Adam Mickiewicz University
Faculty of Mathematics and Computer Science

Joanna Berlińska

Scheduling divisible loads in heterogeneous distributed systems

Ph.D. Thesis

Supervisor: Prof. Dr. Habil. Maciej Drozdowski

Poznań 2011

Acknowledgements

I wish to express my deep gratitude to my supervisor Professor Maciej Drozdowski for his keen interest, inspiration and perfect guidance throughout the completion of this thesis. He introduced me to the exciting field of divisible load theory and motivated me to conduct original research with high standards. I am sincerely grateful for his effort in training me to become a successful researcher.

The research reported in this thesis has been financially supported by the Polish Ministry of Science and Higher Education grants N N206 372039 "Scheduling divisible loads in heterogeneous distributed systems" and N N519 188933 "New problems of scheduling theory – complexity analysis, algorithmization".

The work presented in this thesis has been also partially supported by the scholarship of the Adam Mickiewicz University Foundation for the year 2010.

Contents

1 Introduction

The progress in many disciplines of science and technology is nowadays strongly supported by computational methods. The research is often based on the results delivered by complex and time-consuming calculations. The computational power of a single computer is often insufficient. Hence, performing the computations in distributed environments like grids or clusters becomes a necessity. What is more, using a distributed computer system has many advantages. Large numbers of processors taking part in computations result in big total computing power. The system is scalable and the time needed for computations can be reduced by employing more processors. On the other hand, controlling computations in a distributed system is more complex. In order to obtain high efficiency, the distributed applications need careful scheduling of communications and computations. As the computers may be spread around the world, the communication delays may be quite big and cannot be neglected. The distributed computer system is usually heterogeneous, and consequently, the different parameters of its elements must be taken into account by the scheduling algorithms.

Divisible load theory (DLT) is a model of parallel computations which offers a realistic approach to this problem. It is mostly used to represent processing large amounts of data in distributed systems. It assumes that the input data, called *load*, can be divided into pieces of arbitrary sizes and these pieces can be processed independently in parallel on remote computers. The divisible load model originated in the late 1980s in publications [1, 20]. It was applied to represent distributed computations in a network of workstations in [1]. In [20] a chain network of intelligent sensors was studied. In both cases, the analyzed problem was how to schedule communications and computations, so that the total time needed to process the load of a given size is as short as possible. On the one hand, using more processors reduces computation time, but on the other hand it needs more communications, which cost time. Hence, the problem is which processors should be used and what load quantities they should receive. The mathematical models proposed in the early publications were computationally tractable and reduced the scheduling problem to a set of linear equations. Later on, more complex models were developed and applied to various network topologies [16, 20, 21, 25], systems with memory limitations [12, 30, 37], computation costs [46] and other. The most general divisible load scheduling problem was proved to be **NP**-hard in [48]. Surveys of divisible load theory can be found, e.g., in [3, 14, 24, 45]. We discuss these results in more detail in the following sections.

There are many examples of divisible load computations, like processing measurement data [20], searching for patterns in text and database files [28], image and video processing [38, 39, 43], solving linear algebra problems [22, 32], DNA sequence alignment [47]. As we showed in [7, 10], processing large amounts of data in MapReduce model [23] on dedicated clusters can also be analyzed on the grounds of divisible load theory. Moreover, the computations on volunteer platforms like BOINC and distributed.net fulfill the assumptions about the divisibility and independence of the load grains. Therefore, the progress in DLT is useful in efficiently managing many real distributed applications.

The main goal of this work is the analysis of several divisible load scheduling problems in heterogeneous distributed systems and the construction of algorithms solving these problems. As the analyzed problems are known to be computationally hard, we will propose approximation algorithms and heuristics. The algorithms will be evaluated and compared by both analytical and experimental methods. The divisible load theory will be also applied to model, analyze and schedule computations in new parallel processing environments, like the MapReduce framework. We will construct a mathematical model of such computations and propose scheduling algorithms. Performance limits of the proposed organization of computations will be investigated.

The structure of this thesis is the following. Chapter 2 is dedicated to single-round divisible load scheduling. In the single-round processing each computer receives at most one message with the data to process. The scheduling problem is which processors should take part in computations, what amounts of data they should receive and in what order. Our main contributions presented in Chapter 2 are fully polynomial time approximation schemes for two scheduling problems. These results have been already published in [6]. Extensions to more general cases are also analyzed.

Chapter 3 covers multi-round divisible load scheduling in systems with limited memory. Multi-round processing means that each processor can receive multiple messages with data to process. It is assumed that the whole load is too big to store it in the memories of the computers at the same moment. Therefore, the load must be distributed and processed in many small pieces fitting available memory buffers. We provide an experimental study of the features of near-optimum solutions, and hence, the nature of the scheduling problem. Based on these results, several groups of heuristics solving the analyzed problems are proposed. Their advantages and weaknesses are demonstrated for a wide range of changing system parameters. The experimental comparison of the proposed algorithms with the heuristics known from earlier literature shows that a big improvement in the quality of the obtained solutions has been achieved. The results contained in Chapter 3 have been published in [8, 9, 11, 12].

Chapter 4 introduces MapReduce paradigm for parallel computations. We show that MapReduce computations can be analyzed as two divisible applications, such that the output of the first of them is the input for the second. We formulate the mathematical model of such computations and propose scheduling algorithms. Then, an experimental analysis of the MapReduce performance is provided. These results have been published in [7, 10]. It was the first time when scheduling divisible loads with precedence constraints was studied.

In Chapter 5 the problem considered in Chapter 4 is generalized. We introduce the notion of a multilayer application. An example of a multilayer application is a chain of MapReduce applications, such that one application in the chain produces input for the next application. The influence of the system parameters on the structure of the schedules is studied.

The last chapter contains a summary of all the presented results. We also propose directions for future research on the aspects of divisible load theory addressed in this work.

2 Single-Round Processing

In this chapter we study divisible load scheduling for single-round organization of computations. Let us start with some general assumptions about the computing environment. In this work we assume that each processor comprises a CPU, some memory and a hardware network interface (e.g. NIC and DMA). The words processor, computer and processing element will be used interchangeably, unless said to be otherwise. The CPU and network interface can work in parallel, so that simultaneous computation and communication is possible. Each computer can communicate with at most one processor at a time (i.e. so-called one-port model is used).

In Chapters 2 and 3 we consider classical divisible load scheduling problems in a star network (see Fig. 2.1). The load to be processed is initially located on processor P_0 called the originator, located in the center of the star. The originator is connected to a set of m processors (workers) $\{P_1, \dots, P_m\}$. The originator divides the load into pieces and sends them directly to the workers. Such a logical topology can represent many parallel systems with different physical topologies, like a grid of multiprocessor supercomputers, a cluster of workstations connected via a local area network, or a set of processors sharing a bus in an SMP system. We assume that the originator only dispatches the load to the other processors and performs no computations. In the opposite case, the computational power of the originator can be represented as an additional processor. For simplicity of the mathematical model, the process of returning results to the originator is not

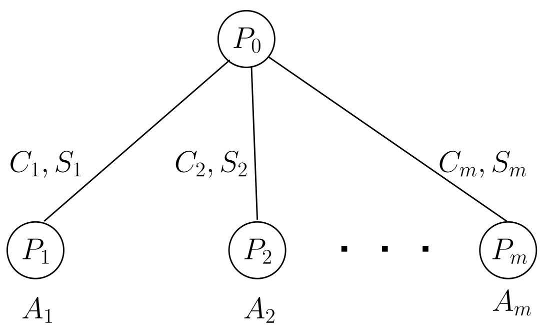


Figure 2.1: Star network topology.

analyzed. Practically, it means that the results returning time is short and can be neglected. It has been shown in [18, 28] that this simplification is not limiting the generality of our considerations, as sending results back can be included in the model.

Each worker P_i is described by its computing rate (inverse of speed, e.g. in seconds per byte), denoted by A_i. Processing load of size α on P_i takes time αA_i. The communication link between P_i and the originator is described by startup time S_i (e.g. in seconds) and communication rate (inverse of bandwidth) C_i. Hence, the time required to send load of size α to processor P_i is $S_i + \alpha C_i$. We will use the notation $A_{max} = \max_{1 \le i \le m} A_i$, $A_{min} = \min_{1 \le i \le m} A_i$, and similarly for the other parameters. In the general case, all parameters A_i, C_i, S_i are nonnegative rational numbers.

Below we formulate several single-round divisible load scheduling problems. We follow the notation used in [48], where different divisible load scheduling problems are denoted by DLS$\{restriction\}$. The *restriction* is the list of additional assumptions in the analyzed problem. These restrictions may be, for example:

- *1Round* for single-round scheduling problems,
- $C_i = 0$ if all the bandwidths are infinite ($C_i = 0$ for all $1 \le i \le m$),
- $S_i = 0$ if there are no startup times ($S_i = 0$ for all $1 \le i \le m$).

The decision version of the general single-round divisible load scheduling problem can be formulated as follows.

Problem 2.1. (DLS{1Round})
Given m workers, their parameters A_i, C_i and S_i for $1 \leq i \leq m$, and two rational numbers $V > 0$ and $T > 0$, is it possible to process load of size V within time T from the moment when the originator starts sending out the load?

We also define the following two optimization problems connected with problem DLS{1Round}.

Problem 2.2. (DLS{1Round}-Opt$V$)
Given a rational time $T > 0$, m workers, their parameters A_i, C_i and S_i for $1 \leq i \leq m$, find the greatest rational number $V_{OPT}(T)$, such that it is possible to process load of size $V_{OPT}(T)$ within time T.

Problem 2.3. (DLS{1Round}-Opt$T$)
Given a rational load size $V > 0$, m workers, their parameters A_i, C_i and S_i for $1 \leq i \leq m$, find the smallest rational number $T_{OPT}(V) \geq 0$, such that it is possible to process the whole load V within time $T_{OPT}(V)$.

Let us note that we are interested not only in finding the optimum time T or the amount of load V, but also in constructing the optimum schedule. Constructing a schedule involves making the following decisions:

- The set $P' \subseteq P$ of processors participating in the computations must be chosen. Depending on the parameters of the processors and communication links, it may be unprofitable to use some of them for computations.
- The communication sequence (also called activation sequence), defining the order in which the processors receive load, must be chosen. For single-round processing, the communication sequence is a permutation of indices of processors from the set P'.
- The sizes of the load parts sent in each message must be selected.

2.1 Earlier Results

The early publications concerning scheduling divisible loads in a star system used a simple linear communication model. All communication startup times S_i were assumed to be equal to zero. The analyzed problems were DLS$\{$1Round, $S_i = 0\}$ and the adequate optimization problems. It was proved independently in [5, 13, 17, 35] that if all workers take part in the computations and finish work at the same moment, then the problem DLS$\{$1Round, $S_i = 0\}$ can be solved by sorting the processors by nondecreasing C_i in the activation sequence. The hypothesis that in the optimum solution all workers participate in computations and finish work simultaneously was proved in [3].

The assumption about linear communication costs usually does not hold in practice. It has a side effect that *all* processors can take part in the computations, no matter how many of them are available, and no matter how far from the originator they are. Hence, a more realistic affine communication model, including startup times, was introduced by Błażewicz and Drozdowski in [17]. In publication [3] it was shown that in the optimum solutions of both optimization versions of the problem DLS$\{$1Round$\}$ all processors taking part in computations finish work at the same moment. Additionally, the authors proved that if the load size V is large enough, then in any optimum solution all workers participate in the computations and they should be activated in the order of nondecreasing C_i.

The complexity of single-round divisible load scheduling problem remained open until 2007. Finally, in [48] it was proved that the problem DLS$\{$1Round, $C_i = 0\}$ is **NP**-complete. The proof was done by reduction from the **NP**-complete 2-Partition problem. The authors proposed pseudo-polynomial dynamic programming algorithms solving the problems DLS$\{$1Round, $C_i = 0\}$-Opt$V$ and DLS$\{$1Round, $C_i = 0\}$-Opt$T$. However, since pseudopolynomial algorithms are in fact exponential, it can be more useful to create polynomial approximation algorithms for these problems. The strongest polynomial time approximation result that can be derived for **NP**-hard problems (unless **P**=**NP**) is a fully polynomial time approximation scheme (FPTAS). An FPTAS for an optimization problem Π with cost function f is an approximation algorithm $\mathcal{A}$ which for any given $\varepsilon > 0$ and an instance I of problem Π

- returns a solution $\mathcal{A}(I)$ such that $|f(\mathcal{A}(I)) - OPT(I)| \leq \varepsilon |OPT(I)|$, where $OPT(I)$ is the optimum cost for instance I, and
- has running time polynomial in the size of I and $1/\varepsilon$.

Constructing fully polynomial time approximation schemes for DLS{1Round, $C_i = 0$}-Opt$V$ and DLS{1Round, $C_i = 0$}-Opt$T$ is the aim of the next two sections.

2.2 FPTAS for Problem DLS{$C_i = 0$}-Opt$V$

Let us start with an observation that if $C_i = 0$ for $1 \leq i \leq m$, then nothing can be gained by sending more then one message to the same processor. Hence, for the divisible load scheduling problem with $C_i = 0$ for all i, there always exists an optimum solution using one round only. Consequently, we can write DLS{$C_i = 0$} instead of DLS{1Round, $C_i = 0$}, because these two problems are equivalent.

We begin our considerations with the problem of optimizing the size of the load processed in a given time T. Similarly as in [48], we assume here that A_i and S_i are integer numbers. The problem can be formulated as follows.

Problem 2.4. (DLS{$C_i = 0$}-Opt$V$)
Given a rational time $T > 0$, m workers, their integer parameters A_i and S_i for $1 \leq i \leq m$, and provided that the bandwidths are infinite, find the greatest rational number $V_{OPT}(T)$, such that it is possible to process load of size $V_{OPT}(T)$ within time T.

Let us note that if $S_i > T$ for some processor P_i, then this processor cannot be used for processing load in time T. Therefore, we assume that $S_i \leq T$ for $1 \leq i \leq m$. Moreover, if $A_i = 0$ for some processor P_i, then P_i can receive and process an infinite amount of load in time S_i. As $S_i \leq T$, the scheduling problem becomes trivial in this case. Hence, we assume that $A_i > 0$ for $1 \leq i \leq m$.

In order to construct an FPTAS solving Problem 2.4, we need to know in what order the processors should be activated. We will use the following proposition given in [48].

Proposition 2.1. *For a given time limit T and a set $P' \subseteq \{P_1, \dots, P_m\}$ of workers taking part in the computations, the maximum load is processed if the workers are ordered according to nondecreasing values of S_iA_i for $P_i \in P'$.*

Proposition 2.1 can be proved by the interchange argument: ordering the processors in P' according to nondecreasing S_iA_i does not reduce the amount of load processed in time T.

As it is known from [3] that in the optimum solution all processors taking part in computations finish work at the same moment, it follows from Proposition 2.1 that the scheduling problem can be reduced to choosing an optimum subset of processors taking part in the computations. Let us assume, without loss of generality, that $S_1A_1 \leq \ldots \leq S_mA_m$. We define a binary vector $\mathbf{x} = (x_1, \ldots, x_m)$ as follows: $x_i = 1$ if processor P_i receives some load to process (i.e. $P_i \in P'$) and $x_i = 0$ in the opposite case ($P_i \notin P'$). The maximum amount of load which can be processed in time T using the subset of processors indicated by $\mathbf{x}$ can be obtained from the formula

$$V_{OPT}(T, \mathbf{x}) = \sum_{i=1}^{m} \frac{Tx_i}{A_i} - \sum_{i=1}^{m} \sum_{j=i}^{m} \frac{x_i x_j S_i}{A_j}. \tag{2.1}$$

The expression $\sum_{i=1}^{m} \frac{Tx_i}{A_i}$ is the amount of load which could be processed in time T by processors indicated by $\mathbf{x}$ if there were no communication delays. Communication with processor P_i takes time x_iS_i. During this time processors P_j, where $j \geq i$, cannot process any load because they did not receive the input yet. Thus, $\sum_{i=1}^{m}\sum_{j=i}^{m}\frac{x_ix_jS_i}{A_j}$ is the amount of load which is lost because of communication delays (cf. [48]).

Our goal is to maximize the size V of load processed in a given time T as a function of a binary vector $\mathbf{x} = (x_1, \dots, x_m)$. Instead of maximizing $V(\mathbf{x})$, we will minimize the value of $-V(\mathbf{x})$. Since x_i are binary variables, we have $x_i^2 = x_i$. Hence we have

$$-V(\mathbf{x}) = -\sum_{i=1}^{m}\frac{T-S_i}{A_i}x_i + \sum_{1\leq i<j\leq m} S_i\frac{1}{A_j}x_ix_j. \tag{2.2}$$

A half-product [2] is a function $f : \{0,1\}^m \to \mathbb{R}$ of the form

$$f(\mathbf{x}) = f(x_1, \dots, x_m) = -\sum_{i=1}^{m} p_ix_i + \sum_{1\leq i<j\leq m} q_ir_jx_ix_j, \tag{2.3}$$

where p_i, q_i, r_i are nonnegative constants for $1 \leq i \leq m$. Thus, $-V(\mathbf{x})$ is a half-product, with $p_i = \frac{T-S_i}{A_i}$, $q_i = S_i$, $r_j = \frac{1}{A_j}$.

An FPTAS for minimizing half-products was proposed by Badics and Boros in [2]. They assumed that the parameters p_i, q_i, r_i are nonnegative integers for $1 \leq i \leq m$. In our case all parameters are nonnegative, but $p_i = \frac{T-S_i}{A_i}$ and $r_j = \frac{1}{A_j}$ are not integer. However, the assumption about integrality of p_i and r_i is used neither for proving the correctness of the Badics and Boros algorithm, nor for estimating its running time. Therefore, we can use the algorithm proposed in [2] to minimize the function $-V(\mathbf{x})$. The algorithm receives number m, vectors $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$ of length m, and a positive approximation precision $\varepsilon < 1$. It returns a binary vector $\mathbf{x}^\varepsilon = (x_1^\varepsilon, \dots, x_m^\varepsilon)$.

For $1 \leq k \leq m$, let $g_k(\mathbf{x}) = -\sum_{i=1}^{k} p_ix_i + \sum_{1\leq i<j\leq k} q_ir_jx_ix_j$ and $Q_k(\mathbf{x}) = \sum_{i=1}^{k} q_ix_i$. The FPTAS for minimizing half-products proposed by Badics and Boros is formulated in Algorithm 2.1 (cf. [2]).

Algorithm 2.1 MINIMIZE-HALF-PRODUCT(m, $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$, ε)

```
STEP 0:
Let $\delta > 0$ be defined by the equation $(1+\delta)^m = 1+\varepsilon$,
let $Q = \sum_{i=1}^{m} q_i$, $N = \lceil \frac{2m \log Q}{\varepsilon} \rceil$, $k = 0$ and $\mathcal{X}_0 = \{()\}$.
STEP 1:
Let $k = k+1$, $\mathcal{X}_k = \emptyset$, $t = 0$, $s = 0$,
$\mathcal{L} = \{(y_1, \dots, y_{k-1}, 0), (y_1, \dots, y_{k-1}, 1) | (y_1, \dots, y_{k-1}) \in \mathcal{X}_{k-1}\}$
STEP 2:
while $s \leq N$ do
  select $\mathbf{z} = (z_1, \dots, z_k) \in \mathcal{L}$ for which $t \leq Q_k(\mathbf{z}) < (1+\delta)^s$
  and for which $g_k(\mathbf{z})$ is the smallest among all such $\mathbf{z}$.
  Let $\mathcal{X}_k = \mathcal{X}_k \cup \{\mathbf{z}\}$, $t = (1+\delta)^s$, $s = s+1$.
end while
STEP 3:
if $k < m$ then
  goto STEP 1
else
  goto STEP 4.
end if
STEP 4:
Select $\mathbf{x}^\varepsilon \in \mathcal{X}_m$ with the smallest $g_m(\mathbf{x}^\varepsilon)$, return $\mathbf{x}^\varepsilon$.
```

It was proved in [2] that

$$f(\mathbf{x}^\varepsilon) \leq f(\mathbf{x}^*) + \varepsilon |f(\mathbf{x}^*)|, \tag{2.4}$$

where $\mathbf{x}^*$ is a vector minimizing f, and the running time of the algorithm MINIMIZE-HALF-PRODUCT is $O(m^2 \log(\sum_{i=1}^{m} q_i)/\varepsilon)$ [2].

Based on these results, we propose Algorithm 2.2 for Problem 2.4 [6].

Theorem 2.2. *Algorithm 2.2 is a fully polynomial time approximation scheme for Problem 2.4 (DLS$\{C_i = 0\}$-OptV).*

Algorithm 2.2 FPTAS-OPT-V$(T, m, \mathbf{A}, \mathbf{S}, \varepsilon)$

```
for $i = 1$ to $m$ do
    $p_i = \frac{T-S_i}{A_i}$
    $q_i = S_i$
    $r_i = \frac{1}{A_i}$
end for
$\mathbf{x}^\varepsilon$=MINIMIZE-HALF-PRODUCT($m$, $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$, $\varepsilon$)
return $\mathbf{x_{FPTAS}}(T,\varepsilon) = \mathbf{x}^\varepsilon$, $V_{FPTAS}(T,\varepsilon) = \sum_{i=1}^m \frac{Tx_i^\varepsilon}{A_i} - \sum_{i=1}^m \sum_{j=i}^m \frac{x_i^\varepsilon x_j^\varepsilon S_i}{A_j}$
```

Proof. Since $\mathbf{x_{FPTAS}}(T,\varepsilon)$ is returned by the MINIMIZE-HALF-PRODUCT algorithm for the function $-V(\mathbf{x})$, we get from (2.4)

$$-V_{FPTAS}(T,\varepsilon) \le -V_{OPT}(T) + \varepsilon|-V_{OPT}(T)|. \tag{2.5}$$

As the amount of load $V_{OPT}(T)$ is always nonnegative, this formula can be rewritten as

$$-V_{FPTAS}(T,\varepsilon) \le -V_{OPT}(T) + \varepsilon V_{OPT}(T). \tag{2.6}$$

Hence,

$$V_{FPTAS}(T,\varepsilon) \ge V_{OPT}(T)(1-\varepsilon). \tag{2.7}$$

Moreover, the running time of Algorithm 2.2 is dominated by the running time of MINIMIZE-HALF-PRODUCT, and is equal to at most $O(m^2 \log(\sum_{i=1}^m S_i)/\varepsilon)$, which is bounded from above by $O(m^2(\log m + \log S_{max})/\varepsilon)$. Hence, Algorithm 2.2 is an FPTAS for Problem 2.4. □

2.3 FPTAS for Problem DLS$\{C_i = 0\}$-Opt$T$

The second optimization problem we will analyze is DLS$\{C_i = 0\}$-Opt$T$, which can be formulated in the following way.

Problem 2.5. (DLS{1Round}-Opt$T$)

Given a rational load size $V > 0$, m workers, their integer parameters A_i and S_i for $1 \leq i \leq m$, and provided that the bandwidths are infinite, find the smallest rational number $T_{OPT}(V) \geq 0$, such that it is possible to process the whole load V within time $T_{OPT}(V)$.

To create an approximation scheme for Problem 2.5, we will use the dual approximation algorithm approach proposed in [34]. As stated in [34], a dual approximation algorithm is an algorithm which finds a superoptimal infeasible solution of a given optimization problem. The performance of the algorithm is measured by the degree of the infeasibility of the solution, controlled by a given value $\varepsilon > 0$. We will construct a dual approximation algorithm for Problem 2.4 (DLS{$C_i = 0$}-Opt$V$). This algorithm should accept a period of time T and accuracy ε ($0 < \varepsilon < 1$), and deliver a schedule processing the load of size at least $V_{OPT}(T)$ in time not longer than $T(1+\varepsilon)$. We propose the following Algorithm 2.3 [6].

Algorithm 2.3 DUAL-OPT-V($T, m, \mathbf{A}, \mathbf{S}, \varepsilon$)

```
  **call** FPTAS-OPT-V($T, m, \mathbf{A}, \mathbf{S}, \varepsilon/2$)
  **return** $\mathbf{x_{DUAL}}(T,\varepsilon) = \mathbf{x_{FPTAS}}(T,\varepsilon/2)$, $V_{DUAL}(T,\varepsilon) = (1+\varepsilon)V_{FPTAS}(T,\varepsilon/2)$
```

In order to prove that Algorithm 2.3 is a dual approximation algorithm for Problem 2.4, we will use the following fact.

Proposition 2.3. *If it is possible to process load of size V in time T using the subset of processors indicated by a binary vector $\mathbf{x} = (x_1, \ldots, x_m)$, then it is also possible to process load of size $V(1+\varepsilon)$ in time at most $T(1+\varepsilon)$, using the same subset of processors.*

Proof. Let V' denote the maximum size of load which can be processed in time $T(1+\varepsilon)$ using the processors indicated by the vector $\mathbf{x}$. From (2.1) we obtain

$$V' = \sum_{i=1}^{m} \frac{T(1+\varepsilon)x_i}{A_i} - \sum_{i=1}^{m}\sum_{j=i}^{m} \frac{x_i x_j S_i}{A_j} \tag{2.8}$$

and

$$V = \sum_{i=1}^{m} \frac{T x_i}{A_i} - \sum_{i=1}^{m}\sum_{j=i}^{m} \frac{x_i x_j S_i}{A_j}. \tag{2.9}$$

Hence,

$$V' = (1+\varepsilon)V + \varepsilon \sum_{i=1}^{m}\sum_{j=i}^{m} \frac{x_i x_j S_i}{A_j} \geq V(1+\varepsilon). \tag{2.10}$$

□

Note that if $T = T_{OPT}(V)$, then by Proposition 2.3 load of size $V(1+\varepsilon)$ can be processed in time not longer than $T_{OPT}(V)(1+\varepsilon)$. Hence, as a corollary, we can formulate the following proposition.

Proposition 2.4. *For any numbers $V \geq 0$ and $\varepsilon > 0$ we have*

$$T_{OPT}(V(1+\varepsilon)) \leq T_{OPT}(V)(1+\varepsilon). \tag{2.11}$$

We will say that an algorithm is a fully polynomial time dual approximation algorithm for a given problem if it is a dual approximation algorithm for this problem with approximation precision ε and its running time is polynomial in both the problem size and $1/\varepsilon$.

Theorem 2.5. *Algorithm 2.3 is a fully polynomial time dual approximation algorithm for Problem 2.4 (DLS$\{C_i = 0\}$-OptV).*

Proof. As $V_{DUAL}(T,\varepsilon) = (1+\varepsilon)V_{FPTAS}(T,\varepsilon/2)$ in Algorithm 2.3, we obtain from (2.7) that

$$V_{DUAL}(T,\varepsilon) \geq (1+\varepsilon)V_{OPT}(T)(1-\varepsilon/2) \geq V_{OPT}(T), \tag{2.12}$$

because $\varepsilon < 1$. Thus, the obtained solution is superoptimal. The time needed to process the load of size $V_{DUAL}(T, \varepsilon)$ is at most $T(1+\varepsilon)$ by Proposition 2.3, as it is possible to process load of size $V_{FPTAS}(T, \varepsilon/2)$ in time T.

The running time of Algorithm 2.3 is determined by the call to algorithm FPTAS-OPT-V, whence it is equal to at most $O(m^2(\log m + \log S_{max})/\varepsilon)$. $\square$

The dual approximation algorithm 2.3 is the key element of the FPTAS solving Problem 2.5 (DLS$\{C_i = 0\}$-Opt$T$), given in Algorithm 2.4.

Algorithm 2.4 FPTAS-OPT-T($V, m, \mathbf{A}, \mathbf{S}, \varepsilon$)

```
$upper = S_{max} + VA_{max}$
$lower = 0$
$LoBo = VA_{min}/m$
**while** $(upper - lower) > \frac{\varepsilon(1-\varepsilon)}{(2-\varepsilon)} LoBo$ **do**
  $T_p = (upper + lower)/2$
  **call** DUAL-OPT-V($T_p, m, \mathbf{A}, \mathbf{S}, \varepsilon$)
  **if** $V_{DUAL}(T_p, \varepsilon) < V(1+\varepsilon)$ **then**
    $lower = T_p$
  **else**
    $upper = T_p$
  **end if**
**end while**
**call** FPTAS-OPT-V($upper, m, \mathbf{A}, \mathbf{S}, \varepsilon/2$)
**return** $\mathbf{x} = \mathbf{x_{FPTAS}}(upper, \varepsilon/2)$, $T = upper$
```

The idea of Algorithm 2.4 is to find a good approximation of $T_{OPT}(V)$ with a binary search. The initial search interval $[lower, upper]$ is defined by trivial lower and upper bounds for $T_{OPT}(V)$. Then, it is iteratively narrowed to its lower or upper half, depending on the results delivered by Algorithm 2.3 for the currently examined value T_p. When the search interval becomes short enough, the searching procedure is finished and the vector $\mathbf{x}$ representing the subset of processors which should be used for computations is obtained by Algorithm 2.2.

Below we prove that Algorithm 2.4 is an FPTAS solving Problem 2.5.

Theorem 2.6. *Algorithm 2.4 is a fully polynomial time approximation scheme for Problem 2.5 (DLS$\{C_i = 0\}$-OptT).*

Proof. Let us start with the observation that at the beginning of the algorithm *upper* and *lower* are trivial upper and lower bounds for $T_{OPT}(V)$. *LoBo* is also a lower bound on $T_{OPT}(V)$ and it is positive, since we assumed that $A_i > 0$ for $1 \le i \le m$.

First, we will analyze the variable *upper* in order to prove that the algorithm always returns a feasible solution. At the beginning of the algorithm we have $upper = S_{max} + VA_{max}$. If this value is not changed in the binary search **while** loop, then the algorithm FPTAS-OPT-V is called for parameters $T = upper = S_{max} + VA_{max}$ and approximation precision $\varepsilon/2$ at the end of executing Algorithm 2.4. The obtained schedule allows for processing the load of size at least V, as it is enough to choose any nonempty subset of the set $\{P_1, \dots, P_m\}$ to process V units of load in time $T = S_{max} + VA_{max}$.

Now let us assume that the value of *upper* is changed at least once to T_p. This happens only if $V_{DUAL}(T_p, \varepsilon) \ge V(1+\varepsilon)$. Therefore, as we have in Algorithm 2.3

$$V_{DUAL}(T, \varepsilon) = (1+\varepsilon)V_{FPTAS}(T, \varepsilon/2), \tag{2.13}$$

there holds

$$V_{FPTAS}(upper, \varepsilon/2) = V_{DUAL}(upper, \varepsilon)/(1+\varepsilon) \ge V \tag{2.14}$$

at any time during the execution of Algorithm 2.4. Hence, the solution obtained by the algorithm FPTAS-OPT-T is always feasible.

Now let us estimate the quality of the obtained solution. We will show that

$$lower < T_{OPT}(V)(1 + \frac{\varepsilon}{2-\varepsilon}) \tag{2.15}$$

throughout the execution of the program. Since initially $lower = 0$, this condition is true before entering into the **while** loop. The value of variable $lower$ is changed to T_p only when $V_{DUAL}(T_p, \varepsilon) < V(1+\varepsilon)$. It follows from (2.13) that

$$(1+\varepsilon)V_{FPTAS}(lower, \varepsilon/2) < V(1+\varepsilon). \tag{2.16}$$

Furthermore, from (2.7) we get

$$(1+\varepsilon)V_{OPT}(lower)(1-\varepsilon/2) < V(1+\varepsilon), \tag{2.17}$$

$$V_{OPT}(lower) < V/(1-\varepsilon/2) \tag{2.18}$$

and finally

$$V_{OPT}(lower) < V(1+\frac{\varepsilon}{2-\varepsilon}). \tag{2.19}$$

Thus, it is impossible to process load $V(1+\frac{\varepsilon}{2-\varepsilon})$ in time $lower$. Hence,

$$lower < T_{OPT}(V(1+\frac{\varepsilon}{2-\varepsilon})). \tag{2.20}$$

By Proposition 2.4 we have

$$T_{OPT}(V(1+\frac{\varepsilon}{2-\varepsilon})) \leq T_{OPT}(V)(1+\frac{\varepsilon}{2-\varepsilon}), \tag{2.21}$$

what proves that (2.15) is true during the binary search.

The binary search is finished when $upper \leq lower + \frac{\varepsilon(1-\varepsilon)}{(2-\varepsilon)} LoBo$. Since $LoBo \leq T_{OPT}(V)$, by (2.15) we get

$$upper \leq T_{OPT}(V)(1+\frac{\varepsilon}{2-\varepsilon}) + \frac{\varepsilon(1-\varepsilon)}{(2-\varepsilon)} T_{OPT}(V) \tag{2.22}$$

and consequently

$$upper \leq T_{OPT}(V)(1+\varepsilon). \tag{2.23}$$

Thus, Algorithm 2.4 delivers the desired approximation of the optimum solution of the problem.

The number of iterations in the binary search is at most equal to $O(\log((S_{max}+VA_{max})/(\frac{\varepsilon(1-\varepsilon)}{(2-\varepsilon)}VA_{min}/m)))$, which is bounded from above by $O(\log m+\log S_{max} + \log A_{max}+\log(1/\varepsilon)+\max(\log V,\log(1/V)))$. The execution time of each iteration is $O(m^2(\log m+\log S_{max})/\varepsilon)$ due to calling Algorithm 2.3. Thus, the running time of the whole algorithm FPTAS-OPT-T is at most $O((\log m+\log S_{max}+\log A_{max}+\log(1/\varepsilon)+\max(\log V,\log(1/V)))m^2(\log m+\log S_{max})/\varepsilon)$. □

2.4 Communication Sequence for Problem DLS{1Round}

It would be desirable to extend the approximability results presented in the preceding sections to problems DLS{1Round}-Opt$V$ and DLS{1Round}-Opt$T$. Note that DLS{1Round,$C_i=0$} is a selection problem. This means that it is computationally hard to select the set P' of participating processors, but for a given P' the optimum activation sequence is known. Moreover, this feature allowed for construction of an FPTAS selecting the set P' of participating processors. The main difficulty in problem DLS{1Round} is that for instances with $C_i>0$, the optimum order of activating the processors is not known. Therefore, the scheduling problems cannot be reduced to just choosing the processors which should take part in computations. Let us remind that a general method of ordering processors should cover special cases:

- ordering processors according to nondecreasing values S_iA_i if all C_i are equal to zero,
- ordering processors according to nondecreasing values C_i if all S_i are equal to zero,
- ordering processors according to nondecreasing values C_i if the load V to

be processed or the time T used for processing is large enough.

Let us analyze the activation sequence for problem DLS{1Round}-Opt$V$ instance with $m = 3$. We will compare the amounts of load which can be processed for activation sequences $\sigma' = (1, 2, 3)$ and $\sigma'' = (2, 1, 3)$. In both cases we assume that all processors finish computations at time T, as this is true in the optimum schedule. It is also assumed that the time T is so large that all processors P_1, P_2, P_3 should take part in the computations in the optimum schedule.

Let α'_i, α''_i denote the sizes of the i-th piece of load sent for activation sequences σ' and σ'', correspondingly. The sizes of the first two parts of load, sent to processors P_1 and P_2 for communication sequence σ', are equal to

$$\alpha'_1 = \frac{T - S_1}{C_1 + A_1} \tag{2.24}$$

and

$$\alpha'_2 = \frac{T - S_1 - C_1\alpha_1 - S_2}{C_2 + A_2}, \tag{2.25}$$

which gives

$$\alpha'_2 = \frac{A_1(T - S_1)}{(C_1 + A_1)(C_2 + A_2)} - \frac{S_2}{C_2 + A_2}. \tag{2.26}$$

Similarly, for communication sequence σ'', the sizes of the first two pieces of load, sent to processors P_2 and P_1 correspondingly, are equal to

$$\alpha''_1 = \frac{T - S_2}{C_2 + A_2} \tag{2.27}$$

and

$$\alpha''_2 = \frac{A_2(T - S_2)}{(C_1 + A_1)(C_2 + A_2)} - \frac{S_1}{C_1 + A_1}. \tag{2.28}$$

Let us observe that the time needed for sending the first two pieces of load may be different for activation sequences σ' and σ''. Therefore, the amount of load processed by computer P_3 may also be different in these two cases. The first two chunks of load are sent in time

$$t' = S_1 + S_2 + C_1 \frac{T - S_1}{C_1 + A_1} + C_2 \left(\frac{A_1(T - S_1)}{(C_1 + A_1)(C_2 + A_2)} - \frac{S_2}{C_2 + A_2} \right) \tag{2.29}$$

if activation sequence is σ', and in time

$$t'' = S_1 + S_2 + C_2 \frac{T - S_2}{C_2 + A_2} + C_1 \left(\frac{A_2(T - S_2)}{(C_1 + A_1)(C_2 + A_2)} - \frac{S_1}{C_1 + A_1} \right) \tag{2.30}$$

if activation sequence is σ''. From (2.29) and (2.30) we obtain

$$\Delta t = t' - t'' = \frac{C_1 A_2 S_2 - C_2 A_1 S_1}{(C_1 + A_1)(C_2 + A_2)}. \tag{2.31}$$

Let t'_3 and t''_3 be the amounts of time used for communication and computations of processor P_3 for sequences σ' and σ''. Note that

$$t''_3 - t'_3 = \Delta t. \tag{2.32}$$

Therefore,

$$\alpha''_3 - \alpha'_3 = \frac{\Delta t}{C_3 + A_3}. \tag{2.33}$$

From equations (2.24)-(2.28) and (2.33), we can compute the difference between the amounts of load processed in both schedules:

$$\begin{aligned} \Delta V = \sum_{i=1}^{3} \alpha''_i - \sum_{i=1}^{3} \alpha'_i &= \frac{T(C_1 - C_2) + A_1 S_1 - A_2 S_2}{(C_1 + A_1)(C_2 + A_2)} \\ &\quad + \frac{C_1 A_2 S_2 - C_2 A_1 S_1}{(C_1 + A_1)(C_2 + A_2)(C_3 + A_3)}. \end{aligned} \tag{2.34}$$

It can be seen that the sign of ΔV depends not only on the parameters of processors P_1 and P_2, but also on A_3 and C_3. Similarly, for $m > 3$ the order in which the first two processors should be activated depends on the parameters of all the remaining processors. Hence, it can be very difficult to decide in what order to activate the processors, because the decision how to sequence, e.g., P_1, P_2 cannot be confined to just P_1, P_2. The first summand in formula (2.34) may suggest sorting the processors according to nondecreasing values of $TC_i + A_iS_i$. Such an algorithm would handle properly the special cases mentioned at the beginning of this section.

However, consider the following counterexample. Let $T = 700$, $m = 4$, and let the parameters of the processors be as given in Table 2.1.

Table 2.1: Processor parameters for the counterexample.

i	A_i	C_i	S_i	$TC_i + A_iS_i$ for $T = 700$
1	0.051	0.129	137.084	97.291284
2	2.146	0.050	34.487	109.009102
3	0.654	0.458	31.565	341.243510
4	1.838	0.152	32.747	166.588986

The amounts of load which can be processed for all activation sequences are given in Table 2.2. If the processors are sorted according to nondecreasing values of $TC_i + A_iS_i$, we obtain communication sequence (1,2,4,3) and the size of processed load is about 3275.0461. On the other hand, the optimum communication sequence is (2,1,4,3), which allows for processing the load of size approximately 3276.4212. Thus, the analyzed algorithm does not deliver the optimum communication sequence.

Another approach to selecting the best communication sequence is to start from the initial sequence $(1, 2, \dots, m)$, and improve it by changing the positions of some processors. Let us assume that it is allowed to perform two operations on the communication sequence: swap a pair of processors or move a single processor to another place in the sequence. Only the moves increasing processed load V for the given schedule length T can be made. However, for the instance given above, the amount of load processed for communication sequence $\sigma_1 = (1, 2, 3, 4)$ is approximately 3276.0243 (see Table 2.2). The only communication sequence

Table 2.2: The size V of load processed for different activation sequences in the counterexample (rounded to 4 digits after decimal point).

Sequence	Processed load V	Sequence	Processed load V
(1,2,3,4)	**3276.0243**	**(1,2,4,3)**	**3275.0461**
(1,3,2,4)	3264.4671	(1,3,4,2)	3265.8734
(1,4,2,3)	3272.7902	(1,4,3,2)	3275.0848
(2,1,3,4)	**3274.1818**	**(2,1,4,3)**	**3276.4212**
(2,3,1,4)	2135.6348	(2,3,4,1)	1963.7528
(2,4,1,3)	3102.9726	(2,4,3,1)	2097.1445
(3,1,2,4)	2040.9951	(3,1,4,2)	2044.6016
(3,2,1,4)	1963.8495	(3,2,4,1)	1792.8317
(3,4,1,2)	1879.3648	(3,4,2,1)	1776.3430
(4,1,2,3)	3104.2910	(4,1,3,2)	3103.4595
(4,2,1,3)	3078.8408	(4,2,3,1)	2076.1120
(4,3,1,2)	2021.0451	(4,3,2,1)	1920.2297

for which it is possible to process larger load, is the optimum sequence $\sigma_2 = (2,1,4,3)$. Yet, it is impossible to obtain this solution by the moves described above, as any allowed change to σ_1 results in decreasing the amount of processed load, and hence cannot be accepted.

The above counterexample proves not only that the described type of greedy algorithms is not capable of solving our problem, but also that it is impossible to find the optimum activation sequence by simply sorting the processors according to some combination of instance parameters. Indeed, note that the communication sequence (1,2,3,4) is better than (1,2,4,3) and the sequence (2,1,4,3) is better than (2,1,3,4). This shows that depending on the amount of time left for processing on P_3 and P_4, it is better to activate one or the other processor earlier. Thus, the order in which processors P_3 and P_4 should be activated depends on the parameters of processors activated before them. Consequently, it is not possible to determine the communication sequence locally, without taking into account the sequence of other processors.

Moreover, for the above instance, the load processed by P_1 if it is activated first is much greater than the load processed by P_2 in the case when the activation sequence starts with 2. Still, in the optimum solution processor P_2 should receive load before P_1. Thus, a greedy algorithm, always appending to the communication sequence the processor which can process the greatest amount of load, also does not deliver optimum solution.

Finally, it can be conjectured that DLS{1Round} is not a selection problem.

2.5 Approximation Algorithms for Problem DLS{1Round}

Without knowing how to order the processors taking part in the computations for problem DLS{1Round}, we are not able to create similar approximation schemes as for problem DLS$\{C_i = 0\}$. Therefore, we present several algorithms with approximation ratio bounded but dependent on the instance parameters.

2.5.1 Problem DLS{1Round}-Opt$V$

The simplest method of creating a solution of problem DLS{1Round}-Opt$V$ is to send the whole load to a single processor only. The size of the load processed by a single processor P_i in time T is equal to $(T - S_i)/(A_i + C_i)$. Thus, we select the processor for which this value is the greatest, as it is shown in Algorithm 2.5.

Algorithm 2.5 SINGLE-PROCESSOR-OPT-V($T, m, \mathbf{A}, \mathbf{C}, \mathbf{S}$)

```
$j = 1$
for $i = 2$ to $m$ do
  if $(T - S_i)/(A_i + C_i) > (T - S_j)/(A_j + C_j)$ then
    $j = i$
  end if
end for
return $\sigma = (j)$, $V = (T - S_j)/(A_j + C_j)$
```

Note that in the optimum schedule at least one processor P_i must process load of size at least $V_{OPT}(T)/m$ (in given time T). Hence, Algorithm 2.5 delivers a solution processing load of size at least $V_{OPT}(T)/m$ and is an approximation algorithm with relative performance guarantee m. Note that this bound is tight. Consider an instance with $A_i = 1$, $C_i = S_i = 0$ for $i = 1, \dots, m$. In the optimum solution, all processors are activated and they process load of size mT. In the solution delivered by Algorithm 2.5 only one processor is activated and the size of the load is T. The running time of Algorithm 2.5 is $O(m)$.

The above approach can be extended by analyzing all communication sequences of length k for some constant $k \leq m$. Similarly as before, we observe that if the optimum solution of the problem activates at least k processors, then it must contain a group of k processors which together process load of size at least $kV_{OPT}(T)/m$. Hence, an algorithm enumerating all possible communication sequences of length k delivers a solution with relative performance guarantee m/k, provided that the optimum solution of the instance of the problem uses at least k processors. Unfortunately, the complexity of such an algorithm is $O(m^k)$ and it grows exponentially with the relative performance guarantee.

Algorithm 2.5 can be also extended to a greedy Algorithm 2.6, selecting the processors in the communication sequence one by one. As long as it is possible to append a processor to the communication sequence, the processor which can process the greatest load is chosen.

