MULTI-INSTALLMENT DIVISIBLE LOADS SCHEDULING

Marcin Lawenda

Thesis under guidance of:
Ph.D., Dr. Habil., Maciej Drozdowski, PUT Associate Professor

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
POZNAŃ UNIVERSITY OF TECHNOLOGY
POZNAŃ, POLAND
JULY 2006

Table of Contents

Abstract

Divisible Load Theory (DLT) studies a new model of distributed systems. It assumes that granularity of the computations is small, and that there are no dependencies between the grains of computations. Consequently, the computations, or the load, can be divided into parts of arbitrary size, and these parts can be processed independently in parallel. The sizes of the load parts should be adjusted to the speeds of communication and computation such that processing finishes in the shortest possible time. Divisible load model proved to be a versatile vehicle in modeling distributed systems.

The purpose of this work is to examine the DLT problems taking into considerations the three points of view: single load with single distribution, single load with multiple distributions and multiple loads with single distribution.

In the single load single distribution approach we assume that one volume of the load is given to process and it is distributed once to a particular processor. We are focusing here on the combinatorial nature of this problem.

In the single load multiple distributions we analyze situation where a single load (or divisible computation) can be distributed many times to a given processing unit. We analyze two algorithms: exact branch&bound method and genetic heuristic.

In the multiple loads single distribution situation many loads (or many divisible jobs) can be scattered in single messages to a given processing unit. It is shown that this problem is computationally hard. Special cases solvable in polynomial time are identified.

Chapter 1

Introduction

1.1 Motivation

In the present days the progress in science is extensively supported by the computational methods. It is hard to find a discipline of science where computers are not used. Many of the fields of science (e.g. biochemistry, physics, astronomy) need substantial computational power to achieve desired results. The computations can be realized using *supercomputers*, *clusters*, or more distributed environments like the *grid*. In all cases a specialized software to manage the work is needed.

Looking at the newest Top500 [38] ranking we can note that the number of cluster systems is rapidly growing, while the number of Symmetric Multiprocessing (SMP) systems is decreasing. It confirms general direction of making high performance systems more and more distributed.

The grid system is an example of the environment where providing distributed

computing power and its distributed consumption can be successfully linked. Nevertheless, we should remember that complexity of controlling these systems is incomparably higher because of their heterogeneity. Considering the structure of such systems (the number of processors, features of communications links, loads for processing, collection of results) we can conclude that effective management is not simple. In order to use these resources in an efficient way appropriate management services have to be developed.

Divisible Load Theory (DLT) is a new branch in the scheduling theory which can be applied to solve this kind of problems. DLT is used to represent communications and computations in distributed computer systems, or transportation and production systems. It is assumed in DLT, that the job (e.g. computation or production) can be divided into parts of arbitrary sizes. These parts can be processed in parallel by remote processing elements (computers, factories, etc.). The communication, or transportation time must be taken into account. Divisible load theory emerged as a new paradigm in parallel processing which links scheduling, communication optimization, and distributed computers performance modeling.

Divisible load model finds applications, for example, in: distributed searching for patterns in databases, text, audio, graphic files, in measurement processing, data retrieval systems, some linear algebra algorithms, and simulation [22, 30, 32, 41]. DLT

considerations can be expanded beyond computer systems. Other example applications include transportation systems and production systems.

In the Section 1.2 "Goal and the scope of the work" the problems covered in this thesis are presented more precisely. Section 1.3 provides mathematical formulation of the problem. A short survey of the divisible load theory can be found in the Section "Related work" 1.4.

1.2 The goal and the scope of this work

The main goal of this work is the analysis of different aspects of multi-installment divisible loads scheduling. An introduction to the DLT will be presented. The considered problems will be formulated mathematically. Three aspects of the DLT problem will be taken into consideration:

- Single load and single distribution.

  It is assumed in this case that there is only one divisible load which is sent to a processor in at most one message [18]. Thus, a processor may, or may not, take part in the computations.

- Single load multiple distributions.

  In this problem it is assumed that the load is sent to the processor in many short massages instead of one big message. When load scattering messages are long, also the time spent waiting for the load is longer. Thus, load distribution scheme

with multiple messages has an advantage in activating processors earlier. This method of scattering load in many small chunks is also called *multi-installment processing* of divisible load [6, 17, 20, 19, 24, 43]. Names multi-installment processing and multiple distributions will be used interchangeably in this work.

- Multiple loads single distribution.

  This problem applies to multiple divisible loads processed on the same computing facility. The loads (or tasks) are scheduling entities independent of each other. They represent different distributed computations which are isolated from each other both in the communication and in the computation stage. Scattering the load of a single task is performed in the same way as for single load single distribution case. This means that a task sends at most one message to each of the processors. The difficulty of this problem rests on the coordination of computations and communications of many divisible loads (tasks).

The above problems will be analyzed along the lines of the computational complexity theory [9, 26], computationally hard problems will be identified. Exponential optimization algorithms will be proposed for the hard problems. For the tractable cases polynomial algorithms will be presented. The proposed algorithms will be used to study typical features of the considered problems.

Utilitarian nature of this work consists in developing and testing DLT scheduling algorithms which can have application in real distributed programs e.g.: database and large data files exploration or in digital images and video files processing [22, 30, 32, 41].

1.3 Problem formulation

In this section we present notions and notations used in this work.

We assume a star interconnection (a.k.a. a single level tree) of set $\mathcal{P}$ of processing elements. In the center of the star a computer P_0 called *originator* (or master, server) is located. Originator distributes the load to processing elements $P_1, \ldots, P_m$ (slaves, workers, clients). We will use word *processor* to denote a processing element with CPU, memory, and communication link. Names *processor*, *computer* and *processing element* will denote the same things. Initially the loads are held by an originator processor P_0. All communications involve the originator, and there are no direct communications between processors $P_1, \ldots, P_m$. For simplicity of presentation we assume that originator is communicating only. Were it otherwise, the computing ability of the originator can be represented as an additional processor. To simplify the mathematical models we assume in some cases that the results returning time is negligible. This simplification is not limiting generality of our considerations because results gathering can be included in the model (see e.g. applications [4, 12, 22, 43]).

The computations start only after receiving the whole message with load. We assume that processors have independent communication hardware which allows for simultaneous communication and computations on the previously received load.

Additional constraints maybe imposed on the processors. Due to the existence of other more urgent computations, or maintenance periods, the availability of processor P_i may be restricted to some interval $[r_i, d_i]$. By such a restriction we mean that computations may take place only in interval $[r_i, d_i]$. A message with the load may arrive, or start arriving before r_i. We assume that computations start immediately after the later of two events: r_i, or the load arrival. The computation time must fit between the latter of the above two events, and d_i.

We consider scheduling multiple divisible loads. Each load, which is a separate parallel application, will be called a task. We will be using names *tasks*, and *loads* interchangeably. The set of tasks is $\mathcal{T} = \{T_1, \ldots, T_n\}$. Each task T_j is represented by the volume of load V_j that must be processed.

Tasks in $\mathcal{T}$ may be reordered by the originator to achieve good performance of the computations. Originator splits the loads of the tasks into parts and sends them to processors $P_1, \ldots, P_m$ for remote processing. Only some subset $\mathcal{P}_j \subseteq \mathcal{P}$ of all processors may be used to process task T_j. We will denote by α_{ij} the size of the part of task T_j sent to processor P_i. α_{ij} are expressed in load units (e.g. in bytes). $\alpha_{ij} = 0$ implies that $P_i \notin \mathcal{P}_j$. The sizes of load parts sum up to the task load, i.e.,

$\sum_{i=1}^{m} \alpha_{ij} = V_j$. Not only $\mathcal{P}_j$ is selected by the originator, but also the sequence of activating the processors in $\mathcal{P}_j$ and the division of load V_j into chunks α_{ij}.

Depending on the heterogeneity of the computing environment, three forms of the star system can be distinguished (we use scheduling theory naming convention [13, 35]):

Unrelated processors – communication rates and startup times are specific for the communication link and for the task. Similarly, processor computing rates depend on the processor and task. We will denote by C_{ij} communication rate, and by S_{ij} the startup time, of the link to processor P_i perceived by task T_j. Transferring α_{ij} load units to P_i takes $S_{ij} + C_{ij}\alpha_{ij}$ time units. A_{ij} will denote the processing rate (reciprocal of computing speed) of processor P_i perceived by task T_j. Computing for load α_{ij} lasts $p_{ij} + \alpha_{ij}A_{ij}$, where p_{ij} denotes computation startup time which elapses before computations start. Using processor P_i bears cost $f_{ij} + \alpha_{ij}l_{ij}$. If memory size is limited to B_{ij} load units, then load chunk must not exceed it, i.e. $\alpha_{ij} \leq B_{ij}$. The case of unrelated processors is the most general one. Both the processors and the tasks differ due to variations in the solved problems, the computer or network architecture.

Uniform processors – communication rates C_i, startup times S_i, and computing rates A_i are specific for the processors but are the same for all tasks. In other words $\forall_{T_j} A_{ij} = A_i, C_{ij} = C_i, S_{ij} = S_i$, for $P_i \in \mathcal{P}$. The class of uniform processors is a special case of the more general class of unrelated processors. Uniform processors represent

identical, or similar, parallel programs executed on heterogeneous system. In the case of uniform processor P_j computation on α_{ij} units of load, and communication last $p_i + \alpha_{ij}A_j$, and $S_j + \alpha_{ij}C_j$, respectively. Using processor P_i bears cost $f_i + \alpha_{ij}l_i$. If memory size is limited to B_i load units, then load chunk must not exceed it, i.e. $\alpha_{ij} \leq B_i$.

Identical processors – communication rates, startup times, and computing rates are the same for all processors and tasks. Thus, $\forall_{P_i \in \mathcal{P}} A_i = A, C_i = C, S_i = S$. Identical processors are further specialization of the uniform processors. They represent, e.g., the same parallel program executed in homogeneous environment for different input data sets. For identical processors computation time on α_{ij} units of load is equal to $p + \alpha_{ij}A$, and communication time is $S + \alpha_{ij}C$. Using processor P bears cost $f + \alpha_{ij}l$. If memory size is limited to B load units, then load chunk must not exceed it, i.e. $\alpha_{ij} \leq B$.

In the case of single load problems, i.e. in Chapter 2 and Chapter 3, the notation will be simplified by dropping the subscript indices related to the tasks. Thus, A_i, C_i, S_i, B_i will denote parameters of processor P_i in a heterogenous system, and α_i will denote the size of the load chunk size sent to P_i.

The goal is to schedule divisible load computations such that schedule length $C_{max} = \max_{P_i \in \mathcal{P}'}\{t_i\}$ is minimum, where t_i is the completion time of the computations on P_i, and processing cost (in the case of unrelated processors) $G = \sum_{j=1}^{n} \sum_{i \in \mathcal{P}_j} (f_{ij} +$

$\alpha_{ij} l_{ij}$) is bounded. An alternative formulation of the problem could be to find a schedule with minimum cost G, such that C_{max} is limited. In most of the cases, however, we will consider objective function C_{max}.

Reference hard problems

In this work proofs of **NP**-hardness will be presented. We will provide polynomial time transformations from the following **NP**-complete problems [26], to decision versions of our divisible load scheduling problems:

- PARTITION

  INSTANCE: A finite set $E = \{e_1, \ldots, e_q\}$ of positive integers.

  QUESTION: Is there a subset $E' \subset E$ such that

$$\sum_{i \in E'} e_i = \sum_{i \in E - E'} e_i = \frac{1}{2} \sum_{i=1}^{q} e_i = F? \tag{1.3.1}$$

- PARTITION WITH EQUAL CARDINALITY

  INSTANCE: A finite set $E = \{e_1, \ldots, e_{2q}\}$ of positive integers.

  QUESTION: Is there a subset $E' \subset E$ such that $\sum_{j \in E'} e_j = \sum_{j \in E - E'} e_j = \frac{1}{2} \sum_{j=1}^{2q} e_j = F$, and $|E'| = |E - E'| = q$?

- 3-PARTITION

  INSTANCE: A finite set $E = \{e_1, \ldots, e_{3q}\}$ of positive integers, such that $\sum_{j=1}^{3q} e_j = Fq$ and $F/4 < e_j < F/2$, for $j = 1, \ldots, 3q$.

QUESTION: Can E be partitioned into q disjoint subsets $E_1, \ldots, E_q$ such that $\sum_{j \in E_i} e_j = F$ for $i = 1, \ldots, q$?

1.4 Related work

Divisible Load Model has been proposed to solve a problem of effective partition of computations in a chain network of intelligent sensors [15]. On one hand it is advantageous to distribute the computations, because it provides more computational power. On the other hand distribution of the computations requires communication which bears additional cost. A simple linear model has been proposed to divide the computations optimally [15]. Later on this model has been extended to deal with other networks such as trees [16] and buses [1]. Below we present this model in the context of a star network studied in this work. Assume we have only one load (one task), all processors take part in the computation, and are activated in the order of their numbers. Results are not returned to the originator. It has been shown [7] that in this case computations must stop on all processors at the same moment. This is illustrated in Fig. 1.1.

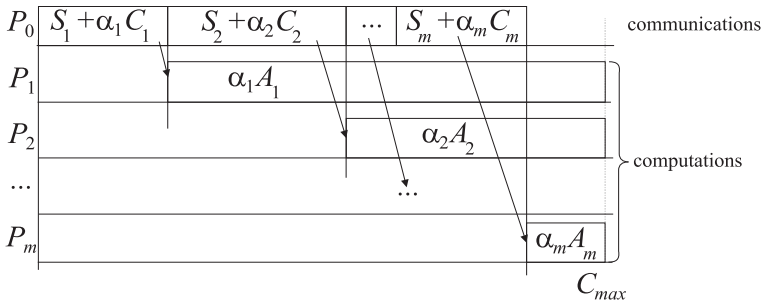

Figure 1.1: Example of the load distribution pattern.

The optimum distribution of the load may be calculated from a system of linear equations:

$$A_i\alpha_i = S_{i+1} + \alpha_{i+1}(C_{i+1} + A_{i+1}) \qquad for\ i = 1, \ldots, m-1 \tag{1.4.1}$$

$$\sum_{i=1}^{m} \alpha_i = V \tag{1.4.2}$$

This simple linear system (1.4.1 – 1.4.2) may be solved in $O(m)$ time due to its diagonal form.

The simplicity of divisible load model attracted attention of scheduling, and parallel computing communities. Many research groups worldwide are interested in developing the DLT these days. One may even say that DLT attracts new researchers every year. Surveys of the divisible load theory can be found in [7, 8, 11, 17, 36].

In the earlier literature, computational tractability of the divisible load model was considered as its great advantage. Yet, some aspects of the combinatorial nature of DLT have been studied before. In [5] it has been shown that the sequence of processor activation in a star network affects schedule length. For a single load it was proved in [5, 7], and independently in [10], that when there are no communication

startup times ($\forall_{P_i} S_i = 0$) the processors have to be activated according to the order of decreasing bandwidth of communication links (i.e. accordingly to the increasing C_i). The case with non-negligible startup times ($\forall_{P_i} S_i > 0$) was studied in [10, 45]. It was determined in [10], and independently in [45], that if communication parameters are identical (i.e. $C_i = C, S_i = S$ for $i = 1, \ldots, m$) then for the shortest schedule the order of decreasing processor speed should be the order of processor activation. This result was obtained under condition that all processors in $\mathcal{P}$ receive non-zero load, and thus, can participate in processing. In [10] it was determined that the problem of divisible load scheduling on a system with startup times and multiple buses is **NP**-hard. The problem of optimizing the cost of a schedule has been studied in [14, 40]. Heuristic rules have been proposed in [14, 40] to select the set of used processors, and determine load assignment, efficiently in terms of the cost and the schedule length.

Scheduling multiple divisible loads has already been considered in DLT for communications without startup times. In [7, 39] it was assumed that the task execution sequence was first-in first-out, processors were uniform, and task computations finished simultaneously. Furthermore, all processors were used by each task. In a multi-job scheme proposed in [7, 39] communications of some task T_j overlap with computations of task T_{j-1} preceding T_j in the execution sequence. This allows to start computations for T_j on some processors $P_1, \ldots, P_{m'}$ immediately after the end of task T_{j-1}. Processors $P_{m'+1}, \ldots, P_m$ are idle until receiving their load share of T_j.

Using the formulae provided in [7, 39] the distribution of the load for T_j can be found in $O(m)$ time, for a given m'. The actual value of m' can be found iteratively in at most m steps. Thus, for a sequence of n tasks the complexity of the algorithm calculating distribution of the load is $O(m^2 n)$.

In [42] the same assumptions on the task sequence, processor selection, simultaneous computation completion, and zero startup time were made. Under the above assumptions a multi-installment load distribution strategy has been proposed to ensure that all processors work continuously on tasks $T_2, \ldots, T_n$. When the overlap of computations on T_{j-1} with the communications of T_j is too short to send the whole load V_j to the processors, and thus avoid idle time (i.e. if $m' < m$), then the load is divided into multiple smaller installments. Since communications are shorter, all processors may receive some load earlier, and may work continuously on T_j. Unfortunately, it was observed in [42] that this strategy does not work for certain combinations of task, and processor parameters. Four heuristics have been proposed in [42]. It was demonstrated by a set of simulations that a multi-installment strategy gives the shortest schedule in most of the cases.

In [29] a probabilistic analysis is given for multiple loads arriving at multiple nodes of a fully-connected network of identical processors. In [33] a steady state of multiple divisible loads executed in an arbitrary network is studied. Sequencing of the communication and computation is ignored in the steady state. Instead of

minimizing schedule length, the total load executed in a unit of time is maximized. Only the fraction of processor time or communication channel bandwidth dedicated to an application has to be determined.

In the earlier publications on multi-installment processing certain assumptions were usually made on the structure of the schedule. For example, messages of equal size were sent to processors in a round-robin fashion [17, 24, 43]. It has been shown [43] that this way of multi-installment processing reduces the length of the schedule in a homogeneous system at most $\frac{e-1}{e}$ times. Unequal load chunk size partitioning has been also proposed, but with a tacit assumption that the set of used processors, and their activation sequence are given and fixed. Furthermore, it was assumed that there are idle times neither in the communication nor in the computations [6, 7, 17]. However, to our best knowledge, the problem of multi-installment divisible load processing with unequal chunk sizes adjusted to the communication and computation speeds, with selection of the set of exploited processors, and selection of their activation sequence has not been studied. This is the subject of Chapter 3 of this thesis.

Memory limitations and single installment communications have been studied in [23, 24, 25, 31, 44]. For a fixed sequence of communications a fast heuristic method has been proposed in [31], and a solution on the basis of linear programming has been given in [23]. A hierarchic memory system has been studied in [24], and a

multi-installment load distribution has been proposed to overcome out-of-core memory speed limitations. In [25] it has been shown that finding optimum divisible load distribution in a system with limited memory sizes and affine communication delay is **NP**-hard. In [44] multi-installment divisible load processing with limited memory has been studied, but the computer system was homogeneous.

The following three chapters deal with different problems analyzed in this work. The first of them (Chapter 2) concerns the problem where a single load is distributed in at most one communication per processor. In Chapter 3, we analyze situation where one load can be distributed many times to a given processing unit. Chapter 4 describes the case of many loads distributed in one message per processor and per load. The work is summarized in Chapter 5. We summarize the main notation at the end of the work.

Chapter 2

Single Load Single Distribution

In this chapter we analyze complexity of scheduling one divisible task, distributed in a single installment. In Section 2.1 we demonstrate that the problem of divisible load scheduling on a star network can be solved in polynomial time for G, and for C_{max} criteria, provided that the set of used processors and the sequence of their activation are given. In Section 2.2 we show that various special cases of these problems are **NP**-hard. Section 2.3 demonstrates that worst case solutions of the problem can be arbitrary bad. Since this chapter studies single task, the notation is simplified by dropping indices related to the multiplicity of the tasks.

2.1 Fixed processor activation sequence

The problem we consider is a bi-criterial optimization problem. The criteria are schedule length C_{max}, and processor usage cost $G = \sum_{i \in \mathcal{P}'}(f_i + \alpha_i l_i)$, where $\mathcal{P}'$ is a set of the exploited processors. This bi-criterial problem can be relaxed to two simpler problems: (i) minimization of C_{max} on condition that $G \leq \overline{G}$, (ii) minimization of

G on condition that $C_{max} \leq \overline{C_{max}}$, where $\overline{G}$ is a predetermined upper bound on the schedule cost, and $\overline{C_{max}}$ is a given upper bound on the schedule length. Both problems can be solved in polynomial time by use of linear programming, provided that the set $\mathcal{P}'$ of used processors and the sequence of their activation is known. Let us consider problem (i) first. We assume that $|\mathcal{P}'| = m'$, and without loss of generality, the sequence of processor activation is $P_1, P_2, \ldots, P_{m'}$. Then, the linear program for (i) is as follows:

minimize C_{max}

subject to:

$$\sum_{k=1}^{i}(S_k + \alpha_k C_k) + p_i + \alpha_i A_i \leq C_{max} \qquad i = 1, \ldots, m' \tag{2.1.1}$$

$$\sum_{k=1}^{i}(S_k + \alpha_k C_k) + p_i + \alpha_i A_i \leq d_i \qquad i = 1, \ldots, m' \tag{2.1.2}$$

$$r_i + p_i + \alpha_i A_i \leq C_{max} \qquad i = 1, \ldots, m' \tag{2.1.3}$$

$$r_i + p_i + \alpha_i A_i \leq d_i \qquad i = 1, \ldots, m' \tag{2.1.4}$$

$$\sum_{j=1}^{m'}(f_j + \alpha_j l_j) \leq \overline{G} \tag{2.1.5}$$

$$0 \leq \alpha_j \leq B_j \qquad j = 1, \ldots, m' \tag{2.1.6}$$

$$\sum_{j=1}^{m'} \alpha_j = V \tag{2.1.7}$$

In the above formulation constraints (2.1.1)-(2.1.4) guarantee that computations are performed in an admissible interval. The left side of inequalities (2.1.1), (2.1.2) is the earliest possible completion time of the computations provided that they are

started immediately after the end of the load transfer. The left side of inequalities (2.1.3), (2.1.4) is the earliest possible completion time of the computations provided that they are started immediately after processor release time. By inequality (2.1.5) total cost of the schedule does not exceed the limit $\overline{G}$. Constraints (2.1.6) ensure that memory buffer sizes are not exceeded, and by (2.1.7) all the load is processed. Let us consider an example.

Example. $m' = 4, V = 20$, parameters of the processor system are the following:

parameter \ processor	P_1	P_2	P_3	P_4
A_i	2	0.5	1	2
B_i	10	10	10	20
C_i	1	0.1	2	2
S_i	1	1	1	2
p_i	0	1	1	0
d_i	10	20	30	200
r_i	0	10	20	20
f_i	1	5	3	2
l_i	0.5	1	0.3	1

The solution for this instance depends on the value of cost limit $\overline{G}$. This is demonstrated for some example values of $\overline{G}$ in the following table:

$\overline{G}$	α_1	α_2	α_3	α_4	C_{max}
$\geq$25.7669	3	10	5.3333	1.3333	26.333
24.25	3	5	7.5	4.5	41.5
24.1334	3	0.00285	7.6666	9.3306	60.656
$<$24.1334	infeasible				

Observe that schedule length increases as the limit put on the costs decreases. For $\overline{G} \geq$ 25.7669 inequality (2.1.5) is ineffective. For $\overline{G} < 24.1334$ the problem is infeasible.

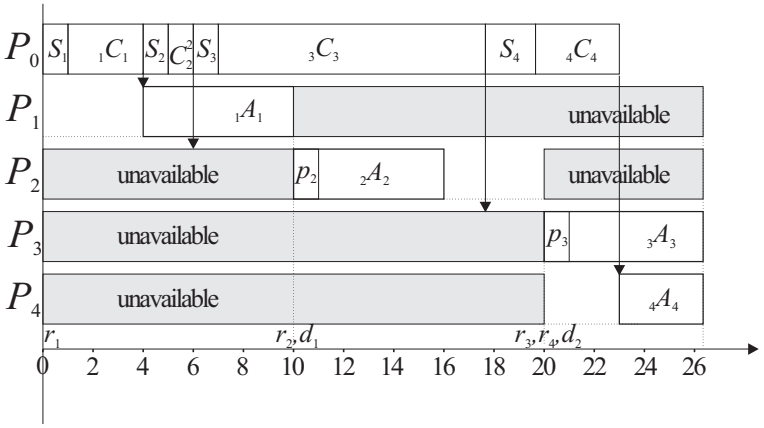

Figure 2.1: Schedule for the example with cost limit $G \geq$25.7669.

The schedule for $\overline{G} \geq$25.7669 is presented in Fig.2.1. The vertical arrows indicate the end of communication from the originator to a certain processor.

Problem (ii) can be also solved in polynomial time by modifying linear program (2.1.1)-(2.1.7). Namely, the roles of the objective function and constraint (2.1.5) must be exchanged. Thus, to solve problem (ii) the minimized objective function should be $\sum_{j=1}^{m'}(f_j + \alpha_j l_j)$, while inequality (2.1.5) should be replaced by $C_{max} \leq \overline{C_{max}}$. Both problems can be solved provided that we know set $\mathcal{P}'$ of active processors and the sequence of their activation. In the next section we will demonstrate that determining them is computationally hard.

2.2 Hard divisible load scheduling problems

In this section we will demonstrate that even restricted cases of scheduling divisible load computations in star networks are computationally hard. All the cases we study

are in class **NP** because it is enough to guess set $\mathcal{P}'$ of the used processors, and the sequence of their activation. Then, the load sizes can be calculated in polynomial time using the methods presented in Section 2.1. We will provide polynomial time transformations from an **NP**-complete problem PARTITION defined in Section 1.3.

We will use DLS abbreviation for divisible load scheduling. Some parameters are not binding for some of the studied cases of DLS. We do not repeat definitions of such parameters, and unless specified otherwise, it is assumed that $B_i = d_i = \infty, C_i = f_i = l_i = p_i = r_i = 0$, for all $P_i \in \mathcal{P}$. In the following w present **NP**-hard cases of DLS problem.

