RAIRO Operations Research
Will be set by the publisher

SCHEDULING MULTILAYER DIVISIBLE COMPUTATIONS *

J.Berlińska[1] and M.Drozdowski[2]

Abstract. We analyze scheduling multilayer divisible computations. Multilayer computations consist of a chain of parallel applications, such that one application produces input for the next one. A simple form of multilayer computations are MapReduce parallel applications. The operations of mapping and reducing are two divisible applications with precedence constraints. We propose a divisible load model and give an algorithm for scheduling multilayer divisible computations. The algorithm is tested in a series of computational experiments. We draw conclusions on schedule patterns and determinants of the performance.

Keywords: Scheduling, divisible loads, parallel processing, multilayer computations, MapReduce

Mathematics Subject Classification. 90B35, 68M20, 68M14

1. Introduction

In this paper we study scheduling chains of computations on the grounds of divisible load theory (DLT). The divisible load theory is a model of distributed processing assuming that the data to be processed, called load, can be divided into pieces of arbitrary sizes. There are no precedence constraints between these pieces, so that they can be processed independently in parallel on remote computers. Communication delays are non-negligible and must be taken into account when

.

* *The research of the first author has been supported by Polish Ministry of Science and Higher Education grant N N206 372039.*

[1] Faculty of Mathematics and Computer Science, Adam Mickiewicz University, Umultowska 87, 61-614 Poznań, Poland; Joanna.Berlinska@amu.edu.pl

[2] Institute of Computing Science, Poznań University of Technology, Piotrowo 2, 60-965 Poznań, Poland; Maciej.Drozdowski@cs.put.poznan.pl

© EDP Sciences 2001

scheduling the computations. Divisible load model originated in the late 1980s [1, 15] when partitioning the load between parallel processors for the shortest schedule was analyzed. On the one hand, using more processors shortens the schedule, because computers receive smaller amounts of load to process. On the other hand, communications cost additional time. Thus, the problem was which processors to employ and what amounts of data should be sent to them. In the early DLT papers the scheduling problem could be reduced to a system of linear equations. More advanced models were developed later, covering scheduling problems for various network topologies [11, 15, 16, 20], systems with memory limitations [5, 7, 28] and computation costs [37]. The most general divisible load scheduling problem was proved to be **NP**-hard in [41]. Divisible load theory turned out to be a versatile tool for modeling and analyzing processing large amounts of data. Its applications include processing measurement data [15], image and video processing [29, 30, 34], search for patterns in database and text files [21] and linear algebra [17, 24]. The volunteer computing applications on BOINC and similar platforms also fulfill the assumptions of divisible computation. For further details on DLT we direct an interested reader to surveys [4, 9, 19, 36].

In this work we analyze scheduling a chain of divisible jobs. The elements of the chain will be equivalently called *layers*. Results from one layer (except the last) are the input for the next layer. An example of a 2-layer divisible application are MapReduce distributed computations. MapReduce is a parallel programming paradigm for processing big datasets on large number of computers [18, 31, 33, 35]. A more detailed description of MapReduce and multilayer divisible computations is given in the next section. The computations are divided into two phases: mapping and reducing. The computers performing mapping will be called mappers and the computers performing reducing will be called reducers. By the result of [41], scheduling MapReduce computations in heterogeneous systems is **NP**-hard in general. Scheduling MapReduce computations in homogeneous systems has been studied in [6,8]. It was assumed in [6,8] that the load is partitioned equally between reducers. Here we relax this assumption. It has been shown in [6] that for MapReduce computations with a single reducer a dominating schedule structure exists. Namely, mappers should finish their computations in the order in which they were activated. A reducer should read the mapper output one at a time, i.e. reading from many mappers simultaneously is not profitable. Unfortunately, a single optimum schedule structure does not seem to exist for many reducers, even if there is only one reducer layer. Therefore, the complexity of scheduling multilayer divisible computations in homogeneous systems remains open. The contribution of this paper may be summarized as follows:
1) A scheduling model for divisible multilayer computations is proposed.
2) A method of scheduling each layer of computations is given.
3) An algorithm for scheduling inter-layer communications is proposed.
4) A special, polynomially solvable case of the problem is provided.
5) The scheduling algorithm is evaluated in a series of computational experiments.
6) Dependence of the structure of the schedules on the system and application

Table 1. Summary of notation.

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});
A	computing rate of a mapper application;
β_{ijk}	the size of the load sent in interval $[t_i, t_{i+1})$ from sender j to receiver k;
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 mappers;
P_i	processor (i.e. computer) i;
R	number of reducer layers;
g_p	layer p result multiplicity fraction;
δ_{pk}	load fraction received by reducer k in layer p;
r_p	number of reducers in layer p;
S	computation startup time, equal for all mappers;
$[t_i, t_{i+1})$	the i-th communication interval in a given layer;
$T_p(x) = a_p^{red} \max\{x, x \log_2 x\}$	layer p computing time function in load size x;
V	total load size, in bytes;

parameters is studied by simulation. This can be used in the future to construct fast and efficient scheduling heuristics.

The rest of this paper is organized as follows. In the next section we give practical motivation of our study. In Section 3 we build a mathematical model of multilayer divisible computations. Scheduling algorithms are proposed in the following section. Section 5 comprises the results of computational experiments. The last section is dedicated to conclusions. The notation used in this paper is summarized in Table 1. Proofs of the theorems are moved to the appendix, for better readability.

2. Motivation

In this section we introduce MapReduce and multilayer divisible applications. We start with an outline of MapReduce [18,31,33,35], as an introductory example of 2-layer computations. MapReduce parallel computations consist in processing input data sets by creating a set of intermediate key/value pairs, and then reducing them to yet another list of key/value pairs. Hence, MapReduce applications are divided into two layers. In the first layer the input dataset (e.g. a text/HTML file) is

processed by a *Map* function, which generates a set of intermediate $(key1, value1)$ pairs. In the second layer the intermediate pairs are sorted by $key1$, and a *Reduce* function merges the pairs with equal value of $key1$, to produce a list of pairs $(key1, value2)$. Consider an example of calculating word frequencies in a huge set of files [18]. The *Map* function emits an intermediate pair $(word, 1)$ for each $word$ in the input dataset. The *Reduce* function sums together all 1s associated with key $word$ and produces pairs $(word, count)$. Many other practical applications can be expressed in MapReduce, see [18, 31, 33, 35].

Both map and reduce operations are distributed computations. The execution of MapReduce application begins with splitting the input files into load units. Many copies of the program start on a cluster of machines. One of the machines, called the master, assigns work to the other computers. There are m map tasks and r reduce tasks to assign. A computer which is assigned a map task (mapper) reads the load units and processes the data using the *Map* function. The output of this function is divided into r parts by a *partitioning function* (usually of the form $hash(key1) \bmod r$) and written to r files on the local disk. Each of these r files corresponds to one reducer. The information about the local file locations is sent back to the master, which forwards it to the reducers.

When a reducer receives this information, it reads the data from the local disks of the mappers. After reading the intermediate data, the reducer sorts them by the intermediate keys so that all occurrences of the same key are grouped together. Each key and the corresponding set of values are then supplied to the *Reduce* function. Its output is appended to a final output file for a given reducer. Thus, the whole output of MapReduce application is available in r output files.

As stated in [18,40], these files are often passed as the input to another MapReduce call. A chain of MapReduce computations can be seen as a sequence of many computational *layers*. Example multilayer divisible computations are iterative MapReduce computations involving data clustering, link analysis, machine learning [22,26]. Chains of MapReduce jobs have been applied in query processing of parallel and distributed databases [13, 40].

Let us analyze an example presented in [39]. A set of weather stations records temperature every hour for many years. We want to calculate, for every weather station and every day of the year, the average maximum temperature. The computation can be decomposed into two MapReduce applications. The first MapReduce computes the maximum daily temperature for every station-date pair. The results are the input for another MapReduce, which computes the average of the maximum daily temperatures for every station-day-month key. The application workflow can be as follows. The original data composed of tuples $(station, date, time, temperature)$ is split into parts and read by the m processors executing the first mapping. For each such tuple the Map_1 function produces a $(key, value)$ pair, where $key = (station, date)$ and $value = temperature$. These intermediate results are partitioned into r_1 files dedicated to the r_1 processors performing reducing for the first MapReduce application. They read the intermediate data, sort them by keys, then the $Reduce_1$ function computes the maximum temperature at each station for every day and outputs tuples $(station, date,$

$\max\{temperature\}$). The same r_1 processors perform also mapping for the second MapReduce application. Thus, each tuple created by the $Reduce_1$ function is passed to the Map_2 function, which removes the year from the date, creating $(key, value)$ pairs, where $key = (station, day, month)$ and $value = \max\{temperature\}$. The pairs are stored in r_2 files which are then read by r_2 processors performing reducing for the second MapReduce. These r_2 reducers read the pairs, sort them by keys, and compute the final result as a list of $(station, day, month, average(\max\{temperature\}))$ tuples.

