporarily unavailable, should be something like one step from many possible ways how to bring scheduling in distributed systems closer to the reality. It should represent some kind of real-time characteristics of the system. Time restrictions can be useful to bring into the system the possibility of expressing for example a planned outage, a system maintenance or a situation when intervals of processor time are allocated to active users to make the whole machine available for the particular user.

Although there are used many criteria in the classical scheduling to optimize the resulting schedule, the minimalization of the schedule length is required most often. But in the real system there are some requirements, problems or restrictions, derivable from their real characteristics, which must be respected. It's the most important thing to take them into account in today's systems, so the other criteria become less important than in the classical scheduling and their priority is getting lower off the reality as described before.

## 6.2 Problem analysis

This section analyzes possible models and methods to solve the problem of time restrictions.

The graph models defined in the chapter 3 as an abstraction of the scheduling problem generalize algorithms which are solving scheduling in distributed systems. These models represent the scheduled application and the scheduling system, so it's useful to define similar models which take conditions in the real today's systems into account. The question is, whether new models should be developed or existing models should be extended.

As written in the previous text, described models contain some elements which could potentially loose their importance in today's systems, e.g., both machines and networks become quicker, so it needn't be necessary to take communication startup cost into account because it tends to zero. On the other side, there are still some slower systems, where this information save its value. Moreover, this attribute should nowadays express an amount of initialization work or a distance between processors rather than some connection cost to begin the communication, so the meaning of communication startup cost is shifted somewhere else and purposes of its usage are quite different than defined in the past. However, if new models are developed to respect some additional conditions, they will be proposed very similarly to the existing models because except for the additional information they must still represent the scheduled application and the scheduling system.

Regarding these thoughts, it seems to be more convenient to extend existing models than developing new models. Some elements to represent time restrictions will be probably added into current models to take time restrictions into account. According to extended models, an algorithm respecting time restrictions will be proposed. There is again the question similar to the previous one, whether a new algorithm should be developed or it's more convenient to extend some existing one and if the second possibility will be selected, it's necessary to choose an algorithm to extend it.

The answer isn't so clear in this case as it was before because it's actually not so easy to find some algorithm which will be possible to extend. The extension should be relatively simple, furthermore, the algorithm must have a good time complexity, behave reasonably regardless of the input data and give acceptable results. Otherwise, it's better to develop a new algorithm according to extended models. This algorithm could be similar to some existing one but it must satisfy characteristics described above.

Searching an algorithm convenient to extend it indicated the HEFT algorithm as a good choice. This algorithm is used in many publications as a referential algorithm, which are almost all newly developed algorithms, trying to be a bit better in some sense, compared to. If the attempt to extend the HEFT algorithm went wrong and it was necessary to propose a new algorithm, a similar approach as in the HEFT algorithm would be probably selected to solve the problem of respecting time restrictions in scheduling because of its simplicity and efficiency. The HEFT algorithm have all following characteristics because of which it seems as a good decision to choose this algorithm and try it to extend:

- It's relatively simple, so it can be simply implemented or extended, too.
- It has a good time complexity in comparison to other algorithms.
- It behaves reasonably regardless of the input data.
- It gives acceptable results for general graphs.
- It's used as a referential algorithm in many publications.

## 6.3 Models extension

This section presents an extension of graph models which are modified to respect time restrictions.

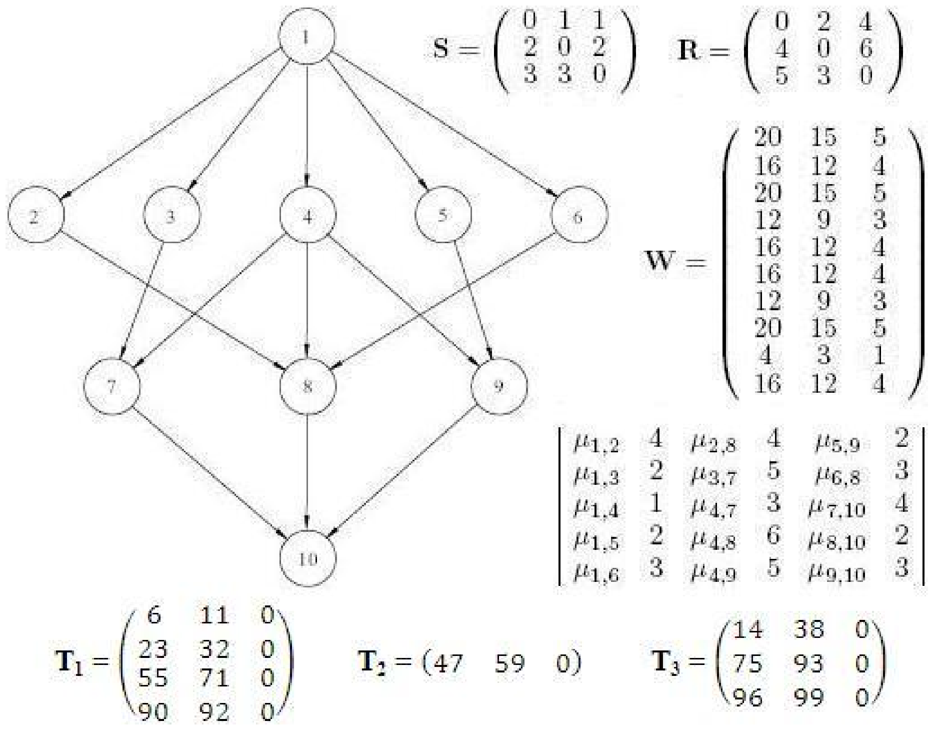
Before a concrete method can be proposed, models must be modified to take time restrictions into account. As decided in the previous section, it's more convenient to extend existing models than developing new models, so some elements to represent time restrictions must be added into current models.

The application model remain unchanged as defined in the section 2.1 because it represents the scheduled application which doesn't contain time restrictions. The application must only respect time restrictions given in the scheduling system, so the computing model must be modified to represent them. The computing model is derived from the computing model as defined in the section 2.2. As shown in Figure 6.1, time restrictions are stored in $p$ matrices $T_k$ of size $t_k \times 3$, where:

- $t_k$ is the number of time restrictions on machine $k$,
- $T_k$ is the matrix for machine $k$,
- the first column of $T_k$ contains the startup time of the time restriction,
- the second column of $T_k$ contains the finish time of the time restriction,

- the third column of $T_k$ contains the percentage availability of machine $k$ in the given time interval.