DLS WITH PROCESSOR RELEASE TIMES (DLSPRT)

INSTANCE: Heterogeneous star $\mathcal{P}$, load size V, time interval T, non-zero processor release times $[r_1, \ldots, r_m]$.

QUESTION: Can load V be processed on $\mathcal{P}$ in at most T units of time?

Theorem 1. *Problem DLSPRT is **NP**-hard.*

Proof. The proof is based on the polynomial time transformation from the PARTITION problem. The instance of DLSPRT is constructed in time $O(q)$ as follows: $m = q; A_i = \frac{1}{e_i}, C_i = 0, S_i = e_i, r_i = F$ for $i = 1, \ldots, q; T = F + 1, V = F$.

Suppose PARTITION has a positive answer. Then the processors corresponding to the elements in set E' receive the load in $\sum_{i \in E'} S_i = \sum_{i \in E'} e_i = F = T - 1$ units of

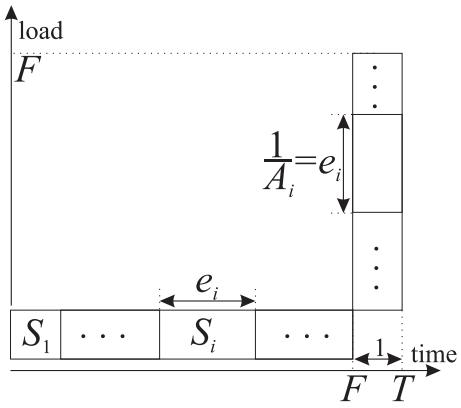

Figure 2.2: Illustration to the proof of Theorem 1

time. Their total speed is $\sum_{i\in E'} \frac{1}{A_i} = \sum_{i\in E'} e_i = F$. Thus, $V = F$ units of load can be processed in the last time unit of the schedule (cf. Fig.2.2).

On the other hand, when the answer to DLSPRT is positive then some set $\mathcal{P}'$ of processors is activated in at most $T = F + 1$ units of time, to process at least F units of the load. Note that all processors become available at $r_i = T - 1$. Since $\forall_{P_i\in\mathcal{P}} S_i \geq 1$, any processor activated in the last time unit of the schedule does not process any load. Thus, the duration of all communications to the processors in $\mathcal{P}'$ does not exceed $T - 1$: $\sum_{P_i\in\mathcal{P}'} S_i = \sum_{P_i\in\mathcal{P}'} e_i \leq T - 1 = F$. The whole load V is processed in the last time unit of the schedule because processors become available at $r_i = T - 1$. Hence, $V = \sum_{P_i\in\mathcal{P}'} \frac{1}{A_i} = \sum_{P_i\in\mathcal{P}'} e_i \geq F$. As $\frac{1}{A_i} = S_i = e_i$, for $i = 1, \ldots, m$, the answer to PARTITION is also positive. □

Before proceeding to the next special case of DLS let us study the amount of load that can be distributed, and processed on a star network with $C_i = 0$, and processors available until finite time d_i, for $i = 1, \ldots, m$. Let us assume that the

sequence of processor activation is fixed, but the set of processors to be activated is yet to be decided. Without loss of generality, let the sequence be $P_1, \ldots, P_m$. Let binary variable $x_i = 1$ denote that processor P_i has been activated in the sequence $P_1, \ldots, P_m$, and $x_i = 0$ that processor P_i is not activated, for $i = 1, \ldots, m$. The amount of load V that can be distributed, and processed in time T is:

$$V = \sum_{i=1}^{m} \frac{x_i d_i}{A_i} - \sum_{1 \le i \le j \le m} x_i x_j \frac{S_i}{A_j} \tag{2.2.1}$$

In the above equation term $\sum_{i=1}^{m} \frac{x_i d_i}{A_i}$ is the amount of load that would be processed by the selected processors provided that they were activated simultaneously at the beginning of the schedule (i.e. communication is timeless). Still, the communication is not timeless. Startup time S_i of the selected processor P_i delays the activation of all processors P_j for $j \ge i$. Therefore, S_i decreases the total load that could be processed by $x_i \sum_{j=i}^{m} x_j \frac{S_i}{A_j}$. Term $\sum_{1 \le i \le j \le m}^{m} x_i x_j \frac{S_i}{A_j}$ in (2.2.1) is the amount of lost load that could not be processed due to the communication delays. Equation (2.2.1) has a graphical interpretation shown in Fig.2.3. The shaded area is the amount of lost load $\sum_{1 \le i \le j \le m}^{m} x_i x_j \frac{S_i}{A_j}$.

DLS WITH PROCESSOR DEADLINES (DLSPD)

INSTANCE: Heterogeneous star $\mathcal{P}$, load size V, finite processor deadlines $[d_1, \ldots, d_m]$.

QUESTION: Can load V be processes on $\mathcal{P}$ before the deadlines $[d_1, \ldots, d_m]$?

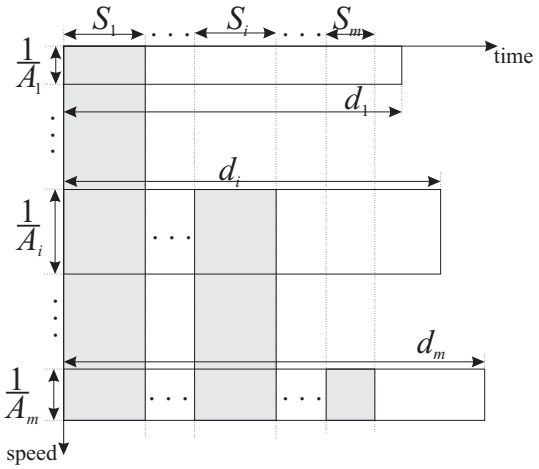

Figure 2.3: Illustration to the proof to Theorem 2

Theorem 2. *Problem DLSPD is **NP**-hard even if the sequence of processor activation is known.*

Proof. We assume that the sequence of processor activation is given. Without loss of generality it is $P_1, \ldots, P_m$. We will prove **NP**-hardness of DLSPD by a polynomial time transformation of PARTITION problem. The transformation is as follows: $S_i = 2e_i, A_i = \frac{1}{2e_i}, d_i = 2F + e_i$, for $i = 1, \ldots, m$. $V = 2F^2$. By substituting these values in equation (2.2.1) we obtain:

$$
\begin{aligned}
&V = \\
&4F\sum_{i=1}^{m} x_i e_i + 2\sum_{i=1}^{m} x_i e_i^2 - 4\sum_{1\le i\le j\le m} x_i x_j e_i e_j = \\
&4F\sum_{i=1}^{m} x_i e_i + 2\sum_{i=1}^{m} x_i^2 e_i^2 - 4\sum_{i=1}^{m} x_i^2 e_i^2 - 4\sum_{1\le i<j\le m} x_i x_j e_i e_j = \\
&4F\sum_{i=1}^{m} x_i e_i - 2\sum_{i=1}^{m} x_i^2 e_i^2 - 4\sum_{1\le i<j\le m} x_i x_j e_i e_j = \\
&2F^2 - 2(\sum_{i=1}^{m} x_i e_i - F)^2 \qquad (2.2.2)
\end{aligned}
$$

In the second line of the above equation we used the fact that $x_i = x_i^2$ for $x_i \in \{0, 1\}$.

By activating the processors corresponding to the elements in set E' in PARTITION problem we have $x_i = 1$ for $i \in E'$, and $x_i = 0$ otherwise, in formula (2.2.2). If there is a positive answer to PARTITION, then $\sum_{i=1}^{m} x_i e_i = \sum_{i \in E'} x_i e_i = F$. Therefore, $V = 2F^2$ units of load are distributed and processed before processor deadlines, as demonstrated in equation (2.2.2). And vice versa, when a feasible schedule exists in which $V = 2F^2$ units of the load is processed, then by inequality (2.2.2), it is possible only if $\sum_{i=1}^{m} x_i e_i = F$, and the answer to PARTITION is positive. □

DLS WITH PROCESSOR STARTUP TIMES (DLSPST)

INSTANCE: Heterogeneous star $\mathcal{P}$, load size V, time interval T, non-zero processor computation startup times $[p_1, \ldots, p_m]$.

QUESTION: Can load V be processed on $\mathcal{P}$ in time at most T?

Theorem 3. *Problem DLSPST is **NP**-hard.*

Proof. This theorem can be proved by a modification of the proof of Theorem 2. In Theorem 2 the maximum computation time available on P_i, provided that communication is timeless, is d_i. In the case of problem DLSPST this amount of time is equal to $T - p_i$. By setting $T = 3F$, and $p_i = F - e_i > 0$ we obtain that $T - p_i = 2F + e_i > 0$. Note that $T - p_i$ here is equal to d_i in the proof of Theorem 2.

If we set other parameters of $\mathcal{P}'$ as in the proof of Theorem 2, then the rest of this proof is analogous to the proof of Theorem 2. □

DLS WITH FIXED PROCESSOR CHARGES (DLSFPC)

INSTANCE: Heterogeneous star $\mathcal{P}$, load size V, time interval T, non-zero charges $[f_1, \ldots, f_m]$ for using the processors, total cost $\overline{G}$.

QUESTION: Can load V be processed on $\mathcal{P}$ in time at most T and cost at most $\overline{G}$?

Theorem 4. *Problem DLSFPC is **NP**-hard.*

Proof. The problem is based on the polynomial transformation of the PARTITION: $m = q, T = 1, \overline{G} = F, V = F, A_i = \frac{1}{e_i}, C_i = S_i = 0, f_i = e_i$, for $i = 1, \ldots, m$. Note that communications are timeless, and processors have one time unit for computations. Thus, the load processed is $V = \sum_{P_i \in \mathcal{P}'} \frac{1}{A_i} = \sum_{P_i \in \mathcal{P}'} e_i$, where $P_i \in \mathcal{P}'$ is the set of activated processors. The cost of activating these processors is $\overline{G} = \sum_{P_i \in \mathcal{P}'} f_i = \sum_{P_i \in \mathcal{P}'} e_i$. Thus, if the cost is $\overline{G} \leq F$, and the size of processed load $V \geq F$, then a positive answer to PARTITION must exist. And vice versa, positive answer to PARTITION implies a positive answer to DLSFPC. □

MAXIMUM SPEED PROBLEM (MS)

INSTANCE: Heterogeneous star $\mathcal{P}$, time interval T, speed R.

QUESTION: Is there a subset $\mathcal{P}'$ of $\mathcal{P}$ with total speed at least R that can be activated in time at most T?

Theorem 5. *MS problem is **NP**-hard.*

Proof. MS problem is in **NP** because NDTM must guess set $\mathcal{P}'$, of processors. Then it is enough to check if $\sum_{i\in\mathcal{P}'} S_i < T$, and $\sum_{i\in\mathcal{P}'} \frac{1}{A_i} > R$.

An instance of the MS Problem can be constructed on the basis of PARTITION instance in the following way: $m = q$; $A_i = \frac{1}{e_i}, C_i = 0, S_i = e_i$ for $i = 1, \ldots, q$. $R = F, T = F$. The instance can be constructed in polynomial time $O(q)$.

Suppose the answer to the PARTITION problem is positive. Then, there is set E' satisfying equation (1.3.1). If we activate the processors corresponding to the elements in set E', then their total speed is $\sum_{i\in E'} \frac{1}{A_i} = \sum_{i\in E'} e_i = F = R$. The time needed to activate these processors is $\sum_{i\in E'} S_i = \sum_{i\in E'} e_i = F = T$. Thus, the set of processors satisfying the conditions of MS exists.

On the other hand, let us assume that the answer to MS problem is positive. Hence, there is some set $\mathcal{P}'$ such that $\sum_{i\in P'} \frac{1}{A_i} = \sum_{i\in P'} e_i \geq R = F$, and $\sum_{i\in P'} S_i = \sum_{i\in P'} e_i \leq T = F$. Consequently, $\sum_{i\in P'} e_i = F$ and the answer to the PARTITION problem is also positive. □

DLS WITH COMMUNICATION STARTUP TIMES (DLSCST)

INSTANCE: Heterogeneous star $\mathcal{P}$, load size V, time interval T, processing rates A_i, startup times S_i, are positive for all processors.

QUESTION: Can load V be processes on $\mathcal{P}$ in time at most T?

Conjecture 6. *Problem DLSCST is **NP**-hard.*

We conjecture that problem DLSCST is **NP**-hard due to its similarity to MS problem: on one hand the activated processors must have sufficient speed to process given volume of load V, on the other hand their work time T is limited.

Now we will analyze two dual versions of DLS with communication startup times: I) Given V find the shortest schedule of length T^*. II) Given schedule length T find the maximum load V^* which can be processed in this time limit. Below we will present a pseudopolynomial algorithm for a special case of divisible load scheduling problem with communication startup times. We will demonstrate that both problems can be solved in pseudopolynomial time if $\forall_{P_i} C_i = 0$. The basic pseudopolynomial algorithm is solving problem II, i.e., instead of minimizing schedule length for a given amount of load V, we will maximize the amount of load processed in some time T. In a dual problem, i.e. for a given V the optimum schedule length can be found by use of a binary search over values of T. First let us establish several facts.

Proposition 7. *For a given time limit T, and set $\mathcal{P}' \subseteq \{P_1, \ldots, P_m\}$ of processors taking part in the computation the maximum load is processed if processors are ordered according to nondecreasing values of $S_i A_i$, for $P_i \in \mathcal{P}'$.*

Proof. The proof is based on an interchange argument. Consider two processors P_i and P_j consecutively activated. Let the communication to the pair start at time $x \leq T - S_i - S_j$. The communications with P_i, P_j are performed in interval $[x, x + S_i + S_j]$. A change in the sequence of T_i, T_j does not influence the schedule for the other

processors. Suppose that the processor sequence is (P_i, P_j). The amount of load processed by the two processors is:

$$V_1 = \frac{T - x - S_i}{A_i} + \frac{T - x - S_i - S_j}{A_j}. \tag{2.2.3}$$

If P_j precedes P_i then the load processed by the two processors is:

$$V_2 = \frac{T - x - S_j}{A_j} + \frac{T - x - S_j - S_i}{A_i}. \tag{2.2.4}$$

From which we get:

$$V_1 - V_2 = \frac{S_j}{A_i} - \frac{S_i}{A_j}. \tag{2.2.5}$$

Thus, the load is greater for the sequence (P_i, P_j) if $S_i A_i < S_j A_j$. □

Proposition 8. *The maximum load V^* which can be processed in time T can be found in $O(m \min\{T, \sum_{i=1}^{m} S_i\})$ time if $C_i = 0$ for $i = 1, \ldots, m$.*

Proof. Let us assume that some sequence of processor activation is fixed, and without loss of generality it is $P_1, \ldots, P_m$. We only have to choose subset of processors to use. The amount of load which can be processed by P_i in time T provided that it finishes communications at time $\tau \geq S_i$ and $C_i = 0$, is $V_i = \max\{0, \frac{T-\tau-p_i}{A_i}\}$. V^* can be calculated as function $V(i, \tau)$ which is maximum load processed by processors selected from set $\{P_1, \ldots, P_i\}$ finishing communications at time τ, for $i = 1, \ldots, m$ and $\tau = 1, \ldots, \min\{T, \sum_{i=1}^{m} S_i\}$. $V(i, \tau)$ can be calculated in $O(mT)$ using the following

recursive equations:

$$V(i,\tau) = \begin{cases} V(i-1,\tau) \quad for \quad \tau < S_i \\ \max \begin{cases} V(i-1,\tau), \\ V(i-1,\tau - S_i) + \max\{0, \frac{T-\tau-p_i}{A_i}\} \end{cases} \quad for \quad \tau \geq S_i \end{cases} \tag{2.2.6}$$

for $i = 1, \ldots, m, \tau = 1, \ldots, \min\{T, \sum_{i=1}^{m} S_i\}$. $V(j,0) = 0$ for $j = 0, \ldots, m$. $V(0,\tau) = 0$, for $\tau = 1, \ldots, \min\{T, \sum_{i=1}^{m} S_i\}$. The maximum load can be found as $V^* = \max_{1 \leq \tau \leq \min\{T, \sum_{i=1}^{m} S_i\}} V(m,\tau)$. Let τ^* satisfy $V^* = V(m,\tau^*)$. The set of processors taking part in the computation can be found by backtracking from $V(m,\tau^*)$ and selecting those processors P_i for which $V(i,\tau) = V(i-1,\tau - S_i) + \max\{0, \frac{T-\tau-p_i}{A_i}\}$. □

Theorem 9. *The minimum schedule length T^* for a given load V can be found in $O((\log V + \log m + \log A_{max} + \log S_{max}) m \min\{S_{max} + A_{max} V, \sum_{i=1}^{m} S_i\})$ time if $C_i = 0$ for $i = 1, \ldots, m$.*

Proof. Let $A_{min} = \min_{i=1}^{m}\{A_i\}, A_{max} = \max_{i=1}^{m}\{A_i\}, S_{max} = \max_{i=1}^{m}\{S_i\}$. In the optimum sequence processors are activated according to the nondecreasing order of $S_i A_i$ by Proposition 7. For a given schedule length T the maximum problem size V^* can be found in $O(m \min\{T, \sum_{i=1}^{m} S_i\})$ time according to Proposition 8. The minimum schedule length can be found by a binary search over the values of T. It remains to show that the number of the calls to the pseudopolynomial time algorithm is limited.

Let $x_i = 1$ if P_i takes part in the computation, and $x_i = 0$ otherwise. The size of the problem which can be processed in time T is (compare (2.2.1)):

$$V = \sum_{i=1}^{m} \frac{T x_i}{A_i} - \sum_{i=1}^{m} \sum_{j=i}^{m} \frac{x_i x_j S_i}{A_j}. \tag{2.2.7}$$

Vector $\overline{x} = [x_1, \ldots, x_2]$ represents a subset of $\{P_1, \ldots, P_m\}$ for which the maximum load is processed in time limit T. Thus, from equation (2.2.7) we conclude that V^* is a piecewise-linear nondecreasing function of T.

Since V^* is a piecewise-linear nondecreasing function of T, the optimum T^* for a given V is a point on one segment or on an intersection of two segments of this piecewise-linear function. Let $\overline{x}$, and $\overline{x}'$ represent two different subsets of processors for which V is maximum at two different schedule lengths. The two linear functions of problem sizes which can be processed in time T by processors corresponding to $\overline{x}$, and $\overline{x}'$ intersect at:

$$T(\overline{x}, \overline{x}') = \frac{\sum_{i=1}^{m} \sum_{j=i}^{m} (x_i x_j - x_i' x_j') \frac{S_i}{A_j}}{\sum_{i=1}^{m} \frac{x_i - x_i'}{A_i}} \tag{2.2.8}$$

Thus, the minimum distance between two different intersections is $\lambda = \frac{1}{A_{max} \sum_{i=1}^{m} \frac{1}{A_i}} \geq \frac{A_{min}}{m A_{max}} \geq \frac{1}{m A_{max}}$. If the difference between two values of T, T' visited in the binary search is smaller than λ, then either the same subset of processors gives maximum load for T and T' or two different subsets of processors are selected for T, T'. In the first case T^* can be found using linear interpolation. In the second case there is one more intersection T'' between T, T', which can be found using (2.2.8), and then

T^* can be found using linear interpolation either to the left or to the right of T''. Since no schedule is longer than $S_{max} + VA_{max}$ and the resolution is λ, the binary search for T^* over T values can be terminated in $O(\log(VA_{max}^2 m + mA_{max}S_{max}))$ which is $O(\log V + \log m + \log A_{max} + \log S_{max})$ steps. The complexity of the whole algorithm is not greater than $O((\log V + \log m + \log A_{max} + \log S_{max})m \min\{S_{max} + VA_{max}, \sum_{i=1}^{m} S_i\})$. □

We conclude that DLS with communication startup times is at most **NP**-hard in the ordinary sense if $\forall_i \;\; C_i = r_i = 0, d_i = \infty$.

2.3 Worst case examples

In this section we consider *the worst cases* that may appear if scheduling decisions ignore certain information. Suppose that we ignore the heterogeneity of the system, and send load parts of equal size to the processors. For instance (compare Fig.2.4a), consider two processors P_1 with parameters $S_1 = 0, C_1 = 0, A_1 = B$, and P_2 with parameters $S_2 = 0, C_2 = 0, A_2 = 1$. We divide the load into two equal chunks of size $\frac{V}{2}$. Resulting schedule has length $\frac{BV}{2}$ but processor P_2 is idle in interval $[\frac{V}{2}, \frac{BV}{2}]$. If we use sizes $\alpha_1 = \frac{V}{B+1}, \alpha_2 = \frac{BV}{B+1}$, then both processors stop computing simultaneously, and schedule length is $\frac{BV}{B+1}$. The ratio of the two schedule lengths is $\frac{B+1}{2}$ which can be arbitrarily big. Hence, in the worst case solutions based on load equal partitioning can be arbitrarily bad in heterogeneous systems. Suppose that we

adjust chunk sizes to the parameters A_i, C_i, but all processors are always used. Let us present another example (see Fig.2.4b). There are two processors with parameters: $S_1 = B, A_1 = 1, C_1 = 1, S_2 = 0, A_2 = 1, C_2 = 1$. If $V < \frac{B}{2}$ then there is no point in using processor P_1 because load of this size may be processed in a shorter time than the communication activating P_1. If we use P_1 then the schedule has length at least B. If we don't, then schedule has length $V(A_2 + C_2) = 2V$. The ratio of the two lengths is at least $\frac{B}{2V}$ which can be arbitrarily big. Thus, if the set of processors is always the same, the resulting schedule can be arbitrarily bad. Suppose that we adjust chunk sizes, and select the processors wisely, but we always use the same sequence $(P_1, \ldots, P_m)$ of processor activation. Let us analyze one more instance (cf. Fig.2.4c). $m = 2, V = 2, S_1 = 0, C_1 = B, A_1 = 1, S_2 = 0, C_2 = A_2 = 0.5$. If we use sequence (P_1, P_2) of processor activation, then the optimum load distribution is $\alpha_1 = \alpha_2 = 1$, and schedule length is $B + 1$. For sequence (P_2, P_1) the optimum distribution is $\alpha_1 = \frac{1}{B+1.5}, \alpha_2 = \frac{2B+2}{B+1.5}$, and schedule length is $\frac{2B+2}{B+1.5}$. The ratio of the two lengths is $\frac{B+1}{2-\frac{1}{B+1.5}}$ which can be arbitrarily big. Thus, schedule can be arbitrary bad if heterogeneity of the system is ignored.

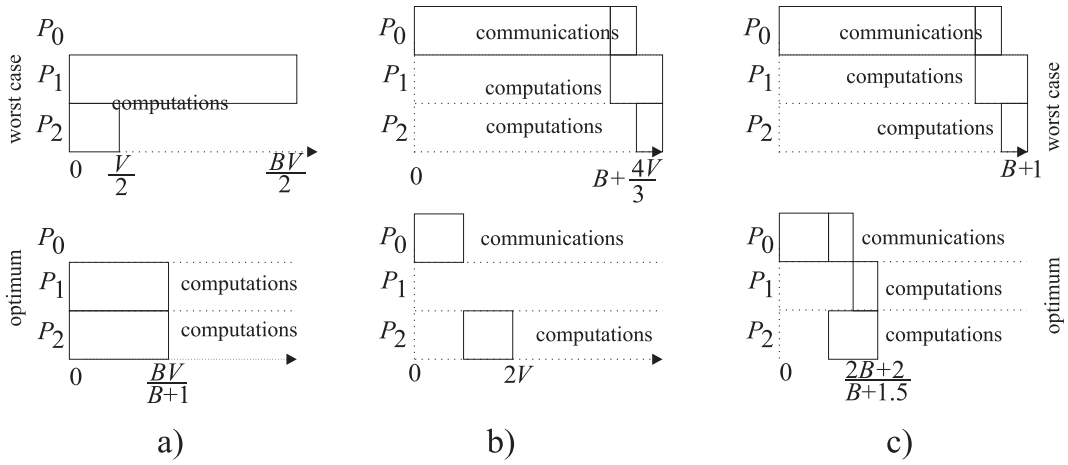

Figure 2.4: The worst case examples. a) ignoring heterogeneity, b) ignoring processor set selection, c) ignoring sequencing of the processor activation.

2.4 Conclusions

In this section we studied the problem of scheduling single divisible load on a star network for the schedule length and the schedule cost criteria. It has been demonstrated that the optimum load distribution can be found in polynomial time by using linear programming, on condition that the set of used processors and the sequence of their activation are known. However, in many cases determining this set is computationally hard. A pseudopolynomial algorithm has been proposed for the case of $\forall_i C_i = r_i = 0, d_i = \infty$ and schedule length criterion.

Chapter 3

Single Load Multiple Distributions

In this section we analyze the problem where one load is distributed to the processors in many installments. Communication delays constitute an important part of the processing time in all distributed algorithms. To reduce the initial waiting for the data, and for initialization of the computations, load is sent in multiple small chunks rather than in a single long message.

The goals of this chapter are twofold: to propose algorithms for the multi-installment divisible load processing including selection of the set and the sequence of processors, and to study influence of the system parameters on the quality of the scheduling problem solutions. For the mathematical simplicity of the considerations we do not include processor P_i startup times p_i, availability constraints $[r_i, d_i]$, and computation cost limit $\overline{G}$. Since only one divisible task is analyzed, we also drop the extended notation related to the multiple tasks.

The rest of this chapter is organized as follows. In Section 3.1.1 we formulate the multi-installment divisible load scheduling problem for heterogeneous systems.

In next Sections 3.1.2 and 3.1.3, respectively, two algorithms are proposed: an optimization branch&bound algorithm, and a heuristic genetic algorithm. The results of computational experiments are presented and discussed in Section 3.1.4. In Section 3.2 we extend our considerations to system with limited memory sizes. Thus, in Section 3.2.1 we formulate the problem, in Sections 3.2.2 and 3.2.3 algorithms are proposed, and in Section 3.2.4 results of the experiments are discussed.

