Divisible Job Scheduling in Systems with Limited Memory

Paweł Wolniewicz

2003

Table of Contents

Chapter 1

Introduction

The need for computing power caused that parallel and distributed computing gains popularity over the recent years. Contemporary applications have very high computing power requirements. One of the most resource demanding applications are genotype sequencing, quantum chemistry and Earth simulation. In most of the cases a single computer is not able to provide computing power satisfactory for needs. Therefore parallel and distributed algorithms and application, which can run on hundreds of processors in the sam time, are necessary. Some parallel applications are run in dedicated supercomputing environment (e.g. NEC supercomputer house for Earth Simulation project). In other projects application are run on existing clusters of computers, or even on personal computers (PCs) made available by the volunteers in Internet.

In the seventieth and eightieth processors used in supercomputers were very expensive and powerful in comparison to the popular processors used in home and office computers. Over the years the technology of processors evolved and now most often supercomputers are build from similar processors and components as PCs. Almost all supercomputer vendors offer machines built from typical commodity-of-the-shelf components. The main difference between PCs and supercomputers now is the number of processors.The fastest supercomputers can be build of hundreds and thousands of processors. One eminent example is the ASCI initiative which resulted in creation

of supercomputers that are in the top of the TOP 500 Supercomputers list [1]. Some of them have several hundreds of processors. For example: .

Recently also grid technologies developed. The idea of the grid is to connect existing computers into some kind of distributed supercomputer. In this way it is easy to achieve very large computing power using standard equipment. In some projects computing power comes from thousands of PC made available by volunteers. Some example are SETI@home project [2], Mersenne Number Project [3], Entropia [4], Distributed Net [5], Folding@home [6] and other.

It should be obvious now that it is necessary to use parallel and distributed algorithms to speed up computations. Creating fast and effective parallel algorithms requires appropriate models of computation. In order to precisely simulate a complex parallel or distributed environment, many parameters and dependencies should be taken into account. A detailed model can be precise but useless because of its complexity. To many parameters result in clutter and obfuscation. Detailed models may complicate understanding of the fundamental phenomena taking place. Therefore, there is a need for specific models of parallel computations that can be a good compromise of detail and correctness.

In this work we consider the divisible job processing model. Divisible jobs can be divided into parts of arbitrary sizes and the parts can be independently processed in parallel. This means that the granularity of the computations is fine, and can be neglected as not restricting the load size selection. There are no order constraints and all parts can be processed in parallel.

This model applies, for example, to processing large measurement data sets (e.g. SETI@home [2]), data mining: searching databases, text, audio, and video files, also to some applications of linear algebra, number theory (e.g. Mersenne project [3], Distributed net [5]), simulation, combinatorial optimization [6, 28, 30, 37]. Divisible load theory (DLT) can be also applied in the analysis of distributed storage systems such as video on demand systems [15], storage area networks or network attached storage systems. In this case the distribution of the information can be optimally

geared to the speeds of the communication network, and transfer rate of the storage devices. The divisible job processing model is a very effective tool for analyzing different topologies of distributed environments: linear arrays, stars, rings, trees, meshes and hypercubes. Practical experiments proved that the divisible job processing model complies with the results of running jobs in real environments.

The purpose of this work is to examine the impact of different memory systems on the performance of divisible load processing in various distributed networks. Three kind of memory systems are considered: hierarchical memory, single level memory with limited buffer size, and systems with limited communication buffers.

Before going into further details let us outline the contents of the thesis. In Chapter 3 systems with single memory level are considered. Chapter 4 is devoted to the systems with hierarchical memory. In Chapter 5 systems with limited communication buffers are presented. In Chapter 6 the impact of multi-installment divisible job processing on the performance is examined. Real divisible jobs experiments are presented in Chapter 7. The main notation and symbols used in this work are summarized in Appendix A.

Chapter 2

Divisible Job Model Fundamentals

In this chapter we provide basic assumptions of the divisible load theory. The abstraction of computer systems is presented and the subject literature is shortly reviewed.

2.1 Formulation of the problem

Terms divisible job and divisible load will be used interchangeably in this work. We will use word processor to denote a single processing unit with CPU, memory, disks, and network interface. The words data and load in context of the size of the load processed by processors will be used interchangeably.

We consider a system with a set of $m+1$ uniform processors $\mathcal{P} = \{P_0, \ldots, P_m\}$ with the additional coordinating processor called master or originator. It is assumed that at the beginning of the computation the whole volume of load with size V is located in the memory of the master processor P_0. Originator scatters the load to m processors of the distributed computer network. In every transmission the communication startup time elapses between the initiation of the communication and sending the first byte through the link. Transferring x units (e.g. bytes) of the load over link i lasts $S_i + xC_i$ units of time (e.g. in seconds). Thus, the communication delay includes constant startup time S_i and linear component xC_i depending on the amount of the transferred data. Computing x units of the load takes xA_i units of time

on processor P_i. Sizes of the data pieces sent to processors are denoted by $\alpha_0, \cdots, \alpha_m$. α_i is the fraction of the volume V which is sent to processor P_i. Sum of all α_i pieces is equal 1.

For a given computing environment (described by parameters C_i, A_i, S_i, V, m) the values of $\alpha_0, \cdots, \alpha_m$ should be such that the length of the schedule, denoted as C_{max} is the shortest possible.

In the following sections we introduce scattering algorithms dedicated to three message routing topologies: a star, an ordinary tree and a binomial tree.

2.2 System architecture

Here we make several assumptions on the nature of the computer system and the application.

The originator P_0 can participate in computation or it can only distribute the load to other processors. It does not change significantly the model because if the originator computes, then it can be represented as an additional processor in a model with the originator communicating only. And vice versa, if we assume that the originator is computing, but in reality it is not, then its computation rate can be represented as $A_0 = \infty$. Thus, the originator would receive no load and $\alpha_0 = 0$.

It is possible to distinguish two kinds of processing elements depending on the ability to communicate and compute in parallel. The processors *with communication front-end* are equipped with the communication hardware, which allows for transmitting and computing in parallel. Processors *without front-end* can either communicate or compute.

It is accepted in the divisible load theory [19, 16, 35] that the time of returning the results of the computations to the originator can be neglected. It does not mean, however, that we exclude applications returning some results. The returning of the results can be incorporated in the divisible load model (cf. applications in [9, 28, 30, 37]). This assumption is made for the sake of simplicity of mathematical modeling

and conciseness of the presentation.

In the following sections we define various communication topologies and strategies.

2.2.1 Routing topologies

In this section we present three archetypal routing topologies used in the load scattering. These topologies can be embedded in various parallel computer interconnection topologies.

Star topology

In this topology the originator P_0 is located in the center of the structure (cf. Fig. 2.1). All the messages are routed from the originator to the processors, or from the processors to the originator. Only one message can be sent or received by the originator at a time. This kind of communications is typical of the bus and can be considered as equivalent to the star interconnection. Hence, the star topology can represent a network of workstations, master-slave, or client-server computations [37]. This interconnection applies also to the networks in which the originator is able to address each slave processor directly, and send load to it. If it is the case then the intermediate communication nodes (if any) can be represented only as an additional communication delay. Therefore, star topology can be called direct communication topology, as well. This is especially justified in communication networks with wormhole routing or circuit switching, for which communication delay does not depend significantly on the distance between the sender and the receiver.

In the star network originator sends chunk $\alpha_i V$ of load to processor P_i. Immediately after receiving its load P_i starts computing, while the originator immediately starts the communication with the next processor.

The process of communication and computation is presented in Fig. 2.3. Processing rate of processor P_i is denoted A_i. Communication links are characterized by startup time S_i and communication rate C_i. If the memory buffer of the processors

is limited then its size will be denoted B_i for buffer of processor P_i, and D_i for the communication buffer of link i.

Star topology is also called a single level tree network in the divisible load literature.

Ordinary tree

Ordinary tree (see Fig. 2.2) is a graph-theoretic structure used in many broadcasting and scattering algorithms for various interconnection topologies [51]. The originator is not able to communicate with all processors directly. Therefore, intermediate processors are used to relay the load to other processors. We consider regular balanced tree in which nodes have out-degree p. p is also the number of ports in each processor that can be used simultaneously to activate other processors. If a processor receives some load to relay, it divides it into p equal parts and retransmits them to the still inactive processors. The set of processors in equal distance from the originator (measured in the number of hops), will be called a *layer*. Let h denote the height of the tree. We assume that the communication medium, and processors are homogeneous. Processing rate is denoted A, communication parameters are denoted S, C. The sizes of memory buffers, if limited, will be denoted B. Processors in the same layer perform the same actions, communicate and compute synchronously. Hence, processors in a layer performs identically and we seek load fractions $\alpha_0, \ldots, \alpha_h$ assigned to each of the processors in layers $0, \ldots, h$ respectively. The ordinary tree has $m = \frac{p^{h+1}-1}{p-1}$ processors, for $p > 1$.

Note that linear array of processors, a.k.a. a chain topology is just an ordinary tree topology with degree $p = 1$

Binomial tree

Binomial tree has been introduced in [57] as a broadcasting structure for a 2-dimensional mesh, and as a scattering structure for 1-, 2-, and 3-dimensional meshes in [27, 35, 36]. Binomial tree (cf. Fig. 2.4) is a tree in which nodes have out-degree p. Each processor

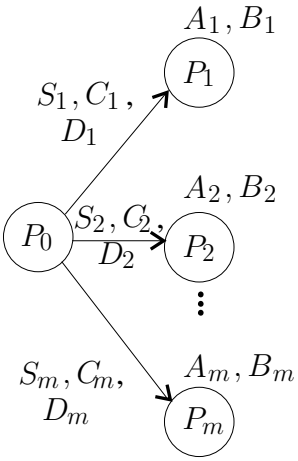


Figure 2.1: Star interconnection.

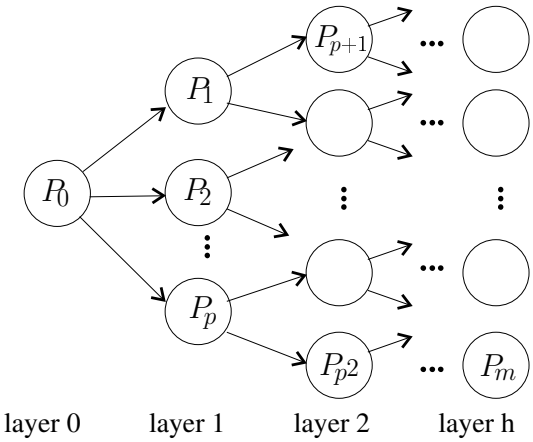


Figure 2.2: Tree interconnection.

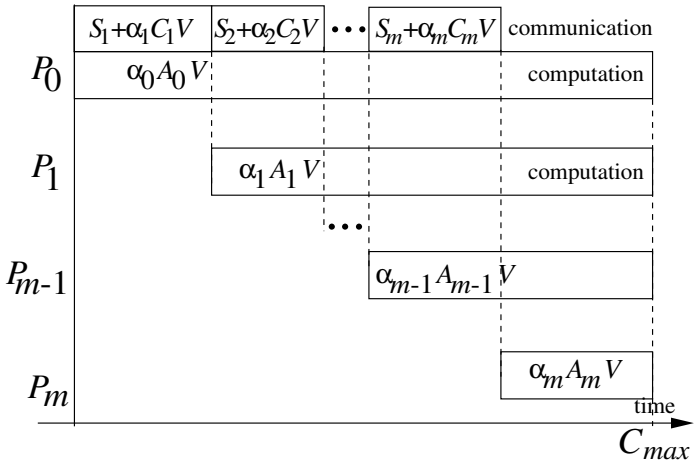


Figure 2.3: Communication and computations in star interconnection.

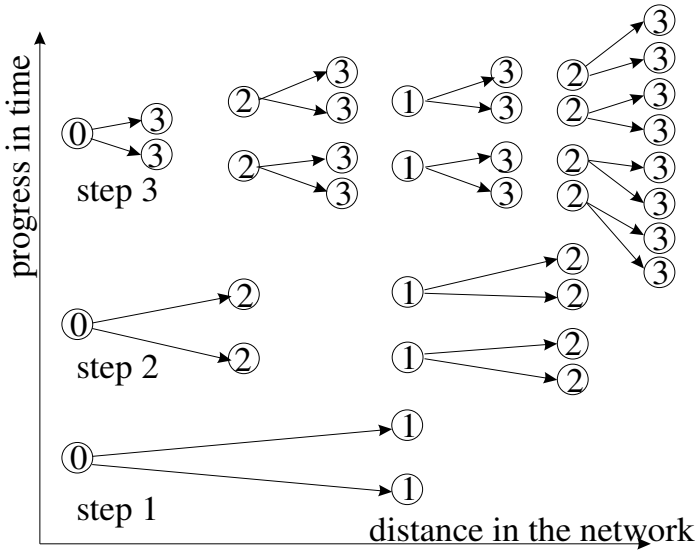
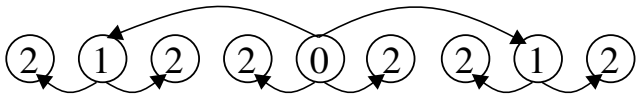


Figure 2.4: Binomial tree. Numbers indicate layers.

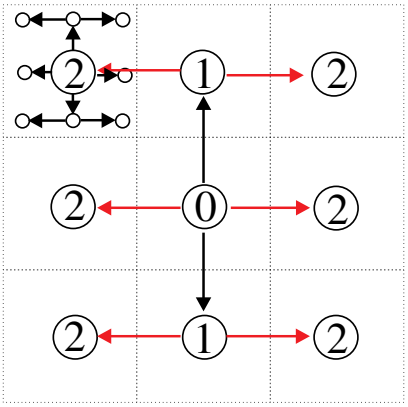
(node) in level $0, \ldots, i-1$ activates p new processors on level i, for $i = 1, \ldots, h$. The set of processors in the same level of the binomial tree will be called a *layer*. Binomial tree takes advantage of the communication delay structure typical of circuit switching and wormhole routing. For these two commutation methods communication delay does not depend significantly on the distance covered by the message. Therefore, it is advantageous to send the load to processors in physically large distance from the originator first, and then to redistribute the load locally in a smaller sub-network. Note that a processor in layer i receives load to be redistributed among its descendants in layers $i+1, \ldots, h$.

Chain, mesh, torus, hypercube or multistage interconnection can be modeled using binary tree topology. Examples of embedding binomial trees in different interconnections are shown in Fig. 2.5.

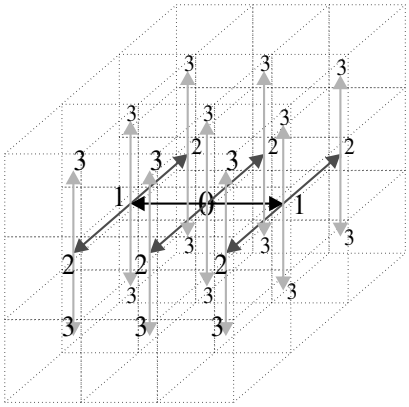
We assume that the communication medium, and processors are homogeneous. Therefore, processors in the same layer work synchronously, i.e. perform the same actions simultaneously. As in the ordinary tree we assume that processors are able to divide the received message into equal parts and simultaneously redistribute the parts to its p ports. The number of processors in a binomial tree with layers $0, \ldots, h$



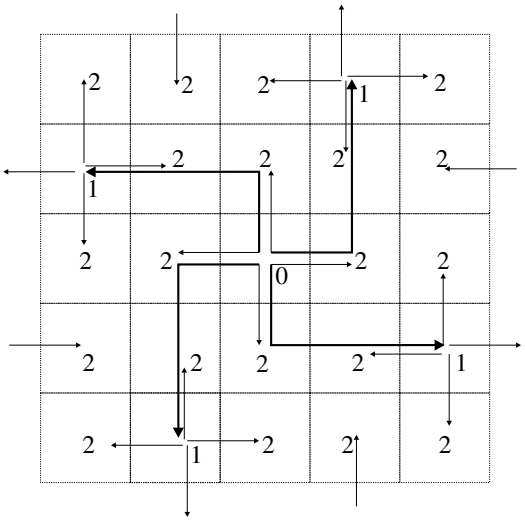
(a) Chain p=1



(b) 2-dimensional mesh p=2



(c) 3-dimensional mesh p=2



(d) 2-dimensional p=4

Figure 2.5: Examples of embedding binomial trees into different topologies.

is $m = (p+1)^h$. There are $p(p+1)^{i-1}$ processors in layer $1 \leq i \leq h$. Our goal is to find distribution of the load $\alpha_0, \ldots, \alpha_h$ among the processors in layers 0 to h respectively, such that schedule length C_{max} is minimal.

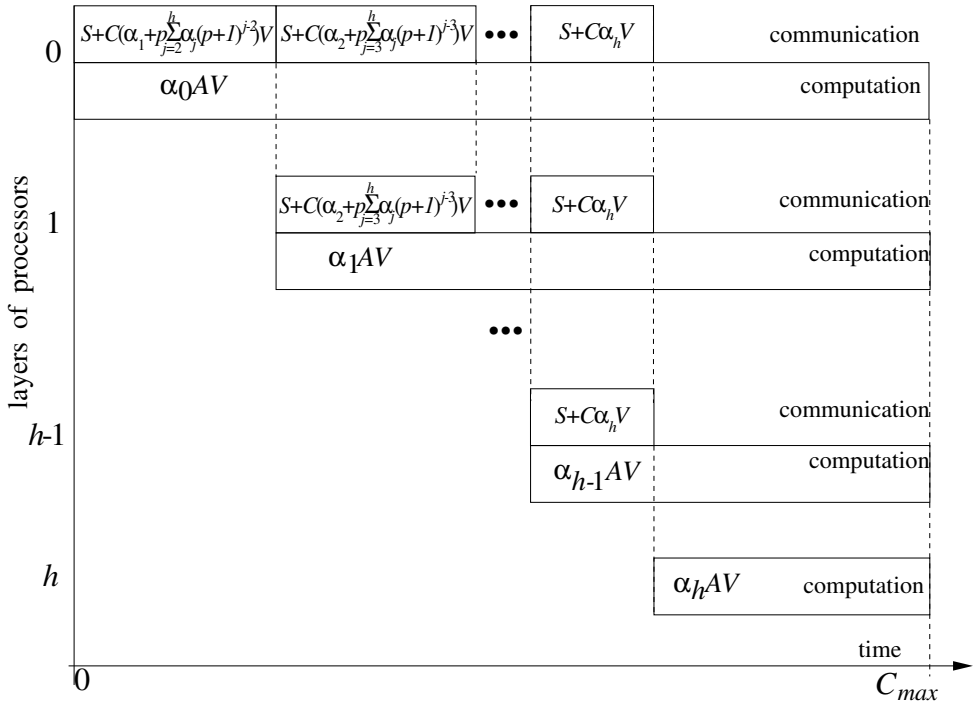
2.2.2 Layer activation order

For binomial trees two different ways of activating the layers have been proposed: The natural order of the layers called NEAREST LAYER FIRST (NLF) [27], and the order of decreasing number of processors in the layer called LARGEST LAYER FIRST (LLF) [45]. Fig. 2.6 shows diagrams of communication and computation for strategies NLF, LLF, respectively. In NLF (cf.Fig. 2.6(a)) layers are activated in the order of the growing distance from the originator layer and receive the load for themselves and for their descendants. Immediately after receiving the load processors start processing their share of the load, while the rest is sent to the following layers. Thus, processors start computing in the order of the layer number. In LLF strategy (cf. Fig. 2.6(b)) the layers start computing in the order $h, h-1, \ldots, 1$. To activate some layer i the intermediate layers $0, \ldots, i-1$ transfer the load to layer i, but do not compute. It was demonstrated in [45] that LLF is optimal activation sequence for binomial trees when there are no memory limitations in the computer system. It has also been demonstrated in [38] that it is not optimal when processors have limited memory buffers. We consider it also in Section 3.2.2. Both layer activation orders can be applied to the ordinary tree topology, too. We write about it in more detail in Section 5.2.2. Observe that the layer activation order does not apply to the star interconnection because all processors are directly accessible from the originator.

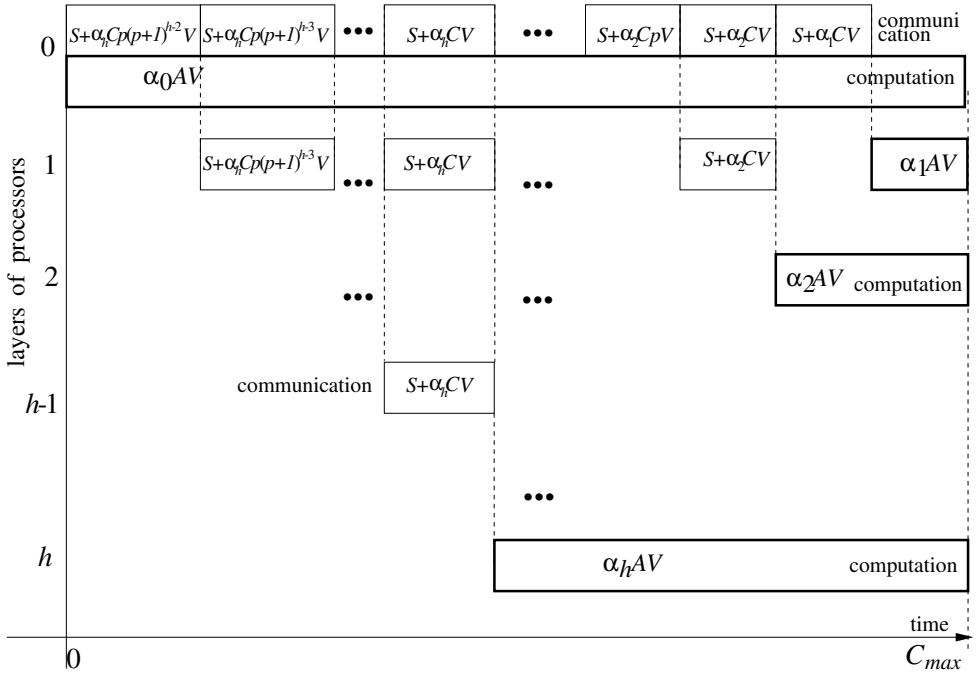
The actual communication interconnections will be modeled using the three topologies described above.

2.2.3 Single-installment and Multi-installment processing

In most of the works regarding divisible jobs, single-installment processing has been assumed. This means that every processor receives its portion of the load only once.



(a) NLF



(b) LLF

Figure 2.6: Layer activation order in binomial tree.

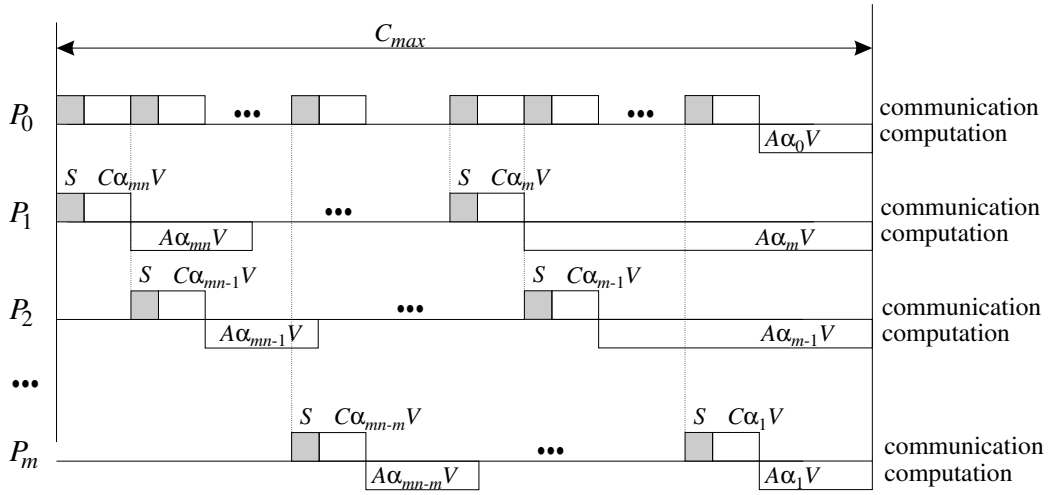


Figure 2.7: Gantt chart for multi-installment processing.

If processors receive data more than once, the processing is called *multi-installment*. We consider the regular type of multi-installment divisible job processing in this work. In this model data are sent to the processor many times and the processors are repeatedly activated in the same order. The load is sent in n cycles, so that after sending a piece to the last processor P_m the next piece is sent to the first processor P_1. As a result every processor gets an equal number of data pieces to process. In a system without communication front-end the originator P_0 starts computing its share of the load after the communications with all processors $P_1, \cdots, P_m$. An example Gantt chart for regular multi-installment processing in heterogeneous system is shown in Fig. 2.7.

In general processors can be activated different number of times and in any order. This kind of irregular activation is not the subject of this work.

Also in multi-installment processing we assume a nonzero startup time. With the zero startup time it is possible to prove that communications should be done in the infinite number of steps, and therefore, it is unrealistic.

2.3 Survey of the Earlier Results

In this section we give a short survey of the divisible load theory (DLT) literature.

Initially the divisible load processing model was used to analyze the trade-off between computation and communication in a distributed network of intelligent sensors. Divisible load model has been introduced in [32], where a linear array of intelligent sensors was considered. The problem was to find optimum balance between advantages of distributed computations on the measurement data, and the costs of communication. The same problem for bus network of sensor driven processors was considered in [11]. Later on, divisible load model has been generalized and extended in various directions.

Divisible jobs scheduling can be applied to many interconnection topologies. It was thoroughly studied for linear arrays [42, 44], busses [10, 12, 62, 64], trees [9, 10, 17, 33, 31, 50], 2D-meshes [24], 2D-toroidal meshes [27], 3D-meshes [36, 46], hypercubes [23, 47] and partitionable networks [52].

The model of communication delay has been generalized by the inclusion of the communication startup costs in [25]. The influence of startup costs on the time performance was also analyzed in [21]. The problem of scheduling divisible load with taking into account the processor release times at the time of load origination was considered in [20, 35]. Both the load distribution and results collection overheads were taken into account in [9, 25, 37].

Divisible load model was capable of incorporating sophisticated communication methods: distributing data in multiple installments [18, 19, 38, 69]. In [18] closed-form solutions were derived for homogeneous single-level tree networks. Subsequently the multi-installment strategy was applied to linear networks and closed-form solutions for processing time for homogeneous networks was presented in [19]. In [69] multi-installment processing with startup costs has been studied and maximum possible gain of regular multi-installment processing for bus networks has been derived. Multi-installment processing as a result of limited memory buffers is considered in [38]. The most of the studies consider the situation that only one load (i.e. one job)

is available for processing. This assumption was relaxed in [63] and a scheduling algorithm employing single installment strategy with FIFO order of the tasks was presented.

Memory limitations at the processor side have been considered in [22, 38, 53]. In [53], the issue of finite size buffers at the processors is addressed for the case of single level tree networks. An off-line algorithm, referred to as Incremental Balancing Strategy(IBS) was proposed. It generates load fractions in an incremental fashion. In each increment distribution of the load is found for processors with available memory according to the standard divisible load theory methods [19], without taking the memory constraints into account. Then, the distribution of the load is scaled proportionately such that at least one buffer is filled completely. The remaining available buffer capacities are the memory sizes in the next increment. It has been also demonstrated that the rule for optimum processor activation sequence proposed in [19] does not work in the case with limited memory. IBS algorithm is not optimal, which will be shown in Chapter 3 using linear programming approach. In [22], approximation algorithms were proposed to generate and round the load fractions for single installment as well as for a strategy in which the load is constrained to have at most K partitions. Ultimate performance bounds were derived for these strategies with integer approximation.

Despite its ability of analyzing intricate details of distributed computer systems, divisible load model remained computationally tractable. In many cases it was even possible to find analytical solutions of the considered models. In [32] a heterogeneous linear network of processors was considered. Under the assumption that all processors stop computation at the same time an algorithm was developed to find the optimal load fractions. The proof that this assumption is a necessary and sufficient condition for obtaining optimal processing time in linear networks appeared in [60]. An analytic proof of this assumption in bus networks without the startup times is presented in [62, 64]. In the case of single-level tree networks, a closed-form expression for the processing time and an algorithm to obtain an optimal tree configuration appeared

in [17, 50]. The optimal sequencing and optimal network arrangement were considered in [17, 23]. For homogeneous linear networks, a closed-form expression for the processing time was presented in [43] and for tree, bus, and linear networks asymptotic solutions have been derived [13, 43]. A study on arbitrary tree networks [9] presents an analytical treatment in deriving optimal sequences using the concepts of equivalent processors. The existence of the closed form analytical solutions is an advantage of the divisible load model over other deterministic scheduling models which are computationally intractable [40].