The running time of Algorithm 2.6 is $O(m^2)$. The results delivered by this algorithm are not worse then for Algorithm 2.5. Still, the performance guarantee m is tight. Indeed, consider the following problem instance. Let $A_1 = 1 - \varepsilon$, $C_1 = T - 1$, $S_1 = 0$, and $A_i = T$, $C_i = 0$, $S_i = 0$ for $i = 2, \dots, m$, where $0 < \varepsilon < 1$ is a small constant. Processor P_1 can process load of size $\frac{T}{A_1 + C_1} = \frac{T}{T - \varepsilon} > 1$ in time T. For $i \geq 2$, processor P_i is capable of processing load of size $\frac{T}{T} = 1$ in time T. Hence, Algorithm 2.6 will choose processor P_1 to obtain the first load

Algorithm 2.6 GREEDY-OPT-V($T, m, \mathbf{A}, \mathbf{C}, \mathbf{S}$)

```
$\sigma = ()$
$V = 0$
$j = 1$
**while** $j \neq 0$ **do**
  $j = 0$
  **for** $i = 1$ **to** $m$ **do**
    **if** $S_i < T$ **and** $i$ is not contained in $\sigma$ **then**
      **if** $j = 0$ **or** $(T - S_i)/(A_i + C_i) > (T - S_j)/(A_j + C_j)$ **then**
        $j = i$
      **end if**
    **end if**
  **end for**
  **if** $j \neq 0$ **then**
    $\sigma = \sigma | j$    {concatenation of $\sigma$ and $j$}
    $V = V + (T - S_j)/(A_j + C_j)$
    $T = T - S_j - C_j(T - S_j)/(A_j + C_j)$
  **end if**
**end while**
**return** $\sigma$, $V$
```

chunk. Sending data to processor P_1 will take time $T_1 = C_1 \frac{T}{T-\varepsilon} = (T-1)\frac{T}{T-\varepsilon}$. The remaining processors P_i will be activated afterwards and each of them will obtain the load of size $(T - T_1)/A_i = \frac{(T-(T-1)\frac{T}{T-\varepsilon})}{T} = 1 - \frac{T-1}{T-\varepsilon} = \frac{1-\varepsilon}{T-\varepsilon}$. Thus, the total size of the processed load will be $V_1 = \frac{T+(m-1)(1-\varepsilon)}{T-\varepsilon}$.

On the other hand, if processor P_1 is activated as the last one, then each of processors $P_2, \ldots, P_m$ receives load of size 1. The time left for communication and computation on P_1 is still T, and P_1 processes load of size $\frac{T}{T-\varepsilon}$. The whole processed load has size $V_2 = m - 1 + \frac{T}{T-\varepsilon}$. Thus, we have $\frac{V_2}{V_1} = \frac{mT-\varepsilon(m-1)}{(m-1)(1-\varepsilon)+T}$ and $\lim_{T\to\infty} \frac{V_2}{V_1} = m$.

The quality of the results obtained by Algorithm 2.6 in comparison to Algorithm 2.5 strongly depends on the processor parameters. To analyze the difference

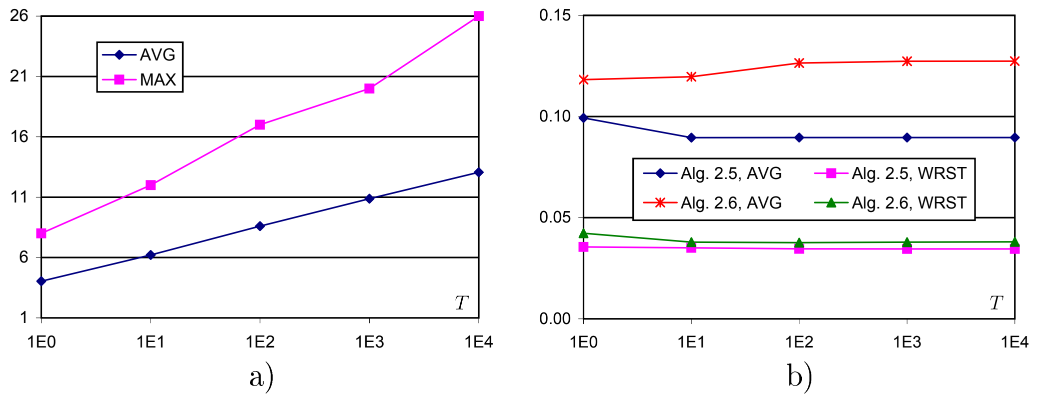


Figure 2.2: Experimental results for the first set of instances (slow communication). a) Number of processors used by Algorithm 2.6. b) Quality of the solutions obtained by Algorithms 2.5 and 2.6.

between the two algorithms we tested them on sets of random instances. Each instance in the first set had $m = 100$ processors, and their parameters A_i, C_i, S_i were chosen randomly from the interval $[0, 1]$. For each generated set of processors, 5 instances were created, with $T = 1, 10, 100, 1000, 10000$. The quality of the obtained solutions was measured as the quotient $\frac{V_a}{UpBo}$, where V_a is the amount of load returned by the tested algorithm, and $UpBo$ is the upper bound on the size of processed load, calculated as $\sum_{i=1}^{m} \frac{T-S_i}{A_i+C_i}$. The results of the experiments are presented in Fig. 2.2. The number of processors used by the greedy Algorithm 2.6 depends on T (see Fig. 2.2a). Although for each value of T there were instances for which only one processor was used, the average and the maximum number of used processors (denoted by AVG and MAX in Fig. 2.2a, correspondingly) increases with T. Despite this, the performance of Algorithm 2.6 does not change much with growing T (cf. Fig. 2.2b), both on average (denoted AVG) and in the worst case (denoted $WRST$). This can be explained by the fact that the processors activated as the last ones receive only very small amounts of load. Moreover, when startup times S_i are small in comparison to T, then the amounts of load processed by a single processor or a fixed group of processors increase roughly linearly with T. The upper bound on the total size of processed load also increases

Table 2.3: The quality of the solutions obtained by Algorithms 2.5 and 2.6 for the second set of instances (fast communication), $T = 10000$.

Algorithm 2.5		Algorithm 2.6	
AVG	WRST	AVG	WRST
0.271073	0.061405	0.855082	0.598274

linearly with T. Hence, the quality of the results obtained by both algorithms is almost constant in relation to the upper bound when T grows beyond 100. Note that it is much better than the worst-case estimate $\frac{1}{m} = 0.01$. It can be also seen in Fig. 2.2b that on average Algorithm 2.6 delivers solutions about 1.5 times better than Algorithm 2.5.

The above results can be explained by the fact that the communication parameters C_i, S_i were chosen from the same range as A_i. The time necessary to send a chunk of data was quite big and only a small number of processors could be activated. Therefore, we created another set of instances, where parameters C_i and S_i were chosen randomly from the interval $[0, 0.001]$. The remaining parameters were selected as in the previous set. Since the startup times S_i were very small in comparison to all used value of T, the quality of the obtained solutions was almost not changing with T. Therefore, we present only the average and the worst performance of both algorithms for $T = 10000$ in Table 2.3. The number of processors used by Algorithm 2.6 was $m = 100$ for all instances in this set. Therefore, the difference between the results obtained by Algorithms 2.5 and 2.6 is greater than for the previous set of instances, for which at most 26 processors were used by the greedy algorithm. The quality of the results of both algorithms is better than for the previous instance set. On average, Algorithm 2.5 allows for processing load of size greater than 27% of the upper bound and Algorithm 2.6 greater than 85%.

We conclude that the difference in the quality of the results obtained by Algorithms 2.5 and 2.6 depends on the communication parameters of the processors.

If communication is slow, then the quality of the obtained results is not very good. However, this can be the effect of the used measure of quality. When communication is slow, the upper bound we calculated may be much greater than the optimum solution. If communication is fast in comparison to computations, then the results obtained by both algorithms get better. The difference between the results of Algorithms 2.5 and 2.6 is increasing and the greedy Algorithm 2.6 delivers solutions of very good quality.

2.5.2 Problem DLS{1Round}-Opt$T$

In order to create an approximation algorithm for problem DLS{1Round}-Opt$T$, we can, similarly as in Algorithm 2.5, consider only communication sequences of length 1. This approach is used in Algorithm 2.7.

Algorithm 2.7 SINGLE-PROCESSOR-OPT-T($V, m, \mathbf{A}, \mathbf{C}, \mathbf{S}$)

```
$j = 1$
for $i = 2$ to $m$ do
    if $S_i + (A_i + C_i)V < S_j + (A_j + C_j)V$ then
        $j = i$
    end if
end for
return $\sigma = (j)$, $T = S_j + (A_j + C_j)V$
```

Note that if processor P_i needs time T to process the load of size V, then it cannot process the load of size V/m faster than in time T/m. As in the optimum solution at least one processor has to receive load of size at least V/m, Algorithm 2.7 returns time $T \leq mT_{OPT}(V)$. Observe that this bound is tight. Consider an instance with $A_i = 1$, $C_i = S_i = 0$ for $i = 1, \ldots, m$. In the optimum solution, all processors are activated and they process load V in time $\frac{V}{m}$. In the solution delivered by Algorithm 2.7 only one processor is activated and it needs time V to process the whole load. The running time of Algorithm 2.7 is $O(m)$.

2.6 Conclusions

In this chapter we analyzed single-round divisible load scheduling in star networks. We proposed fully polynomial time approximation schemes for problems DLS$\{C_i = 0\}$-Opt$V$ and DLS$\{C_i = 0\}$-Opt$T$. As a by-product, a fully polynomial time dual approximation algorithm was designed for the first problem. We also analyzed the scheduling problems in the system with finite bandwidths (i.e. when $C_i > 0$). The order in which the processors should be activated was studied as the main obstacle in creating approximation algorithms for this case. Unfortunately, we showed that some classes of processor sequencing algorithms cannot be used to solve this problem. We conjecture that constructing the optimum sequence can be computationally hard, and DLS{1Round} is not a selection problem. Finally, we proposed simple approximation algorithms giving tight relative performance guarantee m for problem DLS{1Round}-Opt$V$ and for problem DLS{1Round}-Opt$T$.

3 Multi-Round Processing with Limited Memory

The single-round organization of computations has several disadvantages. Firstly, the communication delays may be very long, while no computations can be started until the first processor receives the whole amount of load assigned to it. Secondly, in practice the whole load V is often too big to be stored in the memories of worker processors at the same time. In such a case it is impossible to create a single-round schedule. It would be more profitable to send the load in many small pieces (chunks), so that computations start earlier and fit in computer memories. Consequently, computations could interleave with communications.

In this chapter we study multi-round divisible load scheduling in systems with limited memory. We analyze the star network topology described in Chapter 2. To take into account memory limitations, we introduce one more parameter characterizing each processor P_i. Namely, B_i is the size of memory buffer available on P_i (e.g. in bytes). Our goal is to find a schedule processing the load of a given size in the shortest possible time. As each processor can receive many messages, there are more scheduling decisions to be made than in the case of single-round processing:

- The set $P' \subseteq P$ of processors participating in the computations must be chosen.
- The length n of the communication sequence must be selected. It may be

much larger then the number of processors m.

- The communication sequence must be chosen. For multi-round processing, the communication sequence is an arbitrary sequence whose elements are indices of processors from the set P'.
- The sizes of the load parts sent in each message must be selected.

We start our considerations with a short summary of the previous work on multi-round divisible load scheduling. In Section 3.2 we describe the mathematical model used in this chapter. As our scheduling problem is known to be computationally hard, we propose an exponential Branch&Bound algorithm and a genetic algorithm in Section 3.3. We use the genetic algorithm not only as a metaheuristic solving the scheduling problem, but also to gather information about the features of good quality solutions. The results obtained from an extensive experimental study, as well as some analytical results, are presented in Section 3.4. Based on this information, in Section 3.5 we propose several classes of scheduling heuristics. We analyze and compare them, exposing their advantages and weaknesses.

3.1 Earlier Results

Scheduling divisible loads in systems with limited memory was first analyzed in [37]. The authors considered single-round schedules only, hence they assumed that the whole load fits in the memory buffers of the workers. Other assumptions were that all processors take part in the computations and that the activation sequence is given. The communication delay model was linear ($S_i = 0$ for $1 \leq i \leq m$). A fast heuristic called Incremental Balancing Strategy was proposed. This algorithm did not always deliver optimum solutions, what was shown in [30].

A more general affine communication delay model was studied in [30]. A linear programming formulation of the scheduling problem was designed for a given activation sequence. Choosing the optimum set P' of processors taking part in the computations in systems with limited memory and affine communication model has been shown to be **NP**-hard in [31] and strongly **NP**-hard in [4]. In [31] the authors proposed and evaluated experimentally a Branch&Bound algorithm and several heuristics for single-round scheduling with limited memory.

Multi-round divisible load scheduling with limited memory was first studied in [29]. Only the size of the chunk currently processed by a given processor was subject to the memory limit. The sizes of load parts arriving in the background of computations were not taken into account. A more detailed memory model, in which memory limits affected all chunks of data existing at a given processor, was used in [26]. A Branch&Bound algorithm and a genetic algorithm solving the analyzed scheduling problem were proposed. However, the mathematical model of memory management was simplified to make the problem more tractable. It was assumed that memory occupation is decreasing linearly during the computations. This simplification has been removed in [27]. We discuss it in more detail in the next section.

3.2 Problem Formulation

Before we present the mathematical model used in this chapter, let us briefly analyze different models of memory management. The simplest approach is to assume that only one load chunk may be present in the memory of a computer at a time [31, 37]. The size of a piece of data sent to processor P_i cannot exceed the limit B_i (cf. Fig. 3.1a). Thus, a processor cannot perform computations while receiving a new piece of load. This results in long idle times and decreases the efficiency of processing.

In [26] it was assumed that each processor can store multiple load chunks at the same time and the size of these chunks together cannot exceed the limit

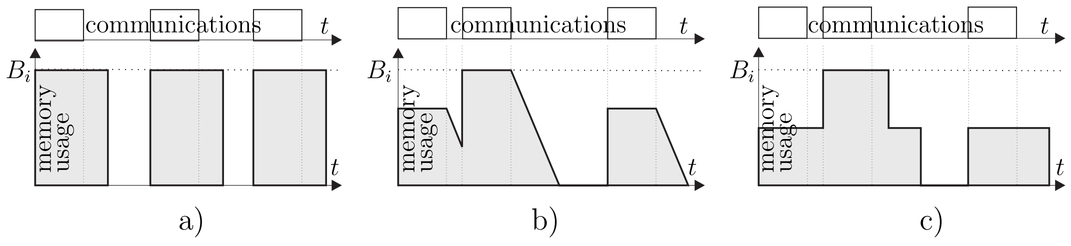


Figure 3.1: Memory management: a) each chunk uses whole buffer, b) memory gradually released, c) block memory releases.

B_i. It was possible to gradually upload the data without stopping the computations. Consequently, the computations could be started quickly by sending short initial chunks, and performed continuously by uploading data while computing. However, to make the problem more tractable, it was also assumed in [26] that memory is released to the operating system with very fine granularity. The size of allocated memory was decreasing linearly during the computations, as shown in Fig. 3.1b, and it was possible to compute the optimum load chunk sizes using linear programming for a given communication sequence. However, this way of releasing memory is rather unusual, because releasing memory in many small pieces would also require allocating memory in very fine pieces. Obtaining sufficient memory for a piece of load would consist of multiple malloc/new calls to the runtime environment. Consequently, acquiring memory would be complicated and time-consuming.

Therefore, in this work we assume that memory allocation and release have block nature. When a load chunk of size α is about to arrive at a processor, a block of memory of size α is requested from the operating system. This block is released immediately after finishing processing the corresponding chunk of data (cf. Fig. 3.1c). The sum of sizes of memory blocks coexisting at processor P_i cannot exceed the limit B_i. In other words, for each moment t, we have $\sum_{l\in\mathcal{H}(i,t)} \alpha_l \le B_i$, where $\mathcal{H}(i,t)$ is the set of chunks received by P_i and not completed by time t. We will be saying that chunks simultaneously existing in the memory buffer *overlap*.

Let us introduce the assumptions and notations necessary to formulate our scheduling problem as a mixed nonlinear mathematical program. The load is delivered to the processors in a sequence of communications. The activation sequence may be arbitrary. In particular, some processors may receive no load, while some other processors receive multiple data chunks. If the message is received by a processor without any load in the buffer, then the computations start immediately after the end of communication. If the buffer already stores some unprocessed chunks, then the processor switches from computing one load chunk to the next one without idle time in the computations. If the whole memory buffer of a processor is occupied, then no more load can be uploaded and, consequently, idle times in communication may appear. We assume that the load chunks assigned to a given worker are processed in the order in which they were received. Let us assume that the sequence $\sigma = (\sigma(1), \dots, \sigma(n))$ of the communications to the processors is given, where $\sigma(i)$ is the index of the processor receiving the i-th chunk. The size of this chunk is α_i. The numbers of the load chunks as they are sent off the originator will be called *global* numbers. For simplicity of notation we will also use a *local* numbering of the chunks received by a certain processor. We define a function $\rho(i, j)$ as a mapping from processor P_i local chunk number j to the global numbering. The number of load pieces received by processor P_i will be denoted by n_i. In the mathematical program we want to construct, it must be guaranteed that chunks simultaneously existing in a processor buffer do not exceed the memory size. To formulate such a constraint we have to know the sets of overlapping load chunks. However, this depends on the communication sequence, chunk communication and computation times, and hence, on the chunk sizes, which are unknown. Let us define binary variables x_{ijk} for $1 \leq i \leq m$, $1 \leq j < k \leq n_i$ in the following way. Variable x_{ijk} is equal to 1 if the j-th chunk on processor P_i overlaps with chunk k on this processor, and equal to 0 otherwise. In other words, $x_{ijk} = 1$ means that processor P_i started receiving chunk k before computing the j-th chunk was finished. Both k and j are local chunk numbers.

Our scheduling problem can be formulated in the following way [11, 27].

minimize T_{max}
subject to

$$\begin{aligned}
t_1 &= 0 && (3.1)\\
t_i &\geq t_{i-1} + S_{\sigma(i-1)} + C_{\sigma(i-1)}\alpha_{i-1} \quad i = 2, \ldots, n, && (3.2)\\
f_{ik} &\geq t_{\rho(i,k)} + S_i + C_i\alpha_{\rho(i,k)} + A_i\alpha_{\rho(i,k)} && (3.3)\\
& \ i = 1, \ldots, m, \quad k = 1, \ldots, n_i, && \\
f_{ik} &\geq f_{i,k-1} + A_i\alpha_{\rho(i,k)} && (3.4)\\
& \ i = 1, \ldots, m, \quad k = 2, \ldots, n_i, && \\
f_{ij} &\geq t_{\rho(i,k)} - (1 - x_{ijk})M \quad i = 1, \ldots, m, && (3.5)\\
& \ j = 1, \ldots, n_i - 1, \ k = j+1, \ldots, n_i && \\
f_{ij} &\leq t_{\rho(i,k)} + x_{ijk}M \qquad i = 1, \ldots, m, && (3.6)\\
& \ j = 1, \ldots, n_i - 1, \ k = j+1, \ldots, n_i && \\
x_{ijk} &\leq x_{ilk} \quad i = 1, \ldots, m, \ j = 1, \ldots, n_i - 1, && (3.7)\\
& \ k = j+2, \ldots, n_i, \ l = j+1, \ldots, k-1 && \\
x_{ijk} &\geq x_{ijl} \quad i = 1, \ldots, m, \ j = 1, \ldots, n_i - 1, && (3.8)\\
& \ k = j+1, \ldots, n_i, \ l = k+1, \ldots, n_i && \\
\alpha_{\rho(i,j)} + \sum_{k=j+1}^{n_i} x_{ijk}\alpha_{\rho(i,k)} &\leq B_i \quad i = 1, \ldots, m, \ j = 1, \ldots, n_i && (3.9)\\
V &= \sum_{i=1}^{n} \alpha_i && (3.10)\\
T_{max} &\geq f_{in_i} \quad i = 1, \ldots, m && (3.11)\\
x_{ijk} &\in \{0, 1\} && (3.12)
\end{aligned}$$

In the above formulation variables α_i define the load partitioning resulting in the minimum schedule length for the communication sequence σ. Inequalities (3.1), (3.2) determine the moments t_i when the originator starts sending the i-th chunk. Constraints (3.3),(3.4) determine the moment f_{ik} when processing chunk k of P_i finishes. Inequalities (3.5), (3.6) guarantee that processing of chunk j is finished before starting sending message k if $x_{ijk} = 0$, or that it is not finished before starting sending message k if $x_{ijk} = 1$. Due to inequalities (3.7), if chunk j is not processed when chunk k arrives, then the chunks between j and k are also unprocessed. Inequalities (3.8) ensure that if chunk j is finished before arriving of some chunk k, then j cannot become unprocessed again. By inequalities (3.9) memory limits are observed. The whole load is processed by (3.10). The schedule length is not shorter than the completion time on any processor by constraints (3.11). Formulation (3.1)-(3.12) is a mixed quadratic mathematical program, as it uses binary variables (x_{ijk}), continuous variables ($\alpha_i, f_{ik}, t_i, T_{max}$), and multiplication of variables in constraints (3.9). Solving mixed quadratic programs is computationally hard. Thus, it can be expected that solving the program (3.1)-(3.12) using general-purpose methods is computationally hard although the activation sequence σ is given. This is in sharp contrast with the complexity of memory management models used in [26, 31], for which linear programs were sufficient to obtain the optimum load partition for a given communication sequence σ. It can be seen that a more careful representation of memory management and chunk overlap made the mathematical model much more involved. Note that (3.1)-(3.12) is very general and may cover various scenarios of optimum memory management. For example, it is capable of representing a number of independent buffers of equal or different sizes swapped on the processors.

Let us note that for given x_{ijk} the formulation (3.1)-(3.12) becomes a linear program (LP). Hence, we will split our problem into two parts. The first, combinatorial part is to choose not only the communication sequence, but also to decide which chunks overlap with each other. The second, algebraic part is to find the optimum load distribution using the linear program for a given communication sequence and overlap information. In the following discussion, we will use a simpler overlap encoding. Instead of binary variables x_{ijk} we will use integer variables z_{ij}, where z_{ij} is the local number of the last chunk overlapped by chunk j on processor P_i. Intuitively, z_{ij} denotes the end of the range of overlapping chunks comprising chunk j on P_i.

The mathematical program computing the optimum load distribution for a given communication sequence σ and overlap information encoded by values z_{ij}, may be formulated as follows [12].

minimize T_{max}
subject to

$$\begin{aligned}
t_1 &= 0 && (3.13)\\
t_i &\geq t_{i-1} + S_{\sigma(i-1)} + C_{\sigma(i-1)}\alpha_{i-1} \quad i = 2, \dots, n, && (3.14)\\
f_{ik} &\geq t_{\rho(i,k)} + S_i + C_i\alpha_{\rho(i,k)} + A_i\alpha_{\rho(i,k)} && (3.15)\\
&\quad i = 1, \dots, m, \quad k = 1, \dots, n_i, &&\\
f_{ik} &\geq f_{i,k-1} + A_i\alpha_{\rho(i,k)} && (3.16)\\
&\quad i = 1, \dots, m, \quad k = 2, \dots, n_i, &&\\
f_{ij} &\geq t_{\rho(i,z_{ij})} \quad i = 1, \dots, m,\ j = 1, \dots, n_i - 1 && (3.17)\\
f_{ij} &< t_{\rho(i,z_{ij}+1)} \quad i = 1, \dots, m,\ j = 1, \dots, n_i - 1 && (3.18)\\
\sum_{k=j}^{z_{ij}} \alpha_{\rho(i,k)} &\leq B_i \qquad i = 1, \dots, m,\ j = 1, \dots, n_i && (3.19)\\
V &= \sum_{i=1}^{n} \alpha_i && (3.20)\\
T_{max} &\geq f_{in_i} \qquad i = 1, \dots, m && (3.21)
\end{aligned}$$

In the above formulation constraints (3.13)-(3.18) correspond to (3.1)-(3.6), and constraints (3.19)-(3.21) correspond to (3.9)-(3.11).

3.3 Branch&Bound Algorithm and Genetic Algorithm

In this section we propose two basic algorithms solving our scheduling problem. We start with an exponential Branch&Bound algorithm. Since its running time

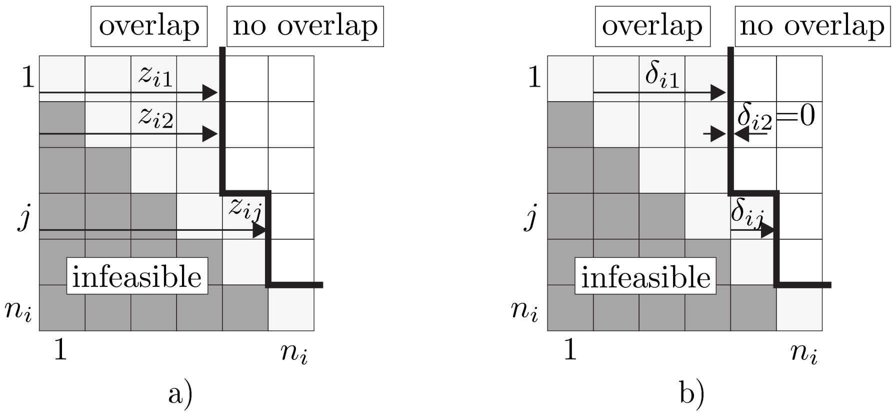


Figure 3.2: Encoding overlaps on P_i using a) z_{ij}, b) δ_{ij}.

is unacceptable for practical use, we construct a genetic algorithm. We tune its parameters based on the results delivered by the Branch&Bound algorithm. Both algorithms solve the combinatorial part of the problem and use the linear program (3.13)-(3.21) to solve the algebraic part.

Before we present the algorithms, let us introduce a more practical overlap encoding, which was used in the actual implementation. The last chunk z_{ij} overlapping with chunk j cannot be sent before j. Thus, values $z_{ij} < j$ are infeasible. Note that if chunk j on processor P_i overlaps with chunk $k > j$, then it must also overlap with all chunks between j and k. Moreover, chunks with numbers greater then z_{ij} cannot overlap with j anymore. Hence, there is a line separating the overlapping and the non-overlapping chunks. Instead of z_{ij} we can use integer variables δ_{ij} denoting by how many chunks the overlapping front is shifted ahead with chunk j on processor P_i (cf. Fig. 3.2). For given values of δ_{ij} we can compute values $z_{ij} = \min\{n_i, \max\{z_{i,j-1}, j\} + \delta_{ij}\}$, where $z_{i0} = 1$. In other words, δ_{ij} is encoding increments $z_{ij} - z_{i,j-1}$. For example, if $\forall i, j, \delta_{ij} = 0$, then chunks do not overlap, if $\forall i, j < n_i, \delta_{ij} = 1$, then each pair of consecutive chunks overlap. This overlap encoding is used in all the following algorithms which directly refer to overlap values.

3.3.1 Branch&Bound Algorithm

A Branch&Bound algorithm (B&B) is a standard technique used to solve hard combinatorial optimization problems. The algorithm is defined by a branching rule and a bounding rule. The branching rule divides the set of possible solutions until distinguishing unique solutions. The bounding rule eliminates the solutions which are infeasible or their quality is certainly not better than the quality of some already known solution.

In our scheduling problem the Branch&Bound algorithm has to find a communication sequence σ and determine chunk overlapping. The communication sequences are built by appending a new processor to an already constructed leading sequence. Thus, any partial sequence σ represents all sequences starting with σ. This set of sequences is branched into subsets of sequences beginning with $(\sigma, P_1), \dots, (\sigma, P_m)$. For each analyzed communication sequence σ chunk overlapping must be chosen. This is done by the second branching scheme. For processor P_i the overlap is determined by a vector $(\delta_{i1}, \dots, \delta_{in_i})$. A sequence $(\delta_{i1}, \dots, \delta_{ij})$ encoding the overlap for the first j chunks received by P_i, is branched into overlap encoding strings $(\delta_{i1}, \dots, \delta_{ij}, 0), \dots, (\delta_{i1}, \dots, \delta_{ij}, n_i - \max\{j+1, z_{ij}\})$.

The enumeration of possible solutions is bounded by two methods. For a given sequence σ a lower bound $LB(\sigma)$ on the schedule length is computed as follows. The startup times in σ are summed up: $\tau_1 = \sum_{i=1}^{n} S_{\sigma(i)}$. The maximum load V' that could be processed during the communication startup times is $V_1 = \sum_{i\in\sigma}(\tau_1 - \sum_{j=1}^{g(i)} S_{\sigma(j)})/A_i$, where $g(i)$ is the index of the first communication to processor P_i in σ. The notation $\sum_{i\in\sigma}$ means that if $i \in \sigma$, then it is counted only once, like a member of a set. The load must be sent from the originator in time at least $\tau_2 = VC_{min}$. In parallel with this communication, at most $V_2 = \tau_2 \sum_{i=1}^{m} \frac{1}{A_i}$ units of load could be processed. If $V_3 = V - V_1 - V_2 > 0$, then this remaining load V_3 will be processed in time at least $\tau_3 = V_3/(\sum_{i=1}^{m} \frac{1}{A_i})$. The lower bound is equal to $LB(\sigma) = \tau_1 + \tau_2 + \max\{0, \tau_3\}$. Let T be the length of the best already known solution. If $T \leq LB(\sigma)$ then the successors of σ are discarded. The second mechanism used for sequence elimination is based on the maximum memory $MEM(\sigma) = \sum_{i=1}^{n} B_{\sigma(i)}$ which could possibly become available in σ. If $MEM(\sigma) < V$, then the memory available for holding the load is insufficient, the communication sequence is too short and must be expanded. In such a case the enumeration of the various overlap sequences was not attempted for the given σ. Note that there are $O(m^n)$ communication sequences of length n for m processors, and for each processor the number of possible ways of overlapping the communication chunks is also exponential in n_i. Hence, due to the high computational complexity, an upper bound n_{MAX} on the length n of generated sequences was also imposed. This was done to make the B&B algorithm more usable, and it was not needed to properly define the algorithm. Consequently, because of the constraint n_{MAX}, in some cases B&B was not able to deliver an optimum, or even a feasible solution.

3.3.2 Genetic Algorithm

A genetic algorithm (GA), similarly as B&B, is a standard technique used to solve hard combinatorial optimization problems. The idea of the genetic algorithm is to mimic the process of evolution in nature. GA is a randomized algorithm which maintains a population of solutions (called chromosomes) instead of a single solution only. Genetic operators are used to transform the population in the direction of improving solutions quality. To define a genetic algorithm, one has to determine solution encoding, the set of genetic operators, algorithm stopping criteria and several implementation-dependent tunable parameters.

In our implementation of GA we encode solutions as pairs of sequences of equal length. The first of them is the communication sequence σ. The second sequence O is used to represent the overlap. More precisely, $O(i)$ is the value of $\delta_{\sigma(i)j}$, where j is the number of load chunks sent to processor $P_{\sigma(i)}$ up to the i-th chunk sent off the originator. The (equal) lengths of σ and O can be adjusted by GA to construct the best solution. Knowing the sequences σ and O, we can formulate the linear program (3.13)-(3.21) calculating values α_i and T_{max} defined in Section 3.2. The fitness (quality) of the solution is measured as the inverse of the schedule length T_{max} obtained from the linear program.

We apply three genetic operators: selection, crossover and mutation. The selection of the solutions for the new population is done by a combination of elitist and roulette wheel method and is strongly connected with the crossover operation. First, chromosomes which should undergo crossover operation are chosen. Chromosomes are selected with probability $\frac{1}{T^j_{max}}/\sum_{j=1}^{G}\frac{1}{T^j_{max}}$, where T^j_{max} denotes the schedule length for chromosome j, and G is the size of the population. The total number of selected parents is Gp_C, where p_C is a tunable algorithm parameter called crossover probability. In the crossover operation the selected parents are randomly paired and combined. For example, let
$[(\sigma_1(1),\dots,\sigma_1(n')),(O_1(1),\dots,O_1(n'))]$
and
$[(\sigma_2(1),\dots,\sigma_2(n'')),(O_2(1),\dots,O_2(n''))]$
be two parent solutions, with communication sequence lengths n', n'', respectively. Let $k \le n', l \le n''$ be two randomly chosen crossover points. The two offspring solutions are encoded in strings
$[(\sigma_1(1),\dots,\sigma_1(k),\sigma_2(l+1),\dots,\sigma_2(n'')),(O_1(1),\dots,O_1(k),O_2(l+1),\dots,O_2(n''))]$,
and
$[(\sigma_2(1),\dots,\sigma_2(l),\sigma_1(k+1),\dots,\sigma_1(n')),(O_2(1),\dots,O_2(l),O_1(k+1),\dots,O_1(n'))]$.
The offspring replaces the parents in the new population. Note that because of choosing two crossover points l and k the offspring string lengths may be different than in their parents. The rest of the new population is selected by the elitist method, so that the best $(1-p_C)G$ chromosomes from the old population are always preserved. The elitist component in the selection is necessary because the

differences in the solution fitness are often very small, and the best solutions may be lost in the randomized selection.

Mutation operator changes randomly chosen genes (i.e. pairs $(\sigma(i), O(i))$) in the population to different values. Each gene is chosen for mutation with probability p_M. Here p_M is a tunable algorithm parameter called mutation probability. When gene $(\sigma(i), O(i))$ is mutated, the number $\sigma(i)$ is changed to a randomly chosen processor index between 1 and m, and the value $O(i)$ is changed by at most 1.

The algorithm stops after a fixed number of iterations it_1. There is also a limit it_2 on the number of iterations without an improvement in the quality of the best solution found so far. If the iteration limit it_2 is reached before it_1, then the population is replaced with randomly generated chromosomes and the search is restarted (the best solution found so far is recorded).

GA is a randomized algorithm whose parameters must be tuned. The following procedure was applied. A set of 200 random instances with $m = 3, \dots, 6, V = 20$, B_i uniformly distributed in $[0, 10]$, A_i, C_i, S_i uniformly distributed in $[0, 1]$, were generated and solved to the optimum by B&B. The average relative distance of the schedule length T_{max} from the optimum length was the measure of the tuning quality. The tunable parameters were selected one by one. The process of selecting the tunable parameters is illustrated in Fig. 3.3. Intuitively, a big population size G should allow for finding good solutions in small number of iterations. However, maintaining big populations is computationally expensive. The population size $G = 20$ was selected as a compromise between the speed of convergence to the near-optimum solutions, and the computational complexity (cf. Fig. 3.3a). To select the crossover probability, the mutation operator was switched off. Crossover probability $p_C = 0.8$ was selected (Fig. 3.3b). It turned out that the majority of the population (80%) are offspring. Thus, it can be concluded that crossover is an effective optimization operator. After fixing G and p_C,

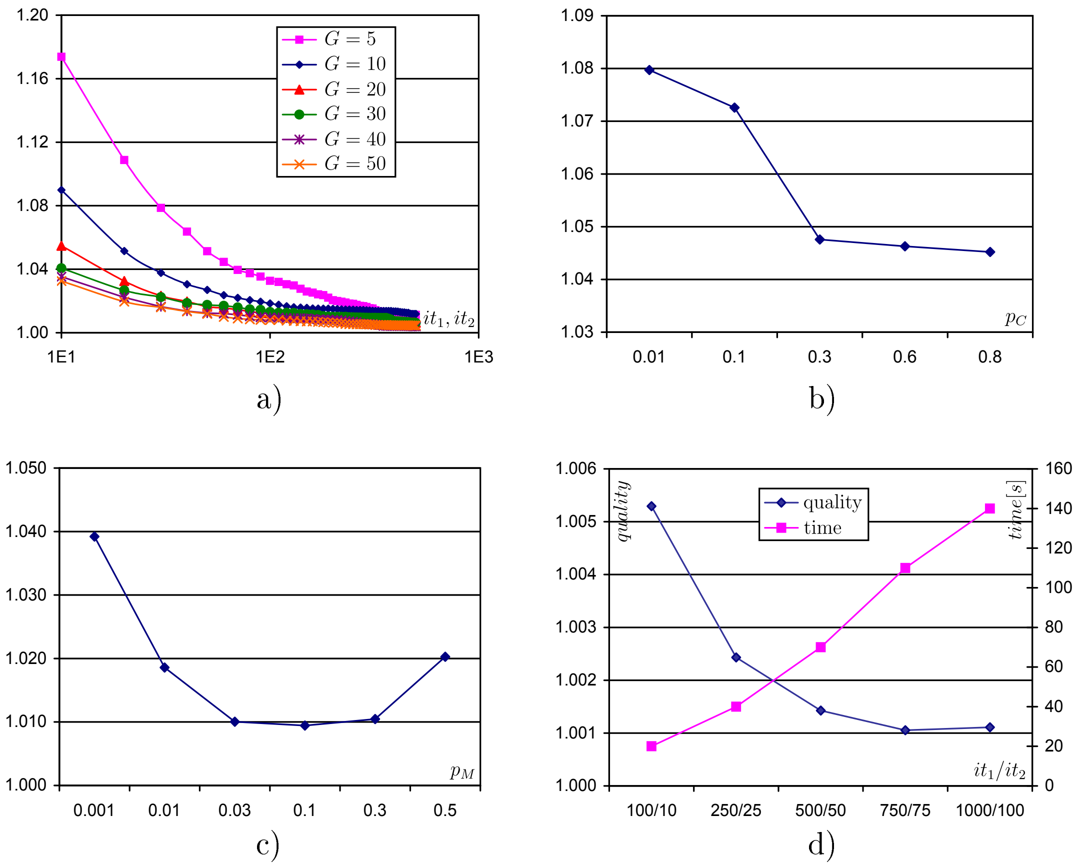
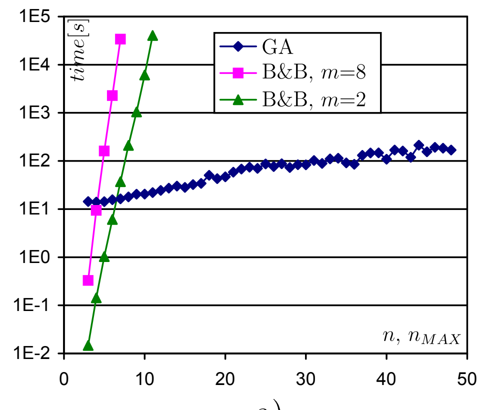


Figure 3.3: GA tuning. a) Solution quality vs. population size G, b) solution quality at 100th iteration vs. p_C, c) solution quality at 100th iteration vs. p_M, d) solution quality and execution time for various iteration limits it_1/it_2.

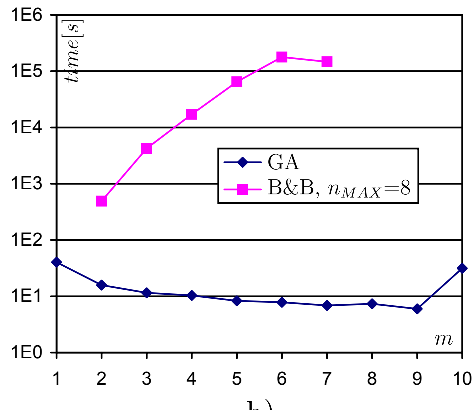
mutation probability $p_M = 0.1$ was chosen (Fig. 3.3c). In Fig. 3.3d the quality of tuning is shown for various combinations of maximum number of iterations and iterations without quality improvement. Note that improving the average solution quality by 0.4% requires nearly 6-fold increase of the execution time. Hence, $it_1 = 100$ and $it_2 = 10$ were selected as a compromise between quality and complexity.

3.3.3 Comparison of B&B and GA

Let us now discuss the advantages and weaknesses of B&B and GA algorithms. In general, B&B guarantees obtaining optimum solutions, but at very high computational cost. Therefore, we had to impose a limit n_{MAX} on the maximum



a)



b)

Figure 3.4: GA and B&B execution times, a) vs. sequence length, b) vs. processors number m.

number of communications in the schedule. This modified B&B algorithm can deliver optimum solutions only for such instances, for which the optimum communication sequence is short enough. In Fig. 3.4 we compare average execution time of B&B and GA on a Pentium IV 1 GHz CPU. In the case of B&B the execution time is shown as a function of n_{MAX} (Fig. 3.4a). We use n_{MAX} because it turned out that this parameter is the main factor determining the size of the search tree in B&B. The minimum possible communication sequence length is $n_{MIN} = \lceil \frac{V}{B_{max}} \rceil$. It is hardly ever the length of the best sequence, or the depth of the B&B search tree. To be certain that the best communication sequence obtained in B&B is indeed optimum, it must have length at most $n_{MAX} - 1$. Instances satisfying this condition are easier to solve than the instances which force B&B to search a tree as deep as n_{MAX}, and presenting the execution times as a function of the guaranteed optimum communication sequence length would not represent the real execution time of B&B. As it can be seen, even average execution time of B&B for $n_{MAX} = 7, m = 8$ is of order of one day on a Pentium IV 1 GHz CPU. Hence, B&B is not an acceptable tool for studying features of great numbers of even moderate size instances. For GA, the execution time is shown vs. the length n of the best obtained communication sequence. In Fig. 3.4b the execution time vs. the number of processors m is shown.

From the tuning process described in the previous section we conclude that GA is capable of delivering high quality solutions on average. The running time of GA is much shorter than for B&B, what can be seen in Fig. 3.4. The main disadvantage of GA as a tool for analyzing the problem properties is that it is a randomized algorithm. In the limit of infinite iteration number, all feasible solutions are reachable in a process of random transformations of the solutions. However, for a finite number of iterations we have no guarantee that the algorithm finds a near-optimum solution. Solutions which are not optimum may be too easy to find by GA, what may give wrong indications on the nature of the solved problem. Another feature of GA is that solutions which have complex structure may be too improbable to be built in a finite number of iterations. For example, the communication sequence may include some processor which is not present in the optimum solution, because the probability of selecting any processor at least once in the sequence is high. Conversely, it is very unlikely that GA builds a long repetitive pattern of communications because the probability of generating a certain pattern decreases exponentially with its length. Another consequence of randomness is that for the same instance GA often returns different solutions in consecutive runs. For example, for a set of 45 random instances each solved 20 times, the quotient $\frac{T_{max}}{\overline{T_{max}}}$ where $\overline{T_{max}}$ is the average schedule length in all runs for a single instance, had the coefficient of variation 6%, and the average (over all quotients $\frac{T_{max}}{\overline{T_{max}}}$) was 0.9997.

We have to conclude that B&B is nearly unusable even on very moderate size instances. GA has much shorter execution time, and in the range in which it could be compared against B&B, the quality of the GA solutions is very good. Hence, despite the limitations of GA, we will use this algorithm as a replacement of B&B in the analysis of the scheduling problem features.

3.4 Properties of the Solutions

In this section we analyze the characteristics of the near-optimum solutions of our scheduling problem. As the algebraic part of the problem is solved optimally for given sequences σ and O, we concentrate on the features in the combinatorial part of the solutions. The following properties are studied:

- the need and the extent of the overlap,
- the length of the communication sequence,
- the number of used processors,
- the set of used processors,
- chunk sizes,
- parameters of instances which make them easy or hard to solve.

We draw conclusions both analytically and on the basis of experimental results. Both GA and B&B were implemented in GNU C++. Linear programs were solved using `lp_solve` package [41]. Over 30000 test instances were generated and solved by GA on Pentium IV 1 GHz CPU with Linux. Unless stated otherwise, the test data were generated in the following way. In the experiments involving analysis of the influence of the system parameters A, B, C, S on solution characteristics, the instance parameters A_i, B_i, C_i, S_i were generated from $U(0, 1]$, i.e. the uniform distribution within range (0,1]. The number of processors was generated from $U[1, 10]$, and all experiments were repeated for $V \in \{2, 5, 10, 20, 50\}$. In the experiments concerning a certain parameter (say A), this parameter was fixed to a given value on all processors (e.g. $\forall i, A_i = 0.01$), and the remaining parameters were generated as described above. For each combination of V and a certain value of the parameter (e.g. $A_i = 0.01$), 1000 instances were generated.

Before we start the analysis of the properties of the solutions, let us point out an important difference in the schedule structure between the divisible load scheduling problem with and without memory limitations. It has been shown in [48] that if there are no memory limitations, then in the optimum solution of the problem there are no idle times in computations and in communications. We prove below that it is not the case when memory limits are present.