3.1 Without memory limits

3.1.1 Problem Formulation

The problem consists in finding the set of used processors, the sequence of their activation, and the sizes of the load chunks α_j such that the processing time C_{max}, including communication and computations, is the shortest possible. For the sake of conciseness we will mean both selecting the set of processors and their activation sequence while saying processor activation sequence. It is not difficult to realize that in a heterogeneous system the optimum processor activation sequence is not arbitrary. For example, there is no point in sending load to a processor which is very slow, while there are faster ones.

Let z denote the number of load chunks (or messages). If the sequence of processors receiving the load chunks is known then our problem can be reduced to a linear program. Let α_i denote size of chunk i. Let $d_i \in \{1, \ldots, m\}$ be the number of the

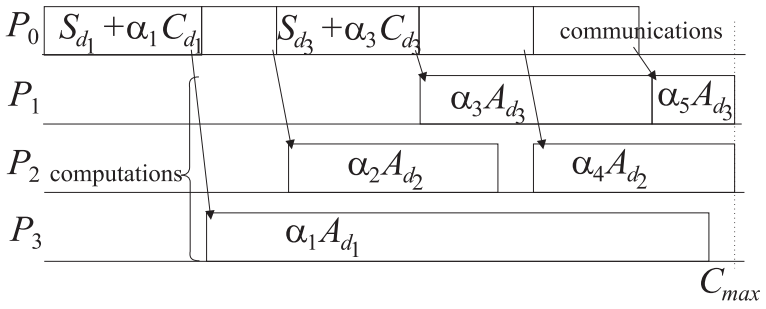

Figure 3.1: Example of load distribution pattern.

processor receiving chunk i. We will denote by $H_i \subseteq \{i, \ldots, z\}$ the set of chunks sent to processor d_i, starting from chunk i. C_{max} denotes schedule length. Fig.3.1 depicts an example schedule with multiple installments. The linear program can be formulated as follows:

minimize C_{max}

subject to:

$$\sum_{j=1}^{i}(S_{d_j} + \alpha_j C_{d_j}) + A_{d_i} \sum_{j \in H_i} \alpha_j \leq C_{max} \quad i = 1, \ldots, z \tag{3.1.1}$$

$$\sum_{i=1}^{n} \alpha_i = V \tag{3.1.2}$$

In constraint (3.1.1) sum $\sum_{j=1}^{i}(S_{d_j} + \alpha_j C_{d_j})$ expresses communication time for chunks $1, \ldots, i$. $A_{d_i} \sum_{j \in H_i} \alpha_j$ is computation time on processor d_i starting from chunk i. Thus, (3.1.1) guarantees that all processors stop computations before the end of the schedule. All work is done by equation (3.1.2). Thus, it is possible to find optimum distribution of the load using formulation (3.1.1)-(3.1.2) if we know the sequence of the processor activation (i.e. values d_i for $i = 1, \ldots, z$).

The subset of processors $P_1, \ldots, P_m$ exploited in the computations and the targets

of the communications are unknown, and must be determined. This task has combinatorial nature. In Sections 3.1.2 and 3.1.3 we propose algorithms that determine destinations for the load chunks. If one ignores proper selection of the chunk destinations, the problem becomes computationally easier because only linear program (3.1.1)-(3.1.2) has to be solved for some assumed chunk destinations $d_1, d_2, \ldots, d_z$. Then, the resulting schedules can be arbitrarily bad in the worst case, as demonstrated in the Section 2.3. How bad the solutions can be on average, if we skip the combinatorial part of the problem, is unknown. We attempt answering this question in Section 3.1.4.

3.1.2 Branch&Bound Algorithm

Branch&bound ($B\&B$) algorithm is an enumerative method that generates and searches the space of possible solutions, while eliminating these subsets of solutions which are infeasible or worse than some already known solution.

Two elements constitute a branch&bound algorithm. The first is *a branching* procedure which divides the solution space into disjoint subsets. Partition of the solution space can be represented as a tree. Each node is a representative of a set of solutions. Dividing such a set is equivalent to generating successors of a node. In our problem we have to select the sequence of the targets for z load chunks. In the root of the tree the sequence is empty. The first chunk may be sent to one of processors P_i,

for $i = 1, \ldots, m$. Therefore, the root has m successors each representing sequences starting with a message sent to processor P_i. The second level of the tree includes two-processor sequences (T_i, T_j). Branching a partial sequence $\overline{d(i)} = (d_1, \ldots, d_i)$, for $i < z$, is done by adding all possible destinations $d_{i+1} \in \{1, \ldots, m\}$. The branching procedure is continued until constructing a sequence of the assumed length z.

The maximum number of the search tree leaves is m^z. As this number grows exponentially with z, it is necessary to prune the search tree by eliminating nodes representing solutions certainly not better than some solution already known. This procedure is the *bounding* element of the algorithm. To determine if a node should be eliminated its *lower bound* of the schedule length is calculated. Suppose the node represents a sequence of l chunks. Thus, values $\overline{d(l)}$ are already determined. The remaining $z - l$ chunks still need to be selected. We assume that these $z - l$ chunks are sent to $z - l$ ideal target processors. The ideal target processor has parameters $A^{id} = \min_{i=1}^{m}\{A_i\}, C^{id} = \min_{i=1}^{m}\{C_i\}, S^{id} = \min_{i=1}^{m}\{S_i\}$, and processes only one load chunk. For such a sequence of l real processors, and $z-l$ ideal ones, a linear program (3.1.1)-(3.1.2) is solved for C_{max} which is the lower bound. The best known solution used in comparisons with the lower bound is found by the algorithm itself. It is the best solution found in any leaf of the search tree. The tree is searched in the depth-first least lower bound order.

3.1.3 Genetic Algorithm

Genetic algorithm (GA) is a randomized search method which implicitly discovers the near-optimum solution by randomly combining pieces of good solutions. The discussion and presentation of genetic algorithm can be found, e.g., in [27, 34]. Here we present basics of our implementation of GA only.

Genetic algorithms imitate evolution of genome. Solutions are encoded as strings of symbols analogously to the encoding of genes in the chromosomes. Some initial population of solutions is generated randomly. Genetic operators transform populations in a direction improving quality of the solutions. Selection, crossover, and mutation are typical genetic operators. *Selection* elects better solutions for the next population. *Crossover* operation generates offspring solutions by randomly combining pieces of the parent strings. Though the offspring is constructed in a random manner, the fragments of a string encoding an optimum solution are indirectly discovered and combined due to the selection and crossover. This happens because better strings usually have higher probability of being elected in the selection, and therefore, have bigger chances of being passed to the offspring in the crossover process. *Mutation* changes randomly some solutions to diversify the search, and to escape local optima. We direct interested readers to monographs [27, 34] for detailed presentation of the genetic search method.

A set of D solutions is a population. Solutions are encoded as strings $\overline{d} = (d_1, \ldots, d_z)$ of chunk destinations. The measure of a chromosome fitness is the value of schedule length C_{max} obtained from the linear program (3.1.1)-(3.1.2) formulated for the sequence of chunk targets given in the chromosome.

Solutions of the population are subject of genetic operators. We used three operators: crossover, mutation, and selection. In the crossover operation two chromosomes are randomly selected, and combined using one point crossover. For example, let $(a_1, a_2, \ldots, a_z)$, $(b_1, b_2, \ldots, b_z)$ be two parent solutions, and let k denote a randomly selected crossover point. The two offspring solutions are $(b_1, \ldots, b_{k-1}, a_k, \ldots, a_z)$ and $(a_1, \ldots, a_{k-1}, b_k, \ldots, b_z)$. The total number of new chromosomes constructed in crossover is Dp_C, where D is the size of the population, and p_C is a tunable algorithm parameter which will be called crossover probability.

Mutation changes Dzp_M random genes (i.e. d_is) to different values. Dz is the total number of genes, p_M is a tunable algorithm parameter which we will call mutation probability.

The selection of the chromosomes for the new population is done by a combination of an elitist and a roulette wheel methods. The best half of the old population is always preserved. A string is passed to the second half of the new population with probability: $\frac{1}{C_{max}(\overline{d_j})} / \sum_{j=1}^{D} \frac{1}{C_{max}(\overline{d_j})}$, where $C_{max}(\overline{d_j})$ is the schedule length for chromosome j.

The populations are modified iteratively. The number of iterations is limited in two ways: there are an upper limit on the total number of iterations, and an upper limit on the number of iterations without quality improvement.

3.1.4 Computational Experiments

Experiment Setting

All the experiments were performed on a PC computer with Pentium IV 1.8GHz, 512MB RAM memory, and Microsoft Windows XP. The executable code was generated by Borland C++ Builder 6.0. All LP formulations were solved by a code derived from `lp_solve` [3]. Unless stated otherwise, the test instances of the scheduling problem were generated in the following way: Processor parameters A_i, C_i, S_i, were generated with uniform distribution from the range [0,1]. Problem size was $V = 1E6$. The processor number was $m = 4$, and the number of chunks was $z = 8$. Each point on the following charts is an average of at least 10 instances.

Tuning the genetic algorithm

In the genetic algorithm, genes of the initial population were generated with uniform distribution from set $\{1, \ldots, m\}$. To tune the genetic algorithm, i.e. to select parameters G, p_C, p_M, and the stopping criterion, we applied the following procedure. A set of 100 random instances were generated as a reference benchmark. An indicator of algorithm performance was the average quality of the best solutions obtained for

these benchmark instances. Intuitively, a big population size D should allow for finding good solutions in a small number of iterations. On the other hand, maintaining big populations is computationally expensive. Population size $D = 50$ was selected as further increases of the population size gave only marginal improvements in the convergence of the solutions quality (cf. Fig. 3.2). For the fixed D crossover probability $p_C = 80\%$ was selected (see Fig.3.3). p_C determines the number of crossover operations. It turned out that majority of the population (80%) are offspring. Hence, it can be concluded that crossover is an effective optimization operator. After fixing G, and p_C, the mutation probability $p_M = 3\%$ was selected. It can be seen in Fig. 3.4 that p_M should be neither too big, nor to small. Too little value of p_M does not provide sufficient search diversity. Too much mutation deters convergence of the solutions. A limit of 10 iterations without solution improvement, and an upper limit of 100 iterations in total were used as stopping criteria. Better combinations of these parameters were also observed, as shown in Fig. 3.5. However, considering acceptable solution quality for "10/100" stopping criteria, and definite computational cost for other combinations we decided to use the least expensive option.

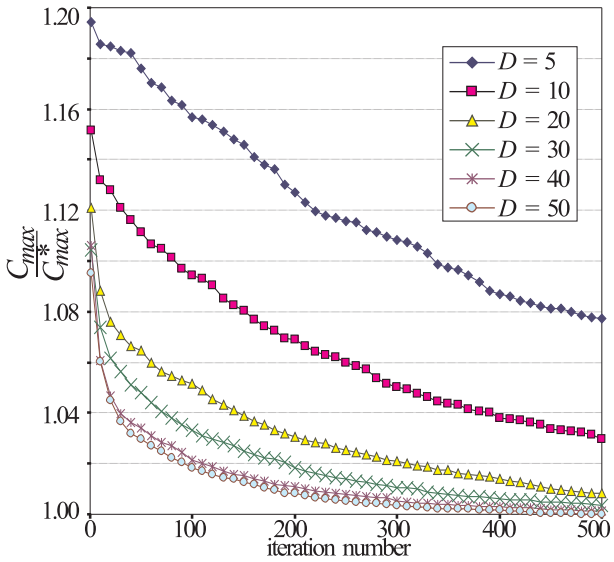

Figure 3.2: Average distance from optimum vs. iteration number and D.

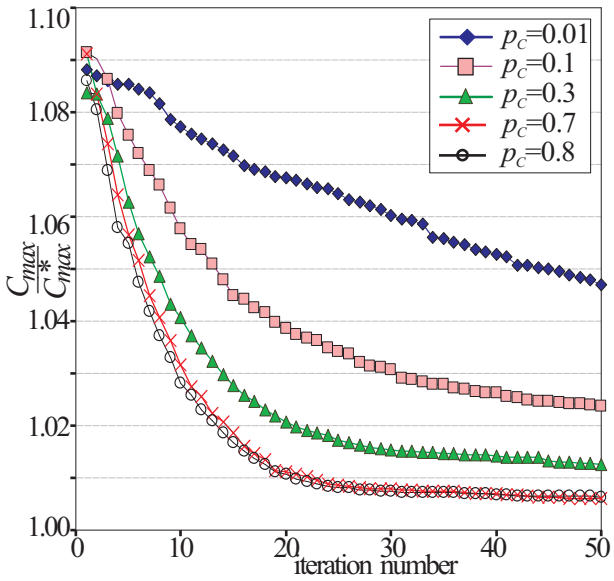

Figure 3.3: Average distance from optimum vs. crossover probability p_C.

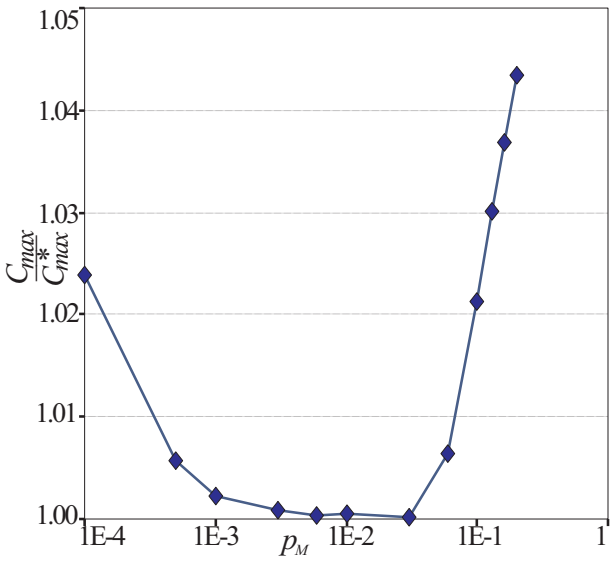

Figure 3.4: Average distance from optimum vs. mutation probability p_M.

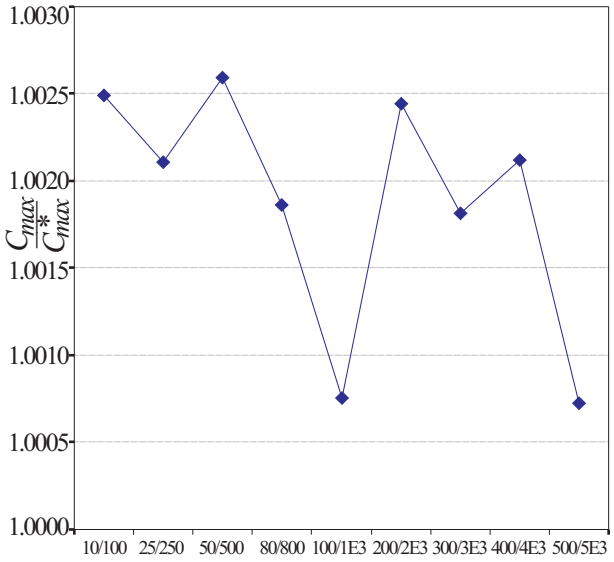

Figure 3.5: Average distance from optimum vs. iteration limits.

Performance of the Algorithms

Running Times

The execution times of the algorithms are collected in Fig.3.6, and 3.7. The running time of the branch and bound is denoted by $B\&B$, and of the genetic algorithm by GA. It can be seen that the branch and bound algorithm has exponential running time in z for fixed m (cf.Fig.3.6). The execution time grows slower as a function of m for fixed z (cf.Fig.3.7) because the maximum number of the search tree leaves is m^z. Nevertheless, execution time of the branch and bound algorithm allows only for solving instances with small m, and z. Execution time of the genetic algorithm grows with z (Fig.3.6) because the length of the string encoding solution is z. For $m = 3, \ldots, 20$ execution time grows less than twice (Fig.3.7). We also tested dependence of the execution times on size V of the problem. For small V execution time of the branch and bound was shorter than for big sizes because less processors had to be activated, and therefore the search trees were smaller. The execution time of the genetic algorithm was independent of V.

Quality of the Solutions

The results of our study on the quality of solutions are collected in Fig.3.8-3.9. The instances in Fig.3.8 had A_i parameters equal to a given value on all processors. The remaining C_i, S_i parameters were generated as described previously. Analogously,

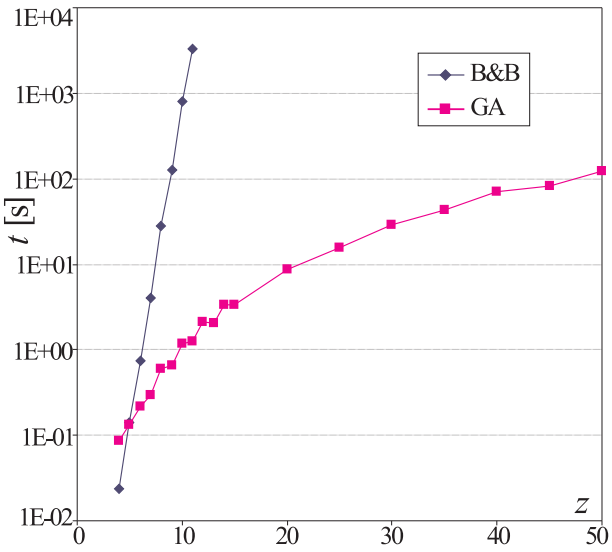

Figure 3.6: Running time vs. z.

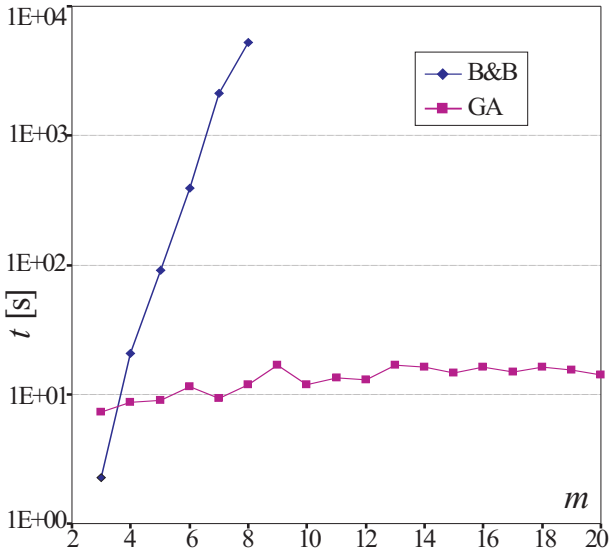

Figure 3.7: Running time vs. m.

for Fig.3.9 parameters C_i were fixed on all processors, and A_i, S_i were randomly generated. Each figure represents quality of the solutions, i.e. the relative distance from the optimum, in three cases: the average solution of a genetic algorithm (denoted GA), the average random solution (denoted RND), and the worst selection of the chunk targets ever observed (denoted Worst). Note that the worst case has its own ordinate ('y') axis different than RND, and GA cases. The random solutions (RND) have random chunk destinations. In all cases load chunk sizes were calculated by linear program (3.1.1)-(3.1.2).

These three cases demonstrate weaknesses and strengths of the two parts in the solution of our problem: the combinatorial part which finds targets for the chunks (d_is), and the linear programming part which calculates optimum chunk sizes (α_is) for the given destinations. It can be seen that genetic algorithm constructs solutions that are very close to the optimum. On average its solutions were not further 0.2% distant form the optimum. The worst solution obtained by the genetic algorithm was 1.1% away from the optimum. Thus, the genetic algorithm is a practical replacement for the optimization branch and bound algorithm which has exponential running time. The random solutions (RND) are also good on average because their distance from the optimum is not greater than approximately 30%. This is good news because solving a complex combinatorial problem of determining chunk targets (be it by a branch and bound or by a genetic algorithm) may be too time consuming and

unprofitable on average. A random, or reasonable selection of processors and their activation sequence, supplemented by a linear program (3.1.1)-(3.1.2) gives solutions of acceptable quality on average. This tells us also about the nature of the problem we are solving. Since relatively good results can be obtained only by adjusting chunk sizes (even for random chunk destinations), the chunk size selection is an important element in the solution of our problem. In other words, linear programming can compensate for some bad decisions in combinatorial part of the algorithms. It can be said that on average the combinatorial part of our problem (i.e. target selection) improves a random solution by approximately 30%. Finally, the worst case really exists. In the worst observed case of the chunk target selection a schedule 35 times worse than optimum was constructed (cf. Fig.3.9).

It is possible to infer from Fig.3.8-3.9 on the features of the solutions and performance of the algorithms. With growing A, C the quality of the random and the worst case is improving. When A is very big schedule length becomes dominated by the computation time. The selection of the chunk destinations is nearly meaningless because schedule length is determined by computation time which is approximately $\frac{AV}{m}$. Similar conclusions can be drawn for parameter C. When C is very big, chunk target selection tends to be immaterial because schedule length is determined by the communication time which is approximately VC. Thus, we can concluded that good solutions are easy to be obtained in the limit situations, when schedule length is

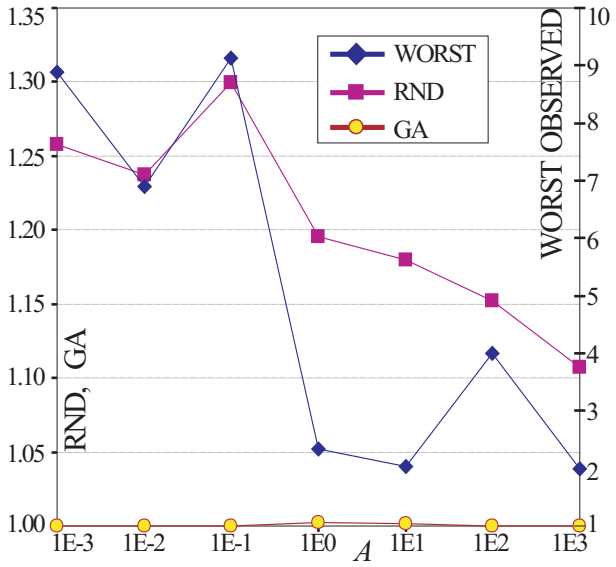

Figure 3.8: Relative distance from the optimum vs. A.

severely dominated either by communication or computation. We also tested dependence on startup time S in range [1E-3,1E3]. It turned out that S constitutes at most $\approx 2\%$ of the communication time, and hence this dependence was not strong.

3.2 With memory limits

In this section we assume that the amount of processor memory dedicated for the computations is limited. The load is not processed instantly. Therefore, in no moment of time can the load accumulated by a processor exceed the amount of available memory. The processors are heterogeneous. The problem is to find the sequence of processor communications, and the sizes of the load chunks such that the length of the schedule is minimum. Similarly as before we propose two algorithms for this problem. Exact branch&bound algorithm, and a heuristic based on the genetic search.

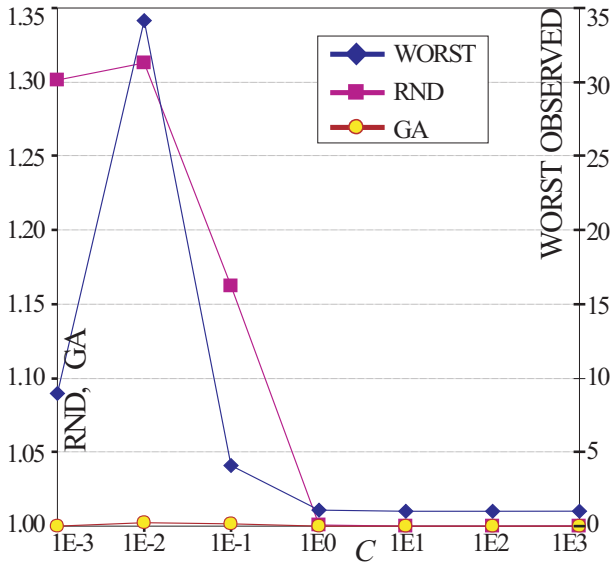

Figure 3.9: Relative distance from the optimum vs. C.

3.2.1 Problem Formulation

We assume that each processor P_i is defined not only by communication link startup time S_i, communication transfer rate C_i, processing rate A_i, but also by memory limit B_i. The load is distributed in multiple small chunks, rather than in one long message. Therefore, the load may accumulate in the processor memory if new chunks arrive faster than the received load is processed. The method of memory management has influence on the conditions that must be met to satisfy memory limitations. Below we discuss some options. In all cases we assume that memory is allocated at the beginning of communication with the arriving load chunk.

1) When load chunk j starts arriving, a memory block of size $\alpha_j \leq B_{d_j}$ is allocated from the memory pool of the computer system. After processing chunk j memory block is released to the operating system. Only after processing the previous chunk,

a new may be received. This approach was assumed in the earlier papers [23, 24, 25, 31, 44] and in Chapter 2. Unfortunately, in the case of multiple load chunks this method is not very effective because the load from the successive chunks may accumulate. Still, memory usage may be limited by gearing the chunk size to the speed of computation and communication such that the load does not accumulate [24].

2) When load chunk j starts arriving, many small blocks of memory are allocated from the memory pool. The size of each small block is equal to the size of the grain of parallelism. The total allocated memory size is α_j. As processing of the load progresses the memory blocks are gradually released to the operating system. This method of memory management is illustrated in Fig.3.10a. Thus, the peak of memory requirement takes place when a new chunk arrives.

3) As in the first case, memory block of size α_j is allocated when chunk j starts arriving. It is released after processing chunk j. However, it is required that the total memory allocated to the chunks present in the processor never exceeds limit B_{d_j}. This method of memory management is illustrated in Fig.3.10b.