The third column of $T_k$ is always 0 for purposes of time restrictions as defined before in this chapter. It's present in the matrix because of reusability of the model which can be also useful in algorithms taking some maintenance of the system, e.g., memory cleaning, into account. Every kind of maintenance needs some resources, so the system isn't fully available to run other tasks and it can slow down the scheduled application in time intervals when the maintenance is executed on the machine. This is the reason to limit percentage of available resources to make this part of resources exclusively available for the application.



$$\mathbf{S} = \begin{pmatrix} 0 & 1 & 1 \\ 2 & 0 & 2 \\ 3 & 3 & 0 \end{pmatrix} \quad \mathbf{R} = \begin{pmatrix} 0 & 2 & 4 \\ 4 & 0 & 6 \\ 5 & 3 & 0 \end{pmatrix}$$

$$\mathbf{W} = \begin{pmatrix} 20 & 15 & 5 \\ 16 & 12 & 4 \\ 20 & 15 & 5 \\ 12 & 9 & 3 \\ 16 & 12 & 4 \\ 16 & 12 & 4 \\ 12 & 9 & 3 \\ 20 & 15 & 5 \\ 4 & 3 & 1 \\ 16 & 12 & 4 \end{pmatrix}$$

$$\begin{vmatrix} \mu_{1,2} & 4 & \mu_{2,8} & 4 & \mu_{5,9} & 2 \\ \mu_{1,3} & 2 & \mu_{3,7} & 5 & \mu_{6,8} & 3 \\ \mu_{1,4} & 1 & \mu_{4,7} & 3 & \mu_{7,10} & 4 \\ \mu_{1,5} & 2 & \mu_{4,8} & 6 & \mu_{8,10} & 2 \\ \mu_{1,6} & 3 & \mu_{4,9} & 5 & \mu_{9,10} & 3 \end{vmatrix}$$

$$\mathbf{T}_1 = \begin{pmatrix} 6 & 11 & 0 \\ 23 & 32 & 0 \\ 55 & 71 & 0 \\ 90 & 92 & 0 \end{pmatrix} \quad \mathbf{T}_2 = \begin{pmatrix} 47 & 59 & 0 \end{pmatrix} \quad \mathbf{T}_3 = \begin{pmatrix} 14 & 38 & 0 \\ 75 & 93 & 0 \\ 96 & 99 & 0 \end{pmatrix}$$

**Figure 6.1** *Application model and computation model matrices of heterogeneous computing systems [7] extended to take time restrictions into account.*

## 6.4 TRHEFT algorithm

This section presents the method proposed to take time restrictions into account. The pseudo-code of the main part of the proposed algorithm is included to show how it works with time restrictions and to make it easier to implement the algorithm.

The TRHEFT (Time restriction heterogeneous earliest finish time) algorithm is the result of the attempt to extend the HEFT algorithm to take time restrictions into account. It was necessary to modify only the part of the processor selection phase, when the $EFT(n_i, p_k)$ value is computed for each processor $p_k$ in the processor set using the insertion based scheduling policy, where $n_i$ is the currently selected task. The $EFT(n_i, p_k)$ value must be computed with respect to time restrictions (see Figure 6.2 for details).

```
while there are unscheduled tasks in the list do
    Select the first task, $n_i$, from the list for scheduling.
    for each processor $p_k$ in the processor set ($p_k \in \mathbb{Q}$) do
        Compute $EST(n_i, p_k)$ value using the insertion based scheduling policy.
        Set the index of the actual time restriction $r_k$ to the starting index $s_k$.
        while $r_k < t_k$ do
            if $n_i$ doesn't finish before $r_k$ starts then
                Shift $EST(n_i, p_k)$ after $r_k$ finishes.
                Increase $r_k$.
            else
                Break the while cycle.
        endwhile
        Compute $EFT(n_i, p_k)$ value using $EST(n_i, p_k)$ value.
        Remember the value to increase $s_k$ if the task $n_i$ is assigned to the processor $p_k$.
    endfor
    Assign the task $n_i$ to the processor $p_j$ that minimizes EFT of the task $n_i$.
    Increase the starting index $s_j$ for processor $p_j$ if necessary using the remembered value.
endwhile
```

**Figure 6.2** *The modified part of the processor selection phase in the TRHEFT algorithm. It's an extension of the HEFT algorithm [16] to take time restrictions into account (see Figure 3.8 for pseudo-code of the HEFT algorithm).*

## 6.5 Complexity analysis

According to [16], the HEFT algorithm has an $O(e \times q)$ time complexity for $e$ edges and $q$ processors. For a dense graph when the number of edges is proportional to $O(v^2)$, where $v$ is the number of tasks, the time complexity of the HEFT algorithm is on the order of $O(v^2 \times q)$.

Because the TRHEFT algorithm is an extension of the HEFT algorithm, its time complexity is higher than the time complexity of the HEFT algorithm. If $EFT(n_i, p_k)$ value is computed, at most all time restrictions on the selected machine can be checked, so the TRHEFT algorithm has an $O(v \times (v+t) \times q)$ time complexity for $v$ tasks and $q$ processors, where $t$ is the maximal number of time restrictions on one machine.

## 6.6 Experimental work

This section presents an algorithm to generate time restrictions and the performance testing of the TRHEFT algorithm. Randomly generated task graphs representing the applications are used as a testing suite and the used testing parameters are included to make repeating of tests possible. The performance testing is done based on the average SLR comparison metrics with respect to graph size, number of available machines and CCR.

The TRHEFT algorithm can't be compared with any other scheduling algorithm because none of the other algorithms takes time restrictions into account. This is the reason why the table of percentage comparisons used in publications as the standard format can't be published if time restrictions must be respected. However, graphs to compare the TRHEFT algorithm with the HEFT algorithm are presented below for illustration.

### 6.6.1 Restriction generator

An algorithm to generate random time restrictions is developed and implemented. The input parameters of the algorithm required to generate time restrictions for a target computing system are the following:

- Average number of time restrictions on a machine, $avg\_rest$.
- Maximal deviation of $avg\_rest$, $max\_dev$.
- Probability that a machine will be unavailable for a longer time after the last generated time restriction starts, $prob\_total$, so this machine can't compute any task of the actual application after the beginning of the last generated time restriction is reached.
- Average restriction length to average task length ratio, $arlatlr$. Matrix $W$ is used to compute average task length and its maximal deviation. Then average restriction length and its maximal deviation can be computed from $arlatlr$.

For each machine, a number of time restrictions is computed from the average number of time restrictions and its deviation. Then a number of tasks per restriction is computed from the number of time restrictions, number of tasks and number of machines. Finally, time restrictions are generated. For each time restriction, start time is computed from the number of tasks per restriction, average task length and its deviation first. Then the end time is computed from the average restriction length and its deviation. If it's the last time restriction on a machine, it can be total with some given probability as explained above.

### 6.6.2 Testing parameters

Random graph generator described in [16] is used to generate random task graphs. For all performance evaluation experiments with respect to the number of available machines, a set $num\_pe = \{4, 8, 16, 32\}$ of fully connected machines with different capabilities is used as a target computing system, otherwise 16 fully connected machines with different capabilities are used as a target computing system ($num\_pe = \{16\}$). For all performance evaluation experiments with respect to graph size, a set $v = \{20, 40, 60, 80, 100, 120\}$ of tasks is used, otherwise a set $v = \{40, 80, 120\}$ of tasks is used. The other input parameters of the directed acyclic graph generation algorithm are set with the following values:

- $CCR = \{0.5, 1.0, 2.0\}$

- $\alpha = \{0.5, 1.0, 2.0\}$
- $out\_degree = \{1, 2, 5, 10, max\}$
- $\beta = \{0.5, 1.0, 2.0\}$

These input parameters have the meaning as follows (see [16] for more details):

- Communication to computation ratio, $CCR$, is the ratio of the average communication cost to the average computation cost.
- $\alpha$ is shape parameter of a graph. If $\alpha >> 1.0$, a dense graph (a shorter graph with high parallelism) will be generated. A longer graph with a low parallelism degree can be generated by selecting $\alpha << 1.0$.
- $out\_degree$ is output degree of a node.
- Range percentage of computation costs of processors, $\beta$, represents the heterogeneity factor for processor speeds.