In many cases it is possible to transform a multilayer application into a single MapReduce, by implementing more complex Map and Reduce functions. However, applications consisting of many, but simpler, stages are easier to develop and maintain [39].

3. Mathematical Model of Multilayer Computations

We assume that multilayer applications are executed by a homogeneous computer system. Computers are equipped with a CPU, memory and independent network interface (e.g. NIC and DMA). Terms computer, processor, worker will be used interchangeably. Processor i will be denoted by P_i. A processor can open one communication channel at a time, i.e. we use the so-called 1-port model. The structure of the interconnection network is unknown in general, but the bisection width is limited. At most l independent communication channels can be simultaneously in use without reducing the channel communication speed. In other words, if two processors can communicate with speed $1/C$ in the otherwise unloaded network, then the bandwidth limitation for the concurrent channels in the whole network is l/C. Bisection width limitations arise as a consequence of specific network structure, as most of the interconnection topologies considered in theory [25], and existing in practice [2,3,14,23] allow for a limited number of parallel connections between independent pairs of computers. The total size of load to be processed is V (e.g. bytes).

Let us note that there is a substantial difference in the way the layers of the application read and process the load. In the first layer the load is read from an unspecific location in the network file system, and this is interleaved with computations. In the following layers the whole data is read from the preceding layer first, and only then can the computations start. Therefore, we will be saying that a multilayer application consists of one mapper layer (which will be called layer 0) and $R \geq 1$ reducer layers. Let m denote the number of mappers (computers in layer 0), and r_p the number of reducers in layer $p = 1, \dots, R$.

The schedule structure of multilayer computations is shown in Fig.1. The computations are divided into $2R + 3$ *stages*, which partially overlap. In the first stage the code is loaded on the processors. Apart from the mapper and reducer application codes, it may also include libraries, virtual machines etc. For simplicity of presentation we assume that the mapper and all the reducer codes are uploaded together and computation startup time elapses only once because when

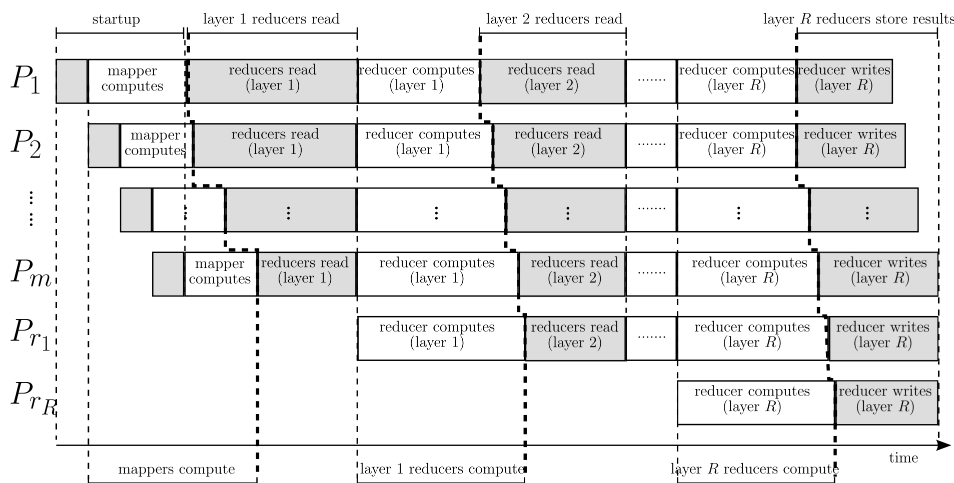

Figure 1. General view of multilayer application schedule structure.

proceeding to the following stages of the computations, the code is already present on the executing processor. We assume that the processors read the code from the network file system. The computation startup time of a single processor is S. The differences in the startup times resulting from different processor locations are assumed to be negligible.

In the second stage, the mappers read load units from the network file system, process them, and store the results in r_1 local files, for r_1 reducers in the first layer. For clarity of presentation, we represent all these operations together as processing with rate A (e.g. sec/byte). A discussion of this model can be found in [6,8]. Let α_i denote the size (e.g. in bytes) of load assigned to mapper i. Since α_i is a rational number it needs rounding to load units used in practical applications. We assume that the effects of load size rounding are negligible. The amount of results produced by the mappers is proportional to the input size: For α_i bytes of input $g_0\alpha_i$ bytes of output are produced.

In the following $2R$ stages reducing is performed. Precisely, in stage number $1+2p$, $1 \le p \le R$, reducers in layer p read load from mappers (if $p = 1$) or reducers in layer $p-1$ (cf. Fig.1). All reducers read the load with equal rate C (expressed e.g. in seconds per byte). At most one channel can be opened between two processors with transfer rate C. The number of simultaneously used channels cannot exceed the bisection width limit l. The partitioning function divides the space of key values into r_p not necessarily equal parts. Let δ_{pk} denote the fraction of results assigned to reducer k in layer p. The amount of results produced by the reducers in layer p for the input size α is $g_p\alpha$. The total amount of load sent to layer p is $V\prod_{i=0}^{p-1} g_i$. Hence, reducer k in layer p reads input of size $\delta_{pk}V\prod_{i=0}^{p-1} g_i$. Note that the partitioning functions in all layers are sender-independent. This means that the proportions between the amounts of load sent to the receivers in

the next layer are the same for each sender in the layer. We use this as an optimistic approximation assuming that each particular key has roughly equal frequency on the senders.

In stages number $2+2p$, $1 \le p \le R$, reducers from layer p sort the input data and perform computations using $Reduce_p$ function. For a multilayer application, $Reduce_p$ comprises both reducing and mapping for the $(p+1)$-st layer. Note that a reducer can start computations only after receiving the whole assigned load, including the load from the machine which finished in the previous layer as the last one. The sequence of communications between the layers is unknown a priori. Hence, it is hard to predict the order in which reducers in the next layer become ready to start their computations. Consequently, we assume that all reducers in layer p start computations not earlier than after transferring all data between layers $p-1$ and p. Let s_p^{red} denote layer p reducer computation startup time, and a_p^{red} (in seconds per byte) its processing rate. As reducer k in layer p receives input of size $\delta_{pk} V \prod_{i=0}^{p-1} g_i$, its execution time is $s^{red} + T_p(\delta_{pk} V \prod_{i=0}^{p-1} g_i)$, where $T_p(x)$ is the running time vs. size x of the input. We will assume that sorting dominates in reducer execution time, and $T_p(x) = a_p^{red} \max\{x, x \log_2 x\}$, since the amount of data is rational and $x > x \log_2 x$ is possible.

In the last, $(2R+3)$-rd stage, the reducers in layer R write the results to the network file system with 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 a 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 modified to handle other organizations of storing the results (see Section 4.1).

In this study we exclude simultaneous execution of several map or reduce tasks on the same computer. Were such colocation possible, it can be represented as several processors, each running a different map task or reduce task. If there are background services executed by the processor (e.g. the network file system), then we assume that these services influence the processor in a stable way, and performance parameters $A, a_p^{red}, C, S, s_p^{red}$ remain constant.

Our goal is 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 $\alpha_1, \dots, \alpha_m$, and schedule the communication between the layers as well as when storing the final results, so that the total schedule is as short as possible.

4. Scheduling Multilayer Computations

In this section we consider load partitioning and communication scheduling for multilayer computations. The schedules are built for one layer at a time starting from the last layer toward the first (mapper) layer. In Section 4.1 a method of load partitioning in reducer layers is given. In Section 4.2 the mapper layer is considered. It is shown in Section 4.3 that a feasible communication schedule for

transferring the loads between the layers always exists. The whole algorithm is summarized in Section 4.4.