Due to the simplicity of mathematical representation, in this work we adopt the second method of memory management. For the same reason we assume that the

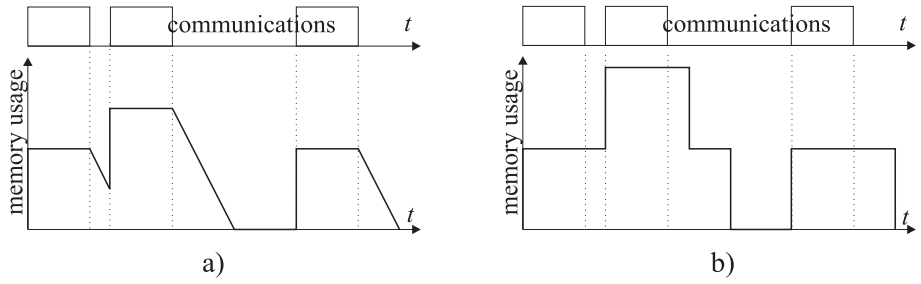

Figure 3.10: Memory usage for a) management method 2, b) management method 3.

time of returning the results of computations is negligible, and that processors cannot compute and communicate simultaneously. Consequently, computations are suspended by communications. We assume that the sequence of processor destinations $\overline{d}$ is given. Hence, we know the number z_i of load chunks sent to processor P_i, and function $g(i,k) \in \{1,\ldots,z\}$ which is the global number of a chunk (i.e. the number of originator message) sent to processor P_i as k-th for $k = 1,\ldots,z_i$. Let t_j denote the time moment when sending load chunk j starts. We will denote by x_{ik} the amount of load that accumulated on processor P_i at the moment when communication k to P_i starts. The problem of optimum chunk size selection can be formulated as the following linear program:

$$\min \quad C_{max}$$

subject to:

$$t_j + S_{d_j} + \alpha_j C_{d_j} \leq t_{j+1} \qquad j = 1,\ldots,z-1 \qquad (3.2.1)$$

$$x_{i,k-1} + \alpha_{g(i,k-1)} - \frac{t_{g(i,k)} - (t_{g(i,k-1)} + S_i + C_i\alpha_{g(i,k-1)})}{A_i} = x_{ik}$$

$$i = 1, \ldots, m, k = 2, \ldots, z_i \tag{3.2.2}$$

$$x_{ik} + \alpha_{g(i,k)} \leq B_i \qquad i = 1, \ldots, m, k = 1, \ldots, z_i \tag{3.2.3}$$

$$t_{g(i,z_i)} + S_i + C_i\alpha_{g(i,z_i)} + A_i(\alpha_{g(i,z_i)} + x_{iz_i}) \leq C_{max} \quad i = 1, \ldots, m \tag{3.2.4}$$

$$\sum_{j=1}^{n} \alpha_j = V \tag{3.2.5}$$

$$x_{i1} = 0, x_{ik} \geq 0 \qquad i = 1, \ldots, m, k = 1, \ldots, z_i \tag{3.2.6}$$

$$t_j, \alpha_j \geq 0 \qquad j = 1, \ldots, z \tag{3.2.7}$$

In the above formulation inequality (3.2.1) guarantees that communications do not overlap. The amount of load accumulated on P_i at the moment when chunk k starts arriving is calculated in equation (3.2.2). By inequality (3.2.3) memory limit is not exceeded. Computations finish before the end of the schedule by constraint (3.2.4), and the whole load is processed by equation (3.2.5). The above formulation can be adjusted to the case of processors equipped with communication front-ends when simultaneous receiving the load and computation is possible. In such a situation constraint (3.2.2) should be split into two constraints: $x_{ik} \geq x_{i,k-1} + \alpha_{g(i,k-1)} - \frac{t_{g(i,k)} - t_{g(i,k-1)}}{A_i}$, and $x_{ik} \geq \alpha_{g(i,k-1)} - \frac{1}{A_i}(t_{g(i,k)} - (t_{g(i,k-1)} + S_i + C_i\alpha_{g(i,k-1)}))$.

We conclude this section with an observation that the optimum multi-installment divisible load distribution in a heterogeneous system with memory limits can be found in polynomial time provided that the sequence of processor communications $\overline{d}$ is given.

In the next section we propose two methods of constructing $\overline{d}$.

3.2.2 Branch&Bound Algorithm

The $B\&B$ algorithm is constructed as described in Section 3.1.2. The lower bound is C_{max} obtained from linear program (3.2.1)-(3.2.5) by assuming that load chunks $i+1,\ldots,z$ are each sent to ideal processors. An ideal processor P_{id} has all the best parameters in the processor set, i.e. $A_{id} = \min_{i=1}^{m}\{A_i\}, C_{id} = \min_{i=1}^{m}\{C_i\}, S_{id} = \min_{i=1}^{m}\{S_i\}, B_{id} = \max_{i=1}^{m}\{B_i\}$. If the resulting linear program (3.2.1)-(3.2.5) is infeasible then it means that volume V is greater than the available processor memory. If the lower bound is greater than or equal to some already known solution then there is no hope that any of $\overline{d(i)}$ descendants will improve the schedule. In both cases $\overline{d(i)}$ is not expanded, and in this way the search tree is pruned.

3.2.3 Genetic Algorithm

A genetic algorithm for the current problem is constructed as described in Section 3.1.3. Quality of some solution $\overline{d}$ is schedule length $C_{max}(\overline{d})$ obtained as a solution of the linear program (3.2.1)-(3.2.5) formulated for sequence $\overline{d}$. Sequences infeasible because of insufficient memory, were eliminated from the population.

3.2.4 Computational Experiments

Experiment Setting

$B\&B$ and GA were implemented in Borland C++ 5.5 and tested in a set of computational experiments run on a PC computer with MS Windows 2000. Linear programs were solved using simplex code derived from `lp_solve` [3]. Unless stated otherwise the instance parameters were generated with uniform distribution from ranges $[0, 1]$ for parameters A_i, C_i, S_i, and $[0, \frac{2V}{z}]$ for parameter B_i. Infeasible instances with $z \max_{i=1}^{m}\{B_i\} < V$ were eliminated.

GA requires tuning, i.e., population size D, parameters p_C, p_M as well as the limits on the iteration number must be determined. We applied the following tuning procedure. A set of 10 random instances with $m = 4$, and $n = 8$ were generated and solved by $B\&B$ and GA. The average relative distance from the optimum was a measure of the quality of tuning. First, population size $D = 40$ was selected as increasing D beyond 40 did not improve significantly solution quality but increased complexity of the algorithm. Second, crossing over probability $p_C = 50\%$, and mutation probability $p_M = 5\%$ were selected for which best quality was obtained in minimum number of iterations. Finally, the limits of iteration numbers 100 (without quality improvement), and 1000 (in total) were selected for which solution distance from the optimum was better than 0.1% (in the tuning process). As in Section 3.1.4 we used random chunk destinations as algorithm RND. The worst activation sequences were also recorded.

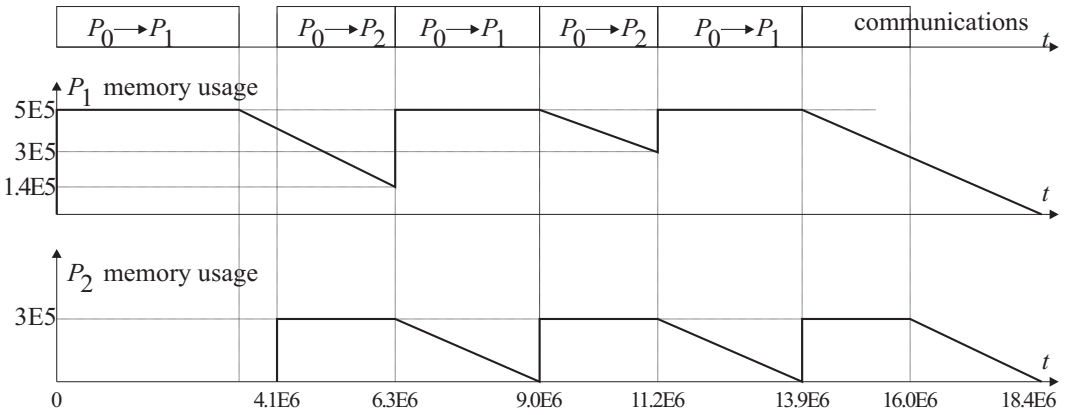

Figure 3.11: Solution for $m = 2, z = 6, V$ =2E6, $A_1 = A_2 = 8.98, C_1 = C_2 = 7.39, S_1 = 2.01, S_2 = 3.02, B_1$ =5E5, B_2 =3E5.

Before discussing the quality of the algorithms let us discuss features of the optimum solutions. In Fig.3.11 an optimum schedule constructed by $B\&B$ algorithm is presented. It is a typical situation that memory buffers are filled to the maximum capacity. We observed that if the number of messages is small then memory buffers were empty when a new message arrived (i.e. $x_{ik} = 0$). With the increasing number of messages z we observed an increase in the number of the optimum solutions in which the old load is not completely processed on the arrival of the new load ($x_{ik} > 0$). Intuitively, this seems reasonable because when z is small each message must carry nearly a maximum load. Hence, messages sent to processor P_i have maximum load B_i, which requires that the load is completely processed before receiving a new chunk.

Running Times

Dependence of the $B\&B$ and GA execution times on z, m, B are shown in Fig.3.12, Fig.3.13, Fig.3.14, respectively. Each point in these diagrams is an average of at least

ten instances. The worst case number of leaves visited in a $B\&B$ search tree is m^z. Thus, the execution time of $B\&B$ is exponential in z for fixed m (see Fig.3.12), and polynomial in m for fixed z (cf. Fig.3.13). The execution time of GA grows with z because the length of the solution encoding and sizes of linear programs increase with z (see Fig.3.12). GA running time dependence on m is weaker: 10-fold increase of m resulted in 60% increase of the execution time (Fig.3.13). In Fig.3.14 dependence of the running time on memory size is shown. For this diagram processor memory sizes B_i were generated with uniform distribution from range $[0, \kappa\frac{2V}{n}]$, where κ is shown along horizontal axis as a 'memory factor'. With growing κ the size of available memory is growing on average, and more solutions are feasible. Consequently, less branches can be cut in $B\&B$, and deeper search trees have to be examined. Finally, when κ is sufficiently big, infeasibility of a solution becomes rare on average, and it is not limiting search tree in $B\&B$. As a result, dependence of $B\&B$ execution time on κ levels-off.

Quality of the Solutions

In this section we examine quality of the solutions constructed by the algorithms, and the impact of heterogeneity of the distributed system on the quality of solutions. First, let us note that GA turned to be very useful in deriving optimum, and near-optimum solutions. Over 55% of the instances were solved to the optimality by GA

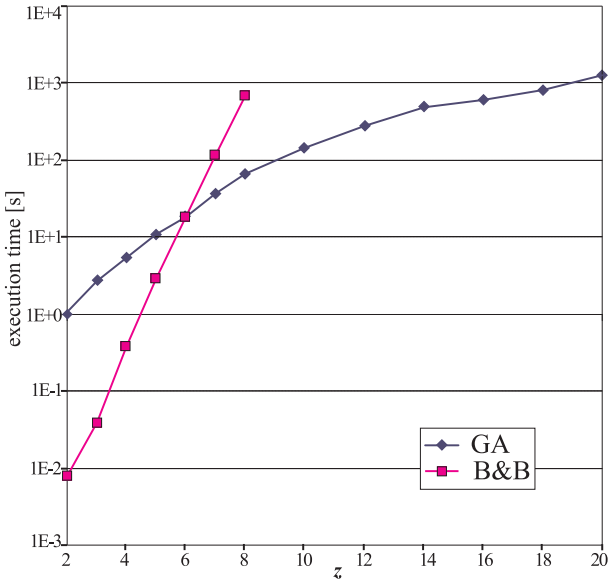

Figure 3.12: Execution time vs. z, for $m = 4$.

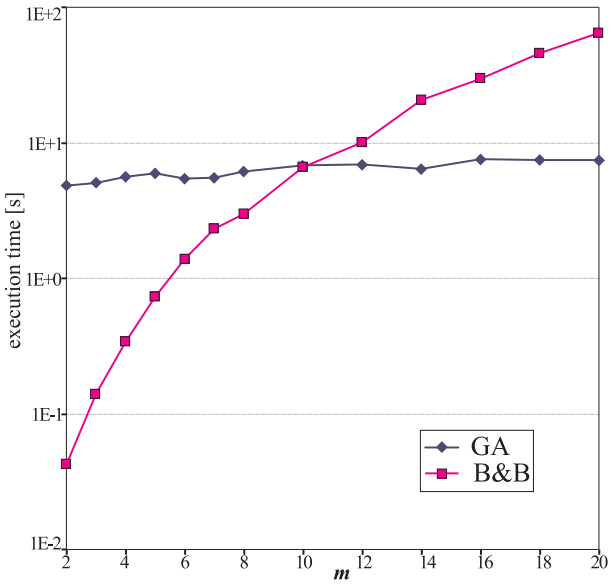

Figure 3.13: Execution time vs. m, for $z = 4$.

algorithm. The biggest observed relative distance from the optimum was 1.2%. In Fig.3.15, Fig.3.16, Fig.3.17 we show dependence of the solutions quality on the range of C, A, B, respectively. Along the vertical axis a relative distance from the optimum is shown for three kinds of solutions: an average for the genetic algorithm (denoted GA), an average for a randomly selected sequence od destinations (RND), and for the worst case ever observed (WORST). Note, that for the WORST dependence a dedicated axis is shown at the let side of the diagrams. For Fig.3.15, the communication rates C_i were generated from range $[1-\lambda_C, 1+\lambda_C]$. The remaining parameters were generated as described above. Thus, with growing λ_C heterogeneity of the communication system was growing. Values of λ_C are shown along horizontal axis denoted as C range. As it can be seen in Fig.3.15 with growing heterogeneity of C_i parameters the quality of both random and the worst case solutions is decreasing. Note, that this dependence is growing especially fast when C_i variation (λ_C) is big. In Fig.3.16 dependence of the solution quality on heterogeneity of A_i parameters is shown. For this diagram parameters A_i were generated from the range $[1-\lambda_A, 1+\lambda_A]$. The other parameters were generated as described above. Some weak dependence of the solution quality on the diversity of A parameter can be observed: with growing A diversity, quality is getting worse. For Fig.3.17 the memory sizes were generated from range $[\frac{2V}{n}-\lambda_B, \frac{2V}{n}+\lambda_B]$, for fixed value of V. The value of λ_B is shown along the horizontal axis. Again, a weak trend of decreasing quality of the solutions can be observed with

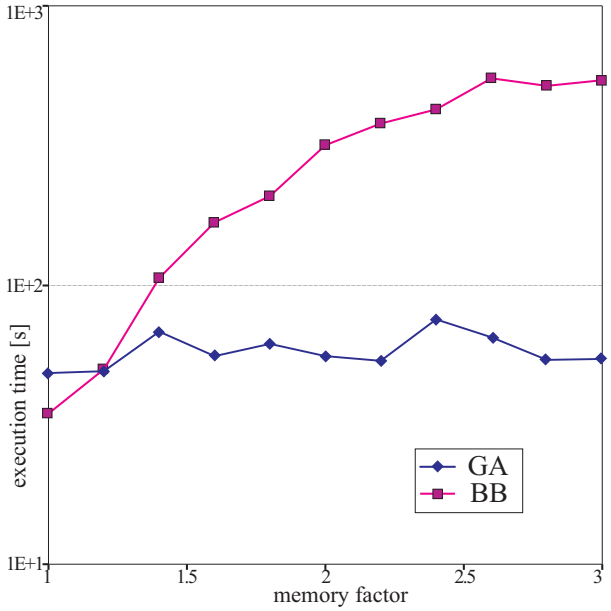

Figure 3.14: Execution time vs. memory size, for $m = 4, z = 4$.

growing λ_B. Note that in Fig.3.16 and Fig.3.17 the distance from the optimum of RND, and WORST solutions is bigger than in Fig.3.15. This demonstrates that narrowing the diversity of C simplifies obtaining a good solution, and communication rate is a key parameter in performance optimization of the heterogeneous systems. Furthermore, the distance between WORST, RND, and GA or $B\&B$ solutions can be used as an estimate of the gain from finding the optimum, or near-optimum, sequence of processor activations. As it can be inferred this kind of gain is $\approx$ 10-40% for the random (RND) solution on average, and $\approx$ 10-fold for in the worst case.

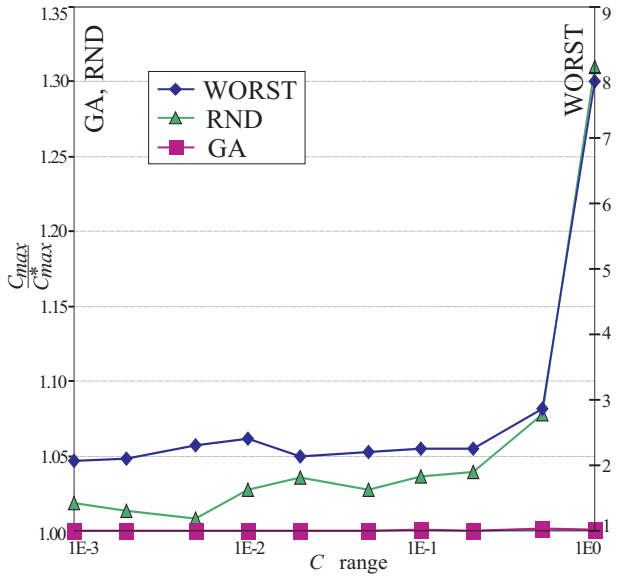

Figure 3.15: Quality of the solutions vs. range of C, $m = 4, z = 8$.

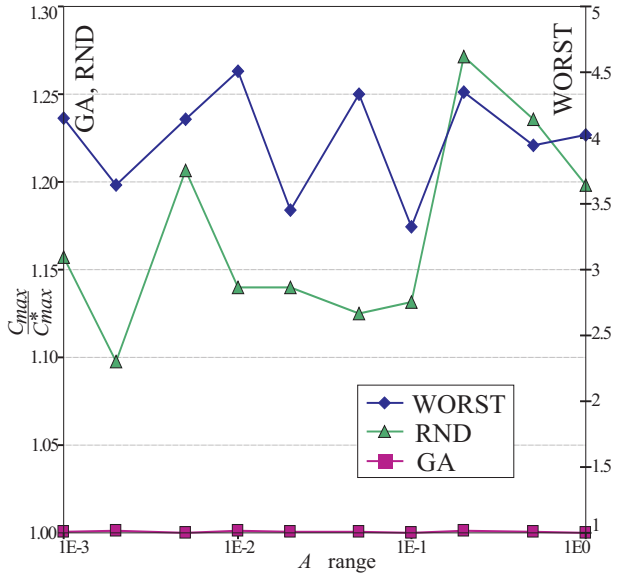

Figure 3.16: Quality of the solutions vs. range of A, $m = 4, z = 8$.

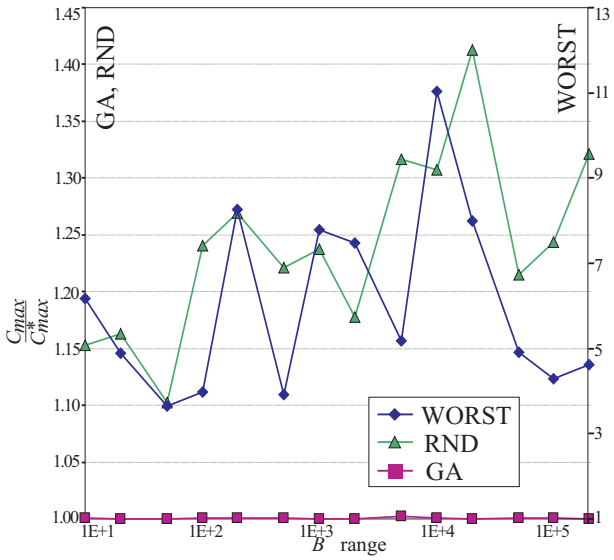

Figure 3.17: Quality of the solutions vs. range of memory sizes, $m = 4, z = 8, V = 1E6$.

3.3 Conclusions

The problems we analyzed in this chapter consist in determining optimum destinations for the load chunks and adjusting their sizes to the speeds of processors, communication links and possibly memory sizes. A linear programming formulation has been proposed for a fixed processor activation sequence. We divided solution methods into two parts: combinatorial one which finds destinations for the load chunks, and a linear programming part which finds optimum chunk sizes for the given targets. We have shown that in the worst case solutions can be arbitrarily bad if any of the two parts is ignored. Two algorithms were proposed to find an optimum, or near optimum processor activation sequence. The running times, and quality of the solutions were compared in a series of computational experiments. It turned out that the

proposed genetic algorithm is very effective in finding near-optimum solutions. In a set of computational experiments we demonstrated that on average the combinatorial part improves the solution quality by approximately 30 %. The algebraic part (LP) is a very important element in the construction of the schedule, and to some extent it is able to compensate bad decisions in the combinatorial part. The impact of heterogeneity on the solution quality has been also studied. It appears that with growing system heterogeneity good quality solutions are harder to be found. Especially narrowing communication speed diversity simplifies obtaining good solutions.

Chapter 4

Multiple Loads Single Distribution

In this chapter we analyze complexity of scheduling divisible loads $T_1, \ldots, T_n$ of sizes $V_1, \ldots, V_n$ on m parallel processors $P_1, \ldots, P_m$ interconnected in a star topology. We assume that processors have sufficient memory buffers to store the received loads, and computations do not have to start immediately after receiving the load. Note that even for uniform and identical processors n tasks are not equivalent to a single task with load $\sum_{j=1}^{n} V_j$ because each task is a separate scheduling entity, separate memory object, and requires a separate set of communications.

By constructing a schedule the originator decides on: the sequence of the tasks, the set of processors $\mathcal{P}_j$ assigned to each of tasks j, the sequence of processor activation, and the sizes of the load parts. Our objective is minimization of the schedule length, denoted by C_{max}. Let us now point out several possible assumptions on the structure of the schedule.

In some cases the time of returning the results may be so short in comparison with the load scattering and computing phases, that the result returning may be neglected

in the construction of the schedule. This assumption is commonly used in modeling divisible load computations [7, 17, 36]. It has been observed in the earlier DLT papers that if the result returning time may be neglected, then the schedule for a single task is the shortest when all the activated processors complete computations at the same moment. This requirement may be extended to the multiple loads case. We will say that a schedule has *simultaneous completion* property if the computations on all parts of each task finish simultaneously. Simultaneous completion of the computations may be also justified by technological reasons: When a parallel application finishes at the same time on all processors, then managing it in a parallel computer batch system is simpler than if it were finishing on different processors in widely scattered moments of time.

On the other hand, the process of result returning may be equally time consuming as load distribution and computations. In such cases we will assume that the amount of returned results is $\beta_j \alpha_{ij}$, which means that the volume of results is proportional to the amount of received load, and coefficient β_j is application specific. The result returning phase must be explicitly scheduled. We assume that transfer rates and startup times are the same for sending the load to the processors, and for returning the results.

It is assumed in this chapter that the originator constructs *permutation schedules* (see e.g. [13, 35] for the classic definition). By permutation schedule we mean that

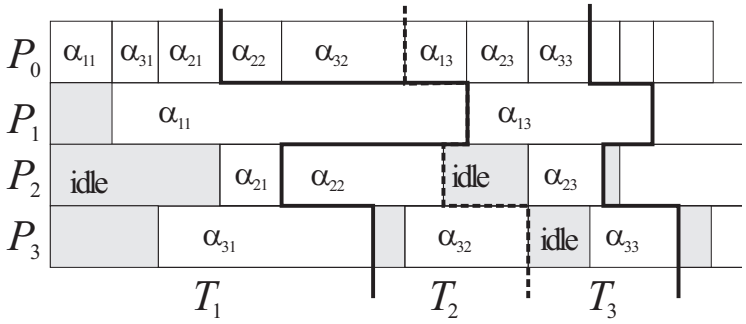

Figure 4.1: An example of a permutation schedule.

a task is sent to the processors only *once*, that a processor executes the task only *once*, and the sequence of the tasks is the same on all processors. Consequently, communications and computations are non-preemptive, i.e. cannot be suspended and restarted later. If $P_i \notin \mathcal{P}_j$, then a dummy computation interval of length 0 is inserted on P_i. An example of permutation schedule is shown in Fig.4.1. When returning of the results is considered we will also assume permutation schedules by which we mean that the order of the tasks in distribution, computation, and result collection phases is the same.

To simplify presentation some elements of the general formulation of divisible load scheduling problem as defined in Chapter 1 will not always be used here. Unless stated otherwise it is assumed that memory size is not bounding $\forall_{i,j} B_{ij} = \infty$, processor availability is not constrained ($\forall_i r_i = 0, d_i = \infty$), costs of computations are negligible, computation startup time is zero $\forall_{i,j} p_{ij} = 0$. Further organization of this chapter is the following. In Section 4.1 computationally hard cases are identified. In Section 4.2 some polynomially solvable cases of the problem are presented. Bounds on the

quality of approximation algorithms are given in Section 4.3. In Section 4.4we analyze the problem of scheduling identical tasks on identical processors This problem is particularly interesting due to compact encoding of the instances.

4.1 Complexity

In this section we identify **NP**-hard, or **NP**-hard in a strong sense [26] cases of multiple divisible load scheduling problem. In our proofs of the computational complexity we will be using the **NP**-complete problems: PARTITION, PARTITION WITH EQUAL CARDINALITY and the strongly **NP**-complete 3-PARTITION problem, defined in Section 1.3.

Theorem 10. *Multiple divisible load scheduling problem is **NP**-hard even for one* $(m = 1)$ *unrelated processor, when result returning is considered.*

Proof. Before presenting the proof let us comment on a single unrelated processor. In classic deterministic scheduling theory single processor cannot be unrelated. However here, we have in fact two processors: communication medium is one classic theory processor, and P_1 is another classic processor. For $m = 1$ this problem is obviously in **NP** because NDTM has to guess the sequence of tasks execution. We will show that our scheduling problem is **NP**-hard by the following polynomial time transformation from PARTITION, to a decision version of our problem:

$n = q + 1$,

$V_j = 1, \beta_j = 1$ for $j = 1, \ldots, n$,

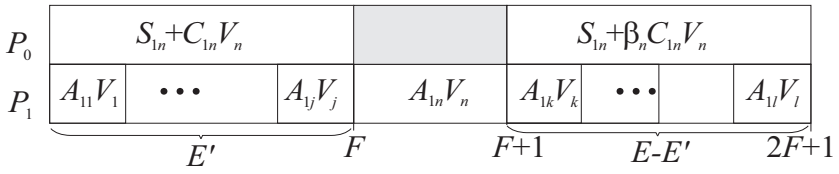

Figure 4.2: Illustration to the proof of Theorem 10.