System parameters used by divisible load theory can be easily obtained in practice. Not only, was the divisible load model successful in theoretically analyzing distributed computer systems, but also its predictions have been confirmed in real computer systems [28, 30, 37, 41]. We write about it in Chapter 7. Furthermore, divisible load model was a base for analyzing multimedia retrieval systems [15], image processing [9], and cost optimization in the design of distributed systems [65]. In [41] a load distribution strategy is designed and analyzed to carry out matrix-vector product computations on a cluster of workstations. Also, this study shows the applicability of divisible load theory to design strategies that are suitable for a network of workstations. The objective of minimizing the monetary costs involved in the process of divisible load processing was analyzed in [65, 61]. The study of monetary costs minimization using DLT resulted in US patent no. 5889989 for a load sharing controller for optimizing monetary costs [61].

It can be concluded that divisible load theory is a new versatile paradigm of distributed computing. Surveys on divisible load processing can be found in [16, 19, 26, 35]. A short summary of the divisible job approach from the more than ten years perspective can be found in [59].

Chapter 3

Systems with Single Memory Level

In this chapter we analyze computer systems with limited memory sizes. Each processor can hold only an amount of load limited by the size of available memory. In clusters of workstations it appeared [37] that the linear dependence of processing time on the size of work is satisfied only if the computations are restricted to the core memory (RAM). Larger work chunks imply using virtual memory. When virtual memory is used, the dependence of processing time on the amount of data becomes more complex. Also the processing speed of the computers is lower. Hence, for efficiency reasons it is preferable to avoid using virtual memory and restrict the load to limited amount B_i of core memory available at processor P_i. Values of B_i are determined by the computing environment and are constant. To focus on direct impact of the buffer sizes relative to the volume V we introduce also variables B'_i which denote buffer sizes relative to the value of V, i.e. $B'_i = \frac{B_i}{V}$.

We assume that the critical restriction on the size of memory is put during the computation phase. It can be the case of problems where small data sets are unpacked or big data structures arise in computation from small amount of the input data. Therefore, the size of communicated message is not limited otherwise than by the memory capacity of the receiver.

In the following sections we study the computational complexity of the problem. Than we analyze the problem of optimum load distribution under the assumptions of

a fixed and arbitrarily chosen processor activation sequence.

3.1 Complexity of divisible job scheduling with limited memory buffers

In this section we will prove that scheduling divisible load in systems with limited memory is NP-hard.

Theorem 3.1. *Scheduling a divisible job in a star network with limited memory buffers is* **NP***-hard.*

Proof. We show that our problem is **NP**-hard by Turing reduction of the PARTITION problem:

PARTITION

Given set $E = \{a_1, \ldots, a_q\}$ of integers decide if there exists set $E' \subset E$, such that $\sum_{j \in E'} a_j = \sum_{j \in E - E'} a_j = \frac{1}{2} \sum_{j=1}^{q} a_j = L$. Without loss of generality we assume that all a_j are even (which can be achieved e.g. by multiplying all a_i's by 2).

The reduction consists in the construction of the divisible job instance on the basis of the PARTITION instance. The first instance can be answered positively if and only if for the second one the answer is positive. Construction of the scheduling instance is as follows: $m = q + 1$, $V = L^6 + L$, $C_1 \ldots C_m = 0$, $S_i = a_i$, $A_i = \frac{L}{a_i}, B_i = a_i$ for $i = 1, \ldots, q$, $S_m = L, C_m = 0$, $A_m = \frac{1}{L^6}, B_m = L^6$. The originator P_0 does not compute, i.e. $A_0 = \infty$. We ask if it is possible to process volume V of load on the above network in time at most $2L + 1$.

Suppose the answer to the PARTITION is positive and E' is a set for which $\sum_{j \in E'} a_j = L$. Now, we are able to construct a feasible schedule for the scheduling problem: First, the processors with indices corresponding to the elements of E' are sent load $\alpha_i V = a_i$, for $i \in E'$, in time L. This part of the work is completed not later than at $2L$. Then, P_m receives load L^6, and completes at $2L + 1$ (cf. Fig. 3.1).

Now suppose the answer to the scheduling problem is positive. This means that

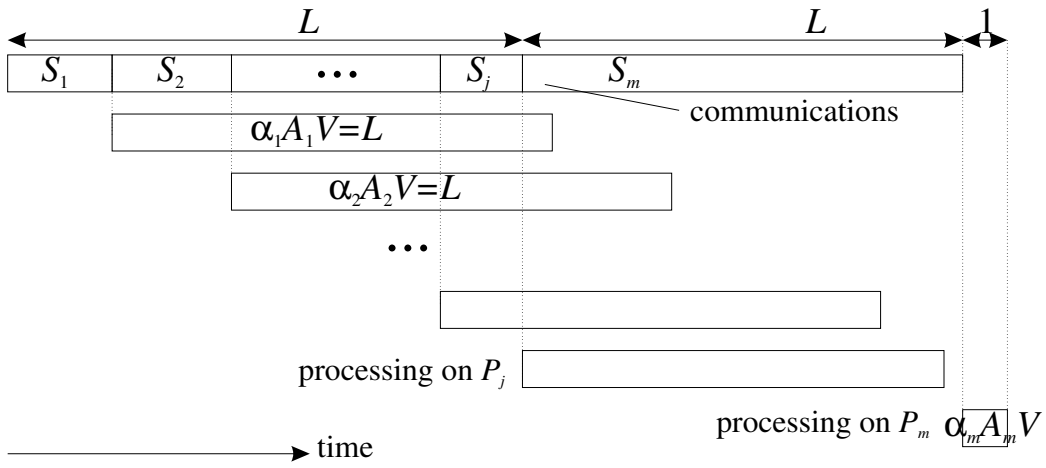


Figure 3.1: Illustration to the proof of Theorem 3.1.

$L^6 + L$ units of load can be processed in $2L + 1$ units of time. Consequently P_m must be used because only P_m has sufficient memory size. Communication with P_m takes L units of time and $L+1$ units of time remain available for communication with other processors in some set F. Since all S_i are even (because a_i are even) only L units of time can be used for communications. Communications to processors in F last $\sum_{i \in F} S_i = \sum_{i \in F} a_i \leq L$. On the other hand the processors in set F must compute at least $V - \alpha_m V \geq L$ units of work. Hence, we have $\sum_{i \in F} a_i = \sum_{i \in F} B_i \geq \sum_{i \in F} \alpha_i \geq L$. Together we have $\sum_{i \in F} a_i = L$, and the answer in PARTITION is also positive. □

3.2 Fixed processor activation sequence in systems with limited memory

In this section we propose a new method of finding solutions with guaranteed optimality for the problem of scheduling divisible loads in networks of processors with limited memory and communication startup times. The method introduces mathematical programming to the realm of divisible load theory. We analyze two network types: star and binomial tree. The implications of memory limitations for the performance are studied.

3.2.1 Star

We assume that the originator both communicates and computes, and that simultaneous computation and communication is possible. We assume that the sequence of sending the load to the processors is $P_1, \ldots, P_m$, and is fixed. Our problem can be formulated as a linear program LP SSML (for LP Star Single Memory Level):

LP SSML

minimize C_{max}

subject to:

$$\alpha_i V A_i + \sum_{j=1}^{i} (S_j + \alpha_j V C_j) \leq C_{max} \qquad \text{for } i = 0, \ldots, m \qquad (3.1)$$

$$\alpha_i V \leq B_i \qquad \text{for } i = 0, \ldots, m \qquad (3.2)$$

$$\sum_{i=0}^{m} \alpha_i = 1 \qquad (3.3)$$

$$\alpha_i \geq 0 \qquad \text{for } i = 0, \ldots, m \qquad (3.4)$$

Let us explain the above formulae. We are to minimize schedule length C_{max} by finding values of variables α_i, C_{max} such that: by equations (3.1) each processor completes not later than at C_{max}, by equations (3.2) no processor is assigned more load than the size of its memory, according to equation (3.3) all the load fractions add up to the total load. In the equation (3.1) for $i = 0$ we have $\sum_{j=1}^{0}(S_j + \alpha_j V C_j) = 0$, because no communication is needed.

LP SSML has $m + 2$ variables and $3m + 4$ constraints. The solution of LP SSML is a point in $m+2$-dimensional space. Constraints (3.1), ..., (3.4) restrict the area of admissible solutions to a convex polyhedron. It is known that the optimum solution is located in one of the polyhedron corners. Unfortunately, the location of the optimum depends on the problem instance and no closed-form expression of α_i seems possible. The linear programs can be solved in polynomial time, e.g. in $O(m^{3.5}L)$ time [48, 54] using the interior point methods, where L is the length of the string encoding all the parameters (A_i, C_i, S_i, B_i, V) of LP SSML. Linear programs can be solved by many

public domain and licensed codes. All linear programming formulations in this work were solved by `lp_solve` ver. 2.0 [14], a public domain linear programming code. Our method is more time-consuming but it is also more robust than the IBS heuristic proposed in [53]. Consider Example 3 from [53].

Example. $m = 4$ (i.e. we have originator and 4 additional processors). Processing rates are: $A_0 = 1, A_1 = 5, A_2 = 4, A_3 = 3, A_4 = 2$. Available memory sizes are: $B_0 = 10, B_1 = 20, B_2 = 45, B_3 = 15, B_4 = 30$. Communication rates are $C_1 = 4, C_2 = 3, C_3 = 2, C_4 = 1$. All startup times are $S_i = 0$, for $i = 1, \ldots, 4$. V=100. By solving SSML LP we obtain:

processor	B_i	$\alpha_i V$	communication completion	computation completion
P_0	10	10	0	10
P_1	20	15	60	135
P_2	45	30	150	270
P_3	15	15	180	225
P_4	30	30	210	270

This schedule has $C_{max} = 270$, and is shorter than the one found by IBS algorithm in [53] by 5 units of time. This is so because the optimality of LP SSML formulation is guaranteed, whereas IBS is a fast heuristic. The length of the schedule is determined by the completion of computations on processors P_2 and P_4. P_1, P_2 memories are not fully utilized. Note that in the interval [10,210] P_0 is not computing but only communicating. □

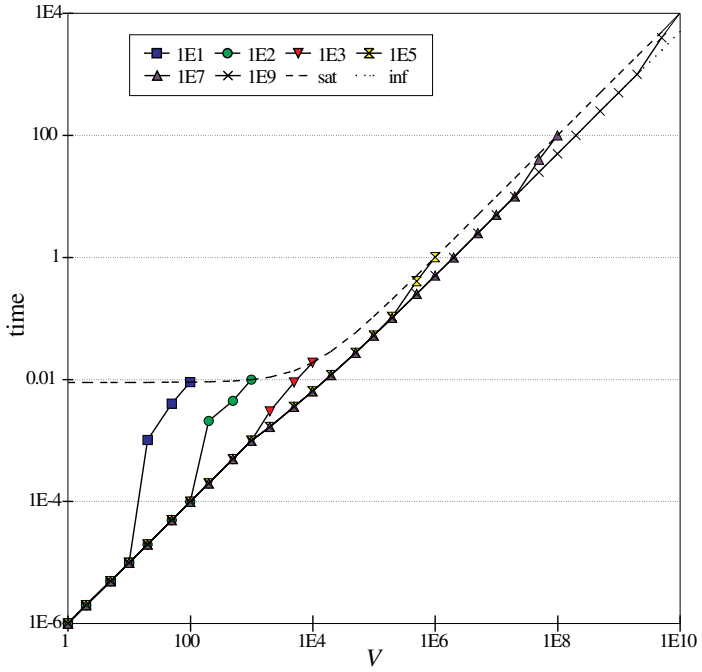
Performance modeling

Now, we will discuss the influence of memory size on the performance of star networks.

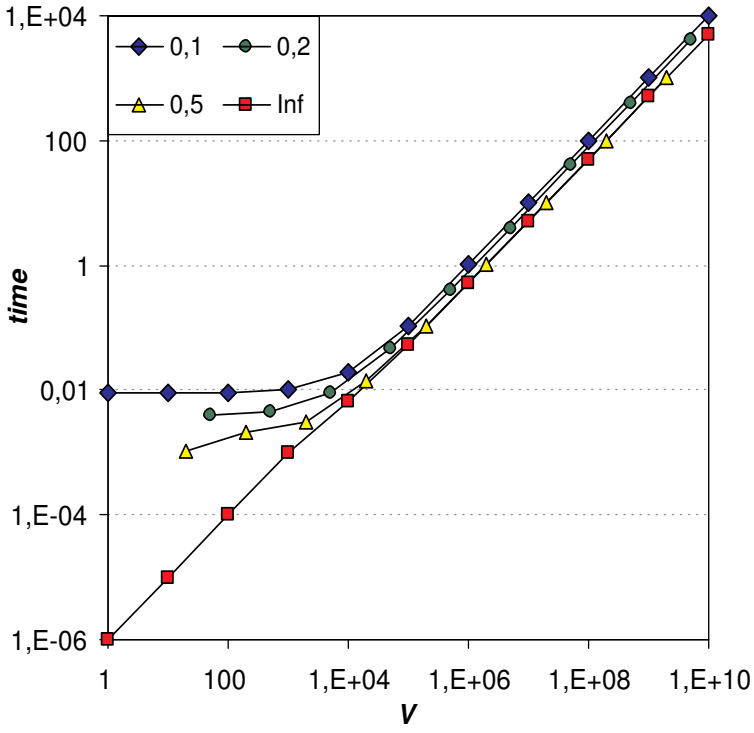
We modeled a system of initiator and 9 identical processors with $A_i = A$=1E-6, connected by identical communication links with startup $S_i = S = 0.001$, $C_i = C$=1E-6. The sizes of available memory were equal B on all processors and the originator. A feasible solution of LP may not exist when the sum of buffer capacities is smaller than V. When a feasible solution existed we recorded the best solution for

one of the number of processors from the range $1, \ldots, 10$ (including the originator). The results of modeling are collected in Fig. 3.2. On the horizontal axis we have size of the problem V, on the vertical axis we have schedule length C_{max}. Plots for memory sizes from $B = 10$ to $B =$1E9 are presented. Fig. 3.2(a) presents schedule length for buffers sizes expressed in the absolute terms (eg. in bytes). As it can be verified with $B = 10$ we can solve problems with size up to $V = 100$ on ten processors. Two more reference lines denoted "sat" and "inf" are depicted in Fig. 3.2(a). Line "sat" represents a system with total memory sizes exactly equal to V. This means that $B = \frac{V}{m+1}$ and memory buffers are *saturated*. Schedule length in a saturated system is $C_{max}^{sat} = \sum_{j=1}^{m}(S + \frac{V}{m+1}C) + \frac{V}{m+1}A$. Line "inf" represents schedule length C_{max}^{inf} in a system with *unlimited memory*. In this case memory size is big enough to hold any loads and we can calculate the distribution of the load according to the classical divisible load theory methods [19, 35]. The plots of processing times for particular memory sizes are located between lines "inf" and "sat". As it can be seen line "sat" approaches line "inf" at $V \approx$1E4. For bigger volumes the two lines form a kind of *tunnel* in which schedule length for the particular memory size must fit. The width of this tunnel shows influence of memory limitation on the schedule length because its upper line represents the system which has just as much memory as needed to hold the load, while the lower line represents a system which has unlimited available memory. Fig. 3.2(b)presents schedule length for buffers sizes expressed in relative terms as the ration of V. By thre small relative difference between lines "inf" and "sat" we can conclude that for large problem sizes V the impact on memory sizes is limited. The position of processing time within the tunnel described above depends on the problem size but only on the buffer sizes relative to the problem size. From the results presented in Fig. 3.2 it can be concluded that for small problems (when V is less than 1E4), memory limitations are very important, because load imbalance may be incurred by insufficient memory sizes.

In order to demonstrate the influence of memory size constraints on C_{max} we collected in Fig. 3.3 values for the "inf" and "sat" cases for various processing rates



(a) Absolute buffer sizes B.



(b) Relative buffer sizes B'.

Figure 3.2: Schedule length for a star network depending on problem size and memory buffer size.

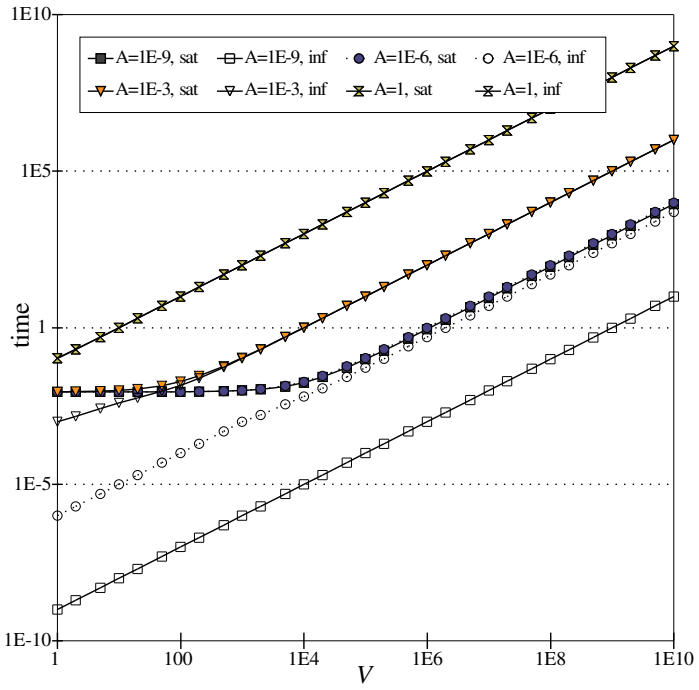


Figure 3.3: Schedule length for a star network with unlimited and saturated memory

A. A system with A =1E-9 has the fastest, while the system with $A = 1$ has the slowest processors. The plots for $A = 1$ are so close that they are drawn one on another. Also the lines for the saturated systems with A =1E-9 and A =1E-6 are so close that they are indistinguishable. For small A schedule length in saturated system is dominated by communications which last $\sum_{j=1}^{m}(S + \frac{VC}{m+1})$. Therefore, the lines for A =1E-9 and A =1E-6 in saturated system are very close. The increase of A results in lines "inf" going up in the Fig. 3.3. The "sat" lines must be located above "inf" lines. As A increases the difference between "inf" and "sat" decreases such that eventually for $A = 1$ they are indistinguishable. It can be observed that for computationally intensive applications which have big A, and big volumes V the difference between "sat" and "inf" cases is small and schedule length is dominated by communication and processing speeds.

Now we will calculate width of the tunnel, i.e. the ratio of schedule lengths in the saturated and unlimited memory cases, on a set of identical processors for big problem

sizes V and computationally intensive applications. An application is computationally intensive if the total computation time AV dominates the communication time CV. In the following we denote by $\rho = \frac{C}{A}$, and $\sigma = \frac{S}{A}$.

Lemma 3.2. *In the star interconnection* $\lim_{V\to\infty,\rho\to 0} \frac{C_{max}^{inf}}{C_{max}^{sat}} = 1$.

Proof. In the saturated case $C_{max}^{sat} = \sum_{i=1}^{m}(S + \frac{VC}{m+1}) + \frac{AV}{m+1} = mS + \frac{V(mC+A)}{m+1}$. It was proved [19] that when results are not returned, and memory is unlimited, all processors must stop computing in the same moment of time. From this observation we infer that time $A_i\alpha_i V$ of computing on processor P_i activated earlier is equal to time $S_{i+1} + \alpha_{i+1}V(A_{i+1} + C_{i+1})$ of sending the load to processor P_{i+1} activated later and computing on P_{i+1}. As we consider identical processors we have: $A\alpha_i V = S + \alpha_{i+1}V(A + C)$. $\alpha_i V$ can be expressed as a linear function of $\alpha_m V$: $\alpha_{m-i}V = \alpha_m V(1+\rho)^i + \frac{\sigma}{\rho}((1+\rho)^i - 1)$ for $i = 1, \ldots, m$. All α_is must add up to 1. Therefore, $V = \sum_{i=0}^{m}\alpha_i V = \alpha_m V\frac{(1+\rho)^{m+1}-1}{\rho} + \frac{\sigma}{\rho^2}((1+\rho)^{m+1} - 1 - \rho - m\rho)$. From which $\alpha_m V$, and $C_{max}^{inf} = A\alpha_0 V$ can be calculated:

$$C_{max}^{inf} = \frac{A[V - \frac{\sigma}{\rho^2}((1+\rho)^{m+1} - 1 - \rho - m\rho)]\rho(1+\rho)^m}{(1+\rho)^{m+1} - 1} + \frac{A\sigma}{\rho}((1+\rho)^m - 1) \quad (3.5)$$

Finally, we have the desired ratio for big V:

$$\lim_{V\to\infty}\frac{C_{max}^{inf}}{C_{max}^{sat}} = \frac{(m+1)\rho(\rho+1)^m}{(m\rho+1)((\rho+1)^{m+1}-1)}. \quad (3.6)$$

For computation intensive applications $A \gg C$ and $\rho \to 0$. After applying de l'Hôpital rule we obtain:

$$\begin{aligned}\lim_{V\to\infty,\rho\to 0}\frac{C_{max}^{inf}}{C_{max}^{sat}} &= \lim_{\rho\to 0}\frac{(m+1)\rho(\rho+1)^m}{(m\rho+1)((\rho+1)^{m+1}-1)} \\ &\stackrel{H}{=} \lim_{\rho\to 0}\frac{(m+1)[(\rho+1)^m + m\rho(\rho+1)^{m-1}]}{m((\rho+1)^{m+1}-1) + (m\rho+1)(m+1)(\rho+1)^m} = 1\end{aligned} \quad (3.7)$$

□

We conclude that in the case of big problem sizes V and computationally intensive applications executed on a set of identical processors, memory limitations are not as

restrictive, for the schedule length, as computation and communication speeds. This observation is confirmed by Fig. 3.2, and Fig. 3.3. On the other hand it should not be forgotten that this result applies to homogeneous computing systems. In heterogeneous systems, the difference between C_{max}^{inf} and C_{max}^{sat} can be arbitrarily big. For example, when a fast processor has small memory buffer and a slow processor has a large buffer then the equivalent speed of the system is dominated by the slow processor in the "sat" case. Furthermore, in practice parameters A, C, S may depend on change with the amount of the assigned load [37]. We discuss it in Chapter 4 and Chapter 7.

From equation (3.6) a width of the tunnel for fixed ρ, and m tending to infinity can be derived:

Lemma 3.3. *In the star interconnection*

$$\lim_{V\to\infty, m\to\infty} \frac{C_{max}^{inf}}{C_{max}^{sat}} = \frac{e^{\frac{1}{k}}}{(e^{\frac{1}{k}}-1)(k+1)}.$$

where $k = \frac{A}{C(m+1)}$.

Proof. Let us assume that $\frac{C}{A} = \rho = \frac{1}{k(m+1)}$. Then from (3.6) we have

$$\begin{aligned} \lim_{V\to\infty} \frac{C_{max}^{inf}}{C_{max}^{sat}} &= \frac{(m+1)\rho(\rho+1)^m}{(m\rho+1)((\rho+1)^{m+1}-1)} \\ &= \frac{\frac{1}{k}(\frac{1}{k(m+1)}+1)^m}{(\frac{m}{k(m+1)}+1)((\frac{1}{k(m+1)}+1)^{m+1}-1)} \end{aligned} \tag{3.8}$$

After observing that $\lim_{x\to\infty}(1+\frac{1}{kx})^x = \lim_{x\to\infty}(1+\frac{1}{k(x+1)})^x = e^{\frac{1}{k}}$ we have:

$$\lim_{V\to\infty, m\to\infty} \frac{C_{max}^{inf}}{C_{max}^{sat}} = \frac{\frac{1}{k}e^{\frac{1}{k}}}{(\frac{1}{k}+1)(e^{\frac{1}{k}}-1)} = \frac{e^{\frac{1}{k}}}{(k+1)(e^{\frac{1}{k}}-1)} \tag{3.9}$$

□

Let us note that k has a practical meaning. If $k < 1$ then the processing rate for all processors Am is less than transmission rate and parallel processing has no sense for such a system. Therefore, k can be treated as a global characteristic of a system.

We finish this section with an observation on the way of activating the processors in the solutions of the linear problem LP SSML. Activation of the processors is ruled by two effects: memory limitations and schedule length minimization. When memory size on one processor is small then more processors must be used, though it is not as efficient as it would be in unlimited memory case. On the other hand, when computation times are short in relation to communication times then it is advantageous to use few processors. Hence, in our performance simulations for A =1E-6 less machines were used for some given volume V than for A =1E-3.

3.2.2 Binomial trees

In this section we consider communications in binomial trees under two activation strategies: Nearest Layer First (NLF) and Largest Layer First (LLF).

Nearest Layer First

The problem of determining optimal distribution of load V in a binomial tree of degree p under NLF strategy can be formulated as the following linear program:
LP NLF:
minimize C_{max}
subject to:

$$\alpha_i VA + \sum_{j=1}^{i}(S + C\alpha_j V + CpV \sum_{k=j+1}^{h} (p+1)^{k-j-1}\alpha_k) \leq C_{max} \quad i = 0, \ldots, h \quad (3.10)$$

$$\alpha_0 + p\sum_{i=1}^{h}(1+p)^{i-1}\alpha_i = 1 \quad (3.11)$$

$$B \geq \alpha_i V \geq 0 \qquad i = 0, \ldots, h \quad (3.12)$$

In LP NLF α_i, for $(i = 0, \ldots, h)$, are variables denoting the amount of load assigned to each processor in layer i. In inequalities (3.10) term $\sum_{j=1}^{i}(S + C\alpha_j V + CpV \sum_{k=j+1}^{h}(p+1)^{k-j-1}\alpha_k)$ is the communication time spent until activating layer

i. Note that layers $1, \ldots, i$ receive load not only for themselves but also the load to be redistributed to layers $i+1, \ldots, h$. Constraints (3.10) ensure that all layers stop computing before the end of the schedule C_{max}. By equation (3.11) all the load is processed, and by (3.12) assignments of the load can be accommodated in the memory buffers of the processors. LP NLF is formulated with the assumption that all h layers are working. However, it may happen that fewer layers will process all the load. In such a case some layers are not assigned any load, but still communication startup time appears in inequalities (3.10). This case is easy to recognize: some layers receive 0 load, and decreasing h reduces C_{max}. Hence, less layers should be used. By binary search over the admissible numbers of layers the appropriate value of h can be found.

Now, we will study performance of NLF algorithm in a binomial tree of degree $P = 4$, and with $h = 7$ layers ($m = 78125$ processors). This tree can be embedded into a 2-dimensional toroidal mesh as described in [27] (see Fig 2.5(d)). We modeled a system with $A = C =$1E-6, $S =$1E-3, and memory sizes from $B = 10$ to $B=$1E9. The schedule lengths C_{max} vs. size of the problem is depicted in Fig. 3.4. Line "inf" represents a system with unlimited memory. Line "sat" represents a system with total memory size equal V. Thus, in saturated case each processor has memory buffer of size $B = \frac{V}{(p+1)^h}$. Schedule lengths for "sat" and "inf" cases are very close to each other in the case of big volumes V. As it was in the case of star topology, the two lines form a tunnel in which plots for particular memory sizes are located. In Fig. 3.5 only "sat" and "inf" cases are depicted for various processing rates A. The behavior is similar to the star topology: For big load volumes V the two lines are parallel. As A increases (e.g. because the application is computationally intensive) the "inf" line moves up until it overlaps with line "sat". Now we are going to calculate the relative width of the tunnel for big V and $\frac{A}{C}$.

Lemma 3.4. *Under NLF strategy in binomial tree* $\lim_{V\to\infty,\rho\to 0} \frac{C_{max}^{inf}}{C_{max}^{sat}} = 1$.

Proof. We will give a formula for the ratio of schedule length C_{max}^{sat} in the saturated case and C_{max}^{inf} in the unlimited memory case. In the saturated case all processors are assigned the same load equal to the buffers size $B = \alpha_i V = \frac{V}{(p+1)^h}$, for

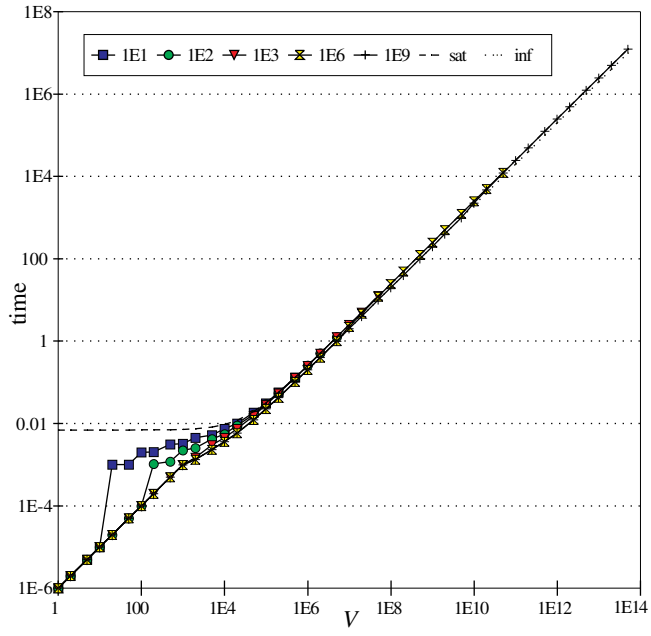


Figure 3.4: Schedule length in a binomial tree under NLF strategy.