4.1. Load Partitioning for Reducer Layers

The load distribution for layer $p = R, \dots, 1$ depends on the fractions of load received by the reducers in layer $p + 1$. Therefore, the values $\delta_{p+1,k}$ should be calculated before δ_{pk}. Consequently, the algorithm proceeds backward through the reducer layers and finishes with finding the load distribution for the mapper layer. Let $t_0 = 0$ denote the start time of computations in layer p. Let $t_1 \leq \dots \leq t_{r_p}$ denote the moments when reducers in layer p finish their computations. As all reducers are identical and start computations simultaneously, we may assume that they are ordered by the computation completion times, and reducer k finishes computations at moment 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 optimum load partitioning in layer p:

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

$$s_p^{red} + T_p(\delta_{pi} V \prod_{q=0}^{p-1} g_q) \leq t_i \quad \text{for} \quad i = 1, \dots, r_p \tag{2}$$

$$C \sum_{j=1}^{i} \beta_{ijk} \leq t_{i+1} - t_i \text{ for } i = 1, \dots, r_p, k=1, \dots, r_{p+1} \tag{3}$$

$$C \sum_{k=1}^{r_{p+1}} \beta_{ijk} \leq t_{i+1} - t_i \text{ for } i = 1, \dots, r_p,\ j=1, \dots, i \tag{4}$$

$$C \sum_{j=1}^{r_p} \sum_{k=1}^{r_{p+1}} \beta_{ijk} \leq l(t_{i+1} - t_i) \quad \text{for} \quad i = 1, \dots, r_p \tag{5}$$

$$\beta_{ijk} = 0 \quad \text{for } j=1, \dots, r_p, i = 1, \dots, j-1,\ k = 1, \dots, r_{p+1} \tag{6}$$

$$\sum_{i=1}^{r_p} \beta_{ijk} = \delta_{p+1,k} \delta_{pj} V \prod_{q=0}^{p} g_q \quad \text{for} \quad j = 1, \dots, r_p,\ k = 1, \dots, r_{p+1} \tag{7}$$

$$\sum_{j=1}^{r_p} \delta_{pj} = 1 \tag{8}$$

We minimize the length of the schedule from the moment when reducers in layer p start computations to the time when they finish communicating with reducers in layer $p+1$. By equations (2) reducer i from layer p finishes computations at the moment t_i, for $1 \leq i \leq r_p$. Constraints (3)-(5) guarantee that all communications fit in the communication intervals together, bisection width limit is observed, and

1-port model is enforced. The method of building a schedule for communications will be given in Section 4.3. By (6) no reducer sends results before finishing computations. Each reducer in layer p sends all results by (7) and the whole load is processed by (8). Note that load fractions $\delta_{p+1,k}$ are constants computed in the previous step of the optimization. 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.

For simplicity of exposition we define $r_{R+1} = 1$ and $\delta_{R+1,1} = 1$. Consequently, only one reducer from the last layer R can store results at a time. However, the above program can be modified so that storing results is confined by bisection width limit l, or the results are stored locally. In the first case, it is enough to omit constraint (3). The second case can be handled by including the results writing time in function T_p, and substituting constraints (3)-(7) with $t_i \leq t_{r_R+1}$ for $i = 1, \dots, r_R$.

Let us note that the constraints (2) are not linear because of the form of function T_p. In order to provide a practical method for solving (1)-(8), we transform this program into a linear program. We will approximate the function T_p with a piecewise linear convex function T'_p. For $x \in [0, 2)$ we set $T'_p(x) = T_p(x) = x = a_0 x + b_0$ for $a_0 = 1$, $b_0 = 0$. For each interval $[2^y, 2^{y+1})$, for positive integer $y \leq \log_2 V$, the values $a_y = (T_p(2^{y+1}) - T_p(2^y))/(2^{y+1} - 2^y)$ and $b_y = T_p(2^y) - a_y 2^y$ are calculated. Then, we set $T'_p(x) = a_y x + b_y$ for $x \in [2^y, 2^{y+1})$. Thus, the constraints (2) are changed to

$$s_p^{red} + a_y \delta_{pi} V \prod_{q=0}^{p-1} g_q + b_y \leq t_i \quad \text{for} \quad i = 1, \dots, r_p, y = 0, \dots, \lfloor \log_2 V \rfloor, \tag{9}$$

what increases the number of constraints by $r_p \lfloor \log_2 V \rfloor$. The relative error caused by this approximation decreases with growing V. In our experiments (see Section 5), the sizes of load obtained by the reducers are larger than 1E5. For such values the approximation error is less than 1%. Such error is on par with typical accuracy of measuring system parameters 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 $T'_p(x) \geq T_p(x)$ for $x \leq V$, the load partitioning obtained for function T'_p allows to create a feasible solution with the original function T_p.

4.2. Load Partitioning for Mapper Layer

In this section we consider scheduling mapper computations and communication between the mappers and the first reducer layer. Let $t_1 \leq \dots \leq t_m$ be the moments when mappers finish their computations. Let t_{m+1} be the moment when mapper to reducer communications finish. 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. Value $z_{ij} = 1$ means that mapper j has finished computations by time t_i and can send some load in interval $[t_i, t_{i+1})$. In the opposite case $z_{ij} = 0$.

We will denote by β_{ijk} the amount of results read by reducer k from mapper j in interval $[t_i, t_{i+1})$. Let M be a big constant, e.g. $M \gg mS + V(A+C)$. The optimum load partitioning and the sequence of finishing computations by the mappers can be computed from the following integer linear program:

$$\text{minimize} \quad t_{m+1} \tag{10}$$
$$jS + A\alpha_j \geq t_i - z_{ij}M \quad \text{for} \quad i = 1, \dots, m,\ j = 1 \dots, m \tag{11}$$
$$jS + A\alpha_j \leq t_i + (1 - z_{ij})M \quad \text{for} \quad i = 1, \dots, m,\ j = 1 \dots, m \tag{12}$$
$$C\sum_{j=1}^{m} \beta_{ijk} \leq t_{i+1} - t_i \text{ for } i = 1, \dots, m, k{=}1, \dots, r_1 \tag{13}$$
$$C\sum_{k=1}^{r_1} \beta_{ijk} \leq t_{i+1} - t_i \text{ for } i = 1, \dots, m,\ j{=}1, \dots, m, \tag{14}$$
$$C\sum_{j=1}^{m}\sum_{k=1}^{r_1} \beta_{ijk} \leq l(t_{i+1} - t_i) \quad \text{for} \quad i = 1, \dots, m \tag{15}$$
$$\beta_{ijk} \leq z_{ij}V \quad \text{for } i = 1, \dots, m,\ j{=}1, \dots, m,\ k = 1, \dots, r_1 \tag{16}$$
$$\sum_{i=1}^{m} \beta_{ijk} = \delta_{1k} g_0 \alpha_j \quad \text{for} \quad j = 1, \dots, m,\ k = 1, \dots, r_1 \tag{17}$$
$$\sum_{i=1}^{m} \alpha_i = V \tag{18}$$
$$z_{i+1,j} \geq z_{ij} \quad \text{for} \quad i = 1, \dots, m-1,\ j = 1, \dots, m \tag{19}$$
$$\sum_{j=1}^{m} z_{ij} = i \quad \text{for} \quad i = 1, \dots, m \tag{20}$$
$$z_{ij} \in \{0, 1\} \quad \text{for} \quad i = 1, \dots, m,\ j = 1, \dots, m \tag{21}$$

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 mapper to reducer communications. Inequalities (11) and (12) guarantee that mappers finish computations in the order defined by variables z_{ij}. By (13) and (14) no mapper or reducer communicates longer than the communication interval. By (15) the bandwidth limit is observed. Inequalities (16) guarantee that no load is sent by a mapper which has not finished computations. Each reducer receives appropriate amount of results by (17) and the whole load is processed by (18). Constraints (19)-(21) ensure that there is one-to-one correspondence between the mappers and time 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 $m^2 r_1 + 4m^2 + 2mr_1 + m + 1$ constraints in the above linear program.

The order in which the mappers should finish their computations is unknown in general. This resulted in using binary variables in the mathematical program

(10)-(21). If $S = 0$, then the mappers are activated simultaneously, and hence are not ordered by the sequence of activating them. Consequently, the binary variables z_{ij} are not needed, formulation (10)-(21) becomes an LP similar to (1)-(8), solvable in polynomial time. In Theorem 4.1 we prove that if the load is distributed equally between the reducers in the first layer, then the mappers should finish computations in the FIFO order, i.e. in the order of their activation. Such a distribution may emerge in practice, e.g., as a result of using a partitioning function of the form $hash(key1)$ mod r_1 common in MapReduce applications. Also in this case binary variables can be removed from the LP (10)-(21). Whether the FIFO schedule structure is a global optimum, remains an open question. Some preliminary computational experiments indicate that it may be the case.