Proposition 3.1. *The optimum solution of an instance of the divisible load scheduling problem with limited memory may contain idle times in the computations and in the communication.*

Proof. Suppose $m = 1$, $A_1 = 1, B_1 = \frac{V}{2}, C_1 = 0, S_1 = M$, where $M \gg V$ is a big constant. The minimum number of communications is $n_{MIN} = \frac{V}{B_1} = 2$, for which the schedule length is $T_{max} = 2M + V$. There is an idle interval of length $\frac{V}{2}$ in the communications, and an idle interval of length M in the computations. Idle times in the computations cannot be closed because any load which fits in memory size $B_1 = \frac{V}{2}$ is processed in shorter time than the startup time $S = M \gg V$. Suppose that we want to close the idle interval in the communications by sending messages shorter than $\frac{V}{2}$. However, in this case at least three chunks would have to be sent from the originator. Then schedule length would be at least $3M$. Since M can be arbitrarily big in relation to V, the difference between the length of a schedule with idle times and the length of the schedule without idle times in communication can be arbitrarily big in absolute terms. □

3.4.1 Depth of Overlap

The depth of overlap, defined by numbers δ_{ij}, shows how many chunks interfere with each other. The existence of non-zero overlaps means that the processor must accumulate the load to be processed. It is of practical importance to verify if the accumulation of the load is actually necessary, and to what degree.

Let us start with single processor considerations. When $m = 1$, the scheduling problem may seem simple, but it is not trivial, since to construct a schedule one has to decide on the overlap and the sizes of load chunks. It is also of practical importance because it indicates how a very powerful server should cooperate with each of the worker computers.

We will be saying that solutions for which chunks overlap by not more than 1, i.e. $\forall i, j, \delta_{ij} \leq 1$, have overlap at most 1. If $\forall i, j < n_i, \delta_{ij} = 1$, then we will be saying that a solution has overlap 1. Let us analyze a specific overlap configuration. Assume that a schedule has overlap 1 ($\forall j < n_1, \delta_{1j} = 1$), so that the chunks overlap with their direct predecessor and direct successor (if any). If chunk 1 has size α_1, then by (3.19) chunk 2 has size at most $\alpha_2 \leq B_1 - \alpha_1$, chunk 3 has size at most $\alpha_3 \leq B_1 - \alpha_2$, etc. Thus, if all pairs of chunks have their maximum sizes, then the sizes of all chunks are in fact determined by a single variable α_1. The size of processed load is $\frac{n}{2}B_1$ if communication sequence has even number n of messages, or it is $\frac{n-1}{2}B_1 + \alpha_1$ if n is odd. Hence, it is possible to construct such a schedule if the number of messages is at least $n = \lceil \frac{2V}{B_1} \rceil$. We will say that solutions for $m = 1$ with $n = \lceil \frac{2V}{B_1} \rceil$ and overlap 1 are *coupled*, because consecutive chunks create couples coexisting in memory.

Proposition 3.2. *The coupled solutions are not arbitrarily bad.*

Proof. For the optimum solution we have $T^*_{max} \geq \lceil \frac{V}{B_1} \rceil S_1 + C_1 V = n_{MIN} S_1 + C_1 V$ and $T^*_{max} \geq A_1 V$. For a coupled solution, $T_{max} \leq \lceil \frac{2V}{B_1} \rceil S_1 + C_1 V + A_1 V = n_{CPL} S_1 + C_1 V + A_1 V$ and $n_{CPL} \leq 2 n_{MIN}$.

1. If $A_1 V \leq n_{MIN} S_1 + C_1 V$ then

$$\frac{T_{max}}{T^*_{max}} \leq \frac{n_{CPL} S_1 + C_1 V + A_1 V}{n_{MIN} S_1 + C_1 V} \leq \frac{3 n_{MIN} S_1 + 2 C_1 V}{n_{MIN} S_1 + C_1 V} \leq 3.$$

2. If $n_{MIN} S_1 + C_1 V \leq A_1 V$ then

$$\frac{T_{max}}{T^*_{max}} \leq \frac{n_{CPL} S_1 + C_1 V + A_1 V}{A_1 V} \leq \frac{2(n_{MIN} S_1 + C_1 V) + A_1 V}{A_1 V} \leq 3.$$

□

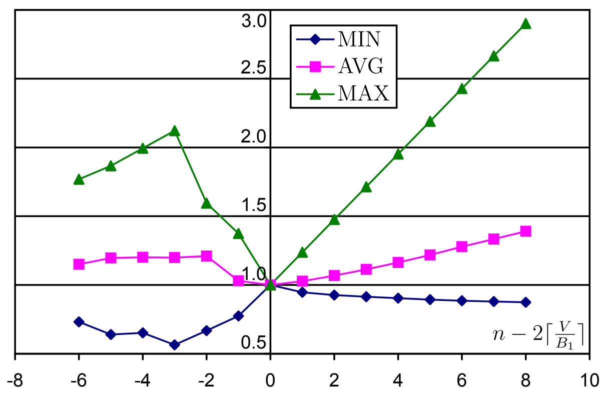


Figure 3.5: $m = 1$, quality of the solutions with various communication sequence lengths and the best overlap, relative to coupled solutions.

The above proposition gives an indication on the quality of coupled schedules in the worst case. The average quality of such solutions was tested experimentally. In Fig. 3.5 the quality of schedules for $m = 1$, various sequence lengths, and the best overlap chosen by B&B algorithm is shown. The coupled solution quality is used as a reference, and is represented by the point at the coordinates $(0, 1)$. Solutions with forced shorter sequences are shown on the negative part of horizontal axis and solutions with forced longer sequences on the positive part. The best, the worst, and the average distance from the coupled solution is shown. The results in Fig. 3.5 represent 888 randomly generated instances with $A_1, C_1, S_1 \sim U[0, 1]$, $B_1 \sim U(0, 10), V = 10$. It can be seen that typically the best solutions are not very much better than the coupled ones. Increasing n beyond $\lceil \frac{2V}{B_1} \rceil$ is not reducing schedule length by more than approximately 13%. Thus, on average coupled solutions provide a simple and efficient method of solving the combinatorial part of our problem on a single processor. Let us note that the optimum communication sequence length n may be smaller or greater than $\lceil \frac{2V}{B_1} \rceil$ depending on the instance.

Now we will move to analyzing the overlap for larger numbers of processors. Let us start with an observation that arbitrarily deep overlaps may be necessary.

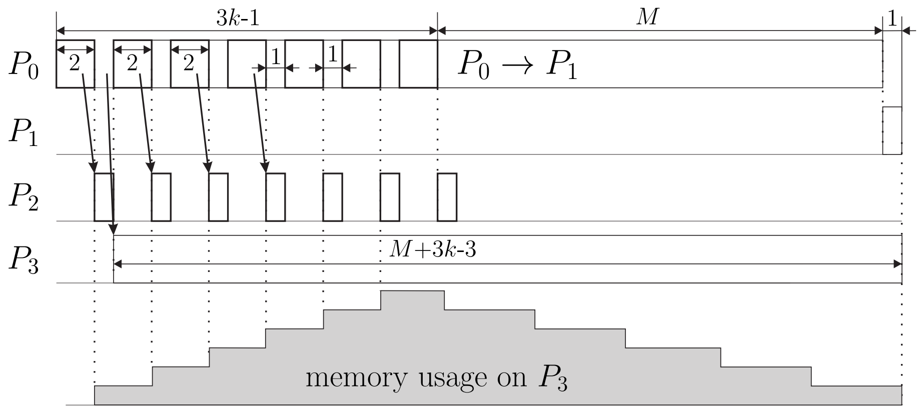


Figure 3.6: An instance with arbitrarily big overlap in Theorem 3.3.

Theorem 3.3. *There exist instances whose optimum solutions contain arbitrarily big overlap.*

Proof. Let k, M be two integers, where $k > 5$ is even, and $2^k - 3k - 1 > M > 3k + 1$. Consider the following example: $m = 3, V = 2^{2k} + k2^k + 1$,

$A_1 = \frac{1}{2^{2k}}, B_1 = 2^{2k}, C_1 = 0, S_1 = M$,

$A_2 = \frac{1}{2^k}, B_2 = 2^k, C_2 = 0, S_2 = 2$,

$A_3 = M + 3k - 3, B_3 = V, C_3 = k - 1, S_3 = 0$.

We want to build a schedule of length $T = M + 3k$. We will show that no shorter schedule may exist.

To process V, one activation of P_1 is necessary. Processors P_2 and P_3 cannot process the load V in time T. Indeed, note that in time T processor P_3 is capable of processing at most $\frac{T}{A_3} = \frac{M+3k}{M+3k-3} = 1 + \frac{3}{M+3k-3} < 2$ units of load. Hence, if processor P_1 was not activated, then P_2 would have to process more than 2^{2k} units of load. This would require sending more than $\frac{2^{2k}}{B_2} = 2^k$ messages to P_2, and would take time longer than $2^k S_2 = 2^{k+1} > T$. Thus, processor P_1 must take part in computations. Moreover, P_1 cannot be activated more than once because $2S_1 = 2M > T$.

Now we will prove that processor P_2 must receive k messages, processing of which is not overlapping.

Consider the minimum load $V - B_1 = k2^k + 1$ remaining to be processed by

P_2, P_3. As we noted above, in time T processor P_3 is capable of processing less than 2 units of load. Thus, to process the remaining load, P_2 must receive at least k messages. If in the k messages each one carries load B_2, then the whole communication to P_2 and computation on P_2 can be feasibly performed in time $3k$ as shown in Fig. 3.6. Note that if k chunks are sent to P_2, then none of them may overlap. Were it otherwise, the maximum load which could be sent to P_2 would be $(k-1)B_2$, and P_3 would have to process load of size at least 2^k+1, what is impossible in time T.

On the other hand, assume that P_2 receives at least $k+1$ messages. Then, the time of communication with P_1 and P_2 is at least $M + S_2(k+1) = M + 2k + 2$. There remain $k-2$ time units for communication with P_3. The maximum amount of load which can be sent to P_3 in this time is $\frac{k-2}{k-1}$. Hence, P_2 must receive load of size at least $k2^k + 1 - \frac{k-2}{k-1} = k2^k + \frac{1}{k-1}$.

Consider the overlapping of chunks sent to P_2. A full buffer B_2 of data on P_2 is processed in time $A_2B_2 = 1 < S_2$. Hence, processing of each chunk is finished before receiving the next chunk is completed. This means that the maximum possible overlap on P_2 is 1. We will divide the set of all chunks sent to P_2 in the following way. Let the first load chunk i overlap with the next δ_i chunks ($\delta_i \in \{0,1\}$). To obey memory limits, the group of $\delta_i + 1$ consecutive chunks $i, \dots, i+\delta_i$ may contain load of size at most B_2. The next group of chunks starting with chunk $i+\delta_i+1$, and containing $\delta_{i+\delta_i+1}+1$ chunks, is independent, in the sense that they may carry another volume of size at most B_2. Thus, the set of messages sent to P_2 can be divided into groups, each of which contains 1 or 2 chunks and carries load of size at most B_2. Let l_0 be the number of groups with overlap 0 (single, non-overlapping chunks) and l_1 be the number of groups with overlap 1 (pairs of overlapping chunks). The total number of messages sent to P_2 is $l_0 + 2l_1 \geq k+1$. The number of groups is $l_0 + l_1 \geq k+1$, because the load sent to P_2 is greater than kB_2. Let V_0 denote the total amount of load contained in the groups with overlap 0. As the maximum load which can be contained in the groups with overlap 1 is $l_1 B_2 = 2^k l_1$, we have $V_0 \geq (k - l_1)2^k + \frac{1}{k-1}$.

Consider the minimum time of communication and computations on P_2 and communication with P_1. Sending a chunk of data to P_2 takes time S_2. The computations of load contained in groups with overlap 0 sent to P_2 are not overlapped by communications with P_2 and they are executed in total time $A_2 V_0$. Computations of at most B_2 load units can be performed in parallel with startup S_1 on P_1. Hence, computation and communications of P_2 together with the startup time S_1 take at least time

$S_2(l_0 + 2l_1) + A_2 V_0 + S_1 - A_2 B_2 =$

$2l_0 + 4l_1 + \frac{V_0}{2^k} + M - 1 \geq$

$2l_0 + 4l_1 + k - l_1 + \frac{1}{2^k(k-1)} + M - 1 =$

$M + k + (l_0 + l_1) + (l_0 + 2l_1) + \frac{1}{2^k(k-1)} - 1 \geq$

$M + k + (k+1) + (k+1) + \frac{1}{2^k(k-1)} - 1 =$

$M + 3k + 1 + \frac{1}{2^k(k-1)} > M + 3k = T.$

We proved that it is infeasible to send more than k messages to P_2. Therefore, P_2 must receive exactly k messages, processing of which must not overlap.

There are at most $k+2$ free intervals in communications with P_1 and P_2, of total length k. We will show now that the length of each such interval must be smaller than 3. Indeed, consider the minimum load which must be processed by P_2, equal to $k2^k - \frac{3}{M+3k-3}$. As chunks sent to P_2 do not overlap, only one buffer of data may be processed in parallel with communications to P_1 and P_2, in time at most 1. The remaining load must be processed during the intervals with no communications to P_1 and P_2. Thus, the maximum time available for processing on P_2 is $k+1$. In each interval with no communications to P_2 at most one buffer of data can be processed. Hence, if the length of any such interval is at least 3, then the time which can be used for processing the load on P_2 decreases to $k-1$, which is insufficient to process load of size $k2^k - \frac{3}{M+3k-3}$. Hence, the length of

Table 3.1: Relative frequency of the overlaps in all chunks.

overlap	0	1	2	> 2
frequency	0.835	0.154	0.010	0.001

each interval which can be used for communications with P_3 is smaller than 3.

Since P_1 receives one message and P_2 receives k messages, at most $2^{2k} + k2^k$ units of load are processed on P_1 and P_2. Processor P_3 must compute the remaining amount of at least $V - B_1 - kB_2 = 1$ unit of load, what takes time at least $M+3k-3$. Thus, there may be at most 3 idle time units in processing on P_3, which means that in parallel with startup time S_1 processor P_3 must compute at least load of size $\frac{M-3}{M+3k-3}$. As each chunk of data must be sent to P_3 in an interval of size smaller than 3, the size of a chunk received by P_3 is at most $\frac{3}{k-1}$. Hence, the number of messages waiting to be processed on P_3 when communication to P_1 starts must be at least $\frac{(k-1)(M-3)}{3(M+3k-3)}$, which tends to $\frac{k-1}{3}$ as M tends to infinity. We conclude that it is possible to construct an instance whose optimum schedule requires arbitrarily deep overlap.

$\square$

Although arbitrarily deep overlap is possible in the worst case, the experimental results show that it is not common in practice. The data were gathered from the solutions delivered by GA for 19953 randomly generated instances with $A_i, B_i, C_i, S_i \sim U[0, 1]$, $m \sim U[1, 10]$ and $V \in \{2, 5, 10, 20, 50\}$. The depth of the overlap of all chunks in all sequences of the solutions generated by GA for the above instances is presented in Table 3.1.

A more detailed view of the chunk overlaps is shown in Fig. 3.7. The vertical axis is the relative frequency of instances with a certain fraction of chunks with a certain overlap. For example, (see the rightmost box "1" for overlap $O = 0$), approximately 36% of all instances have only chunks with overlap 0. The absence of a point in box "0" for overlap 0 means that there were no instances without

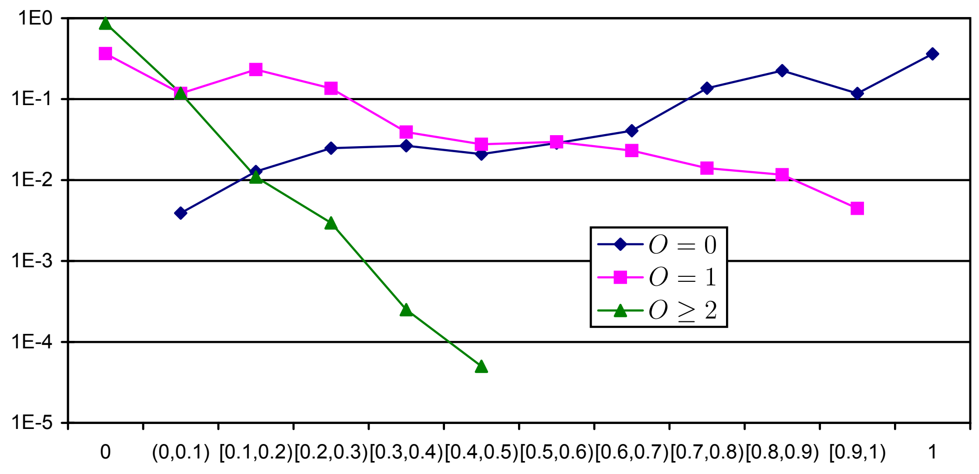


Figure 3.7: Histogram of overlap frequencies in the instances.

a chunk with overlap 0. The number of solutions for which the chunks with overlap 1 are 90% to 99.99% of all the chunks in the communication sequence is approximately 0.4% of all instances (box "[0.9,1)" for overlap $O = 1$). On the other hand, approximately 36% of all instances have no chunk with overlap 1. Finally, overlaps 2 and bigger are very rare: approximately 87% solutions have no chunk with overlap 2 or greater, and only 0.005% instances have solutions with overlap at least 2 in more than 40% of all the chunks in the solution. Thus, it can be seen that overlap deeper than 1 is rare, because it constitutes approximately 1% of all chunks in all solutions.

The analysis of the depth of the overlap leads to the following conclusions. On a single processor the solutions with $n = \lceil \frac{2V}{B} \rceil$ and overlap 1 are good on average. For multiple processors ($m > 1$) the overlap may be arbitrarily deep in general. Still, overlaps greater than 1 are rare in practice.

3.4.2 Length of the Communication Sequence

The number of communications n is a very important characteristic of the solution, because it is a key determinant of the complexity of the algorithms solving the problem. The minimum required length of the communication sequence depends on V and B_is. Therefore, it seems reasonable to use this minimum number $n_{MIN} = \lceil \frac{V}{B_{max}} \rceil$ of communications as a reference. We start with an observation for the case of $m = 1$.

Proposition 3.4. *For $m = 1$ the schedule for sequence of length n_{MIN} can be at most twice as long as the optimum schedule.*

Proof. Schedule length T_{max} for a sequence with the smallest possible length n_{MIN} is not greater than $n_{MIN}S_1 + C_1V + A_1V$. On the other hand, for the optimum solution, $T^*_{max} \geq n_{MIN}S_1 + C_1V$ and $T^*_{max} \geq A_1V$.

1. If $A_1V \leq n_{MIN}S_1 + C_1V$ then

$$\frac{T_{max}}{T^*_{max}} \leq \frac{n_{MIN}S_1 + C_1V + A_1V}{n_{MIN}S_1 + C_1V} \leq 2.$$

2. If $A_1V \geq n_{MIN}S_1 + C_1V$ then

$$\frac{T_{max}}{T^*_{max}} \leq \frac{n_{MIN}S_1 + C_1V + A_1V}{A_1V} \leq 2.$$

Let us also note that for a schedule with the number of communications not greater than $n_{MIN}k$, we have $T_{max} \leq n_{MIN}kS_1 + C_1V + A_1V$. Hence, considering the two above cases it can be proved that $\frac{T_{max}}{T^*_{max}} \leq k + 1$. □

We show below that the above result cannot be transferred to the case $m > 1$.

Proposition 3.5. *A communication sequence with the minimum possible number of chunks n_{MIN} can be arbitrarily bad for the schedule length. The length of the optimum communication sequence can be arbitrarily big in relation to n_{MIN}.*

Proof. Consider an example: $m = 2, A_1 = 1, B_1 = 1, C_1 = 0, S_1 = 1, A_2 = \frac{1}{V}, B_2 = V, C_2 = 0, S_2 = M$, where M is a big constant. The minimum number of communications is $n_{MIN} = 1$, and it results in a schedule of length $M + 1$. On the other hand if P_1 is used only and no chunks overlap, then the schedule length is $\lceil V \rceil + V$, and the number of communications is $n = \lceil V \rceil$. The ratio of the two schedule lengths is $\frac{M+1}{\lceil V \rceil + V}$, which can be made arbitrarily big by selection of M and V.

For the second part of the proposition, note that the number of messages in the optimum communication sequence proposed in the previous part of the proof can be arbitrarily big. $\square$

Let us now analyze the length of communication sequences generated by GA. The values of the relative communication sequence lengths $\frac{n}{n_{MIN}}$ are shown in Fig. 3.8. Each of the charts shows the average (AVG) and the largest (MAX) relative communication sequence lengths. In Fig. 3.8a the communication lengths are shown for various A values. It can be seen that usually $\frac{n}{n_{MIN}}$ is not very big. On average $n \approx 1.39 n_{MIN}$, which is calculated over all instances with changing A. The length of the sequence grows with A, what is most evident for the largest registered relative lengths. This phenomenon can be attributed to the way of calculating n_{MIN}. For example, for $V = 2$ and $B_i \in (0, 1]$, the expected n_{MIN} is 4, and in extreme cases it can be just $n_{MIN} = 2$. On the other hand, as processors get slower (A is increasing), it is more and more profitable to use all m available processors. Thus, $\frac{n}{n_{MIN}}$ grows with A. This increase is stronger for small V, and weaker for bigger V.

In Fig. 3.8b a similar dependence is shown for changing $\frac{B}{V}$. The length of the communication sequence quickly increases with $\frac{B}{V}$. This can be explained by the following two facts. On the one hand, for $\frac{B}{V}$ approaching 1, n_{MIN} is also approaching 1, but as in Proposition 3.5, other parameters of the system make it profitable to build sequences with $n \gg 1$. On the other hand, as $\frac{B}{V}$ approaches 0, more short communications must be made to send the load off the originator. Each message carries cost of some startup S_i. Therefore, communication startup costs dominate in the schedule length. To minimize this cost, it is advantageous to

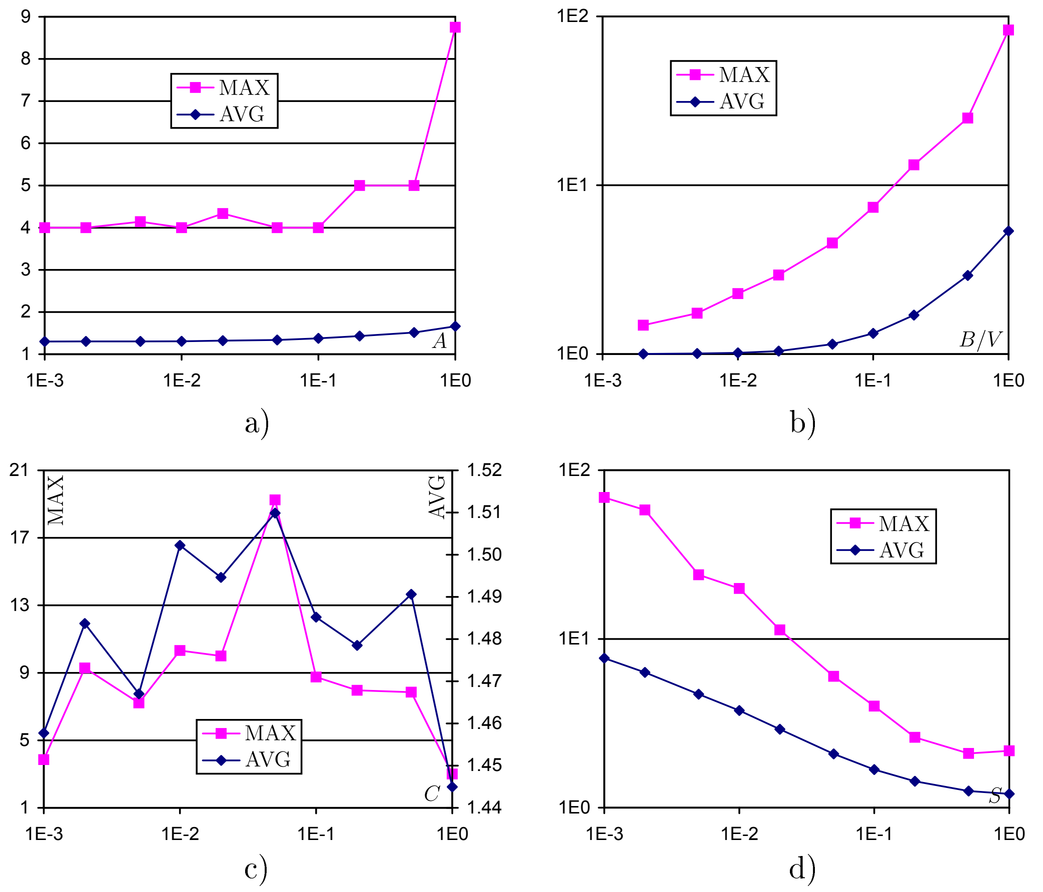


Figure 3.8: Relative sequence length $\frac{n}{n_{MIN}}$ in the solutions of GA, a) vs. A , b) vs. $\frac{B}{V}$, c) vs. C, d) vs. S.

send as few messages as possible. Hence, n tends to n_{MIN} when $\frac{B}{V}$ is decreasing. Similar observations can be made for big values of S (cf. Fig. 3.8d). For big S it is profitable to send as few messages as possible. This, in turn, exposes the need for big communication buffers. The behavior of $\frac{n}{n_{MIN}}$ for small S must be contrasted with Fig. 3.8a. When $S \approx \frac{1}{2}$ on average, as in Fig. 3.8a, then $\frac{n}{n_{MIN}} \approx 1.39$. If $S = 0.001$, as in Fig. 3.8d, then $\frac{n}{n_{MIN}} \approx 8$. This means that big startup time is a considerable disincentive to building long communication sequences.

In Fig. 3.8c the dependence of $\frac{n}{n_{MIN}}$ on C is shown. Note that this figure has two vertical axes. The shapes of MAX and AVG are similar, but for the average case the changes are in the range of approximately 5%. This should be surprising because multi-installment divisible load processing was introduced to reduce the time of initial waiting for load. Growing value of C should be an incentive to build shorter messages and longer communication sequences. This tendency can be seen only for small values of C. Yet, in our setting of the experiments the expected value of the startup times is $\frac{1}{2}$. This is a disincentive to build long communication sequences as explained on the example of Fig. 3.8a and Fig. 3.8d. Hence, the dependence of average $\frac{n}{n_{MIN}}$ on C is very weak. Moreover, with growing C the algorithm tends to compensate increasing communication costs by sending fewer messages. Thus, initial waiting for the load is meaningless compared to the whole communication cost.

From the above analysis of the communication sequence length we draw the following conclusions. Startup times S_i are an important element of communication time and they constitute the main disincentive to building long communication sequences. For startup times of the same order as communication time per unit of load (C), or computation time per unit of load (A), communication sequences have lengths about $1.4n_{MIN}$. For small S the sequences can be approximately 8-10 times longer than n_{MIN} on average. Moreover, S_i and B_i are in a sense coupled in determining the system performance: small B_is expose costs of communication startups, big S_is expose the need for processors with big communication buffers.

3.4.3 Number of Used Processors

In this section we study the number m' of processors from the set $\{P_1, \ldots, P_m\}$ which take part in the computations. This characteristic of a solution is of practical importance. In contemporary grid and cluster systems very large numbers of processors are available. It is necessary to know how many of them should be used and how to adjust their number for different applications. Is is easy to construct biased instances, for which only one processor should be used (e.g. because all other processors have very large startup times S_i), or for which all processors should be used. It is known [3] that if there are no startup times

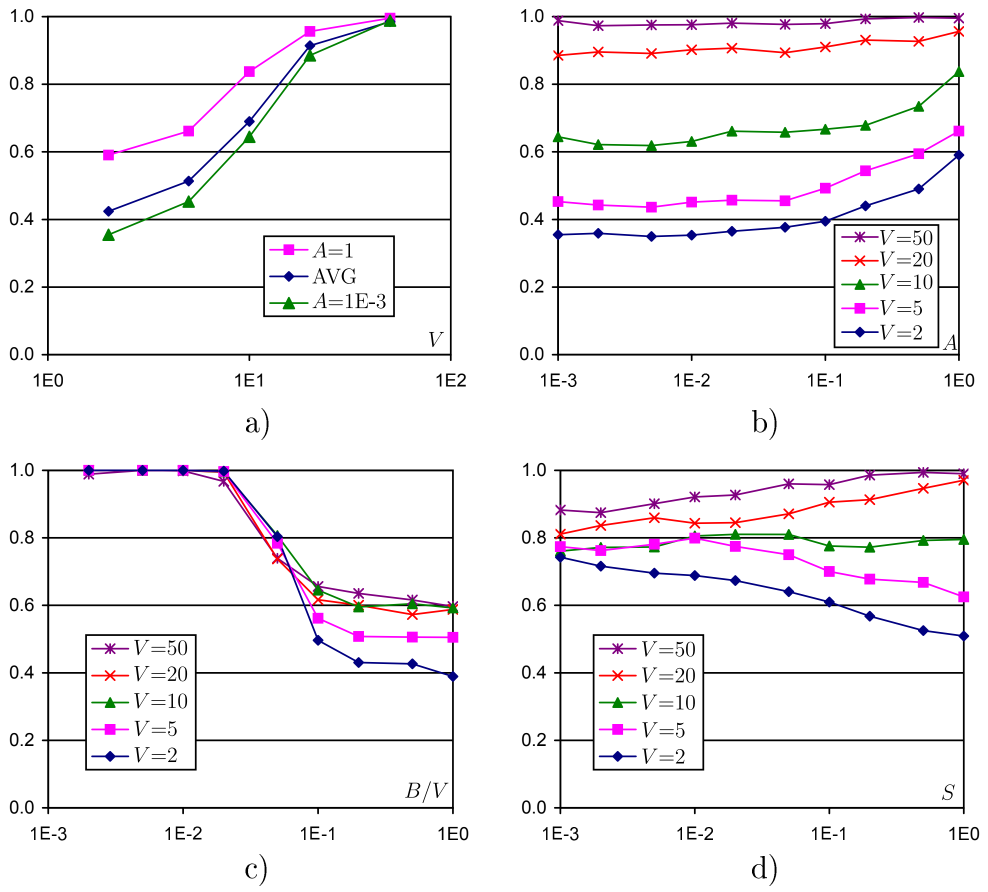


Figure 3.9: Relative number $\frac{m'}{m}$ of different used processors in the solutions of GA, a) vs. V, b) vs. A c) vs. $\frac{B}{V}$, d) vs. S.

($S_i = 0$ for all i), then computations can be started on any number of processors. On the other hand, if communication startup times are present, then in single-installment processing using all processors is a matter of sufficiently large volume of load V [3]. Hence, it may be expected that the number of used processors in multi-installment processing should grow with decreasing startup times and increasing volume of the load.

The relations between the relative number $\frac{m'}{m}$ of used processors and the values of selected parameters in the solutions found by GA are presented in Fig. 3.9. Fig. 3.9a shows that with growing amount of load V the number of different used processors is increasing, as could be intuitively expected. This result was confirmed in all the experiments we performed. This has a practical consequence, that for larger problems it is profitable to use more processors (even not very effective) instead of sending bigger number of load chunks only to a smaller set of more effective processors.

The dependence of $\frac{m'}{m}$ on A is shown in Fig. 3.9b. It can be seen that m' increases with A only for small problem sizes (small V). For small V only a few chunks need to be sent. Therefore, for small A the algorithm minimizes the schedule length by selecting only a few processors with big memory buffers and fast communication links. If A as big, then computing time dominates in the schedule length and it is profitable to distribute and parallelize computations. Hence $\frac{m'}{m}$ is growing in this case. For big V the number of chunks must be big independently of the value of A, communication time (mainly startup times S_i) is dominating over computation time, and A is less important in determining the schedule length. Therefore, A does not influence m' for big V.

In Fig. 3.9c we present the dependence of $\frac{m'}{m}$ on $\frac{B}{V}$. In our method of test instance generation the average number of processors is close to 5. Hence, for $\frac{B}{V} < \frac{1}{5}$ the memory space necessary to process load V is created by using many load chunks, and many processors working in parallel. On the other hand, when $\frac{B}{V} > \frac{1}{5}$, the size of the memory is often sufficient to process the whole load in just one installment. Therefore, good solutions typically use only a few processors with fast communication and computation.

Fig. 3.9d shows the relation between S, V and $\frac{m'}{m}$. With growing amount of load V the number of different used processors is increasing as in previously described experiments. For small V the number of different used processors decreases with S, which is in accord with our earlier expectations. However, for big V the increasing S results in increased $\frac{m'}{m}$. This counterintuitive behavior can be partially explained by the way of generating test instances. Note that startup times of all processors are equal in the experiments depicted in Fig. 3.9d. When V is big, then the number of sent chunks must also be big. With growing S, startup times dominate in the schedule length and other parameters, by which the processors differ, become meaningless. Therefore, GA becomes myopic to the differences in processor parameters, and hence more processors are drawn to the solutions.

The dependence of $\frac{m'}{m}$ on C (not shown here) is very weak. This is a very surprising situation because in many DLT papers the communication rate C was considered crucial for the system performance. Only for small V and big C (close to 1) is the number of used processors slightly decreasing with growing C. This is a consequence of the startup time domination in the communication time. Only for small V the number of messages is small and hence the total startup cost is small. Then, GA optimizes the schedule by using a small number of efficient processors. This result does not eliminate C as an important schedule structure determinant, as will be shown in the following sections.

We finish this section with the following conclusions. The number of different used processors differs depending on the settings. In general it is increasing with V. In our experiment setting startup times dominated the schedule length, especially when the number of chunks had to be big because V was big or B was small. When A is big and the computation time is at least comparable with the communication time, then it is profitable to use many processors to parallelize computations. When C is big and its contribution to the communication time is comparable or greater than the contribution of the startup times, then it is profitable to choose only a small number of fast processors.

3.4.4 Dominating Set of Processors

In the previous section we considered only the number of processors which receive any load, not the degree of their participation in the computations. Here we analyze the distribution of the load between the processors. Our goal is to determine if there is any inequality in the load distribution, and if this is the case, then what kind of processors dominate in the computations.

The first tool we applied in analyzing inequality in the load distribution is the Gini index [33]. It is an indicator of some parameter deviation from the uniform distribution and is commonly used in economics to quantify inequality in wealth distribution. The closer the Gini index is to 0, the more equal and uniform the distribution of the load is. The closer the Gini index is to 1, the more unequal the distribution of the parameter is. The value of the Gini index for a set of values $\{y_i : i = 1, \dots, n\}$ can be calculated from the formula

$$G = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} |y_i - y_j|}{2n \sum_{i=1}^{n} y_i}. \tag{3.22}$$

We calculated the Gini indices for the amount of the load received by the processors (which we will denote GiL), and for the number of received messages (which we will refer to as $Gi\#$). For example, $GiL = 1$ implies that the whole load V is processed by a single processor, and $GiL = 0$ means that each processor receives the same amount of load. Selected results are presented in Fig. 3.10. The general observation is that GiL and $Gi\#$ demonstrate the same tendencies.

It can be seen in Fig. 3.10a that GiL is decreasing with increasing V, which means that with growing size of the load its distribution becomes more equal. This situation has been observed in all experiments. The dependence of GiL on A is shown in Fig. 3.10b. Only for small V does A influence the load distribution. For small V the number of used processors is small and it is profitable to select the best of them, while for big V the number of load chunks must be big anyway, which means that the communication time is long and the computation time (hence A) has a little influence on the schedule length. Consequently, for big V the values of GiL do not depend on A. This situation is similar to Fig. 3.9b depicting $\frac{m'}{m}$ vs. A. A strong change of GiL with $\frac{B}{V}$ is observed in Fig. 3.10c when $\frac{B}{V} \approx \frac{1}{5}$. For smaller values of $\frac{B}{V}$ the load distribution is more equal, for bigger $\frac{B}{V}$ the load distribution is more unequal. This is caused by the fact that

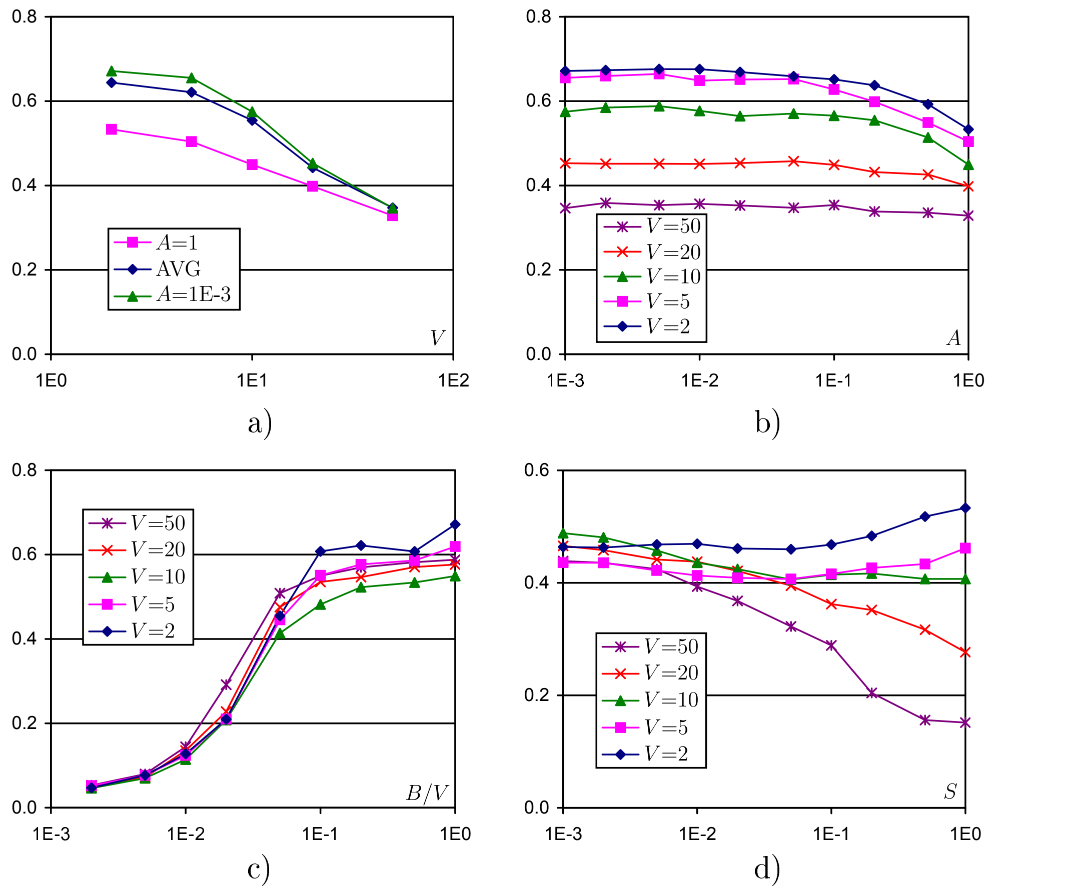


Figure 3.10: Gini index of the GA solutions, a) GiL vs. V, b) GiL vs. A, c) GiL vs. $\frac{B}{V}$, d) $Gi\#$ vs. S.

for $\frac{B}{V} > \frac{1}{5}$ only one installment is sufficient to process the whole load. These results conform with the results depicted in Fig. 3.9c. In Fig. 3.10d $Gi\#$ is shown for changing S. Again, similarly to Fig. 3.9d, with growing S the diversity of used processor sets depends on V. For small problem sizes it is profitable to use fewer processors, hence $Gi\#$ is big, what signifies inequality. For big V the number of used processors is big, communication startup times dominate in the schedule length, and the algorithm does not distinguish processors with different parameters well, hence more of them are included in the communication sequence, and the messages are distributed more equally.

Unfortunately, the Gini index is hard to interpret. For example, it is hard to say if a certain value of GiL, $Gi\#$ already represents inequality or not. Only general tendencies of changing inequality can be observed. Here, the tendencies of Gini index only confirm the analysis of the number of used processors. Moreover, one cannot determine, using GiL, $Gi\#$, what processors dominate in the load distribution (if any). Therefore, we applied one more indicator of the load distribution inequality.

The second measure of processor domination in the computations is based on the analysis of the sets of processors receiving the largest amount of load. Let V_{max} be the greatest total load received by any processor. We call a set of processors *load frequent* if it includes all processors which receive at least $\frac{V_{max}}{2}$ units of load. The processors in the load frequent set are called load frequent, or just frequent.

We want to examine how much load and how many messages are sent to the frequent processor set. The results of this study are shown in Fig. 3.11. All values presented in this figure are relative: processor numbers are shown with respect to m, and the loads are shown relative to V. In Fig. 3.11c,d, the horizontal axes represent all parameters $A, \frac{B}{V}, C, S$, in range $[0, 1]$ for four different relations. A general observation is that the functions of the number of load frequent processors in A (Fig. 3.11a), and in $\frac{B}{V}, C, S$ (not shown here) have very similar tendencies as the functions of $\frac{m'}{m}$ in the above parameters (see Fig. 3.9). However, the range of changes of the number of frequent processors vs. V is narrower than the range of changes in $\frac{m'}{m}$. For example, in Fig. 3.9a the number of used processors changes in range approximately [0.4,1]. Here, the range of changes is approximately [0.3,0.5] (cf. Fig. 3.11b). In the experiments with changing $\frac{B}{V}, C, S$ even smaller ranges were observed. It can be concluded that the size of the frequent set of processors is growing with V, but not as quickly as the number of different used processors m'. This is because only a selected set of processors is frequently used while many other processors get to the solution due to the randomized selection.

In Fig. 3.11c the load of the processor receiving the greatest amount of data is

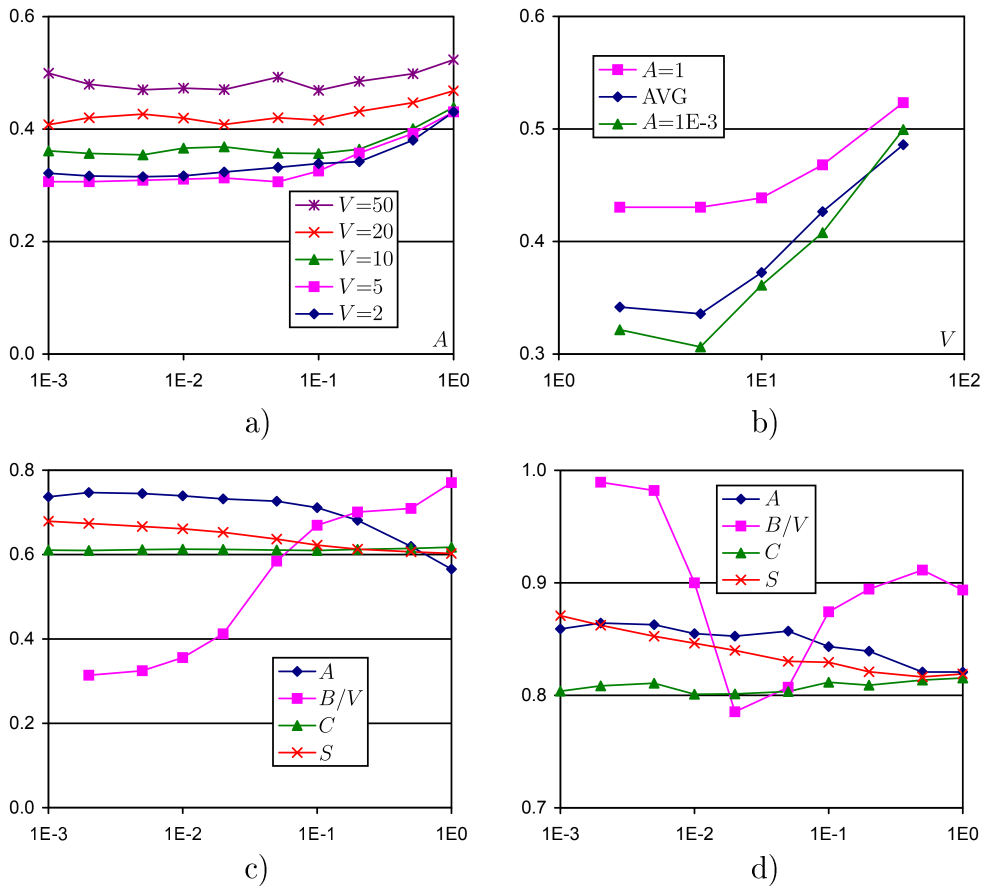


Figure 3.11: Load frequent processor sets in GA solutions. a) Number of frequent processors vs. A, b) number of frequent processors vs. V, c) load of the most loaded processor and d) load of all the frequent processors vs. $A, \frac{B}{V}, C, S$.

depicted vs. changing $A, \frac{B}{V}, C, S$. Independently of the type of changes, the most loaded processor receives $0.6V$-$0.75V$ on average. With growing A computation time starts dominating in the schedule length, the processor selection method tends to build more computing power, and more processors are appended to the frequent set. Hence, the size of the greatest part of load sent to a single processor is decreasing. Growing $\frac{B}{V}$ allows for using fewer processors and for economizing on the communication time by sending smaller number of larger pieces of data. Hence, for big $\frac{B}{V}$ the most loaded processor receives load of size almost $0.75V$. For small $\frac{B}{V}$ a big number of communications must be made anyway, what exposes the cost of communication startup times dominating in the schedule length. Consequently, GA becomes myopic to other processor parameters, the frequent set has more processors, and the load is more dispersed between the processors. The dependence on S, shown in Fig. 3.11c, is very weak. However, this is an average over many sizes V. A more detailed picture exposes diversity with V similar to the one shown in Fig. 3.9d, though in much narrower range. Unlike in Fig. 3.9d, the load sizes are generally decreasing with increasing S, even for big loads V. Similarly to the results in Section 3.4.3, the size of the biggest part of the load received by a single processor does not depend on C.