A generator described in the previous section is used to generate time restrictions. Input parameters of this algorithm are set with the following values:

- $avg\_rest = 3$
- $max\_dev = 2$
- $prob\_total = 50.0$
- $arlatlr = 1.0$

For each combination of parameters described above, 100 random task graphs are generated, so performance evaluation experiments with respect to graph size, number of available machines and CCR use 27000, 54000 and 13500 random task graphs respectively as a testing suite.

### 6.6.3 Performance evaluation

For all experiments in this section, average SLR of random generated task graphs is always evaluated with respect to one parameter regardless the other parameters, e.g., if the average SLR of random generated task graphs with respect to graph size should be figured out for 20 tasks, all results for 20 tasks are averaged despite the values of other parameters.

The schedule of the TRHEFT algorithm is expected to be longer than the schedule of the HEFT algorithm because time restrictions take some additional time. However, the TRHEFT algorithm produced a better schedule in most cases of experiments as shown in the graphs below because it's shorter than the schedule produced by the HEFT algorithm. This result is caused by values of parameters in the time restrictions generation algorithm.

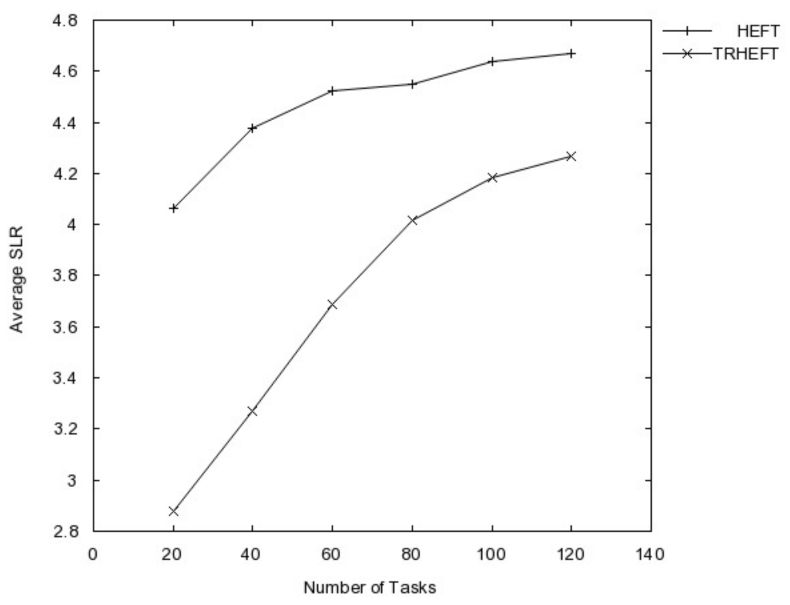


**Figure 6.3** *Average SLR of random generated graphs with respect to graph size (arlatlr set to 1.0).*

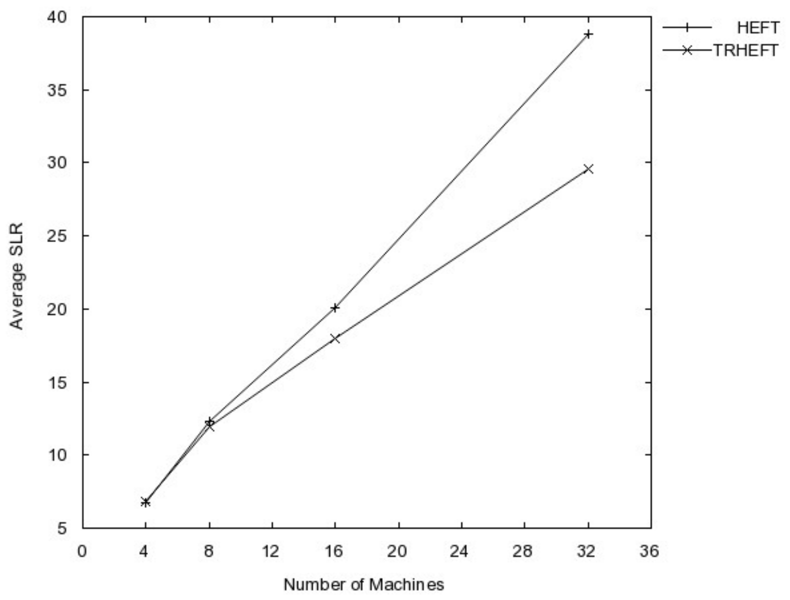


**Figure 6.4** *Average SLR of random generated graphs with respect to number of available machines (arlatlr = 1.0).*

If a small number of time restrictions in the target scheduling system is assumed, then *arlatlr* is the most important parameter because it influences the result the most. It seems that tasks are scheduled differently because of small delays in the form of time restrictions and therefore they are scheduled more equally which brings better usage of machine time in the computing system. It happens because it's profitable to pay the communication cost and compute the task on another machine to make the schedule more optimal. This machine isn't loaded in the HEFT algorithm at all or it isn't loaded as it could be.

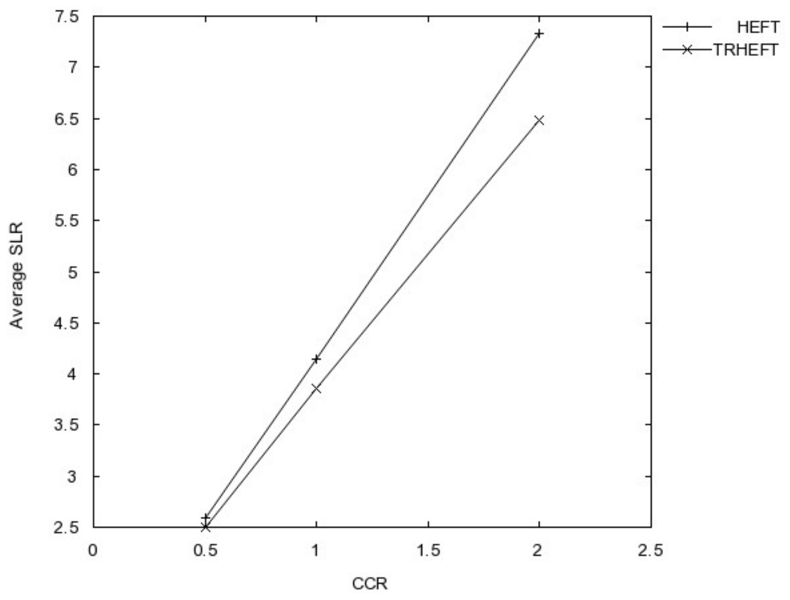


**Figure 6.5** *Average SLR of random generated graphs with respect to CCR (arlatlr = 1.0).*

Next tests are done because of these unexpected results to confirm thoughts and conclusions about the *arlatlr* parameter. Let's set the value of *arlatlr* to 0.1 in the next experiment, which results are presented in the graphs below, so it's 10 times smaller than before. The other parameters remain unchanged.