$S_{1j} = 0$ for $j = 1, \ldots, n$,

$C_{1j} = 0$ for $j = 1, \ldots, q$, $C_{1n} = F$,

$A_{1j} = e_j$ for $j = 1, \ldots, q$, $A_{1n} = 1$.

We ask if a schedule with length at most $y = 2F + 1$ exists. Suppose, that the PARTITION instance has a positive answer. Then a feasible schedule of length $2F + 1$ can be constructed as shown in Fig.4.2.

Suppose the scheduling problem instance has a positive answer. Then task T_n is continuously performed because $S_{1n} + V_n C_{1n} + V_n A_{1n} + S_{1n} + \beta_n V_n C_{1n} = 2F + 1 = y$. As computations are non-preemptive, each of the tasks $T_1, \ldots, T_q$ must fit either into interval $[0, F]$, or into interval $[F + 1, 2F + 1]$. For the set of tasks $\mathcal{T}_{[0,F]}$ which computations are performed in $[0, F]$ we have $\sum_{T_j \in \mathcal{T}_{[0,F]}} A_{1j}V_j = \sum_{T_j \in \mathcal{T}_{[0,F]}} e_j \leq F$. Analogously, for the tasks in interval $[F + 1, 2F + 1]$: $\sum_{T_j \in \mathcal{T}_{[F+1,2F+1]}} A_{1j}V_j = \sum_{T_j \in \mathcal{T}_{[F+1,2F+1]}} e_j \leq F$. Thus, the PARTITION instance also has a positive answer. Consequently, the scheduling problem is **NP**-hard. □

Theorem 11. *If result returning time is negligible, then multiple divisible load scheduling problem for two* $(m = 2)$ *unrelated processors is* ***NP****-hard in the strong sense.*

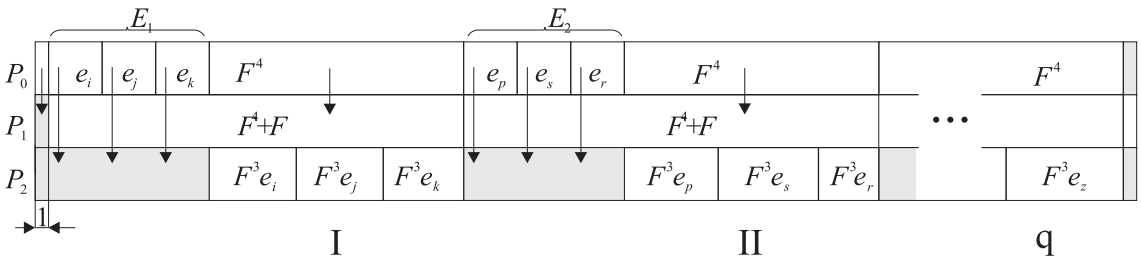

Figure 4.3: Illustration to the proof of Theorem 11.

Proof. We prove the theorem by reduction from 3-PARTITION. We assume (without loss of generality) that $F > q$. Were it otherwise, all e_j can be multiplied by $q > 1$ to fulfill this requirement. The instance of the scheduling problem can be constructed as follows:

$n = 4q + 1$, $V_j = 1$ for $j = 1, \ldots, n$,

$S_{1j} = 0, S_{2j} = 0, C_{1j} = 1, C_{2j} = e_j, A_{1j} = \infty, A_{2j} = F^3 e_j$ for $j = 1, \ldots, 3q$.

$S_{1,3q+1} = 0, S_{2,3q+1} = 0, C_{1,3q+1} = 1, C_{2,3q+1} = 1, A_{1,3q+1} = F^4 + F, A_{2,3q+1} = \infty$,

$S_{1j} = 0, S_{2j} = 0, C_{1j} = F^4, C_{2j} = 1, A_{1j} = F^4 + F, A_{2j} = \infty$ for $j = 3q + 2, \ldots, 4q$,

$S_{1,4q+1} = 0, S_{2,4q+1} = 0, C_{1,4q+1} = F^4, C_{2,4q+1} = 1, A_{1,4q+1} = 1, A_{2,4q+1} = \infty$,

$y = q(F^4 + F) + 2$.

We ask whether a schedule not longer than y exists. If 3-PARTITION instance has positive answer then a feasible schedule of length y may look like the one in Fig.4.3. Observe that processor P_2 can start executing tasks immediately after its first communication. Thus, there can be also other schedules not longer than y when a 3-PARTITION exists.

Suppose, a feasible schedule not longer than y exists. Due to the values of parameters A_{ij}, tasks $T_1, \ldots, T_{3q}$ can be executed on P_2 only, and tasks $T_{3q+1}, \ldots, T_{4q+1}$ on P_1 only. The total computing time on P_1 is $q(F^4 + F) + 1 = y - 1$, while the shortest load distribution operation lasts one unit of time. As a result, P_1 must compute all the time with the exception of the first time unit when the load of T_{3q+1} is sent. The sum of all communication times is equal to $y-1$. Thus, originator must communicate all the time with the exception of the last time unit when task T_{4q+1} must be executed on P_1.

Total computing requirement put on P_2 by tasks $T_1, \ldots, T_{3q}$ is qF^4. After excluding the first communication of T_{3q+1}, P_2 can be idle at most $qF + 1$ time units. To avoid idling on P_1, sending the load for the second task executed on P_1 must start at time $F + 1$ at the latest. Therefore, no more load can be sent to P_2 than for three tasks. Suppose that two tasks T_i, T_j are started on P_2 before sending the load for the second task on P_1, and T_i is started first. Then, there would be excessive idle time on P_2 since the end of T_j computations till the end of the communication operation of the second task executed by P_1. Let us calculate this idle time. $F^4 + e_j$ is the span of the interval since the end of T_i communication operation (moment when P_2 can start computing) till the end of the communication operation of the second task executed by P_1 (when P_2 can start receiving any new load). $F^3(e_i + e_j)$ is the time of computing operations which can be executed on P_2 in this interval. The idle time

on P_2 would be at least $F^4 + e_j - F^3(e_i + e_j)$. Since $F > q$ and $F > 1$ we have $F^4 + e_j - F^3(e_i + e_j) > F^4 - F^3(F-1) = F^3 \geq F^2 + F^2 > qF + 1$, while the idle time on P_2 cannot be greater than $qF + 1$. Hence, exactly three communications to P_2 must be done before sending the second task to P_1.

The sum of computation times of the first three tasks T_i, T_j, T_k allocated to P_2 must be equal to F^4. If it is less, then it is at most $F^3(e_i + e_j + e_k) \leq F^4 - F^3$ which results in $F^3 > qF + 1$ idle time on P_2 while communication of the second task allocated to P_1 with the originator. Suppose it is more, then sending their loads last longer than F and the sending operation of the second task allocated to P_1 cannot start at time $F + 1$, which results in additional idle time on P_1. Consequently, the three tasks must be processed in exactly F^4 time units. Otherwise schedule of length y cannot exist.

The same reasoning can be applied to the following tasks assigned to P_1. The load distribution operations of these tasks cannot be started later than by $1+iF^4+(i+1)F$ for $i = 1, \ldots, q-1$. This creates free time interval for at most three communications of the tasks assigned to P_2. Also no less than three tasks can be started by the originator, otherwise there will be excessive idle time on P_2. The processing times of the three tasks must be equal to F^4, otherwise either P_2 or the originator must be idle for a too long time. We conclude that for each triplet of tasks assigned to P_2 their computing time is F^4. Hence, 3-PARTITION instance has a positive answer. □

In the following theorem we consider a simpler case of uniform processors, but with simultaneous completion required, i.e., each task must be finished at the same time on all used processors.

Theorem 12. *If result returning time is negligible and simultaneous completion is required, then multiple divisible load scheduling on uniform processors is* ***NP****-hard already for two* $(n = 2)$ *tasks, even if the sequence of the tasks is known.*

Proof. First we will calculate the amount of a single application load that can be distributed, and processed on a star network with $C_i = 0$, until time τ. Without loss of generality, let us assume that the sequence of processor activation is $P_1, \ldots, P_m$. The amount of load V that can be distributed, and processed in time τ is (compare formula 2.2.1):

$$V = \sum_{i=1}^{m} \frac{\tau}{A_i} - \sum_{i=1}^{m} \sum_{j=i}^{m} \frac{S_i}{A_j} \tag{4.1.1}$$

Suppose that $\frac{1}{A_i} = S_i$ for all i. Formula (4.1.1) reduces to:

$$\begin{aligned} & V = \\ & \tau \sum_{i=1}^{m} S_i - \sum_{i=1}^{m} \sum_{j=i}^{m} S_i S_j = \\ & \tau \sum_{i=1}^{m} S_i - \frac{1}{2}(\sum_{i=1}^{m} S_i)^2 - \frac{1}{2} \sum_{i=1}^{m} S_i^2 \end{aligned} \tag{4.1.2}$$

Note that V in (4.1.2) does not depend on the sequence of processor activation.

We will show **NP**-hardness of the problem by a polynomial time transformation of PARTITION problem. Assume that $e_i > 2$ for $i = 1, \ldots, q$. Were it otherwise, all

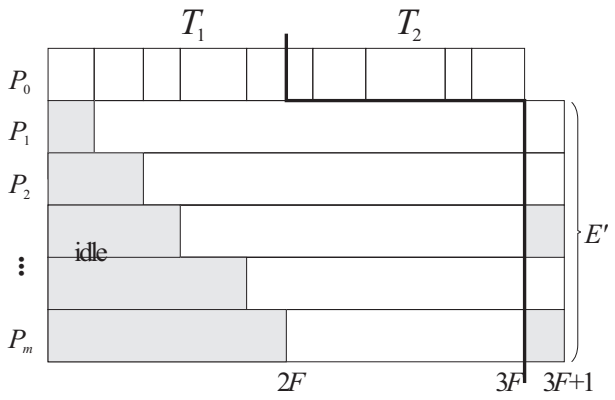

Figure 4.4: Illustration to the proof of Theorem 12.

e_i may be multiplied by 2 without changing the answer to the PARTITION instance. The transformation of a PARTITION instance to a scheduling problem instance is as follows:

$n = 2, m = q,$

$S_i = e_i, A_i = \frac{1}{S_i} = \frac{1}{e_i}, C_i = 0,$ for $i = 1, \ldots, q$

$V_1 = 4F^2 - \frac{1}{2}\sum_{i=1}^{m} e_i^2, V_2 = F$

$y = 3F + 1.$

As already mentioned the sequence of task execution is given: T_1 precedes T_2. We ask if a schedule of length at most y exists. Suppose the answer is positive for PARTITION problem. A feasible schedule for the instance of the scheduling problem is shown in Fig.4.4.

Processors corresponding to set E' in PARTITION are used by T_2. Let us check that the schedule is feasible. T_1 completes computations at time $\tau = 3F$. If we supply the values of startup times S_i, and processing rates A_i into equations (4.1.1),

(4.1.2), we get $3F\sum_{i=1}^{m} e_i - \frac{1}{2}(\sum_{i=1}^{m} e_i)^2 - \frac{1}{2}\sum_{i=1}^{m} e_i^2 = 6F^2 - \frac{1}{2}(2F)^2 - \frac{1}{2}\sum_{i=1}^{m} e_i^2 = V_1$. Thus, T_1 is executed feasibly. The communications of T_1 finish at time $2F$, therefore communications of T_1 which take $\sum_{i\in E'} S_i = F$ fit in F time units of available time. In the last time unit of interval $[y-1, y]$ the selected processors process $\sum_{i\in E'} \frac{1}{A_i} = \sum_{i\in E'} e_i = F$ units of load. Hence, also T_2 is executed feasibly.

Suppose that a schedule of length y exists. Task T_1 is executed first. All m processors must be used by T_1. Suppose it is otherwise, and some processor is not exploited. Without loss of generality we can renumber the processors such that P_m is the unused processor. By (4.1.2) the volume of the processed load for T_1 is at most $V_1' = y\sum_{i=1}^{m-1} S_i - \frac{1}{2}(\sum_{i=1}^{m-1} S_i)^2 - \frac{1}{2}\sum_{i=1}^{m-1} S_i^2 = (3F+1)\sum_{i=1}^{m-1} e_i - \frac{1}{2}(\sum_{i=1}^{m-1} e_i)^2 - \frac{1}{2}\sum_{i=1}^{m-1} e_i^2 = (3F+1)\sum_{i=1}^{m} e_i - (3F+1)e_m - \frac{1}{2}(\sum_{i=1}^{m} e_i)^2 - \frac{1}{2}\sum_{i=1}^{m} e_i^2 + \frac{1}{2}(e_m^2 + 2e_m\sum_{i=1}^{m-1} e_i) + \frac{1}{2}e_m^2 = V_1 + 2F - (3F+1)e_m + e_m^2 + e_m\sum_{i=1}^{m-1} e_i = V_1 + 2F - e_m(3F + 1 - \sum_{i=1}^{m} e_i) = V_1 + 2F - e_m(F+1) < V_1$ because $e_m > 2$. Hence, all m processors must be used by task T_1. If all processors are used then T_1 communications complete by $2F$, and due to simultaneous completion requirement, its computations finish at time $3F$. This leaves interval $[2F, 3F+1]$ free for communications, and interval $[3F, 3F+1]$ for the computations of T_2. Note that $\forall_i S_i \geq 2$, and any communication in interval $[3F, 3F+1]$ gives no contribution to the processed load of task T_2. Consequently communications of set $\mathcal{P}'$ of the processors selected for executing T_2 must satisfy $\sum_{i\in\mathcal{P}'} S_i = \sum_{i\in\mathcal{P}'} e_i \leq F$. The load of T_2 processed in interval $[3F, 3F+1]$ must

satisfy $\sum_{i\in\mathcal{P}'} \frac{1}{A_i} = \sum_{i\in\mathcal{P}'} e_i \geq F$. Thus, the answer is positive for PARTITION instance if the elements corresponding to the processors in set $\mathcal{P}'$ are selected to set E'. □

The case of arbitrary processor sequence is not simpler. We explain it in the following observation.

Observation 13. *If result returning time is negligible and simultaneous completion is required, then multiple divisible load scheduling on uniform processors is **NP**-hard even for two ($n = 2$) tasks, and arbitrary sequence of the tasks.*

Proof. The proof for the previous problem can be adjusted to the current situation. If the sequence of tasks is (T_2, T_1), then the length of the schedule is at least the length of the communications of T_2 plus the length of the schedule for T_1. Communications of T_2 last at least $\min_{P_i\in\mathcal{P}}\{S_i\} = \min_{i\in E}\{e_i\} > 1$. The duration of T_1 processing is at least $3F$ (see the proof of Theorem 12). The schedule length is at least $3F + 2$. Thus, only sequence (T_1, T_2) allows for a schedule of length at most $3F + 1$, which by the proof of Theorem 12 exists if and only if PARTITION exists. □

Note that Theorems 10 and 11 can be proved also for non-permutation schedules. Preemption in computation and communication can be eliminated by sufficiently long communication or computation startup times. Theorem 12 relies on the simultaneous completion of T_1, and hence it holds also for non-permutation schedules.

Now we will prove that scheduling multiple divisible loads on parallel identical processors is also **NP**-hard. In the proof of **NP**-hardness we will use the **NP**-complete

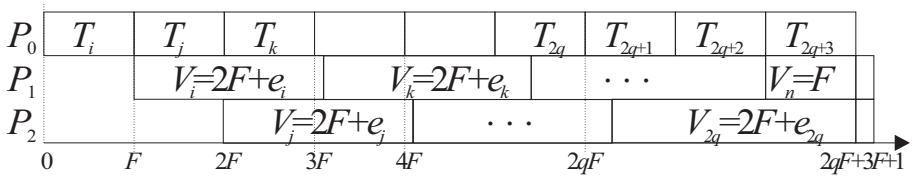

Figure 4.5: Illustration to the proof of Theorem theo-id-par.

problem PARTITION WITH EQUAL CARDINALITY defined in Section 1.3.

Theorem 14. *Scheduling of multiple divisible loads is **NP**-hard even for two identical processors.*

Proof. We will show that our scheduling problem is **NP**-hard by presenting a polynomial time transformation from PARTITION WITH EQUAL CARDINALITY to a decision version of our problem. Without loss of generality we assume that e_i are multiples of 3, for $i = 1, \ldots, 2q$. Were it otherwise all e_i can be multiplied by 3 to satisfy this requirement. The transformation is as follows:

$m = 2, S = F, C = 0, A = 1, n = 2q + 3,$

$V_j = 2F + e_j$ for $j = 1, \ldots, 2q$, $V_{2q+1} = F, V_{2q+2} = 1, V_{2q+3} = 1,$

We ask if a schedule with length at most $y = (2q + 3)F + 1$ exists. Suppose, that the PARTITION WITH EQUAL CARDINALITY instance has a positive answer. Then a feasible schedule of length $(2q + 3)F + 1$ can be constructed as shown in Fig.4.5.

Suppose there is a schedule of length at most y. Without loss of generality, let P_1 be the first processor which started computations. Note that only $2q + 3$ communications of length $S = F$ can be initiated and completed. Also $n = 2q + 3$. Thus, the load for each task is sent in only one message, and each task can be executed

on one processor only. The time required for processing the tasks $\sum_{j=1}^{n} V_j A = 4qF + 3F + 2$ is equal to the length of available processing intervals on processors P_1, and P_2 with the exception of the $3F$ time units of initial waiting for the load. Hence, there is no idle time in the computations on the processors. Tasks $T_{2q+1}, T_{2q+2}, T_{2q+3}$ have to be sent from the originator at the end of the schedule, because no other tasks with communications initiated at $2qF, (2q+1)F, (2q+2)F$ can be finished by $y = (2q+3)F + 1$. T_{2q+2}, T_{2q+3} must be executed on different processors because processing times of other tasks are multiples of 3 and otherwise there would be an idle time on some processor. This leaves $(2q+2)F$ free time on P_1, and $(2q+1)F$ time on P_2. Let $\mathcal{T}_i$ denote the set of tasks executed by processor P_i, for $i = 1, 2$. If T_{2q+1} were processed by P_2, then at most $q-1$ tasks from $\{T_1, \ldots, T_{2q}\}$ would be processed on P_2. Processing the tasks on P_2 would take $2F(q-1) + \sum_{T_j \in \mathcal{T}_2} e_j + F < (2q+1)F$, and an idle time would arise on P_2. Thus, T_{2q} must be executed on P_1, exactly q tasks from $\{T_1, \ldots, T_{2q}\}$ have to be on P_2, and the same number on P_1. Since T_{2q+1} is executed on P_1, the processing requirements of the tasks on P_1, and on P_2 must be equal to the remaining available time, i.e., $\sum_{T_j \in \mathcal{T}_1 - \{T_{2q+1}\}} AV_j = 2qF + \sum_{T_j \in \mathcal{T}_1 - \{T_{2q+1}\}} e_j = \sum_{T_j \in \mathcal{T}_2} AV_j = 2qF + \sum_{T_j \in \mathcal{T}_2} e_j = 2qF + F$. Hence, $\sum_{T_j \in \mathcal{T}_1 - \{T_{2q+1}\}} e_j = \sum_{T_j \in \mathcal{T}_2} e_j = F$, and the answer to PARTITION WITH EQUAL CARDINALITY is affirmative. □

In the proof of Theorem 14 tasks use one communication. Yet, the restriction to a single communication is not presumed, but it follows from the features of the instance.

Therefore, Theorem 14 applies both to single-distribution and to multi-installment processing. For similar reasons it applies to permutation, and not-permutation schedules, in the case with or without the simultaneous completion of the computations.

We conclude from the above results that the problem of scheduling multiple divisible loads is computationally hard. This means in practice that this problem has a hard combinatorial core which could not be expected by the earlier DLT literature. The main sources of the computational complexity are sequencing the tasks, selecting the processors to use, sequencing processor activation.

4.2 Polynomial cases

4.2.1 Fixed activation order, no result returning

When the task execution sequence, the set of used processors, and the processor activation orders are known, then the optimum distribution of the load can be found by using linear programming. Let us first study the case when simultaneous completion of the computations is not required, and results returning time can be ignored. For the sake of notation simplicity, and without loss of generality, let us assume that the order of task execution coincides with task numbers. The set of processors exploited by T_j is $\mathcal{P}_j$. The order of processor activation can be different for each task. Let the number of the i-th processor activated for task T_j be given by function $h(j,i)$. The amount of load from task $j = 1,\ldots,n$ sent to processor $i = 1,\ldots,m$ is denoted by $\alpha_{ij} \geq 0$.

The optimum distribution of the load can be found by the linear program:

minimize C_{max}

subject to:

$$\sum_{j=1}^{l-1}\sum_{i=1}^{|\mathcal{P}_j|}(S_{h(j,i)j}+\alpha_{h(j,i)j}C_{h(j,i)j})+$$
$$\sum_{i=1}^{k}(S_{h(l,i)l}+\alpha_{h(l,i)l}C_{h(l,i)l})+$$
$$\sum_{j=l}^{n}\alpha_{h(l,k)j}A_{h(l,k)j}\leq C_{max}$$
$$l=1,\ldots,n,\ k=1,\ldots,|\mathcal{P}_j| \tag{4.2.1}$$

$$\sum_{i\in\mathcal{P}_j}\alpha_{ij}=V_j \quad j=1,\ldots,n \tag{4.2.2}$$

Term $\sum_{j=1}^{l-1}\sum_{i=1}^{|\mathcal{P}_j|}(S_{h(j,i)j}+\alpha_{h(j,i)j}C_{h(j,i)j})$ in inequalities (4.2.1) is the time of sending the load for tasks $T_1,\ldots,T_{l-1}$. Sending the load to processors $h(l,i)$ activated as $i=1,\ldots,k$ in the sequence of processors executing task T_l lasts $\sum_{i=1}^{k}(S_{h(l,i)l}+\alpha_{h(l,i)l}C_{h(l,i)l})$. $\sum_{j=l}^{n}\alpha_{f(l,k)j}A_{h(l,k)j}$ is the time of computing the load parts of tasks $T_l,\ldots,T_n$, sent to the processor $h(l,k)$, activated as the k-th for task T_l. Thus, inequalities (4.2.1) ensure that computations complete before the end of the schedule. By constraints (4.2.2) all tasks are fully processed. Let us consider an example.

Example 1. $m=3, n=2, |\mathcal{P}_j|=m, h(j,i)=i$, for $j=1,2$, i.e., all processors are used, and the order of processor activation coincides with processor numbers for both tasks. Processors are identical: $\forall_{i,j}A_{ij}=1$, $\forall_{i,j}C_{ij}=1$, $\forall_{i,j}S_{ij}=1$. $V_1=32, V_2=2$. For these values the solution from (4.2.1)-(4.2.2) is: $\alpha_{11}=18.5, \alpha_{21}=$

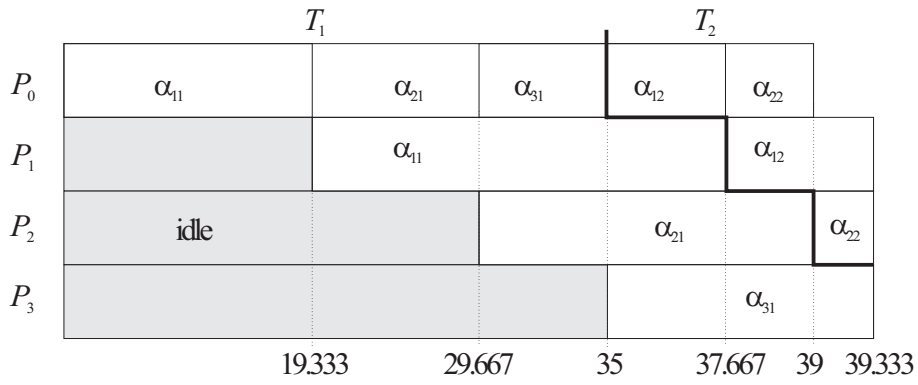

Figure 4.6: Optimal schedule for Example 1 (does not preserve proportion).

$9.75, \alpha_{31} = 3.75, \alpha_{12} = 2.0, \alpha_{22} = 0, \alpha_{32} = 0, C_{max} = 40$. The last two communications of T_2 contain no load, because $\alpha_{22} = 0, \alpha_{32} = 0$, but still contribute startup times $S_1 = S_2 = 1$. Thus, this is not the best activation order, and processor P_3 need not be used in processing T_2. After removing P_3 from $\mathcal{P}_2$ we get from (4.2.1)-(4.2.2) the optimum solution: $\alpha_{11} \approx 18.333, \alpha_{21} \approx 9.333, \alpha_{31} \approx 4.333, \alpha_{12} \approx 1.667, \alpha_{22} \approx 0.333, C_{max} \approx 39.333$, shown in Fig.4.6. Exclusion of both P_3, and P_2 from processing T_2 does not reduce schedule length anymore. □

Observe that in the optimum schedule for Example 1 computations on T_1 do not finish on all processors at the same time. It demonstrates that simultaneous completion of the computations on all processors for all tasks is not necessary for the optimality of the solution. This observation departs from the standard situation in single divisible load processing. Let us analyze one more example.