The total amount of load assigned to all frequent processors is shown in Fig. 3.11d. It can be seen that the frequent processor set collects more than $0.8V$ on average. The function of the total load received vs. $\frac{B}{V}$ has a minimum. This unexpected phenomenon can be explained in the following way. For big values of $\frac{B}{V}$ only a few processors take part in the computation because a single installment is sufficient to process the whole load. Therefore, the number of messages is small, load chunks have sizes close to processor memory buffer sizes, the frequent set has small cardinality and receives almost the whole load. With decreasing $\frac{B}{V}$ more and more processors receive some load, and the contribution of the most loaded processors is decreasing as depicted in Fig. 3.11c. However, when $\frac{B}{V}$ becomes extremely small, the communication startup cost is dominating the schedule length, GA becomes unaware of processor parameters, and more of the processors are randomly included in the frequent set. Therefore, the cardinality of the frequent set is growing and also the total load in the frequent set is growing.

Similar results were obtained for the set of processors receiving the greatest number of messages (instead of the greatest amount of load). We finish the above considerations with a conclusion that the frequent set of processors really exists. With the exception of the instances biased by small $\frac{B}{V}$ or big S, when almost all processors are frequent, the frequent set contains approximately $40 - 50\%$ of all available processors. They receive 80-85% of the whole load, again with the

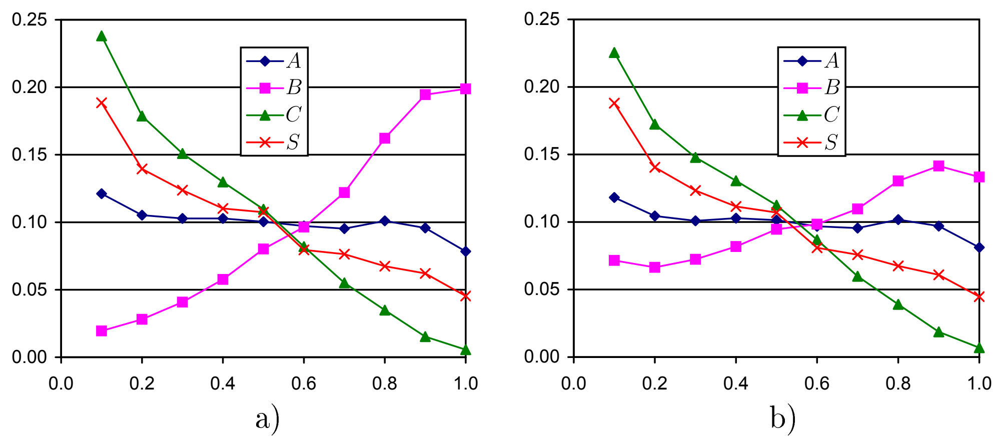


Figure 3.12: Received load and number of messages vs. processor rank in GA solutions. a) Load vs. rank, b) number of chunks vs. rank.

exception of the cases biased by small $\frac{B}{V}$ or big S.

The results in Fig. 3.10 and Fig. 3.11 confirm the existence of the sets of processors receiving more load, and hence dominating in the computations. Yet, in our test instances, when studying influence of a certain parameter, all processors had this parameter equal. We learned on the importance of the considered parameter via the consequences of its low or high values. However, the effects of the diversity of the given parameter were switched off. We did not verify how important a parameter could be if it had different values in the processor set. Therefore, another set of 1000 instances were generated with $V = 100$, m generated from $U[1, 100]$, and A_i, B_i, C_i, S_i generated from $U[0, 1]$. We examined the fraction of the whole load and the number of received messages against the rank of processors in the order of a certain parameter value. The results of this study are shown in Fig. 3.12.

In Fig. 3.12 the processors were grouped into sets comprising 10% of the processors ranked according to a certain parameter. For example, value 0.2 on the horizontal axis in Fig. 3.12 represents processors with relative rank $\frac{i}{m}$ in the range $(0.1, 0.2]$. The values on the vertical axes are relative: the size of the load is shown with respect to V and the number of received messages with respect to the total number of messages. The four functions depicted in Fig. 3.12 correspond to four different rankings: according to A, B, C, S. Let us remind that for A, C, S smaller values represent better performance, and for B bigger values are better. The relationships are similar for the received load (Fig. 3.12a) and for the number of messages (Fig. 3.12b). Therefore, we will discuss only the load distribution. The distribution of the load is tightly connected with all processor parameters. It is evident that processors which have best communication links with respect to C or S, or the biggest memory buffers receive more load to process. The processors with small B or big S, C receive almost no load. For parameter A the relationship is weaker but it is still noticeable (the coefficient of correlation between A and the upper limit of rank box interval is approximately -0.84).

We finish the study of the dominating set of processors with the following observations based on the computational experiments. The dominating processor set exists. The frequent processor set, as we defined it, comprises approximately 40-50% of all processors. In the biased case of big S or small $\frac{B}{V}$ the load is distributed almost equally and the frequent processor set may include nearly all processors. There is a strong correlation between processor parameters and the amount of load received for processing. This effect is strongest for parameter C and weakest for parameter A.

3.4.5 Chunk Size Saturation

The next element of the schedule structure we want to analyze are the sizes of load chunks. After determining the sequence of communications and the overlaps, a linear program was used to find the load distribution. Since the computational cost of linear programming may be considered high, it would be profitable to eliminate it in constructing good quality solutions. To examine the structure of the load partitioning we analyzed the number of chunks whose sizes are equal to the size of the target processor buffer, i.e. $\alpha_i = B_{\sigma(i)}$. We will call such chunks *full chunks*. It would be a very attractive solution to use just the processor buffer

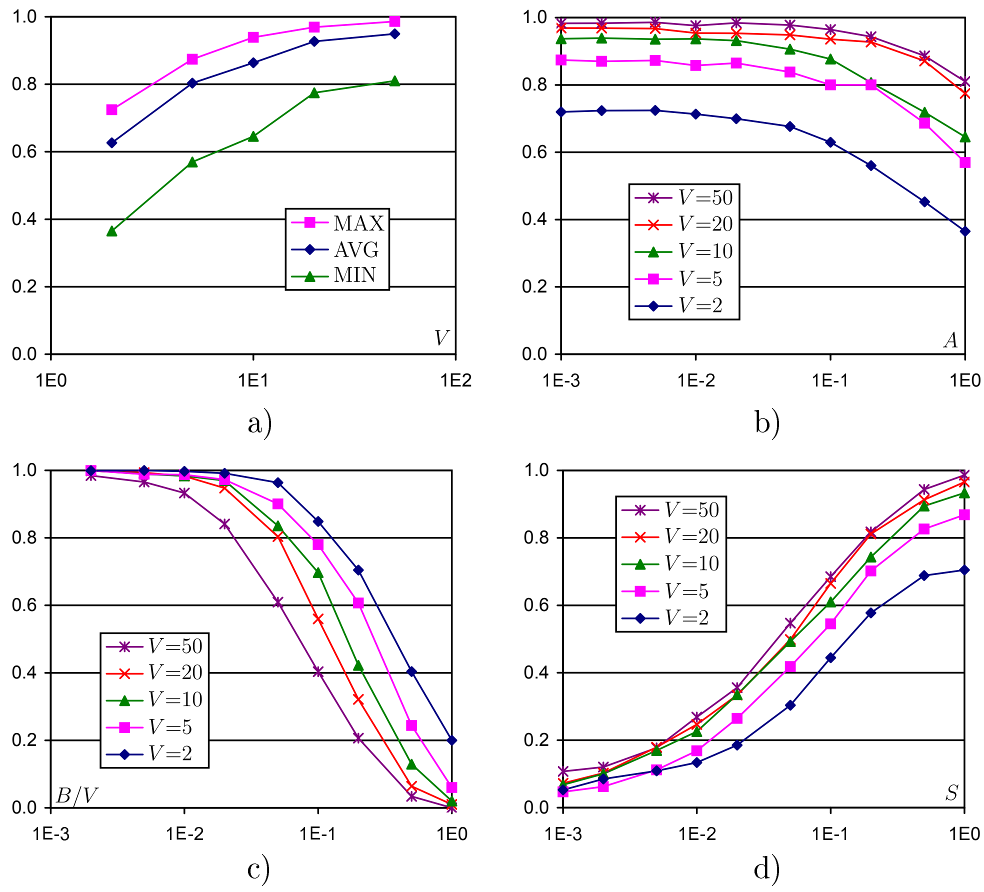


Figure 3.13: Average number of full chunks in GA solutions, a) vs. V in experiments with changing A, b) vs. A, c) vs. $\frac{B}{V}$, d) vs. S .

size as the chunk size, thus eliminating the need for linear programming. Still, such an approach eliminates the possibility of chunk overlapping. The results of the experiments on load chunk sizes are shown in Fig. 3.13.

In all the pictures shown in Fig. 3.13 the number of full chunks is shown in relation to the total number of chunks n. The number of full chunks is almost always high or noticeable, but not all chunks are full. It can be seen in Fig. 3.13a,b,d that with growing load size V the number of full chunks is also growing. This is intuitively reasonable because bigger load V requires more messages which expose the costs of startup times. These can be reduced by using as few messages as possible, and consequently filling the buffers more completely. This is also confirmed in Fig. 3.13c where the number of full chunks is shown against changing

$\frac{B}{V}$ and various values of V. When $\frac{B}{V}$ is small, then the number of messages must be big, hence the startup times dominate in the schedule length, and to reduce their contribution, the buffers are more fully filled. This situation is repeated in Fig. 3.13d where the number of full chunks increases with the startup times. With growing A (Fig. 3.13b) the number of full chunks is decreasing because the computation time, and not the startup times, increasingly dominates in the schedule length. Observe that in Fig. 3.13c the number of full chunks decreases with V, what may be attributed to the randomized nature of GA. When V is growing, but $\frac{B}{V}$ remains constant, the computation time and the part of communication time determined by parameter C dominate over the startup times. Thus, it is profitable to send more smaller messages in order to parallelize the computations in a greater degree. Hence, with growing V the fraction of chunks which are not full is also growing.

3.4.6 When Is It Hard to Find a Good Solution?

To summarize the analysis of the features of the obtained solutions, we study what makes an instance of our problem easy or hard to solve. Let us introduce the goal of this section in more detail. Heuristics build good quality solutions for many combinatorial optimization problems. However, this good performance may sometimes be attributed to the nature of the problem, not a heuristic. Thus, it is possible that our genetic algorithm builds good solutions not because it is well designed, but because in some cases our scheduling problem may be easy to solve. If we learn which instances are easy or hard to solve, then we will gain some new insights into the nature of the problem, and real merits of GA.

We have to decide how to verify which instances are easy, and which ones are hard to solve. We will compare the quality of the solutions obtained in three ways for various types of instances. The worst solution observed provides an indication on how bad a solution may be. The random solutions are not biased to being good or bad. GA solutions are optimized and supposed to be good. The three solution types indicate what can be achieved in the worst case, without great efforts (random solutions), and at considerable cost of optimization. If GA solutions did not differ much from the random solutions, then it would signify bad GA design. All the three types of solutions were obtained using the GA infrastructure. The random solution is the best one in the initial GA population of $G = 20$ solutions. The worst solution is the worst one observed in the course of solving given instance by GA. In all the three cases linear programming was used to obtain the best chunk sizes α_i and the schedule length for a given combinatorial part of the solution. The quality of the solutions is measured as the relative distance from the lower bound calculated in the following way. The minimum communication time is $\tau_1 = n_{MIN}S_{min} + VC_{min}$. In this time at most $V_0 = (\tau_1 - S_{min})\sum_{i=1}^{m}\frac{1}{A_i}$ load could be processed. The remaining load $V - V_0$ is processed in time at least equal to $\max\{0, \frac{V}{\sum_{i=1}^{m}1/A_i} - \tau_1 + S_{min}\}$. Thus the lower bound is equal to

$$LB = \tau_1 + \max\{0, \frac{V}{\sum_{i=1}^{m}1/A_i} - \tau_1 + S_{min}\}. \tag{3.23}$$

In Fig. 3.14 we show the influence of the system parameters on the quality of the above three solution types. The points on these charts represent average quality over the set of used test instances. Fig. 3.14a,b,c show the results for the first set of random instances and $V = 20$. It is striking that the worst case solutions (denoted $WRST$) can be over one order of magnitude further from the lower bound than the random solutions (denoted RND) or the solutions of the genetic algorithm (denoted GA). Moreover, GA solutions are substantially better than RND solutions, which means that GA really works. Now let us analyze the tendencies in Fig. 3.14a,b,c. As it can be seen in Fig. 3.14a, with growing C all the lines tend to 1. This means that as the communication speed decreases, the schedule length becomes dominated by the time of sending load off the originator. Hence, in such a biased case it is easier to obtain good solutions. Similar tendency

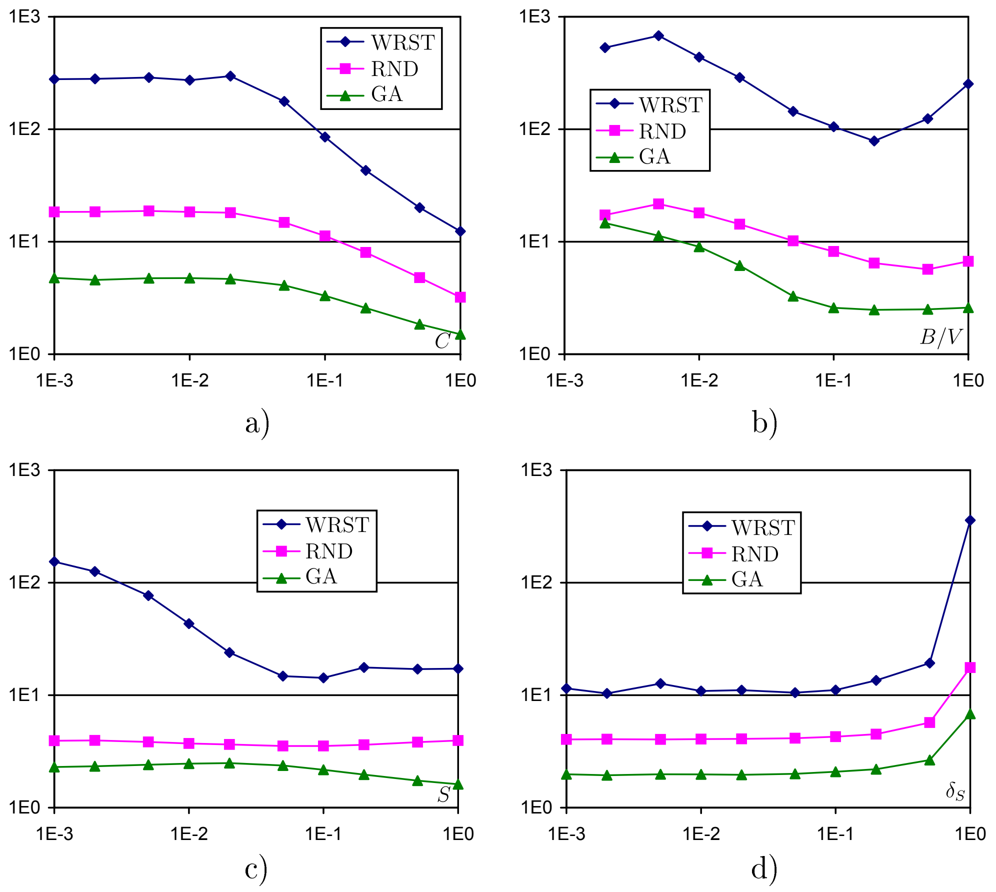


Figure 3.14: Quality of the solutions with reference to the lower bound for $V = 20$, a) vs. C, b) vs. $\frac{B}{V}$, c) vs. S, d) vs. the dispersion of S.

was observed for growing parameter A (not shown here).

In Fig. 3.14b the dependence of the solutions quality on changing $\frac{B}{V}$ is shown. With growing $\frac{B}{V}$ all the three types of solutions get closer to the lower bound. It is intuitively attractive to conclude that with growing $\frac{B}{V}$ the solutions are less dominated by choosing processors with small startups S_i, and good solutions are easier to obtain because we are less limited with the choice of the processor. Not disregarding this growing flexibility, it should not be forgotten that the construction of the lower bound (3.23) influences the results presented here. The lower bound is based on the assumption that the smallest S_i coincide with the biggest B_i, which is rarely true. Hence, for small $\frac{B}{V}$ and a big number of the startups the error resulting from this simplification may be significant. This may result in the big distance of the solutions from the lower bound. With increasing $\frac{B}{V}$ the domination of the startup costs in the schedule length decreases, the contribution of the transfer and the computation time increases, and the lower bound is representing this situation better. Thus, the results in Fig. 3.14b indeed confirm that with growing $\frac{B}{V}$ it is getting easier to obtain solutions closer to the lower bound, but it is achieved by using fewer messages and communication startup times. Moreover, for the biggest $\frac{B}{V}$ solutions $WRST, RND$ get slightly worse and GA solutions do not. This means that even if memory buffers are big, it is necessary to adjust the set of used processors. The genetic algorithm handles this better than in the RND solutions.

In Fig. 3.14c the dependence of the three types of solutions on changing parameter S is shown. A counterintuitive tendency of improving $WRST$ solution quality with growing S can be observed. With growing S the contribution of the startup times to the schedule length is growing, independently of the chosen set of processors. Therefore, the difference between the worst solution and the lower bound is decreasing with growing S. The genetic algorithm performs better than RND because it is able to build solutions with relative quality improving even with increasing domination of the startup time.

In Fig. 3.14d the quality of the solutions for growing dispersion of S is shown. The test instances for Fig. 3.14d were generated as in the first set of instances with $V = 20$, except for parameter S, which was generated with uniform distribution from range $[\frac{1-\delta_S}{2}, \frac{1+\delta_S}{2}]$. The value of δ_S is shown on the horizontal axis in Fig. 3.14d. As it can be seen, with growing δ_S, and hence growing heterogeneity of the system, the quality of all three types of solutions is worsening. This means that our problem becomes harder to solve with growing heterogeneity of the computing environment. Similar experiments were performed for controlled dispersion $\delta_A, \delta_B, \delta_C$ of parameters $A, \frac{B}{V}, C$, respectively. In all these experiments the dependence of the quality of the solutions on the range of diversity has a very similar shape as in Fig. 3.14d. This confirms once again that in heterogeneous systems good quality solutions are harder to obtain. Let us use the range of the changes of the worst-case solutions quality as an indicator of the sensitivity to the dispersion of a certain parameter. For δ_S changing from 1E-3 to 1, the distance from the lower bound grew ≈ 34 times. For similar changes of: (1) δ_C the distance changed ≈ 14 times, (2) δ_A it changed ≈ 1.8 times, (3) δ_B it changed ≈ 1.3 times. This means that the diversities of S and C have the strongest influence on the difficulty of obtaining good solutions, and the diversity of A and $\frac{B}{V}$ the smallest.

We finish these considerations with the following conclusions.

- It is easier to obtain good quality solutions when the communication time or the computation time dominates in the schedule length.
- It is easier to obtain good quality solutions for big memory buffers.
- It is easier to obtain good solution quality for homogeneous systems. Solution quality is particularly sensitive to the dispersion of communication parameters S, C, and less to the dispersion of $A, \frac{B}{V}$.
- The genetic algorithm really works, because it builds considerably better solutions than RND. Moreover, in some cases it is able to counteract the general tendencies of the solution quality represented in $RND, WRST$.

3.4.7 Conclusions

Overall, the experiments performed in Section 3.4 revealed a complex and sometimes counterintuitive interaction of the system parameters in determining good quality solutions. The following observations have been made.

- In the worst case an arbitrarily big number of messages may have to be accumulated on a processor in the optimum solutions. However, it turned out that in the near-optimum solutions obtained by the genetic algorithm chunk overlap is rare.

- There is a minimum number of messages that must be sent anyway. Using this number of communications may result in arbitrarily bad solutions. In the computational experiments it has been established that the number of messages is a small multiple of the minimum possible number. The communication startup time is the main disincentive to using great numbers of messages in delivering the load to the processors.

- There are inequalities in the load distribution and there exists a dominating set of processors which receives most of the load. The size of the dominating set of processors is growing with the load size V. There is a strong correlation between the parameters of a processor and its contribution in the load processing. Processors with faster communication links, bigger memory buffers, and computing faster receive more load. It appears that the order of parameter importance in the load distribution is C_i, B_i, S_i, A_i.

- A majority of load chunks, although not all, carry maximum possible load (equal to the size of the receiver's memory buffer). The number of full chunks grows with V, and is strongly correlated with parameters S_i, B_i.

- The problem has a natural tendency to become easier to solve when one parameter dominates in the schedule length. For example, big values of all A_i in relation to C_i, S_i simplify obtaining good solutions.

- Another side of the above observation is that it is relatively easy to build biased instances whose solutions are dictated by extreme values of a certain parameter, e.g. extremely slow communication or computation, or very small memory buffers.

- In a sense, parameters B_i and S_i work together when building a biased instance. Small memory buffers B_i incur many communications, which expose the cost of the startup times S_i. Conversely, big startup times may be compensated by the use of long messages which require big memory buffers.

- Good quality solutions are harder to obtain in heterogeneous systems.

3.5 Heuristics

The aim of the research presented in Section 3.4 was to gather information about desirable properties of the solutions of our problem. Based on this information, we propose several groups of heuristics. We also present the algorithms known from earlier literature. We start with very simple algorithms, which do not use the information about the nature of the problem. They are meant to verify if the algorithms presented later perform well or not. The most complex of the algorithms solving our problem is the genetic algorithm described in section 3.3.2. We compare the quality and the running time of all the proposed algorithms in a series of computational experiments. We examine how the system parameters influence the relations between the solutions generated by the algorithms. These experiments not only show which heuristic is better, or worse, to use for a given type of instance of the problem, but may also be used to verify the conclusions drawn from the experiments presented in the previous section.

3.5.1 Random Heuristics

The first group of algorithms we present are random heuristics. They were designed mainly to verify the performance of other algorithms by comparing the obtained solutions with what can be gained without effort, by random algorithms. Analyzing several different types of random heuristics may be helpful for distinguishing the most important elements in the process of creating a solution.

The first three random heuristics, introduced in [27], use two-step approach. They choose a communication sequence and overlap values in a random process, but afterwards the optimum chunk sizes are computed by LP (3.13)-(3.21). Heuristic **Rnd1** appends random processors to the communication sequence until the accumulated memory is not smaller than the load size V. All chunk overlaps δ_{ij} are set to 0 (no chunks overlap). This construction implies that the communication sequences created by Rnd1 will be short.

Heuristics Rnd2 and Rnd3 are strongly connected with Rnd1, as they use the communication sequence delivered by Rnd1. Heuristic **Rnd2** doubles this communication sequence and applies overlap 1 to all chunks. Overlap 1 means that each two consecutive chunks sent to the same processor overlap. The communication sequences created by Rnd2 are up to twice as long as in the case of Rnd1. However, they do not have to be exactly twice longer, as it is possible that some chunk sizes computed by the LP will be equal to 0.

Heuristic **Rnd3** lengthens the communication sequence obtained from Rnd1, appending a random number of chunks sent to random processors. The maximum length of the appended sequence is 3 times the original sequence length. The overlap values δ_{ij} are chosen randomly from values 0 and 1. Since some chunks overlap and their sizes together cannot exceed memory limits, the total memory collected may be smaller than the load size V. Therefore, some solutions delivered by Rnd3 may be infeasible. The communication sequences generated by Rnd3 are usually much longer than the sequences delivered by Rnd1.

Heuristics Rnd1 – Rnd3 randomize only the solution of the combinatorial part of the problem while still using LP (3.13)-(3.21) to find the chunk sizes. Therefore, we introduce another random heuristic **Rnd4** [9]. This algorithm is substantially different because it does not use LP to choose the chunk sizes. Rnd4 not only sends load to random processors, but also chooses chunk sizes randomly between 0 and memory buffer size for a given processor. If the size of the currently free memory on the receiver processor is not sufficient, sending the chunk is postponed until enough memory is released. The originator remains idle during this time. The whole process is repeated until all the load is sent. Computing overlap values is unnecessary in this algorithm, but it is possible to calculate them from the generated schedule.

The solution obtained by Rnd4 can be further improved by heuristic **Rnd4LP**. This algorithm uses the communication and overlap sequences delivered by Rnd4, but the chunk sizes are computed using LP (3.13)-(3.21).

3.5.2 First Free Heuristic

The next heuristic we created is called first free (FF) heuristic. This is one more simple algorithm designed mainly to test the quality of the other algorithms and the difficulty of test instances. In this algorithm each chunk has the maximum possible size (equal to the memory limit of the processor receiving this chunk). Hence, no chunks may overlap and all overlap values δ_{ij} must be 0. At the beginning, each of the processors $P_1, \dots, P_m$ receives one chunk. Each processor which finishes processing a chunk is added to a queue of free processors. Messages with new load are always sent to the first processor in this queue (i.e. the one which finished previous computations at the earliest moment). Note that FF does not order the processors in the first load distribution round. Thus, the sequence of the first m communications can be considered random. This may result in low quality solutions, especially when the number of necessary communications is low.

3.5.3 Appender Heuristics

The idea of appender heuristics (or appenders for short) first emerged in [27]. The appenders are meant to mimic the construction of the communication sequence σ and overlaps O as in B&B algorithm. For the purpose of constructing the communication sequence it is assumed that each chunk has (phantom) size equal to the memory buffer size on the chosen receiver processor. A message is always sent to the first free processor. In the search for a free processor, the workers are always checked in the same order, depending on a particular appender heuristic. Processors are appended to the communication sequence until the sent (phantom) load is at least three times greater than V. This excessive reservation of capacity for load processing is made to give some freedom in selecting chunk sizes. Afterwards, overlap $\delta_{ij} = 1$ is applied to all chunks. This choice is motivated by the observations that chunk overlaps often allow for better performance, but big overlaps do not seem necessary in most cases. The optimum chunk sizes are computed by LP (3.13)-(3.21).

The four basic appender heuristics **apA**, **apC**, **apS** and **apB**, search for a receiver of each chunk, checking one parameter only. The chunk will be sent to a free processor with the best value of this parameter (i.e. the smallest A_i, C_i, S_i and the greatest B_i, correspondingly). The heuristic apA, as the only algorithm from these four, was introduced and tested in [27]. Still, on the basis of the results from Section 3.4, we expect that parameter A should not be used as the main factor in choosing the processor. Thus, we want to compare this algorithm with the algorithms using different parameters.

A little more sophisticated approach was used in appenders **apSBC** and **apSBCA**. Heuristic apSBC, introduced in [27], searches for a free processor which can receive a whole buffer of data in the shortest time (i.e. the one with the smallest value of $S_i + B_iC_i$). Heuristic apSBCA is similar, but it takes into account the time of both communication and computation. Therefore, the processor with the smallest value of $S_i + B_i(C_i + A_i)$ is chosen.

Note that in algorithms apSBC and apSBCA processors with small memory buffers may be preferred, what is probably disadvantageous for the solution quality. Therefore, we propose the last two appender heuristics, **apSBCr** and **apSBCAr**, which use a modified idea of apSBC and apSBCA (cf. [8, 9]). Instead of the communication time or communication and computation time they take into account the time per unit of load. The processors are checked in the order of increasing values of S_i/B_i+C_i (for apSBCr) or $S_i/B_i+C_i+A_i$ (for apSBCAr). In this way, the processors with large, and not small memory buffers are preferred.

3.5.4 Best Rate Heuristics

The heuristics in the last group are called best rate heuristics [8, 9] and denoted by **BRx**, where $x \in \{1, \dots, 6\}$. In heuristics BRx it is assumed that the size of each chunk sent to processor P_i is equal to $\beta_i = B_i/x$. In order to choose a receiver of a chunk of data, for each processor P_i we compute the time T_i needed to process this chunk, were it sent to this processor:

$$T_i = \max\{\max\{t_0, \tau_i\} + S_i + \beta_i(C_i + A_i), t_i + \beta_i A_i\} - t_0, \tag{3.24}$$

where:

- t_0 is the time when the originator can start sending the chunk,
- τ_i is the time when enough memory becomes available at processor P_i,
- t_i is the moment when processor P_i completes processing the preceding chunks and can start processing the current chunk.

The load is always sent to the processor with the best current processing rate, i.e. with the minimum value of T_i/β_i. This process is repeated as long as there is some unprocessed load. The construction of the algorithm prevents using more memory than available. The values of chunk overlaps do not have to be computed in this algorithm, but they can be obtained from the generated schedule. As the chunk sizes are equal to B_i/x in heuristic BRx, the possible overlap values δ_{ij} are $0, \dots, x-1$. Thus, there is no chunk overlap in the solutions generated by BR1, but the remaining algorithms from this group can create solutions with overlapping chunks.

The result of each of the BRx heuristic may be improved by **BRxLP** heuristic. BRxLP uses the communication sequence and overlap values delivered by BRx, while the chunk sizes are computed using LP (3.13)-(3.21). Our experiments showed that the difference between the results obtained by BRx and BRxLP heuristics is very small, especially for larger values of x. Hence, in the next section we will present the results obtained by BRxLP heuristics only for $x = 1$ and $x = 2$.

3.6 Comparison of the Heuristic Algorithms

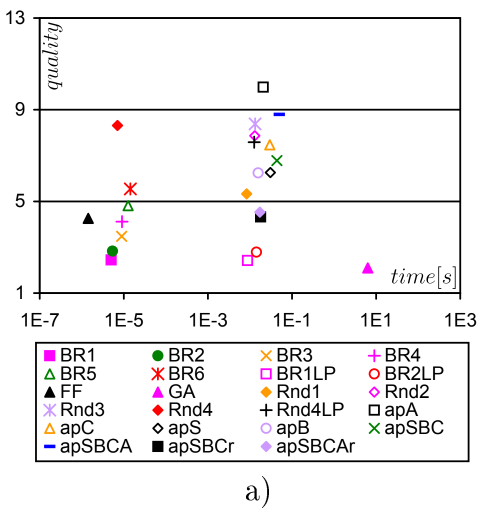
In this section we present the experimental results concerning the quality of the solutions and the computational costs of the heuristics presented in Section 3.5 and the genetic algorithm described in Section 3.3.2. Assessing quality of the algorithms is essentially a bicriterial problem, because the quality of the solutions is bought at some computational cost. The goal of this study is to analyze the quality of the solutions and the computational cost of obtaining them. The worst case estimations of the approximability ratios tend to be excessively pessimistic. Algorithms with high order of the worst case complexity sometimes have acceptable runtime. Hence, worst case estimations of the approximation ratio or the complexity do not seem to be a good tool to compare the practical trade-off between the quality and the cost. Therefore, experimental analysis is applied here. We use the same set of instances as in Section 3.4. We will examine the performance results of over 20 algorithms, demonstrating their advantages and weaknesses for different system parameters. As it was not possible to obtain the optimum results for the generated instances, the lower bound (LB), defined in Section 3.4.6 by formula (3.23), was used as a reference. The quality of all algorithms was measured as the average relative distance of the solutions from the lower bound.

In the following discussion the performance of the algorithms is shown on diagrams in which the horizontal axis is the average execution time, and the vertical axis is the average relative distance from the lower bound. For example, in Fig. 3.15 a good algorithm should be represented by a point as close as possible to the lower-left corner of the diagram which represents good quality and short execution time. In the sense of computational cost an algorithm dominates all the algorithms positioned to the right of it in the diagram. In the sense of solution quality an algorithm dominates all the algorithms positioned above it. Some algorithm may have the shortest execution time for a given quality, and vice versa, the best quality at a given computational cost. Thus, it is possible to consider some algorithms Pareto-optimal as non-dominated with respect to the quality and the run time.

3.6.1 Load Size

In the first series of experiments we examined the influence of the load size V on the quality of the results obtained by different algorithms. We present here the results obtained for the extreme values $V = 2$ and $V = 50$. Let us remind that in our problem the load is *arbitrarily* divisible, and even for $V = 2$ hundreds of communications may be performed. To control the running time of GA, we assumed that the number of communications in a schedule must be smaller than 1000. The same upper bound was applied to all other algorithms.

The relationship between the performance of different algorithms for $V = 2$ is presented in Fig. 3.15a. The analyzed algorithms can be divided into three groups based on their execution times. The fastest group are the algorithms not using LP (FF, BRx and Rnd4). The second group are heuristics computing optimal chunk sizes with LP (Rnd1 - Rnd3, Rnd4LP, appenders, BRxLP). The slowest of all algorithms is GA, as it creates many solutions and uses LP. Note that GA can be stopped after some number of iterations. Hence GA may be rep-



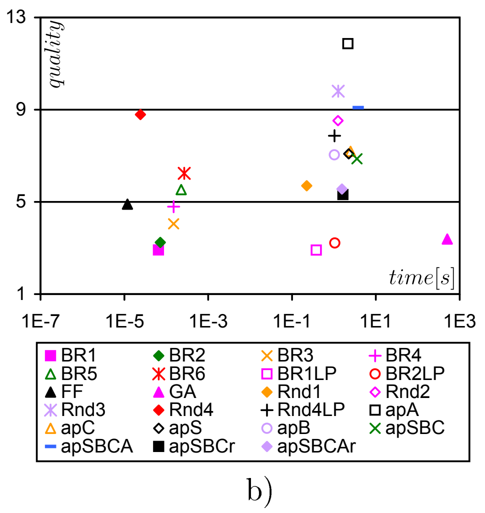


Figure 3.15: Solution quality vs. execution time for different problem sizes V, a) $V = 2$, b) $V = 50$.

resented by a dependence of quality versus time in Fig. 3.15. Since the existence of such a dependence does not change our conclusions, we decided to represent the performance of GA with just one point to make the picture more readable.

The best solutions are obtained by GA and then by heuristics BR1LP, BR2LP. Algorithms BR1 and BR2 are only slightly worse. The quality of BRx heuristics decreases with increasing x. This can be explained by the fact that dividing the memory buffers into more parts leads to sending a bigger number of smaller chunks. When there are too many messages, the contribution of startup times becomes too big and makes the whole schedule longer.

Most of the appender heuristics perform similarly to the random algorithms and are worse than our simplest heuristic FF. Thus, appenders are not good for solving our scheduling problem. This situation is similar for most of the test instances. The best of all appender heuristics are apSBCr and apSBCAr. Thus, we can conclude that our modification to appenders apSBC, apSBCA lead to a big improvement in the results in comparison to the appender heuristics proposed in the earlier literature. Appender apA delivers the worst solutions of all the studied algorithms. This confirms the observation from Section 3.4, that the computation speed alone cannot be the most important parameter to determine the order of sending data chunks.

The results obtained by the analyzed algorithms for $V = 50$ are presented in Fig. 3.15b. It can be seen that the quality and the execution time of almost all algorithms become worse for bigger V. This behavior is understandable, because larger instances intuitively should be harder to solve. For $V = 50$ the genetic algorithm is outperformed by heuristics BR1 and BR2 (even the variants without LP). Indeed, when there is more load to be processed, longer communication sequences are needed, and the search space of GA becomes much larger. Consequently, the chances of finding a good solution decrease, since the number of iterations performed in GA remains the same. There is no such effect on simple heuristics, and except for GA, there are no important changes in the relationships between the performance of the algorithms compared to $V = 2$.

We can conclude this section with an observation that the growing load size makes the problem harder from the point of view of the running time and quality for all proposed algorithms. Its impact on the genetic algorithm seems stronger than on other heuristics. However, mutual relationships in the performance of the heuristics remain almost unchanged for different values of V.

3.6.2 Startup Time

The results obtained for the extreme values of startup times ($S = 0.001$ and $S = 1$) are presented in Fig. 3.16. For small S the best results are obtained by GA. The second best algorithms are apC and apSBCr (their points overlap in Fig. 3.16a). This may be surprising, because we stated earlier that appender heuristics do not work well in general. However, when startup times are very small in relation to the other parameters, there is no need to keep the communication sequence short. Moreover, since in the analyzed instances parameter S is the same for all processors, only the other parameters are important. Parameter C seems to play the main role in this case. The results obtained by apSBCr are very

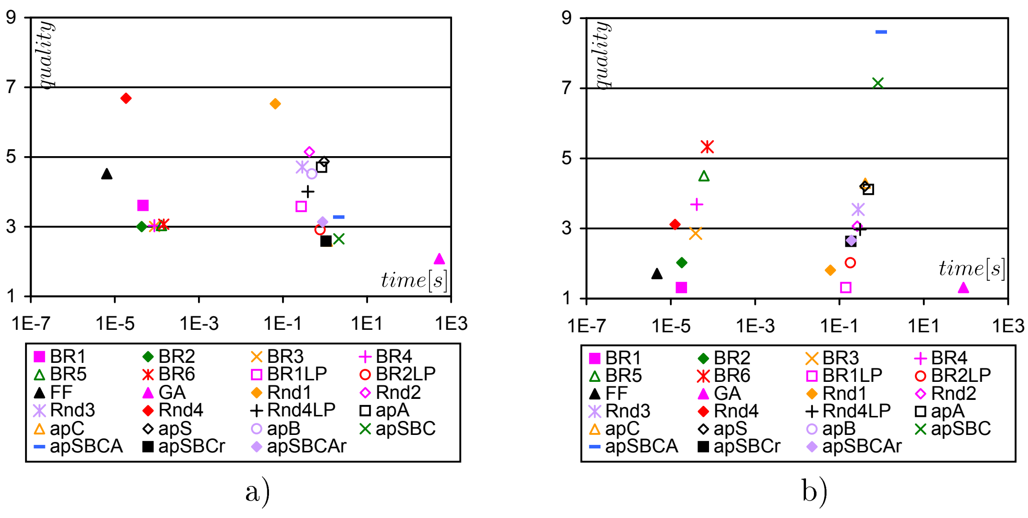


Figure 3.16: Solution quality vs. execution time for different communication startup times, a) $S = 0.001$, b) $S = 1$.

similar to the ones delivered by apC, because the values of $S_i/B_i + C_i$ are close to C_i for very small S_i. This conforms with the earlier results in DLT [3, 14, 17] stating that communication rate C is a key performance parameter in divisible load processing when $S_i = 0$ for all i.

The results of all BRx heuristics are similar, because the main difference between them is the number of messages sent and hence the contribution of the startup times, which has almost no importance for very small S. Heuristics sending always a full buffer of data (FF, Rnd1, BR1, BR1LP) perform badly, because they do not use overlapping, create longer waiting intervals during the communication and bigger imbalance in the computation completion times. This can be avoided at a little cost by sending many short messages if startup times are short. There are almost no differences in the results of BR2 - BR6. Thus, dividing the memory buffers into two parts (and hence overlap at most 1) seems enough to take the advantage of accumulating the load on processors.

The situation becomes completely different for $S = 1$. The main objective is now to minimize the contribution of startup times which dominates the schedule length. Therefore, algorithms creating the shortest communication sequences construct the best solutions. Even a very simple algorithm FF delivers solutions of good quality, because it always sends a full buffer of data. The differences between BRx heuristics become explicit. Splitting the communication into more parts leads to a big decrease in the solution quality. Heuristic BR1 works well, but with increasing x each BRx is getting worse, up to BR5 and BR6 being the worst of all BRx algorithms. It is worth mentioning that GA can handle this situation, creating shorter sequences and obtaining as good results as BR1. Appenders apA and apC do not perform well, because they do not take into account memory buffer sizes. As it is better to send a smaller number of messages, processors having big memory buffers should be preferred. Therefore, appenders apB, apSBCr and apSBCAr are better. Heuristics apSBC and apSBCA create extremely bad solutions. As all startup times are equal, the processors with smaller memory buffers may be preferred by these algorithms, what leads to constructing very long communication sequences and reinforces the contribution of startup times in the schedule length.

Except for the algorithms creating very long communication sequences, the solutions obtained for $S = 1$ have generally higher quality than for $S = 0.001$. This may be attributed to two facts. When big startup times dominate the whole schedule length, the other processor parameters are not very important anymore. Therefore, it is easier to construct good solutions, taking into account only one parameter instead of some combination. The second reason is that if startup times are big and equal, the lower bound LB better coincides with the actual optimum schedule length.

3.6.3 Communication Rate

The charts concerning parameter C (Fig. 3.17) show that for $C = 1$ our problem is much easier to solve than for small C. The schedule length is dominated by the communication time and it is not difficult to find a solution with the schedule length close to the lower bound. After magnifying Fig. 3.17b we could observe

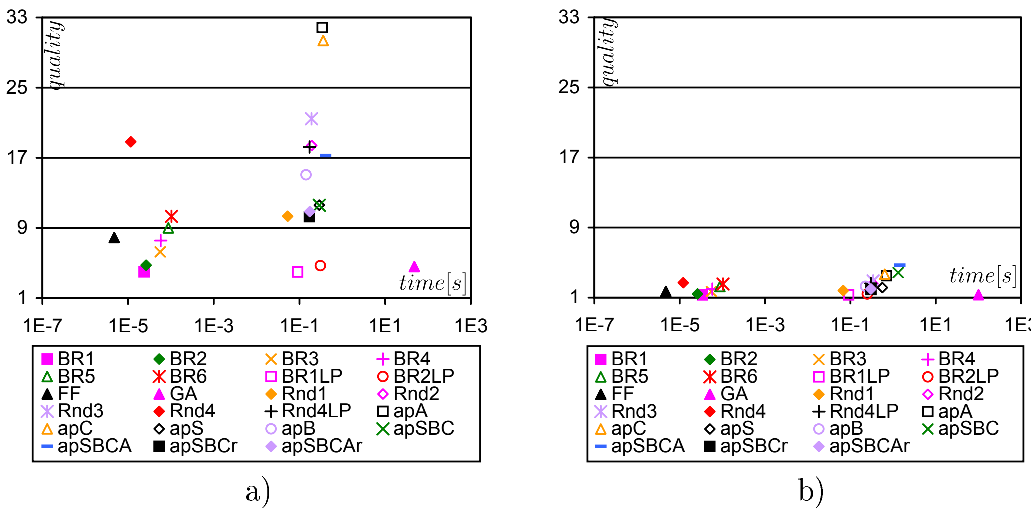


Figure 3.17: Solution quality vs. execution time for different communication rates, a) $C = 0.001$, b) $C = 1$.

that the relationships between different algorithms remain similar for $C = 0.001$ and for $C = 1$, with the best solutions delivered by heuristics BR1, BR1LP and very similar results of GA. This suggests that the construction of our algorithms is not sensitive to changing values of communication rate. The observed improvement in the solution quality for $C = 1$ in comparison to $C = 0.001$ is due to the nature of the problem, which is easier to solve for big communication rate C.

3.6.4 Memory Limit

Let us remind that we cannot analyze the influence of parameter B only, while using instances with different load sizes V. The parameter we should rather examine is the relative buffer size, i.e. B/V. This value determines the number of communications needed in a schedule and is a natural parameter of the problem instance.