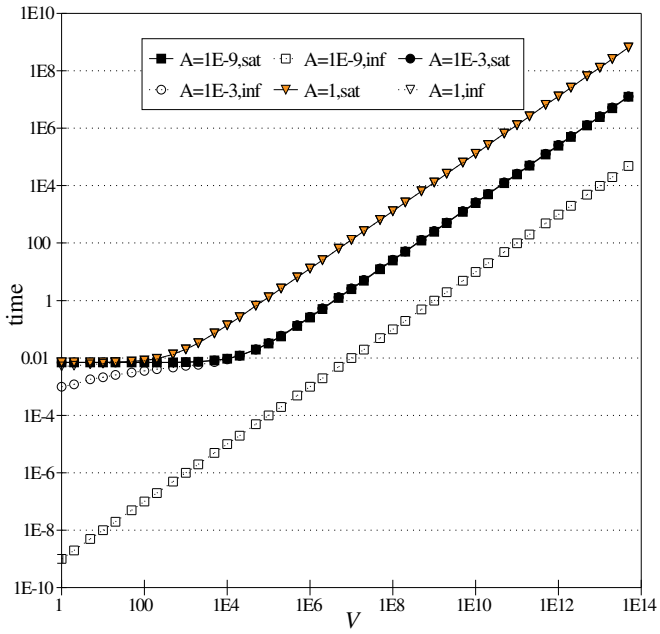


Figure 3.5: Schedule length in a binomial tree under NLF strategy for saturated and unlimited memory.

$i = 0, \ldots, h$. C^{sat}_{max} is determined by the duration of all communications plus processing on layer h. Thus, $C^{sat}_{max} = hS + \frac{CV}{m}\sum_{j=1}^{h}(1 + p\sum_{k=j+1}^{h}(p+1)^{k-j-1}) + \frac{AV}{m}$, where $m = (p+1)^h$ is total number of processors. This formula can be reduced to $C^{sat}_{max} = hS + \frac{VA}{m} + \frac{VC((p+1)^h-1)}{mp}$. The formula expressing C^{inf}_{max} has been derived in [45]: $C^{inf}_{max} = A(V + \frac{\sigma}{p+\rho})\frac{p(p+\rho+1)^{-h}+\rho}{p+\rho} + \frac{\sigma(hp-1)}{p+\rho}$. Hence,

$$\frac{C^{inf}_{max}}{C^{sat}_{max}} = \frac{A(V + \frac{\sigma}{p+\rho})\frac{p(p+\rho+1)^{-h}+\rho}{p+\rho} + \frac{\sigma(hp-1)}{p+\rho}}{hS + \frac{VA}{m} + \frac{VC((p+1)^h-1)}{mp}} \tag{3.13}$$

Since $m = (p1)^h$ it can be verified that
$\lim_{V\to\infty,\rho\to 0}\frac{C^{inf}_{max}}{C^{sat}_{max}} = \lim_{\rho\to 0}\frac{\frac{p(p+\rho+1)^{-h}+\rho}{p+\rho}}{\frac{1}{m}+\frac{\rho((p+1)^h-1)}{mp}} = \frac{\frac{p(p+1)^{-h}}{p}}{\frac{1}{(p+1)^h}} = 1$. □

Thus, in binomial trees spanned in homogeneous computer networks, under NLF strategy, when size V of the problem is big, and the problem is computationally intensive ($\rho \to 0$), the influence of the limited memory is insignificant.

Largest Layer First

In this section we consider a different strategy of activating the layers. According to LLF strategy h is the first layer, and 1 is the last layer activated. We will give a linear program solving the problem of distributing the load optimally in binomial tree under LLF. Then, we compare the results of modeling performance of systems with LLF and NLF scattering methods.

Before formulating a linear program for LLF strategy let us analyze the duration of the communication from the originator to layer i. There are $p(p+1)^{i-1}$ processors in layer i. First, the originator sends over each of its p communication links $p(p+1)^{i-2}\alpha_i V$ load units to layer 1. The remaining load $p(p+1)^{i-2}\alpha_i V$ will be sent to layer i via direct successors of the originator in layers $2, \ldots, i$ (cf. Fig. 5.6). Each processor in layer $j < i-1$ sends $p(p+1)^{i-j-2}\alpha_i V$ units of data to layer $j+1$. The remaining $p(p+1)^{i-j-2}\alpha_i V$ units of the load are sent from layer j to layer i via j's direct binomial tree successors in layers $j+1, \ldots, i$. Finally, layers $0, \ldots, i-1$ send $\alpha_i V$ load units to layer i. Note that all layers communicate synchronously, and the same amounts

of load are sent from active layers to the next activated layer. Total communication time is equal to $Si + C\alpha_i V(1 + p\sum_{j=0}^{i-2}(p+1)^{i-j-2}) = Si + C\alpha_i V(p+1)^{i-1}$. The problem can be solved by a linear program:

LP LLF:

minimize C_{max}

subject to:

$$\alpha_i VA + \sum_{j=i}^{h}(Sj + C(p+1)^{j-1}\alpha_j V) \leq C_{max} \qquad i = 0, \ldots, h \tag{3.14}$$

$$\alpha_0 + p\sum_{i=1}^{h}(1+p)^{i-1}\alpha_i = 1 \tag{3.15}$$

$$B \geq \alpha_i V \geq 0 \qquad i = 0, \ldots, h \tag{3.16}$$

In LP LLF inequalities (3.14) guarantee that all processors finish computing before the end of the schedule. By equation (3.15) all the load is processed, and by constraints (3.16) all processors are able to accommodate the assigned load. It may happen that the assumed number of layers h is too big and a reduction of h results in shorter schedule. Yet, the problem becomes more involved because we send to the larger layer first. A solution of LP LLF may activate layers non-continuously. Some layers may receive load for processing, while the remaining layers would still contribute startup time S in inequalities (3.14), though they receive nothing. We observed that in the solutions of LP LLF layers with higher index (i.e. with more processors) are assigned some load first in consecutive manner (without gaps). Thus, for the given h it is possible to check LP LLF only with the last layers $h, \ldots, h-j$. The best number of utilized layers can be found by binary search over the range of h. In the worst case this procedure must be repeated for various values of h. Hence, the total number of calls to LP LLF needed to find optimum distribution of the load is $O(h \log h)$, where $h = \log_{p+1} m$, and m is the number of available processors. In the following we prove that this strategy leads to optimal solutions because it is always profitable to activate layer $i+1$ (with more processors) before layer i.

Lemma 3.5. *Let C^i_{max} denote schedule length for some volume V assigned to layer i but not to layer $i+1$, and C^{i+1}_{max}, when V is assigned to $i+1$, but not to i. Then, $C^i_{max} > C^{i+1}_{max}$.*

Proof. Let us calculate length of the schedule when layer i is used to process V, but layer $i+1$ is not exploited. Layer i has $p(p+1)^{i-1}$ processors. Thus, $C^i_{max} = S + C(p+1)^{i-1}\frac{V}{p(p+1)^{i-1}} + \frac{AV}{p(p+1)^{i-1}} = S + \frac{CV}{p} + \frac{AV}{p(p+1)^{i-1}}$. Analogously, $C^{i+1}_{max} = S + \frac{CV}{p} + \frac{AV}{p(p+1)^i}$. Hence, $C^i_{max} > C^{i+1}_{max}$ for $i > 0$. □

By the above lemma it is profitable to activate the layers consecutively from the layer with more processors to the layer with less processors (without gaps in between).

We studied the performance of a computer network with embedded binomial tree under LLF strategy. In order to find the shortest processing time over various orders of activating layers we used the result of Lemma 3.5, and increased the number of active layers from the last one to the first. The solution with the smallest schedule length was selected. In general, the behavior of C_{max} under changing V, B, A is very similar to the case of NLF behavior. Schedule lengths in the saturated system and in the system with unlimited memory is presented in Fig. 3.5. Also here a tunnel between "inf" and "sat" cases can be observed. In the following lemma we will show that for big volumes and computation-intensive applications the relative difference between the "inf" and "sat" cases is very small.

Lemma 3.6. *Under LLF strategy in binomial tree* $\lim_{V\to\infty,\rho\to 0} \frac{C^{inf}_{max}}{C^{sat}_{max}} = 1$.

Proof. Schedule length in the saturated case is $C^{sat}_{max} = \sum_{j=1}^{h}(Sj + \frac{VC}{m}(p+1)^{j-1}) + \frac{AV}{m} = S(h+1)h/2 + \frac{V}{m}(C\frac{m-1}{p} + A)$, where $m = (p+1)^h$ is the total number of processors. The formula for C^{inf}_{max} has been given in [45]:

$$C^{inf}_{max} = \frac{AV}{M} + \frac{A\sigma p}{\rho M}\sum_{j=1}^{h}\frac{c_{\pi(j)}-1}{P^\pi_j}(\sum_{i=1}^{j}(h-i+1)P^\pi_{i-1}),$$

where: $M = 1 + \frac{p}{\rho}(1 - \frac{1}{P^\pi_h})$, $c_{\pi(j)} = 1 + \rho(p+1)^{h-j}, c_{\pi(0)} = 1$, and $P^\pi_j = \prod_{i=0}^{j} c_{\pi(i)}$.

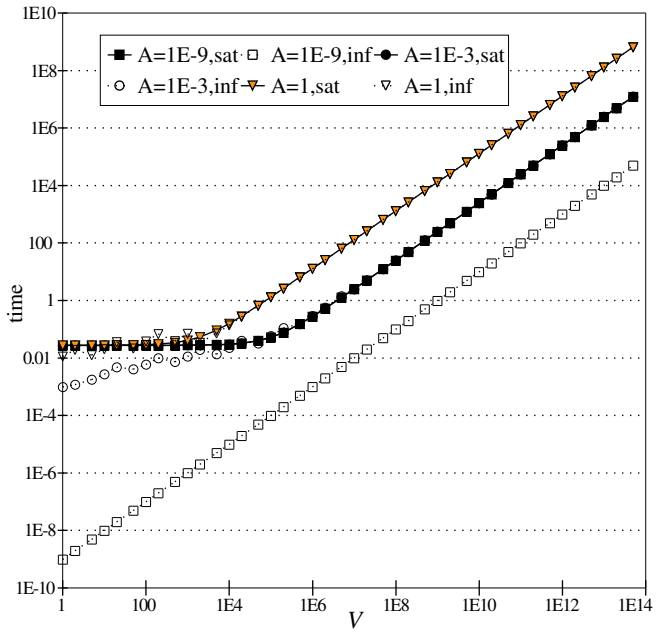


Figure 3.6: Schedule length in a binomial tree under LLF strategy with unlimited and saturated memory

Thus in LLF strategy,

$$\frac{C_{max}^{inf}}{C_{max}^{sat}} = \frac{\frac{AV}{M} + \frac{A\sigma p}{\rho M}\sum_{j=1}^{h}\frac{c_{\pi(j)}-1}{P_j^\pi}(\sum_{i=1}^{j}(h-i+1)P_{i-1}^\pi)}{S(h+1)h/2 + \frac{V}{m}(C\frac{m-1}{p}+A)} \tag{3.17}$$

When the volume of load is big and the application is computationally intensive, we have:

$$\lim_{V\to\infty,\rho\to 0}\frac{C_{max}^{inf}}{C_{max}^{sat}} = \lim_{V\to\infty,\rho\to 0}\frac{\frac{VA}{1+\frac{p}{\rho}(1-\frac{1}{P_h^\pi})}}{\frac{VA}{m}(\rho\frac{m-1}{p}+1)} =$$

$$\lim_{\rho\to 0}\frac{m}{(1+\frac{p}{\rho}(1-\frac{1}{P_h^\pi}))(\rho\frac{m-1}{p}+1)} = \lim_{\rho\to 0}\frac{m}{1+\frac{p}{\rho}(1-\frac{1}{P_h^\pi})} \stackrel{H}{=} 1$$

since $\lim_{\rho\to 0} P_h^\pi = 1$, we applied de l'Hôpital rule and obtained:
$\lim_{\rho\to 0}\frac{p(1-\frac{1}{P_h^\pi})}{\rho} \stackrel{H}{=} \lim_{\rho\to 0}\frac{p(\sum_{i=0}^{h-1}(p+1)^i+2\rho((p+1)^3+\cdots+3\rho^2((p+1)^6+\ldots)}{(P_h^\pi)^2} = m-1.$ □

A similar conclusion can be drawn as in the star interconnection and as for a binomial tree under NLF strategy. In binomial trees spanned in homogeneous computer networks, under LLF strategy, when size V of the problem is big, and the problem is

computationally intensive ($\rho \to 0$), then the relative influence of the limited memory on the processing time is negligible.

In our modeling of LLF strategy we observed several interesting facts:

- It was shown in [45] that LLF strategy is optimal in a system with unlimited memory. In the saturated system it is not, because LLF has greater number of communication startups than NLF. This communication overhead is not compensated for by a better distribution of the load and shorter computation time.
- In the earlier publications on divisible load theory [27, 45] systems with unlimited memory were considered (i.e. case 'inf'). Linear Programming formulations had more restricted form and e.g. inequality (3.14) had form of equation. As a result in LLF strategy, when volume V is small and the available memory is not restricted, only few layers can be activated (even if we have many processor layers) to satisfy the classical version of LP LLF. Thus, small increase of V may be satisfactory to activate more layers and in this way reduce schedule length. This is demonstrated in the example presented below. Consequently, with V increasing C_{max}^{inf} may decrease. This is evident in Fig. 3.5 where lines for 'inf' case and $A =$ 1E-3, and $A = 1$ are not smooth for small V.
- The above irregular behavior was not observed in the LP NLF model.
- We observed that for $A \approx C$ only the last layer was populated. When $A \gg C$ the layers closer to the originator were more often populated.
- None of NLF, LLF strategies dominates the other in all cases. However, for big volumes and LLF shorter schedules were obtained.

Example. Consider a system with $h = 2, p = 4, A =$ 1E-3, $C =$ 1E-6, $S =$1E-3, $V = 20$. In the system with unlimited memory [45] equations describing distribution of the load have positive solution only for one layer (5 processors altogether). Schedule length in this case is $C_{max}^{sat} \approx 0.0048$. However, when $V = 24$ all 25 processors can

be activated, and $C_{max}^{sat} \approx 0.003$. Using LP LLF, and only the last layer we obtain $C_{max} \approx 0.0029$ in the first, and $C_{max} \approx 0.0031$, in the second case. □

3.2.3 Conclusions

In this section we analyzed divisible load distribution in systems with a single level of limited memory. Interconnection topologies of a star, and binomial tree under two different distribution strategies were studied. It appeared that in homogeneous systems and big computationally intensive applications mainly the processor and communication speeds limit performance of the systems. This conclusion is satisfied as long as the load fits into the available memory buffers and processing rates are constant for all assigned sizes of the load. For practical reasons these assumptions should be released. Such a relaxation is considered in chapter 4.

In our discussion we assumed that only the size of the receiver memory is restricting distribution of the load. The communication system is not limiting the size of the message. This may not be the case in practice. Therefore, a system with limited communication system capacity will be a subject of the further analysis in chapter 5.

In the next section we study the case of and arbitrary processor activation order.

3.3 Arbitrary activation sequence in star

In this section we address the problem of finding the optimal sequence of activating processors in a star network when memory buffers have limited sizes and communication delays include startup times. In the preceding discussion it was assumed that the sequence of activating processors is fixed. Here we relax this restriction and allow for selecting the best sequence of activating processors. This problem was raised in [53].

3.3.1 Linear programming approach

In this section we formulate the problem of selecting the optimum processor activation order as a mathematical programming problem.

Let us denote by a binary variable x_{ij}, for $i, j = 1, \ldots, m$, the order of activating the processors. $x_{ij} = 1$ denotes that P_j is activated on i-th position in the sequence. Otherwise $x_{ij} = 0$. The problem of optimal activation of the processors and distribution of the load can be formulated as a mixed nonlinear programming problem:

MNP:

minimize C_{max}

subject to:

$$\alpha_0 A_0 V \leq C_{max} \tag{3.18}$$

$$\sum_{k=1}^{i}\sum_{j=1}^{m} x_{kj}(\alpha_j V C_j + S_j) + \sum_{j=1}^{m} x_{ij}\alpha_j V A_j \geq C_{max} \qquad \text{for } i = 1, \ldots, m \tag{3.19}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad \text{for } j = 1, \ldots, m \tag{3.20}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1 \qquad \text{for } i = 1, \ldots, m \tag{3.21}$$

$$x_{ij} \in \{0, 1\} \qquad \text{for } i, j = 1, \ldots, m \tag{3.22}$$

$$\alpha_0 + \sum_{j=1}^{m}\sum_{i=1}^{m} x_{ij}\alpha_j = 1 \tag{3.23}$$

$$B_j \geq \alpha_j V \geq 0 \qquad \text{for } j = 0, \ldots, m \tag{3.24}$$

The above MNP formulation is a mixed problem because we have both binary variables x_{ij}, and continuous variables α_i, C_{max}. MNP is nonlinear because in constraints (3.19), (3.23) we have multiplication of the variables. Equations (3.18) and (3.19) demand that all processors finish computing before C_{max}. In inequalities (3.19) term $\sum_{k=1}^{i}\sum_{j=1}^{m} x_{kj}(\alpha_j V C_j + S_j)$ is the time of sending the load to the processor activated as i-th in the sequence, and $\sum_{j=1}^{m} x_{ij} A_j \alpha_j V$ is the computation time of the i-th

processor in the sequence. Constraints (3.20)-(3.22) guarantee that the sequence of activating the processors is correct: each PE is activated at most once by (3.20), each position in the activation sequence is occupied by at most one processor by (3.21). Due to weak form of the inequalities some processors may remain idle. Equation (3.23) guarantees processing of the whole load. Observe that some machines may be missing in the activation sequence, and $x_{ij} = 0$ for $i, j = 1, \ldots, m$ is a valid solution for constraints (3.20)-(3.22). Yet, it would not be a valid solution to our problem because appropriate communication time would not appear in equations (3.19). In order to prevent such a situation term $\sum_{i=1}^{m} x_{ij}\alpha_j$ in equation (3.23) guarantees that only the chunks sent to the processors (i.e. with $x_{ij} = 1$) are counted as really processed. Equations (3.24), guarantee that the load can be feasibly assigned to the processors. Let us apply the above formulation to solve Example 3 from [53].

Example. We have the same data as in the previous example: $m = 4, V = 100$, $A_0 = 1, A_1 = 5, A_2 = 4, A_3 = 3, A_4 = 2$, $B_0 = 10, B_1 = 20, B_2 = 45, B_3 = 15, B_4 = 30$, $C_1 = 4, C_2 = 3, C_3 = 2, C_4 = 1$, $S_i = 0$, for $i = 1, \ldots, 4$.
MS Excel ver.7.0 managed to obtain the following solution to MNP:

processor order	B_i	α_i	communication completion	computation completion
P_0	10	10	0	10
P_2	45	35.2941	105.8824	247.0588
P_4	30	30	135.8824	195.8824
P_1	20	12.3529	185.2941	247.0588
P_3	15	12.3529	210	247.0588

The sequence of activating the processors, according to the solver we used, is P_2, P_4, P_1, P_3. Schedule length is $C_{max} = 247.0588$, and it is better than the one found in [53]. The reasons for this were given earlier: IBS strategy proposed in [53] is a heuristic, not an optimization algorithm. For the same instance with $V = 50$ (also considered in [53]) the following solution was obtained for MNP:

processor order	B_i	α_i	communication completion	computation completion
P_0	10	10	0	10
P_4	30	24.277	24.277	72.832
P_3	15	9.711	43.699	72.832
P_2	45	4.162	56.185	72.832
P_1	20	1.850	63.584	72.832

Thus, the sequence found is P_4, P_3, P_2, P_1, and $C_{max} = 72.832$. □

The computational complexity of the general purpose mixed nonlinear solvers applied to MNP is high. These codes are capable of solving hard computational problems such as traveling salesman problem, quadratic assignment problem, and even more involved ones. It has been shown in Section 3.1 that the problem of scheduling divisible loads in a star network with limited processor memory buffers and communication startup times is **NP**-hard. According to the current state of knowledge [40] only algorithms with computational complexity growing exponentially with the size of the problem are known for this kind of problems. Thus, the codes finding optimal solutions of MNP have the worst-case execution time growing exponentially, e.g. with the number of binary variables x_{ij}. As exponential functions increase explosively with the value of the argument, exponential-time algorithms are in practice restricted to small instances of the solved problem. This leaves space for heuristic methods which find good solution fast, and this is the advantage of IBS strategy proposed in [53].

3.3.2 Branch and Bound algorithm

Let us note that for fixed sequence of processors activation MNP problem reduces to LP problem defined in section 3.2.1. Therefore it can be deduced that the difficulty of our problem consists in determining the set of processors to be activated and the sequence of the activation. An exact optimization algorithm can be based on an enumeration of all such sets and sequences.

The method of dividing the set of all processor activation sequences into subsets that are exhaustively searched and/or eliminated constitutes the *branching* scheme of a branch-and-bound (B&B) algorithm. A branch-and-bound algorithm implicitly builds a tree with all possible solutions. In our B&B algorithm no decision is taken in the root of the tree, and the set of processors to be activated is empty. The first level of the tree consists of solutions with only one processor: $(P_1), (P_2), \ldots, (P_m)$. Each partial solution (P_i) is a root of the subtree comprising solutions starting with the activation of P_i, for $i = 1, \ldots, m$. The subsets of the solutions represented by the nodes of the first level are divided (branched) at level two, to represent solutions consisting of two processors. Hence, (P_i, P_j) for $j \in \{1, \ldots, m\} - \{i\}$ are successors of (P_i) for $i = 1, \ldots, m$. At the third level (P_i, P_j, P_k) for $k \in \{1, \ldots, m\} - \{i, j\}$ are successors of (P_i, P_j). Thus, a level r solution $(P_i, \ldots, P_k)$ has successors $(P_i, \ldots, P_k, P_l)$ at level $r + 1$ obtained by a concatenation of $(P_i, \ldots, P_k)$ with processor P_l which has not been activated in sequence $(P_i, \ldots, P_k)$. The depth of the tree is at most m because no more than m processors can be activated. Note, that both the leaves of the tree and the internal nodes are potential solutions. The tree is searched in the depth-first order.

The second important component of B&B algorithm is *bounding* which allows for pruning search tree nodes representing subsets of solutions certainly not better than some known solution. For each node $a = (P_i, \ldots, P_k)$ of the search tree a lower bound $LB(a)$ on the schedule length of all the a's successors is calculated. This lower bound is compared with the length C of the best known solution. If $C \leq LB(a)$, then there is no hope that any successor of node (solution) a improves the best known solution. Therefore, successors of a are not considered any more. Value C is updated each time a better solution is found. Initially, the lower bound for node a was calculated as the optimum C_{max} in the linear program (3.1)-(3.4) assuming a processor activation sequence $(P_i, \ldots, P_k, P_{id}^{|Z|}(Z))$, where $Z = \mathcal{P} - \{P_i, \ldots, P_k\}$ is the set of the processors not included in a, and symbol $P_{id}^{|Z|}(Z)$ stands for a sequence of $|Z|$ copies of an ideal processor $P_{id}(Z)$. $P_{id}(Z)$ has all the best parameters of the processors in the set

Z. Hence, $P_{id}(Z)$ has processing ratio $A_{id} = \min_{P_l \in Z}\{A_l\}$, communication link with communication rate $C_{id} = \min_{P_l \in Z}\{C_l\}$, and startup time $S_{id} = \min_{P_l \in Z}\{S_l\}$, memory buffer size is $B_{id} = \max_{P_l \in Z}\{B_l\}$.

Unfortunately, the lower bound calculated in this way has two disadvantages. Firstly, when communication delay is big it may happen that the whole volume V of the load can be processed on only a few processors in shorter time than the time needed to activate all m processors, and our lower bound is not correct. Consider an example: $V = 10$, $m = 20$, $A_i = 1, C_i = 0, S_i = 1, B_i = 10$, for $i = 1, \ldots, m$. The load may be processed by using only four processors in $C^*_{max} = 5$ using distribution $\alpha_1 = 4, \alpha_2 = 3, \alpha_3 = 2, \alpha_4 = 1$. The time needed to activate all $m = 20$ processors is $20 > C^*_{max}$. Note that if all m processors were used then some of them would receive no load (in the example $\alpha_l = 0$, for $l = 5, \ldots, m$). Thus, the solutions of the linear program (3.1)-(3.2) for sequence $(P_i, \ldots, P_k, P_{id}^{|Z|}(Z))$ with $\alpha_l = 0$, for some l, indicate that the sequence is too long, and the lower bound is inaccurate. In other words, some processor P_l introduces a communication delay contributing to the schedule length, but does not compute. Therefore, P_l can be eliminated from the activation sequence without increasing the schedule length. We cannot remove, however, the real processors from sequence a. The second disadvantage of the above method appears when the ideal processor is superior to the real processors in the sequence a. Then, the real processors may, again, receive no load. Both situations, can be dealt with by decreasing the number of ideal processors until all processors in the sequence receive some load. For the above reasons the procedure of calculating the lower bound has been extended by iterative decreasing the number of ideal processors until all processors receive some load.

3.3.3 Heuristic algorithms

The problem of optimal scheduling divisible load in a heterogeneous star with communication startup times and limited memory sizes is computationally hard. The exact optimization algorithm presented in the previous section has exponential execution

time int the worst case. Therefore, it is reasonable to consider heuristic algorithms as alternative methods of finding solutions to our problem. Heuristics, are low-order polynomial time algorithms providing feasible solutions. However, the solutions derived by them are not guaranteed to be optimal.

The heuristic methods we studied try to find the best sequence of processor activation, and the set of working processors. The first set of heuristics activates all available processors according to a single processor parameter. Then the distribution of the load (i.e. α_i's) is found using formulation (3.1)-(3.4). Thus, we studied heuristic A which ordered processors according to the nondecreasing value of processing rates (i.e. A_i's). Analogously, heuristics C, and S were analyzed. Heuristic, B ordered processors according to nonincreasing value of buffer sizes B. The second set of heuristics intends to combine two parameters of processors. Heuristic C/A orders processors from the processor with the least value of $\frac{C_i}{A_i}$ to the processor with the biggest one. Heuristic S/A, is formulated similarly. Heuristic SC orders processors according to the increasing value of S_iC_i. Analogously, method B/A orders processors according to the decreasing values of $\frac{B_i}{A_i}$. The above group of heuristics will be called *primary heuristics*.

Not always are we allowed to take all the processors available in a computer system. More often only a subset is admissible. Activating too many processors may introduce costly communication delays. Consequently, a second group of heuristics has been devised. Processors are ordered as in one of the previous methods, but only a *minimum* admissible number of the processors sufficient to hold the whole load V in the memory are selected from the beginning of the list. The distribution of the load is calculated using a linear program analogous to formulation (3.1)-(3.4). Heuristics of this type are conservative in using processors because only few of them are selected from potentially large processor set. This group of heuristics will be called *m-heuristics*. Therefore, heuristics called *mA, mB, mC, mS, mC/A, mS/A, mB/A, mCS* were studied.

Two additional heuristics have been used as a reference. Algorithm *Rnd* activates

all processors in random order. Method *mRnd* orders processors randomly, and uses a minimum number of the processors heading the list which suffice to hold the whole load.

3.3.4 Computational experiments

In this section we examine performance of the heuristics, and B&B algorithm. All computational experiments were performed on a PC, with 950MHz Pentium III, and Windows 2000. The algorithms were implemented in Borland C++ version 5.5.1. The data sets for the tests were generated using uniform distribution of the parameters from interval $[0, x]$, where x was given. Two sets of experiments were conducted. The first one was intended to resemble parameters of a real computer system, where computations are slow and time-consuming, while communication is relatively fast. Thus, processor parameters were drawn from the following intervals: $A_i \in [0, 1\text{E-}2]$, $B_i \in [0, 1\text{E}6]$, $C_i \in [0, 1\text{E-}6]$, $S_i \in [0, 1\text{E-}2]$. The instances generated in this way will be called *dataset 1.* In the second set of experiments parameters A_i, C_i, S_i were drawn from interval [0,1], and B from interval [0,1E6]. This dataset will be called *dataset 2.* The parameters of the processors were generated independently of the other processors' parameters. Thus, the computing and communication environments are heterogeneous. Unless otherwise specified $m = 8, V = 1E6$. The influence of processor parameters (A_i, B_i, C_i, S_i) on the performance of the algorithm has also been examined. In these experiments all the processors had the same fixed value of the examined parameter. For example, when the influence of processing rate was the subject of the study, then the value of processing rate for all processors has been set on the same value A. The other parameters B_i, C_i, S_i, m, V were generated as previously specified. Each point in the following charts is an average of at least ten instances. The execution time of the algorithms is analyzed first, the quality of the heuristic solutions is considered later.