Theorem 4.1. *If $\delta_{1k} = \frac{1}{r_1}$ for $k = 1, \dots, r_1$, then the FIFO order of finishing mapper computations is optimum.*

Proof. See Appendix. □

4.3. Scheduling Communications

After solving the mathematical programs given in Sections 4.1 and 4.2, the amounts of data sent between each pair of computers in each interval are known. A feasible communication schedule can be built for each interval $[t_i, t_{i+1})$ in each computation layer using a two-stage approach similar to the one used for problem $R|pmtn|C_{max}$ [10, 12, 27]. Then, a schedule for all load transfers can be built by concatenating the partial schedules for the consecutive intervals. Note that our communication scheduling problem is not identical with $R|pmtn|C_{max}$, because the bisection width limit is absent in $R|pmtn|C_{max}$. Hence, a schedule for $R|pmtn|C_{max}$ is not necessarily a feasible schedule for our problem. The problem of constructing a communication schedule can be analyzed in terms of matching decomposition [38]. To make the paper self-contained we give a dedicated algorithm in detail and prove its feasibility.

Consider one of the intervals $[t_i, t_{i+1})$ with the optimum load transfers β_{ijk} from sender j to receiver k delivered by formulations (1)-(8), (10)-(20). Let us denote the number of load senders for the given interval by n_1 and the number of 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 interval. Thus, $w_{jk} \le 1$ is the fraction of the length of the current interval used to transfer load from sender j to receiver k. Note that $\sum_{j=1}^{n_1} \sum_{k=1}^{n_2} w_{jk} \le l$ by (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 other rows and columns, and

- exactly l elements in total if bisection width limitation is active for W, or at most l elements otherwise.

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

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

In the above 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 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 active bisection width limitation is also obeyed.

In each iteration of the while loop either a row or a 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 (cf. Fig.2). Beyond the sink and source, the network has n_1 nodes corresponding to senders, n_2 nodes corresponding to 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 bisection

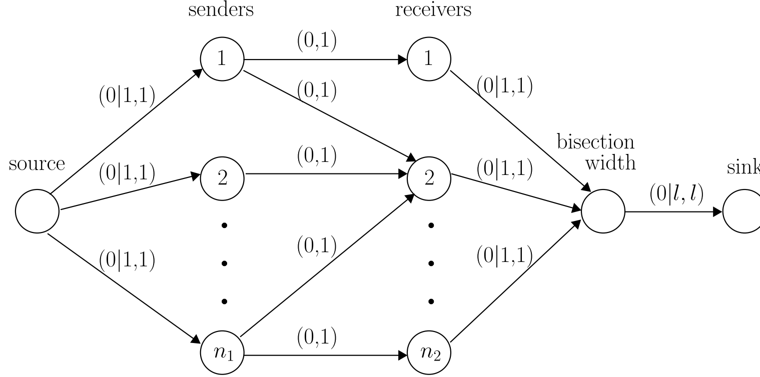

Figure 2. Network for finding set $\mathcal{F}$. Arcs are labeled with (lower, upper) bounds. Values $a|b$ are used for non-critical—critical nodes.

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 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 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. 2 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 described network is equivalent to finding set $\mathcal{F}$: arcs from sender node j to receiver node k with positive flow indicate $w_{jk} \in \mathcal{F}$.

Theorem 4.2. *Set $\mathcal{F}$ and hence a feasible flow always exist.*

Proof. See Appendix. □

Since network flows in a graph with n nodes can be found in $O(n^3)$ time, set $\mathcal{F}$ for communication between layers p and $p+1$ can be found in $O((\max\{r_p, r_{p+1}\})^3)$ time, because $r_p \geq n_1, r_{p+1} \geq n_2$. A schedule for each interval can be found in $O(\max\{r_p, r_{p+1}\})^5)$ because there are at most $O(\max\{r_p, r_{p+1}\})^2)$ iterations each of which is done in $O(\max\{r_p, r_{p+1}\}^3)$ time. There are at most r_p intervals in the communication schedule. Consequently a schedule for communications in one layer can be calculated in $O((\max\{r_p, r_{p+1}\})^6)$. With respect to mappers and the first reducer layer it is $O((\max\{m, r_1\})^6)$.

4.4. A Complete Algorithm

In order to calculate load partitioning for the whole multilayer application, the mathematical programs described above should be put together and solved as one

compound program. However, it is very hard in practice. Firstly, such a program is large even for small instances. For example, according to the formulas given in Sections 4.1 and 4.2, an instance of $R = 3, m = r_1 = r_2 = r_3 = 10$, $V = 2^{30}$ results in 3184 rational variables, 100 binary variables and 4104 constraints (after linear approximation of constraints (2)). Secondly, α_j, δ_{pk} are variables in the compound mathematical program, constraints (7), (17) are quadratic, and a compound program amounts to solving a quadratic programming problem (which is **NP**-hard in general). Therefore, we propose to build a schedule for each layer separately, from the last to the first layer. Then, values $\delta_{p+1,k}$ are fixed while calculating load distribution in layer p and the above mathematical programs become linear. Load partitioning for the reducers can be found in time $O(LP(r_p^2 r_{p+1} + 2r_p + 1, r_{p+1} r_p^2/2 + r_p(3r_{p+1} + r_p + \lfloor \log_2 V \rfloor)/2 + 5r_p/2 + 1))$ for layer $p = R, \dots, 1$, where $LP(a, b)$ is complexity of linear programming with a variables and b constraints. Calculating load partitioning in mappers requires time $O(MLP(m^2, m^2 r_1 + 2m + 1, m^2 r_1 + 4m^2 + 2mr_1 + m + 1))$ where $MLP(a, b, c)$ is complexity of solving a mixed linear program with a binary variables, b rational variables and c constraints. The complexity of calculating mapper load distribution can be lowered by use of Theorem 4.1. If load distribution is equal in layer $p = 1$ then FIFO order of mapper completion times is optimum. Then binary variables are unnecessary in (10)-(21), and it can be solved in $O(LP(m^2 r_1 + 2m + 1, m^2 + m(2r_1 + 1) + 1))$ time as a standard linear program. We discuss in Section 5 when equal load partitioning emerges in the first reducer layer. Overall, the complexity of scheduling a single layer and its communications is $O(LP(r_p^2 r_{p+1} + 2r_p + 1, r_{p+1} r_p^2/2 + r_p(3r_{p+1} + r_p)/2 + 5r_p/2 + 1) + \max\{r_p, r_{p+1}\})^6)$ in the case of reducers, and $O(LP(m^2 r_1 + 2m + 1, m^2 + m(2r_1 + 1) + 1) + (\max\{m, r_1\})^6)$ in the case of mappers with FIFO assumption.

5. Computational Experiments

The goal of this section is to confront three issues of scheduling multilayer divisible applications. In Section 5.1 we evaluate running time, and hence, practical applicability of our algorithm. Section 5.2 is dedicated to the analysis of scalability of multilayer divisible applications. The impact of the instance parameters on the structure of schedules is studied in Section 5.3. All linear programs were solved using lp_solve linear programming library [32]. The code was implemented in C++ in Microsoft Visual Studio 2012.

5.1. Performance of the Algorithm

Since the algorithm introduced in Section 4 has quite high order of computational complexity we study here its practical execution time. The experiment is designed to verify running time against key parameters m, r_p, V while randomizing the remaining numerical values. We will study execution time of a single reducer layer because in each layer the same algorithm is repeated, and analogous

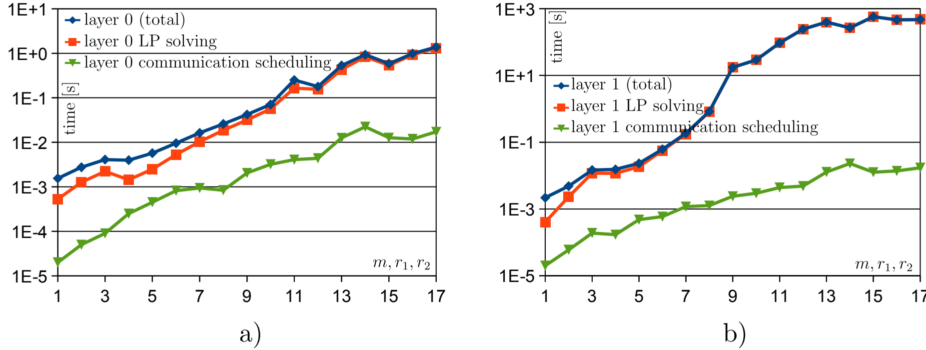

FIGURE 3. Algorithm execution time vs. m, r_1, r_2, a) layer 0, b) layer 1.

instance parameters determine complexity in the same way. The reducers in the last layer write their results sequentially and scheduling this layer is computationally simpler. Hence, we do not report execution times for the last layer. Unless stated to be otherwise, we used $R = 2$ layers with $m = r_1 = r_2 = 10$, $V = 1E15$, $g_p = 0.1$ (for $p = 0, 1, 2$). The bisection width limit l was chosen randomly from $\{m, \lceil m/2\rceil, \lceil m/4\rceil\}$. Processor parameters $A, a_p^{red}, C, S, s_p^{red}$ were generated from the uniform distribution U[1E-10,1E-5]. In the following figures each point represents an average from 100 instances. All the computations were executed on a PC computer with Intel Core i5-2500K running at 3.3 GHz and 6 GB RAM.