For $B/V = 1$, it is possible to send the whole load as one message, and there is no need to create long communication sequences. Therefore, all the algorithms work faster than for smaller memory buffers (cf. Fig. 3.18a). The best solutions are achieved by GA, which becomes very effective when it does not have to create

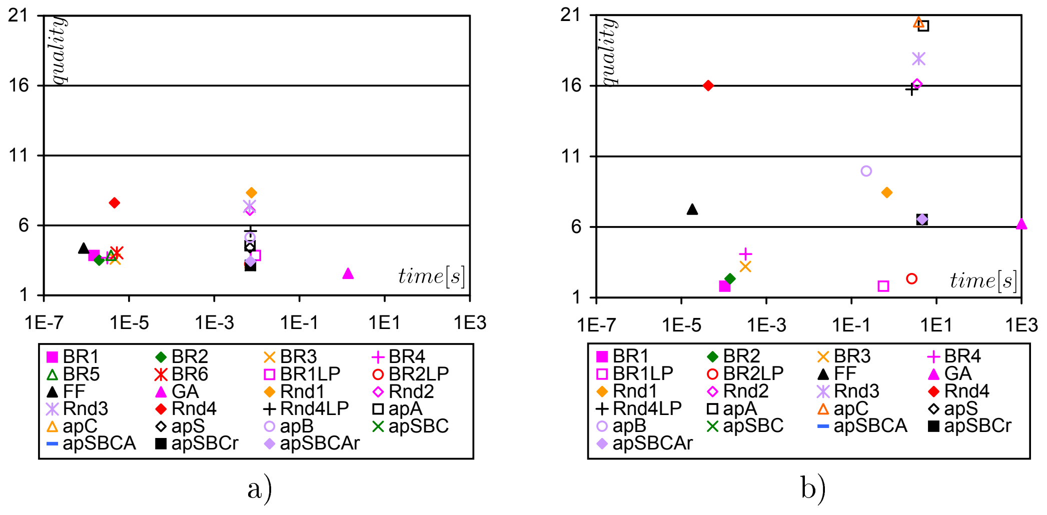


Figure 3.18: Solution quality vs. execution time for different memory limits, a) $B/V = 1$, b) $B/V = 0.005$.

and combine long sequences. Good results are also provided by heuristic BR2 and appenders apSBCr, apSBCAr. Heuristic BR1 is now a little worse than BR2 because it sends only one message in the whole schedule and cannot balance the use of computers performing fast computations and fast communication.

The smallest value of B/V which could be reliably solved by all algorithms (especially GA) without restricting the instance parameters was 0.005. Heuristics BR5 and BR6 had to produce communication sequences longer than the fixed limit we used (1000 and 1200 messages correspondingly). Therefore, they are not presented in Fig. 3.18b. For $B/V = 0.005$, each communication sequence length had to be at least equal to 200. With so many messages sent, startup times dominate the schedule length. Therefore, the best results are obtained by algorithms BR1 and BR2, creating short sequences and minimizing the contribution of startup times. Creating and combining very long sequences is a barrier for GA effectiveness, which performs similarly to appenders apS, apSBC, apSBCA, apSBCr and apSBCAr. These five algorithms deliver almost the same results (the points overlap in Fig. 3.18b) because small values of B expose the significance of parameter S in the last four appenders.

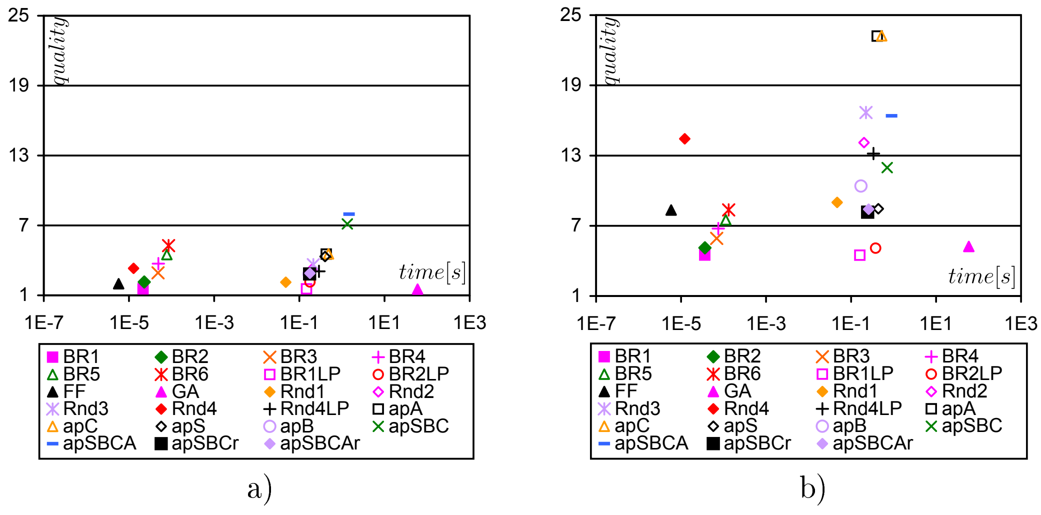


Figure 3.19: Solution quality vs. execution time for different dispersions of startup time, a) $\delta_S = 0.001$, b) $\delta_S = 1$.

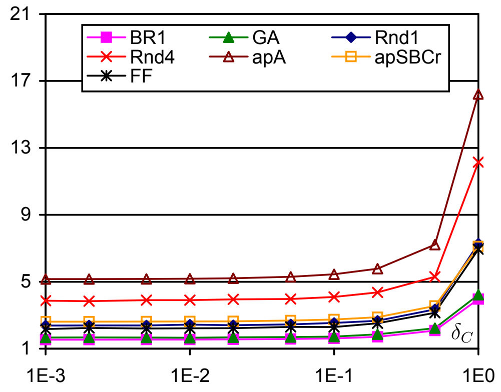
3.6.5 Computation Rate

It seems that changing the value of parameter A does not affect the relations between the solutions delivered by different algorithms. In both cases, $A = 0.001$ and $A = 1$, the chart obtained (not shown here) is very similar to the one presented in Fig. 3.15a. Increasing A leads to a slight improvement of all obtained results. This is an intuitively expected effect of a single parameter dominating the schedule length. It could be also observed for big values of C (Fig. 3.17b).

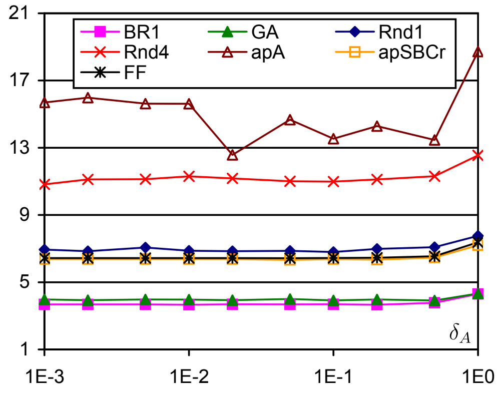
3.6.6 Parameters Dispersion

In this section we examine the influence of system heterogeneity, i.e. of the dispersion of the processor parameters. The method of generating test instances was described in Section 3.4.6.

The results for the dispersion of startup times are shown in Fig. 3.19. For larger dispersion algorithm FF loses quality in relation to BRx algorithms. There is also a reshuffling among appender algorithms. All the results get much worse when parameter S is chosen from a wider range. This phenomenon was observed for the dispersion of all parameters. However, the scale of the effect is different for



a)

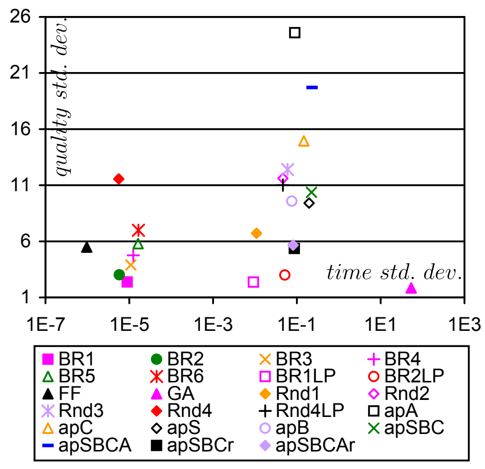


b)

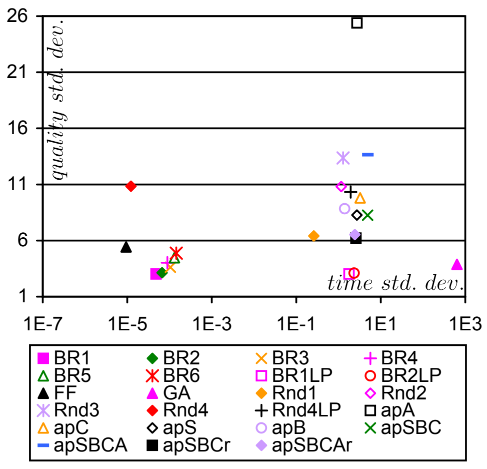
Figure 3.20: Average solution quality vs. parameter dispersion, a) δ_C, b) δ_A.

different parameters. It seems that the most important factor is the dispersion of S, then C, B/V, and finally A has the smallest impact. Thus, by its nature our problem is more difficult in a more heterogeneous system.

The relations between the solutions quality and the dispersion of parameters C and A are shown in Fig. 3.20. These charts confirm the difference between the importance of the dispersion of these two parameters. Changing δ_A has a small influence on the solutions quality. On the other hand, decreasing the dispersion of parameter C leads to a big improvement in the obtained results. Note that the quality of the solutions obtained in experiments on δ_A is similar to the ones with big dispersion of parameter C (the right end of Fig. 3.20a). This means that smaller differences between the communication speed of the processors make the problem easier to solve, but smaller differences in the computation speed do not help. The charts obtained for changing δ_S and δ_B (not shown here) are similar to Fig. 3.20a, although the changes in the solutions quality are smaller. We conclude that narrowing the range of A has almost no effect, when the dispersion of the other parameters is still big. Narrowing the range of the other parameters makes obtaining quality solutions easier.



a)



b)

Figure 3.21: Standard deviation of the solution quality vs. standard deviation of execution time, a) $V = 2$, b) $V = 50$.

3.6.7 Performance Dispersion

In the last group of experiments we analyze the dispersion of the solution quality and execution times for a set of test instances. The purpose of this section is to check if the algorithms good on average are also stable in cost and in quality.

In Fig. 3.21 the standard deviation of the solution quality is shown against the standard deviation of execution times. Before discussing the results let us comment on this way of the result depicting. In a set of test instances the dispersion of the execution times naturally exists for any algorithm. An algorithm solving our problem to optimality (e.g. B&B) has no dispersion of the relative distance from the optimum solution, but it has some dispersion of the distance from the lower bound. Thus, if we compare the dispersion of the solution quality relative to the lower bound, then some dispersion also naturally exists. Still, it is possible to compare algorithms with each other. An algorithm with very stable performance would be located in the lower-left corner of Fig. 3.21.

From Fig. 3.21 we conclude that with respect to the stability of the solution quality, the picture is very similar as for the average values. Algorithms that deliver best solutions on average also have the smallest standard deviation of the solution quality. As for the standard deviation of the execution time, it can be concluded that the deviation increases with the complexity of the algorithms. Thus, FF, Rnd4, BRx have the smallest standard deviation, then the algorithms using linear programming, and finally, GA has the biggest execution time standard deviation. We also analyzed dispersion in the relative sense and applied coefficients of variation (not shown here). For small problem sizes ($V = 2$) the picture is very similar to Fig. 3.21. For bigger problem sizes ($V = 50$) coefficients of variation for all algorithms are clustered around 1, and hence points representing all the algorithms are very close to each other. In this case algorithm BR6 dominates all other algorithms, while BR1LP and BR2LP have the two biggest coefficients of variation for the execution time.

We finish this section with a conclusion, that algorithms which are not dominated in the sense of average quality and average execution times are also non-dominated in the sense of dispersions of these values. Thus, good algorithms in the average sense have also stable performance.

3.7 Summary

The choice of a practical algorithm solving our problem depends on the budget of time to construct a solution and on the system parameters. A series of experiments showed that in a non-biased case the best quality results are achieved by the genetic algorithm and by heuristics BR1, BR2. Best rate heuristics (BR1, BR2) seem to offer good quality and very low cost. Thus, BR1, BR2 can be recommended as a universal solution. The genetic algorithm offers quality solutions, but at considerable cost. The number of communications is the factor determining complexity of GA. Hence, GA can be recommended only if the time budget is sufficiently big or $V/B \approx 1$.

The simplest heuristic FF may also be interesting as it is the fastest method, and its quality is not the worst. The class of appender heuristics in almost all tests turned out to be dominated both in the execution time and in the quality. Hence, we do not recommend the use of appender heuristics proposed in [27]. There seems to be no apparent advantage in heuristics using LP as a refinement after choosing the communication sequence and the load chunk overlap. It may be concluded that the combinatorial part of the problem determines the quality of a solution in a greater degree than the algebraic part.

We also reconfirmed the observations on the nature of the problem itself gathered in Section 3.4. The problem is easier to solve if one of the parameters A or C is big and determines the schedule length. Changing communication startup time S from close to zero to a dominating component of the schedule length changes the problem qualitatively. The memory size and the communication startup time are connected. On the one hand, small memory sizes impose numerous communications, and hence, expose the cost of startup times. On the other hand, the cost of big communication startup times may be reduced by sending as few messages as possible to processors with big memory buffers. It appeared easy to create biased test instances hard to solve by some type of heuristics. System heterogeneity makes the problem more difficult to solve for all proposed algorithms. Therefore, it is not advantageous to use very heterogeneous computing platforms. It seems advisable to group computing clients into classes according to, e.g., similar values of communication speeds, and to dedicate separate servers for each such class. Then, good schedules should be easier to build by the nature of the problem itself.

4 MapReduce Computations

In this chapter we analyze a new type of distributed computations embodied in the so-called MapReduce paradigm. In the previous chapters we analyzed scheduling one load volume in a star network topology. Now we move to analyzing two operations, mapping and reducing, interpreted as two divisible applications with precedence constraints. We start this chapter with the description of MapReduce paradigm and the distributed processing environment. Then we formulate the mathematical model of MapReduce computations and propose scheduling algorithms. The algorithms are tested in a series of computational experiments. Performance limits of MapReduce are also investigated. The results presented in this chapter are the first application of divisible load theory to processing applications with precedence constraints.

4.1 Outline of MapReduce

MapReduce is a programming model for processing large data sets on big numbers of computers. It can be implemented in many ways, and indeed it has various implementations [23, 40, 42, 44]. Notably, MapReduce has been applied as a production system at Google for processing Internet data [23]. Hence, it is very practical to analyze scheduling and performance of MapReduce. Here, we will outline MapReduce as described in [23]. In short, MapReduce computations consist in processing input data set by creating a set of intermediate key/value pairs, and then reducing them to yet another list of key/value pairs. The computations are performed in parallel.

In more detail, MapReduce applications are divided into two steps and defined by two functions: *Map* and *Reduce*. In the first step a Map function processes the input data set (e.g. a text or HTML file), and generates a set of intermediate $(key1, value1)$ pairs. In the second step these intermediate pairs are sorted by $key1$, and a Reduce function merges the intermediate pairs with equal values of $key1$, to produce a list of pairs $(key1, value2)$. In this way, the input data set is transformed into a list of key/value pairs.

Let us consider two examples given in [23]. Counting occurrences of words in a big set of documents can be organized in the following way. The Map function generates an intermediate pair $(word, 1)$ for each $word$ in the input file(s). The intermediate pairs are reduced by summing ones, and thus producing pairs $(word, count)$. In the inverted index computation all documents comprising certain $words$ must be identified. The Map function emits pairs $(word, docID)$ for each $word$ in the input file(s), where $docID$ is a document identifier (e.g. a URL of a web page). In the Reduce function all $(word, docID)$ pairs are sorted by $word$, and pairs $(word, list\_docIDs)$ are emitted, where $list\_docIDs$ is a sorted list of $docID$s. There are many types of practical applications which can be expressed in the MapReduce model. More detailed and advanced examples can be found in [23].

Both map and reduce operations are performed in parallel in a distributed computer system. Processing a MapReduce application starts with splitting the input files into load units, called splits in [23] (see Fig. 4.1). Many copies of the program start on a cluster of machines. One of the machines, called the master, assigns work to the other computers (workers). There are m map tasks and r reduce tasks to assign. In the further discussion the map tasks will be called *mappers*, and the reduce tasks *reducers*. A worker which received a mapper reads

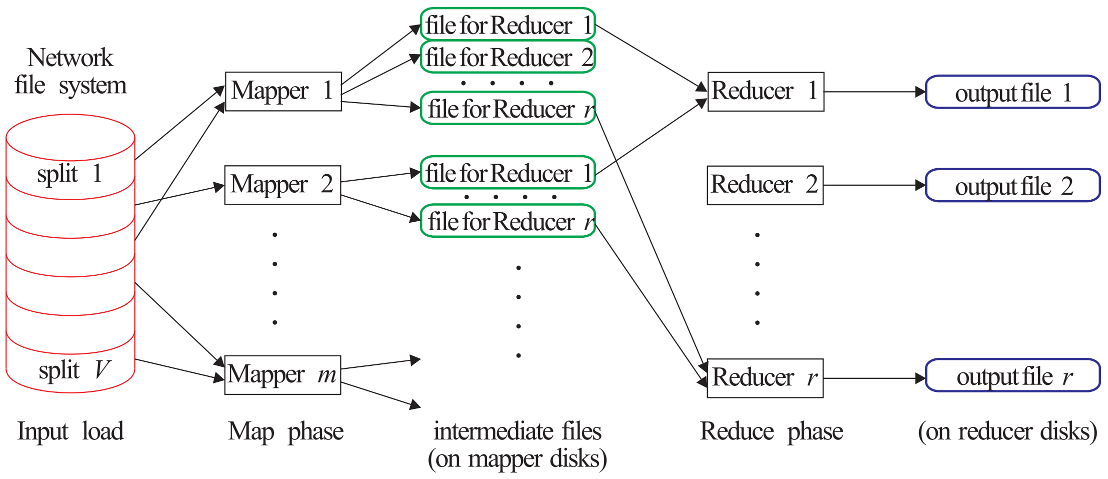


Figure 4.1: MapReduce execution overview.

the corresponding input load unit and processes the data using the Map function. The output of this function is divided into r parts by a *partitioning function* and written to r files on the local disk. Each of these r files corresponds to one of the reducers. Usually the partitioning function is of the form $hash(key1)$ mod r. The information about local file locations is sent back to the master, which forwards it to the reduce workers.

When a reduce worker receives this information, it reads the buffered data from the local disks of the map workers. After reading all intermediate data, the reduce worker sorts them by the intermediate keys in order to group together all occurrences of the same intermediate key. Each key and the corresponding set of values are then processed by the Reduce function. The generated output is appended to a final output file for a given reducer. Thus, the output of MapReduce is available in r output files. The execution of MapReduce is completed when all reducers finish their work.

4.2 Mathematical Model of MapReduce

In this section we formulate a mathematical model of MapReduce computations [7, 10]. We will pass from the "microscopic" view of the computations to a coarser "macroscopic" model used in the following sections. We simplify the perception of MapReduce computations to build a mathematically and conceptually tractable representation of the complex computing platform and the distributed application. Notation introduced in this section is summarized in Table 4.1.

Let us start with the model of the communication network. The structure of the network is unknown in general, but it is known that the bandwidth of the unthrottled communication channels which can be simultaneously used is limited. We will represent this limitation as the number l of communication channels which can be simultaneously in use without reducing the channel communication speed. Thus, if two processors can communicate with speed $1/C$ in the otherwise unused network, then the bandwidth limitation for the concurrent channels in the whole network is l/C. When referring to the above limit on the number of concurrent channels we will be talking about the *bisection width* limit.

We perceive the mappers and the reducers in a more coarse way than in [23]. In [23] a mapper is an application executing the Map function for one load unit. The size lu of a load unit is 16-64MB, and a processor receives approximately 100 load units [23]. Here we will assume that a single mapper is an application executing the Map function for all the load (i.e. all load units) assigned to a certain processor. Similarly, we unify all reducer computations assigned to a certain processor to a single (compound) reducer. Let m denote the number of mappers (consequently, also processors executing them), and let r denote the number of reducers. We assume that a mapper and a reducer can be executed on the same processor, but a reducer starts work only after the mapper finishes computations. Thus, the mapper and the reducer computations do not interleave on the same processor. It is usually assumed that $m \geq r$, but it is also possible to represent $m < r$ in our model. We exclude simultaneous execution of several mappers, or reducers, on the same computer. Were such coallocation possible, it can be represented in our model as several processors, each running a different

Table 4.1: Summary of notation for scheduling MapReduce applications.

α_i	size of the load processed by mapper i; in bytes;
a^{map}, c_i, s_i	microscopic computing rate, communication rate, communication startup time for processor P_i executing mapper i, respectively; expressed in seconds per byte (a^{map}, c_i) and in seconds (s_i);
a^{red}, s^{red}	microscopic computing rate and computation startup time for reducer application, equal for all processors, respectively; expressed in seconds per byte (a^{red}) and in seconds (s^{red});
$A_i = \frac{s_i}{lu} + a^{map} + c_i$	macroscopic computing rate of processor P_i executing a mapper application;
C	communication rate for reading mapper results by the reducers; expressed in seconds per byte;
γ_0	mapper result multiplicity fraction;
l	bisection width limit, expressed in parallel channels;
lu	size of the load unit, in bytes;
m	number of mappers;
P_i	processor executing mapper i;
r	number of reducers;
S	computation startup time, equal for all processors;
$T(m, r)$	schedule length on m mappers and r reducers;
T	schedule length, simplified notation for given m, r;
$\tau(x)$	reducer computing time function in load size x;
$t^{red} = s^{red} + \tau(\gamma_0 V/r)$	execution time of a reducer;
V	the whole load size, in bytes;

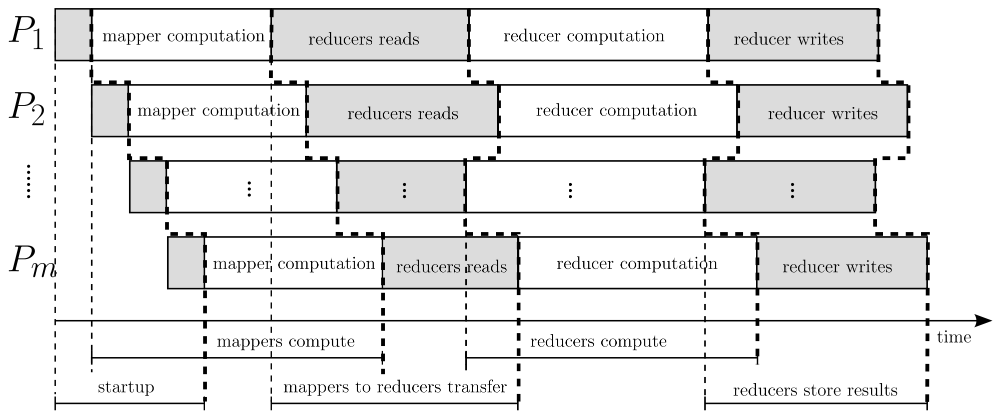


Figure 4.2: General view of MapReduce schedule structure.

mapper or reducer. The total size of the load to be processed is V.

A rough schedule structure of MapReduce computations is shown in Fig. 4.2. Detailed schedule structures are analyzed in the following sections. MapReduce computations are divided into several phases which may partially overlap. In the first stage the code for the mapper and reducer applications is loaded on the processors. For the sake of simplicity of presentation we assume that the mapper and reducer codes are uploaded together. We assume that most of the processors read the code from the network file system. The code may include virtual machines, libraries, the mapper and the reducer codes themselves. Thus, the computation startup time S may be quite long. The computation startup time elapses only once because when processing the following load chunks the code already resides on the executing processor. The differences in the startup time between the processors are negligible. We assume that processors read the code one by one. Although a more effective organization of the code broadcast is possible, we choose this simple distribution scheme to avoid more specific assumptions on the network structure and on the implementation of MapReduce.

In the second stage each mapper reads load units from the network file system, processes them, and stores the results in r local files for r reducers. A microscopic view of processing a single load unit of size lu (e.g. in bytes) by mapper i is shown in Fig. 4.3. Processor P_i (running mapper i) reads lu bytes

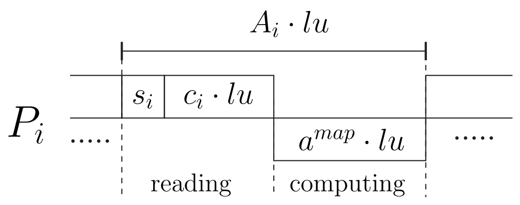


Figure 4.3: Microscopic view of Map computations for a single load unit.

of input in time $s_i + c_i \cdot lu$. Although computers are identical, the load may be read from local or from remote locations. Consequently, s_i, c_i are different for different processors. The fixed time delay s_i includes both communication and computation startup times needed in practice to start the computations and read the next load unit. The lu bytes of input are processed in time $a^{map} \cdot lu$. This time comprises both computations and storing the results in local files. Thus, we assume that from the point of view of computations only, processors are essentially the same, because local computing rate a^{map} is the same for all processors. The total time of processing a load unit is $s_i + (c_i + a^{map})lu$. Since the load reading, processing and storing operations are repeated many times (for hundreds of load units [23]), we simplify the representation of these operations to processing with rate A_i. It follows from the above discussion that $A_i \cdot lu = s_i + (c_i + a^{map})lu$, and the operations performed by a mapper may be perceived as if processing the load with the average rate $A_i = s_i/lu + c_i + a^{map}$. Here A_i depends on lu, but the size of the load unit is fixed for a MapReduce execution, hence also A_i is constant. In the following discussion we will use this coarse representation of mapper computations as performed with rate A_i. Let α_i denote the total size of load assigned to mapper i. According to the methodology of DLT we assume that α_i is a rational number. This simplification implies that the load assignment obtained in our model needs rounding to load units used in practical MapReduce. We assume that the effects of such load rounding are negligible. It will be assumed that the amount of results produced by the mappers is proportional to the input size. For α_i bytes of input $\gamma_0\alpha_i$ bytes of output are produced.

In the third stage (cf. Fig. 4.2) the results stored on the mappers are read by the reducers. We assume that the partitioning function divides the space of key values into r equal parts. This is achieved by the use of hashing in distributing the mapper output as described in Section 4.1. Consequently, the size of the input for each reducer is equal to $\gamma_0 V/r$ bytes in m chunks of sizes $\gamma_0\alpha_1/r, \dots, \gamma_0\alpha_m/r$. Each chunk comes from a dedicated file on a different processor. We assume that the reducers read the load from the mappers with equal rate C. There may be some advantages in the communication speed if a mapper and a reducer are executed on the same processor. Still, each reducer has to read its input from all mapper workers and such advantages cancel out when averaged over all the inputs. Moreover, the advantage of the local read in relation to the whole reading time diminishes with the increasing number of mappers m. Consequently, we assume that the differences in the communication rate for the transfers from the mappers to the reducers are negligible. Each of r reducers reads its input from mapper i in time $\gamma_0\alpha_i C/r$ unless there is bandwidth limitation. At most one channel can be opened to a mapper with transfer rate C. The methods of incorporating bandwidth limitations in the communication model are described in the following sections.

In the fourth stage r reducers sort the input data, perform reduce operations, and finally in the fifth stage store the results in the network file system. Let s^{red} denote the reducer computation startup time, and a^{red} (in seconds per byte) the reducer processing rate. Parameter a^{red} represents computations, transfers to local disks and storing the results in the network file system. All reducers receive input of roughly the same size $\gamma_0 V/r$. Consequently, all reducers have equal execution time $t^{red} = s^{red} + \tau(\gamma_0 V/r)$, where $\tau(x)$ is the running time of a reducer vs. the size x of the input. We will assume that the reducer execution time is $\tau(x) = a^{red}(x \log_2 x)$, which corresponds to the complexity of sorting. Here we assumed that writing the reducer results in the last stage is contention-free. This may not be true in general. Precautions to avoid reducers writing contention are mentioned in the further sections and in Chapter 5.

We assume that if there are other background services executed by the processors (e.g. for the network file system), then they influence the processor performance in a constant way. In other words, simultaneous computation and communication is possible, but performance parameters a^{map}, a^{red}, c_i, S, s_i, s^{red} remain constant.

MapReduce implementation includes procedures to tolerate failures. We do not include them explicitly in our model. However, a simple optimistic model of failure handling can be assumed for the purposes of performance modeling. Since the fault tolerance methods are based on retrying failed computations, these features can be represented as processing load greater than V (for mapping) or running additional mappers and reducers. The size of the additional load can be estimated using historical data on the failures.

Our goal is to partition the input load of size V into mapper chunks $\alpha_1, \dots, \alpha_m$ and schedule mapper to reducer communications so that the total schedule length T is as short as possible.

4.3 Schedule Dominance Properties

In this section we analyze schedule dominance properties for MapReduce computations. We start with presenting the optimum scheduling strategy for the case when only one reducer takes part in processing. Afterwards, we study schedule dominance properties for processing with many reducers.

4.3.1 Processing with a Single Reducer

We will say that the order of reading the results from the mappers by a reducer is the FIFO order if a reducer reads its inputs (mapper outputs) in the order

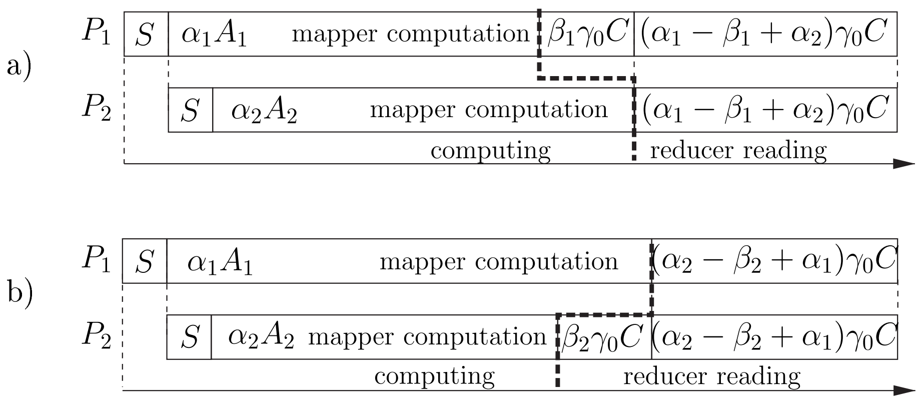


Figure 4.4: Reducer read orders. a) FIFO schedule structure, b) LIFO schedule structure.

of starting computations on the mapper processors (Fig. 4.4a). The opposite sequence of reading the results, starting from the last activated mapper processor, and finishing with the mapper activated as the first one, will be called the LIFO order (Fig. 4.4b). The results can be read from the mappers *sequentially*. This means that only after reading the whole file from mapper i can a reducer start reading the file from mapper $i+1$ (in the given sequence, e.g. FIFO or LIFO). In the opposite case a reducer may open two communication channels to mappers i and $i+1$ and read the files concurrently. In the latter case the bandwidth $1/C$ of the input to the processor running a reducer is *shared* by both channels. Below we argue that faster processors should start computations first, and that the results should be read sequentially in the FIFO order.

Proposition 4.1. *When there is only one reducer ($r = 1$), a MapReduce schedule activating mapper processors in the order of nondecreasing A_i, with sequential FIFO reducer reads, is optimum.*

Proof. We will show that the above schedule structure is optimum by comparing the amounts of load processed by the mapper processors in a given time T against different schedule organizations. The schedule structure proposed above allows for processing bigger load in time T than in other schedules. Therefore, it also allows for processing given load V in the shortest time. Let us analyze the case with two mappers ($m = 2$).

Let us first analyze the FIFO structure (see Fig. 4.4a) with bandwidth sharing. The reducer reads from the first mapper the load of size $\gamma_0\alpha_1$. Let $0 \le \gamma_0\beta_1 \le \gamma_0\alpha_1$ be the part of load read from the first mapper while the second mapper is still computing. The remaining part $\gamma_0(\alpha_1 - \beta_1)$ is read in parallel with the results from the second mapper. The speed of reading mapper results is determined by the shared bandwidth $\frac{1}{C}$ of the reducer input interface. Thus, we have the following relationships in the computing and communication times:

$$P_1: \qquad S + \alpha_1 A_1 + \beta_1\gamma_0 C + (\alpha_1 - \beta_1 + \alpha_2)\gamma_0 C = T \tag{4.1}$$

$$P_2: \qquad 2S + \alpha_2 A_2 + (\alpha_1 - \beta_1 + \alpha_2)\gamma_0 C = T, \tag{4.2}$$

from which we obtain

$$P_1: \qquad S + \alpha_1(A_1 + \gamma_0 C) + \alpha_2\gamma_0 C = T \tag{4.3}$$

$$P_2: \qquad 2S + \alpha_2(A_2 + \gamma_0 C) + (\alpha_1 - \beta_1)\gamma_0 C = T. \tag{4.4}$$

From (4.3) we obtain

$$\alpha_2 = \left(T - S - \alpha_1\left(A_1 + \gamma_0 C\right)\right)/\gamma_0 C \tag{4.5}$$

which substituted in (4.4) yields

$$\alpha_1 = \frac{TA_2 + S\gamma_0 C - SA_2 - \beta_1\gamma_0^2 C^2}{A_1 A_2 + A_1 C\gamma_0 + A_2 C\gamma_0}. \tag{4.6}$$

Returning with α_1 to (4.5), the load α_2 is

$$\alpha_2 = \frac{TA_1 - 2SA_1 - S\gamma_0 C + \beta_1\gamma_0 C A_1 + \beta_1\gamma_0^2 C^2}{A_1 A_2 + A_1 C\gamma_0 + A_2 C\gamma_0}. \tag{4.7}$$

Together we have

$$\alpha_1 + \alpha_2 = \frac{(T-S)(A_1+A_2) - SA_1 + \beta_1\gamma_0 C A_1}{A_1A_2 + A_1C\gamma_0 + A_2C\gamma_0}. \tag{4.8}$$

Note that the above load is increasing with β_1. Hence, it is biggest if $\beta_1 = \alpha_1$. This means that the bandwidth is not shared while reading the results from the second mapper. Therefore, the equation system (4.3)-(4.4) gets the following form:

$$P_1: \qquad S + \alpha_1 A_1 + (\alpha_1+\alpha_2)\gamma_0 C = T \tag{4.9}$$

$$P_2: \qquad 2S + \alpha_2(A_2 + \gamma_0 C) = T. \tag{4.10}$$

From (4.10) we obtain

$$\alpha_2 = \frac{T-2S}{A_2+\gamma_0 C}, \tag{4.11}$$

and by observing that $S + A_2\alpha_2 = \alpha_1(A_1+\gamma_0 C)$ (cf. Fig. 4.4a) we get

$$\alpha_1 = \frac{TA_2 + S\gamma_0 C - SA_2}{(A_1+\gamma_0 C)(A_2+\gamma_0 C)}. \tag{4.12}$$

The total size of the processed load is

$$\alpha_1 + \alpha_2 = \frac{T(A_1+A_2) + T\gamma_0 C - 2SA_1 - SA_2 - S\gamma_0 C}{(A_1+\gamma_0 C)(A_2+\gamma_0 C)}. \tag{4.13}$$

Let us now analyze the LIFO result reading order (cf. Fig. 4.4b). First let us check if bandwidth sharing while reading mapper results is profitable. Let $0 \le \gamma_0\beta_2 \le \gamma_0\alpha_2$ be the part of the results read by the reducer from P_2 while P_1 is still computing. Analogously to (4.3), (4.4) we obtain in the LIFO case:

$$P_1: \qquad S + \alpha_1(A_1+\gamma_0 C) + (\alpha_2 - \beta_2)\gamma_0 C = T \tag{4.14}$$

$$P_2: \qquad 2S + \alpha_2(A_2+\gamma_0 C) + \alpha_1\gamma_0 C = T. \tag{4.15}$$

From (4.15) we derive α_1 and substitute it in (4.14), from which we obtain

$$\alpha_2 = \frac{TA_1 - S\gamma_0 C - 2SA_1 - \beta_2\gamma_0^2 C^2}{A_1A_2 + A_1C\gamma_0 + A_2C\gamma_0}. \tag{4.16}$$

By substituting α_2 in (4.15) we have

$$\alpha_1 = \frac{TA_2 - SA_2 + S\gamma_0 C + \beta_2 A_2\gamma_0 C + \beta_2\gamma_0^2 C^2}{A_1A_2 + A_1C\gamma_0 + A_2C\gamma_0}. \tag{4.17}$$

Together the processed load is

$$\alpha_1 + \alpha_2 = \frac{(T-S)(A_1+A_2) - SA_1 + \beta_2 A_2\gamma_0 C}{A_1A_2 + A_1C\gamma_0 + A_2C\gamma_0}. \tag{4.18}$$

As in (4.8), it is a function strictly increasing with β_2. Hence, it is most effective to make $\beta_2 = \alpha_2$, i.e. the maximum possible. Consequently, bandwidth sharing while reading the results from two mappers is not profitable. Now we will calculate what amount of load is processed in the LIFO mode in given time T, provided that $\beta_2 = \alpha_2$. From (4.14)

$$\alpha_1 = \frac{T-S}{A_1 + \gamma_0 C}. \tag{4.19}$$

By observing that $A_1\alpha_1 = S + (A_2 + \gamma_0 C)\alpha_2$ and using the above value of α_1 we obtain

$$\alpha_2 = \frac{TA_1 - 2SA_1 - S\gamma_0 C}{(A_1+\gamma_0 C)(A_2+\gamma_0 C)}. \tag{4.20}$$

Together the load processed in the LIFO mode without bandwidth sharing is

$$\alpha_1 + \alpha_2 = \frac{T(A_1+A_2) + T\gamma_0 C - 2SA_1 - SA_2 - 2S\gamma_0 C}{(A_1+\gamma_0 C)(A_2+\gamma_0 C)}. \tag{4.21}$$

Comparing (4.13) and (4.21) we see that the FIFO order of the reducer input reading is more profitable because the numerator in (4.13) is bigger by $S\gamma_0 C$.

It remains to determine the optimum order of starting the computations on the processors. If we switch the order of activating the processors from (P_1, P_2), to (P_2, P_1) then the processor indices in (4.13) get swapped and the processed load is

$$\alpha'_1 + \alpha'_2 = \frac{T(A_1 + A_2) + T\gamma_0 C - 2SA_2 - SA_1 - S\gamma_0 C}{(A_1 + \gamma_0 C)(A_2 + \gamma_0 C)} \tag{4.22}$$

Subtracting $\alpha_1 + \alpha_2$ in equation (4.13) from $\alpha'_1 + \alpha'_2$ in the above equation we get

$$(\alpha'_1 + \alpha'_2) - (\alpha_1 + \alpha_2) = \frac{SA_1 - SA_2}{(A_1 + \gamma_0 C)(A_2 + \gamma_0 C)}. \tag{4.23}$$

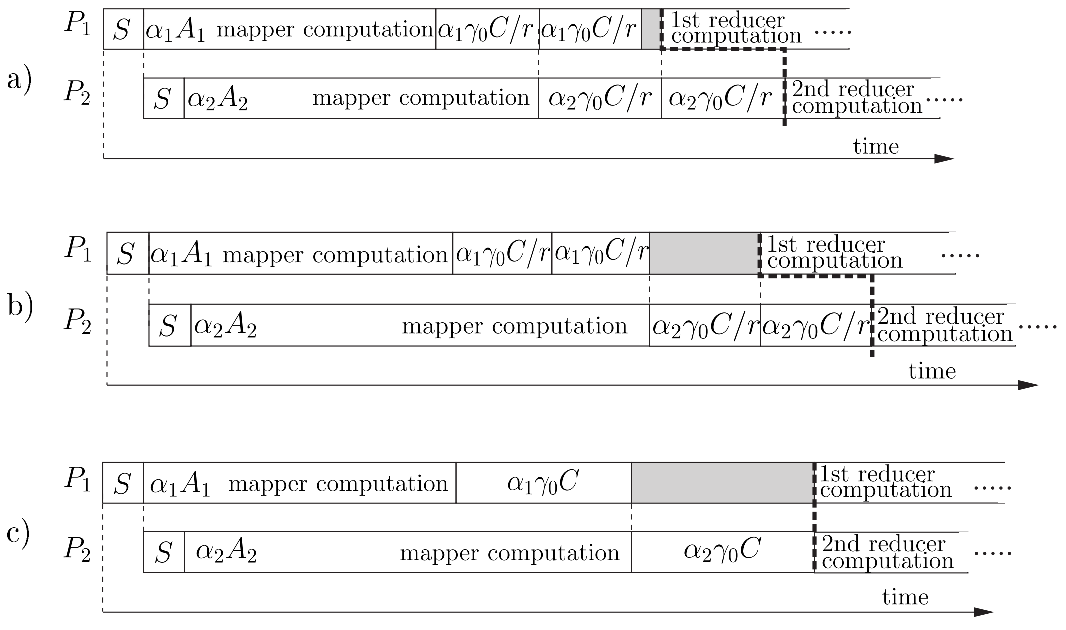
Thus, the load processed in time T increases after the swap only if $A_1 > A_2$. This means that in the order (P_1, P_2) we would have started computations on a slower processor first. Hence, the faster processor should start the computations earlier.

We demonstrated that for two mappers, sharing bandwidth while reading outputs from the mappers is not profitable both in the LIFO and in the FIFO order of reading. Of the two orders FIFO is better, and for FIFO the faster processor (i.e. the one with the smaller A_i) should be started first. This result can be iteratively extended to more than just two mappers. □

4.3.2 Processing with Many Reducers

In this section we consider scheduling for more than one reducer. Unfortunately, a generally optimum schedule structure, similar to the one defined in Proposition 4.1 for a single reducer, does not seem to exist for many reducers. On the contrary, it will be shown that each of the alternative schedule structures with many reducers can dominate the other under certain conditions.

As suggested by Proposition 4.1, we assume the FIFO order of finishing the computations on the mappers and that a single reducer is not reading the results from two (or more) mappers in parallel. As explained in Section 4.2, the amounts of load read by all reducers are the same. The actual processors running the

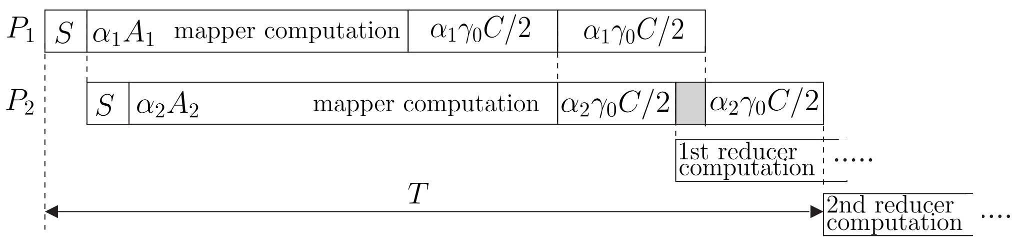


a)
P_1
S
$\alpha_1 A_1$ mapper computation
$\alpha_1\gamma_0 C/r$
$\alpha_1\gamma_0 C/r$
1st reducer computation .....
P_2
S
$\alpha_2 A_2$ mapper computation
$\alpha_2\gamma_0 C/r$
$\alpha_2\gamma_0 C/r$
2nd reducer computation .....
time
b)
P_1
S
$\alpha_1 A_1$ mapper computation
$\alpha_1\gamma_0 C/r$
$\alpha_1\gamma_0 C/r$
1st reducer computation .....
P_2
S
$\alpha_2 A_2$ mapper computation
$\alpha_2\gamma_0 C/r$
$\alpha_2\gamma_0 C/r$
2nd reducer computation .....
time
c)
P_1
S
$\alpha_1 A_1$ mapper computation
$\alpha_1\gamma_0 C$
1st reducer computation .....
P_2
S
$\alpha_2 A_2$ mapper computation
$\alpha_2\gamma_0 C$
2nd reducer computation .....
time

Figure 4.5: Many reducers exemplary read schedule structures. a) Case A, reducers read in parallel, b) case B, reducers read sequentially, c) reducers share bandwidth.

reducers can be arbitrary free machines. For example, P_1 can execute reducer 1 after completion of mapper 1, or it can be some other processor from a separate computer pool if such a pool exists.

The alternative communication schedules are shown in Fig. 4.5. In the first schedule type (Fig. 4.5a) the end of reading the results from P_i by the first reducer is synchronized with the end of the computations on P_{i+1}. Note that in this schedule different reducers read different mapper results in parallel, what may violate the bisection width limit. For the time being, we assume that the bisection width is not exceeded. We will call this schedule type case A. In the second type of schedule (Fig. 4.5b) the reducers read output from the mappers sequentially. The end of reading the data by the last reducer from mapper P_i coincides with the end of computation on mapper P_{i+1}. Here all reducer reads are sequential, only one communication channel is used at a time. Therefore, the speed of communication is the same as in one-to-one communication without network contention. We will refer to the second type of schedule as to case B.

a)


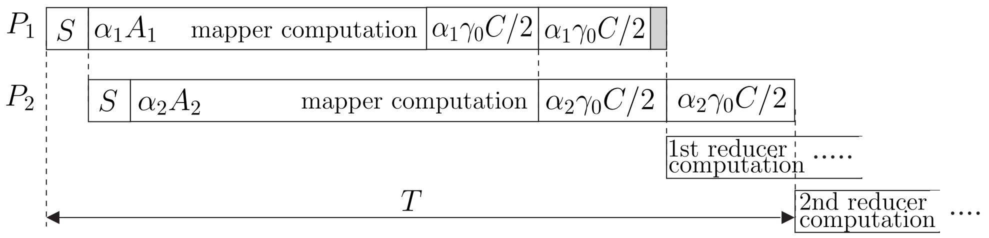
b)


Figure 4.6: Special cases of the first reducers read orders. a) Case A.1, b) Case A.2.