Time performance of the algorithms

In Fig. 3.7 execution time of the B&B, and heuristic algorithms as a function of processor number m is presented. In order not to confuse the reader by excessive number of details only four lines are shown in Fig. 3.7. The first two lines from the top represent average execution time of B&B algorithm for datasets 1, and 2. The third line from the top shows an average execution time of all primary heuristics. The primary heuristics obey the same rule: sorting the processors according to some parameter(s), calculating distribution of the load for all m processors using linear programming. Hence, there is no big variation of the execution time among the primary heuristics, and it is sufficient to represent them by an average execution time. The fourth line is an average execution time of m-heuristics. Also m-heuristics have the same fixed structure, and can be represented by a single line without losing much of information. As it can be clearly seen B&B algorithm has an exponential running time. The execution time of B&B algorithm depends on the data set. The instances with fast communication tend to be computationally harder. The execution times of the primary heuristics are polynomial functions of m. For m-heuristics the execution time is almost constant because the cost of linear programming is dominating their running time. The sizes of the linear programmes of m-heuristics are very similar because volume V of the load is constant, and memory buffers sizes are drawn in the way not depending on m.

In Fig. 3.8 dependencies of the B&B execution time on processing rate (denoted A), buffer size (B), transfer rate (C), and load size (V) for the dataset 1 are depicted. As it was explained, dependence on each of the parameters A, B, C was tested after fixing the same value of the given parameter on all processors. The dependence of the B&B execution time on communication startup time S in range [1E-6,1E0] has also been tested, but no relation between S and the execution time has been observed.

It can be seen in Fig. 3.8(a) that the instances with very small A, or with very big A are easy to solve. In the first case processors are very fast and there is no incentive to use many processors except for the need for sufficient memory size to hold the load.

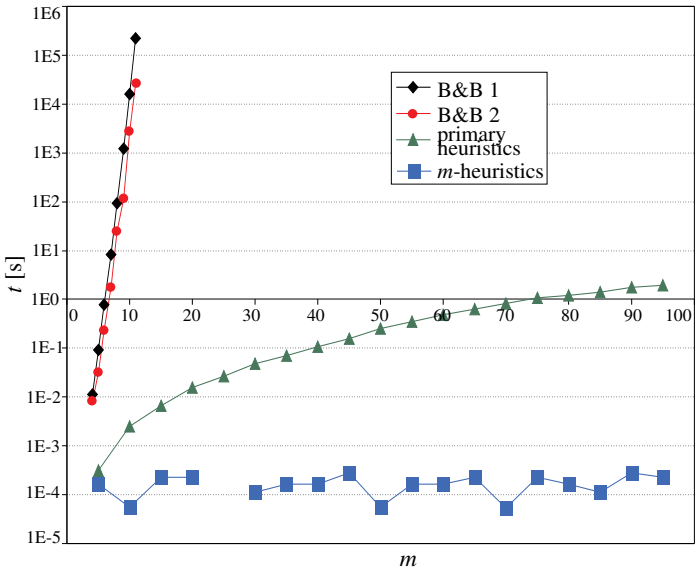


Figure 3.7: Execution times of the B&B algorithm and heuristics vs. m.

The communication costs can be minimized by using only few processors. Hence, the optimum solution is usually found in the upper levels of the search tree, and the leaves of the search tree are hardly ever reached. On the other hand, when all processors are very slow (A is big), the computation time dominates the schedule length. The influence of the communication delays is minor. Due to the rounding error the procedure calculating the lower bound becomes "myopic" and does not recognize the differences in communication cost resulting from various processor activation sequences. The procedure returns values equal to the length of the schedule when all processors are working without communication delays. A solution with all m processors is found in the first leaf of the search tree. After this all existing branches are pruned because the lower bound is the same as the schedule length for the solution found in the first leaf.

In Fig. 3.8(b) dependence of B&B execution time on the size of memory buffer is presented. When $B_i = V/m$ all processors must be used, and their buffers must be fully utilized. Therefore, all leaves of the search tree represent the only feasible solutions. Nevertheless, B&B algorithm verifies all internal nodes of the search tree

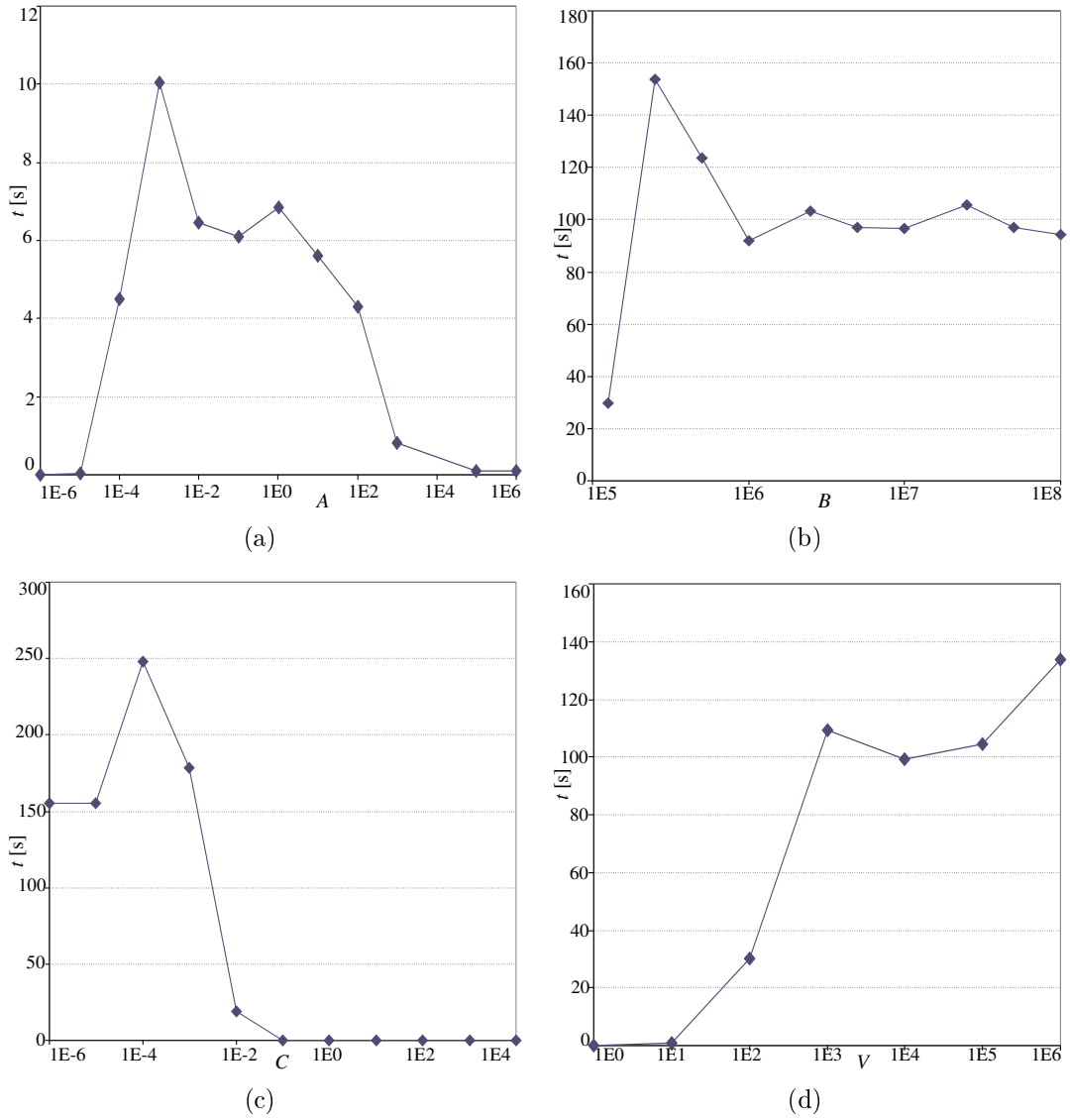


Figure 3.8: Execution times of the B&B algorithm vs a) A, b) B, c) C, d) V.

just to find that they are infeasible. Identifying an infeasible linear programme is faster on average than solving the same size feasible linear programme. Therefore, the execution time of the algorithm initially grows with growing memory size. When memory sizes of the processors become sufficiently big, they no longer determine selection of the best solution, and the dependence of the B&B execution time on B levels off.

In Fig. 3.8(c) dependence on the transfer rate is shown. The instances with small C are the hardest ones. When C is small, all the communication links have high bandwidth, and it is possible to activate all processors at relatively low cost. The dependence of the processing time on the processor sequence is not very significant because all processors work, and computation time dominates. Hence, the lower bounds are close to the value of the optimum solution, and it is not possible to prune the search tree at the very initial stages. When C is very big communication delays are very big compared to the computation time. As it was in the case of big A, also for big C the lower bound becomes "myopic", and does not recognize differences in the schedule length resulting from different activation sequences. The schedule length of the first feasible solution is equal to the lower bound calculated in all other nodes of the tree, and the tree is pruned.

In Fig. 3.8(d) dependence of the B&B execution time on the size V of the load is shown. For small V computation time is short because small problem sizes do not justify communication costs induced by the startup times. Hence, the optimum solution is found in one of the initial stages of the search tree. With growing V the number of activated processors is growing, and the size of the searched tree is growing too.

From the above considerations we infer not only on the computational tractability of particular instances, or patterns of the optimum solutions, but also on the correctness of the results derived by the B&B algorithm. These depend on the value of the parameters because the representation of the floating point numbers has a limited accuracy which may result in a premature search termination.

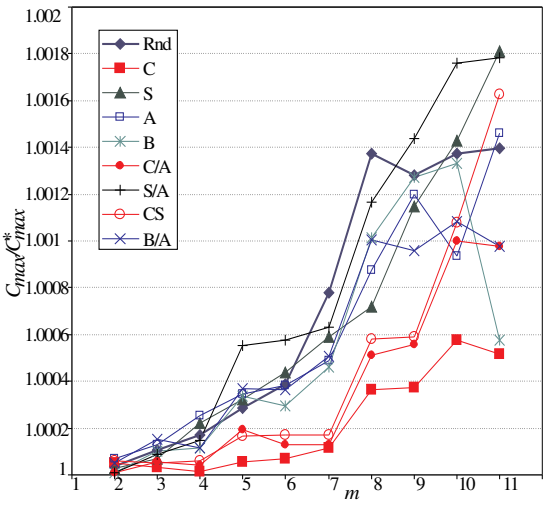


Figure 3.9: Average quality of the primary heuristics vs the number of processors m, dataset 1, $m \leq 11$.

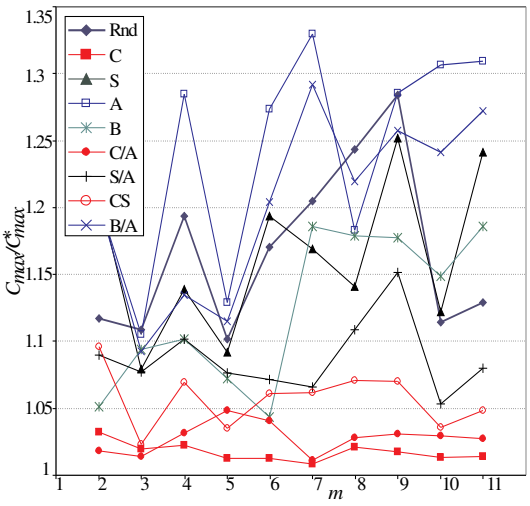


Figure 3.10: Average quality of the primary heuristics vs the number of processors m, dataset 2, $m \leq 11$.

Quality of the solutions

In this chapter we examine quality of the solutions obtained by the heuristic methods. In all the figures the average relative distance from the optimum solution C^*_{max} or a lower bounds on the schedule length lb is shown on the vertical axis. The closer a line is to value 1 the better performance is. First we study the influence of the processor number m, then of the other instance parameters.

In Fig.3.9 the dependence of the average quality of the primary heuristics solutions on the number of processors m is depicted, for the dataset 1. Though C is small in dataset 1, heuristics C, and C/A give good solutions. On the other hand heuristic S/A in many cases performs even worse than the solution selected randomly by heuristic *Rnd*. In Fig.3.10 the same dependence is shown, but for dataset 2. As it can be seen the variance of the quality is much bigger than in dataset 1. Still, heuristics $C, C/A, CS$ based on transfer rate C give the best solutions. Heuristics $A, B/A$ give the worst solutions.

Fig.3.11 shows average quality of the primary heuristic solutions for $m \in [5, 95]$. In these experiments the optimum schedule length was unknown for computational

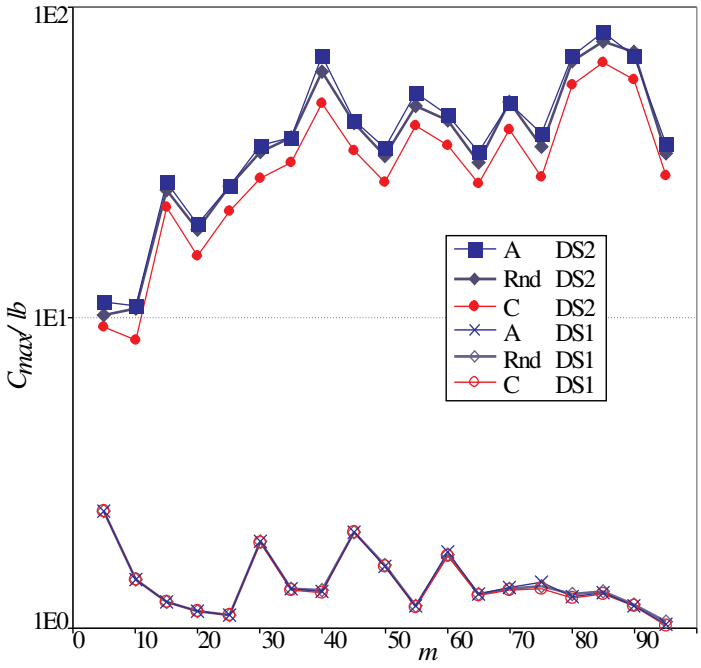


Figure 3.11: Average quality of the primary heuristics vs the number of processors m, for big m.

complexity reasons. The quality of the solution was expressed as an average distance from a lower bound on the schedule length. The lower bound was calculated as $lb = \min_{i=1}^{m}\{S_i\} + (\frac{\min_{i=1}^{m}\{C_i\}}{\min_{i=1}^{m}\{A_i\}} + 1)\frac{V}{\sum_{i=1}^{m}\frac{1}{A_i}}$. In the above lower bound $\sum_{i=1}^{m}\frac{1}{A_i}$ is total speed of the processors, $\frac{V}{\sum_{i=1}^{m}\frac{1}{A_i}}$ is the computation time under assumptions that all processors work in parallel, $\frac{1}{\min_{i=1}^{m}\{A_i\}}\frac{V}{\sum_{i=1}^{m}\frac{1}{A_i}}$ is the smallest possible assignment of the load to a processor assuming that all processors work in parallel. Thus, $\min_{i=1}^{m}\{S_i\} + \frac{\min_{i=1}^{m}\{C_i\}}{\min_{i=1}^{m}\{A_i\}}\frac{V}{\sum_{i=1}^{m}\frac{1}{A_i}}$ is a lower bound on the communication delay. Only the extreme results of all heuristics achieved by heuristics A, C, and the reference heuristic Rnd are shown in Fig.3.11. As it can be seen there is no significant difference in the performance of all the primary heuristics. Heuristic C weakly dominates in dataset 2. The dataset 2 is harder for primary heuristic and big m than dataset 1.

In Fig.3.12, and Fig.3.13 dependence of the average quality of the m-heuristics solutions for datasets 1, and dataset 2, respectively, are depicted. For dataset 1

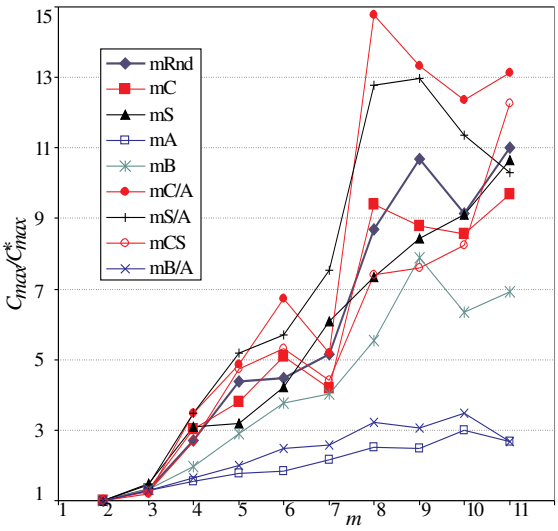


Figure 3.12: Average quality of the m-heuristics vs the number of processors m, dataset 1, $m \leq 11$.

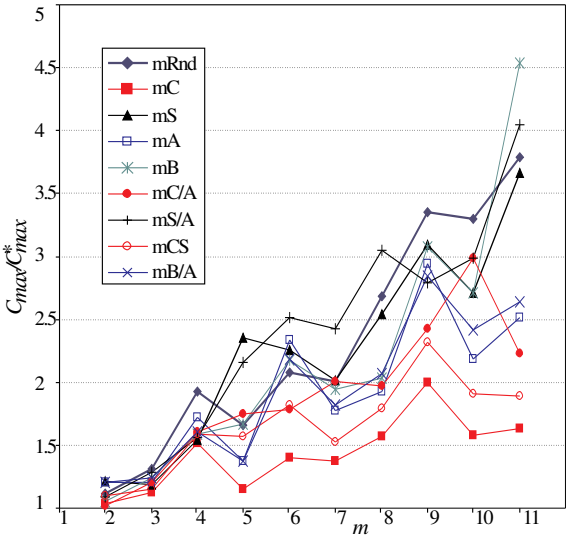


Figure 3.13: Average quality of the m-heuristics vs the number of processors m, dataset 2, $m \leq 11$.

heuristics $A, mB/A$ are better (cf.Fig.3.12), and for dataset 2 heuristic C is dominating (cf.Fig.3.13). Intuitively this is a reasonable situation because in dataset 1 communication is fast (C is small) and secondary parameters such as A_i, B_i come into play.

For big numbers of processors performance of m-heuristics is shown in Fig.3.14 for dataset 1, and in Fig.3.15 for dataset 2. Again an average distance from the lower bound lb is the value shown on the vertical axis. For dataset 1 (Fig.3.14) heuristics based on processing rate and memory size, $mA, mB/A$, are the best. On contrary for dataset 2 (Fig.3.15) heuristics based on communication rate $mC, mCS, mC/A$ are the best. The reason for this behavior is that dataset 1 has small communication rates, and startup times. Hence, other parameters come into play in dataset 1: computing rates A_i, and memory sizes B_i. The heuristics based on these parameters are able to give good solutions. For more general instances, as in dataset 2, the communication transfer rate is the dominating parameter.

Now we will examine the quality of the solutions obtained by the heuristics, as a

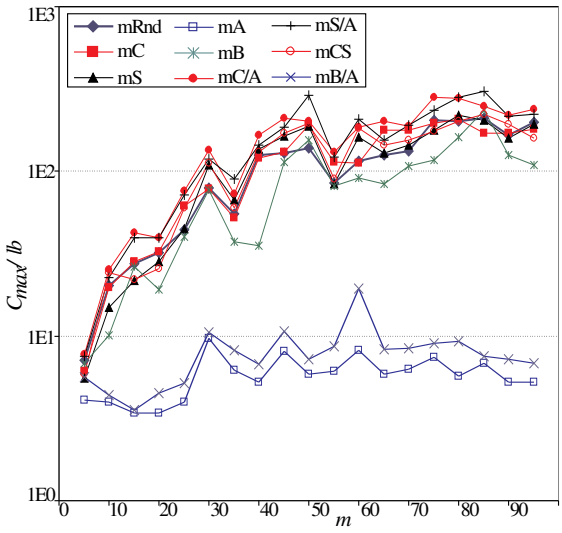


Figure 3.14: Average quality of the m-heuristics vs the number of processors m, dataset 1 and big m.

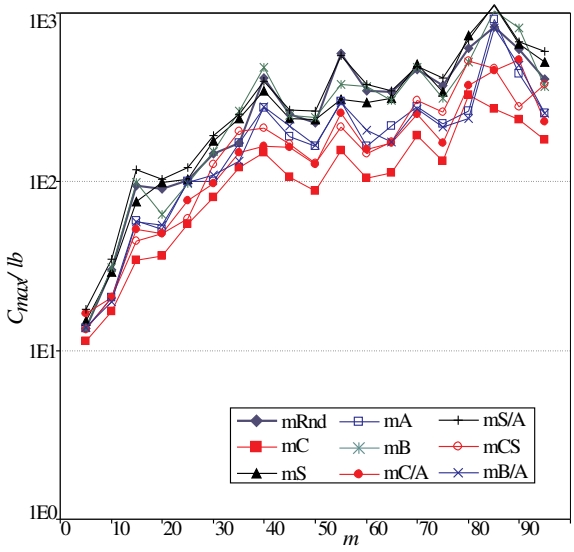
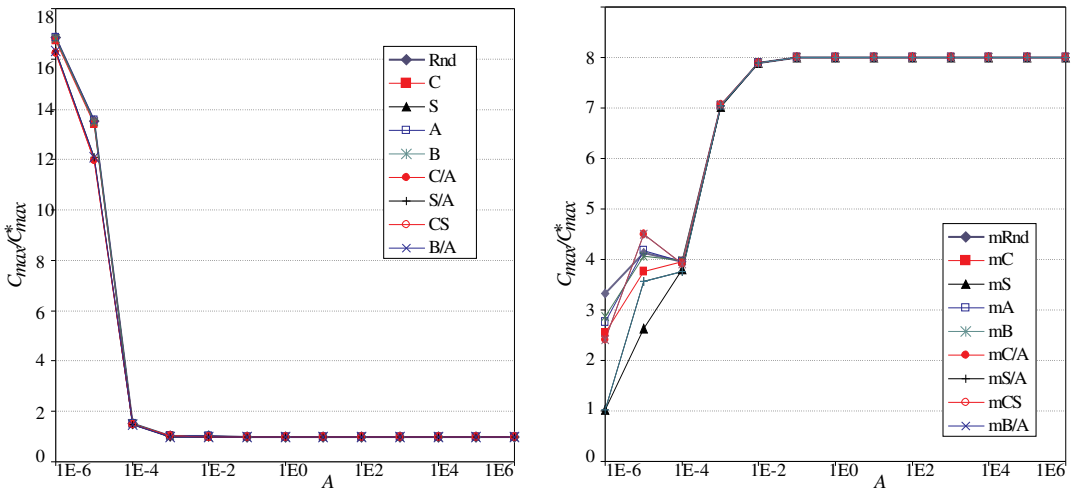


Figure 3.15: Average quality of the m-heuristics vs the number of processors m, dataset 2 and big m.

function of the processing rate, the communication rate, the startup time, the memory buffer size, and the volume of the load. All results are obtained for dataset 1. As it can be seen in Fig.3.16 there is no big difference in the performance of the heuristics when A is changing. Only for small A can any difference be observed. For big A computation time dominates in the schedule length. Changes in the communication cost resulting from reordering the processors are minor compared to the computation time. Hence, the sequence of processor activation is almost immaterial. Albeit only one processor is able to compute all the load, all m processors are used in the optimum solutions when A is big. Since m-heuristics tend to use as few processors as possible the average quality of their solutions is m times worse than the optimum (cf.Fig.3.16(b)).

Dependence of the average quality of the solutions on transfer rate can be seen in Fig.3.17. If C is big then communication delay dominates in the schedule length, and the sequence of processor activation has minor influence on the schedule length. For this reason the bigger C is the better quality of the heuristic solutions is. No



(a) (b)

Figure 3.16: Average quality of the solutions vs A for a) primary heuristics, b) for m-heuristics.

primary heuristic clearly outperforms the other primary heuristic (Fig.3.17(a)). For m-heuristics (Fig.3.17(b)), methods $mA, mB/A$ are the best. This is a result of two facts: in dataset 1 communication delays are not big, and only minimum set of processors should be used. Thus, fast processors and processors with big memory buffers are preferred.

The dependence on buffer size B, startup time S, and load size V are similar. Therefore we only outline the results for these dependencies. For $S \in$[1E-6,1E0], and $B \in$[1.25E5,1E8], the average distance of the primary heuristic solutions from the optimum is less than 4%. For $V \in$[1E0,1E6] the distance from the optimum decreases with V increasing in a way resembling dependence on A (cf. Fig.3.16(a)). The performance difference between the primary heuristics is minor. Methods mA, mD/A dominated among the m-heuristics when S, B, V were varying in the aforementioned intervals. The reason for this behavior is that dataset 1 has small communication costs. No dependence of the quality on S, V has been observed for m-heuristics. The quality of solutions derived by m-heuristic deteriorates when the buffer size increases because less processors are used.

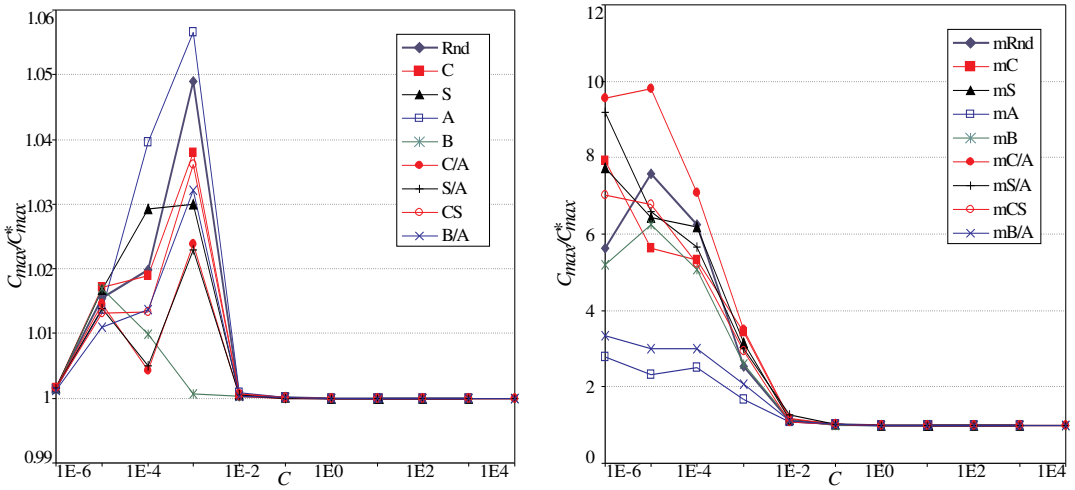


Figure 3.17: Average quality of the solutions vs C for a) primary heuristics, b) for m-heuristics.

We conclude from the above results that m-heuristics give worse solutions than the primary heuristics. On the other hand m-heuristics are much faster. Heuristics C, mCS based on transfer rate are the best when communication delays are not negligible. When communication delays are small (as in dataset 1), and only a small set of processors can be used (as in m-heuristics), then selecting fastest processors with the biggest memory sizes is advantageous, which is not surprising. Thus, the performance of the heuristics depend very much on the communication and computation environment.

3.3.5 Conclusions

In this chapter we analyzed a problem of optimal distributing of the divisible computations in a star-network of heterogeneous processors with single level of limited memory and non-zero communication startup times. The problem has been shown to be computationally hard. Therefore, exponential optimization algorithm and polynomial-time heuristics have been presented. The execution times of the proposed methods have been measured and analyzed. As a result it has been observed that the usability

of the optimization algorithm is limited not only by its running time, but also by the limited accuracy of the floating point numbers representation. The quality of the solutions generated by heuristics has been also examined. It appears that heuristics based on communication transfer rate are superior. However, in certain situations, also other parameters, such as computing rate and memory buffers may come into play.