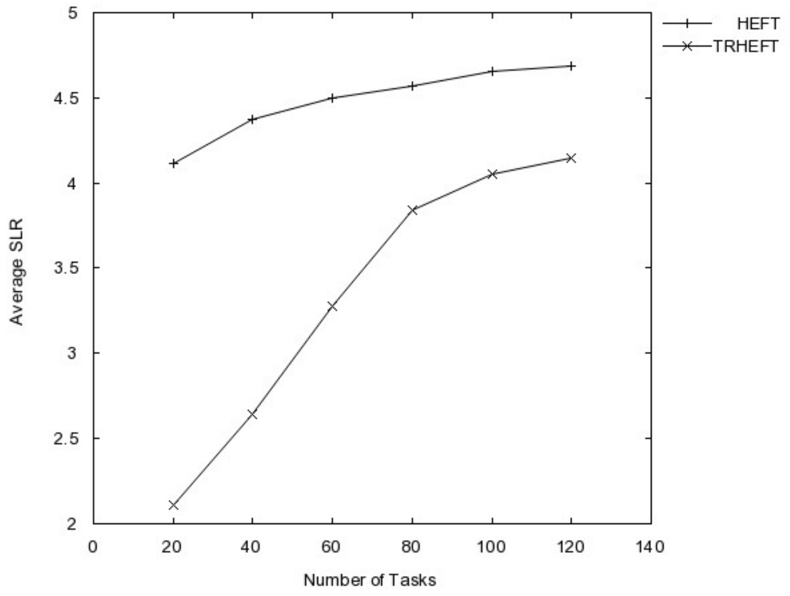


**Figure 6.6** *Average SLR of random generated graphs with respect to graph size (arlatlr set to 0.1).*

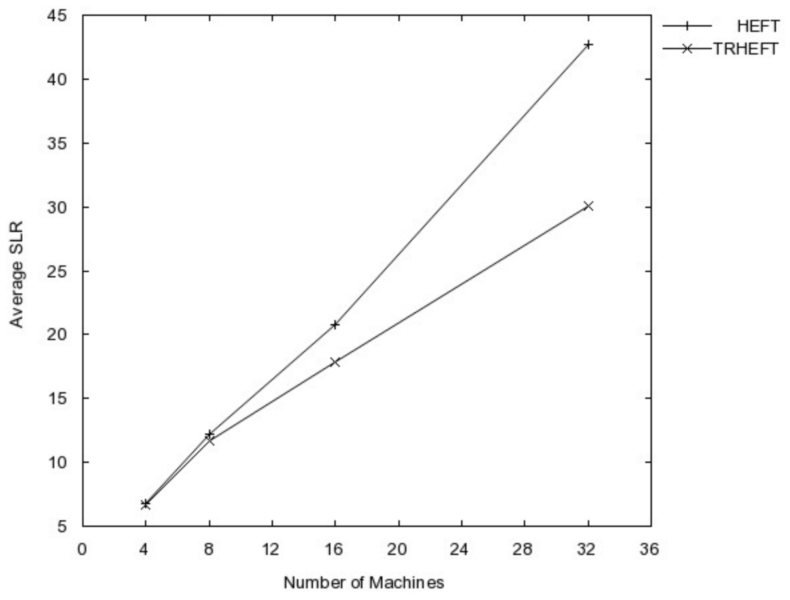


**Figure 6.7** *Average SLR of random generated graphs with respect to number of available machines (arlatlr = 0.1).*

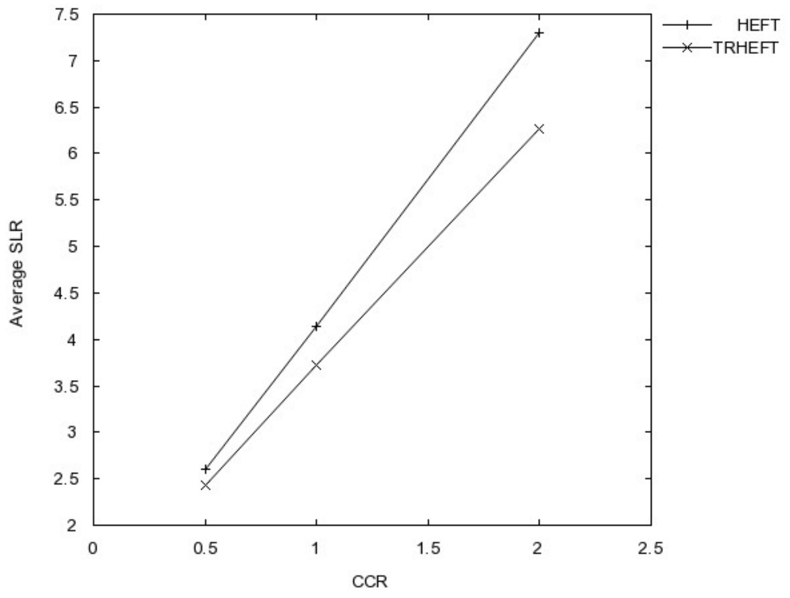


**Figure 6.8** *Average SLR of random generated graphs with respect to CCR (arlatlr = 0.1).*

The results confirm that the smaller *arlatlr* parameter causes the better visibility of the performance difference between the TRHEFT and the HEFT algorithm.

If the *arlatlr* parameter was too small, it's possible that the schedule of the TRHEFT algorithm wouldn't differ from the schedule of the HEFT algorithm and the schedule of the TRHEFT algorithm would be longer than the schedule of the HEFT algorithm because of time restrictions. However, it's too much time-consuming to specify closer when the *arlatlr* parameter is too small, so this problem is out of range of this work, unfortunately.

If time restrictions weren't respected, the TRHEFT algorithm could be used to solve the scheduling problem as defined in the section 2.3 as well as the HEFT algorithm and their results could be compared. Then the TRHEFT algorithm uses small delays to make resulting schedules better. These delays are generated before the algorithm starts, so their values are fixed. Let's repeat previous experiment with the same values of parameters ($arlatlr = 0.1$). Following tables contain percentage comparisons of the TRHEFT algorithm with the HEFT algorithm:

| | | HEFT |
|---|---|---|
| | Better | 97.03% |
| TRHEFT | Equal | 0.0% |
| | Worse | 2.97% |

**Table 6.1** *Schedule quality of random generated graphs with respect to graph size.*

| | | HEFT |
|---|---|---|
| | Better | 81.95% |
| TRHEFT | Equal | 0.0% |
| | Worse | 18.05% |

**Table 6.2** *Schedule quality of random generated graphs with respect to available machines.*

| | | HEFT |
|---|---|---|
| | Better | 96.43% |
| TRHEFT | Equal | 0.0% |
| | Worse | 3.57% |