In the third type of schedule (Fig. 4.5c) the reducers read the results from the mappers one by one, but the bandwidth is equally shared between the reducers. The end of reading from mapper P_i coincides with the end of computations on mapper P_{i+1}. This case is very similar to Case B. Hence, we do not analyze it separately in the further discussion.

To demonstrate the lack of dominance of the above communication schedule structures we will calculate the amount of load processed on two mappers and transferred to two reducers ($m = r = 2$) in time T. Note that since the execution times of the reducers are equal, the minimization of T is equivalent to the minimization of the whole schedule length.

Case A. We can distinguish two sub-cases (Fig. 4.6). In the first one (case A.1) there is an idle time in the communications with P_2. This means that reading results from P_1 is longer than from P_2. Hence $\alpha_1 \geq \alpha_2$. In the second sub-case (case A.2) communication with P_2 is longer than with P_1, and $\alpha_1 \leq \alpha_2$.

Case A.1. In the first sub-case we have the conditions:

$$\alpha_1(A_1 + \gamma_0 C/2) = \alpha_2 A_2 + S \tag{4.24}$$

$$S + \alpha_1(A_1 + \gamma_0 C) + \alpha_2\gamma_0 C/2 = T \tag{4.25}$$

$$\alpha_1 \geq \alpha_2. \tag{4.26}$$

Hence, we obtain:

$$\alpha_1 = \frac{TA_2 - A_2S + \gamma_0 CS/2}{A_1A_2 + A_1\gamma_0 C/2 + A_2\gamma_0 C + \gamma_0^2C^2/4} \tag{4.27}$$

$$\alpha_2 = \frac{TA_1 + T\gamma_0 C/2 - 2A_1S - 3/2\gamma_0 CS}{A_1A_2 + A_1\gamma_0 C/2 + A_2\gamma_0 C + \gamma_0^2C^2/4} \tag{4.28}$$

$$\alpha_1 + \alpha_2 = \frac{T(A_1 + A_2 + \gamma_0 C/2) - S(2A_1 + A_2) - \gamma_0 CS}{A_1A_2 + A_1\gamma_0 C/2 + A_2\gamma_0 C + \gamma_0^2C^2/4}, \tag{4.29}$$

with an additional requirement $\alpha_1 \geq \alpha_2$ equivalent to:

$$T(A_2 - A_1 - \gamma_0 C/2) \geq A_2S - 2A_1S - 2\gamma_0 CS. \tag{4.30}$$

Case A.2. In the second sub-case we have the conditions:

$$\alpha_1(A_1 + \gamma_0 C/2) = \alpha_2 A_2 + S \tag{4.31}$$

$$2S + \alpha_2(A_2 + \gamma_0 C) = T \tag{4.32}$$

$$\alpha_1 \leq \alpha_2. \tag{4.33}$$

Hence, we obtain:

$$\alpha_1 = \frac{TA_2 - A_2S + \gamma_0 CS}{A_1A_2 + A_1\gamma_0 C + A_2\gamma_0 C/2 + \gamma_0^2C^2/2} \tag{4.34}$$

$$\alpha_2 = \frac{TA_1 + T\gamma_0 C/2 - 2A_1S - \gamma_0 CS}{A_1A_2 + A_1\gamma_0 C + A_2\gamma_0 C/2 + \gamma_0^2C^2/2} \tag{4.35}$$

$$\alpha_1 + \alpha_2 = \frac{T(A_1 + A_2 + \gamma_0 C/2) - S(2A_1 + A_2)}{A_1A_2 + A_1\gamma_0 C + A_2\gamma_0 C/2 + \gamma_0^2C^2/2}, \tag{4.36}$$

with an additional requirement $\alpha_1 \leq \alpha_2$ equivalent to:

$$T(A_2 - A_1 - \gamma_0 C/2) \leq A_2S - 2A_1S - 2\gamma_0 CS. \tag{4.37}$$

At least one of the conditions (4.30), (4.37) is always satisfied. If both are satisfied, then the load amounts given by (4.29) and (4.36) are equal.

Case B. In the current schedule structure we have the conditions (cf. Fig. 4.5b):

$$\alpha_1(A_1 + 2\gamma_0 C/2) = \alpha_2 A_2 + S \tag{4.38}$$

$$2S + \alpha_2(A_2 + 2\gamma_0 C/2) = T. \tag{4.39}$$

Thus, we obtain:

$$\alpha_1 = \frac{TA_2 - A_2 S + \gamma_0 CS}{A_1 A_2 + A_1\gamma_0 C + A_2\gamma_0 C + \gamma_0^2 C^2} \tag{4.40}$$

$$\alpha_2 = \frac{TA_1 + T\gamma_0 C - 2A_1 S - 2\gamma_0 CS}{A_1 A_2 + A_1\gamma_0 C + A_2\gamma_0 C + \gamma_0^2 C^2} \tag{4.41}$$

$$\alpha_1 + \alpha_2 = \frac{T(A_1 + A_2 + \gamma_0 C) - S(2A_1 + A_2) - \gamma_0 CS}{A_1 A_2 + A_1\gamma_0 C + A_2\gamma_0 C + \gamma_0^2 C^2}. \tag{4.42}$$

Let us now compare the amounts of load processed in time T in the above analyzed schedule structures. By comparing (4.29), (4.36), (4.42) we can see that none of the schedule structures always results in the biggest processed load for a given time T. Thus, no single communication schedule structure seems to be optimum in all cases. To build the optimum schedule a more general tool, possibly incorporating all possible structures, must be applied. On the other hand, if we concentrate only on the part of (4.29), (4.36), (4.42) which grows with T, then it can be concluded that for very big T (which may result from a need for processing very big loads) the load processed in cases A.1, A.2 is larger than in case B. For example, the difference between (4.29) and (4.42) in the part proportional to T is equal to

$$\frac{T(A_1+A_2+\gamma_0 C/2)}{A_1A_2+A_1\gamma_0 C/2+A_2\gamma_0 C+\gamma_0^2C^2/4} - \frac{T(A_1+A_2+\gamma_0 C)}{A_1A_2+A_1\gamma_0 C+A_2\gamma_0 C+\gamma_0^2C^2} =$$

$$\frac{T\gamma_0 C/2(A_1^2+3/2A_1\gamma_0 C+A_2\gamma_0 C/2+\gamma_0^2C^2/2)}{(A_1A_2+A_1\gamma_0 C/2+A_2\gamma_0 C+\gamma_0^2C^2/4)(A_1A_2+A_1\gamma_0 C+A_2\gamma_0 C+\gamma_0^2C^2)} > 0 \tag{4.43}$$

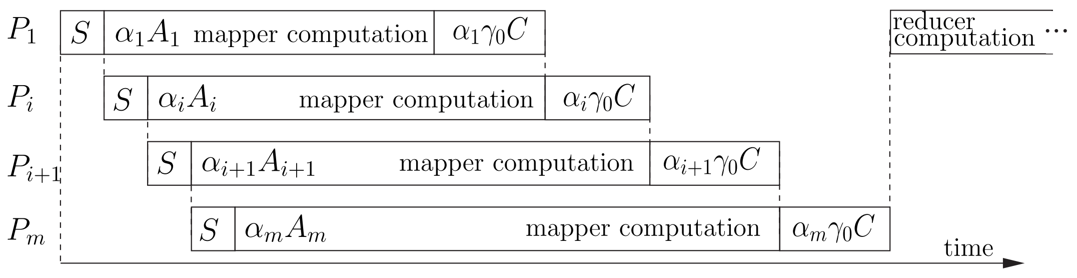

P_1
P_i
P_{i+1}
P_m
S
$\alpha_1 A_1$ mapper computation
$\alpha_1\gamma_0 C$
$\alpha_i A_i$ mapper computation
$\alpha_i\gamma_0 C$
$\alpha_{i+1} A_{i+1}$ mapper computation
$\alpha_{i+1}\gamma_0 C$
$\alpha_m A_m$ mapper computation
$\alpha_m\gamma_0 C$
reducer computation ...
time

Figure 4.7: The schedule structure for a single reducer.

A similar inequality can be derived for (4.36) and (4.42). Therefore, in the further discussion we will be using schedules based on case A.

4.4 Scheduling Algorithms

In this section we propose algorithms for the load partitioning in MapReduce computations. For processing with a single reducer, when optimum schedule pattern is known, we give an algorithm yielding the optimum load partitioning. In the case of many reducers the optimum schedule structure is not known. Hence, we propose two heuristic scheduling methods, based on the results obtained for a single reducer and inequality (4.43), and consider their advantages and disadvantages. A general scheduling algorithm for a sequence of reducing applications, each of which may be executed on many processors, will be presented in Chapter 5. Since all reducers have equal execution time t^{red}, we concentrate on minimizing the length of the partial schedule comprising mapper computations and mapper to reducer transmissions.

4.4.1 Single Reducer

Let us remind that it follows from Proposition 4.1 that the mappers should start the computations in the order of increasing A_is and the outputs from the mappers are read sequentially. Let us assume that processors $P_1, \dots, P_m$ running the mappers are numbered according to increasing A_is, i.e. $A_1 \leq A_2 \leq \dots \leq A_m$.

A schedule for the above setting is shown in Fig. 4.7. From Proposition 4.1 and from Fig. 4.7 we infer that the time of computing on processor P_i and reading its results is equal to the time of startup and computing on processor P_{i+1}. Hence we get a system of linear equations determining the load partitioning:

$$(A_i + \gamma_0 C)\alpha_i = S + A_{i+1}\alpha_{i+1} \qquad \text{for } \ i = 1, \ldots, m-1 \tag{4.44}$$

$$\sum_{i=1}^{m} \alpha_i = V. \tag{4.45}$$

The above linear system can be solved in $O(m)$ time for α_is by the reduction of α_i to affine linear functions of α_m, i.e. $\alpha_i = l_i + k_i\alpha_m$. More precisely, from (4.44)

$$l_m = 0 \qquad k_m = 1 \tag{4.46}$$

$$\begin{aligned}
\alpha_i \quad &= \frac{S}{A_i + \gamma_0 C} + \frac{A_{i+1}}{A_i + \gamma_0 C}\alpha_{i+1} = \\
&= \frac{S}{A_i + \gamma_0 C} + \frac{A_{i+1}}{A_i + \gamma_0 C}(l_{i+1} + k_{i+1}\alpha_m) = \\
&= \left(\frac{S}{A_i + \gamma_0 C} + \frac{A_{i+1}l_{i+1}}{A_i + \gamma_0 C}\right) + \left(\frac{A_{i+1}k_{i+1}}{A_i + \gamma_0 C}\right)\alpha_m = \\
&= l_i + k_i\alpha_m \qquad \text{for } \ i = m-1, \ldots, 1.
\end{aligned} \tag{4.47}$$

By substituting α_is in (4.45) we obtain

$$\alpha_m = \frac{V - \sum_{i=1}^{m} l_i}{\sum_{i=1}^{m} k_i} \tag{4.48}$$

and the remaining α_is are obtained from (4.48) and (4.47). Let us note that α_m in (4.48) may be negative. This negative solution is a demonstration that at the current parameters A_i, γ_0, C, S, V the number of processors m is too big to use them all. Therefore, if $\alpha_m < 0$, then the number of processors m must be decreased.

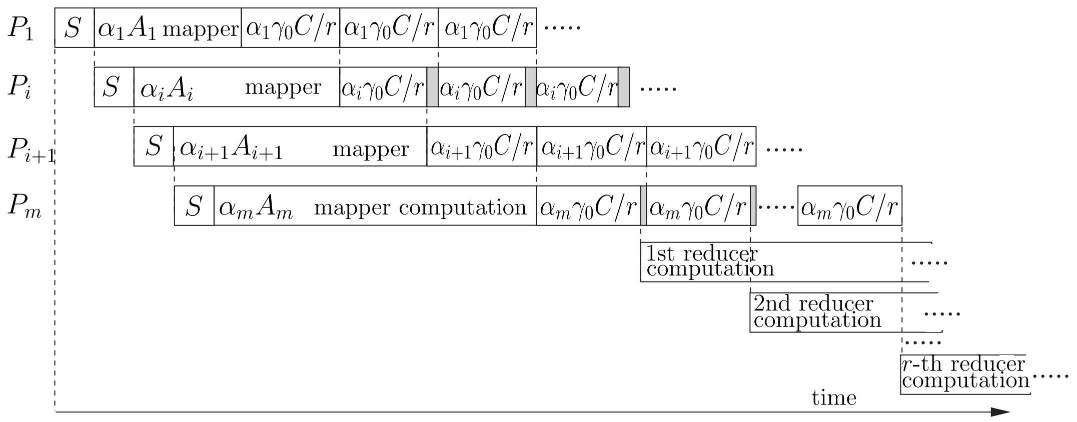


Figure 4.8: A schedule for many reducers. The first method.

If all α_i are nonnegative, then the total schedule length is (cf. Fig. 4.7)

$$T = mS + \alpha_m(A_m + \gamma_0 C) + t^{red} = S + \alpha_1 A_1 + \gamma_0 CV + t^{red}, \qquad (4.49)$$

where $t^{red} = s^{red} + \tau(\gamma_0 V/r)$.

4.4.2 Many Reducers

Below we propose two scheduling methods for MapReduce computations with many reducers. Each of them has its advantages and weaknesses. We summarize them at the end of this section.

The first method of load partitioning for many reducers is a natural extension of the method for a single reducer. The schedule structure is shown in Fig. 4.8. In this method the end of the read by the first reducer from P_i coincides with the mapper completion time on P_{i+1}. The method of calculating $\alpha_1, \dots, \alpha_m$ for $r = 1$ presented in Section 4.4.1 can be applied here with using the communication time $\gamma_0 C/r$ in place of $\gamma_0 C$. Thus, the load partitioning is determined by the system of linear equations:

$$(A_i + \gamma_0 C/r)\alpha_i = S + A_{i+1}\alpha_{i+1} \qquad \text{for } i = 1, \dots, m-1 \qquad (4.50)$$

$$\sum_{i=1}^{m} \alpha_i = V. \tag{4.51}$$

The solution of this system is given by formulas:

$$\alpha_i = l_i + k_i\alpha_m \qquad \text{for } i = m-1, \dots, 1 \tag{4.52}$$

$$l_m = 0, \quad k_m = 1 \tag{4.53}$$

$$l_i = \tfrac{S + A_{i+1}l_{i+1}}{(A_i + \gamma_0 C/r)}, \quad k_i = \tfrac{A_{i+1}k_{i+1}}{(A_i + \gamma_0 C/r)} \quad \text{for } i = m-1, \dots, 1 \tag{4.54}$$

$$\alpha_m = \tfrac{V - \sum_{i=1}^{m} l_i}{\sum_{i=1}^{m} k_i}. \tag{4.55}$$

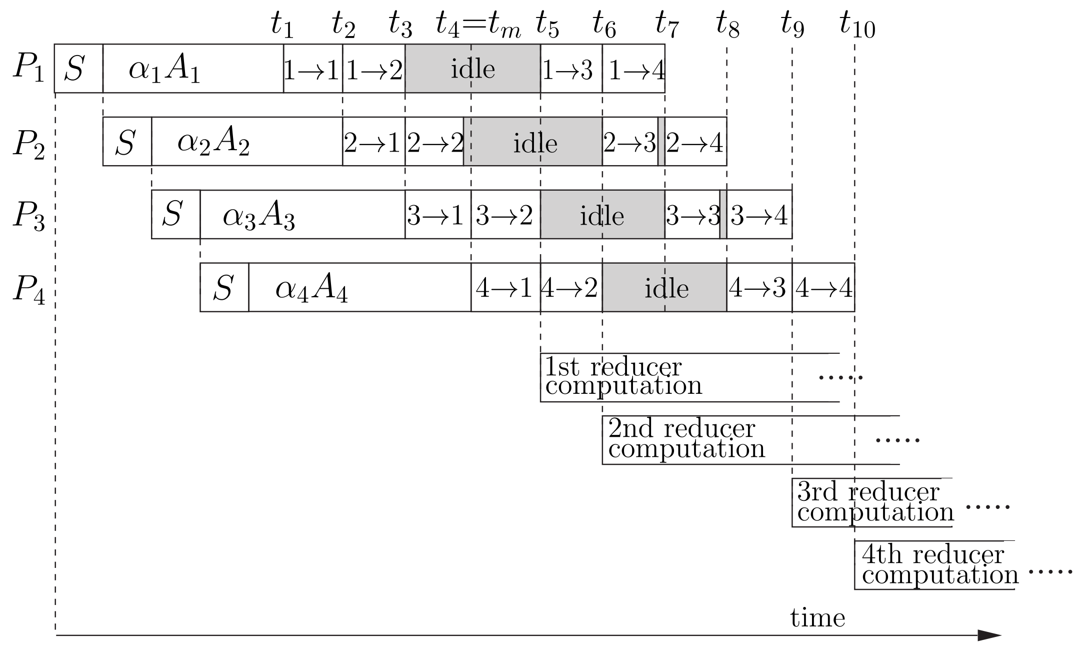
None of the mappers is read simultaneously by many reducers and no reducer reads outputs from many mappers in parallel. The bandwidths of the mappers' network output interfaces and the reducers' network input interfaces are not shared. Yet, the bisection width limitations are not obeyed if $l < r$. The schedule length is

$$T = \max_{i=1}^{m}\{iS + \alpha_i(A_i + \gamma_0 C) + \frac{\gamma_0 C}{r}\sum_{j=i+1}^{m}\alpha_j\} + t^{red}, \tag{4.56}$$

where $t^{red} = s^{red} + \tau(\gamma_0 V/r)$.

This schedule may be implemented as follows. Whenever P_i finishes transferring its results to reducer j, it notifies P_{i+1} to begin a transfer to j. Then P_i starts transferring results to reducer $j+1$, provided that it has been already notified to do it by P_{i-1}.

The second method assumes that the order of mapper to reducer communications is given, and they are preassigned to certain time intervals. The communication schedule structure is shown in Fig. 4.9. A mapper to reducer transfer appears in exactly one time interval. Hence, in each interval $[t_i, t_{i+1})$ a complete set of results of size $\gamma_0\alpha_j/r$ is read from a mapper P_j. All reducers read mapper processors in the same order: $P_1, P_2, \dots, P_m$. The order of reducer reads is the same for all the read mappers. New communication operations are started



t_1 t_2 t_3 $t_4=t_m$ t_5 t_6 t_7 t_8 t_9 t_{10}
P_1 S $\alpha_1 A_1$ 1→1 1→2 idle 1→3 1→4
P_2 S $\alpha_2 A_2$ 2→1 2→2 idle 2→3 2→4
P_3 S $\alpha_3 A_3$ 3→1 3→2 idle 3→3 3→4
P_4 S $\alpha_4 A_4$ 4→1 4→2 idle 4→3 4→4
1st reducer computation
2nd reducer computation
3rd reducer computation
4th reducer computation
time

Figure 4.9: A schedule for many reducers. The second method. Notation $i \to j$ means transfer of mapper i results to reducer j.

as soon as the mappers finish their computations and the sufficient number of communication channels (not exceeding the bisection width l) is available. The bisection width limitation is obeyed, as well as sequential reading of the mapper results by the reducers. This schedule can be implemented as in the previous method with additional precautions not to exceed the bisection width limit l. For example, whenever the number k of completed load transfers on mapper 1 is such that $k \bmod l = 0$, mapper 1 waits to be notified by mapper m that $k-l+1$ transfers from mapper m are completed.

Let us analyze the number of necessary communication intervals. If $l \geq r$, then the bisection width limit l is not bounding, and the number of intervals necessary to perform m reads by each of r reducers is $m+r-1$. On the other hand, if the number of simultaneous channels is $l < r$, then after opening l read channels by l reducers the $(l+1)$-th reducer shall wait until the completion of the read operation of the first reducer from the m-th mapper. This requires $m-l$ additional communications of reducer 1 with mappers $P_{l+1}, \ldots, P_m$ to release a communication channel. Consequently, $m-l$ idle intervals appear in the reads from each mapper. Then, at the end of each interval $[t_m, t_{m+1}), \dots, [t_{m+l}, t_{m+l+1})$ a new read operation is started by reducers $l+1, \dots, l+l$. Thus, after $m-l$ idle intervals, read operations are performed in the following l intervals. The sequences of $m-l$ idle intervals are inserted in the schedule $\lceil \frac{r}{l} \rceil - 1$ times. Overall, there are $(\lceil \frac{r}{l} \rceil - 1)(m-l)+m+r-1$ intervals in the communication schedule. For simplicity of notation let us introduce a function $itv(i,j)$ which returns the number of the interval in which reducer j reads output of mapper i (counting starts with value 1 for interval $[t_1, t_2)$). The values of $itv(i,j)$ can be calculated as follows:

$$itv(i,j) = \left(\left\lceil \frac{j}{l} \right\rceil - 1 \right) m + i + (j-1) \bmod l \tag{4.57}$$

for $i = 1, \dots, m, j = 1, \dots, r$. Let $vti(i)$ be the set of mappers which are read in interval i, i.e.

$$vti(i) = \{a : itv(a,b) = i, b \in \{1, \dots, r\}\}. \tag{4.58}$$

The values of $vti(i)$ can be tabulated in $O(mr)$ time. The partition of the load can be calculated from the following linear program.

$$\text{minimize} \quad t_{itv(m,r)+1} \tag{4.59}$$

$$iS + A_i \alpha_i = t_i \quad \text{for} \quad i = 1, \dots, m \tag{4.60}$$

$$\frac{\gamma_0 C}{r} \alpha_k \le t_{i+1} - t_i \quad \text{for} \quad i = 1, \dots, itv(m,r),\ k \in vti(i) \tag{4.61}$$

$$\sum_{i=1}^{m} \alpha_i = V \tag{4.62}$$

In the above linear program α_i, t_i are variables. We minimize the completion time of the last communication $t_{itv(m,r)+1}$. By constraints (4.60) the computations finish before reading from the mappers is started. Inequalities (4.61) guarantee that all communications fit in the time intervals where they are assigned. The whole load is processed by (4.62). The linear program (4.59)-(4.62) has $itv(m,r)+$

$1+m$ variables, which is $O(mr/l)$, and at most $m+1+itv(m,r)l$ constraints, which is $O(mr)$.

The above linear program can be further simplified. Let us remind that the reducers read equal size outputs from a certain mapper. For example, all communications (i,j) from a fixed mapper i to reducers $j=1,\dots,r$ have the same size $\gamma_0\alpha_i/r$. Consequently, intervals $[t_{ma+i}, t_{ma+i+1})$, and $[t_{mb+i}, t_{mb+i+1})$ have equal length because they comprise read operations from the same set of mappers, for some positive integers $i, a<b$ such that $mb+i \le itv(m,r)$. The block of intervals $[t_m, t_{m+1}), \dots, [t_{2m-1}, t_{2m})$ is repeated $(\lceil\frac{r}{l}\rceil-1)$ times. After them $(r-1) \bmod l$ intervals follow which repeat the lengths of some earlier intervals. Namely, the distance between $t_{itv(m,(\lceil\frac{r}{l}\rceil-1)l+1)}$ and $t_{itv(m,r)+1}$ is equal to the distance between t_m and $t_{m+((r-1) \bmod l)+1}$. Consequently, the length of the schedule until the end of mapper to reducer communications is

$$\begin{aligned} & t_m + (\lceil\frac{r}{l}\rceil - 1)(t_{2m}-t_m) + (t_{m+((r-1) \bmod l)+1} - t_m) = \\ = \; & (\lceil\frac{r}{l}\rceil - 1)(t_{2m}-t_m) + t_{m+((r-1) \bmod l)+1}. \end{aligned} \tag{4.63}$$

The values of variables t_{am+i} for $1<a\le\lceil\frac{r}{l}\rceil-1$ and $0\le i<m$ can be calculated as $t_m + a(t_{2m}-t_m) + (t_{m+i}-t_m) = a(t_{2m}-t_m) + t_{m+i}$. Hence, LP (4.59)-(4.62) can be reduced to

$$\text{minimize} \quad (\lceil\frac{r}{l}\rceil - 1)(t_{2m}-t_m) + t_{m+((r-1) \bmod l)+1} \tag{4.64}$$

$$iS + A_i\alpha_i = t_i \quad \text{for} \quad i=1,\dots,m \tag{4.65}$$

$$\frac{\gamma_0 C}{r}\alpha_k \le t_{i+1} - t_i \quad \text{for} \quad i=1,\dots,2m, k\in vti(i) \tag{4.66}$$

$$\sum_{i=1}^{m} \alpha_i = V. \tag{4.67}$$

The functions of the constraints in the above LP are the same as in the earlier one. The number of variables is $3m$, the number of constraints is at most $2ml+m+1$.

The objective function (4.64) reduces to t_{m+r} if $r \leq l$.

The above two methods of scheduling have advantages and disadvantages. The first algorithm is mathematically simple and easy to implement in practice. On the other hand it ignores the network bisection width. The second one is more precise in representing bandwidth limitations. Additionally, since the reducer computations start times are spread in time, the reducer writes are also spread in time, what allows to avoid network contention while writing the final results. On the other hand, this method makes specific assumptions (although not unrealistic) on the structure of the schedule, and requires more careful coordination (synchronization) of communications.

4.5 Performance Analysis

Below we analyze the influence of the system parameters on the performance of MapReduce computations. All linear programs were solved using lp_solve linear programming library [41]. Unless stated otherwise, we assume the following reference system and application parameters: lu =16E6, $m = 1000$, $r = l = 100$, $S = 1$, $C = c^{map}$ =1E-8, $\gamma_0 = 0.1$, $a^{map} = a^{red}$ =1E-7, $s^{map} = s^{red}$ =1E-2, V = 1E15. The above values can be interpreted as follows. The size of a load unit is approximately 16 MB. There are 1000 mappers, 100 reducers, and the bisection width is not bounding. We will study the influence of the limited bisection width in the further text. The computation startup time S is 1s. The communication rate, both for the mappers and for the reducers, is 10ns/B. The computation startup times for each unit of load on the mapper, and for the reducers are 10ms. The load size is 1PB.

If the bisection width l is not bounding, then both methods of load partitioning presented in Section 4.4 give similar results (within the range of analyzed parameter values). Therefore, with the exception of channel number l consider-

Table 4.2: MapReduce phase duration vs. problem size V.

V	startup	mapping	mapper to reducer transfer	reducing	T
1E17	2.10E-04%	2.43%	0.231%	97.6%	4.77E08
1E16	2.25E-03%	2.61%	0.249%	97.4%	4.44E07
1E15	2.44E-02%	2.83%	0.269%	97.2%	4.10E06
1E14	0.265%	3.19%	0.291%	96.8%	3.78E05
1E13	2.87%	4.73%	0.303%	95.3%	3.49E04
4.383E12	6.65%	6.64%	0.292%	93.3%	1.50E04

ations, we will present the results obtained for the first method, which is much faster.

Let us start with analyzing characteristic schedule features. This will be useful in understanding the following results. In Table 4.2 we have collected the relative durations of the MapReduce phases for various problem sizes. Note that the percentages do not sum to 100% because the phases partially overlap. The last line is given for the smallest load for which $m = 1000$ mappers could be effectively used. For smaller V some of the α_is become negative. It can be seen in Table 4.2 that the schedule length is dominated by the reducing time, and this domination grows with the problem size V. This observation remains valid also for higher complexity functions than $\tau(x) = a^{red}(x \log_2 x)$, because then the reduction time dominates even more.

In Fig. 4.10 the imbalance of the mapper load distribution is presented. On the horizontal axis the indices of the mappers are shown. A bigger number means that the mapper is activated later. On the vertical axis the fractions $\alpha_i/(V/m)$ are shown. The dependencies are depicted for instances with one parameter changed with respect to the reference system. Instances with C = 1E-9, $\gamma_0 = 0.01, r = 1000$ are shown as one line as they all represent equal load partition. It can be seen that for some system configurations the load on the mappers increases, for some other configurations the load fractions decrease. The border

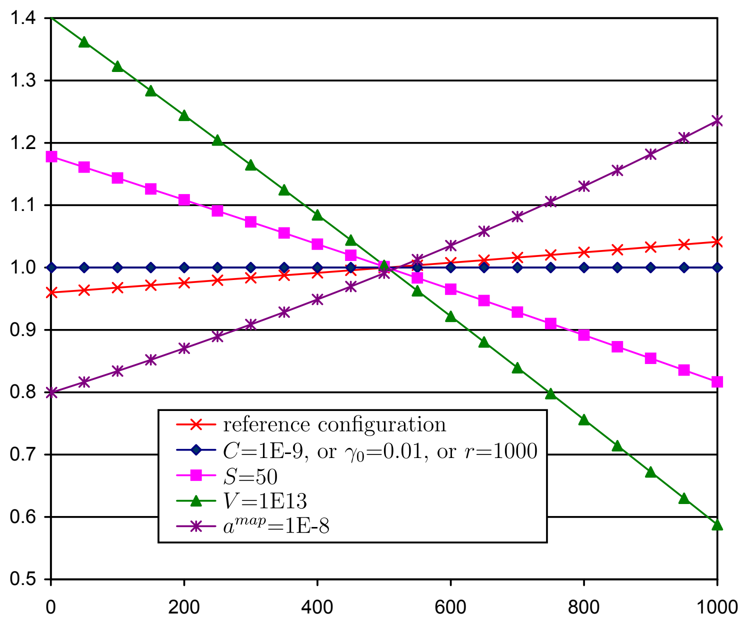


Figure 4.10: Skew of mapper loads, for varied system parameters.

cases are systems which satisfy $Srm = \gamma_0 VC$. This formula can be derived analytically for a homogeneous system by calculating by how much the load in a pair of processors P_i, P_{i+1} must differ to satisfy equation (4.50). Precisely, let $\alpha_i = \alpha_{i+1} + \Delta$, $A_i = A_{i+1} = A$. From equation (4.50) we have

$$(A + \gamma_0 C/r)(\alpha_{i+1} + \Delta) = S + A\alpha_{i+1}. \tag{4.68}$$

From this it can be derived that

$$\Delta = (Sr - \alpha_{i+1}\gamma_0 C)/(Ar + C\gamma_0). \tag{4.69}$$

By assuming that the load distribution is equal, $\alpha_{i+1} = V/m$ and $\Delta = 0$, we obtain the above mentioned formula. We observed that the systems with $Srm < \gamma_0 VC$ have increasing load distribution $\alpha_1 < \alpha_2 < \ldots < \alpha_m$, while systems with $Srm > \gamma_0 VC$ have decreasing load distribution $\alpha_1 > \alpha_2 > \ldots > \alpha_m$. The second case seems advantageous for the overall performance because the mapper completion times are less scattered in time (see Fig. 4.8). Note that the reference system has the less advantageous, increasing load distribution.

In the following discussion we use speedup as a performance index. Classically, speedup is calculated as acceleration of computations on a certain number of processors with reference to the execution time on a single machine. However, mapping and reducing can be performed on different numbers of processors, what makes the dependence 2-dimensional, and consequently harder to understand. For clarity of the following charts it is often more convenient to use different reference systems than a single-processor configuration. Therefore, we define speedup in a slightly more general way:

$$\varsigma(a,b) = \frac{T(a,b)}{T(m,r)}, \tag{4.70}$$

where $T(a,b)$ is the schedule length for the reference system with a mappers and b reducers, and $T(m,r)$ is the schedule length for the tested system with m mappers and r reducers. Here m, r are subject to change, and a, b remain constant. In some tests even m, r remain fixed, and other parameters (e.g. lu, a^{map}, l) are varied. According to equation (4.70), $\varsigma(1,1)$ is equivalent to the classic speedup. The above definition emphasizes the reference system which may be different for different charts.

The performance of MapReduce with respect to growing number of mappers m is shown in Fig. 4.11. It can be observed in Fig. 4.11 that for $r \geq 100$ the number of mappers that can be effectively exploited is smaller than 20000. Moreover, with growing m the speedup $\varsigma(1,1)$ levels off around r when $m > r$, but it grows with increasing r. This can be explained in the following way. The whole processing time has three main components: the time of mapping, the interval of data transfer from the mappers to the reducers, and the reducing time. For variable number of mappers m, and the remaining system configuration fixed, only the first interval is changing its length with m. If $m < r$ then increasing mapper number m reduces the schedule length, and hence the speedup grows

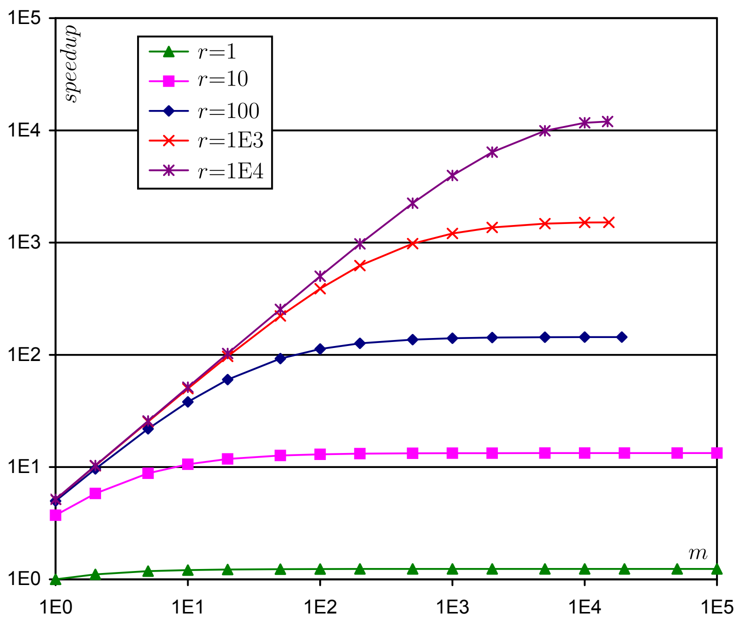


Figure 4.11: Speedup $\varsigma(1,1)$ vs. the number of mappers m for various numbers of reducers r. Load size $V = 1\text{E}15$.

nearly linearly. For $m > r$ the mapping time becomes much smaller than the other two intervals. Hence, the reducing time dominates and determines the speedup for $m > r$. The reducing time, in turn, decreases with r even faster than linearly because the complexity function of reducing operations is nonlinear in V/r. Hence, the speedup increases with r slightly faster than r when $m \gg r$. Let us note that the above observations depend very much on the amount of results $\gamma_0 V$ produced by the mappers. We discuss it in the further text (cf. Fig. 4.13).

The performance of MapReduce with respect to changing number of mappers m and problem size V, for fixed number of reducers $r = 100$, is shown in Fig. 4.12. Note that in Fig. 4.12 the speedup $\varsigma(1,100)$ is shown, i.e. it is calculated according to (4.70) with respect to the system with $a = 1, b = 100$. When calculated with respect to a single machine (i.e. for $\varsigma(1,1)$) then with changing V the speedups differ from each other by not more than 27%, and the lines nearly overlap in a setting similar to Fig. 4.11. Therefore, we decided to use $\varsigma(1,100)$ in Fig. 4.12 to expose better the influence of V on the performance. It can be observed in Fig. 4.12 that for smaller problem sizes (V =1E13, 1E14) the lines end

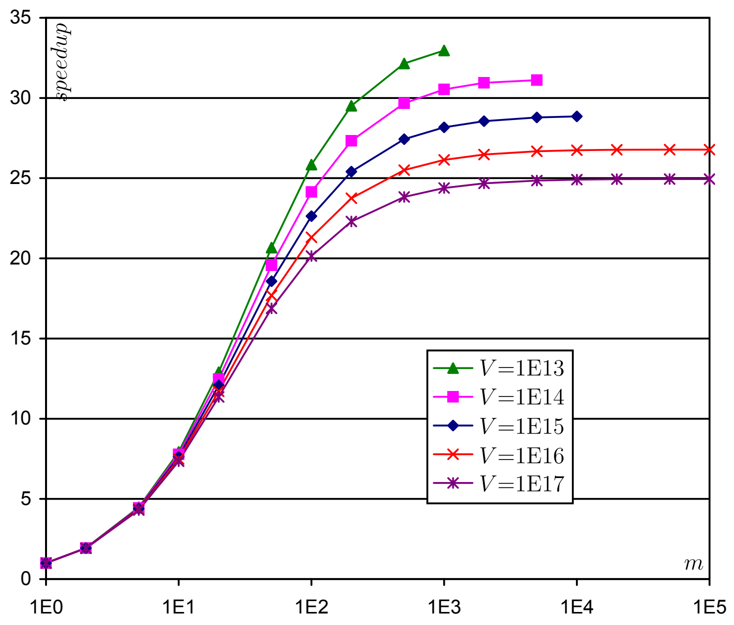


Figure 4.12: Speedup $\varsigma(1, 100)$ vs. the mappers number m and problem size V, for $r = 100$.

before 10000 mappers. This means that a system with a certain big number of mappers cannot be effectively exploited because the load is too small considering the computation startup times and processing rates. With growing problem size V the speedup $\varsigma(1, 100)$ is getting smaller. This is a consequence of the following facts. The mapping time and the communication time grow nearly linearly with V. On the other hand, the complexity function of reducing grows with V faster than linearly. Hence, when V grows, the reducing time grows in relation to the mapping time and the communication time. When V is big, increasing m reduces the schedule length in a smaller degree than when V is small. Consequently, the bigger load sizes V are, the smaller the speedups that are achievable by changing mappers number m.

In Fig. 4.13 the dependence of speedup $\varsigma(1, 1)$ on the multiplicity γ_0 of the results produced by the mappers is shown for changing m and fixed $r = 100$. Let us remind that on average for each input load unit lu the mappers produce $lu\gamma_0$ results. Thus, the bigger γ_0 is, the more data is transferred from the mappers to the reducers. When γ_0 is very small, the reducing time and the time of transfer

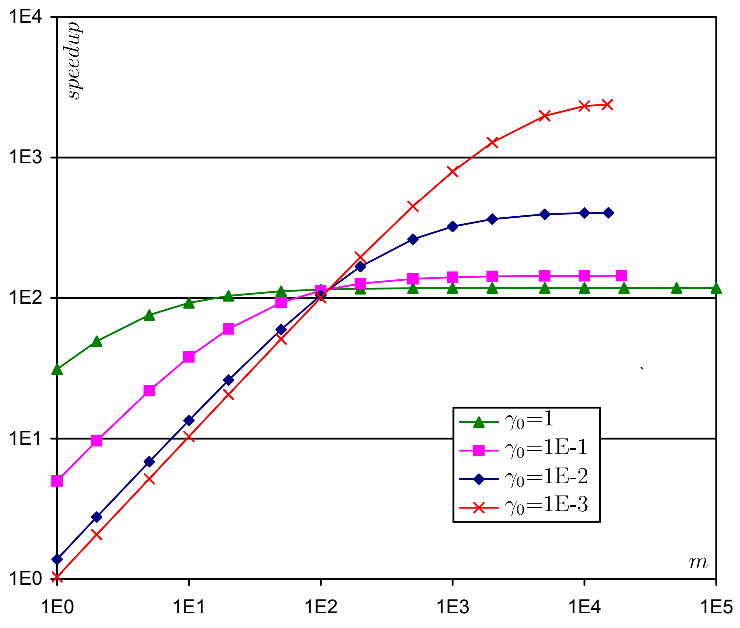


Figure 4.13: Speedup $\varsigma(1,1)$ vs. mappers number m and result multiplicity factor γ_0, for $r = 100$.

from the mappers to the reducers are very short, and the mapping time dominates. On the other hand, for big γ_0 the reducing time dominates in the schedule length. Hence, changes of γ_0 control the speedup in Fig. 4.13 in a two-fold way. The first is the speedup for $m = 1$, and the second is the maximum speedup for big numbers of mappers m. Note that here the speedup is shown for a system with a fixed number $r = 100$ of reducers. Thus, already for $m = 1$ we have some speedup with respect to the single-machine system (where $a = b = 1$ in equation (4.70)). For $m = 1$ the bigger γ_0 is, the more computations shift to the reducers, and the more the $r = 100$ reducers have to do. Consequently, for $m = 1$ (and $r = 100$) bigger γ_0 results in bigger speedup. For very big m the mapping time is already short, and the schedule length is determined by the reducing time. As a result, when γ_0 is big ($\gamma_0 = 1$, or $\gamma_0 = 0.1$) the speedup saturates around the number of reducers r. On the other hand, when γ_0 is small, reducing no longer dominates in the schedule length, while mapping prevails. Then, the mapping time can be reduced by adding mappers, and the speedup is linear in a far wider range of mapper numbers m, up to nearly $m = 1000$ for $\gamma_0 =$ 1E-3. It can be

Table 4.3: Speedup $\varsigma(1,1)$ vs. load unit size lu, for $m = 1000, r = 100$.

lu	1E3	1E4	1E5	1E6	1E7	1E8
A_i	1.011E-5	1.110E-6	2.100E-7	1.200E-7	1.110E-7	1.101E-7
$\varsigma(1,1)$	749.9	308.9	160.9	142.5	140.6	140.4

concluded from Fig. 4.13 that γ_0 is a very important parameter for scalability of MapReduce computations. With small values of γ_0 MapReduce scales well with m, and systems with $m \gg r$ can be effectively used.

Now let us analyze the impact of the load unit size lu on the performance of MapReduce. In Table 4.3 we show the relation between load unit size lu, the resulting mapper processing rate A_i, and speedup $\varsigma(1,1)$ for the system with $m = 1000$, $r = 100$ and all parameters fixed except for lu. It can be seen that the impact of lu is visible only if lu is very small and s_i/lu is greater than or close to $\max\{a^{map}, c_i\}$, i.e. when lu is selected extremely badly. It can be concluded that for reasonable lu sizes ($lu \geq$ 1E6) the impact of lu is small. We have an artifact of big speedup when lu is small ($lu = 1000$). Obviously, bigger speedup for $lu = 1000$ does not mean that the computations are finished in a shorter time. For $lu = 1000$ the cumulative processing rates A_i of mappers are very big, mappers work slowly, and mapping time is big in the whole schedule length. For big lu, cumulative processing rates A_i of the mappers are smaller (processors are faster), the mapping time has a smaller contribution to the schedule length, and using $m = 1000$ mappers reduces the schedule length relatively fewer times than when parameters A_i are big. Consequently, for $m = 1000$ we have better speedup with $lu = 1000$ than for $lu =$ 1E8.