Chapter 4

Systems with Hierarchical Memory

In the earlier DLT literature processing time dependence on the size of the load was linear. This is justified in flat (non-hierarchical) memory systems. Though core memory sizes grew rapidly over the years, the memory size limitations are an important factor in high-performance computing. In the earlier papers [53, 38] considering limited memory in DLT it was assumed that memory limits are restrictive, i.e. assigning load beyond memory limit is forbidden, and results in an infeasible solution. Yet, in most of contemporary computer systems memory is hierarchical. The higher certain level of memory hierarchy is, the faster transmission can be achieved. But also the higher certain level of memory hierarchy is, the smaller the memory size is. The lowest memory levels are implemented either as virtual memory storing memory pages on disks or as files directly accessed by the application. Huge sizes of disk storage can be achieved at relatively low costs using off-the-shelf components. Thus, instead of strictly forbidding a load assignment exceeding certain memory level size, it is more practicable to use the next memory level with longer access time, and hence, smaller computing rate. We will call the applications using external memory (i.e. disks) the *out-of-core* computations. In Fig. 4.1 we demonstrate that using out-of-core memory makes a big difference in the computation speed. A dependence of the processing time of a simple search for a pattern in a linear array vs. array size for various computing platforms is shown in Fig. 4.1. Even for this simple application, with a predictable

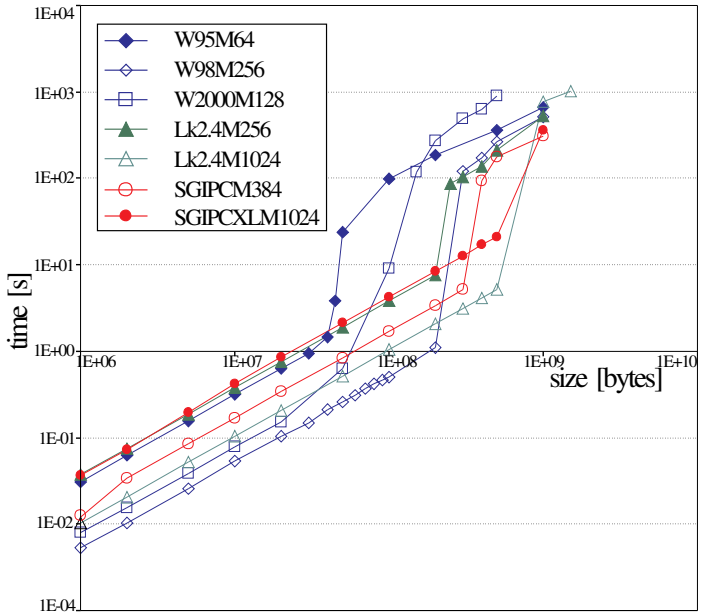


Figure 4.1: Processing time of a simple search for a pattern in a linear array vs. array size. In the legend Wx denotes Windows version x, Lk2.4 denotes Linux with kernel version 2.4, SGIPC - SGI Power Challenge, SGIPCXL - SGI Power Challenge XL, My denotes core memory size y MB.

memory access pattern using more memory than available in the core results in an increase of the execution time by at least an order of magnitude.

There is a broad class of the out-of-core parallel applications. These include data-intensive algorithms [66] for processing information from large scientific experiments, data mining, visualization [8, 34], simulation, often with the need for solving large linear algebra problems [67]. Gaussian [7] is an example of a commercial package using out-of-core memory. In [29] an environment for out-of-core parallel applications has been proposed. Computational fluid dynamics and large linear algebra problems have been used as benchmarks. The access to the major data arrays was achieved by using indirect addressing that has been known at the runtime only. It turned out that by using locality in the algorithm and dividing the arrays into small sections that fit in the core memory a fourfold reduction of the execution time has been obtained compared to the use of virtual memory.

Thus, an alternative to the out-of-core processing is to divide the load into many small chunks that fit into the available core memory. The chunks are sent to processors in an iterative manner. In this way it is possible to perform fast computations at the cost of additional communications. We will call this way of computing a *multi-installment* divisible load processing. Multi-installment processing has been considered, e.g., in [18, 69] and is also subject of Chapter 6. In this chapter we compare efficiency of the out-of-core with the efficiency of multi-installment computations.

4.1 Mathematical Models

In this section we formulate mathematical models for divisible load computations in a system with hierarchical memory, and for multi-installment computations.

Let us start with the description of the system architecture. We assume a star interconnection network. The load is sent to the processors in a single communication. P_1 receives the load first, P_2 as the second processor, etc. P_m receives its load as the last one. The originator does not compute, but communicates only. As already mentioned in Section 2.2 this assumption does not limit the generality of our considerations, because computations on the originator can be represented as an additional processor.

The computations are performed by processors connected to the hierarchical memory systems. The highest level is constituted by processor registers. The lowest level is disk storage. The memory sizes increase, and transfer rates decrease with the decreasing hierarchy level. Hence, the processing time depends on the amount of allocated memory. Processing time t_i on processor P_i is a piece-wise linear function of the assigned load x: $t_i = \max\{A_{ij1} + xA_{ij2}\}$ (cf. Fig. 4.2), where A_{ij1}, A_{ij2} are the coefficients of the linear function describing processing time on processor P_i, at jth hierarchy level. Note that A_{i11} is the cost of starting computations on processor P_i. For practical reasons only two levels of memory hierarchy: core memory, and virtual memory (or other form of disk storage), are considered in this chapter. The

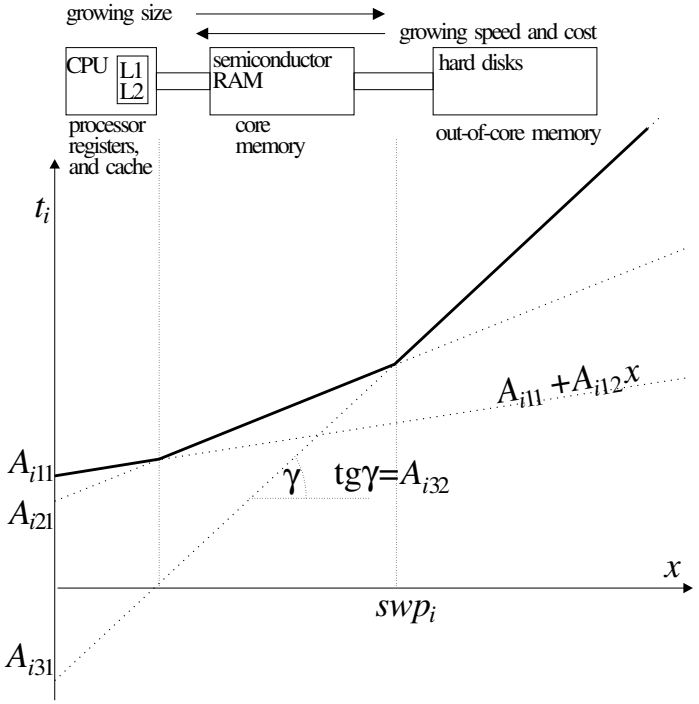


Figure 4.2: Memory hierarchy diagram, and a piece-wise linear dependence of processing time on the size of the load.

reason for this simplification is that divisible load computations are well suited for data parallel applications processing large volumes of data. Therefore, high levels of memory hierarchy, such as processor registers and caches, are not able to hold a substantial part of the assigned load. Due to the uniform and regular structure of divisible load applications memory access patterns are very predictable and cache management algorithms make this memory level transparent. The processor cache level of memory hierarchy could be visible for the application if the memory access pattern were random. However, to our best knowledge, no divisible load processing problem has been presented with random memory access patterns. The simplification of the model to only two memory levels can be easily relaxed as we explain in the further part of this section. We denote processing time on processor P_i as $t_i = \max\{A^l_{i1} + xA^l_{i2}, A^h_{i1} + xA^h_{i2}\}$, where A^l_{i1}, A^l_{i2} are the coefficients of the linear

function describing computing time in the core memory, and A^h_{i1}, A^h_{i2} are the analogous coefficient for computing out-of-core using disk storage. The size of the load swp_i beyond which operating system starts using the disk, and for which the above two functions are equal, i.e. $A^l_{i1} + swp_i A^l_{i2} = A^h_{i1} + swp_i A^h_{i2}$ will be called a *swap point* of processor P_i.

Let us observe that the above piece-wise linear dependence of the processing time on the load size may have also a different nature. Not only can the memory hierarchy be modeled in this way but also referencing memory on remote hosts or nonlinear dependence of the processing time on the problem size can be dealt in this way. Hence, after approximating a nonlinear convex function of the processing time by a piece-wise linear convex function of the load size our method can be used to represent more complex DLT applications.

We will formulate the problem of constructing optimum distribution of the divisible load computations as a linear program. Linear programming, is a special case of mathematical programming. It is used for modeling problems in science and engineering [56]. Let us denote by α_i the amount of load assigned to processor P_i, and by C_{max} the completion time of processing. Our problem can be formulated as a linear program:

LP SHM

minimize C_{max}

subject to:

$$\sum_{j=1}^{i}(S_j + \alpha_j C_j V) + t_i \leq C_{max} \quad i = 1, \ldots, m \tag{4.1}$$

$$A^l_{i1} + \alpha_i A^l_{i2} V \leq t_i \quad i = 1, \ldots, m \tag{4.2}$$

$$A^h_{i1} + \alpha_i A^h_{i2} V \leq t_i \quad i = 1, \ldots, m \tag{4.3}$$

$$\sum_{i=1}^{m} \alpha_i = 1 \tag{4.4}$$

$$\alpha_i \geq 0 \quad i = 1, \ldots, m$$

The above formulation has $2m+1$ variables, and $4m+1$ constraints. On the left-hand

side of inequalities (4.1) communication time $\sum_{j=1}^{i}(S_j + \alpha_j C_j V)$ until activating P_i is added to processing time t_i on P_i. Hence, inequalities (4.1) guarantee that all processors stop computing before the end of the schedule. Inequalities (4.2), (4.3) together model a piece-wise linear processing time function of the assigned load. Observe that (4.2), (4.3) restrict processing time t_i from below, but do not bind it from above. Sufficiency of these two constraints is guaranteed by the features of linear programming [56]. As the linear program constraints formulate a $2m+1$-dimensional convex polyhedron, and the objective function is a linear function of the program variables (C_{max}), the optimum solution is a point in a $2m + 1$-dimensional space located in an extreme corner of the polyhedron. The constraints intersecting in the optimum corner of the polyhedron are limiting the optimum value of the objective function, and are called active. If one of the constraints (4.2), (4.3) is active for some i then it is satisfied with equality, and t_i is exactly equal to the piece-wise function expressing the processing time. If none of the constraints (4.2), (4.3) is active for some i, and both are satisfied with inequality, then it means that processor P_i is idle for some time after completing computation phase. By inequality (4.4) all the load is processed.

Note that formulation (4.1)-(4.4) can be augmented by adding a constraint of the form $\alpha_i V \leq B_i$, to limit the total memory usage on some processor P_i. Constraints analogous to (4.2), (4.3) can be added to represent additional memory hierarchy levels. For the feasibility of this method it is necessary, that the dependence of processing time on the volume of load be a piece-wise linear convex function. The shape of the convex polyhedron and the location of the optimum extreme corner depend on the numerical values of the coefficients in constraints (4.1)-(4.4). Therefore, no closed-form expression of α_i seems possible. Consequently, general analytical solutions are hard to be expected.

Let us use an example to compare the above model with the earlier DLT approach. Consider a homogeneous system with two processors, and computing time function described by the parameters: $A_{11}^l = A_{21}^l = 1, A_{12}^l = A_{22}^l = 1, A_{11}^h =$

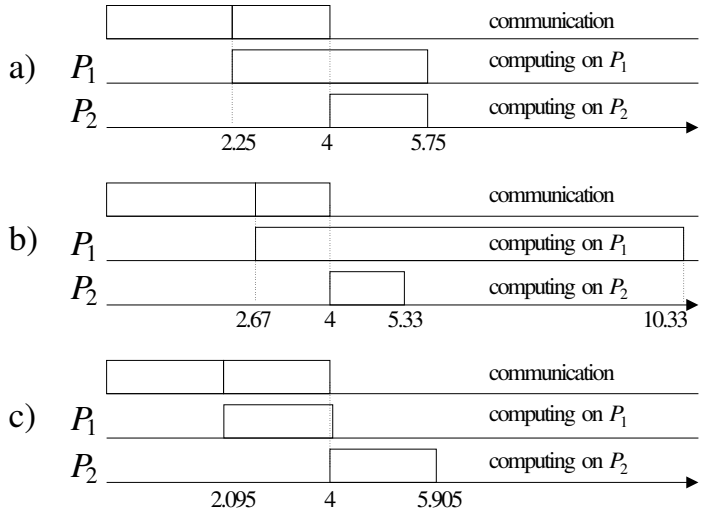


Figure 4.3: Schedules for load distributions calculated assuming a) hierarchical model of the memory, b) computing in the core memory only, c) computing out-of-core only.

$A^h_{21} = -9, A^h_{12} = A^h_{22} = 10$. Hence, the swap points are at load size $\frac{10}{9}$, and core memory is approximately ten times faster than the external storage because $\frac{A^h_{12}}{A^l_{12}} = \frac{A^h_{22}}{A^l_{22}} = 10$. The communication transfer rate is $C = 1$, and startup time is $S = 1$. The load volume is $V = 2$. By solving formulation (4.1)-(4.4) we obtain a solution $\alpha_1 V = 1.25, \alpha_2 V = 0.75, C_{max} = 5.75$ and the schedule shown in Fig. 4.3a. If a standard DLT methodology were used we would have to assume that processing time is a strictly linear function of the load. Thus, computing x units of the load would take either xA^l_{12} (if we assume optimistically that only core memory is used) or xA^h_{12} (if computing takes place out-of-core only). In the first case standard DLT theory [16, 19, 35] gives solution $\alpha_1 V = \frac{5}{3}, \alpha_2 V = \frac{1}{3}, C_{max} \approx 4.333$. But the real schedule length for this load distribution would be approximately 10.333, due to the hierarchical structure of the memory (see Fig. 4.3b). In the second case the standard DLT solution is $\alpha_1 V \approx 1.095, \alpha_2 V \approx 0.905, C_{max} \approx 13.048$. Yet, the real schedule length for this load distribution is approximately 5.905 (see Fig. 4.3c). As it can be seen in Fig. 4.3 neglecting memory hierarchy results in the significant load imbalance. The decisions made on the basis of the average processing rates can be even worse in heterogeneous systems. Let us consider a two processor system with processor

P_1 core memory size V, and processor P_2 core size 0. The computing speed on the second memory level is equal for both processors. A decision made on the basis of the speed at the second memory level splits the load equally between the processors. The optimum, however, is to give majority of the load to P_1. Depending on the speed of P_1 for the in-core computations the ratio of the optimum schedule length and the length of the schedule based on average speed can be very big.

Now we will formulate a simple algorithm for multi-installment divisible load processing, and a method for adjusting its parameters. Multi-installment processing is considered in more detail in Chapter 5 and Chapter 6. By the use of installments we want to exploit fast computing within the limits of the available core memory, while keeping communication costs low. Let us assume that the multi-installment algorithm divides the whole volume V into equal chunks of size δ. The processors are assigned load repetitively in rounds, i.e. in the manner $P_1, P_2, \ldots, P_m, P_1, P_2, \ldots$. The selection of the optimum chunk size δ is a non-trivial problem. Therefore, we give bounds on reasonable δ sizes and propose a heuristic method indicating a potentially good value.

Chunk size δ cannot exceed the swap point of any of the processors, i.e. $\delta \leq swp_i$ for $i = 1, \ldots, m$. Secondly, it cannot be too small because too many messages will be used, and communication costs will dominate the processing time. Let us calculate the minimum chunk size for which multi-installment processing is still better than the computations out-of-core. When the second memory level is used, the load must be at least as big as $m \times swp$. Thus, we may assume that processing time is dominated by computation time. A rough estimate of out-of-core processing rate for big volumes is $\lim_{V\to\infty} \frac{C_{max}}{V} = \frac{1}{\sum_{i=1}^{m} \frac{1}{A_{i2}^h}}$, where $\sum_{i=1}^{m} \frac{1}{A_{i2}^h}$ is the total speed of all the processors. An estimate of processing time for multi-installment processing with small load chunks and dominating communication time is $\frac{V}{m\delta}(\sum_{i=1}^{m} S_i + \delta \sum_{i=1}^{m} C_i) + A_{1i}^l + \delta A_{2i}^l$, where $\frac{V}{m\delta}$ is the number of communication rounds, $\sum_{i=1}^{m} S_i + \delta \sum_{i=1}^{m} C_i$ is the communication time per round, and $A_{1i}^l + \delta A_{2i}^l$ is the computation time for the last chunk. Hence an estimate of processing rate is $\lim_{V\to\infty} \frac{C_{max}}{V} = \frac{1}{m\delta}(\sum_{i=1}^{m} S_i +$

$\delta \sum_{i=1}^{m} C_i$). The multi-installment mode is faster when its processing rate is smaller than the one for the out-of-core mode: $\frac{1}{m\delta}(\sum_{i=1}^{m} S_i + \delta \sum_{i=1}^{m} C_i) < \frac{1}{\sum_{i=1}^{m} \frac{1}{A_{i2}^h}}$, from which we get $\delta > \frac{\sum_{i=1}^{m} S_i}{\frac{m}{\sum_{i=1}^{m} \frac{1}{A_{i2}^h}} - \sum_{i=1}^{m} C_i}$. Thus, chunk size δ should be selected from the range $(\frac{\sum_{i=1}^{m} S_i}{\frac{m}{\sum_{i=1}^{m} \frac{1}{A_{i2}^h}} - \sum_{i=1}^{m} C_i}, \max_{i=1}^{m}\{swp_i\})$. For uniform computing systems this expression can be simplified to $(\frac{mS}{A_2^h - mC}, swp)$, where swp is the swap point.

When δ increases the load imbalance may arise and some processors may have to wait idle for the completion of the computations on other processors. Furthermore, the bigger δ is the longer the processors must wait before starting the computations. On the other hand, if δ decreases then the number of messages grows and communication overhead increases. Hence, it can be expected that for some instances of the system parameters an optimum value of δ exists for which processing time is minimum. We propose a heuristics to select δ. The value of δ should be such that a processor is computing during the whole communication round when originator is sending the load to the processors. This results in a requirement $A_{i1}^l + \delta A_{i2}^l \geq \sum_{j=1}^{m}(S_j + \delta C_j)$, for processor P_i. Taking into account all processors: $\delta = \max_{i=1}^{m}\{\frac{\sum_{j=1}^{m} S_j - A_{i1}^l}{A_{i2}^l - \sum_{j=1}^{m} C_j}\}$. For uniform computing system the above formula expressing δ can be simplified to:

$$\delta = \frac{mS - A_1^l}{A_2^l - mC} \tag{4.5}$$

where A_1^l, A_2^l are parameters of the linear function of processing time in the core memory, C, S are communication time parameters. Note that δ can be calculated in this way only if the numerator and the denominator are of the same sign. In equation (4.5) the numerator is positive when $mS > A_1^l$, which means that a processor is able to start computation within the duration of activating communication to all processors. If the numerator is negative then messages arrive faster than the processors are able to process them, and the load will accumulate in communication buffers. Consequently, the numerator and the denominator must be positive. The denominator $A_2^l - mC$ is positive when $\frac{A_2^l}{m} > C$ which means that computing rate of all processors together is greater than communication rate, or in other words, communication speed is greater

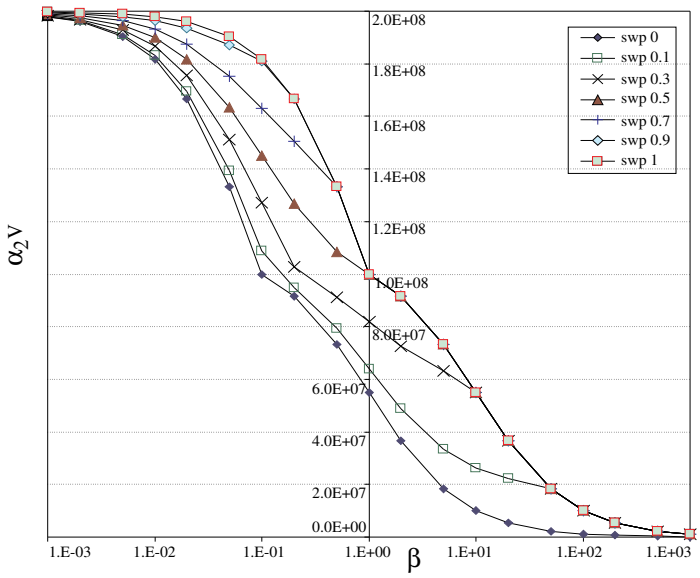


Figure 4.4: Changes in the load partition for $m = 2$ and various $\beta = \frac{A^h_{22}}{A^l_{21}}$, and swp_2.

than the total computing speed of all processors. If the denominator is negative then the total computing speed of processors is greater than the communication speed and some idle times will arise on some processors. The negative denominator or the fact that equation (4.5) expresses a value outside of the admissible interval introduced in the preceding paragraph do not limit applicability of the multi-installment strategy. It means that chunk size δ must be selected in a different way. In practice, by selection of δ applications can be experimentally tuned to obtain good performance.

4.2 Performance Modeling

In this section we present results of modeling dependence of a computing system performance on the model parameters. Over 2400 instances of the linear programs were solved by `lp_solve`, a free linear programming code [14].

Let us analyze optimum distribution of the load under various swap point values and speeds of the processors. This dependence for two processors ($m = 2$) and load size $V =$ 2E8 is presented in Fig. 4.4. We assumed that parameters of P_1 are fixed to

$A^l_{11} = 0, A^l_{12}$ =1E-3, $A^h_{11} = -$9E5, A^h_{12} =1E-2 (hence $swp_1 =$ 1E8). The parameters of speed and swap point of P_2 were variable, except for $A^l_{21} = 0$. In Fig. 4.4 the load $\alpha_2 V$ assigned to processor P_2 is presented on the vertical axis, on the horizontal axis the ratio $\frac{A^h_{22}}{A^h_{12}} = \frac{A^l_{22}}{A^l_{12}} = \beta$ of the processor speeds is shown, various values of the processor P_2 swap points $\frac{swp_2}{V}$ are represented by different curves. As we move to the right along the horizontal axis, the speed of processor P_2 decreases, and its load also decreases. The curve for $\frac{swp_2}{V} = 0$ represents P_2 using the second level of memory only (disk), while $\frac{swp_2}{V} = 1$ represents P_2 able to hold all load V in the first level of memory (core). As the swap point swp_2 increases also the load size $\alpha_2 V$ increases. Curves for $\frac{swp_2}{V} < 1$ do not cross the curve $\frac{swp_2}{V} = 1$ because at the point of such an intersection the load assigned to processor P_2 is small enough to be held in the core, i.e. $\alpha_2 V \leq swp_2$, and the real location of the swap point of P_2 is meaningless. Three intervals of processing rate ratio β can be distinguished in Fig. 4.4. When $\beta <$1E-1 then P_2 has the second memory level faster than the first memory level of P_1. In the interval [1E-1,1E0] P_2 is faster than P_1 but only when core memory is used on P_2. In the third interval of $\beta > 1$, P_2 is slower than P_1 independently of the memory level used. In these three intervals $\alpha_2 V$ changes with different speeds under β changes. This can be seen especially for swap points $swp = 0$, and $swp = 1$ for which the curves are not smooth. When the two processors are identical ($\frac{swp_2}{V} = 0.5, \beta = 1$) the distribution of the load is not exactly equal because processor P_1 receives the load first and computes longer. It can be concluded that even though the mathematical model is linear, the optimum distribution of the load changes nonlinearly with growing difference of the processors.

In the following part of this section we consider homogeneous computing systems only. Therefore we will use a simplified notation in which A^l_1, A^l_2 are parameters of the linear function of processing time for the core memory, and A^h_1, A^h_2 for the out-of-core memory. C, S are communication time parameters. Unless otherwise specified we considered a system with $m = 10$ processors, $A^l_1 = 0, A^l_2 =$ 1E-3, $swp =$1E8, $\frac{A^h_2}{A^l_2} = 10$ and communication parameters $C =$ 1E-6, $S =$ 1E-3.

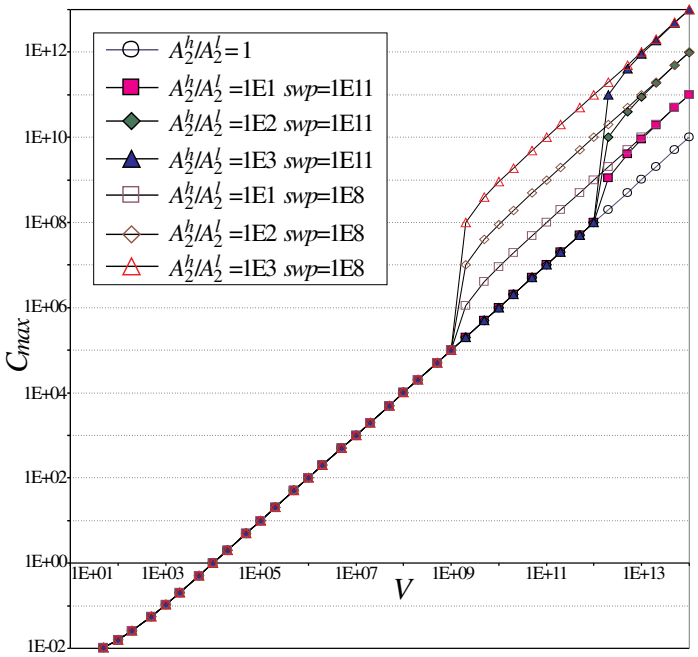


Figure 4.5: Processing time vs. V for various A_2^h/A_2^l and swp.

Fig. 4.5 demonstrates dependence of processing time C_{max} on the problem size V, for various values of the ratio of processing rates in core and out-of-core $\frac{A_2^h}{A_2^l}$, and swap points swp = 1E8, or swp = 1E11. As it can be predicted the $\frac{A_2^h}{A_2^l}$ has some influence on processing time, when the second memory level is used, which is the case for $V > m \times swp$, i.e. load size exceeding the total core memory size.

In Fig. 4.6 dependence of the processing time on size of the problem V for various computing speeds is shown. The curves represent systems with different speeds. A dotted reference line shows communication time equal to $mS + VC$ which is a lower bound on the processing time. In all cases the ratio $\frac{A_2^h}{A_2^l}$ of the processing rates in core and out-of-core and swap points swp were fixed. It can be observed that increasing speed beyond certain level is not profitable because communication becomes a bottleneck. Note that for A_2^l =1E-3, A_2^l =1E-5, and A_2^l = 1E-7 some points are missing in Fig. 4.6. It is the case when some of the processors receive no load. This means that not all m processors can be effectively used because computing on less than m processors is shorter than activating all the processors.

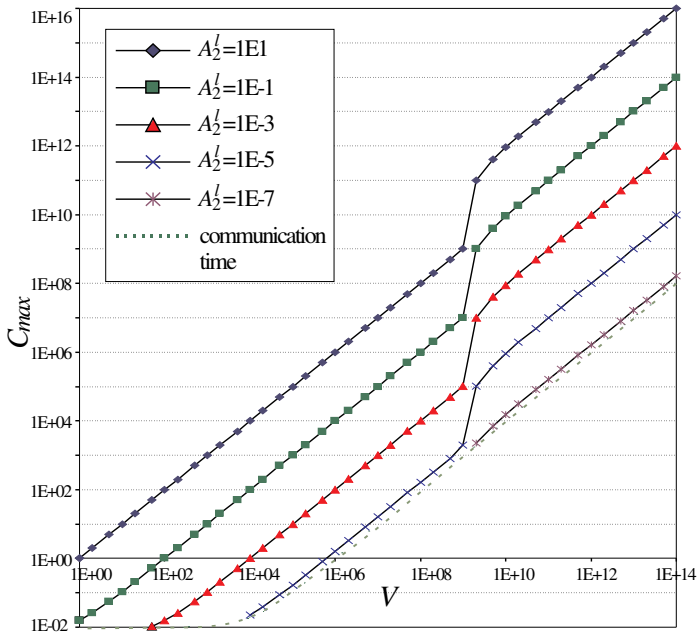


Figure 4.6: Processing time vs. V for various A_2^l and fixed swp, A_2^h/A_2^l.

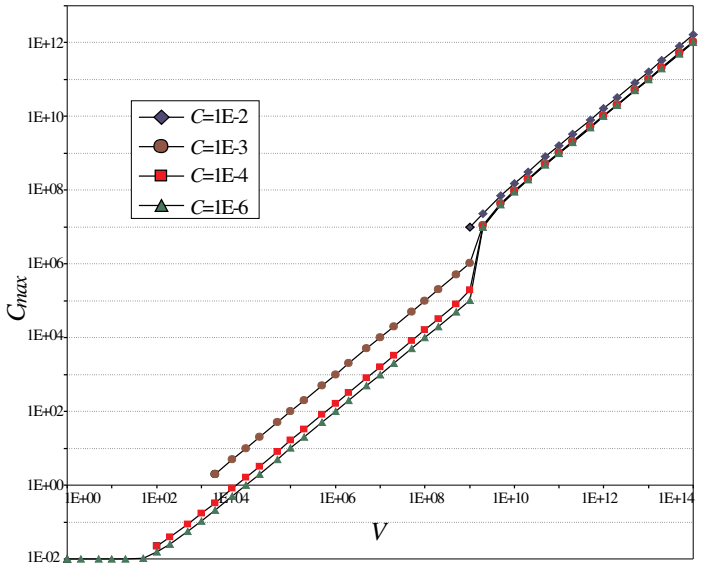


Figure 4.7: Processing time vs. V for various C and fixed swp, A_2^h/A_2^l.