Suppose that tasks are of equal size $\forall_{T_j} V_j = V$, processors are identical, and $\forall_{T_j} \mathcal{P}_j = \mathcal{P}$, i.e., each task uses all processors. We experimentally studied patterns

that appear in the optimal load distribution under the above conditions. When communication delays are big in comparison with computing time then not all processors are exploited. It is the case when $C \gg \frac{A}{m}$. When communication delays are of similar order as computations ($C \approx \frac{A}{m}$) then load of each task is distributed nearly equally between the processors. The exceptions are the leading and trailing tasks. In the leading tasks distribution is unequal so that waiting for the first load chunk to process is minimized. In the trailing tasks the distribution is also unequal such that processors stop computing at the same time. This is demonstrated in Fig.4.7 where changes of α_{ij} from task to task are shown for $m = 3$. Each line in Fig.4.7 and Fig.4.8 represents the load from the consecutive tasks assigned to a certain processor. On the contrary, when communication delays are short in comparison with computing times, e.g. when $C \ll \frac{A}{m}$, then total load of all tasks is distributed nearly equally between the processors, but computations of each task are concentrated on one processor. This is demonstrated in Fig.4.8. Such a situation is not very comfortable for a user of a parallel application because a distribution optimal globally (for all tasks) is not necessarily a solution which is using parallelism.

It was assumed in (4.2.1)-(4.2.2) that the computation completion times are arbitrary. If simultaneous completion is required, then a linear programming formulation can be given to deal with the simultaneous completion. Let $\tau_l \geq 0$ denote the completion time of computations on task T_l. The following linear program solves the case

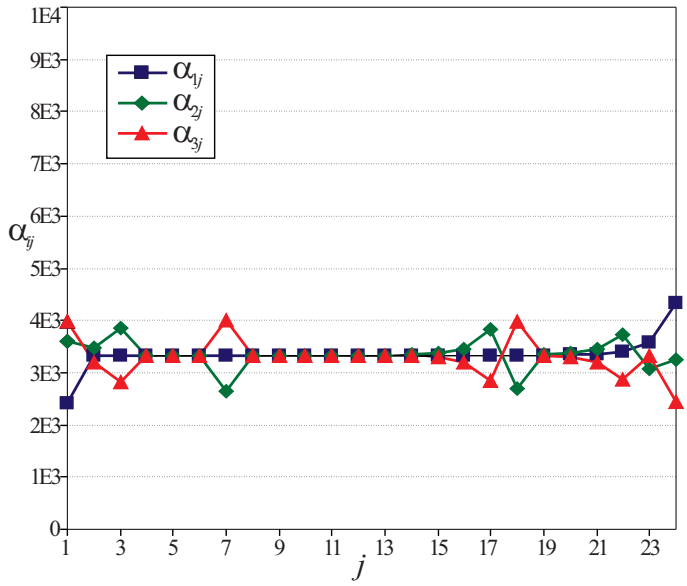

Figure 4.7: Distribution of the load (α_{ij}) vs task number (j); $m = 3, n = 24, V = 1E4, C = S = 1,\ A = 3$.

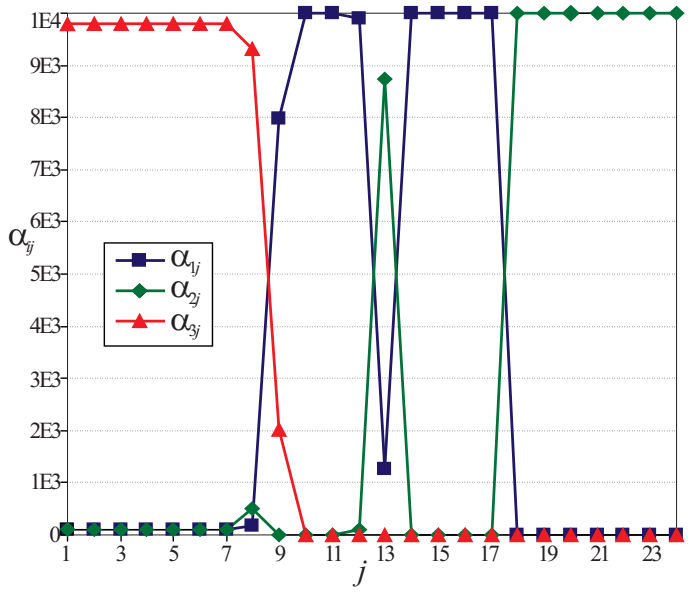

Figure 4.8: Distribution of the load (α_{ij}) vs task number (j); $m = 3, n = 24, V = 1E4, C = S = 1,\ A = 1\text{E}2$.

with simultaneous completion:

minimize C_{max}

subject to:

$$\tau_{l-1} + \alpha_{h(l,k)l} A_{h(l,k)l} \leq \tau_l$$

$$l = 2, \ldots, n,\ k = 1, \ldots, |\mathcal{P}_j| \tag{4.2.3}$$

$$\sum_{j=1}^{l-1} \sum_{i=1}^{|\mathcal{P}_j|} (S_{h(j,i)j} + \alpha_{h(j,i)j} C_{h(j,i)j}) + \sum_{i=1}^{k} (S_{h(l,i)l} + \alpha_{h(l,i)l} C_{h(l,i)l}) + + \sum_{j=l}^{n} \alpha_{h(l,k)j} A_{h(l,k)j} \leq \tau_l$$

$$l = 1, \ldots, n,\ k = 1, \ldots, |\mathcal{P}_j| \tag{4.2.4}$$

$$\tau_n = C_{max} \tag{4.2.5}$$

$$\sum_{i \in \mathcal{P}_j} \alpha_{ij} = V_j \quad j = 1, \ldots, n \tag{4.2.6}$$

By inequalities (4.2.3), the computations of task T_l can be feasibly performed in interval $[\tau_{l-1}, \tau_l]$. Inequalities (4.2.4) ensure that communications and computations of task T_l are completed by time τ_l. By (4.2.5) the end of the last task is also the end of the schedule. The tasks are fully processed by (4.2.6).

Let us now return to Example 1. For linear program (4.2.3)-(4.2.6) a solution $\alpha_{11} = 19, \alpha_{21} = 9, \alpha_{31} = 4, \alpha_{12} = 1, \alpha_{22} = 1, C_{max} = 40$ is obtained, which is longer than the one presented in Fig.4.6.

4.2.2 Fixed activation order, with result returning

The methods used in the previous section can be extended to deal with the returning of the results. Without loss of generality we assume that tasks are executed in the order of their numbers, and this is also the order of sending the loads from the originator to the processors. Yet, the set of processors used by a task, the sequence of processor activation, and the sequence of result collection can be arbitrary. Let us denote by:

$T_{j^{dd}}$ - the last task which distributes the load before task T_j distributes its load,

$T_{j^{rd}}$ - the last task which returns its results to the originator before task T_j distributes its load,

$T_{j^{dr}}$ - the last tasks which distributes the load before task T_j returns its results,

$T_{j^{rr}}$ - the last task which returns its results to the originator before returning task T_j results,

t_j^D - the time moment when distribution of task T_j load starts,

t_j^R - the time moment when collection of task T_j results starts,

t_{ij} - the time moment when P_i finishes computing load α_{ij},

$h(j,i)$ - the number of the i-th processor activated for task T_j,

$g(j,i)$ - the processor returning results as i-th in the sequence, for task T_j.

Optimum distribution of the load can be found by solving the following linear program:

minimize C_{max}

subject to:

$$t_j^D \geq t_{j^{dd}}^D + \sum_{i=1}^{|\mathcal{P}_{j^{dd}}|} (S_{h(j^{dd},i)j^{dd}} + \alpha_{h(j^{dd},i)j^{dd}} C_{h(j^{dd},i)j^{dd}}) \;\; j = 1, \ldots, n, \tag{4.2.7}$$

$$t_j^D \geq t_{g(j^{rd},k)j^{rd}} + \sum_{i=k}^{|\mathcal{P}_{j^{rd}}|} (S_{g(j^{rd},i)j^{rd}} + \beta_j \alpha_{g(j^{rd},i)j^{rd}} C_{g(j^{rd},i)j^{rd}})$$

$$j = 1, \ldots, n, \; k = 1, \ldots, |\mathcal{P}_{j^{rd}}| \tag{4.2.8}$$

$$t_j^R \geq t_{j^{dr}}^D + \sum_{i=1}^{|\mathcal{P}_{j^{dr}}|} (S_{h(j^{dr},i)j^{dr}} + \alpha_{h(j^{dr},i)j^{dr}} C_{h(j^{dr},i)j^{dr}}) \;\; j = 1, \ldots, n \tag{4.2.9}$$

$$t_j^R \geq t_{g(j^{rr},k)j^{rr}} + \sum_{i=k}^{|\mathcal{P}_{j^{rr}}|} (S_{g(j^{rr},i)j^{rr}} + \beta_j \alpha_{g(j^{rr},i)j^{rr}} C_{g(j^{rr},i)j^{rr}})$$

$$j = 1, \ldots, n, \; k = 1, \ldots, |\mathcal{P}_{j^{rr}}| \tag{4.2.10}$$

$$t_j^R \geq t_{g(j,k)j} \;\; j = 1, \ldots, n, \; k \in \mathcal{P}_j \tag{4.2.11}$$

$$t_{kj} \geq t_j^D + \sum_{i=1}^{k} (S_{h(j,i)j} + \alpha_{h(j,i)j} C_{h(j,i)j}) + \alpha_{h(j,k)j} A_{h(j,k)j}$$

$$j = 1, \ldots, n, \; k \in \mathcal{P}_j \tag{4.2.12}$$

$$t_{kj} \geq t_{kj^{dd}} + \alpha_{kj} A_{kj} \;\; l = 1, \ldots, n, \; k \in \; \mathcal{P}_j \cap \mathcal{P}_{j^{dd}} \tag{4.2.13}$$

$$C_{max} \geq t_{g(n,k)n} + \sum_{i=k}^{|\mathcal{P}_n|} (S_{g(n,i)n} + \beta_j \alpha_{g(n,i)n} C_{g(n,i)n}) \;\; k = 1, \ldots, |\mathcal{P}_n| \tag{4.2.14}$$

$$\sum_{i \in \mathcal{P}_j} \alpha_{ij} = V_j \quad j = 1, \ldots, n \tag{4.2.15}$$

Inequalities (4.2.7), (4.2.8) guarantee that distribution of T_j load may take place only after all the preceding communication operations. Similarly, (4.2.9), (4.2.10) ensure that collection of T_j results follow after the preceding communication operations. By inequalities (4.2.11) returning of the results can start when the results are available.

Computation of some part of the task T_j on processor k can finish only after receiving the load part and processing it by inequalities (4.2.12). By inequalities (4.2.13) computations exploiting the same processor do not overlap. The end of the schedule is set by the end of returning the results of the last task by inequalities (4.2.14). All the load is processed by equation (4.2.15).

4.2.3 Computation cost, processor availability, and memory limits

The linear program for the calculation of the load distribution can be generalized to handle computation cost, restrictions on processor availability, and memory sizes. We will give the formulation as a proof of feasibility of such an extension. For the sake of simplicity we assume that processor availability applies to all the tasks, and that the results are not returned to the originator. It is assumed that the load may be sent to the processors before actually starting the computations. The received load is buffered in the external storage (e.g. hard disk) until processing the load of a task is started. The external storage is relatively cheap, therefore it is possible to accumulate big loads on it. On the other hand, core memories are more expensive and smaller. The core memory size restrictions apply when an application is running. Therefore, memory restrictions apply on per-task base: no chunk i of a task j may have size α_{ij} greater than the memory limit B_{ij} on processor i. Let $\overline{G}$ denote the limit on the costs of computation. The optimum distribution of the load can be found using the

following linear program:

minimize C_{max}

subject to:

$$\sum_{j=1}^{a-1}\sum_{i=1}^{|\mathcal{P}_j|}(S_{h(j,i)j}+\alpha_{h(j,i)j}C_{h(j,i)j})+\sum_{i=1}^{k}(S_{h(a,i)a}+\alpha_{h(a,i)a}C_{h(a,i)a})+ +\sum_{j=a}^{n}\alpha_{h(a,k)j}A_{h(a,k)j}\leq C_{max}\quad a=1,\ldots,n,\ k=1,\ldots,|\mathcal{P}_j| \tag{4.2.16}$$

$$\sum_{j=1}^{a-1}\sum_{i=1}^{|\mathcal{P}_j|}(S_{h(j,i)j}+\alpha_{h(j,i)j}C_{h(j,i)j})+\sum_{i=1}^{k}(S_{h(a,i)a}+\alpha_{h(a,i)a}C_{h(a,i)a})+ +\sum_{j=a}^{n}\alpha_{h(a,k)j}A_{h(a,k)j}\leq d_{h(a,k)}\quad a=1,\ldots,n,\ k=1,\ldots,|\mathcal{P}_j| \tag{4.2.17}$$

$$r_{h(a,k)}+\sum_{j=a}^{n}\alpha_{h(a,k)j}A_{h(a,k)j}\leq C_{max}\quad a=1,\ldots,n,\ k=1,\ldots,|\mathcal{P}_j| \tag{4.2.18}$$

$$r_{h(a,k)}+\sum_{j=a}^{n}\alpha_{h(a,k)j}A_{h(a,k)j}\leq d_{h(a,k)}\quad a=1,\ldots,n,\ k=1,\ldots,|\mathcal{P}_j| \tag{4.2.19}$$

$$\sum_{j=1}^{n}\sum_{k=1}^{|\mathcal{P}_j|}(f_{h(j,k)j}+\alpha_{h(j,k)j}l_{h(j,k)j})\leq\overline{G} \tag{4.2.20}$$

$$\sum_{i\in\mathcal{P}_j}\alpha_{ij}=V_j\quad j=1,\ldots,n \tag{4.2.21}$$

$$\alpha_{ij}\leq B_{ij}\quad i=1,\ldots,m,\ j=1,\ldots,n \tag{4.2.22}$$

The sum of the first two components of the left side of inequalities (4.2.16), (4.2.17) is the end of the communication time for task a sent to the k - the processor used by this task. The last component of the left side of inequalities (4.2.16)-(4.2.19) is the computation time of the parts of the tasks assigned to the same processor as task j part k and after task j. Thus, the left sides of inequalities (4.2.16), (4.2.17) express the time when the computations finish, provided that their start is determined by the

end of the communications. The first component of (4.2.18), (4.2.19) is the earliest start time of computations, when it is determined by the restricted availability of a processor. Thus, inequalities (4.2.18), (4.2.19) express the time when the computations finish, provided that their start is determined by the availability of the processor. Inequalities (4.2.16), (4.2.18) guarantee that computations on each processor finish before the end of the schedule. By inequalities (4.2.17), (4.2.19) computations finish also before the end of processor availability. Constraint (4.2.20) ensures that the limit on the computation cost is not exceeded. No task uses more memory than the amount admissible for the given task on the processor by (4.2.22).

4.2.4 Continuous computing

In this section we assume simultaneous completion, use of all processors by each task, and negligible result returning time. Moreover, it is assumed that the tasks occupy processors continuously from the start of computations on the first task, till the end of the last task. We will call this situation *continuous computing*. An example of a continuous computing is shown in Fig.4.9. Let us start with some observations.

Observation 15. *When computing is continuous, only the schedule for the first task decides on the length of the whole schedule.*

Proof. Since all processors are used by each task, selection of the processor set is immaterial. With the exception of the first task, the sequence of processor activation can be arbitrary because processors are used in the same interval due to continuous

a)

b)

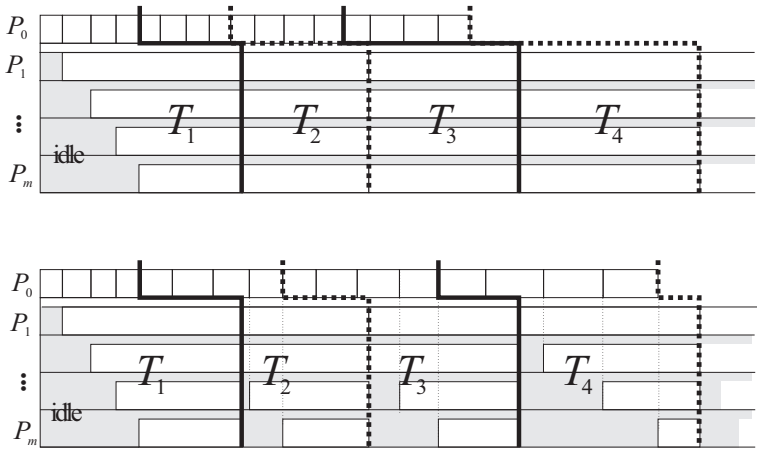

Figure 4.9: Example of a) continuous, b) non-continuous computing.

computing and simultaneous completion. With the exception of the first task the load assigned to processor P_i for task T_j is $\alpha_{ij} = \frac{V_j}{A_{ij} \sum_{l=1}^{m} \frac{1}{A_{lj}}}$, and a decision on task chunk sizes is not necessary. □

Note that also the sequence of the tasks matters because only some task sequences may result in continuous computing (cf. the proofs of Theorem 12, and Observation 13).

Observation 16. *If the first task has the shortest possible completion time, and computing is continuous, then the schedule for all tasks is optimum.*

Proof. The schedule cannot be shorter because all processors work in parallel since the end of the first task, and the first task is completed in the shortest possible time. □

Continuous computing is possible when the load of any task T_j is distributed to the processors before starting of the computations on T_j. This leads to the following greedy approach to the construction of continuous schedules:

1. For $i = 1$ to n do: select T_i as the first task, and construct for it an optimum

schedule $\mathcal{S}_i$.

2. Select the shortest schedule $\mathcal{S}$ in $\mathcal{S}_1,\ldots,\mathcal{S}_n$. Set $\mathcal{T}' = \mathcal{T} - \{T_i\}$;

3. While $\mathcal{T}' \neq \emptyset$ do:

3.1. select task T_j which communication interval $\sum_{l=1}^{m}(S_{lj}+C_{lj}\alpha_{lj})$ fits in the interval between the end of communication and the end of computation of the preceding task, and which maximizes interval between T_j ends of communications and computations, i.e., which maximizes $A_{1j}\alpha_{1j} - \sum_{l=1}^{m}(S_{lj} + C_{lj}\alpha_{lj})$, where $\alpha_{lj} = \frac{V_j}{A_{lj}\sum_{k=1}^{m}\frac{1}{A_{kj}}}$; if there is no task satisfying the above conditions then stop;

3.2. append T_j to the end of schedule $\mathcal{S}$; set $\mathcal{T}' = \mathcal{T}' - \{T_j\}$.

If the resulting schedule has continuous computing property, then it is optimal by Observation 16. Beyond the construction of the optimum schedule for the first task the above algorithm can be implemented to run in $O(nm + n^2)$ time.

Unfortunately, it is hard to claim that the above algorithm builds optimum schedules in polynomial time in general. To our best knowledge, the complexity of scheduling single divisible load (step 1 in the above algorithm) on heterogeneous star remains open in the general case. Two decisions must be made: the set of used processors, and the sequence of their activation must be selected. If $\forall_{i,j} S_{ij} = 0$ then it can be shown that all processors take part in computations, and they should be activated according to increasing C_{ij} for task T_j [2, 7, 10]. If $\forall_{i,j} S_{ij} \neq 0$ then it can be shown

that processors participating in the computation should be activated according to increasing C_{ij} for task T_j [2]. Yet, it is not known which processors should be used (cf. Fig.4.4). Hence, it is not guaranteed that the above method constructs a schedule with continuous computing property if such a schedule exists.

We are going to propose sufficient conditions under which an optimum schedule can be constructed by the above algorithm. This means that in the set of optimum schedules with continuous computing there is a subset satisfying our conditions. Suppose the processors are identical. Executing the tasks according to the increasing sizes (V_j) will be called SPT (for Shortest Processing Time) sequence. Let us assume that tasks are ordered according to SPT rule, i.e., $V_1 \leq V_2 \leq \cdots \leq V_n$.

Theorem 17. *If computing is continuous, and* $\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{\frac{A}{m} - C}$ *then SPT maximizes the interval between the completion of the task communication, and starting of its computations on identical processors.*

Proof. The requirement $\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{\frac{A}{m} - C}$ can be rewritten as $\forall_{T_j \in \mathcal{T}} \frac{AV_j}{m} > Sm + CV_j$, which means that load distribution time is shorter than computation using all processors in the same interval. This requirement should be satisfied by real parallel applications which have high computing demands. The proof is based on pairwise interchange.

Consider two tasks T_i, T_j executed continuously one after another. For the simplicity of presentation let us denote by $e = Sm + CV_i, E = \frac{AV_i}{m}, f = Sm + CV_j, F = \frac{AV_j}{m}$. Note that $e < E, f < F$. A task preceding T_i, T_j completes its communication x_0

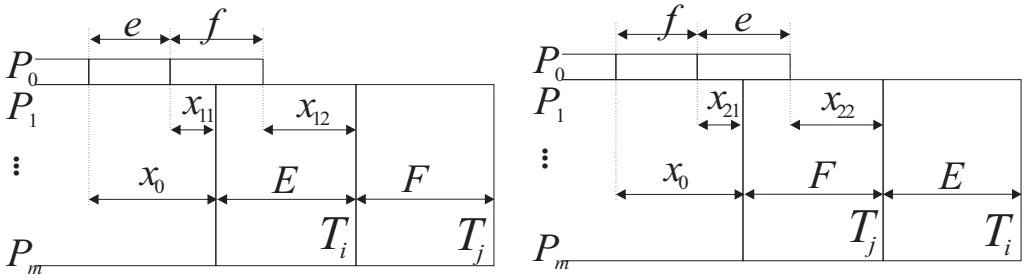

Figure 4.10: Illustration to the proof of Theorem 17.

units of time before the end of its computations (cf. Fig.4.10). Suppose that T_i precedes T_j (cf. Fig.4.10a). The time from the end of T_i communication to the start of T_i computation is $x_{11} = x_0 - e$. The length of the interval since the end of T_j communication till the beginning of T_j computation is $x_{12} = x_0 + E - e - f$. The worse of the two interval lengths is $\min\{x_{11}, x_{12}\}$. Now suppose that the order of the two tasks is inverted. Then the lengths of the intervals are (cf. Fig.4.10b) $x_{21} = x_0 - f, x_{22} = x_0 + F - e - f$. The smaller of the intervals is $\min\{x_{21}, x_{22}\}$. Let us analyze the conditions under which it is better to execute the two tasks in the order (T_i, T_j), than in the order (T_j, T_i), i.e. when $\min\{x_{11}, x_{12}\} > \min\{x_{21}, x_{22}\}$. Note that changing the order of the two tasks does not influence the rest of the schedule.

Let us assume that $\min\{x_{11}, x_{12}\} = x_{11}$ then $x_0 - e < x_0 + E - e - f$, hence $f < E$. Suppose that $\min\{x_{21}, x_{22}\} = x_{21}$ then $x_0 - f < x_0 + F - e - f$, hence $e < F$. Sequence (T_i, T_j) is better when $x_{11} = x_0 - e > x_{21} = x_0 - f$ from which we get $e = Sm + V_iC < f = Sm + V_jC$, and $V_i < V_j$. Thus, SPT sequence is desired. Suppose that $\min\{x_{21}, x_{22}\} = x_{22}$ then $e > F$. Sequence (T_i, T_j) is better

if $x_{11} = x_0 - e > x_{22} = x_0 + F - e - f$, and $f > F$ which is in contradiction with $f = Sm + CV_j < \frac{AV_j}{m} = F$.

Let us assume that $\min\{x_{11}, x_{12}\} = x_{12}$ then $f > E$. If $\min\{x_{21}, x_{22}\} = x_{21}$ then $e < F$. Sequence (T_i, T_j) is better when $x_{12} = x_0 + E - e - f > x_{21} = x_0 - f$, from which we get $E > e$. Altogether we have $e < E < f$, and $V_i < V_j$. Finally, if $\min\{x_{21}, x_{22}\} = x_{22}$, and $e > F$. If we put together the conditions for this case we have: $e < E, e > F, f < F, f > E$. Using $e < E < f$, and $e > F > f$ we get a contradiction. We may conclude that by using interchanges between all pairs of consecutive tasks (excluding the first task), any continuous computing sequence can be changed to a continuous computing SPT sequence. □

Thus, if it is possible to maintain continuous computing at all, then SPT will also do it, provided that $\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{\frac{A}{m} - C}$. The conditions of the optimality of SPT sequence in continuous computing are the following.

Theorem 18. *SPT is the optimum task sequence on identical processors if* $\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{\frac{A}{m} - C}$, *and* $x_j > Sm + V_{j+1}C$ *for* $j = 1, \ldots, n-1$, *where* $x_1 = \frac{CV_1 - \frac{SA}{C}[(1+\frac{C}{A})^m - (1+\frac{C}{A})] + (m-1)S}{(1+\frac{C}{A})^m - 1}$, $x_j = x_{j-1} + \frac{V_j A}{m} - Sm - V_j C$ *for* $j = 2, \ldots, n-1$.

Proof. If it is possible to maintain continuous computing on identical processors at all, then according to Theorem 17, SPT sequence will also have this property because SPT maximizes the distance between task communication completion and computation start. By Observation 16 the first task must be finished in the shortest possible time. On identical processors task T_1 with the smallest load V_1 satisfies

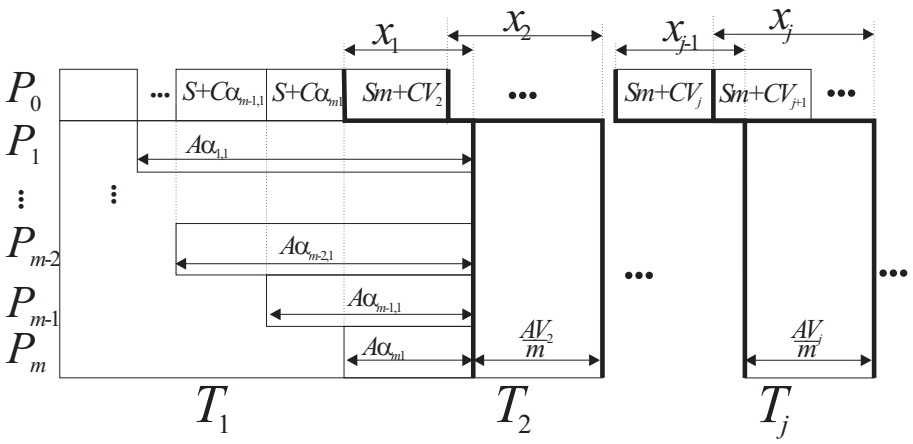

Figure 4.11: Illustration to the proof of Theorem 18.

this condition. Hence, the SPT task sequence is optimal among the schedules with continuous computing property on identical processors.