While reading input for mapping, some machines may access their input data from local disks. This results in smaller values of s_i, c_i, what gives some performance advantage. Since the overall influence of s_i is minor (cf. Table 4.3), we analyzed the influence of c_i only. We depict the performance advantages due

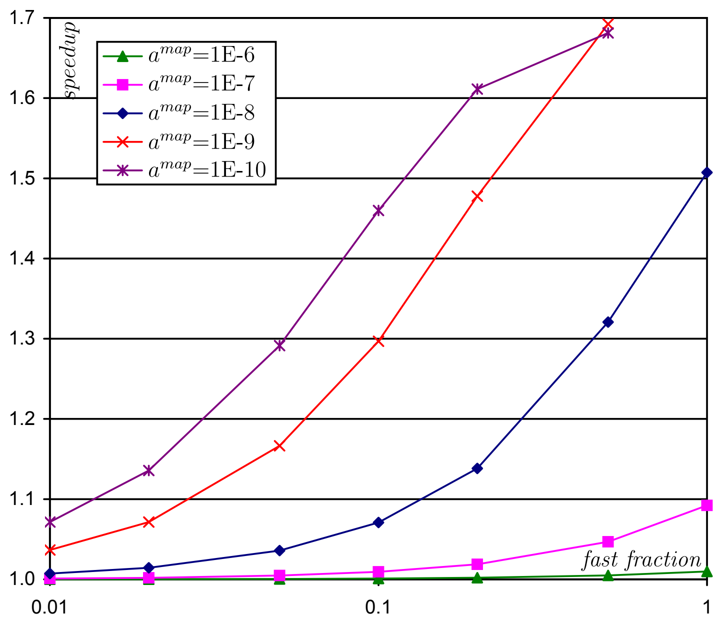


Figure 4.14: Speedup $\varsigma(1000, 100)$ vs. the fraction of fast mappers and microscopic computing rate a^{map}. Standard c_i = 1E-8, fast c_i = 1E-10.

to the local reads in Fig. 4.14. Since the schedule length is dominated by the reducing time, we eliminated it in Fig. 4.14 by showing the speedup with respect to the time by when mapper to reducer communications finish, for $m = 1000$, $r = 100$. To draw Fig. 4.14 we assumed that reading from a local file is 100 times faster than reading from the network. Thus, the standard reading rate is c_i = 1E-8, and the fast reading rate is c_i = 1E-10. On the horizontal axis the fraction of fast processors in the whole pool of mappers is shown. For example, value 0.1 means that 10% of mappers read their inputs locally. It can be seen that the smaller the microscopic mapping rate a^{map} is, the bigger the gain from having some computers reading their inputs faster. If the microscopic computing rate a^{map} is small, then the communication rates are not dominated by the computing rate. On the other hand, another effect can be observed. Note that with changing a^{map}, the load partition and the schedule proportions also change. The load distribution resulting from equations (4.50) and (4.51) is very imbalanced when a^{map} is very small. Precisely, if a^{map} is small, then also A_i are small, values k_i in (4.54) quickly decrease, while values of l_i stabilize. Consequently, the load

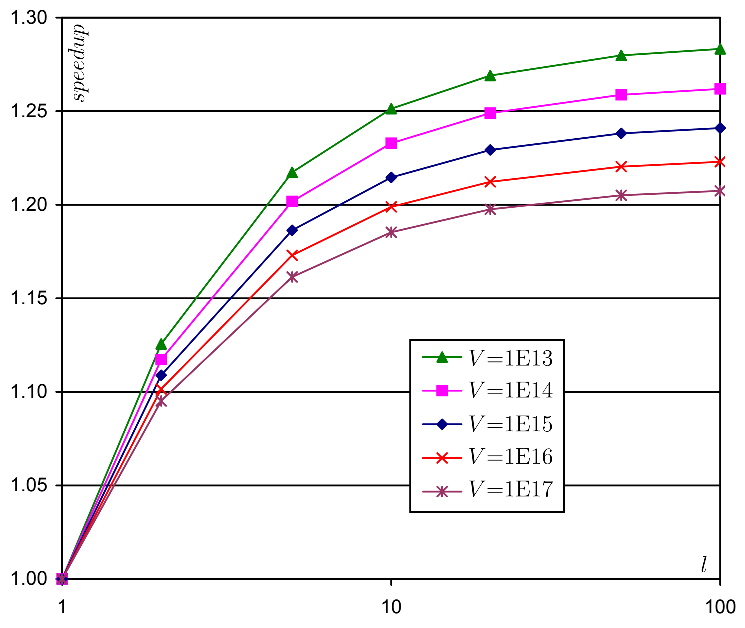


Figure 4.15: Speedup $\varsigma(1000, 100)$ vs. the bisection width l and problem size V.

partitioning is very unequal, and the schedule length is not shorter as one could expect. This is an artifact caused by assuming a particular schedule structure. For example, if a^{map} = 1E-9, or 1E-10 and all mappers (100%) read fast, then the speedup decreases. For clarity, we removed the corresponding two points from Fig. 4.14. Let us note that the speedup from fast local reading is smaller than 0.3% when calculated with included reducing time. This means that the reducing time domination suppresses overall gains from the performance optimizations in other stages of MapReduce.

Finally, let us discuss the impact of the limited bisection width l. As already mentioned, if the bisection width is not bounding, the differences between the load partitioning according to the two methods of scheduling with many reducers are negligible. However, only the second method takes into account the bisection width limit. Thus, in the following we report the results obtained using the latter method. The impact of the bisection width l and problem size V on the speedup $\varsigma(1000, 100)$ is shown in Fig. 4.15. It can be observed that with increasing number of channels the relative speedup stabilizes. This means that new channels from

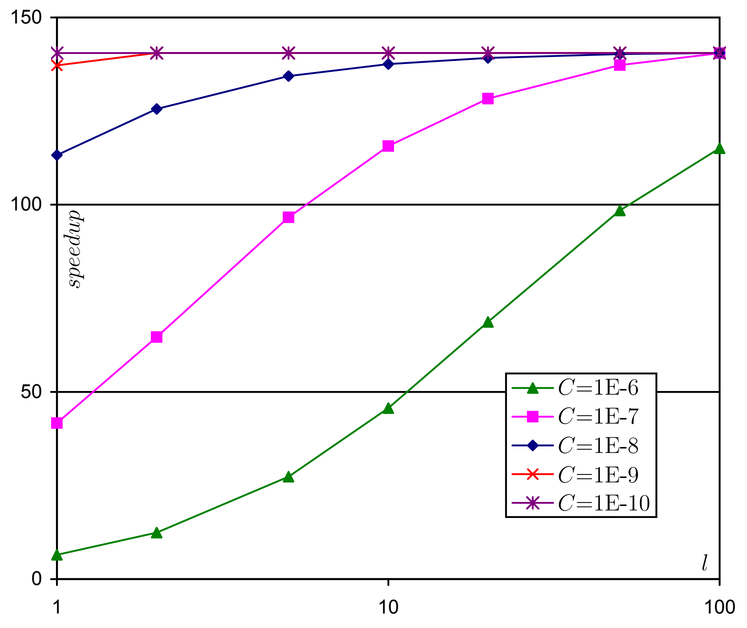


Figure 4.16: Speedup $\varsigma(1,1)$ vs. the bisection width l and communication rate C.

the mappers to the reducers have a gradually decreasing impact on the schedule length which becomes dominated by the mapping and the reducing time. With increasing problem size V the gains from additional channels l, and hence bigger bandwidth, are relatively smaller because the reducing time increases faster than linearly with V, while the communication time increases only linearly with V. Therefore, speedup $\varsigma(1000, 100)$ in Fig. 4.15 decreases with problem size V.

In Fig. 4.16 the impact of the bisection width l and the communication rate C is presented. As it can be seen, the faster the communication is (i.e. the smaller C is), the smaller the impact of the bisection width. Intuitively, this behavior is expected. When the speed of communication from the mappers to the reducers is small, then adding new channels increases the bandwidth and reduces the load transfer time significantly. Hence, for C = 1E-6 the speedup increases with the bisection width l. On the other hand, if the communication speed is high (C = 1E-9, C = 1E-10), then the bandwidth from the mappers to the reducers is sufficient, and adding new channels has no impact on the performance which is determined by the mapping and the reducing time.

4.6 Summary

In this chapter we proposed a mathematical model of MapReduce application. We proposed two scheduling algorithms of mapper to reducer transfers. The first one is mathematically simple, but it does not take into account the bisection width limitations of the data center network. The second method obeys the bisection width limitation, but requires more careful organization of data transfers. When the bisection width was not a limiting factor, then the results obtained from both methods were very similar. Then, we analyzed the influence of the system parameters on the performance of MapReduce computations. The following observations have been made:

- The complexity of reducing operation is higher than the other components of MapReduce computations. This has the following consequences.
  - MapReduce computations scale well with the number of reducers r. However, each reducer produces one output file and a large number of output files may be impractical in some applications.
  - The amount of results $\gamma_0 V$ produced by the mappers is a key parameter controlling the performance of MapReduce, as $\gamma_0 V$ shifts the bulk of the computation cost between mapping and reducing. The bigger $\gamma_0 V$ is, the smaller the contribution of the mapping time, and the less the number of mappers m decides about the performance. If $\gamma_0 \approx 1$, then the number of mappers m need not be greater than the number of reducers r.
  - Reducing time domination can override gains from some optimizations, e.g. from reading load from local disks by some mappers.

  It seems that the reducing operation shall become a bottleneck for the performance of MapReduce.

- Increasing the number of channels l or the communication speed $1/C$ compensate each other because they both increase the bandwidth between the mappers and the reducers.
- The influence of the load unit size lu on the performance is marginal.

5 Multilayer Divisible Applications

In the previous chapter we introduced a mathematical model and two scheduling algorithms for MapReduce applications. A MapReduce application consists of two computational stages: mapping and reducing. It is stated in [23] that the output of a MapReduce is often the input to another such application. Hence, it is justified to treat a chain of such divisible computations processing the load one after another as one application consisting of many stages which we will call *layers*. Therefore, in this chapter we study multilayer divisible applications. In the following, we generalize the mathematical model proposed in the previous chapter to handle multilayer applications. The generalization consists, e.g., in allowing for unequal load distribution in *all* layers. Then, we propose scheduling algorithms which make fewer assumptions on the schedule structure than in the previous chapter. Afterwards, the quality and the features of the schedules generated by our algorithms are analyzed in a series of computational experiments.

5.1 Model of Multilayer Applications

In this section we formulate a mathematical model of multilayer computations. The notation is summarized in Table 5.1.

In a MapReduce application the mapper layer interleaves reading the data from some place in the network with computing, while the reducer layer first obtains the whole input from the mapper layer. Thus, there is a significant difference

Table 5.1: Summary of notation for scheduling multilayer applications.

α_i	the load size processed by mapper i; in bytes;
a_p^{red}, s_p^{red}	the computing rate and the computation startup time for reducers in layer p; in seconds per byte (a_p^{red}) and in seconds (s_p^{red});
β_{ijk}	the size of the load sent in interval $[t_i, t_{i+1})$ from sender j to receiver k;
A	computing rate of a processor executing mapper application;
C	communication rate for reading data by the reducers and storing the final results;
l	bisection width limit, expressed in parallel channels;
m	number of mapper processors;
P_i	processor i;
R	number of reducer layers;
γ_0	mapper result multiplicity fraction;
γ_p	layer p reducer result multiplicity fraction;
δ_{pk}	load fraction received by reducer k in layer p;
r_p	number of reducer processors in layer p;
S	computation startup time, equal for all processors;
T	schedule length;
$[t_i, t_{i+1})$	the i-th communication interval in a given layer;
$\tau_p(x) = a_p^{red} x \log_2 x$	layer p reducer computing time function in load size x;
V	the whole load size, in bytes;

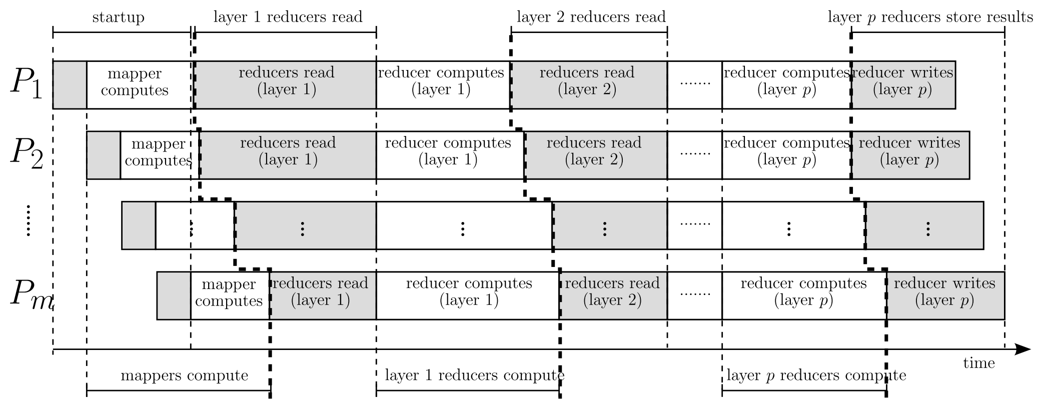


Figure 5.1: General view of multilayer application schedule structure.

in the way of reading the input load. In a sense, the mappers read "ambient" data, while the reducers obtain their inputs from specific mappers taking part in the computations. The mappers data delivery needs no special scheduling, while the reducers load delivery does need it. Similarly, in a multilayer application only the initial layer obtains the load from distributed network locations and can process a part of the data before reading the rest of input. The processors in all the remaining layers obtain data from the preceding layer. They need to read all the data and only after that can they start processing. Let us remind that the reducers have to receive the whole data set before starting the computations because usually sorting is involved. In the following, we will be saying that a multilayer application consists of one mapper layer and $R \geq 1$ reducer layers. The number of mappers will be denoted by m, as in Chapter 4. The number of reducers in layer p, $1 \leq p \leq R$, will be denoted by r_p. For convenience, the mapper layer will be also referred to as layer 0.

A rough schedule structure of multilayer computations is shown in Fig. 5.1. Multilayer computations are divided into $2R + 3$ stages, which partially overlap. The first two stages are the same as in MapReduce computations (see Section 4.2). In the startup stage, the code for all applications is loaded on the processors. The computation startup time of each mapper processor is S.

In the second stage, each of the mappers reads the load, performs computations and saves the results in r_1 local files for r_1 reducers from the first reducer layer. In the previous chapter we perceived mapper i operations as processing load with some average rate A_i. The mapper processing rates were different, resulting from reading the load locally or from remote locations. However, it turned out in Section 4.5 that the influence of differences in the mapper parameter s_i was marginal. The influence of differences in c_i was so small, that to make it more significant we restricted our considerations to the mapper layer only (cf. Fig. 4.14). Otherwise, the influence of heterogeneity in c_i was suppressed by the reducing duration. For multilayer computations, there are many reducer layers and the duration of their computations makes an even bigger contribution in the overall schedule length. It can be expected that also here the influence of the mapper heterogeneity is small. Hence, for simplicity of presentation, we assume here that the processing rates of all mappers are equal to A. As before, by α_i we denote the size of the load assigned to mapper i, and by γ_0 the mapper result multiplicity fraction.

In the following $2R$ stages the reducing operations take place in the consecutive reducer layers. More precisely, in stage number $1 + 2p$, $1 \le p \le R$, the reducers in layer p read load from the mappers (if $p = 1$) or the reducers in layer $p-1$ (cf. Fig. 5.1). The partitioning function used in layer $p-1$ divides the space of key values into r_p not necessarily equal parts. We assume that the partitioning functions used in all layers of computations are, in a sense, sender-independent. The proportions between the amounts of load sent to the receivers in the next layer should be the same for each sender. Were it otherwise, different senders would distribute the data with the same key to different receivers, thus violating the integrity of the results. Therefore, we determine the load distribution in a given layer $p \ge 1$ by the fractions of load which should be received by each computer. Let δ_{pk} denote the fraction of results assigned to reducer k from layer p. The amount of results produced by the reducers in layer p for the input of size α

is $\gamma_p\alpha$. The total amount of load sent to layer p is $V\prod_{i=0}^{p-1}\gamma_i$. Hence, reducer k in layer p receives input of size roughly equal to $\delta_{pk}V\prod_{i=0}^{p-1}\gamma_i$. All reducers read the load with equal rate C. The bisection width limit is denoted by l.

In stages number $2+2p$, $1 \leq p \leq R$, the reducers from layer p sort the input data and perform the computations using a $Reduce_p$ function. For a multilayer application, the function $Reduce_p$ in a sense comprises both reducing operations of the p-th application in the chain and mapping operations of the $(p+1)$-st application in the chain. Because the pattern of communications between the processors from consecutive layers is unknown in general, and a reducer can only start computations after it receives the whole assigned load, we assume that all reducers in layer p start the computations at the same time, after transferring all data between layer $p-1$ and p. We denote the computation startup time of a reducer in layer p by s_p^{red} and its processing rate by a_p^{red}. The computation time of a reducer in layer p, receiving the load of size x, is defined by the function $\tau_p(x) = a_p^{red}x\log_2 x$. As reducer k in layer p receives input of size $\delta_{pk}V\prod_{i=0}^{p-1}\gamma_i$, its execution time is $s_p^{red} + \tau_p(\delta_{pk}V\prod_{i=0}^{p-1}\gamma_i)$.

Finally, in the last, $(2R+3)$-rd stage, the reducers in layer R store the results in the network file system with equal writing rate C. The output of a MapReduce application is usually available in multiple files to be used by other MapReduce applications. However, since we analyze here the whole sequence of such applications, producing a compact set of results, we assume that the final output should be saved in a single file. Still, the scheduling algorithms proposed in the further text can be easily modified to handle other organizations of storing results, e.g. keeping the data on the reducers from the last layer or parallel writing to the network file system, confined by the bisection width limit l (cf. Section 5.2.1).

A scheduling algorithm for multilayer applications has to choose the fractions δ_{pk} of the load received by the reducers in each layer, partition the input load of size V into mapper chunks $\alpha_1, \ldots, \alpha_m$, and schedule mapper to reducer and reducer to reducer communications, as well as storing the results by the reducers from the last layer, so that the total schedule length T is as small as possible.

5.2 Scheduling Algorithms

In this section we propose algorithms for load partitioning and communication scheduling for multilayer computations. The scheduling problem is complex and an instance contains a lot of parameters. Therefore, for clarity, we present the algorithm divided into parts corresponding to the $R+1$ computation layers. This is possible due to the assumption that all reducers in a given layer start the computations at the same moment. The computations of different layers are separated in time from each other and the schedule can be built for one layer at a time.

Note that the optimum load distribution between the mappers (layer 0) depends on the fractions of the load which should be sent to the first layer of reducers (δ_{1k}). Similarly, the load distribution for the reducers in layer p depends on the fractions of load received by the reducers in layer $p+1$. Therefore, we present the schedule construction starting with the last layer and we proceed to layer 0. Consequently, while scheduling computations in layer p, the fractions $\delta_{p+1,k}$ are already known. After presenting the algorithms for separate computation layers, we show how to construct the scheduling algorithm for the whole multilayer application.

5.2.1 Load Partitioning for Reducer Layers

Processing in the last reducer layer may seem different than in the previous layers, because the results are not sent to another set of reducers. Still, storing results sequentially in the distributed file system can be seen as sending data to one more layer consisting of a single processor. Hence, we can define $r_{R+1} = 1$ and

$\delta_{R+1,1} = 1$, and use similar formulas for obtaining the load distribution in all reducer layers. The algorithm finding the load distribution in reducer layer p ($p = 1, \ldots, R$) is described below.

We assume that all reducers in layer p start computations at time $t_0 = 0$. Let us denote by $t_1 \leq \ldots \leq t_{r_p}$ the moments when reducers in layer p finish their computations. As all reducers are identical and start computations at $t_0 = 0$, we can assume that the reducers are ordered by their computation completion times. Thus, reducer k finishes computations at time t_k. Let t_{r_p+1} be the moment when all reducers finished writing their results. The amount of load sent in interval $[t_i, t_{i+1})$ from reducer j in layer p to reducer k in layer $p+1$ will be denoted by β_{ijk}. The following mathematical program computes the optimum values of variables t_i, δ_{pj} and β_{ijk}. Note that the load fractions $\delta_{p+1,k}$ are constants computed in the previous step of the optimization.

$$\text{minimize} \quad t_{r_p+1} \tag{5.1}$$

$$s_p^{red} + \tau_p(\delta_{pi} V \prod_{q=0}^{p-1} \gamma_q) \leq t_i \quad \text{for} \quad i = 1, \ldots, r_p \tag{5.2}$$

$$C \sum_{j=1}^{i} \beta_{ijk} \leq t_{i+1} - t_i \text{ for } i = 1, \ldots, r_p, k = 1, \ldots, r_{p+1} \tag{5.3}$$

$$C \sum_{k=1}^{r_{p+1}} \beta_{ijk} \leq t_{i+1} - t_i \text{ for } i = 1, \ldots, r_p, \; j = 1, \ldots, i \tag{5.4}$$

$$C \sum_{j=1}^{r_p} \sum_{k=1}^{r_{p+1}} \beta_{ijk} \leq l(t_{i+1} - t_i) \quad \text{for} \quad i = 1, \ldots, r_p \tag{5.5}$$

$$\beta_{ijk} = 0 \quad \text{for } j = 1, \ldots, r_p, i = 1, \ldots, j-1, \; k = 1, \ldots, r_{p+1} \tag{5.6}$$

$$\sum_{i=1}^{r_p} \beta_{ijk} = \delta_{p+1,k} \delta_{pj} V \prod_{q=0}^{p} \gamma_q \quad \text{for} \quad j = 1, \ldots, r_p, \; k = 1, \ldots, r_{p+1} \tag{5.7}$$

$$\sum_{j=1}^{r_p} \delta_{pj} = 1 \tag{5.8}$$

In the above formulation, we minimize the length of the schedule from the moment when the reducers in layer p start computations to the moment when they finish communicating with the reducers in layer $p+1$. By inequalities (5.2) reducer i in layer p finishes computations no later than at the moment t_i, for $1 \leq i \leq r_p$. Constraints (5.3)-(5.5) guarantee that all communications fit in the communication intervals together and that the bisection width limit is observed. By (5.6) no reducer sends its results before finishing the computations. Each reducer in layer p sends all its results by (5.7) and the whole load is processed by (5.8). There are $r_p^2 r_{p+1} + 2r_p + 1$ variables and $r_{p+1} r_p^2/2 + r_p(3r_{p+1} + r_p)/2 + 5r_p/2 + 1$ constraints in the given program.

As we mentioned in Section 5.1, the above mathematical program can be easily modified for layer $p = R$ to handle different methods of storing the final results. For example, if parallel writing confined only by the bisection width limit l is possible, it is enough to omit constraints (5.3). If the results are stored locally on each of the reducers from the last layer, constraints (5.3) should be omitted and additionally l should be changed to r_R in constraints (5.5). Alternatively, the latter case may be handled in an even simpler way, by including the results writing in function τ_p, and substituting constraints (5.3)-(5.7) with $t_i \leq t_{r_R+1}$ for $i = 1, \dots, r_R$.

Let us note that constraints (5.2) are not linear because of the form of the function τ_p. In order to provide a practical method of solving (5.1)-(5.8), we transform this program into a linear program. We approximate the function τ_p with a piecewise linear convex function τ'_p. For each interval $[2^y, 2^{y+1})$, for $0 \leq y \leq \log_2 V$, the values $a_y = (\tau_p(2^{y+1}) - \tau_p(2^y))/(2^{y+1} - 2^y)$ and $b_y = \tau_p(2^y) - a_y 2^y$ are calculated. Then, we set $\tau'_p(x) = a_y x + b_y$ for $x \in [2^y, 2^{y+1})$. Thus, the constraints (5.2) are changed to

$$s_p^{red} + a_y \delta_{pi} V \prod_{q=0}^{p-1} \gamma_q + b_y \leq t_i \quad \text{for} \quad i = 1, \dots, r_p, y = 0, \dots, \log_2 V, \tag{5.9}$$

what increases the number of constraints in the mathematical program by $r_p \log_2 V$. The relative error caused by this approximation decreases with growing x. In our experiments (which will be described in Section 5.3), the sizes of load obtained by the reducers are larger than 1E5. For such values the approximation error is less than 1%. Such a range of error is on par with typical accuracy of measuring system parameters such as A, C, a^{red}, s^{red}. Hence, it should be sufficient for practical purposes. If necessary, a better approximation accuracy can be achieved by considering intervals shorter than $[2^y, 2^{y+1})$. As $\tau'_p(x) \geq \tau_p(x)$ for $x \leq V$, the load partitioning obtained for the function τ'_p allows to create a feasible solution with the original function τ_p.

5.2.2 Load Partitioning for Mapper Layer

In this section we analyze scheduling the mapper computations and the communications between the mappers and the first layer of reducers, so that this phase of processing is as short as possible. The optimum load partitioning between the reducers in the first layer, given by fractions δ_{1k}, has been already found by the mathematical program described in the previous section. As the optimum order of finishing computations by the mappers is not known, we will use binary variables z_{ij} $(1 \leq i, j \leq m)$ to define this order. Precisely, if mapper j finishes processing as the k-th of all mappers, then we set $z_{ij} = 0$ for $1 \leq i \leq k-1$ and $z_{ij} = 1$ for $k \leq i \leq m$. Thus, $z_{ij} = 1$ means that mapper j has finished computations by time t_i, and consequently, can send some load in interval $[t_i, t_{i+1})$.

Let $t_1 \leq \ldots \leq t_m$ be the times when the mappers finish their computations. Let t_{m+1} be the moment when the mapper to reducer communications finish. We will denote by β_{ijk} the amount of results read by reducer k from mapper j in time interval $[t_i, t_{i+1})$. Let M denote a big constant. For example, $M \gg mS + (C+A)V$. The optimum load partitioning and the sequence of finishing computations by the mappers can be computed from the following linear program.

$$\text{minimize} \quad t_{m+1} \tag{5.10}$$

$$jS + A\alpha_j \geq t_i - z_{ij}M \quad \text{for} \quad i = 1, \dots, m,\ j = 1 \dots, m \tag{5.11}$$

$$jS + A\alpha_j \leq t_i + (1 - z_{ij})M \quad \text{for} \quad i = 1, \dots, m,\ j = 1 \dots, m \tag{5.12}$$

$$C \sum_{j=1}^{m} \beta_{ijk} \leq t_{i+1} - t_i \text{ for } i = 1, \dots, m, k{=}1, \dots, r_1 \tag{5.13}$$

$$C \sum_{k=1}^{r_1} \beta_{ijk} \leq t_{i+1} - t_i \text{ for } i = 1, \dots, m,\ j{=}1, \dots, m, \tag{5.14}$$

$$C \sum_{j=1}^{m} \sum_{k=1}^{r_1} \beta_{ijk} \leq l(t_{i+1} - t_i) \quad \text{for} \quad i = 1, \dots, m \tag{5.15}$$

$$\beta_{ijk} \leq z_{ij}V \quad \text{for } i = 1, \dots, m,\ j{=}1, \dots, m,\ k = 1, \dots, r_1 \tag{5.16}$$

$$\sum_{i=1}^{m} \beta_{ijk} = \delta_{1k}\gamma_0\alpha_j \quad \text{for} \quad j = 1, \dots, m,\ k = 1, \dots, r_1 \tag{5.17}$$

$$\sum_{i=1}^{m} \alpha_i = V \tag{5.18}$$

$$z_{i+1,j} \geq z_{ij} \quad \text{for} \quad i = 1, \dots, m-1,\ j = 1, \dots, m \tag{5.19}$$

$$\sum_{j=1}^{m} z_{ij} = i \quad \text{for} \quad i = 1, \dots, m \tag{5.20}$$

In the above program, z_{ij} are binary variables, and $\alpha_j, \beta_{ijk}, t_i$ are rational variables. We minimize t_{m+1} which is the length of the schedule until the end of the mapper to reducer communications. Inequalities (5.11) and (5.12) guarantee that the mappers finish computations in the order defined by variables z_{ij}. By (5.13) and (5.14) no mapper or reducer communicates longer than the communication interval. By (5.15) the bisection width limit is observed. Inequalities (5.16) guarantee that no load is sent by a mapper which has not finished computations. Each reducer receives the appropriate amount of results by (5.17) and the whole load is processed by (5.18). Constraints (5.19)-(5.20) ensure that there is one-to-one correspondence between the mappers and moments t_i, $1 \leq i \leq m$, when they finish computations. There are $m^2 r_1 + 2m + 1$ rational variables, m^2

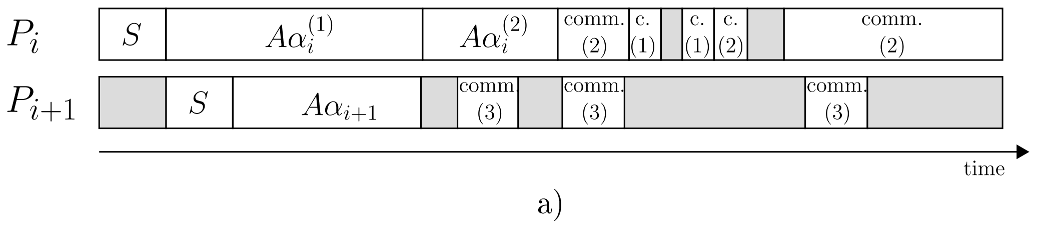
binary variables and $m^2r_1 + 4m^2 + 2mr_1 + m + 1$ constraints in the above linear program.

5.2.3 The Complete Load Partitioning Algorithm

In order to create a load partitioning algorithm for the whole multilayer application, the mathematical programs described above should be put together and solved as one program. However, this leads to many practical difficulties. Firstly, such a program contains $\sum_{p=1}^{R}(r_p^2 r_{p+1} + 2r_p + 1) + m^2 r_1 + 2m + 1$ rational variables and m^2 binary variables. The number of constraints is $\sum_{p=1}^{R}\Big(r_{p+1}r_p^2/2 + r_p(3r_{p+1}+r_p)/2 + 5r_p/2 + 1 + r_p \log_2 V\Big) + m^2 r_1 + 4m^2 + 2mr_1 + m + 1$. Hence, the mathematical program is very large even for very small instances. Secondly, all values α_j and δ_{pk} are variables in the compound mathematical program. Hence, in the constraints corresponding to (5.7) and (5.17) the variables are multiplied and the program is not linear. Thus, it is very hard to solve this program in practice. Therefore, in the computational experiments presented in Section 5.3 an algorithm solving the problem separately for each layer was used. The load distribution obtained in this way for a given layer may be suboptimal from the point of view of the whole multilayer application execution time. Still, it can be used as an approximation of the solutions to start a study of the problem features.

5.2.4 Finishing Mapper Computations Order

The order in which the mappers should finish their computations is unknown in general. This resulted in using binary variables in the mathematical program for the load partitioning in the mapper layer. However, if the startup time S is negligible, then the mappers are not distinguished by the order of starting them. Consequently, the binary variables are not needed in formulation (5.10)-(5.20), which becomes similar to (5.1)-(5.8). Furthermore, we prove below that if the load is distributed equally between the reducers in the first layer, then the



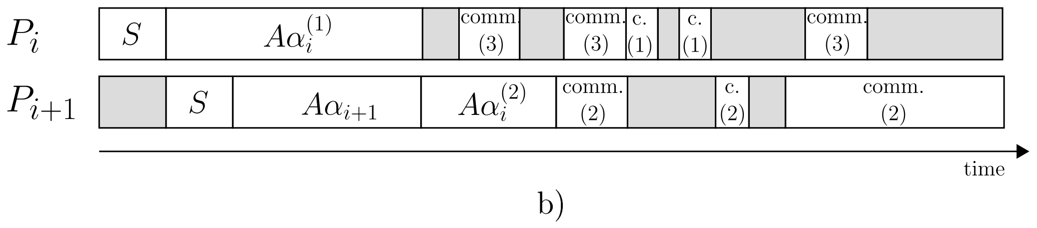


Figure 5.2: Communication pattern in schedules a) σ_1 and b) σ_2. Labeling (i) of the communication intervals is explained in the main text.

mappers should finish computations in the FIFO order. Also in this case the binary variables can be removed from the LP (5.10)-(5.20).

Theorem 5.1. *If $\delta_{1k} = \frac{1}{r_1}$ for $k = 1, \ldots, r_1$, then the FIFO order of finishing mapper computations is optimum.*

Proof. We will show that FIFO is a dominating structure by calculating the amount of load processed in a given time, and by the interchange argument. Assume that in a schedule σ_1 of the mapper phase processor P_{i+1} finishes computations before P_i. The amount of load processed by P_{i+1} in this schedule is α_{i+1}, and the amount of load processed by P_i is $\alpha_i = \alpha_i^{(1)} + \alpha_i^{(2)}$, where $A\alpha_i^{(1)} = S + A\alpha_{i+1}$ (see Fig. 5.2). Since $S > 0$, $A > 0$, we have $\alpha_i^{(1)} \geq \alpha_{i+1}$. We will construct a schedule σ_2 in which processor P_i is assigned load of size $\alpha_i^{(1)}$ and processor P_{i+1} receives load of size $\alpha_{i+1} + \alpha_i^{(2)}$. Therefore, processor P_i finishes computations before P_{i+1} in σ_2. The amounts of load assigned to processors other than P_i and P_{i+1} remain the same as in σ_1. We will show that it is possible to schedule the mapper to reducer communications in σ_2 so that the total length of σ_2 is not greater than the length of σ_1.

Let us choose a set $\mathcal{I}$ of time intervals in which P_i sent load to the reducers in schedule σ_1 and which did not overlap with any communications from P_{i+1} in σ_1, such that $\sum_{I\in\mathcal{I}}|I| = \gamma_0 C(\alpha_i^{(1)} - \alpha_{i+1})$. This operation is possible because the total length of the intervals in which P_i communicates in σ_1, and which do not overlap with communications from P_{i+1}, is equal to at least $\gamma_0 C(\alpha_i^{(1)} + \alpha_i^{(2)} - \alpha_{i+1}) \geq \gamma_0 C(\alpha_i^{(1)} - \alpha_{i+1})$. Note that this set usually may be chosen in many different ways. The total length of the intervals in set $\mathcal{I}$ allows for sending load of size $\alpha_i^{(1)} - \alpha_{i+1}$.

Let us introduce the following labeling of the communication intervals in which at least one of processors P_i and P_{i+1} sends load in σ_1. The intervals from $\mathcal{I}$ receive label 1, the other communication intervals in which P_i sends load get label 2, and all communication intervals containing communications from P_{i+1} receive label 3 (cf. Fig. 5.2).

We schedule the communications in σ_2 so that processor P_i performs all communications in intervals labeled with 1 or 3, and P_{i+1} sends load in intervals labeled with 2. The total length of intervals marked with 2 is $\gamma_0 C(\alpha_i^{(1)} + \alpha_i^{(2)}) - \sum_{I\in\mathcal{I}}|I| = \gamma_0 C(\alpha_{i+1} + \alpha_i^{(2)})$. The total length of intervals labeled with 3 is $\gamma_0 C\alpha_{i+1}$. The intervals marked with 1 and 3 do not overlap. Therefore, processors P_i and P_{i+1} have enough time to send the required amount of data to the reducers. The communications from processors other than P_i and P_{i+1} remain scheduled in the same way as in schedule σ_1. The bisection width limit is still observed, because we only swapped some communication slots between processors P_i and P_{i+1}.

However, further changes in the communication schedule are needed to guarantee that each reducer receives a proper amount of load from processors P_i and P_{i+1}. Note that the previous transformations do not guarantee that the communication schedule for reducers (reading) remains unchanged. Since the load assignments change, some reducer may have to read from two mappers simultaneously. The communication schedule will be changed globally, not only for

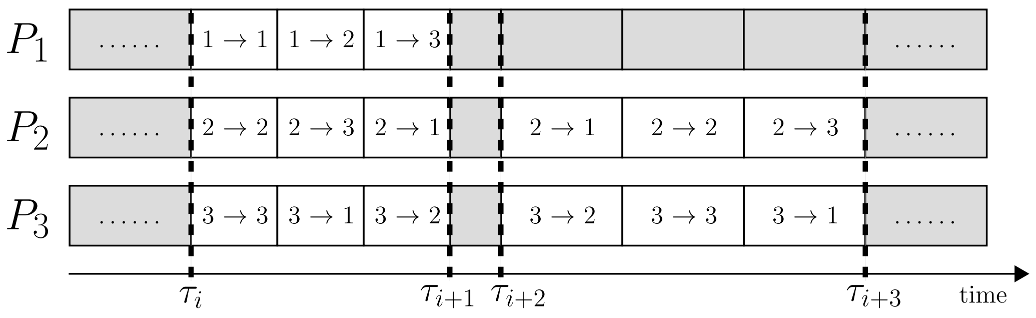


Figure 5.3: Scheduling mapper to reducer computations in σ_2, for $m = 3$, $r_1 = 3$. Notation $j \to k$ stands for: mapper j communicates with reducer k in layer 1.

processors P_i and P_{i+1}. Let us define $t_1 < \ldots < t_q$ as all distinct moments in schedule σ_1 when any mapper to reducer communication starts or finishes. Thus, in each interval $I_i = [t_i, t_{i+1})$ each mapper either communicates all the time with the same reducer, or it does not communicate at all. As the schedule is feasible, in each interval I_i there are at most $\min(l, r_1)$ mappers sending some load.

Let us divide each interval I_i into r_1 subintervals $I_{i1}, \ldots, I_{ir_1}$ of equal length (cf. Fig. 5.3). Let $P'_1, \ldots, P'_{m'}$ be the processors which send some load in interval I_i. Note that necessarily $m' \leq l$ and $m' \leq r_1$. In schedule σ_2 processor P'_j will communicate with reducers $j, j+1, \ldots, r_1, 1, \ldots, j-1$ in intervals $I_{i1}, \ldots, I_{ir_1}$, correspondingly (cf. Fig. 5.3). As $m' \leq r_1$, no reducer reads more than one mapper at a time in schedule σ_2. The bisection width limit is not violated in σ_2 because $m' \leq l$. Furthermore, all mappers send the same amount of load as in schedule σ_1 and each reducer receives the same amount of load from any given mapper. Therefore, the obtained schedule σ_2 is feasible and its length is not larger then the length of σ_1.

Repeating the above procedure for each pair of processors P_j, P_{j+1}, such that P_{j+1} finished computations before P_j, we prove that there exists an optimum schedule in which the mappers finish the computations in the FIFO order. $\square$

Whether the FIFO schedule structure is generally optimum, remains an open question. Some computational experiments indicate that it may be the case.

5.2.5 Scheduling Communications

After solving the mathematical programs given in Sections 5.2.1 and 5.2.2, the amounts of data to be sent between each pair of computers in each time interval are known. A feasible communication schedule can be built for each interval $[t_i, t_{i+1})$ between two layers using a two-stage approach similar to the one used for problem $R|pmtn|C_{max}$ [15, 19, 36]. Then, a schedule for all load transfers can be built by the concatenation of the partial schedules for the consecutive intervals. However, let us observe that here the algorithm for $R|pmtn|C_{max}$ is not sufficient because we have the bisection width constraints, not present in problem $R|pmtn|C_{max}$. Hence, we generalize the former approach. Below we give the scheduling method in detail, and prove its feasibility.

Consider one of the intervals $[t_i, t_{i+1})$ with the load transfers β_{ijk} from sender j to receiver k delivered by formulations (5.1)-(5.8) or (5.10)-(5.20). Let us denote the number of the load senders for the given interval by n_1 and the number of the receivers by n_2, i.e. $n_1 = |\{j : \beta_{ijk} > 0\}|$, $n_2 = |\{k : \beta_{ijk} > 0\}|$. Let $W = [w_{jk}]$ be the $n_1 \times n_2$ matrix defined by $w_{jk} = C\beta_{ijk}/\Delta t$, where $\Delta t = t_{i+1} - t_i$ is the length of the time interval. Thus, $w_{jk} \leq 1$ is the fraction of the length of the current time interval used to transfer load from sender j to receiver k. Note that $\sum_{j=1}^{n_1} \sum_{k=1}^{n_2} w_{jk} \leq l$ by (5.5), (5.15).

Row j of matrix W, corresponding to sender j, will be called critical if $\sum_{k=1}^{n_2} w_{jk} = 1$. Similarly, the k-th column of W, corresponding to receiver k, will be called critical if $\sum_{j=1}^{n_1} w_{jk} = 1$. We will be saying that the bisection width limitation is active for matrix W if $\sum_{j=1}^{n_1} \sum_{k=1}^{n_2} w_{jk} = l$. Let us define a set $\mathcal{F}$ of positive elements of matrix W, containing:

- exactly one element from each critical row or column, and
- at most one element from each non-critical row or column, and
- exactly l elements in total if the bisection width limitation is active for W, or at most l elements in the opposite case.

Thus, $\mathcal{F}$ corresponds to a set of concurrent communications in a feasible schedule. Algorithm 5.1 constructs the optimum schedule for interval $[t_i, t_{i+1})$ by concatenating partial schedules of length $\varepsilon > 0$ for a given set $\mathcal{F}$.

Algorithm 5.1 MULTILAYER-COMMUNICATIONS

```
$\Delta t := t_{i+1} - t_i$
while $\Delta t > 0$ do
    construct set $\mathcal{F}$
    $v^1_{min} := \min_{w_{jk} \in \mathcal{F}} \{w_{jk}\}$
    $v^1_{max} := \max_{j \in \{j' : w_{j'k} \notin \mathcal{F} \text{ for all } k=1,\ldots,n_2\}} \{\sum_{k=1}^{n_2} w_{jk}\}$
    $v^2_{max} := \max_{k \in \{k' : w_{jk'} \notin \mathcal{F} \text{ for all } j=1,\ldots,n_1\}} \{\sum_{j=1}^{n_1} w_{jk}\}$
    if $|\mathcal{F}| < l$ then
        $v^2_{min} := \frac{l - \sum_{j=1}^{n_1} \sum_{k=1}^{n_2} w_{jk}}{l - |\mathcal{F}|}$
    else
        $v^2_{min} := 1$
    end if
    $\varepsilon := \min\{v^1_{min}, 1 - v^1_{max}, 1 - v^2_{max}, v^2_{min}\}$
    for each $w_{jk} \in \mathcal{F}$ do
        schedule communication from sender $j$ to receiver $k$ in interval
        $[t_{i+1} - \Delta t, t_{i+1} - \Delta t + \varepsilon \Delta t)$
    end for
    for each $w_{jk} \in \mathcal{F}$ do
        $w_{jk} := w_{jk} - \varepsilon$
    end for
    $\Delta t := \Delta t(1 - \varepsilon)$
    if $\Delta t > 0$ then
        for each $w_{jk}$ do
            $w_{jk} := w_{jk}/(1 - \varepsilon)$
        end for
    end if
end while
```

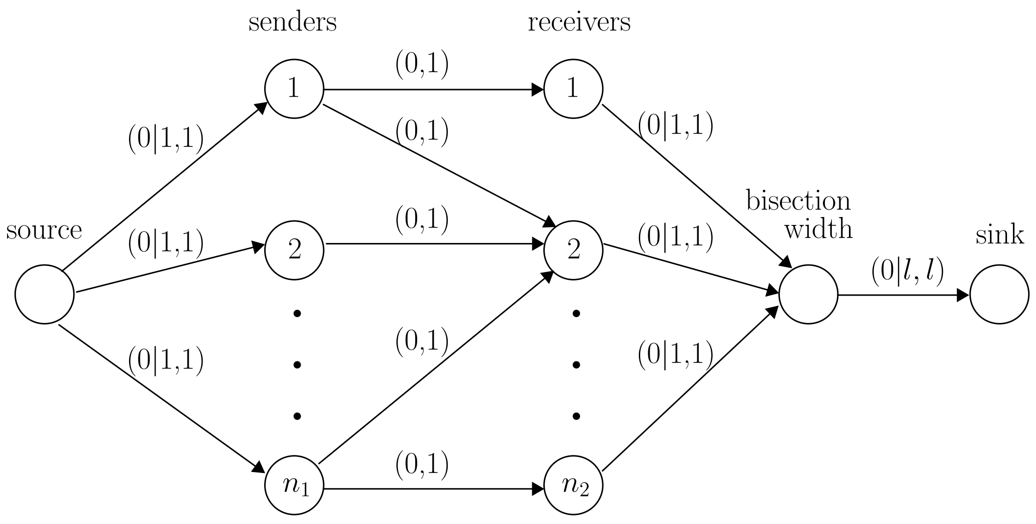


Figure 5.4: Network for finding set $\mathcal{F}$. Arcs are labeled with (lower, upper) bounds. Notation $a|b$ is used for non-critical|critical nodes (see the explanation in the main text).

In this algorithm ε is defined so that the elements of W:

- never become negative by the choice of v^1_{min}, which means that a communication is not performed after the proper amount of load is sent,
- the constraints on the sums of elements of W in any row or column are not violated by the choice of v^1_{max}, v^2_{max}, and hence the critical communications are always executed,
- the constraint on the sum of elements of W is not violated by the choice of v^2_{min}, and the active bisection width limitation is also obeyed.

In each iteration of the while loop either a row or column of W becomes critical, or an element of W is decreased to 0, or the bisection width limit becomes active. Hence, the algorithm consists of at most $n_1 + n_2 + n_1 n_2 + 1$ iterations.

It remains to give an algorithm that finds set $\mathcal{F}$ for a given matrix W. This can be done by using network flow formulation (see Fig. 5.4). Beyond the sink and the source, the network has n_1 nodes corresponding to the senders, n_2 nodes corresponding to the receivers, and a node representing the bisection width limitation. There is an arc between sender j and receiver k if and only if $w_{jk} > 0$. The arcs from the source to the senders, from the senders to the receivers, and from the receivers to the bisection width limitation node have capacities bounded from above by 1. The arcs from the source to the non-critical senders, all arcs from the senders to the receivers, and the arcs from the non-critical receivers to the bisection width limitation have lower bound of capacity equal to 0. For the arcs from the source to the critical senders and for the arcs from the critical receivers to the bisection width limitation node the flows are bounded from below by 1. The arc from the bisection width limit node to the sink has capacity l. If the bisection width limit is active then its flow is bounded from below by l, and by 0 otherwise. For conciseness, in Fig. 5.4 the notation $a|b$ is used for lower bounds on the flow of arcs which lead from or to the non-critical|critical nodes. Finding a feasible flow in the above network is equivalent to finding the set $\mathcal{F}$: the arc from sender node j to receiver node k with positive flow indicates $w_{jk} \in \mathcal{F}$.