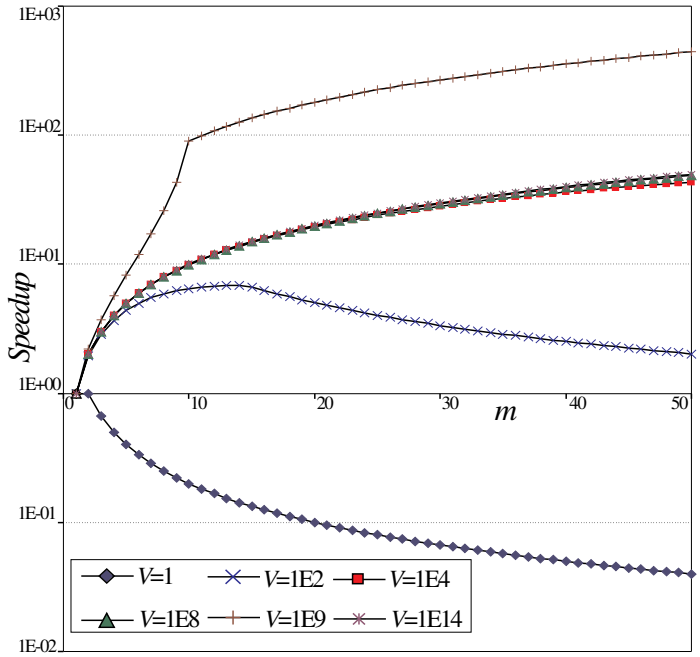


Figure 4.8: Speedup vs m for various V.

Dependence of the processing time on size of the problem V for various communication speeds is shown in Fig. 4.7. It can be observed that processing time decreases with C decreasing only up to a certain limit beyond which computing speed is the limiting factor. Also here not in all cases $m = 10$ processors can be used. When communication speed is small C = 1E-2 all processors can be used for load sizes $V \geq$1E9. As the communication speed increases (i.e. C is decreasing) the size of the problem for which all processors can be used also decreases.

In Fig. 4.8 speedup for various processor numbers m and problem sizes V is shown. The size of $V = 1$ (e.g. byte), certainly, is not practical but it shows behavior of the model. As it can be seen for problem sizes $V = 1$ speedup decreases all the time. It is the case because the load is too small and one processor is able to perform all the computations within the time of activating additional processors. The additional processors receive no load and only unnecessary communication cost is induced. The case of load size V = 1E2 is similar when the number of processors m exceeds 14. For $m \leq 14$ speedup is growing which indicates some profit from parallelism. For other

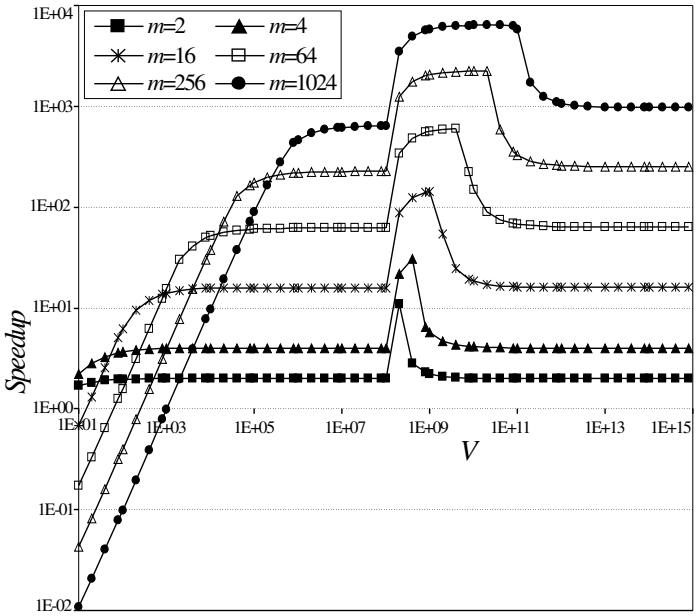


Figure 4.9: Speedup vs V for various m.

problem sizes V speedup is growing. Both when load size is smaller than the core memory $V \leq swp$, and when the problem size by very much exceeds the total core memory size $V \gg m \times swp$, the speedup is similar and close to linear. Therefore, lines for $V =$ 1E4, $V =$ 1E8, $V =$ 1E14 overlap. When $V \approx m \times swp$ =1E9 superlinear speedup can be observed, because using m processors allows for holding most of the load in the core memory, while computing on one processor requires using slower external memory. Fig. 4.8 shows speedup obtained under assumption that exactly m processors are activated by an appropriate message even if some of them receive no load to process. It has been observed that the number of processors for which speedup achieved its maximum, also corresponds to the maximum number of processors for which all processors receive some load. More insight into the behavior of the speedup is given by Fig. 4.9 presenting speedup vs V for various m. It can be seen that superlinear speedup is achieved for the problem sizes V in the range approximately $(swp, m \times swp]$. When the number of processors is too big, speedup decreases with decreasing load V.

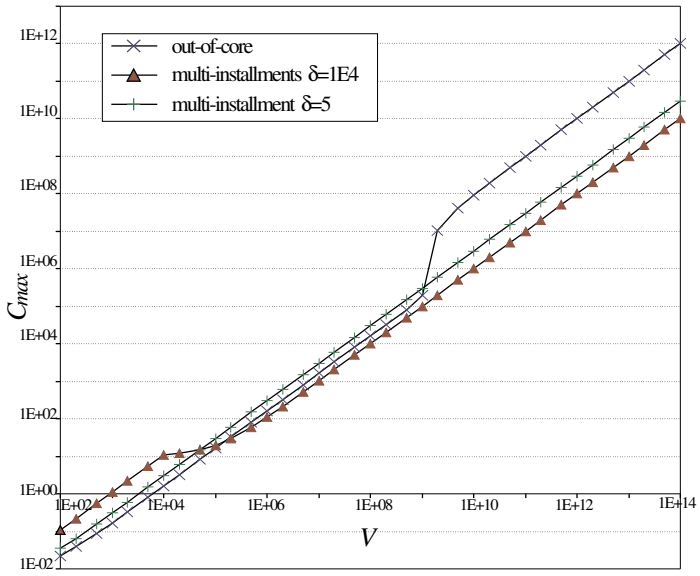


Figure 4.10: Processing time vs. V for multi-installment and out-of-core computations.

4.3 Out-of-core and multi-installment load processing

In this section we compare two modes of processing divisible loads: out-of-core computations which use external memory against multi-installment processing of small pieces of the load on the first level of memory hierarchy, but at the cost of additional communications.

We considered a homogeneous system with $m = 10$ processors, communication rate $C = 9.99\text{E-}5$, communication startup time $S = 1\text{E-}3$, and computing time function coefficients $A_1^l = 0, A_2^l = 1\text{E-}3$, $A_1^h = -9.9\text{E}6$, $A_2^h = 1\text{E-}1$ (hence $swp = 1\text{E}8$). We used equation (4.5) to calculate the load chunk size $\delta = 1\text{E}4$. The dependence of processing time (C_{max}) on the problem size V is shown in Fig. 4.10. Note that both axes are logarithmic, and a small constant difference in this figure can be a big difference in the absolute terms. The three lines in Fig. 4.10 depict processing time in the out-of-core mode, multi-installment using $\delta = 1\text{E}4$, and multi-installment using $\delta = 5$. For $V < 1\text{E}4$ multi-installment with $\delta = 1\text{E}4$ is the worst because only one

chunk of the load is sent and only one processor works, while the other processors remain idle. For $V \in [1E4, m \times swp]$ processing time increases slowly, in the case of multi-installments with $\delta = 1E4$, because more than one load chunk must be sent and additional processors are activated. For load chunk $\delta = 5$ multi-installment processing time is shorter than with $\delta = 1E4$ for loads V smaller than approximately 1E5. It is also better than the out-of-core computation when the second level of memory comes into use. Multi-installment with $\delta = 5$ is worse than distributing the load according to the linear program (4.1)-(4.4) when the core memory is used. It is because the latter distribution has only one communication per processor, and a perfect load balance resulting in simultaneous completion of computations on all processors. As it can be seen in Fig. 4.10 multi-installment mode of processing outperforms the out-of-core computations even for the chunk sizes δ smaller than the one selected according to equation (4.5).

The predictions of our model are confirmed by the computational experiments conducted on a cluster of $m = 3$ Pentium III computers with 1Gbyte of the core memory. The operating system was Red Hat Linux 6.2. The test application was searching for a pattern in a binary file. Communications were done on the basis of a socket library. Fig. 4.11 shows processing time vs. V/m, for out-of-core computations using virtual memory, and multi-installment processing with chunk sizes 1E3, 1E4, 1E6, 1E8. A dotted line representing the linear part CV of the communication time has been added as a reference line. The dashed reference line at the bottom is the computation time on a single processor working off-line. In the out-of-core processing the use of virtual memory is evident when the load assigned to a processor exceeds core memory size. In multi-installment mode processing time is even worse than the out-of-core processing for δ=1E3 because communication overhead dominates. Increasing the chunk size δ reduces the total processing time but only to the limit of communication time required to scatter the load. Therefore, the lines for $\delta = 1E6$, and $\delta = 1E8$ overlap.

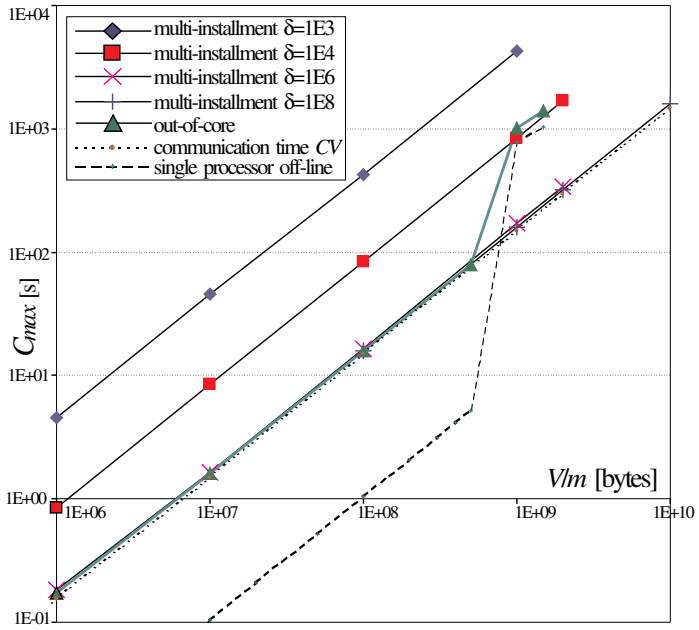


Figure 4.11: Processing time vs. $\frac{V}{m}$ for multi-installment and out-of-core computations on a cluster of Linux PC computers.

4.4 Conclusions

In this chapter we proposed a new mathematical model for distributed processing divisible loads. The model based on linear programming is capable of representing piece-wise linear convex processing time functions of the assigned load. In particular, systems with memory hierarchy can be represented in this way. The influence of the model parameters on the performance of the computing system has been studied. Efficiency of distributed processing divisible loads in installments and out-of-core modes were compared. Multi-installment processing appears to be advantageous for reasonably selected load chunks sizes.

Chapter 5

Systems with Limited Communication Buffers

5.1 Introduction

In this chapter we examine the impact of communication buffer size D on the performance of divisible load processing in various distributed networks. If the communication buffer size is limited then no message may be bigger than D units of load. To our best knowledge, it is the first attempt of this kind in the divisible load theory. The ideal goal of this study is to propose a method of adjusting the size of communication buffer size to the parameters of the system, and the application. We also extend the applicability of divisible load theory, and propose a general methodology of studying the interaction between the communication and computing subsystems under limited communication buffer sizes.

As the communication buffer has size D, loads with size greater than D cannot be sent to the processors in one message. Therefore, communication buffers are filled and messages are sent to their destinations several times. Hence, load is distributed in n *stages*. n_{min} will denote minimum possible number of stages. We assume that there are no memory buffer limitations at the processors and arbitrary load may accumulate over the course of processing. We assume that each load scattering stage

is a repetition of the same communication pattern. Instead of studying scattering algorithms for multiple possible interconnection topologies such as meshes, hypercubes, trees, multistage interconnections we consider three fundamental structures of the scattering and broadcasting algorithms introduced in Section 2.2. Sometime it is convenient to express buffer size as a fraction of whole data volume V. Therefore we will denote by $D' = \frac{D}{V}$ the fraction of whole load constituted by communication buffer size D.

In the ordinary tree topology we distinguish two additional cases depending on the ability (or inability) of the nodes of dealing with more than one message simultaneously. If the node can handle only one message at a time, then the message must be fully received first, and then relayed in full (if needed). We will call this 1-*buffer* case. When the node is capable of simultaneously dealing with more than one message then it is possible to overlap sending one message with receiving another message. This situation will be called 2-*buffer* case. In both cases we assume that processors are able to divide the received message and simultaneously redistribute it via p ports.

In Binomial trees overlapping in time distribution of the loads to different layers and/or stages is not possible for two reasons. In binomial trees nodes from layers $0, \ldots, l$ work synchronously to activate layer $l+1$ for $l = 0, \ldots, h-1$. Therefore, there is no room in the communication algorithm for simultaneous distribution of the load for some other layer of the same stage or the load of the next stage. Furthermore, in some networks the same communication link is used in the opposite directions in different steps of the scattering algorithm, to activate nodes of the consecutive layers. In such cases it is impossible to simultaneously scatter load dedicated to different layers. Consequently, we do not consider 1- and 2- buffer cases in binomial trees.

Note, that using LLF instead of NLF order, 2-buffers instead of 1-buffer, binomial trees instead of ordinary trees are examples of optimizations that can be implemented in the communication algorithms. It will be demonstrated that their impact is limited when it comes to the interactions with the computations.

For the simplicity of the presentation we assume in this chapter that originator

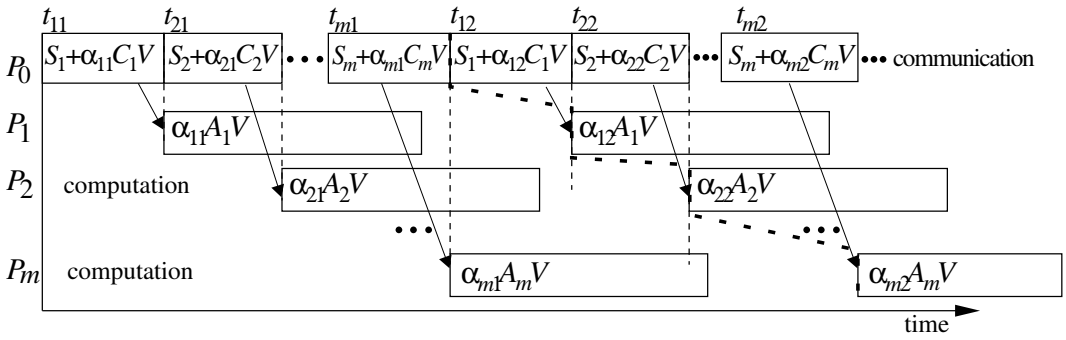


Figure 5.1: Communications and computations in a star interconnection

P_0 does not compute but communicates only.

5.2 Mathematical models

In this section we formulate the problems of finding optimum distribution of the load, taking into account the limited size of the communication buffer.

5.2.1 Star

The minimum number of stages needed to transfer volume V of load to m processors using communication buffer size D is $n_{min} = \lceil \frac{V}{Dm} \rceil$. A Gantt chart depicting communications and computations in a star network is presented in Fig. 5.1. In this section the following extension of the standard notation is used:
α_{ik} - the fraction of volume V sent to processor P_i in stage k,
t_{ik} - the start time of the communication to processor P_i in stage k.
The problem of determining optimum distribution of the load in the star topology for a given number of stages $n \geq n_{min}$ can be formulated as a linear program:

LP **LoadDirect**
minimize C_{max}

subject to:

$$t_{ik} + C\alpha_{ik}V + S \leq t_{(i+1)k} \qquad i = 1, ..., m-1, k = 1, ..., n, \tag{5.1}$$

$$t_{mk} + C\alpha_{mk}V + S \leq t_{1(k+1)} \qquad k = 1, \ldots, n-1 \tag{5.2}$$

$$t_{ik} + S + C\alpha_{ik}V + AV\sum_{l=k}^{n}\alpha_{il} \leq C_{max} \qquad i = 1, \ldots, m, k = 1, \ldots, n \tag{5.3}$$

$$\alpha_{ik}V \leq D_i \qquad i = 1, \ldots, m, k = 1, \ldots, n \tag{5.4}$$

$$\sum_{i=1}^{h}\sum_{k=1}^{n}\alpha_{ik} = 1 \tag{5.5}$$

$$t_{ik} \geq 0, \alpha_{ik} \geq 0 \qquad i = 1, \ldots, h, k = 1, \ldots, n \tag{5.6}$$

In linear program **LoadDirect** inequalities (5.1) ensure that communications of the same stage do not overlap. By (5.2) the succeeding stages do not overlap. Inequalities (5.3) guarantee that all computations can be completed in time C_{max}, by (5.4) the communication buffers do not overflow. Equation (5.5) guarantees that all the load is processed.

5.2.2 Ordinary tree

We assume that processors in the layers work synchronously, and we do not have to analyze separately processors of the layers. We use additional notation:
α_{ik} - The fraction of volume V sent to each processor of layer i in stage k,
t_{ikl} - the start time of sending load to processors in destination layer i from the intermediate node(s) in layer l in stage k, $i = 1, \ldots, h, k = 1, \ldots, n, l = 0, \ldots, i-1$.

Let us analyze the number of stages. Note that the load for the deeper layers is transferred from the originator to a processor in layer 1, via a communication buffer of size D. Thus, successors of this processor receive at most D units of load altogether in a single instalment. Hence, the number of stages is $n \geq n_{min} = \lceil \frac{V}{Dph} \rceil$.

NLF activation order

1-buffer. The communication and computation Gantt chart for this case is presented in Fig. 5.2. Linear program for the problem is:

LP **TreeNLF-1**
minimize C_{max}
subject to:

$$t_{ikl} + Cp^{i-l-1}\alpha_{ik}V + S \leq t_{ik(l+1)} \qquad i = 1, ..., h, k = 1, ..., n, \quad l = 0, ..., i-2 \tag{5.7}$$

$$t_{ikl} + Cp^{i-l-1}\alpha_{ik}V + S \leq t_{(i+1)k(l-1)} \qquad i = 1, ..., h-1, k = 1, ..., n, \quad l = 1, ..., i-1 \tag{5.8}$$

$$t_{hki} + Cp^{h-i-1}\alpha_{hk}V + S \leq t_{i(k+1)(i-1)} \qquad i = 1, ..., h-1, \quad k = 1, ..., n-1 \tag{5.9}$$

$$t_{1k0} + C\alpha_{1k}v + S \leq t_{2k0} \qquad k = 1, \ldots, n \tag{5.10}$$

$$t_{ik(i-1)} + C\alpha_{ik}V + S + AV\sum_{l=k}^{n}\alpha_{il} \leq C_{max} \qquad i = 1, \ldots, h, k = 1, \ldots, n \tag{5.11}$$

$$p^{i-1}\alpha_{ik}V \leq D \qquad i = 1, \ldots, h, k = 1, \ldots, n \tag{5.12}$$

$$\sum_{i=1}^{h}\sum_{k=1}^{n} p^i\alpha_{ik} = 1 \tag{5.13}$$

$$\alpha_{ik} \geq 0 \qquad i = 0, \ldots, h, k = 1, \ldots, n \tag{5.14}$$

$$t_{ikl} \geq 0 \qquad i = 1, \ldots, h, k = 1, \ldots, n, \quad l = 0, \ldots, i-1 \tag{5.15}$$

In the linear program **TreeNLF-1** inequalities (5.7) guarantee that a new communication can only start if the message is fully received first. By (5.8) the next communication to the succeeding layer may not start unless the relaying of the previous message is finished and the buffer at the next layer is ready to be reused. Inequalities

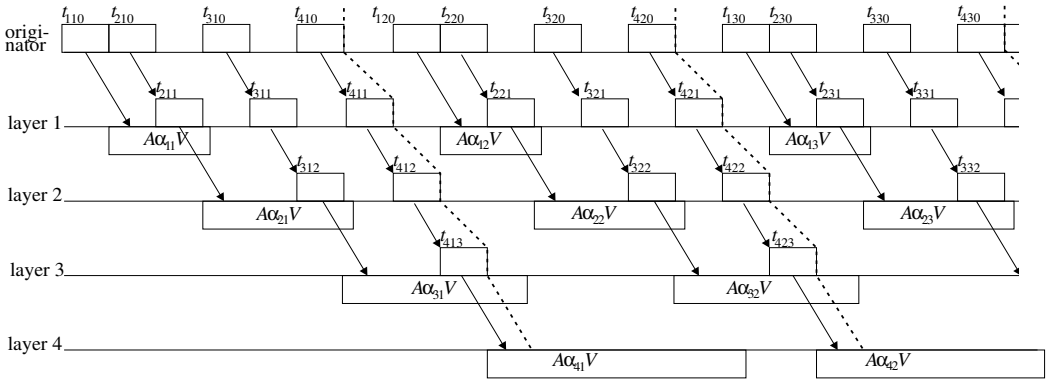


Figure 5.2: Communication and computation in an ordinary tree, NLF 1-buffer.

(5.9) ensure that the communications of the consecutive stages do not overlap. As we use NLF strategy the second communication at the tree top level can start immediately after the first communication to layer 1, which is closest to the originator. Hence we have constraints (5.10). Inequalities (5.11) ensure that all the load received by the processors is processed before the end of the schedule at C_{max}. By (5.12) messages fit into the communication buffers, and by (5.13) all the load is processed.

2-buffers. We assume here that the number of buffers is sufficient to receive and send messages at the same time, and thus, two consecutive communications can be performed simultaneously. However, the buffers are reused in the third following communication. If the load received in the previous message has not been relayed, then the next communication wishing to use this buffer must wait. This restriction is introduced to prevent accumulation of the load in the intermediate layers. The communication and computation Gantt chart for this case is presented in Fig. 5.3. In the following linear program we skip part of the constraints which are the same as (5.11)-(5.15). This is done for the clarity of the presentation

LP **TreeNLF-2**

minimize C_{max}

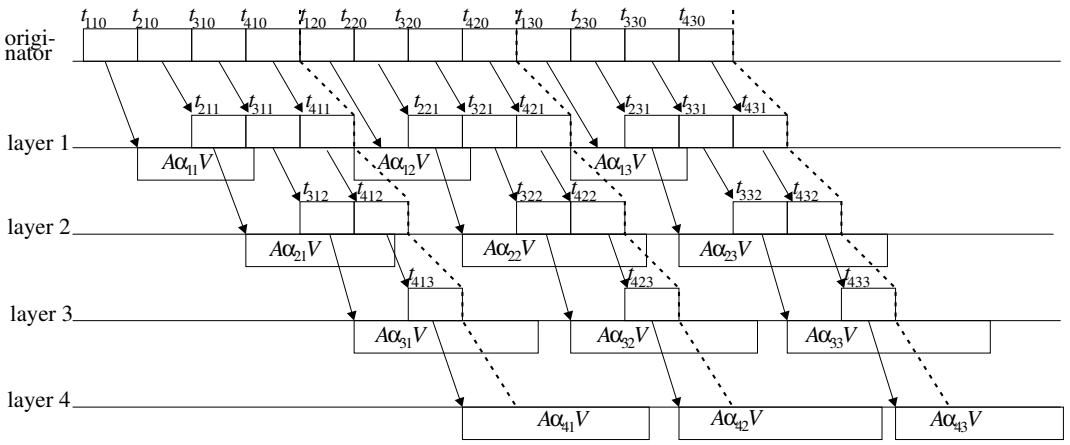


Figure 5.3: Communication and computation in an ordinary tree, NLF 2-buffer.

subject to:

$$t_{ik(l+1)} \geq t_{ikl} + Cp^{i-l-1}\alpha_{ik}V + S \qquad i = 1,...,h, k = 1,...,n, \quad l = 0,...,i-2 \tag{5.16}$$

$$t_{(i+1)kl} \geq t_{ikl} + Cp^{i-l-1}\alpha_{ik}V + S \qquad i = 1,...,h-1, k = 1,...,n, \quad l = 1,...,i-1 \tag{5.17}$$

$$t_{(i+2)k(l-1)} \geq t_{ikl} + Cp^{i-l-1}\alpha_{ik}V + S \qquad i = l+1,...,h-2, k = 1,...,n \quad l = 1,...,h-2 \tag{5.18}$$

$$t_{1(k+1)0} \geq t_{(h-1)k1} + Cp^{h-2}\alpha_{(h-1)k}V + S \qquad k = 1,...,n-1 \tag{5.19}$$

$$t_{2(k+1)0} \geq t_{hk1} + Cp^{h-1}\alpha_{hk}V + S \qquad k = 1,...,n-1 \tag{5.20}$$

$$t_{2(k+1)1} \geq t_{hk2} + Cp^{h-2}\alpha_{hk}V + S \qquad k = 1,...,n-1 \tag{5.21}$$

$$t_{i(k+1)(i-1)} \geq t_{hk(i-1)} + Cp^{h-i}\alpha_{hk}V + S \qquad i = 1,...,h, k = 1,\ldots,n-1 \tag{5.22}$$

In the above LP **TreeNLF-2** inequalities (5.16) guarantee that retransmission of the load to the deeper layers may start only after fully receiving the message. By (5.17) the communications on the same link do not overlap. Inequalities (5.18)-(5.21) ensure that sending a new portion of the load does not start before the buffer at the receiver is released. By (5.22) communications of the succeeding stages do not coincide. The remaining constraints are the same as constraints (5.11)-(5.15) in LP **TreeNLF-1**.

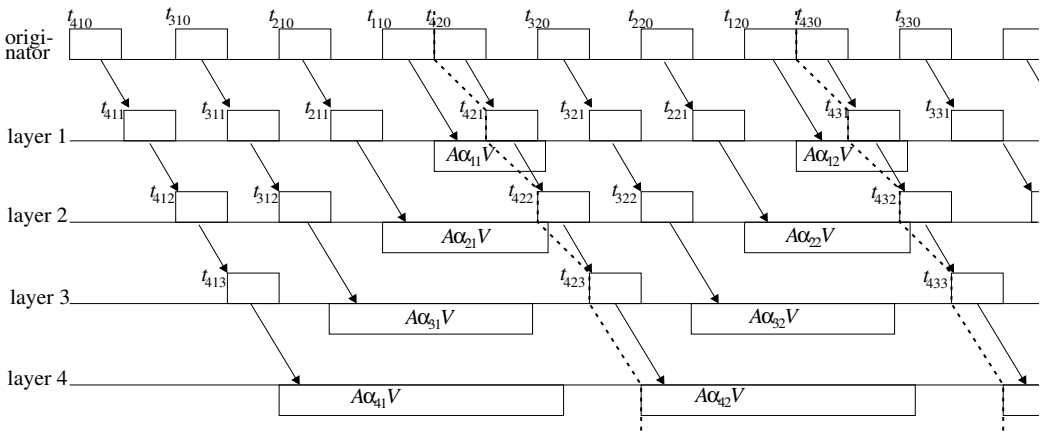


Figure 5.4: Communication and computation in an ordinary tree, LLF 1 buffer.

LLF activation order

In this section the layers are activated in the order of decreasing number of processors.

1-buffer It is assumed here that only one buffer of size D is available at each communicating node. The communication and computation Gantt chart for LLF communication strategy and one buffer in an ordinary tree is presented in Fig. 5.4. In the linear program for this problem constraints identical with (5.11)-(5.15) are omitted for the clarity of the presentation.