It still remains to ensure that continuous computing is possible. It is the case if $x_j > Sm + V_{j+1}C$, for $j = 1, \ldots, n-1$, where x_j is the time between end of task T_j communication, and the start of its computation (cf. Fig.4.11). This condition demands that communication of T_{j+1} finishes before its computation has to start. The length x_j of the interval for the communication of T_{j+1} is equal to $x_j = x_{j-1} + \frac{V_j A}{m} - Sm - CV_j$ for $j = 2, \ldots, n-1$. Length x_1 of the first interval is $x_1 = A\alpha_{m1}$. Now we calculate α_{m1}. Since computations on T_1 must finish simultaneously on all processors we have (cf. 1.4.1).

$$A\alpha_{i1} = S + \alpha_{i+1,1}(A + C) \qquad i = 1, \ldots, m-1 \tag{4.2.23}$$

α_{i1}, for $i = 1, \ldots, m-1$, can be expressed as a function of α_{m1}:

$$\alpha_{i1} = \alpha_{m1}(1 + \frac{C}{A})^{m-i} + \frac{S}{A}\sum_{j=0}^{m-i-1}(1 + \frac{C}{A})^j. \tag{4.2.24}$$

The size of the first task is:

$$V_1 = \sum_{i=1}^{m} \alpha_{m1}(1+\frac{C}{A})^{m-i} + \frac{S}{A}\sum_{i=1}^{m-1}\sum_{j=0}^{m-i-1}(1+\frac{C}{A})^{j} \tag{4.2.25}$$

from which we derive:

$$x_1 = A\alpha_{m1} = \frac{V_1 - \frac{SA}{C^2}[(1+\frac{C}{A})^m - (1+\frac{C}{A})] + \frac{(m-1)S}{C}}{\frac{1}{C}[(1+\frac{C}{A})^m - 1]}. \tag{4.2.26}$$

□

We finish our considerations of continuous computing with an observation. Continuous schedules have very simple structure. This seems to hold a promise of a simple algorithm constructing optimum continuous schedules. Unfortunately, it turned out that multiple divisible load scheduling problem is complex enough, such that instances must be very restricted to construct guaranteed optimum continuous schedule.

4.2.5 $m = 1$

Observation 19. *If result returning time is negligible, then multiple divisible load scheduling problem for one* $(m = 1)$ *unrelated processor is solvable in* $O(n \log n)$ *time by Johnson's algorithm [28].*

Proof. If the results are not returned, and only one machine $(m = 1)$ is available, then execution of a task reduces to two operations: communication operation involving originator P_0, followed by computation operation involving P_1. This situation is equivalent to two-machine flowshop. Two-machine flowshop is solvable in $O(n \log n)$ time by Johnson's algorithm [28] (or see e.g. [13, 35]). □

For the completeness of the presentation let us note that in our case Johnson's algorithm divides the set of tasks into two subsets: $\mathcal{T}_1$ comprising the tasks for which $S_{1j} + C_{1j}V_{1j} < A_{1j}V_{1j}$, and set $\mathcal{T}_2$ comprising the remaining tasks. Tasks in $\mathcal{T}_1$ are executed in the order of increasing $S_{1j}+C_{1j}V_{1j}$, while tasks in $\mathcal{T}_2$ are ordered according to decreasing $A_{1j}V_{1j}$. $\mathcal{T}_1$ is executed first.

Though this special case may seem trivial, it represents practical situations when parallel computations both start and complete in roughly the same time on all processors. In such situations all processors are working in parallel, and behave like a single processing facility, i.e., a single processor.

4.3 Approximability

In this section we study the bounds on the quality of approximation algorithms for multiple divisible load scheduling problem. By a greedy heuristic we mean an algorithm which is not unnecessarily delaying communications and computations. This means that if there is some load to be distributed and communication medium is available, then the load is immediately distributed. If there is some load already at a processor and the processor is free, then the computation on the load is immediately started.

Theorem 20. *Length C^H_{max} of a schedule built by any greedy heuristic H solving multiple divisible load scheduling problem on identical processors satisfies:*

$$\frac{C^H_{max}}{C^*_{max}} \leq 2m,$$

where C^*_{max} is the optimum schedule length.

Proof. Intervals of two types can be distinguished in any schedule for our problem: Intervals of total length E_C when initiator performs communications, and intervals of total length E_A when initiator does not perform any communications because all processors compute. In the case of identical processors $E_C = \sum_{j=1}^{n}(\sum_{P_i \in \mathcal{P}_j} S + CV_j)$. Note that $nS + C\sum_{j=1}^{n} V_j \leq C^*_{max}$ because each load must be sent. In the worst case some heuristic may activate all processors while only a single processor is necessary for each task. Consequently, $\sum_{j=1}^{n}\sum_{P_i \in \mathcal{P}_j} S - nS = S\sum_{j=1}^{n}(|\mathcal{P}_j| - 1) \leq Sn(m-1) \leq (m-1)C^*_{max}$. Some heuristic may also tend to use less processors than necessary. In the worst case $|\mathcal{P}_j| = 1$, and $E_A \leq \sum_{j=1}^{n} AV_j$. Note that $\sum_{j=1}^{n} \frac{AV_j}{m} \leq C^*_{max}$. Hence $E_A \leq mC^*_{max}$. Altogether we have $C^H_{max} = E_C + E_A \leq C^*_{max} + (m-1)C^*_{max} + mC^*_{max}$.

□

Let us note that from the worst case analysis in Section 2.3 it follows that a solution built by an arbitrary greedy algorithm in heterogeneous system can be arbitrarily bad.

The results of Theorem 20 can be further strengthened. If $S = 0$, then in the above proof $\sum_{j=1}^{n}\sum_{P_i \in \mathcal{P}_j} S - nS = 0$, and the ratio of schedule lengths can be narrowed to $\frac{C^H_{max}}{C^*_{max}} \leq m + 1$. If $\forall_{T_j \in \mathcal{T}} CV_j + mS < \frac{V_j A}{m}$, then $E_C \leq \sum_{j=1}^{n}(mS + CV_j) \leq \sum_{j=1}^{n} \frac{V_j A}{m} \leq C^*_{max}$. Consequently $\frac{C^H_{max}}{C^*_{max}} \leq m + 1$.

In the latter case a better bound can be obtained by a heuristic CC attempting to build a schedule with continuous computing property: Divide the load of each task into m equal parts and send them to the processors. For each task start computations

synchronously on all processors as soon as all processors have received their share of the load. Since all processors compute in parallel we have $E_A \leq C^*_{max}$. Hence, $E_C + E_A \leq 2C^*_{max}$, and $\frac{C^{CC}_{max}}{C^*_{max}} \leq 2$.

4.4 Identical processors, identical tasks

It follows from Theorem 14, and results in [21], that the only problem with non-zero startup time which may admit polynomial solvability is scheduling of identical tasks on identical processors. Now it is assumed that returning the results lasts negligible time and processors can compute and communicate simultaneously, memory size is sufficiently big. We assume that all tasks have load V for $j = 1, \ldots, n$, and that also $C = 0$. Observe that the parameters of an instance in this special case are A, S, m, V, n, and a polynomial time algorithm must have a running time polynomial in $\log A, \log S, \log m, \log V, \log n$. On the other hand, sheer assignment of the tasks to the processors requires pseudopolynomial time $\Omega(n)$. Furthermore, it is not known if our problem is in class **NP** at all. If it belongs to **NP** then a solution, which is a sequence of communications, should be a string of a length polynomial in $\log A, \log S, \log m, \log V, \log n$. Intuitively, it may be expected that in our case a polynomial algorithm, within the given time, can only identify patterns of the optimum schedule. Unfortunately, the results of Section 4.4.2 show no apparent regularity of optimum communication sequences, and such pattern solutions may be hard to be

found. The compact description of the problem instance causes difficulties in classifying complexity. Therefore, we separated this problem into new section in this work.

We will use startup time S as time unit, and will denote by k the computation time of a task on a single processor, i.e. $k = \frac{VA}{S}$. Let us start with some observations on possible patterns of the optimum schedules.

Observation 21. *The earliest time moments when the communications may be finished, and the computations started are integer time instants (0, 1, 2, 3, ...).*

Note that within k time units of some task j computation on a single processor, the load of $\lceil k \rceil$ other tasks can be sent to the processors. Out of this $\lceil k \rceil - 1$ tasks can be sent to other processors than j, but the $\lceil k \rceil$-th task after j may be executed on the same processor as task j, because at the time of receiving the load for $\lceil k \rceil$-th task after j, the processor computing task j will be free (Fig.4.12). We will say that the number of processors m is *not bounding* if the number of processors effectively used follows from the number of communications which may be performed in the interval of a single task computation, i.e. when $m \geq \lceil k \rceil$. In the opposite case we will say that the number of processors is *bounding*.

Observation 22. *When the number of processors is bounding, and processors are computing all the time since the earliest possible activation time to the simultaneous completion of the computations, then schedule is optimum.*

Proof. Such a schedule cannot be made shorter because there is no idle time on the processors which could be eliminated. □

Observation 23. *When processor number is not bounding, and initiator is sending the load all the time with the exception of an interval not longer than 1, at the end of the schedule, and the processors computing in this last interval finish the computations simultaneously, then the schedule is optimum.*

Proof. Let us assume, for the time being, that the number of processors is infinite. If the initiator sends the load to a new processor in each time unit, and each task, with the exception of at most $\lfloor \frac{k}{2} \rfloor$ trailing tasks (this number is explained in the following theorem), is executed on a different processor, then these tasks cannot be completed earlier (see Fig.4.12a) because initiator is communicating all the time, and no additional processor can be activated for some task without delaying some other task. At the end of the schedule originator may be idle at most k time units after sending the load of the last task. These time units may be used to activate additional processors, and parallelize execution of the trailing tasks. If a new processor is activated in each time unit, with the exception of an interval of length at most 1 at the end of the schedule, then no more processors can be activated at all. If the computation on the trailing tasks activates processors at the earliest possible moments, processors work without idle time, and finish computations simultaneously, then the trailing tasks cannot be completed earlier. Observe, that this reasoning does not require infinite number of processors. The above schedule can be folded to $\lceil k \rceil$ processors (see Fig.4.12b). □

It implicitly follows from Observations 22, 23, that there are optimal schedules

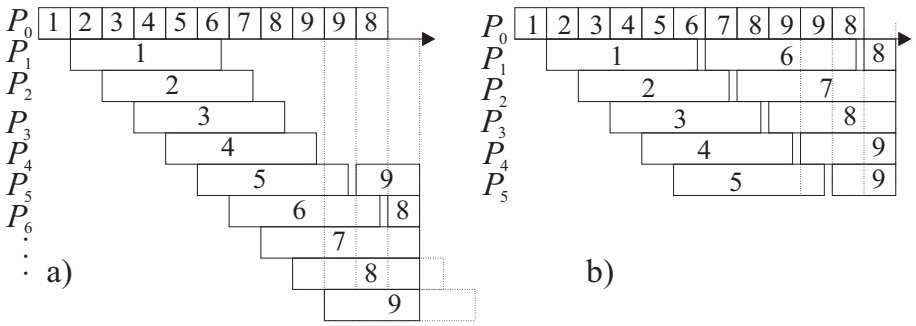

Figure 4.12: Illustration to the proof of Observation 23.

in which processors stop computing simultaneously. In the following discussion we assume $m > 1$ because the case of a single processor is solved by a greedy algorithm.

4.4.1 $n > \min\{\lceil k \rceil, m\}$

In the special case considered here the number of tasks is big enough to perform several repetitions of a certain pattern in a schedule. We discuss it in the following theorems.

Theorem 24. *If $n > \min\{\lceil k \rceil, m\}$ then the optimum schedule for identical tasks on a star of identical processors with $C = 0$ can be determined in polynomial time.*

Proof. The proof analyzes possible cases, and proposes an optimum schedule for each of them. The schedules have two parts: a leading part which we will call a *main sequence*, and a trailing part which will be called a *tail*. In the main sequence initiator sends messages to consecutive processors in time intervals with integer ends. It means that after sending load of some task j to processor P_i in interval $[l, l+1]$, task $j+1$ is sent to processor $P_{(i+1) \bmod (\min\{m, \lceil k \rceil\})+1}$ in interval $[l+1, l+2]$, where l is integer. Task j is processed in interval $[l+1, l+k+1]$, and task $j+1$ in interval

$[l+2, l+k+2]$. W.l.o.g. let $P_{\min\{m,\lceil k\rceil\}}$ be the processor executing the last task from the main sequence. Since $n > \min\{m, \lceil k\rceil\}$ the optimum schedule has both the main sequence, and the tail. Let a denote the number of tasks in the tail, and t the time moment when the load of the last task of the main sequence is sent to the processor. The tail schedules differ in the particular cases. Below we list and analyze possible cases.

1. *k is integer.* $\lceil k\rceil = k$

1.1. *Processor number is not bounding.* Since $m \geq k$, the number of usable processors is k and it is determined by the number of communications which can be done during the computation of a single task. The schedules proposed in this case are optimum by Observation 23.

1.1.1. *k is odd.* In this case $C_{max} = t + k$, and the tail comprises $a = (k-1)/2$ trailing tasks. Task $n-a+i$, for $i = 1, \ldots, a$, is sent to processors P_i, P_{k-i} in intervals $[t-1+i, t+i], [t+k-i-1, t+k-i]$, respectively. It is processed on P_i, P_{k-i} in intervals $[t+i, t+k]$, $[t+k-i, t+k]$, respectively. Here $C_{max} = t+k$ (see Fig.4.13a.).

1.1.2. *k is even.* In the current case $a = k/2$, $C_{max} = t+k+\frac{1}{2}$. Task $n-a+i$ is sent to processors P_i, P_{k-i+1} in intervals $[t-1+i, t+i], [t+k-i, t+k-i+1]$, respectively. It is processed on P_i, P_{k-i} in intervals $[t+i, t+k+\frac{1}{2}]$, $[t+k-i+1, t+k+\frac{1}{2}]$, respectively. (see Fig.4.13b.)

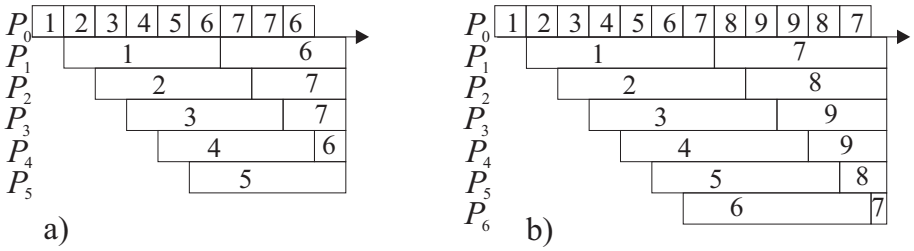

Figure 4.13: Proof of Theorem 24. Processor number is not bounding.

1.2. *Processor number is bounding.* We have $m < k$. The schedules proposed here are optimum by Observation 22.

1.2.1. $m \leq 3$. Here a=1. The last task uses all m processors. The communications to them can be feasibly made in interval $[t, t+m]$ because $m < k$, $m \leq 3$ and k is integer. Processing time available on the processors until the end of the main sequence is at most $3 \leq k$ (because $1 < m < k$ and k is integer) which can be consumed for computations of a single task. The schedule of the tail simultaneously finishes on all processors (Fig.4.14a).

1.2.2. $m > 3$ *and odd.* $a = m, C_{max} = t + (m+1)/2 + k$. The first $(m+1)/2$ tasks continue the schedule of the main sequence tasks, finishing at $t + (m+1)/2 + k$ on processor $P_{(m+1)/2}$. The remaining $(m-1)/2$ tasks are split into two parts and executed on processors placed symmetrically to $P_{(m+1)/2}$. Precisely, task $n-i-1$ (for $i = 1, \ldots, (m-1)/2$) is executed in intervals $[t + (m+1)/2 + i, C_{max}]$, and $[t + (m+1)/2 + k - i, C_{max}]$, on processors $P_{(m+1)/2+i}, P_{(m+1)/2-i}$, respectively (Fig.4.14b).

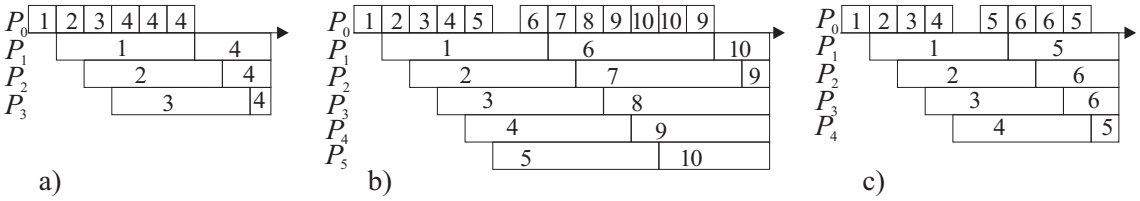

Figure 4.14: Proof of Theorem 24. Processor number is bounding.

1.2.3. $m > 3$ *and even.* $a = m/2, C_{max} = t + k + (k - m + 1)/2$. Tasks of the tail are scheduled symmetrically on processors $P_{m/2-i+1}$ and $P_{m/2+i}$ analogously to case 1.1.2. (Fig.4.14c).

2. *k is fractional*

2.1. *Processor number is not bounding.* Since $m \geq \lceil k \rceil$, schedule length is determined by the processing power which can be engaged during the schedule. The maximum processing power is used when the originator starts computations on inactive processor at the end of every time unit. Though there may be $(\lceil k \rceil - k)$-long idle times on the processors, a schedule constructed as if k were $\lceil k \rceil$ (case 1.1) is optimal by Observation 23 (cf. e.g. Fig.4.12)

2.2. *Processor number is bounding.* The main sequence starts computations on the processors at the earliest time moments by Observation 21, and processors have no idle times. Since the differences between the ends of the computations on the processors at the end of the main sequence are integers, which is dictated by communication delays, it is possible to apply the constructions from case 1.2, such that processors stop computing simultaneously. This schedule is optimal by Observation 22.

Identifying a proper case for the construction of a tail requires checking fractionality, parity of k, and comparing k with m. All this can be done in polynomial time $O(\max\{\log m, \log k\})$. □

4.4.2 $n \leq \min\{\lceil k \rceil, m\}$

The case with too few tasks to have both the main sequence, and a tail is more involved. We start an analysis with the case of non-bounding processor number.

Non-bounding processor number

Since the processor number is not bounding there is no advantage in sending load to a processor more than once. Hence, with each time unit of the schedule computation on one new processor can be activated. In time $x > 1$, $\lfloor x \rfloor$ processors $P_1, \ldots, P_{\lfloor x \rfloor}$ are able to compute for times $x-1, \ldots, x-\lfloor x \rfloor$, respectively. The total processing capacity in $x > 1$ units of time is $\sum_{i=1}^{\lfloor x \rfloor}(x-i)$. For example, in a schedule of length 7, different processors will work for $1, 2, 3, 4, 5, 6$ time units, performing 21 units of work. Our problem of scheduling n tasks with computational demand k can be represented as finding decomposition of numbers $x-1, \ldots, x-\lfloor x \rfloor$ into n sums equal at least k. From this transformation of the scheduling problem to a number theory problem, we can draw a conclusion, that an ideal solution in which all tasks finish computations simultaneously, not always exists. This number-theoretic problem, in turn, may be represented as another scheduling metaphor. We will treat the n tasks

of the original problem, as processors, and processing capacities $x-1,\ldots,x-\lfloor x\rfloor$ of the original processors, as tasks. This transforms our original scheduling problem into problem $P||C_{max}$ with n processors, tasks of length $x-1,\ldots,x-\lfloor x\rfloor$, and a demand that each processor works for at least k units of time. Problem $P||C_{max}$ can be solved by a pseudopolynomial algorithm for a fixed number of processors [37]. Yet, it can be observed that the transformation to problem $P||C_{max}$, though helpful, is not sufficient because the completion time of a task (in the original problem) depends on the number of processors executing it. We demonstrate it in the following example.

Example 2. $n=3, k=11$. Since $3k=33$, schedule length must be at least 8. Assume that $C_{max}=8+x$, and $0\le x\le 1$. With a schedule of this length processors will work for $x, 1+x, 2+x,\ldots,7+x$ time units. Two possible assignments of these computing intervals to tasks are shown in the following table (intervals are identified by their lengths):

solution A			solution B		
task	intervals	sum	task	intervals	sum
1:	$7+x, 2+x, 1+x$	$10+3x$	1:	$7+x, 3+x$	$10+2x$
2:	$6+x, 3+x, x$	$9+3x$	2:	$6+x, 4+x$	$10+2x$
3:	$5+x, 4+x,$	$9+2x$	3:	$5+x, 2+x, 1+x, x$	$8+4x$

Solution A has $x=1$, and $C_{max}=9$. Solution B has $x=\frac{3}{7}$, and $C_{max}=8.75$, because each task requires $k=11$ units of computation time. □

Our problem can be solved by a dynamic programming algorithm. Suppose that

the optimum schedule length is $C + x$, where $0 \leq x \leq 1$. Our method finds the smallest x for which a feasible schedule exists, or determines that no schedule with $x \leq 1$ exists. Let the intervals on the processors be ordered according to their decreasing length, i.e. $C + x - 1, C + x - 2, \ldots, 1 + x, x$ The method is based on the calculation of function

$$u(a_1, b_1, \ldots, a_j, b_j, \ldots, a_n, b_n, i) = \begin{cases} 1 & \text{if a schedule exists which is using} \\ & \text{the first } i \text{ intervals, task } j \text{ receives} \\ & a_j \text{ units of processing in interval} \\ & [1, C] \text{ and uses } b_j \text{ processors in in-} \\ & \text{terval } [C, C + x], \\ 0 & \text{otherwise.} \end{cases}$$

for $a_j = 0, \ldots, k,\ b_j = 0, \ldots, k,\ j = 1, \ldots, n,\ i = 0, \ldots, C$. Function f may be calculated using the following recursive equations:

$$u(a_1, b_1, \ldots, a_n, b_n, 0) = \begin{cases} 1 & \text{if } a_1 = b_1 = \cdots = a_n = b_n = 0 \\ 0 & \text{otherwise,} \end{cases}$$

$$u(a_1, b_1, \ldots, a_j, b_j, \ldots, i + 1) = u(a_1, b_1, \ldots, a_j - C + i + 1, b_j - 1, \ldots, i)$$

for $a_j = 0, \ldots, k,\ b_j = 0, \ldots, k,\ j = 1, \ldots, n,\ i = 0, \ldots, C - 1$. Given the values of $u(a_1, b_1, \ldots, a_n, b_n, C) = 1$ it is possible to calculate schedule length $C_{max}(a_1, b_1, \ldots, a_n, b_n, C) = C + \max_{j=1}^{n}\{\frac{k-a_j}{b_j}\}$, where $x = \max_{j=1}^{n}\{\frac{k-a_j}{b_j}\}$ is the extension of the schedule beyond C. The optimum schedule extension x can be found for such vector $(a'_1, b'_1, \ldots, a'_n, b'_n, C)$ for which $\max_{j=1}^{n}\{\frac{k-a_j}{b_j}\}$ is minimum, i.e.

$$(a'_1, b'_1, \ldots, a'_n, b'_n, C) = \underset{(a_1, b_1, \ldots, a_n, b_n, C)}{argmin} \{\max_{j=1}^{n}\{\frac{k - a_j}{b_j}\} | u(a_1, b_1, \ldots, a_n, b_n, C) = 1\}.$$

If $x > 1$ then the initial value of C was assumed too small. In the opposite case the

optimum assignment of the intervals can be deduced from the values of function u by backtracking from $u(a'_1, b'_1, \ldots, a'_n, b'_n, C)$ to $u(0, \ldots, 0)$.

Observation 25. *There is an algorithm with complexity $O(k^{2k+1} \log k)$, for scheduling identical divisible tasks on a star of identical processors, when $n < min\{\lceil k \rceil, m\}$ and processor number is not bounding.*

Proof. Calculation of function u as describes above requires determining $(k+1)^{2n}C$ values of u. $C \leq 2k$ because a schedule with optimum length $C_{max} \geq 2k+1$ admits sequentially executing two tasks on the same processor. This means that the number of tasks is $n > \min\{m, \lceil k \rceil\}$, which is a different case studied in Section 4.4.1. Since $n < k$, and $C \leq 2k$, the total number of calculations for a certain C does not exceed $(k+1)^{2k}2k$ which is $O(k^{2k+1})$. A proper schedule length C may be found in $\log k$ steps of binary search over values $1, \ldots, 2k$. □

Unfortunately, the dynamic programming method described above is not polynomial. However, we infer from Observation 25 that this case can be solved in constant time if k is fixed. Example optimum communication patterns for $n = 2$, and several small values of k are shown in the following table. Symbol • denotes that a communication must be performed, but the schedule length is the same no matter if we send load of task T_1, or of T_2.

k	communication pattern
1	$(1, 2)$
2	$(1, 2, 2)$
3	$(1, 2, 2)$
4	$(1, 2, 2, 1)$
5	$(1, 2, 2, 1)$
6	$(1, 2, 2, 1, \bullet)$
7	$(1, 2, 2, 1, \bullet)$
8	$(1, 2, 1, 2, 2, 2)$
9	$(1, 2, 2, 1, 1, 2)$
10	$(1, 2, 1, 2, 2, \bullet)$
11	$(1, 1, 2, 2, 2, 2, 2)$
12	$(1, 2, 2, 1, 2, 1, 1)$

No apparent regularity can be observed in the above communication patterns. Thus, it is possible that a polynomial length description of the solution may not exist. This poses a question if the current problem is **NP** at all. For several special cases some features of the optimum solution can be determined.