**Table 6.3** *Schedule quality of random generated graphs with respect to CCR.*

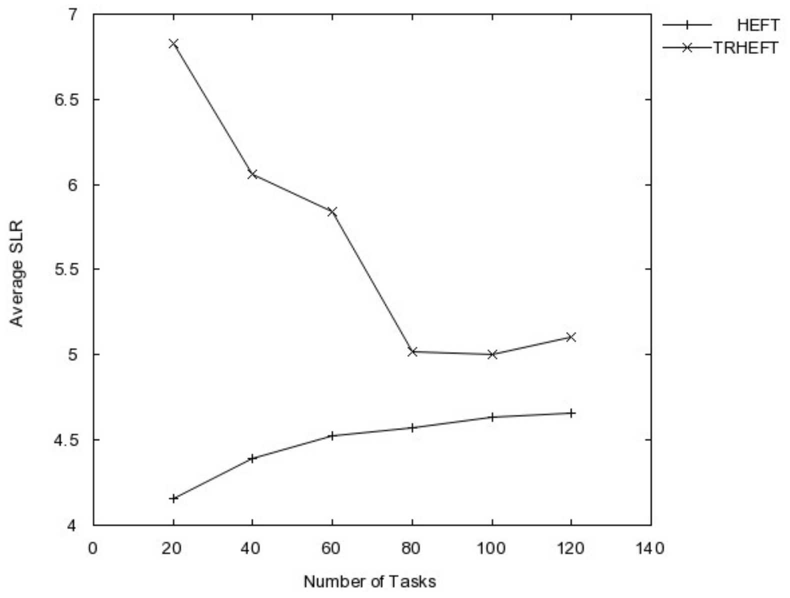


**Figure 6.9** *Average SLR of random generated graphs with respect to graph size (arlatlr set to 10.0).*

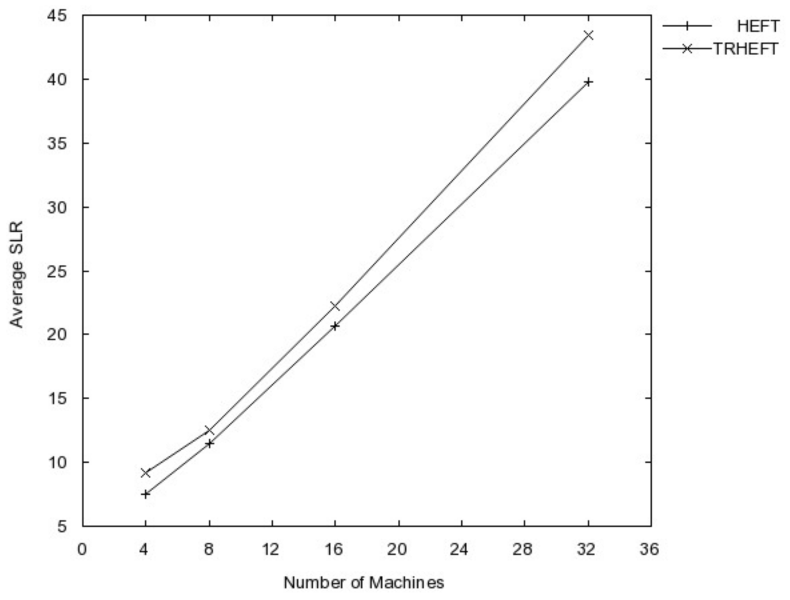


**Figure 6.10** *Average SLR of random generated graphs with respect to number of available machines (arlatlr = 10.0).*

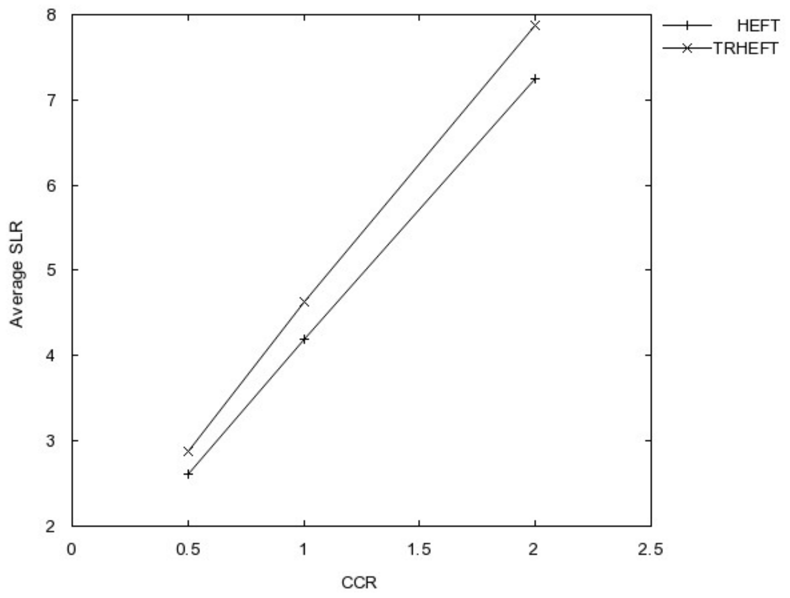


**Figure 6.11** *Average SLR of random generated graphs with respect to CCR (arlatlr set to 10.0).*

Let's set the *arlatlr* parameter value to 10.0 in the next experiment. This test should demonstrate that long time restriction intervals lengthen the schedule produced by the TRHEFT algorithm so much that it's worse than the schedule produced by the HEFT algorithm. The results confirm that the greater *arlatlr* parameter (the longer time restrictions) causes that the schedule produced by the TRHEFT algorithm isn't better than the schedule produced by the HEFT algorithm, so some upper bounds exist for the *arlatlr* parameter value in the TRHEFT algorithm to produce a better schedule than the HEFT algorithm. These bounds differ in most cases for different computing systems depending on values of parameters.

## 6.7 Conclusion

The TRHEFT algorithm produced a better schedule than the HEFT algorithm in some cases as shown in Figures 6.3 – 6.11 which is caused by the value of the *arlatlr* parameter. Tasks can be scheduled differently and more equally because of time restrictions which brings better usage of machine time in the computing system. If time restrictions needn't be respected, the TRHEFT algorithm can be compared with the HEFT algorithm as shown in Tables 6.1 – 6.3. It can be interpreted as a case when the TRHEFT algorithm uses small delays to produce better schedules than produces the HEFT algorithm.

# Chapter 7

# Available data

This chapter brings available data into the scheduling problem. The structure of the chapter is very similar to the previous chapter, so the problem of available data is described first and it's also explained what's respecting of such kind of restrictions good for and why it's useful to deal with this problem. After that, the analysis of possible models and methods to solve the problem follows, so the reasons why to extend some existing models and methods or why to develop new algorithms are given as well as which methods should be chosen to extend them. Finally, two methods are proposed, analyzed and experimentally tested.

## 7.1 Problem description

This section describes the problem of available data. Examples are given to illustrate the usage in real systems.

The section 6.1 describes, how today's real systems differ from the classical scheduling, so there are answered many questions which could be discussed in this section, too. Excepting time restrictions, available data are another step how to bring scheduling in distributed systems closer to the reality. It should represent data which is arising somewhere at some time and it's needed there or somewhere else. This approach is more interesting and more useful in real-time than if data are available at the beginning of the computation as it's used in the usual model.

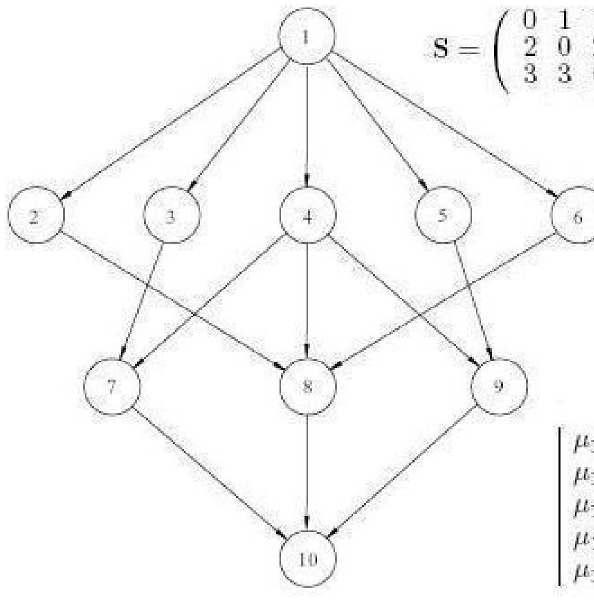
Although classical criteria as the minimalization of the schedule length can't be ignored in systems where some data is arising, it's more important than anything else to take available data into account. Sensor networks must be mentioned as an example of usage because this term appears more and more.

## 7.2 Problem analysis

This section should analyze possible models and methods to solve the problem. However, all these issues are discussed in the previous chapter, so see the section 6.2 for more details. The HEFT algorithm could be probably modified even better to take available data into account than to respect time restrictions because it seems to be easier to add some data into the system than to restrict the availability of the system itself.