We will now prove that a feasible flow, and hence the set $\mathcal{F}$, always exist. Consider a weighted bipartite graph $G = (X \cup Y, E, w)$, such that there are n_1 vertices in X, corresponding to the rows of matrix W and n_2 vertices in Y, representing the columns of W. The set E comprises an edge between vertices $u_j \in X$ and $v_k \in Y$ if and only if $w_{jk} > 0$, and the weight of this edge is equal to w_{jk}. Note that the sum of the weights of all edges incident to any given vertex is not greater than 1, and the sum of all edge weights in G is at most l. We will say that a vertex is critical if it corresponds to a critical row or column in W. Thus, the sum of weights of edges incident to a critical vertex in G is equal to 1. Let us denote the number of critical vertices in X by c_X and the number of critical vertices in Y by c_Y. The subsets of critical vertices in X and Y will be denoted by X_c and Y_c correspondingly. Let G_c denote a subgraph of G induced by the set of critical vertices, i.e. $G_c = G[X_c \cup Y_c]$. Let w_c be the sum of the edge weights in subgraph G_c.

In order to prove that a feasible flow in the network described above always exists, we need to show that there is always a matching M_c in G such that

1) M_c covers all critical vertices,

2) M_c has size at most l, and

3) if the bisection width limit is active, then the size of M_c is exactly l.

We prove it in Theorems 5.4 and 5.5 eventually, but for this we need some preliminary results. First, in Lemma 5.2 we show that a matching satisfying the above condition 1) always exists.

Lemma 5.2. *A matching in G covering all critical vertices always exists.*

Proof. This follows directly from the proof given in [15, 36] for the algorithm solving problem $R|pmtn|C_{max}$. $\square$

Note that if $c_X + c_Y \leq l$ then the above result implies that there exists a matching in G of size at most l covering all critical vertices. For the opposite case, such a matching must contain at least $c_X + c_Y - l$ pairs of critical vertices matched with each other, in order not to violate condition 2). We prove in Lemma 5.3 that a matching consisting of $c_X + c_Y - l$ pairs of critical vertices exists in this case. In Theorem 5.4 we use this fact to prove that there exists a matching satisfying both conditions 1) and 2) given above. Finally, in Theorem 5.5 it is proved that if the bisection width limit is active, then a matching satisfying conditions 1), 2) and 3) exists.

Lemma 5.3. *If $c_X + c_Y > l$, then there exists a matching of size at least $c_X + c_Y - l$ in the graph G_c.*

Proof. The sum of all edge weights in graph G is not smaller than $\sum_{j \in X_c} w_{jk} + \sum_{k \in Y_c} w_{jk} - w_c$. Hence, by (5.5), (5.15)

$$\sum_{j \in X_c} w_{jk} + \sum_{k \in Y_c} w_{jk} - w_c \leq l. \tag{5.21}$$

As the sum of weights of edges incident to a critical vertex in G is equal to 1, we obtain from (5.21)

$$w_c \geq c_X + c_Y - l. \tag{5.22}$$

Now consider the minimum vertex cover of G_c. Since the sum of weights of edges incident to any vertex in G_c is not greater than 1, at least w_c vertices are necessary in the G_c vertex cover. Thus, by (5.22) the minimum vertex cover of G_c has at least $c_X + c_Y - l$ elements. By König's theorem, the size of the maximum matching in G_c is equal to the size of the minimum vertex cover. Hence, there exists a matching of size at least $c_X + c_Y - l$ in G_c. □

Theorem 5.4. *There exists a matching in G of size at most l covering all critical vertices.*

Proof. If $c_X + c_Y \leq l$, the thesis follows from Lemma 5.2. Assume that $c_X + c_Y > l$. Consider the maximum matching M in G_c. Suppose that not all critical vertices are matched by M. We will show that for each critical vertex $v \in Y$ unmatched by M either

a) there exists an even length M-alternating path π_1 starting in v and ending with a non-critical vertex $v' \in Y$ (cf. Fig. 5.5a), or
b) there exists an odd length M-augmenting path π_2 starting from v (Fig. 5.5b).

Suppose that there is no M-alternating path π_1 starting in v and ending with a non-critical vertex $v' \in Y$. Consider the graph G_v induced by the set of all M-alternating paths starting in $v \in Y$. Since no alternating path π_1 ending in non-critical $v' \in Y$ exists, all vertices of G_v contained in Y are critical. Graph G_v contains also all neighbors in X of these vertices. By Lemma 5.2, there exists a matching in G_v covering all its critical vertices from the set Y. As v is the only critical vertex in G_v contained in Y and not matched by M, there exists an M-augmenting path starting in v. In other words, we necessarily have case

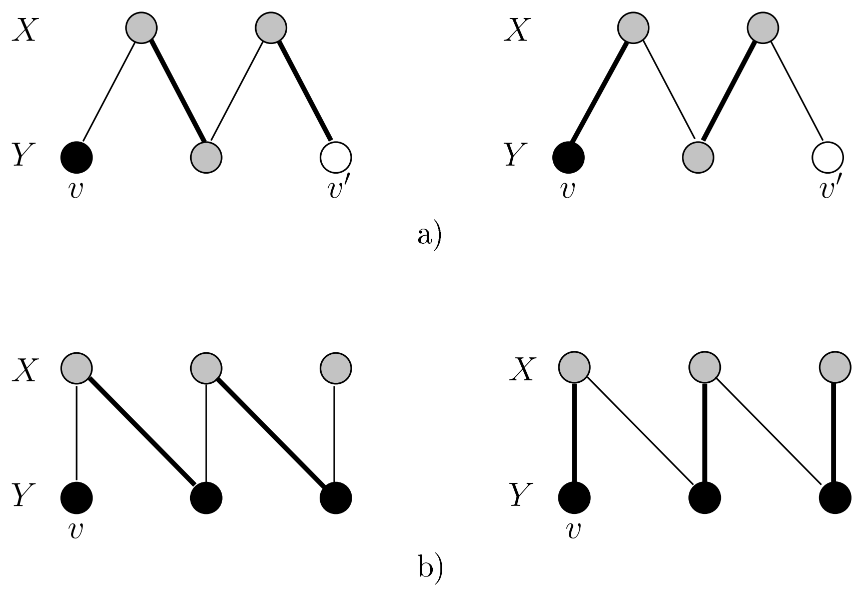


Figure 5.5: Augmenting matching M. Black nodes are critical, non-critical nodes are white, gray nodes may be critical or not. The bold edges are in M. The left figure is the initial matching M, the right figure is the augmented matching. a) Case a - alternating path starting in critical v and finishing in non-critical $v' \in Y$. b) Case b - augmenting path starting in critical v.

b (Fig. 5.5b). Analogous reasoning can be applied to the unmatched critical vertices in X.

Thus, for each unmatched critical vertex v we can find either an M-alternating path π_1 or an M-augmenting path π_2. We set $M' = M \oplus \pi_1$ or $M' = M \oplus \pi_2$ correspondingly, where the symbol $\oplus$ denotes the symmetric difference. In both cases, no critical vertices become unmatched by M', we gain at least one critical vertex matched by M', and the number of edges in M' is increased by at most 1 (see Fig. 5.5a,b). The size of the initial matching M was $e_m \geq c_X + c_Y - l$ by Lemma 5.3. At most $c_X + c_Y - 2e_m$ critical vertices in G were unmatched in M. Thus, we obtain a matching covering all critical vertices, with at most $e_m + c_X + c_Y - 2e_m \leq l$ edges. □

The last thing to prove is the existence of a matching satisfying conditions 1), 2) and 3) in the case when the bisection width limit is active.

Theorem 5.5. *If the sum of edge weights in G is equal to l, then there exists a matching in G of size l, covering all critical vertices.*

Proof. We can apply the same procedure as in the proof of Theorem 5.4 to obtain a matching M of size at most l, covering all critical vertices. The sum of weights of the edges incident to any vertex in G is at most 1. Hence, if the sum of all edge weights in G is l, then the minimum vertex cover in G contains at least l vertices. By König's theorem, the size of the maximum matching in G is at least l. Therefore, if $|M| < l$, we can further augment the matching M until it has exactly l edges. $\square$

5.3 Computational Experiments

In this section the influence of the instance parameters on the schedules for multilayer applications is analyzed. Unless written to be otherwise, the reference system configuration used in the experiments is the following. There are $R = 2$ reducer layers. Each layer consists of 5 processors ($m = r_1 = r_2 = 5$). The size of the test instances is a result of both high complexity of the scheduling algorithm and the achievable numerical precision. The reducer layers are characterized by parameters $s_p^{red} =$ 1E-2, $a_p^{red} =$ 1E-7, $\gamma_p = 0.1$ (for $p = 1, 2$), The mapper parameters are $A =$ 1E-7, $S = 1$ and $\gamma_0 = 0.1$. The communication rate is $C =$ 1E-8, and the bisection width limit $l = 5$ is not restricting the communication. The initial amount of load is $V =$ 1E15.

5.3.1 Speedup of Multilayer Applications

In Chapter 4 we analyzed the influence of the number of processors in the two computational layers and the other system parameters on the relative length of the obtained schedules. Qualitatively, the results obtained for the mathematical model presented in this chapter do not differ much from the previous ones. As an

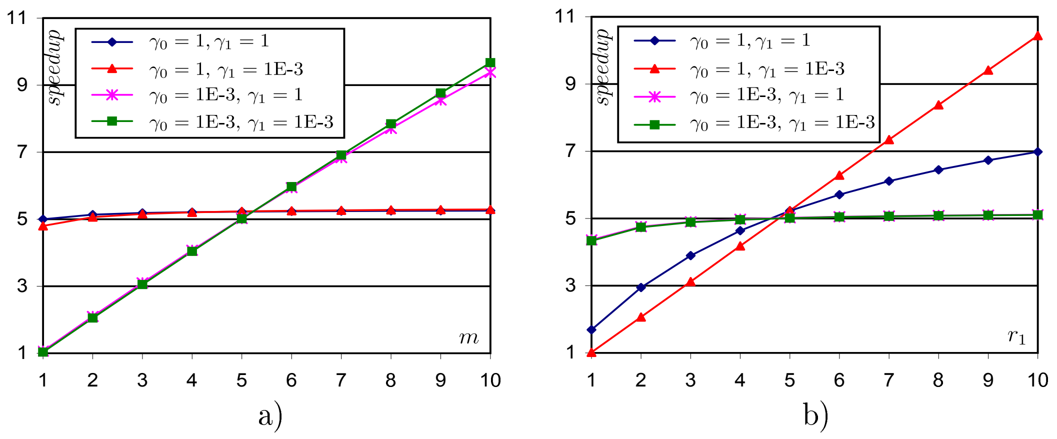


Figure 5.6: Speedup for different γ_0, γ_1, a) vs. m, for $r_1 = r_2 = 5$, b) vs. r_1, for $m = r_2 = 5$.

example, we present in Fig. 5.6 the speedup for changing m and r_1 (in relation to the system with $m = r_1 = r_2 = 1$). We analyzed cases with big ($\gamma_p = 1$) and small ($\gamma_p =$ 1E-3) load multiplicity fractions in each layer. It turned out that the value of γ_2 has almost no impact on the speedup. This can be explained by the fact that γ_2 influences only the time needed to store the final results, which is very short in comparison to the whole schedule length. Therefore, we present only the instances with $\gamma_2 = 1$ in Fig. 5.6.

As it could be expected, the fractions γ_0 and γ_1 influence the performance of the application. It can be seen that the application scales well with the mapper number m if γ_0 is small (see Fig. 5.6a). In this case, the reducers receive little load and do not dominate in the computations. On the other hand, if γ_0 is big, then the number of mappers has a small impact on the speedup because the bulk of computations takes place in layer 1, and the application scales better with the number of reducers r_1 (cf. Fig. 5.6a and Fig. 5.6b). The range of the speedup is determined not only by γ_0, but also by γ_1. If γ_1 is big, then the reducers in the second layer receive big load and their contribution to the schedule length is comparable with the first layer. On the other hand, if γ_1 is small, then the execution time of the whole application is dominated by the first reducer layer. Then, r_1 has the greatest influence on the schedule length. The influence of r_2

on the performance of the application is significant for the speedup only if both γ_0 and γ_1 are big. We do not show these results here because they follow the pattern of Fig. 5.6a,b.

In the previous chapter we presented scheduling algorithms for 2-layer applications. The algorithms for $r_1 > 1$ assumed a specific communication schedule structure, which could be an obstacle to finding the optimum solution. In particular, the amounts of load assigned to different processors could be biased. In this chapter we relaxed the assumptions on the communication pattern, as well as on the load partitioning in the reducer layers. Therefore, in the further text we concentrate on the load distribution between the processors in a given layer.

5.3.2 Load Distribution between Reducers

This section is dedicated to analyzing the load distribution in the reducer layers. The number of the reducers in the first layer is set to $r_1 = 10$. Since $r_2 = 5$, the bisection width limit $l = 5$ is not restricting the communication between the first and the second layer. We present the load distribution in reducer layer p as the load fractions received by the consecutive processors relative to the equal distribution, i.e. the values $\delta_{pk}/(1/r_p)$. As the fractions of load received by the reducers in the first layer depend on the load sent to the reducers in the next layer, we start our study with the second layer.

The values of load fractions $\delta_{2,k}$ for different values of communication rate C are shown in Fig. 5.7. Let us remind that according to the model from Section 5.1, the reducers in a given layer start computations at the same moment and finish them in the order of their indices. Hence, the fractions $\delta_{2,k}$ are always nondecreasing. It can be seen in Fig. 5.7a that for very fast communication the load distribution in the second layer of reducers is very equal. For $C =$ 1E-7 the differences are more significant than for smaller values of C, but the fractions $\delta_{2,k}$ still grow nearly linearly. This can be explained by the fact that for very fast

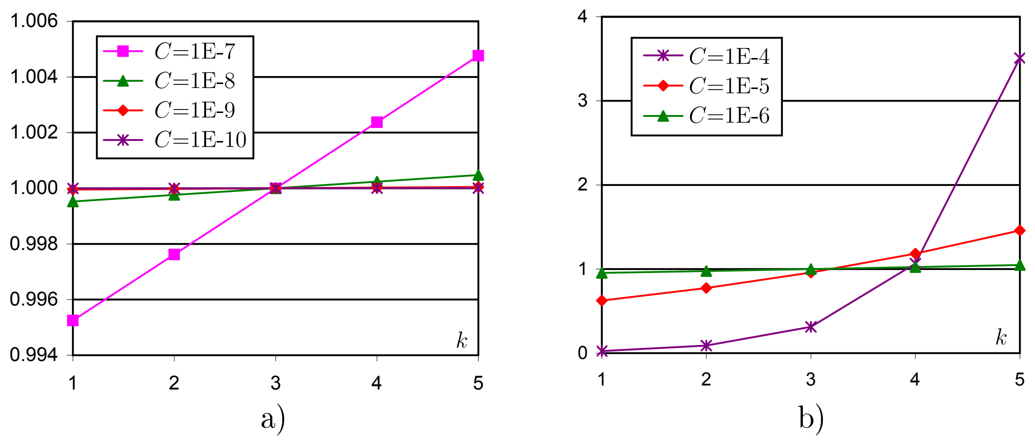


Figure 5.7: Relative load fractions $\delta_{2,k}/(1/r_2)$ vs. communication rate C, a) fast communication (small C), b) slow communication (big C).

communication, the time of computations dominates in the schedule length for a given layer. Therefore, to make this time shorter, the load should be divided equally, so that the computations finish around the same time on all processors. The situation becomes different for slow communication (cf. Fig. 5.7b). For very big values of C (C = 1E-4, C = 1E-5) the time needed for storing the results dominates in the schedule length. Thus, it is profitable to start communications from some reducers very early, while other processors are still computing. This leads to great inequalities in the load distribution between the reducers. The first processors receive very small load, while the last reducer has to process more than a half (for C = 1E-5) or even more than 90% (for C = 1E-4) of all data.

The load distribution between the processors in the first reducer layer is presented in Fig. 5.8. As in the second layer, the distribution is balanced for fast communication and very unequal for slow communication. Another interesting phenomenon can be observed. For fast communication, the reducers can be divided into two groups comprising 5 processors each (see Fig. 5.8a). The processors in a given group receive similar amounts of load. As there are 5 processors in layer 2 which receive data from the reducers in the first layer, we infer that the processors in a given group can use a similar communication pattern, but communicate with the reducers from the second layer in different order. Precisely,

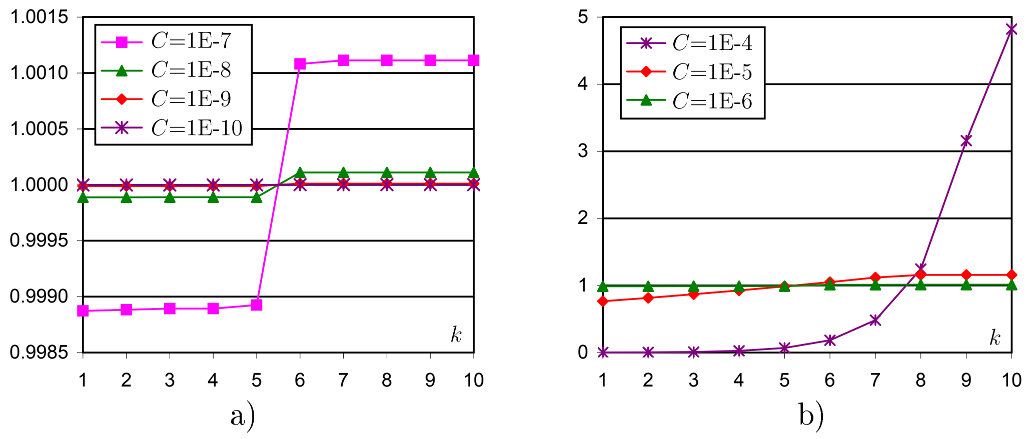


Figure 5.8: Relative load fractions $\delta_{1,k}/(1/r_1)$ vs. communication rate C, a) fast communication (small C), b) slow communication (big C).

for very fast communication, the reducers in layer 1 constitute rectangular blocks of computations of roughly the same time on r_2 processors. The processors in a given group finish computations around the time when the previous group finished sending the results to the next layer of reducers. Thus, it seems that the optimum communication pattern in this case is similar to the first method of scheduling MapReduce computations proposed in Chapter 4. The difference is that in Chapter 4 we synchronized the computations and communications of consecutive processors, and not consecutive groups of several workers. The inequalities in the load distribution between the processors in a given group become larger when C gets larger. This can be caused by a more unequal load distribution in the second reducer layer. It can be seen in Fig. 5.8b that in the case of slow communication the groups of 5 processors cannot be distinguished anymore. It can be inferred that the pattern of communications is very different for slow communications.

In the test instances described above the number r_1 was divisible by r_2. Thus, for fast communication the reducers in the first layer could be divided into groups, each of which comprised r_2 computers. In Fig. 5.9 we show the load distribution in the first reducer layer for $r_1 = 10$ and $r_2 = 4$. In this case, one group of size 4 and three groups of size 2 can be distinguished for $C = 1$E-7, and groups of sizes

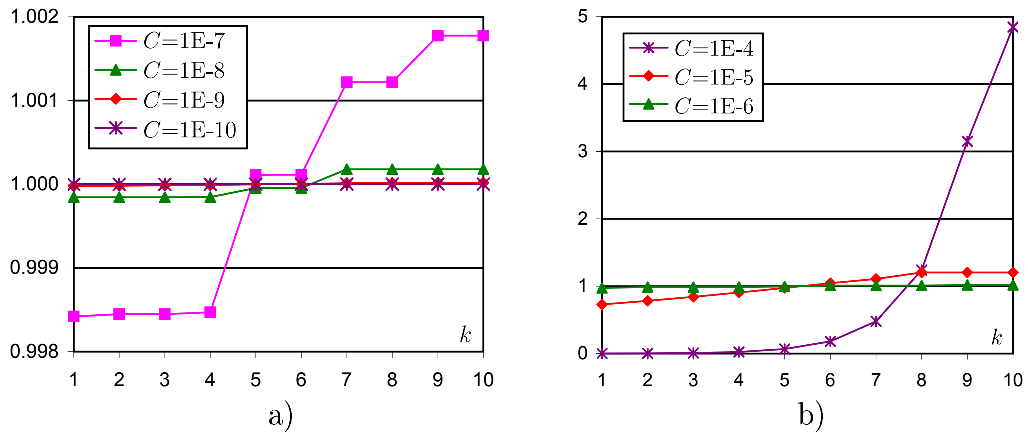


Figure 5.9: Load fractions $\delta_{1,k}$ vs. communication rate C for $r_1 = 10$, $r_2 = 4$, a) fast communication (small C), b) slow communication (big C).

4, 2, 4 are visible for C = 1E-8. Thus, there is no simple repetitive pattern in the load distribution, which could be easily generalized to any system configuration. Additionally, the number and the sizes of the obtained groups depend on parameter C. This suggests that in the systems with fast communication it may be profitable to use the numbers of reducers r_1 divisible by r_2. In such a case, the assumption that the processors are divided into r_1/r_2 groups can be used to base the scheduling algorithm on a predetermined load partitioning pattern. This would result in the design of simpler and faster scheduling heuristics.

The amount of time necessary to send the load from one reducer layer to another depends not only on parameter C, but also on the bisection width limit l. Let us remind that the load distribution in the last layer does not depend on l, as the results are stored sequentially. In Fig. 5.10 we present the load distribution in the first reducer layer for different values of l. The value C = 1E-8 in Fig. 5.10 can be considered fast communication. The results shown in Fig. 5.10a confirm that the groups of processors receiving similar amounts of load are connected with the number of processors which can communicate at the same time. When $l = 2$, groups of 2 processors can be observed in Fig. 5.10a. For $l = 1$ each processor constitutes a separate group. Similarly, for $l = 5$ five-processor groups can be observed. If r_1 is not divisible by l (Fig. 5.10b), then no clear groups of

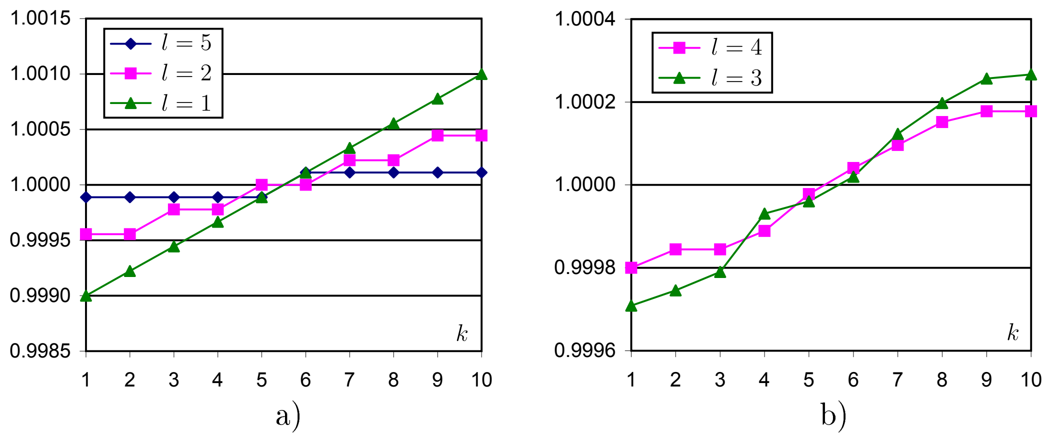


Figure 5.10: Relative load fractions $\delta_{1,k}/(1/r_1)$ vs. the bisection width limit l, a) r_1 divisible by l, b) r_1 not divisible by l.

processors can be distinguished. It seems that the pattern of communication is not so simple anymore.

5.3.3 Load Distribution between Mappers

In this section we analyze the load distribution in the mapper layer. In the following simulations we assumed the FIFO structure of the mapper computations. Whether it is generally optimum, remains an open question. However, we chose this structure for several practical reasons. The startup times are short in relation to the whole schedule, and hence, the order of starting processors and startup procedure have small impact in differentiating the processors. Mixed integer linear programming is computationally hard, and only very small instances can be solved to optimality in acceptable time. The choice of the FIFO order allowed us to solve larger instances of the problem: the number of mappers was $m = 50$ in the experiments presented in this section.

In the first series of experiments we analyzed the load distribution between the mappers for relatively small startup times $S = 1$. The results of the experiments with changing C and l are presented in Fig. 5.11. The load distribution in the mapper layer was shown as the fractions $\alpha_j/(V/m)$. As could be intuitively expected, the results are very similar to the distributions for the first reducer

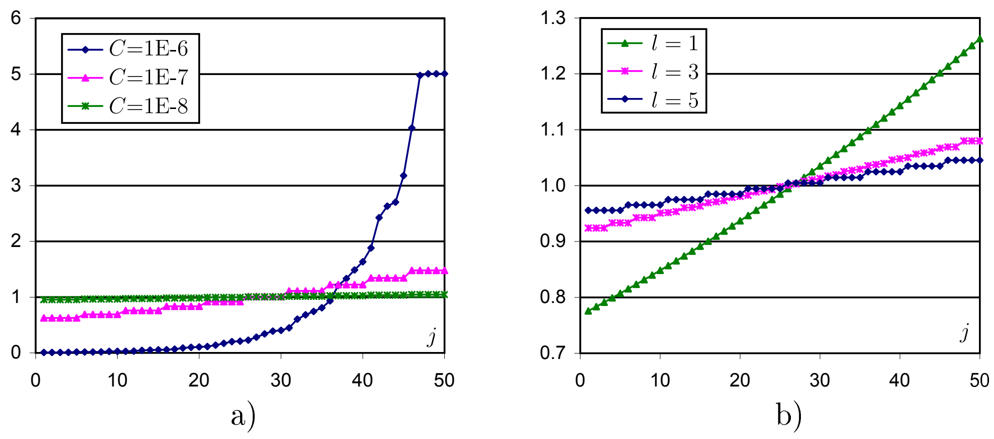


Figure 5.11: Mapper load fractions $\alpha_j/(V/m)$ for $S = 1$, a) vs. C, b) vs. l.

layer. The difference between the computations in the mapper and the reducer layers is the presence of startup time S, but its influence was almost negligible in this set of experiments. Thus, we observed the same phenomena as were described in Section 5.3.2. For example, groups of $\min\{r_1, l\}$ mappers with nearly equal load assignments can be distinguished when C is small and m is divisible by $\min\{r_1, l\}$. When C is big, the majority of the load is processed by the machines activated as the last ones.

In order to better expose the differences between the mapper and the reducer layers we increased the startup time S to 1E4. The results of the experiments with changing communication rate C are shown in Fig. 5.12. For fast communication we observed a qualitative difference in the load distribution (see Fig. 5.12a). The mapper loads are now generally decreasing. Similarly to the reducer layer, the mappers can be divided into groups of consecutive 5 processors (we have $r_1 = l = 5$). However, the fractions of load obtained by the processors in a given group are far from equal. The difference between the amounts of load received by two consecutive processors from the same group is about 1E11 for $C =$ 1E-8, 1E-9, 1E-10. The time needed to process load of this size on a mapper is 1E4, which is equal to the startup time S. Thus, the processors in a given group receive such amounts of data that they finish computations at approximately the same time.

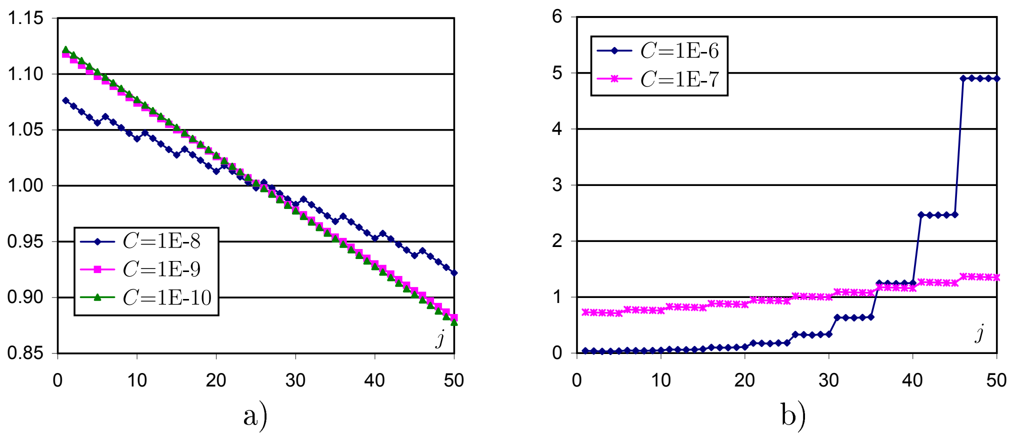


Figure 5.12: Mapper load fractions $\alpha_j/(V/m)$ for S = 1E4 vs. C, a) fast communication (small C), b) slow communication (big C).

Then, they use the available communication channels to send the results to the reducers. The first computer in the next group of mappers receives such amount of load that it still performs computations while all communication channels are used by the previous group. For C = 1E-8 this means that the load obtained by the first processor in a given group is larger than the load assigned to the last processor in the preceding group. Hence comes the characteristic saw-like pattern in Fig. 5.12a.

When the communications gets slower (C is bigger) the situation changes and is more similar to the reducer layer distribution (cf. Fig. 5.12b). The sizes of load assigned to consecutive mappers are increasing and the groups of 5 mappers receiving similar amounts of load can be seen. This can be explained by the fact that for slow communication the startup time S = 1E4 is not significant in the schedule length. Hence, the results obtained here are similar to the reducer layers.

The load distributions in the mapper layer for different bisection width limits l are presented in Fig. 5.13. When the number of mappers m is divisible by l (Fig. 5.13a), then groups comprising l mappers can be observed again. Different tendencies can be seen for different values of l. When $l = 5$, the amount of load assigned to the mappers in a given group and the amounts of load obtained by

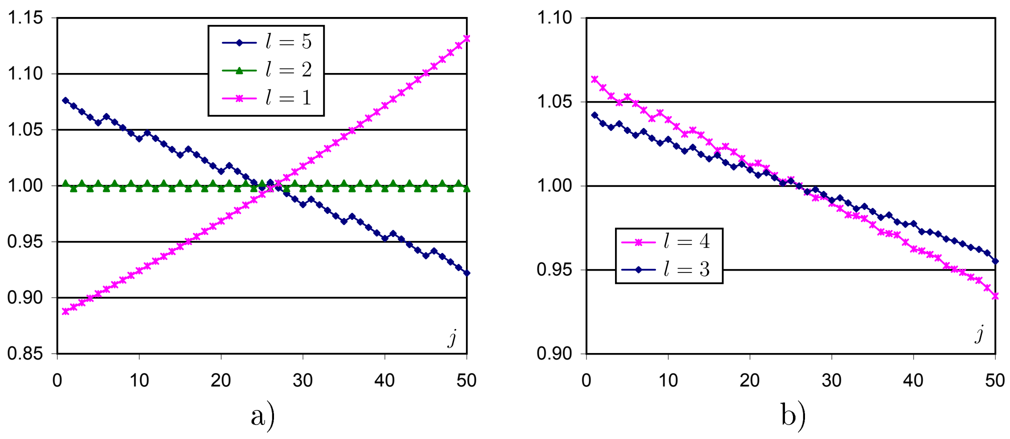


Figure 5.13: Mapper load fractions $\alpha_j/(V/m)$ for $S = 1\text{E}4$ vs. l, a) m divisible by l, b) m not divisible by l.

consecutive groups are decreasing. For $l = 2$, the first mapper in a given pair receives more load than the second, but there are no visible differences between the groups. For $l = 1$, the load sizes assigned to the mappers are increasing. Although these three patterns seem different, they are in fact instantiation of the same type of communication organization. The l mappers in a given group finish the computations around the same moment. The mappers from each following group finish the computations when the preceding group finishes sending results and the communication channels can be used by the next group of processors.

Such an organization of processing is not possible when m is not divisible by l (Fig. 5.13b). In this case, the groups of l mappers can be seen at the beginning of the mapper sequence, but for the mappers activated later, the group pattern gradually disappears. Thus, the schedule starts with blocks of l mappers, which gradually dismantle to single-mapper "groups".

5.4 Summary

In this chapter we introduced multilayer divisible applications and proposed scheduling algorithms for all computation layers. The order in which the mappers should finish their computations was analyzed. We proved that the FIFO order is optimum in some special cases.

The load distribution between the processors in different computation layers was analyzed. It turned out that it is to a large degree determined by the computation rate C. When C is small, the computation time dominates the schedule length. Hence, the load distribution is rather balanced, so that all processors finish computing around the same time. If C is very big, the communication time dominates the schedule length and it is profitable to start the communications as soon as possible. This leads to big inequalities in the load distribution.

Another important parameter influencing the load distribution is the bisection width limit l. If the number of senders (mappers or reducers) is divisible by l and the communication is fast, then the computers form groups of size l. The computers in a given group finish the computations around the same moment and send their results during the same time interval, using the l available communication channels. The next group finishes the computations almost exactly when the communication channels are released. If the number of senders is not divisible by l, the groups of l computers are visible at the beginning of the sender sequence, but then they disappear. Thus, it may be profitable to use the numbers of mappers or reducers divisible by l. In this case, due to the additional information about the schedule structure, faster scheduling algorithms may be devised.

6 Summary and Conclusions

In this work we analyzed scheduling divisible loads in heterogeneous distributed systems. First, we studied classical single-round divisible load scheduling problems in star networks. We proposed fully polynomial time approximation schemes for the problems with infinite bandwidths. This result complements computational complexity analysis of this problem. The obstacles in approaching the more general problem with finite bandwidths were presented. Future research may include further analysis of the approximability of this problem. Another direction is the construction of approximation algorithms for single-round divisible load scheduling with limited memory.

The second problem analyzed in this work was multi-round scheduling in star networks. Such an organization of communications allows for decreasing the initial communication delays and for taking into account the practical memory limitations. We proposed a genetic algorithm solving the corresponding scheduling problem and used it to perform an experimental study of the properties of the near-optimum solutions. Analytically obtained results were also provided. The results were used to construct fast and simple heuristics for our scheduling problem. We analyzed them experimentally and compared with the algorithms known from the earlier literature. The heuristics proposed in this work obtained substantially better results in much shorter time. Some classes of inefficient heuristics were singled out. These results can be used as a base for future research on approximation algorithms for the analyzed problem.

The following parts of this thesis were dedicated to scheduling divisible MapReduce and multilayer computations. Scheduling divisible loads with precedence constraints was not studied before. We proposed mathematical models and scheduling algorithms for the analyzed organization of computations. On the basis of a series of computational experiments we analyzed the influence of the system parameters on the performance of MapReduce applications and the structure of the schedules. These results can be helpful in constructing effective computer networks as well as in designing efficient MapReduce and multilayer divisible applications in practice. The analysis of the load distribution in multilayer computations showed that the communication parameters in a great extent influence the amounts of load which should be assigned to the particular processors. We pointed out that adjusting the number of computers used for processing to other system parameters (e.g. the bisection width limit) may lead to simplifications in the scheduling algorithms and in the structure of the optimum schedule. This fact can be useful both for designing multilayer applications and for controlling their execution. A future research direction is better modeling of MapReduce and multilayer applications. Several problems posed in this work, like finding the optimum order of finishing mapper computations or constructing fast approximation algorithms for scheduling multilayer applications, are also open areas for further study.

Bibliography

[1] R. Agrawal, H.V. Jagadish, Partitioning Techniques for Large-Grained Parallelism, IEEE Transactions on Computers 37 (1988) 1627-1634.

[2] T. Badics, E. Boros, Minimization of Half-Products, Mathematics of Operations Research 23(3) (1988) 649-660.

[3] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, Y. Yang, Scheduling Divisible Loads on Star and Tree Networks: Results and Open Problems, IEEE Transactions on Parallel and Distributed Systems 16 (2005) 207-218.

[4] O. Beaumont, A. Legrand, L. Marchal, Y. Robert, Independent and Divisible Tasks Scheduling on Heterogeneous Star-Shaped Platforms with Limited Memory, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, Technical Report 2004-22 (2004).

[5] O. Beaumont, A. Legrand, Y. Robert, Scheduling Divisible Workloads on Heterogeneous Platforms, Parallel Computing 29(9) (2003) 1121-1152.

[6] J. Berlińska, Fully Polynomial Time Approximation Schemes for Scheduling Divisible Loads, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Waśniewski (Eds.), Parallel Processing and Applied Mathematics: 8th International Conference PPAM 2009, Part II, Lecture Notes in Computer Science 6068 (2010) 1-10.

[7] J. Berlińska, M. Drozdowski, Dominance Properties for Divisible MapReduce Computations, Institute of Computing Science, Poznań University of Technology, Research Report RA-09/09 (2009), http://www.cs.put.poznan.pl/mdrozdowski/rapIIn/ra0909.pdf.

[8] J. Berlińska, M. Drozdowski, Heuristics for Divisible Loads Scheduling in Systems with Limited Memory, Proceedings of the 4th Multidisciplinary International Scheduling Conference: Theory & Applications (2009) 321-329.

[9] J. Berlińska, M. Drozdowski, Heuristics for Multi-Round Divisible Loads Scheduling with Limited Memory, Parallel Computing 36(4) (2010) 199-211.

[10] J. Berlińska, M. Drozdowski, Scheduling Divisible MapReduce Computations, Journal of Parallel and Distributed Computing 71(3) (2011) 450-459.

[11] J. Berlińska, M. Drozdowski, M. Lawenda, Multi-Installment Divisible Loads Scheduling in Systems with Limited Memory, Institute of Computing Science, Poznań University of Technology, Research Report RA-07/08 (2008), http://www.cs.put.poznan.pl/mdrozdowski/rapIIn/ra0708.pdf.

[12] J. Berlińska, M. Drozdowski, M. Lawenda, Experimental Study of Scheduling with Memory Constraints Using Hybrid Methods, Journal of Computational and Applied Mathematics 232 (2009) 638-654.

[13] V. Bharadwaj, D. Ghose, V. Mani, Optimal Sequencing and Arrangement in Distributed Single-Level Tree Networks with Communication Delays, IEEE Transactions on Parallel and Distributed Systems 5(9) (1994) 968-976.

[14] V. Bharadwaj, D. Ghose, V. Mani, T.G. Robertazzi, Scheduling Divisible Loads in Parallel and Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA, (1996).

[15] J. Błażewicz, W. Cellary, R. Słowiński, J. Węglarz, Scheduling Under Resource Constraints - Deterministic Models, Annals of Operations Research 7 (1986).

[16] J. Błażewicz, M. Drozdowski, Scheduling Divisible Jobs on Hypercubes, Parallel Computing 21 (1995) 1945-1956.

[17] J. Błażewicz, M. Drozdowski, Distributed Processing of Divisible Jobs With Communication Startup Costs, Discrete Applied Mathematics 76 (1997) 21-41.

[18] J. Błażewicz, M. Drozdowski, M. Markiewicz, Divisible Task Scheduling - Concept and Verification, Parallel Computing 25 (1999) 87–98.

[19] J. Błażewicz, K. Ecker, E. Pesch, G. Schmidt, J. Węglarz, Scheduling Computer and Manufacturing Processes, Springer, Heidelberg (1996).

[20] Y.-C. Cheng, T.G. Robertazzi, Distributed Computation with Communication Delay, IEEE Transactions on Aerospace and Electronic Systems 24 (1988) 700-712.

[21] Y.-C. Cheng, T.G. Robertazzi, Distributed Computation for a Tree Network with Communication Delays, IEEE Transactions on Aerospace and Electronic Systems 26 (1990) 511-516.

[22] N. Comino, V.L. Narasimhan, A Novel Data Distribution Technique for Host-Client Type Parallel Applications, IEEE Transactions on Parallel and Distributed Systems 13 (2002) 97-110.

[23] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in: OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004) 137-150, `http://labs.google.com/papers/mapreduce.html`.

[24] M. Drozdowski, Scheduling for Parallel Processing, Springer, London (2009).

[25] M. Drozdowski, W. Głazek, Scheduling Divisible Loads in a Three-Dimensional Mesh of Processors, Parallel Computing 25 (1999) 381-404.

[26] M. Drozdowski, M. Lawenda, Multi-Installment Divisible Load Processing in Heterogeneous Systems with Limited Memory, in: R. Wyrzykowski, J. Dongarra, N. Meyer, J. Waśniewski (Eds.), Parallel Processing and Applied Mathematics: 6th International Conference PPAM 2005, Lecture Notes in Computer Science 3911 (2006) 847-854.

[27] M. Drozdowski, M. Lawenda, A New Model of Multi-Installment Divisible Loads Processing in Systems with Limited Memory, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Waśniewski (Eds.), Parallel Processing and Applied Mathematics: 7th International Conference PPAM 2007, Lecture Notes in Computer Science 4967 (2008) 1009-1018.

[28] M. Drozdowski, P. Wolniewicz. Experiments with Scheduling Divisible Tasks in Clusters of Workstations, in: A. Bode, T. Ludwig, W. Karl, R. Wismuller (Eds.), Euro-Par 2000 Parallel Processing: 6th International Euro-Par Conference, Lecture Notes in Computer Science 1900 (2000) 311-319.

[29] M. Drozdowski, P. Wolniewicz, Processing Time and Memory Requirements for Multi-Installment Divisible Job Processing, in: R. Wyrzykowski, J. Dongarra, M. Paprzycki, J. Waśniewski (Eds.), Parallel Processing and Applied Mathematics: 4th International Conference PPAM 2001, Lecture Notes in Computer Science 2328 (2002) 125-133.

[30] M. Drozdowski, P. Wolniewicz, Divisible Load Scheduling in Systems with Limited Memory, Cluster Computing 6 (2003) 19-29.

[31] M. Drozdowski, P. Wolniewicz, Optimum Divisible Load Scheduling on Heterogeneous Stars with Limited Memory, European Journal of Operational Research 172 (2006) 545-559.

[32] D. Ghose, H.J. Kim, Load Partitioning and Trade-Off Study for Large Matrix-Vector Computations in Multicast Bus Networks with Communication Delays, Journal of Parallel and Distributed Computing 55 (1998) 32-59.

[33] C. Gini, Variabilità e mutabilità, C. Cuppini, Bologna (1912).

[34] D. Hochbaum, D. Shmoys, Using Dual Approximation Algorithms for Scheduling Problems: Theoretical and Practical Results, Journal of the ACM 34(1) (1987) 144-162.

[35] H.J. Kim, G. Jee, J.G. Lee, Optimal Load Distribution for Tree Network Processors, IEEE Transactions on Aerospace and Electronic Systems 32(2) (1996) 607-612.

[36] E.L. Lawler, J. Labetoulle, On Preemptive Scheduling of Unrelated Parallel Processors by Linear Programming, Journal of the ACM 25(4) (1978) 612-619.

[37] X. Li, V. Bharadwaj, C.C. Ko, Processing Divisible Loads on Single-Level Tree Networks with Buffer Constraints, IEEE Transactions on Aerospace and Electronic Systems 36 (2000) 1298-1308.

[38] X. Li, V. Bharadwaj, C.C. Ko, Distributed Image Processing on a Network of Workstations, International Journal of Computers and Applications 25 (2003) 1-10.

[39] T. Lim, T.G. Robertazzi, Efficient Parallel Video Processing through Concurrent Communication on a Multi-Port Star Network, in: 2006 Conference on Information Sciences and Systems, Princeton, NJ (2006) 458-463.

[40] J. Lin, C. Dyer, Data-Intensive Text Processing with MapReduce, Morgan & Claypool (2010).

[41] Lp_solve reference guide (2010), http://lpsolve.sourceforge.net/5.5/.

[42] R. Pike, S. Dorward, R. Griesemer, S. Quinlan, Interpreting the Data: Parallel Analysis with Sawzall, Scientific Programming 13 (2005) 277-298.

[43] K. van der Raadt, Y. Yang, H. Casanova, Practical Divisible Load Scheduling on Grid Platforms with APST-DV, Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) (2005) 29.b.

[44] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, Evaluating MapReduce for Multi-Core and Multiprocessor Systems, HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture (2007) 13-24.

[45] T.G. Robertazzi, Ten Reasons to Use Divisible Load Theory, IEEE Computer 36 (2003) 63-68.

[46] J. Sohn, T.G. Robertazzi, S. Luryi, Optimizing Computing Costs Using Divisible Load Analysis, IEEE Transactions on Parallel and Distributed Systems 9 (1998) 225-234.

[47] H.M. Wong, V. Bharadwaj, Aligning Biological Sequences on Distributed Bus Networks: A Divisible Load Scheduling Approach, IEEE Transactions on Information Technology in Biomedicine, 9(4) (2005) 489-501.

[48] Y. Yang, H. Casanova, M. Drozdowski, M. Lawenda, A. Legrand, On the Complexity of Multi-Round Divisible Load Scheduling, INRIA Rhône-Alpes, Research Report 6096 (2007), http://hal.inria.fr/inria-00123711/en/.