Observation 26. *If $n = 1$ then optimum schedule length can be found in $O(\log k)$ time as minimum x satisfying $\sum_{i=1}^{\lfloor x \rfloor}(x - i) \geq k$.*

Observation 27. *If i is an integer, $k \in \{2i, 2i + 1\}$ and $1 < n$, $i \leq n \leq k$, then the optimum task distribution sequence is $1, 2, 3, \ldots, n, n, n - 1, \ldots, n - i + 1$.*

Proof. A pattern of processor availability intervals assignment is shown in Fig.4.15a. It is essential in this pattern that $n > 1$, and task n is assigned intervals of length $i + 1, i$. Since the originator is activating a new processor in each time unit, this pattern is optimal by Observation 23.

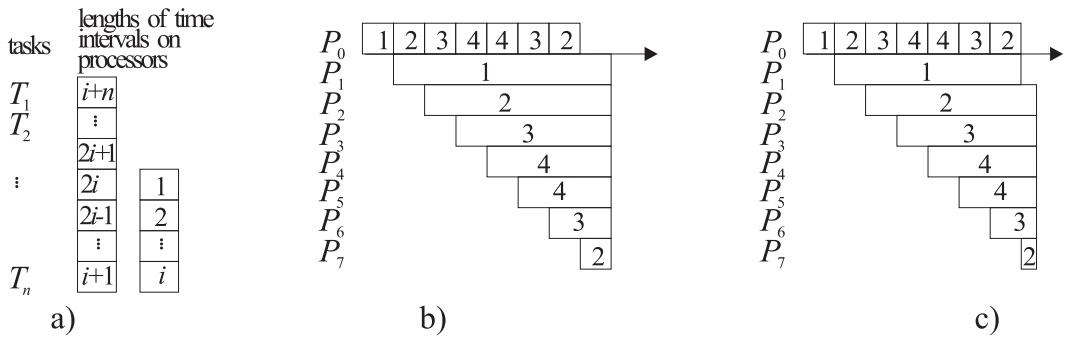

Figure 4.15: Special pattern of load distribution for $n \leq \min\{m, \lceil k \rceil\}$.

The relations between n, k are the following: $i \leq n \leq k = 2i + 1$ where i is a positive integer, with the exception of $i = 1, n = 1$. This pattern can be also applied for even k. If k is even, one should apply a pattern of interval distributions as for $k + 1$, but the last $k = 2i$ processors should stop computing $\frac{1}{2}$ before the end of the schedule dictated by the solution for odd $k + 1$ (see Fig.4.15b for $k = 7$, Fig.4.15c for $k = 6$, and $n = 4$). □

Observation 28. *For $n = 2$ a schedule of length C exists, where C is minimum integer satisfying $2k \leq \sum_{i=1}^{C}(C - i)$.*

Proof. This observation can be shown by demonstrating that it is always possible to partition set $\{C - 1, \ldots, 1\}$ into two subsets A, B such that the sum of elements in A (w.l.o.g.) is equal to k, and at least equal k for set B. It can be shown by induction: For $k = 1, C = 3$, decomposition of set $\{2, 1\}$ is $A = \{1\}$, and $B = \{2\}$. Suppose the observation is satisfied for some $k \geq 1$, and set $\{C - 1, \ldots, 1\}$. Then $A = \{\alpha, \ldots, \omega\}$, and $k = \alpha + \cdots + \omega$. Note that $A \neq \{C - 1, \ldots, 1\}$ because $k = \alpha + \cdots + \omega$, and $\sum_{i=1}^{C}(C - i) \geq 2k$. It follows that also set $\{C' - 1, \ldots, 1\}$,

where $2(k+1) \leq \sum_{i=1}^{C'}(C'-i)$, can be partitioned such that the sum of elements in A' is $k+1$, and in B' is at least $k+1$. Set A' can be constructed by exchanging one of the numbers in A for the one number in $\{C-1,\ldots,1\}-A$ bigger by 1, or adding by $\{1\}$ to A. This is possible because $A \neq \{C-1,\ldots,1\}$. Consequently B' can be constructed by using the remaining elements of $\{C'-1,\ldots,1\}$ because $\sum_{i=1}^{C'}(C'-i) \geq 2(k+1)$. □

Processor number is bounding

Following the discussion from the previous section, let us observe that there seems to be no simple way of determining the number of necessary processors when the processor number is not bounding. Consequently, it is not easy to say, in the general case, whether the processor number is small enough to influence the construction of the schedule. A simple sufficient condition (for processor number restricting the schedule) demands that the amount of required work exceedes the amount of load which can be processed on m processors in at most $m+1$ units of time, i.e. $nk \geq \sum_{i=1}^{m}(m+1-i)$. Note that $m+1$ processors could be activated in $m+1$ units of time if the processor number was not bounding. Since processor number is bounding, unlike in Section 4.4.2, it may be advantageous to send load of different tasks to the same processor. In the following we present a method of calculating schedule length for a given communication pattern.

Minimum schedule length for a given communication pattern

For a given communication pattern the problem of verifying if a schedule of length C exists can be reduced to finding maximum flow in a network. The construction of the network is shown in Fig.4.16. Beyond source s, and sink t there are n nodes in set V_T representing tasks, and at most mn nodes in set V_P which represent positions of the task communications to some processor. Let n_i denote the number of tasks executed on processor P_i. For each communication l to processor P_i a node denoted P_{il}, is created in set V_P (cf. Fig.4.16). The communications are counted from the last message sent to P_i (for which $l = 1$) to the first one (for which $l = n_i$). There are arcs (s, v_j) of capacity k for each node (task) $v_j \in V_T$. For a node $v_j \in V_T$, and $P_{il} \in V_P$ an arc (v_j, P_{il}) is created if task T_j is sent to processor P_i as the l-th message counted from the end communications to P_i. The capacity of arcs (v_j, P_{il}) is not bounded. For a pair of task positions $l, l+1$ on processor P_i, an arc $(P_{il}, P_{i(l+1)})$ is created with the capacity $C - \tau_l$, where τ_l is the time moment when communication l to processor P_i is finished. An arc (P_{in_i}, t) with capacity $C - \tau_{n_i}$ is created for each processor P_i.

A schedule of length C exists for the given network, and communication pattern, if the maximum flow saturates arcs (s, v_j) for each $v_j \in V_T$. A schedule for processor P_i can be constructed analogously to the construction of a schedule for problem $1|r_j, pmtn|C_{max}$. Here, flows $\phi(v_j, P_{il})$ are lengths of the pieces of tasks T_j, ready times are the communication completion times τ_{il} dictated by the communication pattern.

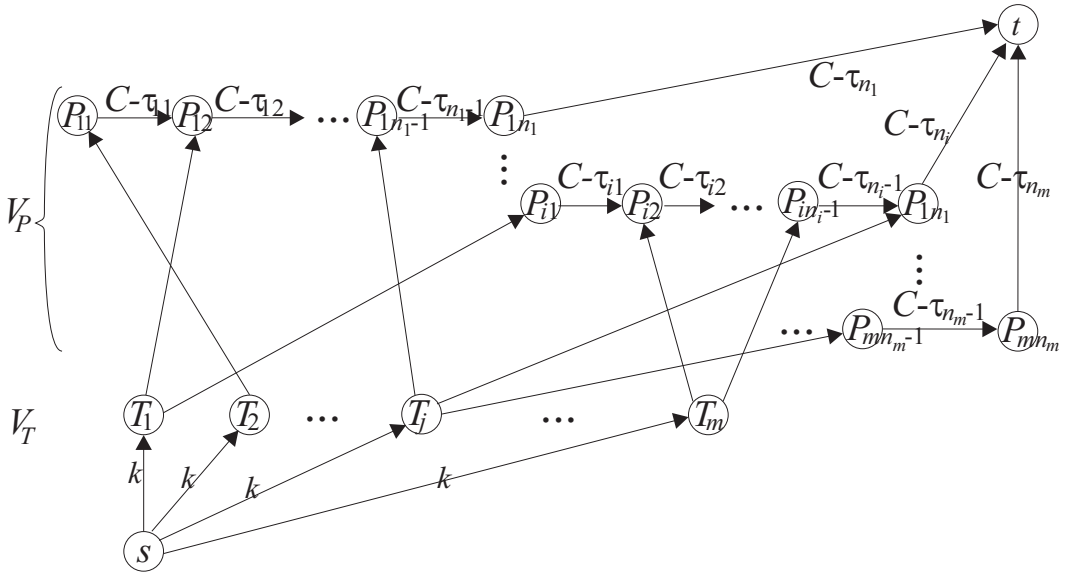

Figure 4.16: Construction of a network for a given pattern.

For a feasible schedule on P_i it is necessary to fulfill $\tau_l + \sum_{h=1}^{l} \phi(v_j, P_{ih}) \leq C$. This inequality is satisfied by the construction of a chain of nodes $P_{i1} \rightarrow P_{i2} \rightarrow \ldots P_{in_i} \rightarrow t$, and the capacities of the subsequent arcs. Note that a flow $\sum_{h=1}^{l} \phi(v_j, P_{ih})$ is passed by arc (P_{il}, P_{il+1}) with capacity $C - \tau_l$ for $l = 1, \ldots, n_i - 1$.

Observation 29. *For a given communication pattern, the optimum schedule length can be found in $O(m^6 \log m)$ time.*

Proof. The above algorithm which verifies feasibility of a schedule with length C, for a given communication pattern, can be used to find the optimum schedule length C^*_{max}. Let Z be the number of communications performed in the communication pattern. Any schedule must be at least as long as Z because each communication lasts a unit time. The optimum schedule length C^*_{max} is the minimum C for which all arcs (s, v_j) are saturated, and consequently the value of the flow is $\phi = nk$. If a schedule with length $C = Z$ does not exist because $\phi < nk$, then its length must be

increased by some value Δ. The value of the flow ϕ is determined by the a capacity of a minimum cut. When C is growing by Δ, then the capacity of the cut may grow by $\mu\Delta$ where μ is a multiplier from set $\{1, \ldots, m\}$. The multiplier μ is determined by the number of arcs on the minimum cut which capacities grow as C is growing. It can be observed in the network (cf. Fig.4.16) that on the cut separating source s from sink t the minimum number of edges with capacity growing with C is one, and the maximum is m. Suppose $\phi < nk$ is the value of the flow obtained for C. The length of the schedule must be increased by $\Delta = (\phi - nk)/\mu$. Though the actual multiplier is not known initially, it can be determined by a binary search over the interval of (discrete) values $\{1, \ldots, m\}$. The number of nodes in the network is $2 + n + \sum_{i=1}^{m} n_i$ which is $O(m^2)$ because no task has to be delivered to a processor twice and hence $n_i \leq n < m$. Thus, for a given communication pattern and value of C the maximum network flow can be constructed in $O(m^6)$. The minimum schedule length for a given communication pattern can be found in $O(m^6 \log m)$ time. □

Observation 30. *An optimum solution can be found in* $O(m^{m^2+6} \log m)$ *time.*

Proof. No task has to be delivered to a processor twice. Hence, the longest communication pattern has at most $nm < m^2$ communications. There are at most m^{m^2} communication patterns, and for each of them the shortest schedule can be constructed in $O(m^6 \log m)$ time. Thus, the complexity of an algorithm based on full enumeration of communication sequences is $O(m^{m^2+6} \log m)$. □

4.5 Conclusions

In this section we studied combinatorial aspects of scheduling multiple divisible loads. It has been demonstrated that this problem is computationally hard for unrelated processors, for uniform processors with simultaneous completion requirement, and even for identical processors. When the order of task execution, the used processors and their activation sequence are given, then the optimum distribution can be found in polynomial time by applying linear programming. The case of a single processor boils down to a well known operations research problem of scheduling in two-machine flowshop. Finally, bounds on the performance of heuristics for the problem have been searched for. The problem of scheduling multiple identical tasks on identical processors turned out to be surprisingly complex. For certain sub-cases solutions can be identified in polynomial time, for other cases only exponential algorithms were proposed.

Chapter 5

Summary

In this work we have examined the DLT problems taking into considerations the three points of view: single load single distribution, single load multiple distributions and multiple loads single distribution. All the three problems are computationally hard in general. However, if the solution of the combinatorial part of the problem is known, i.e. the sequence of communications is given, then these problems can be solved by a reduction to linear programming.

In the single load multiple distributions case two algorithms were proposed: an exact branch&bound algorithm, and a genetic search heuristic. Using the results of the simulations it could be concluded that adjusting sizes of the load chunks to the given communication pattern is an essential element in multi-installment processing. It was also observed that solutions are harder to find when heterogeneity of the computing environment is growing. Especially communication parameter C influence it strong.

In the multiple loads problem special cases solvable in polynomial time were identified, and bounds of the worst-case behavior of the heuristic methods were identified. Finally, a special case of identical tasks and identical processors was studied.

Streszczenie w języku polskim

Teoria szeregowania zadań jednorodnych (ang. Divisible Load Theory (DLT)) zajmuje się nowym modelem systemów rozproszonych. Zakłada się tu, że ziarno obliczeń jest bardzo małe oraz, że nie występują zależności pomiędzy ziarnami obliczeń. W związku z tym obliczenia i zadania mogą zostać podzielone na części o dowolnych rozmiarach. Części te mogą być przetwarzane niezależnie od siebie w sposób równoległy. Rozmiary zadań powinny zostać dopasowane do przepustowości łącz komunikacyjnych oraz szybkości jednostek przetwarzających w ten sposób, aby obliczenia zakończyły się w możliwie najkrótszym czasie. Model zadania jednorodnego dowiódł swojej wszechstronnej przydatności jako narzędzie do modelowania systemów rozproszonych.

Celem tej pracy jest analiza problemu szeregowania zadań jednorodnych z trzech punktów widzenia:

- jednoetapowej dystrybucji pojedynczego zadania,
- wieloetapowej dystrybucji pojedynczego zadania,
- jednoetapowej dystrybucji wielu zadań,

Wszystkie trzy przypadki są obliczeniowo trudne, w ogólności. Jednak gdy znany jest wzorzec komunikacji do wykonania można je rozwiązać przez sprowadzenie do problemu programowania liniowego.

W przypadku jednoetapowej dystrybucji pojedynczego zadania zakładamy, że praca jednego zadania, które ma zostać wykonane jest dystrybuowana jednokrotnie do danego procesora.

Kolejny przypadek dotyczy problemu gdzie mamy do przetworzenia jedno zadanie, które może być przesyłane do danego procesora więcej niż jeden raz. Analizujemy tutaj dwa algorytmy: dokładny branch&bound oraz heurystyczny algorytm genetyczny. Dla obu przypadków zostały wykonane eksperymenty obliczeniowe, porównano czasy wykonania oraz dokładność rozwiązań. W ich wyniku stwierdzono, że wraz ze wzrostem heterogeniczności systemu komputerowego, optymalne rozwiązania są coraz trudniejsze do znalezienia.

Ostatni przypadek dotyczy problemu gdzie mamy wiele zadań, które mogą być dystrybuowane jednokrotnie do danego procesora. Dopuszcza się sytuację gdzie praca jednego zadania może być wysłana kilkukrotnie, ale za każdym razem do innego procesora. Ze względu na to, że problem ten jest obliczeniowo trudny, wskazano przypadki specjalne, które mogą zostać rozwiązane w wielomianowym czasie. Określono również ograniczenie na jakość algorytmów heurystycznych w najgorszym przypadku.

Notation summary

A_i - processing rate (reciprocal of speed) of P_i,

A_{ij} - processing rate (reciprocal of speed) of P_i for load/task j,

α_i - load assigned to P_i for single load,

α_{ij} - load assigned to P_i for load/task j,

B_i - memory size of processor P_i,

B_{ij} - memory size of processor P_i, available for load/task j,

β_j - fraction of input load returned as results for load/task j,

C_i - communication rate (reciprocal of bandwidth) of the link to P_i,

C_{ij} - communication rate (reciprocal of bandwidth) of the link to P_i for load/task j,

$C_{max} = \max\{t_i\}$ - schedule length,

$\overline{C_{max}}$ - an upper limit on schedule length,

D - population size (Chapter 3),

d_i - deadline of P_i, upper limit of P_i availability for computations,

E - set of integers in PARTITION and 3-PARTITION problems,

e_i - value of element i in PARTITION and 3-PARTITION problems,

F - a number defined for PARTITION and 3-PARTITION problems,

f_i - fixed part of the cost of using processor P_i,

$g(j, i)$ - functions addressing task or processors in the sequence of communications,

$G = \sum_{i \in \mathcal{P}'} (f_i + \alpha_i l_i)$ - total cost of the schedule on processors in set $\mathcal{P}'$,

$\overline{G}$ - an upper limit on cost G,

$h(j, i)$ - the number of the i-th processor activated for task T_j,

l_i - coefficient of the linear part of the cost of using P_i,

m - number of processing nodes,

n - number of loads/tasks,

$\mathcal{P}$ - set of available processing nodes,

$\mathcal{P}'$ - set of nodes participating in the computations,

$\mathcal{P}_j$ - set of processors executing task j,

P_i - processing element (processor) i,

p_i - computation startup time on processor P_i,

q - the number of elements in PARTITION and 3-PARTITION problems,

r_i - release time of P_i, lower limit of P_i availability for computations,

S_i - communication startup time of the link to P_i,

t_i - completion of the computations on P_i,

T - upper limit of the schedule length,

T_j - task j,

$T_{j^{dd}}$ - the last task which distributes the load before task T_j distributes its load,

$T_{j^{rd}}$ - the last task which returns its results to the originator before task T_j distributes its load,

$T_{j^{dr}}$ - the last tasks which distributes the load before task T_j returns its results,

$T_{j^{rr}}$ - the last task which returns its results to the originator before returning task T_j results,

t_j^D - the time moment when distribution of task T_j load starts,

t_j^R - the time moment when collection of task T_j results starts,

t_{ij} - the time moment when P_i finishes computing load α_{ij},

τ_l - the completion time of computations on task T_l.

V - single load size,

V_j - size of load/task j,

z - number of installments in multi-installment processing.

Bibliography

[1] S. Bataineh and T.G. Robertazzi. Bus oriented load sharing for a network of sensor driven processors. *special issue on Distributed Sensor Networks of the IEEE Transactions on Systems, Man and Cybernetics*, 21(5):1202–1205, September 1991.

[2] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree networks: Results and open problems. *IEEE Trans. on Parallel and Distributed Systems*, 16:207–218, 2005.

[3] M. Berkelaar. Mixed integer linear program solver. *ftp://ftp.es.ele.tue.nl/pub/lp/_solve*, 1995.

[4] V. Bharadwaj and G. Barlas. Access time minimization for distributed multimedia applications. *Multimedia Tools and Applications*, 12 (2/3):235–256, 2000.

[5] V. Bharadwaj, D. Ghose, and V. Mani. Optimal sequencing and arrangement in distributed single-level tree networks with communication delays. *IEEE Transactions on Parallel and Distributed Systems*, 5, No. 9:968–976, 1994.

[6] V. Bharadwaj, D. Ghose, and V. Mani. Multi-installment load distribution in tree networks with delays. *IEEE Transactions on Aerospace and Electronic Systems*, 31 (2):555–567, 1995.

[7] V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi. Scheduling divisible loads in parallel and distributed systems. *IEEE Computer Society Press*, September 1996. Los Alamitos.

[8] V. Bharadwaj, D. Ghose, and T. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, vol.6(1):7–17, 2003.

[9] J. Błażewicz. Złożoność obliczeniowa problemów kombinatorycznych. *Wydawnictwa Naukowo-Techniczne*, 1988. Warszawa.

[10] J. Błażewicz and M. Drozdowski. Distributed processing of divisible jobs with communication startup costs. *Discrete Applied Mathematics*, 76:21–41, 13 June 1997. Issue 1-3.

[11] J. Błażewicz, M. Drozdowski, and K. Ecker. Management of resources in parallel systems. *J.Błażewicz, K. Ecker, B. Plateau, D. Trystram Handbook on Parallel and Distributed Processing, Springer, Heidelberg*, pages 263–341, 2000.

[12] J. Błażewicz, M. Drozdowski, and M. Markiewicz. Divisible task scheduling - concept and verification. *Parallel Computing*, 25:87–98, 1999.

[13] J. Błażewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Węglarz. Scheduling computer and manufacturing processes. *Springer-Verlag: Heidelberg*, 1996.

[14] S. Charcranoon, T. Robertazzi, and S. Luryi. Load sequencing for a parallel processing utility. *Journal of Parallel and Distributed Computing*, 64:29–35, 2004.

[15] Y.C. Cheng and T.G. Robertazzi. Distributed computation with communication delays. *IEEE Transactions on Aerospace and Electronic Systems*, 24(6):700–712, November 1988.

[16] Y.C. Cheng and T.G. Robertazzi. Distributed computation for a tree network with communication delays. *IEEE Transactions on Aerospace and Electronic Systems*, 26(3):511–516, May 1990.

[17] M. Drozdowski. Selected problems of scheduling tasks in multiprocessor computer systems. *Poznań Univ. of Technology Press, Poznań*, 321, 1997. Series: Monographs, `http://www.cs.put.poznan.pl/mdrozdowski/h.ps`.

[18] M. Drozdowski and M. Lawenda. The combinatorics in divisible load scheduling. *Foundations of Computing and Decision Sciences*, 30(4):297–308, 2005.

[19] M. Drozdowski and M. Lawenda. Multi-installment divisible load processing in heterogeneous systems with limited memory. *PPAM, Pozna, Poland*, Lecture Notes in Computer Science(3911):847–854, September 11-14 2005.

[20] M. Drozdowski and M. Lawenda. On optimum multi-installment divisible load processing. *in.: J.C.Cunha, P.D.Medeiros (eds.), Euro-Par 2005 Parallel Processing*, Lecture Notes in Computer Science(3648):231–240, 2005.

[21] M. Drozdowski, M. Lawenda, and F. Guinand. Scheduling multiple divisible loads. *International Journal of High Perfomance Computing*, 20(1):19–30, 2006.

[22] M. Drozdowski and P. Wolniewicz. Experiments with scheduling divisible tasks in clusters of workstations. *in: A.Bode, T.Ludwig, W.Karl, R.Wismüller (eds.), Euro-Par 2000*, LNCS 1900, Springer-Verlag:311–319, 2000.

[23] M. Drozdowski and P. Wolniewicz. Divisible load scheduling in systems with limited memory. *Cluster Computing*, 6:19–29, 2003.

[24] M. Drozdowski and P. Wolniewicz. Out-of-core divisible load processing. *IEEE Trans. on Parallel and Distributed Systems*, 14(10):1048–1056, 2003.

[25] M. Drozdowski and P. Wolniewicz. Optimum divisible load scheduling on heterogeneous stars with limited memory. *European Journal of Operational Research*, 172:545–559, 2006.

[26] M.R. Garey and D.S. Johnson. Computers and intractability: A guide to the theory of np-completeness. *Freeman*, 1979. San Francisco.

[27] D.E. Goldberg. Genetic algorithms in search, optimization and machine learning. *Addison-Wesley*, 1989.

[28] S.M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1:61–67, 1954.

[29] K. Ko and T.G. Robertazzi. Scheduling in an environment of multiple job submission. *Proceedings of the 2002 Conference on Information Sciences and Systems*, March 2002. Princeton University, Princeton NJ.

[30] P. Li, B. Veeravalli, and A.A. Kassim. Design and implementation of parallel video encoding strategies using divisible load analysis. *IEEE Transactions on Circuits and Systems for Video technology (CSVT)*, 2004.

[31] X. Li, V. Bharadwaj, and C.C. Ko. Optimal divisible task scheduling on single-level tree networks with buffer constraints. *IEEE Trans. on Aerospace and Electronic Systems*, 36(4):1298–1308, 2000.

[32] X. Li, B. Veeravalli, and C.C. Ko. Distributed image processing on a network of workstations. *International Journal of Computers and Applications, ACTA Press*, 25(2):1–10, 2003.

[33] L. Marchal, Y. Yang, H. Casanova, and Y. Robert. A realistic network/application model for scheduling divisible loads on large-scale platforms. *École Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme*, Research Report 2004-21, 2004.

[34] Z. Michalewicz. Genetic algorithms + data structures = evolution programs. *Springer-Verlag*, 1996.

[35] M. Pinedo. Scheduling: theory, algorithms, and systems. *Prentice Hall, Englewood Cliffs*, 1995.

[36] T. Robertazzi. Ten reasons to use divisible load theory. *IEEE Computer*, 36(5):63–68, 2003.

[37] M.H. Rothkopf. Scheduling independent tasks on parallel processors. *Management Science*, 12:437–447, 1966.

[38] TOP500 Supercomputer Sites. *http://www.top500.org/*.

[39] J. Sohn and T. Robertazzi. A muli-job load sharing strategy for divisible jobs on bus networks. *Dept. of Electrical Engineering, SUNY at Stony Brook, Stony Brook, New York*, Technical Report 697, 1994.

[40] J. Sohn, T. Robertazzi, and S. Luryi. Optimizing computing costs using divisible load analysis. *IEEE Transactions on Parallel and Distribute Systems*, 9(3):225–234, 1998.

[41] S. Suresh, V. Mani, S.N. Omkar, and H.J. Kim. Parallel video processing using divisible load scheduling paradigm. *to appear in Korean Journal of Broadcast Engineering*.

[42] B. Veeravalli and G. Barlas. Efficient scheduling strategies for processing multiple divisible loads on bus networks. *Journal of Parallel and Distributed Computing*, 62:132–151, 2002.

[43] P. Wolniewicz. Divisible job scheduling in systems with limited memory. *PhD Thesis, Poznań Univ. of Technology*, 2003. `http://www.man.poznan.pl/~pawelw/phd.pdf`.

[44] P. Wolniewicz and M. Drozdowski. Processing time and memory requirements for multi-instalment divisible job processing. *Parallel Processing and Applied Mathematics: 4th International Conference PPAM 2001*, LNCS 2328, Springer-Verlag:125–133, 2002. in R.Wyrzykowski, J.Dongarra, M.Paprzycki, J.Waniewski (eds.).

[45] L. Xiaolin. Studies on divisible load scheduling strategies in distributed computing systems: Design, analysis and experiments. *PhD thesis, National University*

of Singapore, 2001.