## 7.3 Models extension

This section presents an extension of graph models which are modified to respect available data.



$$\mathbf{S} = \begin{pmatrix} 0 & 1 & 1 \\ 2 & 0 & 2 \\ 3 & 3 & 0 \end{pmatrix} \quad \mathbf{R} = \begin{pmatrix} 0 & 2 & 4 \\ 4 & 0 & 6 \\ 5 & 3 & 0 \end{pmatrix}$$

$$\mathbf{W} = \begin{pmatrix} 20 & 15 & 5 \\ 16 & 12 & 4 \\ 20 & 15 & 5 \\ 12 & 9 & 3 \\ 16 & 12 & 4 \\ 16 & 12 & 4 \\ 12 & 9 & 3 \\ 20 & 15 & 5 \\ 4 & 3 & 1 \\ 16 & 12 & 4 \end{pmatrix}$$

$$\begin{vmatrix} \mu_{1,2} & 4 & \mu_{2,8} & 4 & \mu_{5,9} & 2 \\ \mu_{1,3} & 2 & \mu_{3,7} & 5 & \mu_{6,8} & 3 \\ \mu_{1,4} & 1 & \mu_{4,7} & 3 & \mu_{7,10} & 4 \\ \mu_{1,5} & 2 & \mu_{4,8} & 6 & \mu_{8,10} & 2 \\ \mu_{1,6} & 3 & \mu_{4,9} & 5 & \mu_{9,10} & 3 \end{vmatrix}$$

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{D} = \begin{pmatrix} 3 & 64 & 730 \\ 7 & 15 & 1011 \end{pmatrix}$$

**Figure 7.1** *Application model and computation model matrices of heterogeneous computing systems [7] extended to take available data into account.*

Before a concrete method can be proposed, models must be modified to take available data into account. Similarly as in case of time restrictions, it's more convenient to extend existing models than developing new models, so some elements to represent available data must be added into current models.

The application model remain unchanged as defined in the section 2.1 because it represents the scheduled application which doesn't contain available data. The application must only respect available data given in the scheduling system, so the computing model must be modified to represent them. The computing model is derived from the computing model as defined in the section 2.2. As shown in Figure 7.1, information about available data are stored in two matrices.

Boolean matrix T of size $d \times v$ expresses, if node $v_j$ needs data $d_i$ or not. If $T_{i,j}$ is true (1), node $v_j$ needs data $d_i$. If $T_{i,j}$ is false (0), node $v_j$ doesn't need data $d_i$. The source machine identification, the size of data and the time when data are available on the source machine are stored in matrix $D$ of size $d \times 3$, where:

- the first column of $D$ contains the source machine identification for data $d_i$,
- the second column of $D$ contains the time when data $d_i$ are available on the source machine,
- the third column of $D$ contains the size of data $d_i$.

## 7.4 NDAHEFT algorithm

This section presents the first method proposed to take available data into account. Pseudo-code of the proposed algorithm is included to show how it works with available data and to make it easier to implement the algorithm.

The NDAHEFT (Naive data available heterogeneous earliest finish time) algorithm is the first result of the attempt to extend the HEFT algorithm to take available data into account. It wasn't necessary to modify the HEFT algorithm itself, it's only extended to respect available data (see Figure 7.2 for details). It isn't naive in sense of the time complexity but in sense of the schedule length because it's waiting to have all needed data available before the application computing starts.

```
Compute the start time $s$ when all data is available.
Execute the HEFT algorithm.
Recompute the schedule according to $s$.
```

**Figure 7.2** *The NDAHEFT algorithm is an extension of the HEFT algorithm [16] to take available data into account (see Figure 3.8 for pseudo-code of the HEFT algorithm).*

## 7.5 DAHEFT algorithm

This section presents the second method proposed to take available data into account. The pseudo-code of the main part of the proposed algorithm is included to show how it works with available data and to make it easier to implement the algorithm.

```
while there are unscheduled tasks in the list do
    Select the first task, $n_i$, from the list for scheduling.
    Initialize the empty set of data, $D_i$, needed to compute $n_i$.
    for each available data $d_k$ do
        if $n_i$ needs $d_k$ then
            Add $d_k$ into $D_i$.
    endfor
    for each processor $p_k$ in the processor set ($p_k \in \mathbb{Q}$) do
        Count the maximal time, $m_i$, when data from $D_i$ is available.
        Compute $EST(n_i, p_k)$ value using the insertion based scheduling policy.
        if $EST(n_i, p_k) < m_i$ then
            $EST(n_i, p_k) = m_i$
        Compute $EFT(n_i, p_k)$ value using $EST(n_i, p_k)$ value.
    endfor
    Assign the task $n_i$ to the processor $p_j$ that minimizes EFT of the task $n_i$.
endwhile
```

**Figure 7.3** *The modified part of the processor selection phase in the DAHEFT algorithm. It's an extension of the HEFT algorithm [16] to take available data into account (see Figure 3.8 for pseudo-code of the HEFT algorithm).*

The DAHEFT (Data available heterogeneous earliest finish time) algorithm is the next result of the attempt to extend the HEFT algorithm to take available data into account. This method isn't naive in sense of the schedule length as the previous one. It was necessary to modify only the part of the processor selection phase, when the $EFT(n_i, p_k)$ value is computed for each processor $p_k$ in the processor set using the insertion based scheduling policy, where $n_i$ is the currently selected task. The $EFT(n_i, p_k)$ value must be computed with respect to available data (see Figure 7.3 for details).

## 7.6 Complexity analysis

As written in the previous chapter, the HEFT algorithm has according to [16] an $O(e \times q)$ time complexity for $e$ edges and $q$ processors. For a dense graph when the number of edges is proportional to $O(v^2)$, where $v$ is the number of tasks, the time complexity is on the order of $O(v^2 \times q)$.

Because the NDAHEFT algorithm calls the HEFT algorithm and its other work's time complexity isn't higher than the time complexity of the HEFT algorithm, the NDAHEFT algorithm has the time complexity equal to the time complexity of the HEFT algorithm $O(v^2 \times q)$.

Because the DAHEFT algorithm is an extension of the HEFT algorithm, its time complexity is higher than the time complexity of the HEFT algorithm. If $EFT(n_i, p_k)$ value is computed, all available data are checked, so the DAHEFT algorithm has an $O(v \times (v + d) \times q)$ time complexity for $v$ tasks and $q$ processors, where $d$ is the number of available data.

## 7.7 Experimental work

This section presents the available data generation algorithm and the performance comparison of the NDAHEFT algorithm and the DAHEFT algorithm. Randomly generated task graphs representing the applications are used as a testing suite. The performance comparison is done based on the average SLR comparison metrics with respect to graph size, number of available machines and CCR.

Neither the NDAHEFT algorithm nor the DAHEFT algorithm can be compared with any other scheduling algorithm because none of the other algorithms takes available data into account. This is the reason why the table of percentage comparisons used in publications as the standard format is published only to compare the NDAHEFT algorithm and DAHEFT algorithm to each other if available data must be respected. However, graphs to compare NDAHEFT and DAHEFT algorithms with the HEFT algorithm are presented for illustration.