LP **TreeLLF-1**

minimize C_{max}

subject to:

$$t_{ik(l+1)} \geq t_{ikl} + Cp^{i-l-1}\alpha_{ik}V + S \qquad i = 1, ..., h, k = 1, ..., n, \quad l = 0, ..., i-2 \tag{5.23}$$

$$t_{(i-1)k(l-1)} \geq t_{ikl} + Cp^{i-l-1}\alpha_{ik}V + S \qquad i = 2, ..., h, k = 1, ..., n, \quad l = 1, ..., i-1 \tag{5.24}$$

$$t_{h(k+1)i} \geq t_{(i+1)ki} + C\alpha_{(i+1)k}V + S \qquad i = 0, ..., h-1, k = 1, \ldots, n-1 \tag{5.25}$$

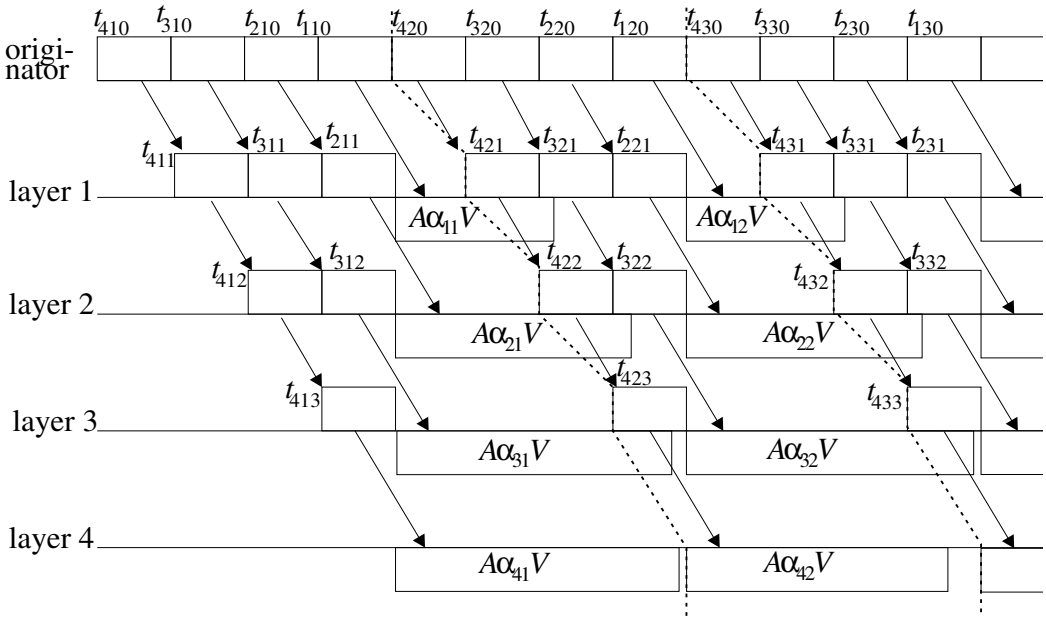


Figure 5.5: Communication and computation in an ordinary tree, LLF 2-buffer case.

In the above formulation inequalities (5.23) guarantee that before the relaying the message is fully received first. By inequalities (5.24) a buffer at the communication switch is not used by two messages simultaneously. Messages from the consecutive stages do not overlap by inequalities (5.25).

2-buffers The communication and computation Gantt chart for LLF communication strategy and two buffers in an ordinary tree is presented in Fig. 5.5.

Linear program for the problem is as follows (constraints (5.11)-(5.15) are omitted for the clarity of the presentation):

LP **TreeLLF-2**

minimize C_{max}

subject to:

$$t_{ik(l+1)} \geq t_{ikl} + Cp^{i-l-1}\alpha_{ik}V + S \qquad i = 1, ..., h, k = 1, ..., n, \quad l = 0, ..., i-2 \tag{5.26}$$

$$t_{(i-1)kl} \geq t_{ikl} + Cp^{i-l-1}\alpha_{ik}V + S \qquad i = 2, ..., h, k = 1, ..., n, \quad l = 0, ..., i-1 \tag{5.27}$$

$$t_{(i-2)k(l-1)} \geq t_{ikl} + Cp^{i-l-1}\alpha_{hk}V + S \qquad i = l+2, ..., h, k = 1, ..., n \quad l = 1, ..., h-2 \tag{5.28}$$

$$t_{h(k+1)0} \geq t_{2k1} + Cp\alpha_{2k}V + S \qquad k = 1, ..., n-1 \tag{5.29}$$

$$t_{h(k+1)i} \geq t_{(i+1)ki} + C\alpha_{(i+1)k}V + S \qquad i = 1, ..., h-1, k = 1, \ldots, n-1 \tag{5.30}$$

In the above linear program **TreeLLF-2** inequalities (5.26) guarantee that the same load is first completely received, only than can it be further relayed. By inequalities (5.27) messages sent by the same layer over the same links do not overlap. By (5.28), and (5.29) no more than two buffers are used in each communication switch. Inequalities (5.30) ensure that messages from the consecutive stages do not overlap.

5.2.3 Binomial tree

Let us analyze the duration of the communication from the originator to layer i in some stage k. In binomial trees nodes receive load once and then redistribute it to the deeper layers of a tree. Therefore, each communication must comprise load not only for the node, but also the load for the successors in a binomial tree. There are $p(p+1)^{i-1}$ processors in layer $i \geq 1$. The originator sends load to layer i in i steps. First, the originator sends over each of its p communication links $p(p+1)^{i-2}\alpha_{ik}V$ load units to layer 1. The remaining load $p(p+1)^{i-2}\alpha_{ik}V$ will be sent to layer i via direct successors of the originator in layers $2, \ldots, i$. Analogously, each processor in layer $j < i-1$ sends load to layer i in $i-j$ steps. First, $p(p+1)^{i-j-2}\alpha_{ik}V$ units of data is sent to layer $j+1$ over each of p ports. The remaining $p(p+1)^{i-j-2}\alpha_{ik}V$ load units are sent from layer j to layer i via j's direct binomial tree successors in layers $j+2, \ldots, i$. Finally, in the last communication step, all layers $0, \ldots, i-1$ simultaneously send $\alpha_{ik}V$ load units to layer i. Note that all layers communicate synchronously, and the same amounts of load are sent from active layers to the next activated layer. Total communication time is equal to $Si + C\alpha_{ik}V(1 + p\sum_{j=0}^{i-2}(p+1)^{i-j-2}) = Si + C\alpha_{ik}V(p+$

$1)^{i-1}$. We will use this closed-form summation result in the following formulations.

As it was noted the layers work synchronously. It means that $t_{ik0}+Si+C\alpha_{ik}V(p+1)^{i-1} = t_{ik1}, t_{ik1}+S(i-1)+C\alpha_{ik}V(p+1)^{i-2} = t_{ik2}, \ldots, t_{ik(i-2)}+C\alpha_{ik}V+S = t_{ik(i-1)}$ (cf. Fig. 5.6, Fig. 5.7). For this reason it is not needed to introduce to the linear programs constraints expressing the fact that the message must be fully received first and only then can it be relayed. Hence, only variable t_{ik0} is needed in the linear program formulations. The remaining variables t_{ikl} for $l = 1, \ldots, i-1$ will not be used.

Let us consider the number of the communication stages. The originator may send at most D units of load to each of its p neighbors in layer 1. As it was said the originator in the first step sends $p(p+1)^{i-2}\alpha_{ik}V \leq D$ load units to its descendants in layer 1. Hence $\alpha_{ik}V \leq \frac{D}{p(p+1)^{i-2}}$. The total number of processors in layer i is $p(p+1)^{i-1}$. Therefore, total load that can be transferred to layer $i > 1$ in a single stage is at most $\frac{Dp(p+1)^{i-1}}{p(p+1)^{i-2}} = (p+1)D$. Note that this quantity does not depend on i. For layer 1 the load is at most pD. As we have layers $1, \ldots, h$ which compute, the total load that can be distributed in one stage is $(h(p+1)-1)D$. Thus, the number of necessary communication stages is $n \geq \frac{V}{D(h(p+1)-1)}$.

NLF activation order

In this section we study Nearest Layer First layer activation strategy, i.e., layers are activated in the order of their distance from the originator. The communication and computation diagram for this case is presented in Fig. 5.6. Linear program formulation is as follows:

LP **BinomialTreeNLF**

minimize C_{max}

subject to:

$$t_{ik0} + C(p+1)^{i-1}\alpha_{ik}V + Si \leq t_{(i+1)k0} \qquad i = 1, ..., h-1, k = 1, ..., n \tag{5.31}$$

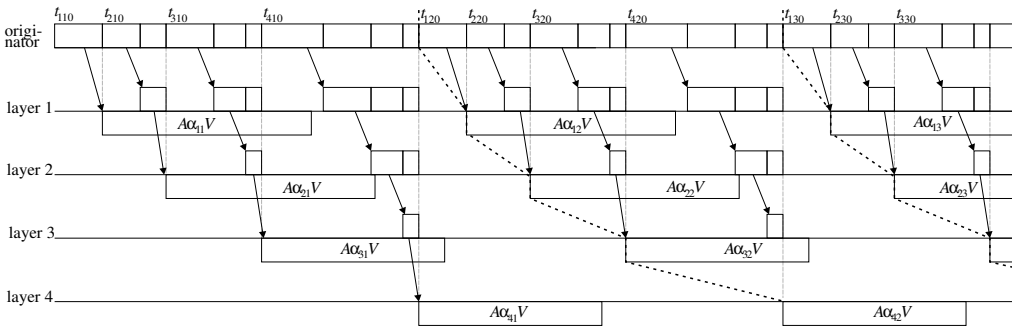


Figure 5.6: Communication and computation in a binomial tree under NLF strategy.

$$t_{hk0} + C(p+1)^{h-1}\alpha_{hk}V + Sh \geq t_{1(k+1)0} \qquad k = 1, ..., n-1 \tag{5.32}$$

$$t_{ik0} + C(p+1)^{i-1}\alpha_{ik}V + Si + AV\sum_{l=k}^{n}\alpha_{il} \leq C_{max} \qquad i = 1, ..., h, k = 1, ..., n \tag{5.33}$$

$$\alpha_{1k}V \leq D \qquad k = 1, ..., n \tag{5.34}$$

$$p(p+1)^{i-2}\alpha_{ik}V \leq D \qquad i = 2, ..., h, k = 1, ..., n \tag{5.35}$$

$$\sum_{i=1}^{h}\sum_{k=1}^{n} p(p+1)^{i-1}\alpha_{ik} = 1 \tag{5.36}$$

$$\alpha_{ik} \geq 0 \qquad i = 0, ..., h, k = 1, ..., n \tag{5.37}$$

$$t_{ik0} \geq 0 \qquad i = 0, ..., h, k = 1, ..., n \tag{5.38}$$

Inequalities (5.31) ensure that the consecutive messages sent by the same layer are not overlapping. By (5.32) the messages from the consecutive stages do not overlap. The remaining constraints are analogous to the ordinary tree case.

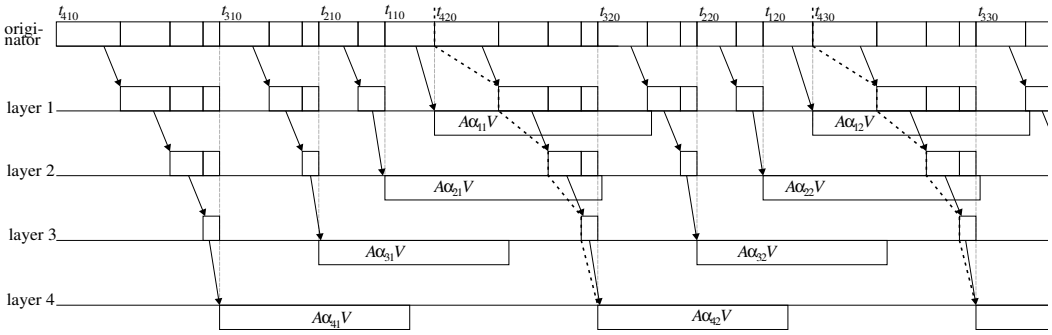


Figure 5.7: Communication and computation in a binomial tree under LLF strategy.

LLF activation order

In this section we study the layer activation order Largest Layer First (LLF), coinciding with decreasing number of processors in the layer. The communication and computation diagram for this case is presented in Fig. 5.7. The linear program in this case is as follows (constraints (5.33)-(5.38) are not repeated for the sake of conciseness):

LP **BinomialTreeLLF**

minimize C_{max}

subject to:

$$t_{ik0} + C(p+1)^{i-1}\alpha_{ik}V + Si \leq t_{(i-1)k0} \qquad i = 2, ..., h, k = 1, ..., n \tag{5.39}$$

$$t_{1k0} + C\alpha_{1k}V + S \leq t_{h(k+1)0} \qquad k = 1, ..., n-1 \tag{5.40}$$

By (5.39) messages sent from a certain layer to other layers do not overlap. Inequalities (5.40) ensure that the succeeding stages do not overlap.

5.3 Performance modeling

Before presenting the results of modeling, let us observe that the space of possible parameter values is enormous, and there is no way to analyze all their possible combinations. Therefore, we restrict the search to the combinations that seem reasonable

for practical applications.

Let us note, that LP formulations in the previous section assumed that all stages and processors will be really needed and activated. It is not always true. When some number of processors or stages is sufficient to process the whole load, then some processors receive no load. Excessive processors introduce additional startup times even though nothing is computed on them. In such situations, the LP formulations should be adjusted by eliminating unnecessary equations related to the superfluous stages and processors.

In the following paragraphs we present results obtained by means of `lp_solve`. Over 2600 instances of LP problems were solved. Though it is a free code `lp_solve` is robust. The description of the largest successfully solved instance exceeded 130 Mbytes. This formulation had over 50000 variables and 156000 constraints. The solution time reached 28 hours on a 1GHz Athlon PC. Unfortunately, as any code, `lp_solve` has its limitations, and we did not manage to solve, e.g., bigger instances. The main reason for failures were numerical instabilities.

5.3.1 Star

In our simulations we modeled a homogeneous system with $m = 10$ processors, $C =$ 1E-6, $A =$ 1E-3, and $S =$1E-3 (these values can be for example: bandwidth 1Mbyte/s, processing rate 1kbyte/s, startup time 1ms). Parameters V, D, were variable.

Let us start this section with considering the observed distribution patterns. In most of the cases the sizes of load chunks sent to the processors grow slowly both with the processor number and the stage number. It is because processors are activated early when the initial chunks are small. In the last stage the chunk sizes decrease in order to achieve simultaneous completion of the computations on all processors. This facilitates perfect load balance. The changes of the data chunks are demonstrated in Fig. 5.8. Processor numbers are aligned along the horizontal axis, sizes of the chunks are presented on the vertical axis. Observe that processor P_{10} received its last load chunk in the penultimate stage, and no load is sent to P_{10} in stage $n = 3$.

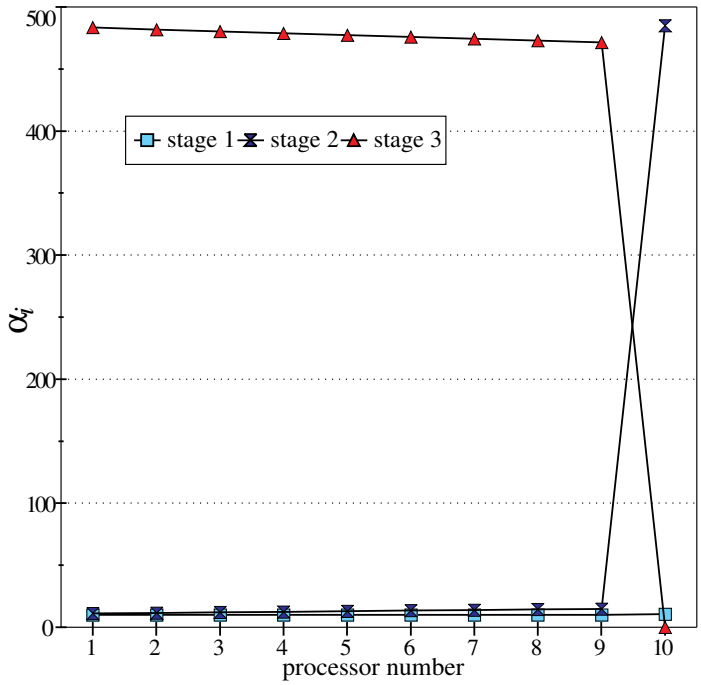


Figure 5.8: Example load chunks sizes in different stages (A = 1E−3, D =1E4, V=5E3) for star.

We show the influence of communication buffer size D on the processing time C_{max} in a Fig. 5.9. The number of stages h was selected such that C_{max} was minimum. Only D =1E0 distinguishes itself from the plots for the other communication buffers. This is a result of excessive fragmentation of communication caused by too small communication buffer. The other D values were sufficiently big to avoid it.

In order to better depict influence of the limited communication buffer size we analyzed the case of the minimum possible number of scattering stages, i.e. $n_{min} = \lceil \frac{V}{Dm} \rceil$. The results are shown in Fig. 5.10. For $D = \infty$ minimum number of stages is $n = 1$, and the minimum number of processors is $m = 1$. The lower bound (LB), representing ideal circumstances of processing the load, is added to show existing potential for the reduction of the processing time. In the ideal case at least one communication startup time must elapse before any processor starts computing. The computing phase may not last shorter than $\frac{VA}{m}$ which is the case of ideal load balance.

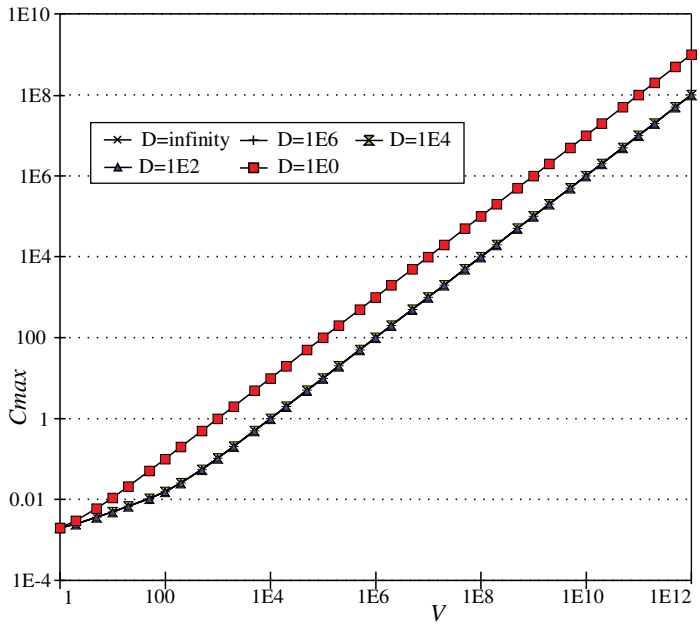


Figure 5.9: C_{max} vs V for various D in a star, A=1E-3 for the best observed n

Thus, $S + \frac{VA}{m}$ is a lower bound we used. In Fig. 5.10 lines for $D = \infty$ and $D = 1E0$ overlap by the selection of values A, C, S. The lines for other communication buffer sizes follow the line of infinite buffer when $V \leq D$, and the lower bound when $V \geq mD$. The changes in processing time for $V \in [D, mD]$ are minor. It is the case because for $V \leq D$, only one processor is activated, as for $D = \infty$. When V exceeds D more than one message must be sent, and it is profitable to activate additional processors. Additional processing power compensates for the growing V, and C_{max} does not increase significantly. If $V > Dm$ all processors are activated, therefore processing time is similar to the lower bound.

Fig. 5.10 shows that relative difference between the processing times for various communication buffer sizes is minor. Yet, the difference in absolute terms can be arbitrarily big. This is demonstrated in Fig. 5.11 showing the difference between the processing time for D=1E2, and other buffer sizes, for n_{min}. There is no difference for $V <$ 1E2. Due to the selection of A, C, S and m values the lines of $D = \infty$ and D=1E0 overlap. The difference of C_{max} for various D grows with V until D,

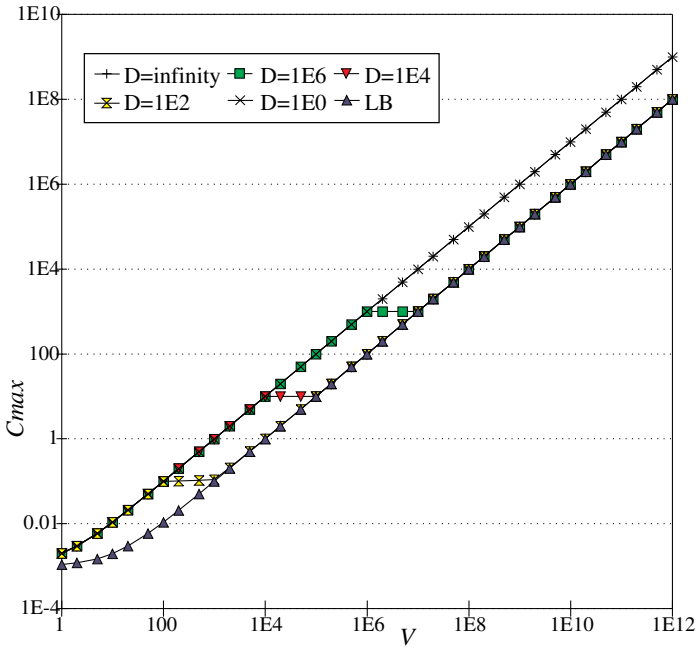


Figure 5.10: C_{max} vs V for various D and n_{min} in a star.

and levels off for $V > mD$. For $V > mD$, D=1E4 and D=1E6 the lines are parallel because the duration of the computing part is the same, and only the communication time in the first stage is different. This is a result of different communication buffers sizes and different message lengths.

In Fig. 5.12 standard deviation of the processor completion times σ for n_{min} is depicted. The value σ is a measure of the processor load imbalance. Observe that the imbalance is the biggest for n_{min}, because $n > n_{min}$ allows for sending messages of differing sizes, hence better load balance is possible. Consequently, Fig. 5.12 shows the worst case of the load imbalance. For small buffers imbalance is smaller. For $D = \infty$ the imbalance can be arbitrarily big because only one processor needs to be activated out of $m > 1$.

On the basis of the above charts one may think that the communication buffer should be small, yet, big enough to avoid excessive message fragmentation. Still, there is one more way in which communication buffer may influence processing time. Consider an example in Fig. 5.13: $m = 3$, $A = C = 1$, $S = 0$, $V = 3$. When $D = 1$,

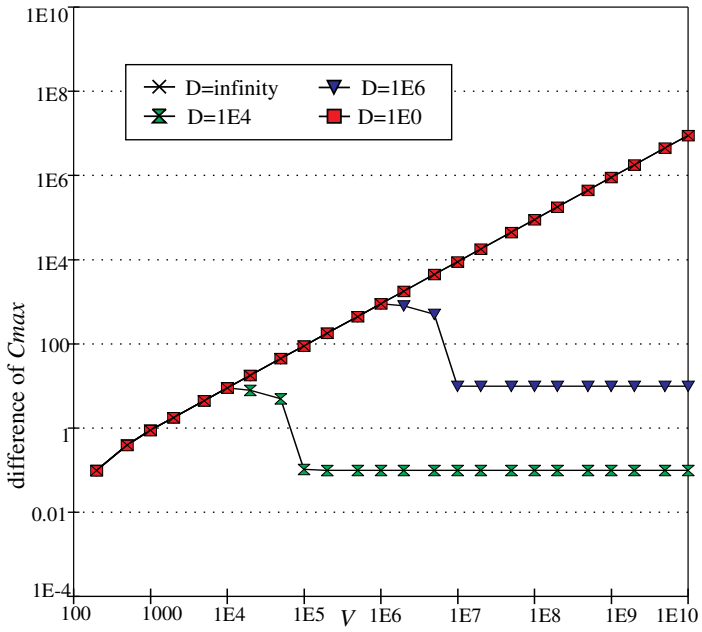


Figure 5.11: Differences between C_{max} for various D in a star, for n_{min}. The processing time for D=1E2 is the reference.

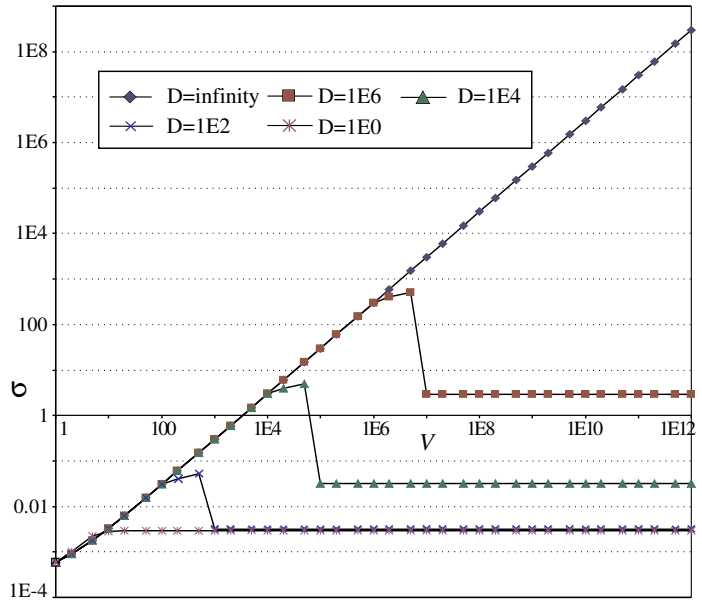
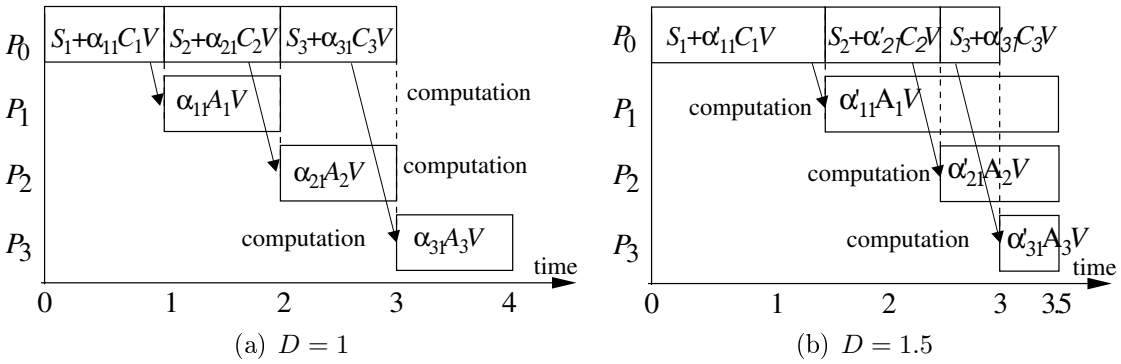


Figure 5.12: Standard variance of processor completion times in a star.



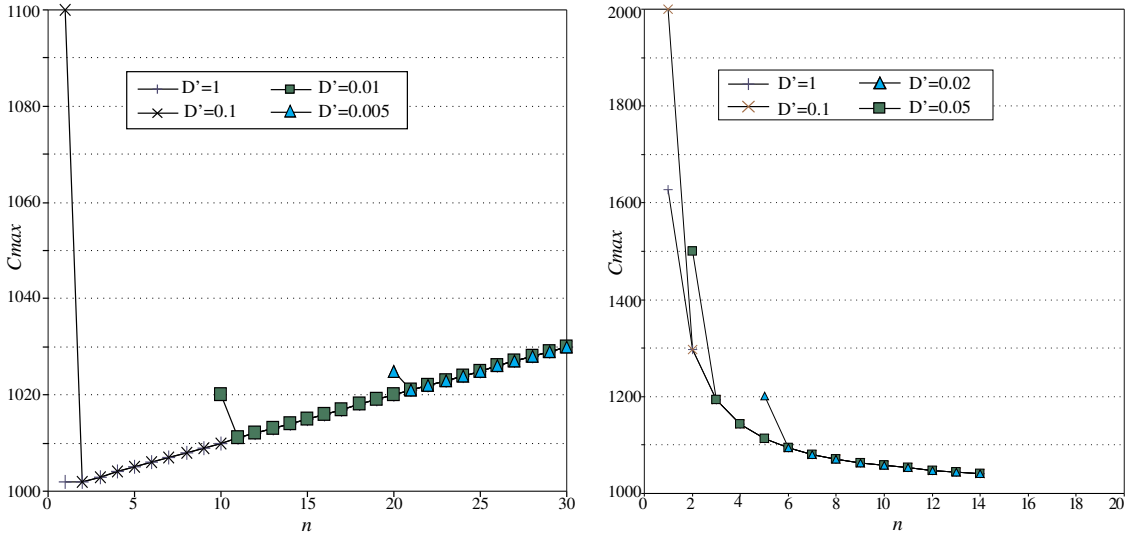
(a) $D = 1$ (b) $D = 1.5$

Figure 5.13: Big buffers may prevent imbalance of the completion time.

only one stage is needed ($n = 1$). The processors receive $\alpha_{11}V = \alpha_{21}V = \alpha_{31}V = D = 1$ of data volume. The communication phase lasts 3 units of time, the computations on the last activated processor complete 4 units of time after the communication start (cf. Fig. 5.13(a). On the other hand, when $D \geq 1.5$ a different load distribution is possible: $\alpha_{11} = 1.5, \alpha_{21}V = 1, \alpha_{31} = 0.5$, all processors simultaneously stop computing 3.5 units of time after the beginning of the process (cf. Fig. 5.13(b)). Thus, too small communication buffer may cause imbalance of the computing completion times also. The load imbalance can be improved by using shorter messages, but this incurs greater number of startup times. Thus the relation between the load balance, message size, buffer size and the number of messages is complex and not straightforward.