In the first set of experiments the numbers of processors in all layers were changed while keeping the condition $m = r_1 = r_2$. The results are shown in Fig. 3. As expected, increasing m, r_1, r_2 strongly affects the running times of the algorithms. It can be seen that linear programming is the most time-consuming part of the computations. The time necessary for creating the communication schedule is 1 to 4 orders of magnitude shorter. For almost all values of m, r_1, r_2 the mapper layer is scheduled much faster than the first reducer layer. This means that inequalities (9) used to linearize constraints in the reducer layers make the LP especially difficult to solve.

The second series of experiments presents the performance of the algorithm for load size V changing from 1E12 to 1E17. It can be verified in Fig. 4 that V does not influence significantly the algorithm running time. Though increasing V results in creating additional constraints in the LPs for the reducer layers, the number of constraints increases by only 14% when going from V =1E12 to V =1E17. It seems that this is not enough to make a clear difference in the linear programming time. The communication scheduling time does not change with V because sending greater amount of data usually requires creating larger chunks of data rather than performing more communications.

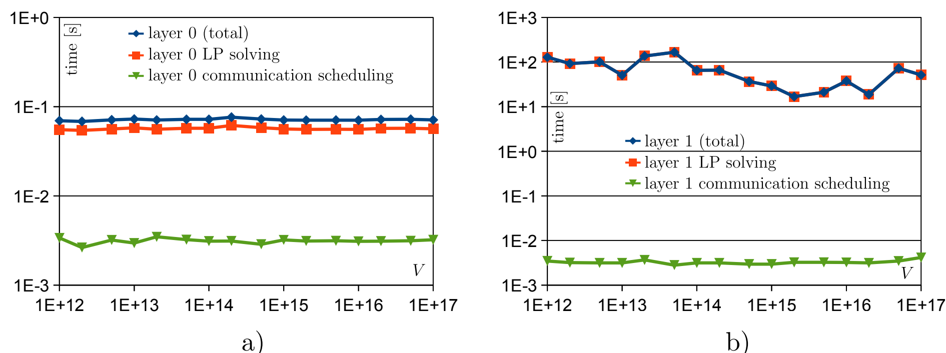

FIGURE 4. Algorithm execution time vs. V, a) layer 0, b) layer 1.

5.2. Speedup of Multilayer Applications

A study of the scalability limits of a parallel application has practical significance because it verifies whether it is possible to accelerate the computations by allowing more computing resources. Therefore, we evaluated speedup of the multilayer divisible application against the number of computers used in each layer.

Unless written to be otherwise, the reference system configuration in the following experiments had parameters: $R = 2, m = r_1 = r_2 = 5, s_p^{red} =$ 1E-2, $a_p^{red} =$ 1E-7, $g_p = 0.1$ (for $p = 0, 1, 2$), $A =$ 1E-7, $S = 1$, $C =$ 1E-8, the bisection width limit $l = 5$ is not restricting the communication, $V =$ 1E15. These parameters were chosen to represent a multilayer divisible application in a small contemporary computer cluster. The size of the test instances is a result of both high complexity of the scheduling algorithm and the achievable numerical precision.

In Fig. 5 we present the speedup of the application for changing m and r_1 (in relation to the system with $m = r_1 = r_2 = 1$). We analyzed cases with big ($g_p = 1$) and small ($g_p =$ 1E-3) load multiplicity fractions in each layer. It turned out that the value of g_2 has almost no impact on the speedup. This can be explained by the fact that g_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 $g_2 = 1$ in Fig. 5.

It can be seen that the application scales well with the mapper number m if g_0 is small (see Fig. 5a). In this case, the reducers receive little load and do not dominate in the computations. On the other hand, if g_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. 5a and Fig. 5b). The range of the speedup is determined not only by g_0, but also by g_1. If g_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 g_1 is small, then the execution time of the whole application

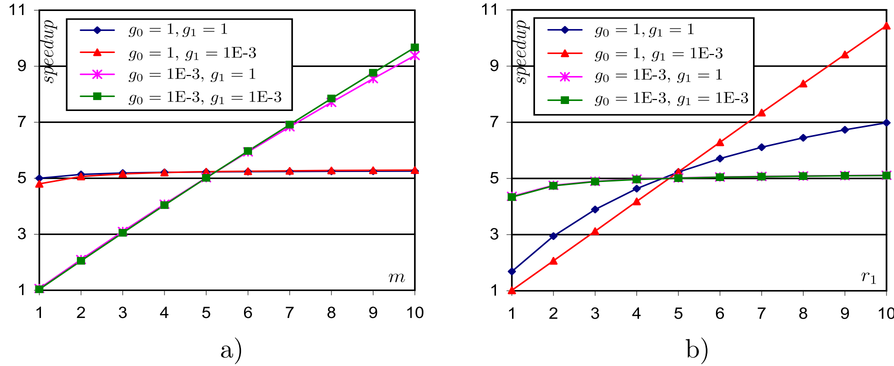

FIGURE 5. Speedup for different g_0, g_1, a) vs. m, b) vs. r_1.

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 g_0 and g_1 are big. We do not show these results here because they follow the pattern of Fig. 5a,b. Overall, the results are similar in nature to the results in [8].

5.3. Load Distribution between Reducers

In [8] scheduling algorithms for 2-layer applications were proposed. The algorithms 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 reducers were equal. In this paper we relaxed the assumptions on the communication pattern, as well as on the equal load partitioning in the reducer layers. Therefore, we analyze the load distribution between the processors in a given layer. We present values $\delta_{pk}/(1/r_p)$, i.e. the load fractions received by the processors in layer p relative to the equal distribution. Since the reducers in a given layer start computations at the same moment and finish them in the order of their indices the fractions $\delta_{p,k}$ are always nondecreasing.

The number of the reducers in the first layer is set to $r_1 = 10$. The bisection width limit $l = 5 = r_2$ is not restricting the communication between the first and the second layer. We start our study with the second (last) layer. The values of load fractions $\delta_{2,k}$ for different values of communication rate C are shown in Fig. 6. It can be seen in Fig. 6a that for very fast communication the load distribution in the second layer of reducers is very flat. This can be explained by the fact that for very fast 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. For very big values of C (C = 1E-4, C = 1E-5 in Fig. 6b) the time needed for storing the results dominates in the schedule length. Thus, it is profitable to start storing the

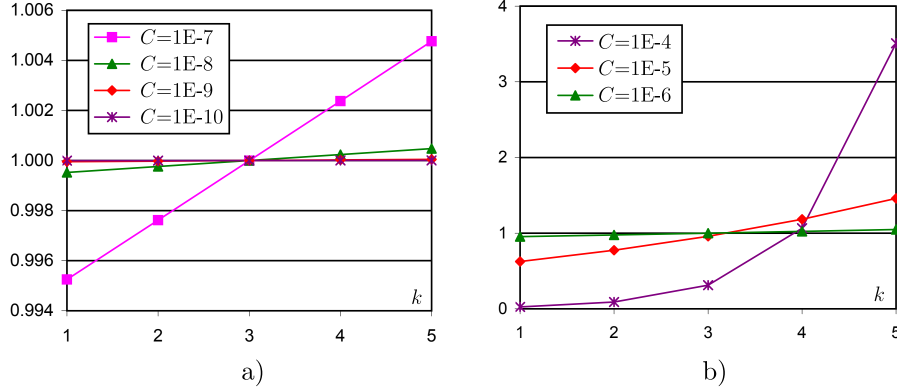

FIGURE 6. Relative load fractions $\delta_{2,k}/(1/r_2)$ vs. communication rate C, a) fast communication, b) slow communication.