### 7.7.1 Data generator

An algorithm to generate random available data is developed and implemented. The input parameters required to generate the data are the following:

- Probability that some data are available on a machine, *prob_machine*.
- Average number of tasks which need data available on a machine, *avg_nodes*.
- Maximal deviation of average number of tasks which need data available on a machine, *nodes_dev*.
- Average size of data in bytes, *avg_size*.
- Maximal deviation of average size of data in bytes, *size_dev*.
- Average time of data availability, *avg_time*.
- Maximal deviation of average time of data availability, *time_dev*.

For each machine, it's decided first according to some given probability if data is available on this machine or it isn't. If data is available on a machine, it's generated. For each available data, the time of availability, size of data and number of tasks are computed from average values and their deviations given as input parameters. Then tasks are added at random. If some task is already present, it's not added for second time but the number of tasks is decreased.

### 7.7.2 Testing parameters

Similarly to the previous chapter, a random graph generator described in [16] is used to generate random task graphs. For all performance evaluation experiments with respect to the number of available machines, a set $num\_pe = \{4, 8, 16, 32\}$ of fully connected machines with different capabilities is used as a target computing system, otherwise 16 fully connected machines with different capabilities are used as a target computing system ($num\_pe = \{16\}$). For all performance evaluation experiments with respect to graph size, a set $v = \{20, 40, 60, 80, 100, 120\}$ of tasks is used, otherwise a set $v = \{40, 80, 120\}$ of tasks is used. The other input parameters of the directed acyclic graph generation algorithm are set with the following values:

- $CCR = \{0.5, 1.0, 2.0\}$
- $\alpha = \{0.5, 1.0, 2.0\}$
- $out\_degree = \{1, 2, 5, 10, max\}$
- $\beta = \{0.5, 1.0, 2.0\}$

These input parameters have the meaning as described in the section 6.6.2. A generator described in the previous section is used to generate available data. The input parameters of the data generation algorithm are set with the following values:

- $prob\_machine = 25.0$
- $avg\_nodes = 2$

- $nodes\_dev = 1$
- $avg\_size = 4096$
- $size\_dev = 3072$
- $avg\_time = 3000.0$
- $time\_dev = 2000.0$

For each combination of parameters described above, 100 random task graphs are generated, so performance evaluation experiments with respect to graph size, number of available machines and CCR use 27000, 54000 and 13500 random task graphs respectively as a testing suite.

### 7.7.3 Performance evaluation

For all experiments in this section, average SLR of random generated task graphs is always evaluated with respect to one parameter regardless the other parameters (see the section 6.6.3 for an example).

Following table contains percentage comparison of the DAHEFT algorithm with the NDAHEFT algorithm:

| | | NDAHEFT |
|---|---|---|
| | Better | 95.58% |
| DAHEFT | Equal | 0.0% |
| | Worse | 4.42% |

**Table 7.1** *Schedule quality of random generated graphs.*

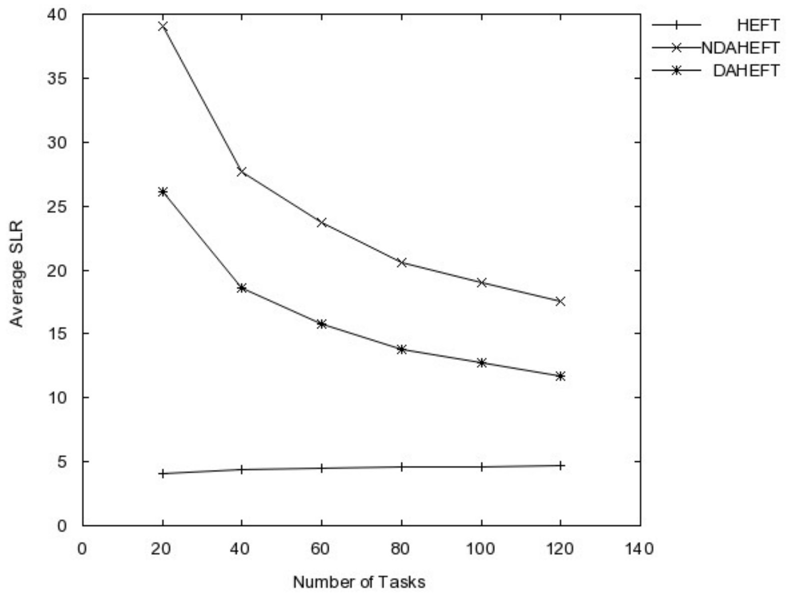


**Figure 7.4** *Average SLR of random generated graphs with respect to graph size.*

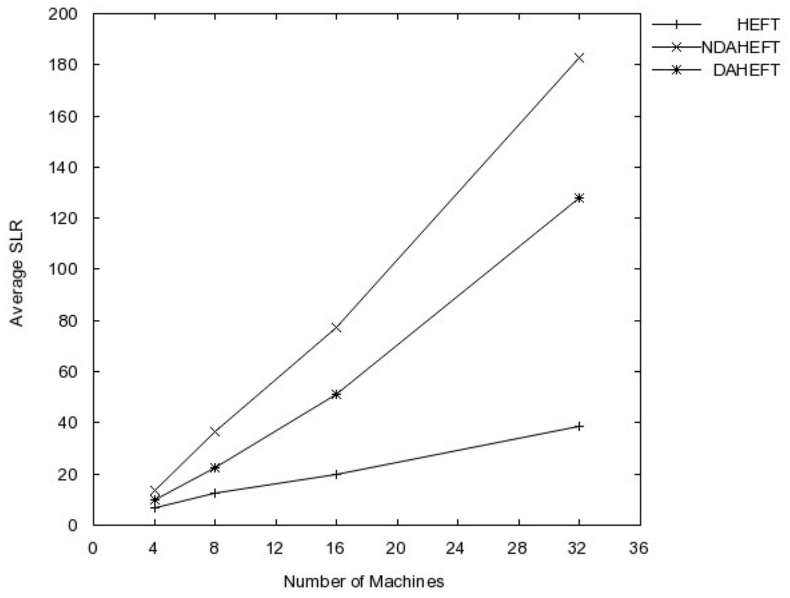


**Figure 7.5** *Average SLR of random generated graphs with respect to number of available machines.*

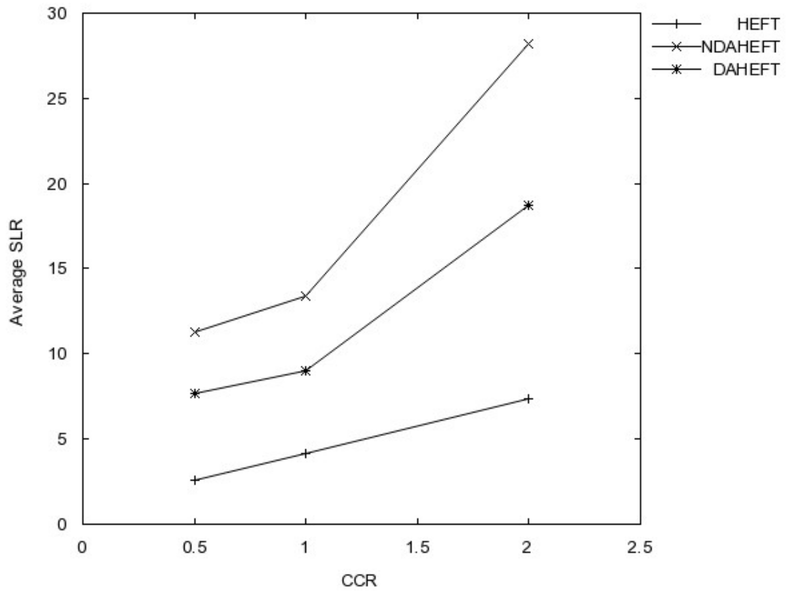


**Figure 7.6** *Average SLR of random generated graphs with respect to CCR.*

The graphs above compare NDAHEFT and DAHEFT algorithms with the HEFT algorithm, although the HEFT algorithm doesn't respect available data.