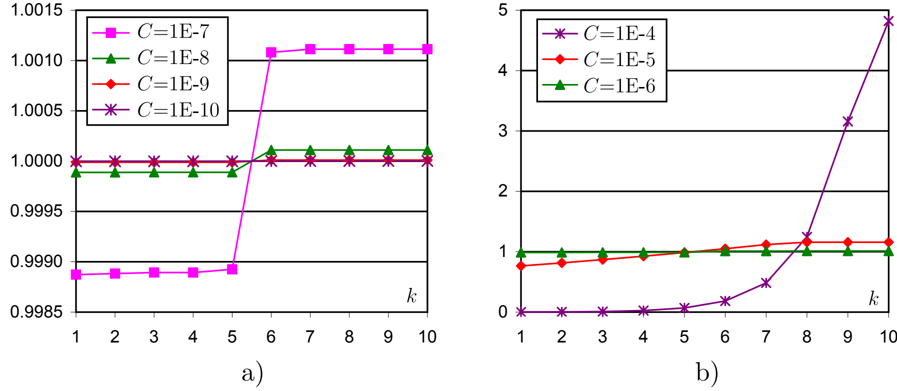

FIGURE 7. Relative load fractions $\delta_{1,k}/(1/r_1)$ vs. communication rate C, a) fast communication, b) slow communication.

results from some reducers very early, while other processors are still computing. This leads to significant inequalities in the reducer load distribution. The first processors receive very small load, while the last reducer has to process about 30% for C = 1E-5, or even more than 70% for C = 1E-4 of all data.

The load distribution between the processors in the first reducer layer (δ_{1k}) is presented in Fig. 7. As in the second layer, the distribution is balanced for fast communication and very unequal for slow communication. Since for fast communication δ_{1k} are nearly equal, application of Theorem 4.1 is justified and mapper completion times can follow FIFO order. Another interesting phenomenon is that for fast communication the reducers can be divided into two groups comprising 5 processors each (see Fig. 7a). The processors in a given group receive similar amounts of load. As there are $r_2 = 5$ processors in layer 2 which receive data

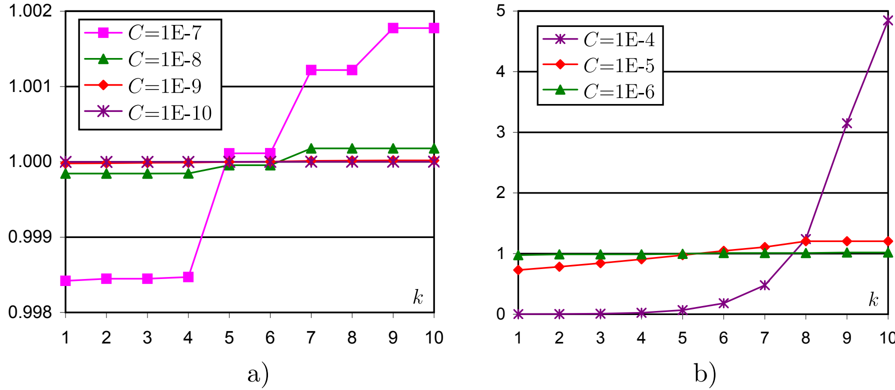

FIGURE 8. Relative load fractions $\delta_{1,k}$ vs. communication rate C for $r_1 = 10$, $r_2 = 4$, a) fast communication, b) slow communication.

from the reducers in layer 1, we infer that the processors in a given group can use a similar communication pattern. Precisely, 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. The inequalities in the load distribution 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. 7b 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 above instances reducer number r_1 was divisible by reducer number 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. 8 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 $C =$ 1E-7, and groups of sizes 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 simple and fast scheduling heuristics.

The time needed to send the load from one reducer layer to another depends on the bisection width limit l. In Fig. 9 we present the load distribution in the first reducer layer for different values of l. The value $C =$ 1E-8 used in Fig. 9 can be considered fast communication. The results in Fig. 9a confirm that the

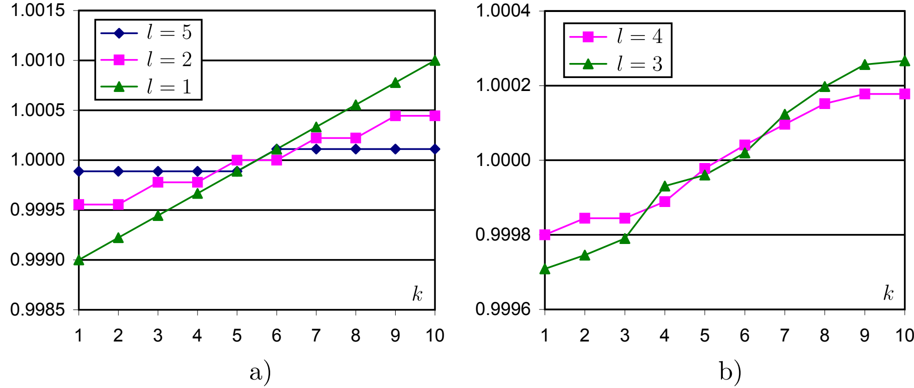

FIGURE 9. Relative load fractions $\delta_{1,k}/(1/r_1)$ vs. the bisection width limit l, a) r_1 divisible by l, b) r_1 not divisible by l.

groups of processors receiving similar load are ruled by the number of processors which can communicate at the same time. When $l = 2$, groups of 2 processors can be observed in Fig. 9a. 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. 9b), then neither clear groups of processors nor communication patterns can be distinguished.

5.4. Load Distribution between Mappers

We analyze here the load distribution in the mapper layer. Whether the FIFO structure of the mapper computations is a global optimum, remains an open question. We use FIFO in the following simulations for practical reasons. The startup times are short in relation to the whole schedule, and hence, the order of starting processors has small impact on differentiating the mappers. Mixed integer linear programming is computationally hard, and only very small instances can be solved to optimality in acceptable time. Thanks to the FIFO order instances with $m = 50$ mappers are considered 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. 10. The load distribution in the mapper layer is shown as the fractions $\alpha_j/(V/m)$. The difference between the load distributions in the mapper and the reducer layers results from the startup times. Yet, this difference was almost negligible. The same phenomena as for the reducers were observed. 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.

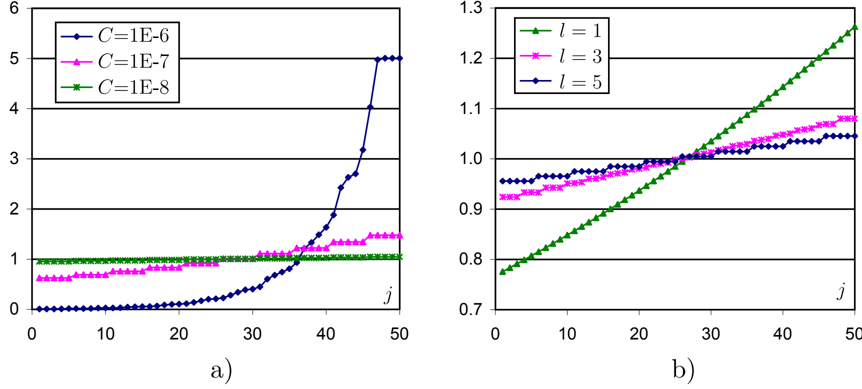

FIGURE 10. Mapper load fractions $\alpha_j/(V/m)$ for $S = 1$, a) vs. C, b) vs. l.

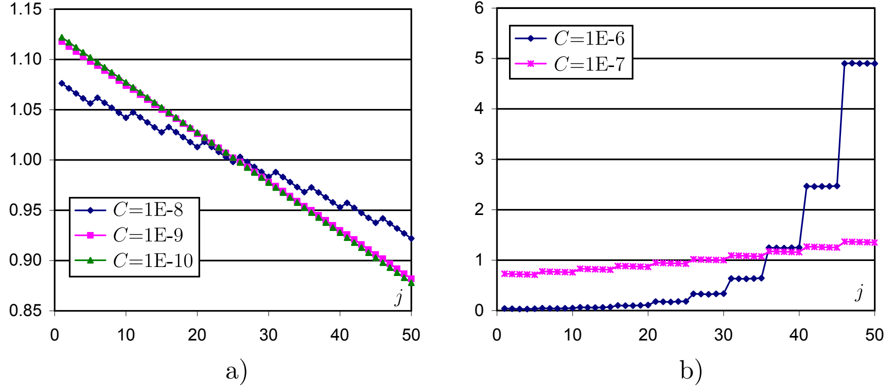

FIGURE 11. Mapper load fractions $\alpha_j/(V/m)$ for S = 1E4 vs. C, a) fast communication, b) slow communication.

In order to better expose the differences between the mapper and the reducer layers we increased the startup time S to as much as 1E4. The results of the experiments with changing communication rate C are shown in Fig. 11. For fast communication we observed a qualitative difference in the load distribution (see Fig. 11a). The mapper loads are now generally decreasing. Similarly to the reducer layer, the mappers can be divided into groups of $r_1 = l = 5$ processors. However, the load fractions obtained by the processors in a 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. Then, they use the available

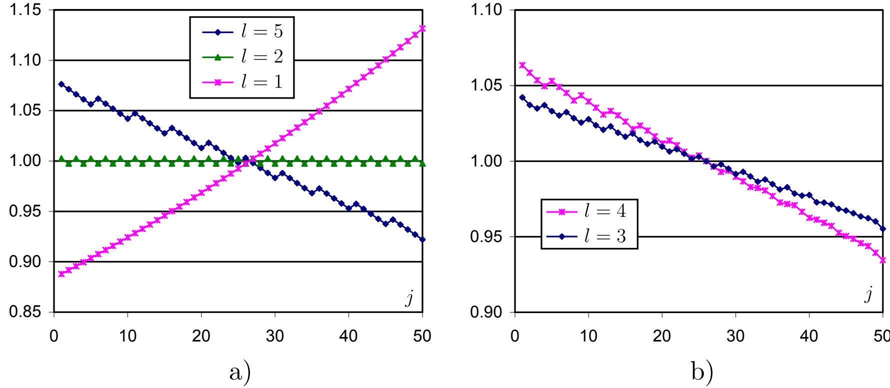

FIGURE 12. Mapper load fractions $\alpha_j/(V/m)$ for $S = 1\text{E}4$ vs. l, a) m divisible by l, b) m not divisible by l.

communication channels to send the results to the reducers in parallel. 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 $C = 1\text{E-}8$ 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. 11a.

When communication gets slower (cf. Fig. 11b) the load sizes of the consecutive mappers are increasing, and groups of 5 mappers receiving similar amounts of load can be seen. This can be explained by the fact that for slow communication mappers start communicating early to account for slow communication medium. This is similar to distributions in reducer layers. The startup time $S = 1\text{E}4$ is not significant enough to have noticeable impact in this case.

The load distributions in the mapper layer for different bisection width limits l are presented in Fig. 12. When the number of mappers m is divisible by l (Fig. 12a), 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 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. 12b). 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".

6. Conclusions

In this work we studied scheduling multilayer divisible applications. Algorithms based on mixed nonlinear programming were proposed, and then simplified to a sequence of linear programs. The order in which the mappers should finish their computations is crucial for determining complexity of our problem. We proved that the FIFO order is optimum in special cases. Whether it is optimum in general, remains an open question. The method of constructing a communication schedule was proposed.

Load distributions in different computational layers were analyzed. To a large degree they are determined by the communication rate C. When C is small, the computation time dominates the schedule length. Then the load distribution is balanced, so that all processors finish computing around the same time. If C is 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 in parallel using the l available communication channels. The next group finishes computations almost exactly when the communication channels are released. In practice it is profitable to use the numbers of mappers or reducers divisible by l because the schedule structure is known. Consequently, faster scheduling algorithms may be devised.

Many aspects remain open for the future research. For example, is FIFO sequence globally optimum? There are practical issues of, e.g., handling volatility of the computational and communication resources, accommodating to unpredictable distribution of the keys, unequal processing times of data units.

References

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

[2] M. Al-Fares, A. Loukissas, A. Vahdat, A Scalable, Commodity Data Center Network Architecture, SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication.

[3] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, Data Management and Transfer in High-Performance Computational Grid Environments, Parallel Computing 28 (2002) 749-771.

[4] 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.

[5] 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.

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

[7] J. Berlińska, M. Drozdowski, Heuristics for multi-round divisible loads scheduling with limited memory, Parallel Computing 36 (2010) 199-211.

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

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

[10] J. Błażewicz, W. Cellary, R. Słowiński, J. Węglarz, Scheduling under resource constraints - deterministic models, Annals of Operations Research, vol.7, 1986.

[11] J. Błażewicz, M. Drozdowski, Scheduling divisible jobs on hypercubes, Parallel Computing 21 (1995) 1945-1956.

[12] J. Błażewicz, K.Ecker, E.Pesch, G.Schmidt, J.Węglarz, Scheduling Computer and Manufacturing Processes, Springer-Verlag: Heidelberg, 1996.

[13] Y.Bu, B.Howe, M.Balazinska, M.D.Ernst, HaLoop: Efficient iterative data processing on large clusters. Proceedings of the VLDB Endowment, 3(1-2), 2010. 285-296.

[14] H. Casanova, Network modeling issues for GRID application scheduling, International Journal of Foundations of Computer Science 16 (2005) 145-162.

[15] Y.-C.Cheng, T.G.Robertazzi, Distributed computation with communication delay, IEEE Transactions on Aerospace and Electronic Systems 24 (1988) 700-712.

[16] 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.

[17] 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.

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

[19] M.Drozdowski, Scheduling for Parallel Processing, Springer, 2009.

[20] M. Drozdowski, W. Głazek, Scheduling divisible loads in a three-dimensional mesh of processors, Parallel Computing 25 (1999) 381-404.

[21] M. Drozdowski, P. Wolniewicz, Experiments with scheduling divisible tasks in clusters of workstations, in: A.Bode, T.Ludwig, W.Karl, and R.Wismuller (Eds.), Proceedings of 6th Euro-Par Conference, Lecture Notes in Computer Science 1900 (2000) 311-319.

[22] J.Ekanayake, H.Li, B.Zhang, T.Gunarathne, S.H.Bae, J.Qiu, G.Fox, Twister: A Runtime for Iterative MapReduce, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. 2010,810-818

[23] N. Farrington, E. Rubow, A. Vahdat, Data Center Switch Architecture in the Age of Merchant Silicon, 17th IEEE Symposium on High Performance Interconnects, 2009. HOTI 2009, 93-102.

[24] 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.

[25] M.D. Grammatikakis, D.F. Hsu, M. Kraetzel, Parallel System Interconnections and Communications, CRC Press, Boca Raton, 2001.

[26] T.Gunarathne, B.Zhang, T.L.Wu, J.Qiu, Portable Parallel Programming on Cloud and HPC: Scientific Applications of Twister4Azure, Proceedings of the 4th International Conference on Utility and Cloud Computing, 2011, pp.97-104.

[27] E.L. Lawler, J. Labetoulle, On preemptive scheduling of unrelated parallel processors by linear programming, Journal of the ACM 25 (1978) 612-619.

[28] 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.
[29] 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.
[30] T. Lim, T.G. Robertazzi. Efficient parallel video processing through concurrent communication on a multi-port star network, Proceedings of the 40th Conference on Information Sciences and Systems, 2006.
[31] J. Lin, C. Dyer, Data-Intensive Text Processing with MapReduce, Morgan & Claypool, 2010.
[32] Lp_solve reference guide, 2010, http://lpsolve.sourceforge.net/5.5/.
[33] R. Pike, S. Dorward, R. Griesemer, S. Quinlan, Interpreting the data: Parallel analysis with Sawzall, Scientific Programming 13 (2005) 277-298.
[34] K. van der Raadt, Y. Yang, H. Casanova. Practical divisible load scheduling on grid platforms with APST-DV. Proceedings of the 19th IPDPS'05, page 29.b, 2005.
[35] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, Evaluating MapReduce for Multi-core and Multiprocessor Systems, Proceedings of International Symposium on High Performance Computer Architecture (HPCA) 2007, pp. 13-24.
[36] T.Robertazzi, Ten reasons to use divisible load theory. IEEE Computer 36 (2003) 63-68.
[37] 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.
[38] A. Schrijver, Combinatorial Optimization: Polyhedra and Efficiency vol. A, vol. 24 of Algorithms and Combinatorics, 2003, Springer.
[39] T. White, Hadoop: The Definitive Guide, O'Reilly Media 2012.
[40] H. Yang, A. Dasdan, R.-L. Hsiao, D.S. Parker, Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters, Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, 2007, pp. 1029-1040.
[41] 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/.