## 7.8 Conclusion

The schedule of the NDAHEFT algorithm is expected to be longer than the schedule of the DAHEFT algorithm which is expected to be longer than the schedule of the HEFT algorithm. These expectations are fulfilled in the results of experiments. High values of the average SLR for the NDAHEFT algorithm and the DAHEFT algorithm are caused by the fact that it's necessary to wait until some data needed to continue the computation are available. It's clear that the average SLR value depends on the time when data is arising very strongly.

# Chapter 8

# Conclusions

This chapter summarizes the results of the work and the fulfillment of goals as presented in the section with objectives at the beginning of the thesis. Finally, improvements and future work plans are described.

Mechanisms applied to optimal task scheduling in heterogeneous distributed systems are analyzed for both compile-time and real-time cases. Several algorithms which respect given time limits or tasks location are proposed.

## 8.1 Time restrictions

The Time restriction heterogeneous earliest finish time (TRHEFT) algorithm is designed to respect time restrictions which represent intervals when some machine is temporarily unavailable. Because it's an extension of the HEFT algorithm, it has two major phases [16]: a task prioritizing phase and a processor selection phase. The processor selection phase takes time restrictions into account.

The TRHEFT algorithm has an $O(v \times (v + t) \times q)$ time complexity for $v$ tasks and $q$ processors, where $t$ is the maximal number of time restrictions on one machine. This time complexity can be compared with the $O(v^2 \times q)$ time complexity of the HEFT algorithm [16]. The TRHEFT algorithm can't be compared with any other scheduling algorithm because none of the other algorithms takes time restrictions into account. But if the different point of view is selected and it's not required to respect time restrictions, then such comparisons are possible. Instead of time restrictions there are artificially inserted short delays, so the TRHEFT algorithm can be compared with the HEFT algorithm. The TRHEFT algorithm uses mentioned short delays to make schedules better in sense of schedule minimalization and outperforms the HEFT algorithm in some cases. The schedule quality of the proposed algorithm is on average as follows:

| | | HEFT |
|---|---|---|
| | Better | 91.8% |
| TRHEFT | Equal | 0.0% |
| | Worse | 8.2% |

**Table 8.1** *Schedule quality of random generated graphs.*

## 8.2 Available data

Naive data available heterogeneous earliest finish time (NDAHEFT) and Data available heterogeneous earliest finish time (DAHEFT) algorithms are designed to respect available data. Because both algorithms are an extension of the HEFT algorithm, they have two major phases [16]: a task prioritizing phase and a processor selection phase. After the processor selection phase of the NDAHEFT algorithm, all resulting start and end times are increased by the minimal starting time. The processor selection phase of the DAHEFT algorithm takes available data into account.

The NDAHEFT algorithm has an $O(v^2 \times q)$ time complexity for $v$ tasks and $q$ processors which is equal to the time complexity of the HEFT algorithm. The DAHEFT algorithm has an $O(v \times (v + d) \times q)$ time complexity for $v$ tasks and $q$ processors, where $d$ is the number of available data. This time complexity can be compared with the $O(v^2 \times q)$ time complexity of the HEFT algorithm [16]. Neither the NDAHEFT algorithm nor the DAHEFT algorithm can be compared with any other scheduling algorithm because none of the other algorithms takes available data into account. Only NDAHEFT and DAHEFT algorithms can be compared to each other if available data must be respected. The schedule quality of the proposed algorithms is on average as follows:

| | | NDAHEFT |
|---|---|---|
| | Better | 95.58% |
| DAHEFT | Equal | 0.0% |
| | Worse | 4.42% |

**Table 8.2** *Schedule quality of random generated graphs.*

## 8.3 Future work

Time restrictions in the TRHEFT algorithm can be improved to contain the value expressing the percentage availability (or unavailability) of the machine in some time interval. Furthermore, new methods can be developed which respect following requirements or restrictions:

- equal load of machines
- given maximal load of machines
- machines have given time distribution of load
- more applications (one is restricted off another)
- given entry and exit task
- more entry and exit tasks

# Bibliography

[1] Braun T., Siegel H. J., Beck N., Boloni L. L., Maheswaran M., Reuther A. I., Robertson J. P., Theys M. D., Yao B., Hengsen D., Freund R. F.: *A Comparison Study of Static Mapping Heuristics for a Class of Meta-Tasks on Heterogeneous Computing Systems*, Heterogeneous Computing Workshop, 1999, 15–29.

[2] Carpenter J., Funk S., Holman P., Srinivasan A., Anderson J., Baruah S.: *A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms*, Handbook of Scheduling: Algorithms, Models, and Performance Analysis, CRC Press, Boca Raton, Florida, 2004.

[3] Cheng A. M. K.: *Real-Time Systems: Scheduling, Analysis and Verification*, John Wiley & Sons, Hoboken, New Jersey, 2002.

[4] Dhall S., Liu C.: *On a real-time scheduling problem*, Operations Research, Vol. 26, 127-–140, 1978.

[5] El-Rewini H., Lewis T. G.: *Scheduling Parallel Program Tasks onto Arbitrary Target Machines*, Journal of Parallel and Distributed Computing, Vol. **9**, 1990, 138–153.

[6] Iverson M., Ozguner F., Follen G.: *Parallelizing Existing Applications in a Distributed Heterogeneous Environment*, Heterogeneous Computing Workshop, 1995, 93–100.

[7] Hagras T. M. G.: *Task Scheduling in Homogeneous and Heterogeneous Computing Systems*, Prague, 2004.

[8] Hagras T. M. G., Janeček J.: *A High Performance, Low Complexity Algorithm for Compile-Time Task Scheduling in Heterogeneous Systems*, Parallel Computing **31**, 2005, 653–670.

[9] Janeček J., Macejko P., Hagras T. M. G.: *Task Scheduling for Clustered Heterogeneous Systems*, Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN), Innsbruck, 2009, 115–120.

[10] Liou J., Palis M. A.: *A Comparison of General Approaches to Multiprocessor Scheduling*, International Parallel Processing Symposium, 1997, 152–156.

[11] Liu C. L.: *Scheduling algorithms for multiprocessors in a hard real-time environment*, JPL Space Programs Summary 37-60, Vol. **II**, Pasadena, California, 1969, 28–31.

[12] Liu C. L., Layland J. W.: *Scheduling algorithms for multiprogramming in a hard-real-time environment*, Journal of the ACM, Vol. **20**, No. **1**, 1973, 46-–61.

[13] Radulescu A., van Gemund A. J. C.: *Fast and Effective Task Scheduling in Heterogeneous Systems*, Heterogeneous Computing Workshop, 2000, 229–238.

[14] Sih G. C., Lee E. A.: *A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures*, IEEE Transactions in Parallel and Distributed Systems, Vol. **4**, No. **2**, 1993, 175–186.

[15] Sinnen O.: *Task Scheduling for Parallel Systems*, John Wiley & Sons, Hoboken, New Jersey, 2007.

[16] Topcuoglu H., Hariri S., Wu M.-Y.: *Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing*, IEEE Transactions in Parallel and Distributed Systems, Vol. **13**, No. **3**, 2002, 260–274.