Appendix

Theorem 4.1. *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 interchange argument. Assume that in schedule σ_1 for 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

$$\alpha_i = \alpha_i^{(1)} + \alpha_i^{(2)}, \tag{22}$$

where $\alpha_i^{(1)}$ is the amount of load processed by P_i until the completion of computations on P_{i+1} (see Fig. 13a). Hence,

$$A\alpha_i^{(1)} = S + A\alpha_{i+1}. \tag{23}$$

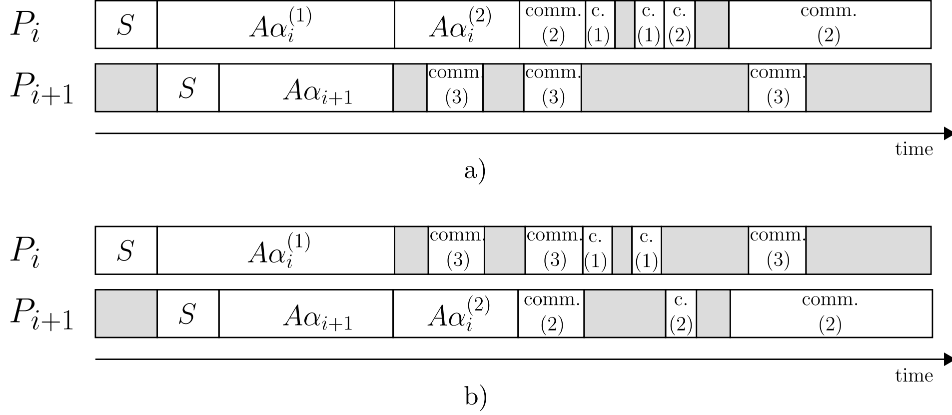

Figure 13. Communication pattern in schedules a) σ_1 and b) σ_2. Labeling (i) of communication intervals is explained in the main text.

and

$$\alpha_i^{(1)} \geq \alpha_{i+1}. \tag{24}$$

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 (cf. Fig. 13b). 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 set $\mathcal{I}$ of intervals in which P_i sent load to reducers and which did not overlap with any communications from P_{i+1} in σ_1, such that the total length of these intervals allows for sending load of size $\alpha_i^{(1)} - \alpha_{i+1}$, i.e.

$$\sum_{I \in \mathcal{I}} |I| = g_0 C(\alpha_i^{(1)} - \alpha_{i+1}). \tag{25}$$

Such a choice is always possible because the total length of 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 $g_0 C(\alpha_i^{(1)} + \alpha_i^{(2)} - \alpha_{i+1}) \geq g_0 C(\alpha_i^{(1)} - \alpha_{i+1})$. Note that $\mathcal{I}$ may be chosen in many different ways.

Let us introduce the following labeling of 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. 13a).

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 (see Fig. 13b). The total length of intervals marked with 2 is $g_0 C(\alpha_i^{(1)} + \alpha_i^{(2)}) - \sum_{I \in \mathcal{I}} |I| = g_0 C(\alpha_{i+1} + \alpha_i^{(2)})$. The total length of intervals

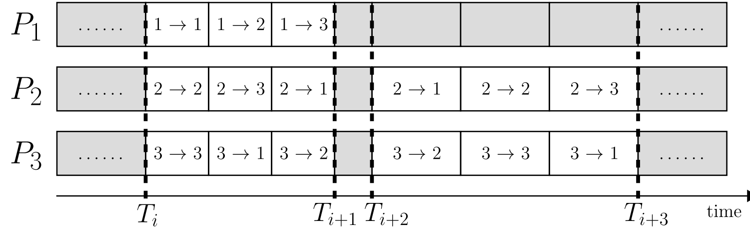

FIGURE 14. Scheduling mapper to reducer computations in σ_2, for $m = 3$, $r_1 = 3$. Notation $j \to k$ stands for: mapper j communicates with reducer k in layer 1.

labeled with 3 is $g_0 C \alpha_{i+1}$. 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 the same 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 communication schedule are necessary to guarantee that each reducer receives a proper amount of load from processors P_i and P_{i+1}. The communication schedule will be changed globally, not only for processors P_i and P_{i+1}. Let us define $T_1 < \cdots < T_q$ 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 $I_{i1}, \dots, I_{ir_1}$ of equal length (cf. Fig. 14). Let $P'_1, \dots, P'_{m'}$ be the processors which send some load in interval I_i in σ_1. Note that necessarily $m' \leq l$ and $m' \leq r_1$. In schedule σ_2 processor P'_j will communicate with reducers $j, j+1, \dots, r_1, 1, \dots, j-1$ in intervals $I_{i1}, \dots, I_{ir_1}$, correspondingly (cf. Fig. 14). 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 in schedule σ_2 as in σ_1 and each reducer receives the same amount of load from any given mapper. Therefore, schedule σ_2 is feasible and is not longer than 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 mappers finish computations in the FIFO order. □

Theorem 4.2. *Set $\mathcal{F}$ and hence a feasible flow always exist.*

Outline of the proof. 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. 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 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 c_X denote the number of critical vertices in X, and c_Y the number of critical vertices in 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, and w_c be the sum of edge weights in G_c. We need to show that there is always a matching M_c in G such that

i) M_c covers all critical vertices,
ii) M_c has size at most l, and
iii) if the bisection width limit is active, then the size of M_c is exactly l.

By Lemma 6.1 a matching satisfying the above condition i) always exists. Note that if $c_X + c_Y \leq l$ then this 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 ii). We prove that it is the case in Lemma 6.2. In Theorem 6.3 we use this fact to prove that there exists a matching satisfying both conditions i) and ii) given above. Finally, in Theorem 6.4 it is shown that if the bisection width limit is active, then a matching satisfying conditions i), ii) and iii) exists.

Lemma 6.1. *A matching in G covering all critical vertices always exists.*

Proof. This follows directly from the proof given in [10,27] for the algorithm solving problem $R|pmtn|C_{max}$. □

Lemma 6.2. *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 G is not smaller than $\sum_{j \in X_c} w_{jk} + \sum_{k \in Y_c} w_{jk} - w_c$. Hence, by (5), (15)

$$\sum_{j \in X_c} w_{jk} + \sum_{k \in Y_c} w_{jk} - w_c \leq l. \tag{26}$$

As the sum of weights of edges incident to a critical vertex in G is equal to 1, we obtain from (26)

$$w_c \geq c_X + c_Y - l. \tag{27}$$

Now consider a 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 (27) the minimum vertex cover of G_c has at least $c_X + c_Y - l$ elements. By König's theorem, the size of 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. □

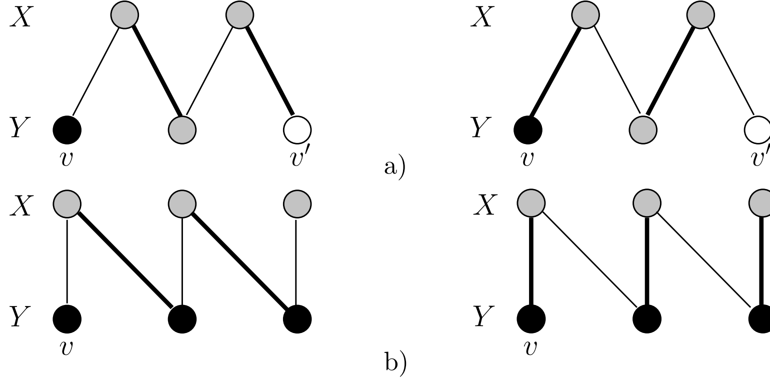

FIGURE 15. Augmenting matching M. Black nodes are critical, non-critical nodes are white, gray nodes may be critical or not. The bold edges are in M. The left figure is the initial matching M, the right figure is the augmented matching. a) case a - alternating path starting in v and finishing in non-critical $v' \in Y$, b) case b - augmenting path starting in v.

Theorem 6.3. *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 6.1. 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 (case 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.15a), or (case b) there exists an odd length M-augmenting path π_2 starting from v (Fig.15b). 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 6.1, 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 b (Fig.15b). Analogous reasoning can be applied to 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. 15a,b). The size of the initial matching M was $e_m \geq c_X + c_Y - l$ by Lemma 6.2. 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, having at most $e_m + c_X + c_Y - 2e_m \leq l$ edges. □

Theorem 6.4. *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 6.3 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. □