In Fig. 5.14 dependence of processing time on D, and n for fixed V is shown. D is expressed as a fraction of V. Initially C_{max} decreases with growing n. The rate of the decrease is fast for small n, but later the returns from increasing n are diminishing. After exceeding a certain limit, C_{max} grows with n. This is an effect of startup time S appearing with each communication. S is added even if the size of the load chunk is 0. This nearly linear growth of C_{max} with n can be considered as an inaccuracy of the model because it means that we still send the assumed number nm of messages even though some of them contain no load.

From the above figures we draw a conclusion that there is some optimal communication buffer size D^*, and number of stages n which on one hand, prevent excessive



(a) A = 1E-4, C = 1E-4, S = 1E-1, V =1E7 (b) A = 1E-3, C = 1E-4, S = 1E-2, V = 1E7

Figure 5.14: Dependence of C_{max} on D, n.

message fragmentation, and the other hand, balances the load well. We discuss this in Section 5.4.

5.3.2 Ordinary and binomial trees

In this section we present results of modeling ordinary and binomial trees. We present relations specific for the trees only because dependence of C_{max} on D, n are similar to the ones for the star. Unless specified differently, we assumed $p = 2, h = 10, A$ = 1E-3, C = 1E-6, S=1E-3 in all the following simulations.

First let us analyze the sizes of the load chunks assigned to the processors of the consecutive layers. Let us note that the loads sent to the deeper layers of both ordinary, and binomial trees are split into chunks each time the load is relayed (cf. inequalities (5.12), (5.35)). The message sent from the originator to the first layer has its size limited to D. Thus, the sizes of messages sent to layer i are at most $\frac{D}{p^{i-1}}$ in the ordinary trees, and $\frac{D}{p(p+1)^{i-2}}$ in the binomial trees. The exponential reduction of the load chunks restricts usability of the deep scattering trees, especially when

the load grains come into play. The optimum distribution of the load among the layers does not expose a fixed regularity. However, some common patterns have been observed. The initial layers with few processors often received no load. It appears to be advantageous to activate deeper layers which have more processors while omitting the initial layers. The size of the chunks sent to the deep layers is restricted by the communication buffer size D used for the communication between the originator and the first layer. Therefore, to exploit the processors of the deep layers to the full extent, the maximum load was sent, i.e. $\frac{D}{p^{i-1}}$ in the ordinary trees, and $\frac{D}{p(p+1)^{i-2}}$ in the binomial trees. The selection of the layers to be used, and the order of activating them remain open problems which have combinatorial nature. Processing in trees resemble a heterogeneous star: the layers are heterogeneous processors connected to the originator via heterogeneous communication links. In the heterogeneous star the optimum selection of the processors to be used and the activation order have combinatorial nature, and is and NP-hard problem as demonstrated in Section 3.1.

The complex nature of the optimum layer activation order is exposed in one more way. A bigger load may be processed in a shorter time than a smaller load when NLF activation order is applied, and h, n are minimum possible. This is illustrated in Fig. 5.15 showing dependence of the processing time on the size of the problem for binomial tree with NLF activation order and $D = 1E2$. Three lines depict processing times for the minimum number of the stages and layers, for the best observed case, and for an alternative communication strategy (LB). It can be seen in Fig. 5.15 that processing time may decrease with increasing V for n_{min}. The explanation for this counterintuitive behavior is that for the given V only a certain number of layers can be activated within the limited span of communication time. Adding more layers is not productive because there is no work for them. On the other hand, adding a little more load allows for activating a new layer which has at least the same number of processors as all the preceding layers (because of NLF activation order). This allows for shifting most of the load to the deep layer, and thus reduces the processing time. The medium line of the best observed case illustrates the possible benefit of increasing the number

of stages. However, there is a technical difficulty in finding optimum distributions for big n caused by the instability of LP solver. The third line (LB) shows potential gains from using a different communication strategy which bases on selecting one deep layer for computations while using the other layers for communications only. The alternative communication strategy was constructed on the following basis. At least one layer must be activated. Suppose only layer i is activated, and the load is sent in equal chunks in all stages. The number of processors in layer i is $p(p+1)^{i-1}$. The processing time is $C_{max}(n) = n * (iS + \frac{V}{np(p+1)^{i-1}}C(p+1)^{i-1}) + A\frac{V}{np(p+1)^{i-1}}$, which is a function of n. The first derivative of C_{max} over n is $iS - \frac{AV}{p(p+1)^{i-1}n^2}$. C_{max} has minimum for $n^* = \sqrt{\frac{AV}{iSp(p+1)^{i-1}}}$. The lower bound can be selected as minimum $C^1_{max}(n^*)$ over layers $i = 1, \ldots, h$. The lower bound found in this way assumes that the communication cycle lasts longer than computing the load received in one stage. Otherwise a different formula expresses the processing time: $C^2_{max} = (iS + \frac{VC}{n^*p}) + \frac{AV}{p(p+1)^{i-1}}$ where the number of iterations is arbitrarily set to n^*. The line LB in Fig. 5.15 shows the maximum of the processing time for these two situations. This strategy is effective as far as processing time is considered, but average utilization of the processing resources may be unsatisfactory. It can be concluded that processing time depends on the selection of the activated layers, and the activation order. To avoid arbitrary decisions in selecting the set of layers to activate we considered the minimum number of layers in the following discussion.

The dependence of C_{max} on V for various D, LLF activation order is shown in Fig. 5.16. These dependencies for binomial tree with NLF activation order and ordinary trees are very similar. In the case of $D = \infty$ only one message is needed to send all the load. Hence, only p processors in layer 1 are activated. Processing time for $D = 1E0$ is bigger than for $D = \infty$. This means that communication buffer is too small, message fragmentation is excessive, communication time dominates and is even longer than processing the whole load on one layer. For $D \in$[1E2,1E6] the changes of the processing time are similar to the case of the star topology (cf. Fig. 5.10). When $V < pD$ only one layer of processors is activated. When $V \in [pD, (h(p+1)-1)D]$

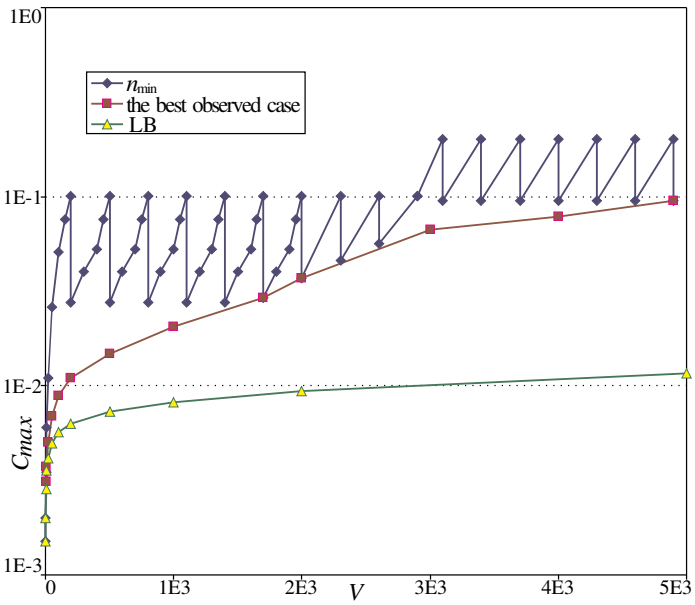


Figure 5.15: Processing time in a binomial tree for small loads (NLF, D = 1E2, A=1E-3, C=1E-6, S=1E-3).

the growing V is compensated for by activating additional layers. Note the little bumps in this interval resulting from the changes of the number of layers that can be activated within the communication time. When $V > (h(p+1)-1)D$ processing times approach the same line. This results from the fact that processing time in the first layer sets the time span of a single stage. In other words communications to the deeper layers and computing in the deeper layers is shorter than computing in the first layer. The computing time in the first layer is DA, when maximum size of the buffer is utilized. The number of stages is $n \geq \frac{V}{D(h(p+1)-1)}$. Thus, combining these two formulae we get processing time $nDA \approx \frac{VA}{(h(p+1)-1)}$ which is the asymptote approached by the lines for $D \in$[1E2,1E6]. This situation could have been avoided provided that the first layer were not computing, but only relaying the load. The lowest line (LB) represents an alternative communication strategy described in the preceding paragraph. Big difference between the LB and other lines demonstrates that processing time can be reduced by using a completely different communication method.

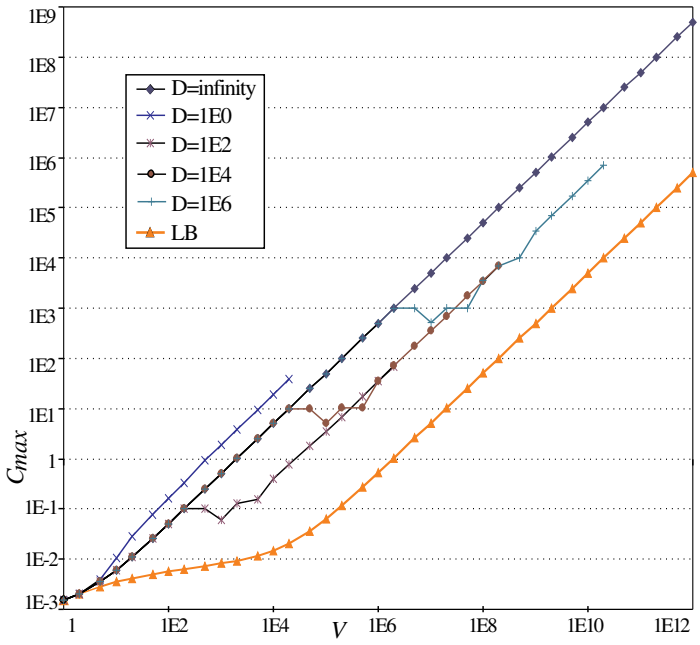
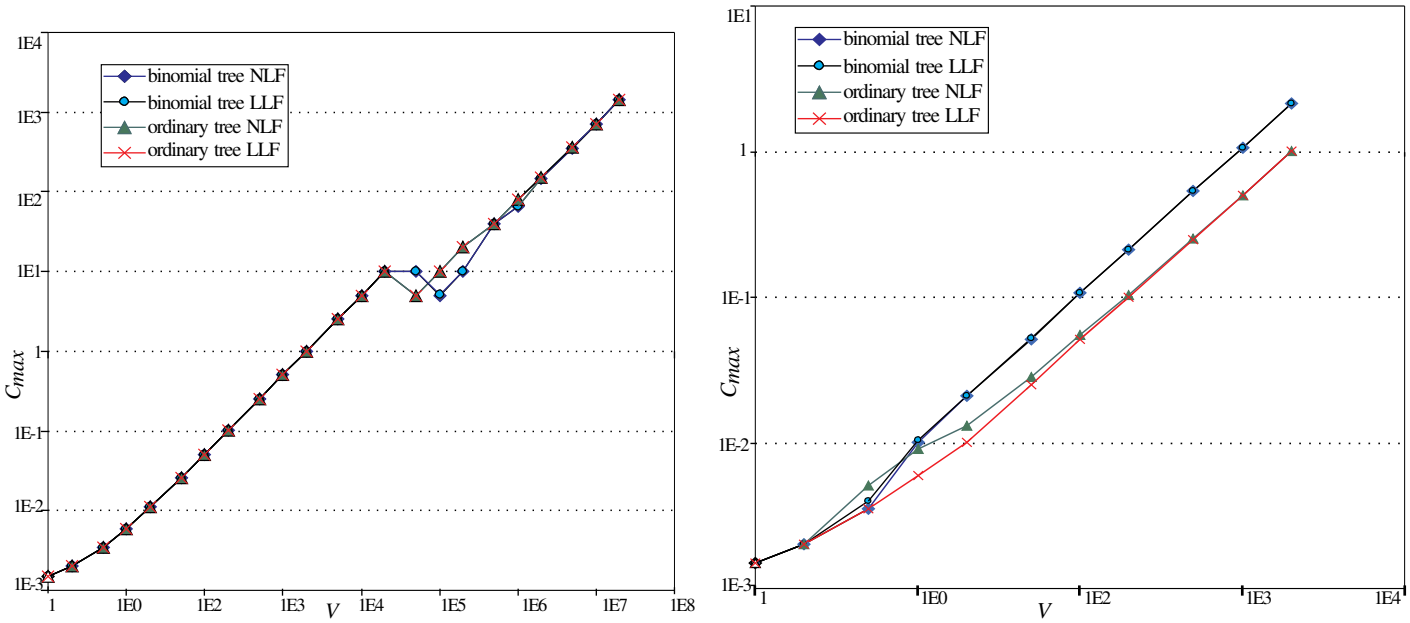


Figure 5.16: Processing time in a binomial tree vs V for various D (LLF,n_{min}).

In the following discussion we compare, in the sense of processing times, NLF with LLF layer activation orders, 1-buffer with 2-buffer case. Let us observe that comparing the binomial trees with the ordinary trees is not easy because different numbers of processors are activated in these structures. Therefore, we compared a binomial tree with $h = 5$ which has $m = 243$ processors with an ordinary tree which has $h = 7$ and $m = 255$ processors. Hence, the difference in processing power is less than 5%. In Fig. 5.17(a) the dependencies of C_{max} on V for the binomial trees and the 1-buffer ordinary trees with NLF and LLF activation orders for $D = 1E4$ are shown. This relation for communication buffer sizes $D > 1E0$ and 2-buffers in the ordinary tree are very similar in the nature. The differences between all the cases appear only in the range $[pD, D(h(p+1)-1)]$. The explanation is as follows: for $V < pD$ only one layer is activated in all cases. For $V > D(h(p+1)-1)$ the whole processing time is dominated by computations on the first layer because the deeper layers receive inadequate load, as mentioned in the preceding paragraph. As it can be seen there are problem sizes where the ordinary tree dominates, and



(a) $D = 1E4$ (b) $D = 1E0$ (n_{min})

Figure 5.17: Processing time in binomial trees and 1-buffer ordinary trees vs V.

problem sizes where the binomial tree dominates. Though the binomial tree has smaller total number of processors, it is hard to claim it is better than the ordinary tree because in neither case has the computational capacity been fully exploited. In Fig. 5.17(b) the same relationship for $D = 1E0$ and ordinary trees with two buffers is shown. For $V > D(h(p+1)-1)$, i.e. when $n > 1$, processing time for ordinary tree is approximately 50% of the time for binomial tree. This situation is caused by different times that elapse between the stages. Though communication times are shorter in binomial trees, the interstage time is longer because communications of the consecutive stages may not overlap (see Fig. 5.6, Fig. 5.7). On the contrary, the communications of the consecutive stages can overlap in the ordinary trees (cf. Fig. 5.2 , Fig. 5.5)

The difference between processing in LLF and NLF layer activation modes for the binomial trees are shown in Fig. 5.18. The NLF activation order is faster than LLF with the exception of several cases for $D = 1$. The LLF is almost always slower due

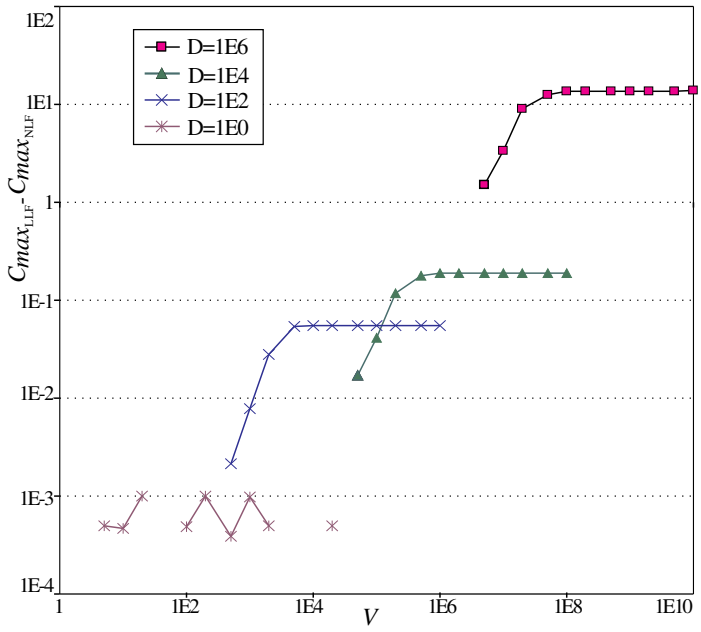


Figure 5.18: Difference of processing time in binomial trees for NLF and LLF activation orders

to the coincidence of the following phenomena: As deeper layers are underutilized, processing time is dominated by computing in the first layer. Thus, the time of activating the first layer is greatly influencing the total processing time. In LLF the first layer is activated as the last one. Therefore, NLF dominates. Note, that this situation completely reverses the domination of the LLF activation order for networks with the unlimited communication buffers shown in [45]. For ordinary trees the difference between the processing time for NLF and LLF activation orders is very similar.

The difference in processing time between 1-buffer and 2-buffer ordinary trees with LLF activation sequence is presented in Fig. 5.19. As it can be seen for $D > 1$ the difference stabilizes. The explanation is the same as for the difference between the NLF and LLF activation orders. Processing time is dominated by the computations on the first layer. 2-buffer tree allows for faster activation of the first layer. For $D = 1$ the difference steadily grows with V because messages are short and the load of a certain layer is computed within the interval of communications with the other

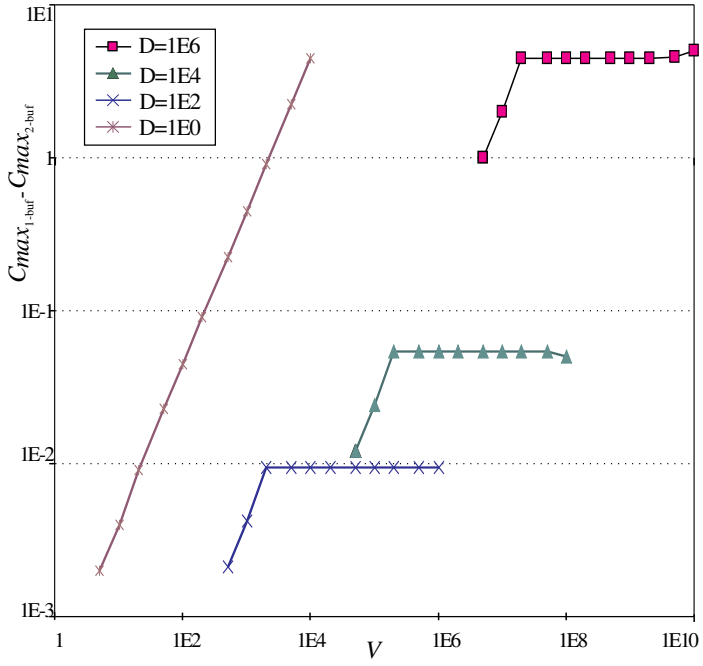


Figure 5.19: Difference of processing time in ordinary trees for 1-buffer and 2-buffer communications, LLF.

layers. Thus, mainly communication time matters in the whole processing time. Communication with 2 buffers are faster than 1-buffer communications. Hence 2-buffer case domination grows with growing V. In NLF activation case situation is similar for $D = 1$. Yet, for $D > 1$ no difference between 1- and 2-buffer cases has been observed. Since computations on layer 1 determine total processing time, and layer 1 is activated first in NLF, advantages of shorter communication to other layers in 2-buffer case have no influence on the processing time.

5.4 Discussion and conclusions

The influence of the limited communication buffer size manifests in several ways. When the communication buffer is too small messages are too short, excessively fragmented and processing time is dominated by communication time. Insufficient communication buffer may cause load imbalance. On the other hand, also big buffers

may cause imbalance when minimum possible number of stages is used. Thus, even for big buffers it is worthy to implement limited sizes of the messages to activate computations quickly, or to balance the load. There is a direct relation between the communication buffer size and the number of stages. It is generally advantageous to have many scattering phases because all processors are activated, and the load is balanced better.

In binomial and ordinary trees we observed that communication buffer size significantly restricted the amounts of load which could be transferred to the deeper layers. The bandwidth of the initial layers was to small to feed deeper layers with load. Consequently, deeper layers completed computations before receiving a new chunk of load, and the processing time was dictated by the first layer. This phenomenon is a great loss of efficiency. Optimized communication methods using, e.g., binomial trees, LLF strategy, or 2-buffer communication nodes, do not outperform their less advanced counterparts, e.g., based on ordinary trees, NLF strategy, 1-buffer nodes. Several remedies can be suggested to alleviate this drawback. It is possible to increase the communication buffer size such that processors receive enough load to keep computing during the whole communication phase of one stage. Still, this solution does not scale well because communication buffers would have to grow exponentially with the number of layers. A better solution is to change the communication algorithm and, e.g., send several messages to the deep layers per each message sent to layer 1. It is possible to use the initial layers for communication only, and the deep layers for computations. Results of using such an alternative strategy are shown as LB in Fig. 5.16.

Let us now address the main goal of this chapter: the optimum size of communication buffer. The size of the buffer cannot be considered separately from the number of stages n. Let us observe that due to the many degrees of freedom in the construction of communication algorithms, and divisible nature of the load, finding a generally optimum solutions may be difficult. It seems difficult to propose an idea on which a proof of the optimality could be based. Nevertheless, some practical and good solutions

are needed. As far as the communication buffer size is considered it is a reasonable idea to have the communication buffer size D^* such that processors keep computing until the next communication phase. In a star it means $AD^* \geq m(S + D^*C)$, and $D^* \geq \frac{mS}{A-mC}$. Note that $D^* > 0$ only when $A > mC$. Thus, idle times arise inevitably when m is too big or A to small. Now we estimate analogous buffer size for binomial and ordinary trees. Let us omit the initial changes of message sizes, and let us assume that all messages sent from the originator are of size D^*. Then, in binomial trees the processors in layer i receive load $\alpha_{ik} = \frac{D^*}{p(p+1)^{i-2}}$ in some stage k. The communication time from the originator to layer i is $Si + C(p+1)^{i-1}\alpha_{ik}$. After substituting α_{ik} and summing over layers $i = 1, \ldots, h$ we get total communication time in one stage $\frac{S(h+1)h}{2} + \frac{CD^*(p+1)}{p}$. Since layer h receives the least load its computing time $\frac{AD^*}{p(p+1)^{h-2}}$ is the shortest in a tree. The requirement that computing time is at least equal to the communication time can be formulated as $\frac{AD^*}{p(p+1)^{h-2}} \geq \frac{S(h+1)h}{2} + \frac{CD^*(p+1)}{p}$, from which we obtain $D^* \geq \frac{S(h+1)hp(p+1)^{h-2}}{2(A-C(p+1)^{h-1})}$. Note that also here D^* exists only when $A > C(p+1)^{h-1}$, i.e. using too many layers h may result in inevitable idle times. In 2-buffer ordinary tree communications from the originator to layer 1 can be overlapped with the communications from layer 1 to the deeper layers. Thus communications in one stage last $h(S + CD^*)$. Computations in layer h which receives the least load last $\frac{D^*A}{p^{h-1}}$. Hence, we get the requirement $D^* \geq \frac{hS}{A/p^{h-1}-Ch}$. In 1-buffer ordinary tree case communication from the originator to layer 1 may not be overlapped with the communication from layer 1 to the deeper layers. The two messages can be sent in time $2S + D^*C + CD^*/p$. Thus, the communication time in one stage lasts approximately $hS + hD^*C(1 + \frac{1}{p})$. The computing time on the last layer is $\frac{D^*A}{p^{h-1}}$. From this we get a requirement $D^* \geq \frac{hS}{\frac{A}{p^{h-1}} - hC(1+\frac{1}{p})}$. Note that the above formulae expressing D^* link the structure of the tree (in values of h, p), communication parameters S, C and computing rate A. Not for all combinations of these parameters can the relation be satisfied.

As far as the number of stages is considered, it should be observed that processing time initially decreases fast with n, but then it stabilizes or even increases (see

Fig. 5.14). Thus, the number of stages should not be significantly bigger than n_{min}. It can be, e.g., $n = n_{min} + k$, where $k < 10$.

Let us summarize what was achieved in this chapter. We proposed a formal methodology of analyzing divisible load computations in a distributed system with limited communication buffers. This method is deterministic and computationally tractable. Modeling performance of various scattering algorithms allowed for studying the influence of communication buffer size on the efficiency of distributed computations. Interactions of several scattering algorithms with computations under limited communication buffer size have been analyzed. We observed severe performance limitations incurred by the tree structures. The results regarding communication optimization reach beyond just selecting a good communication buffer size. The results also show versatility of divisible load theory which establishes a link between scheduling and communication optimization.

Chapter 6

Multi-installment Divisible Job Processing

In this chapter we study regular multi-installment divisible load processing. The problem considered here is similar in many ways to the problem considered in Chapter 5. Yet, the problem of the optimum divisible load processing with multiple installments is approached from a different point of view. Here we assume that the number of installments n and the installment sizes α_i are the decision variables. From n, problem size V and system parameters memory requirements will be derived. Furthermore, unlike as in the algorithm from Chapter 4 we allow for changing the sizes of installments in nearly unlimited range. In Chapter 5 we allowed for accumulation of the load as a result of communication faster than the computations. It is not the case in this chapter.

6.1 Introduction to multi-installment processing

In regular multi-installment processing the load is sent from the originator to the processor many times but the processors are repeatedly activated in the same order. We also assume a nonzero startup time. With the zero startup time it is possible to prove that communications should be done in the infinite number of steps, and

therefore it is unrealistic. In this chapter we also analyze the optimal number of communication steps n^* required to achieve the shortest schedule length.

If the originator communicates with other processors only once, it has been proved that all processors should stop computing at the same time in order to achieve the optimal schedule length. The proof can be found in [19]. The same way of reasoning can be applied to the multi-installment processing. Let us note that two kinds of idle time can appear. The first kind is inactivity in transmission. It appears when the originator has to wait after sending a portion of data, because the next processor is still processing the previous part of the load and is not ready to receive a transmission. The second kind of idle time takes place when the processor has to wait for the next piece of data after completing processing of the previous one, because the originator is busy communicating with other processors.

In this chapter we distinguish two kinds of processing elements depending on the ability to communicate and compute in parallel. The processors with front-end and processors without front-end. We assume homogeneous system, but similar reasoning can be applied to a heterogeneous system.

In this chapter sizes of the load pieces are denoted by $\alpha_1, \ldots, \alpha_{mn}$. The pieces are numbered in the reverse order of sending them. Thus, α_1 denotes the last piece sent to the processor P_m and α_{mn} denotes the first piece sent to the processor P_1.

Lemma 6.1. *In the optimal multi-installment divisible job processing without memory limit, the processors should have no idle times in computing i.e. between completing the processing of the previous piece of the load and starting the next one.*

Proof. Let us assume that in optimal schedule there is only one break in transmission and one processor P_j, after completing computation of previous part of the load must wait for time I before it start receiving the next load. We give a constructive proof that this interval can be closed by shifting some load from the end of the schedule. This load transfer results in the reduction of the schedule length.

Let us calculate the amount of load γ_i which can be subtracted from the end of the schedule so that computations on all processors finish x units of time earlier (cf.

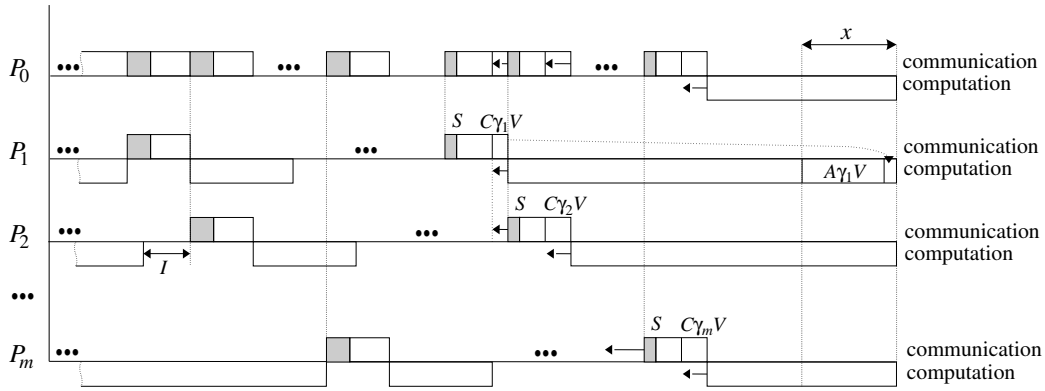


Figure 6.1: Multi-installment processing with computation idle time.

Fig 6.1. Let $\gamma_1, \ldots, \gamma_m$ be the pieces of the load collected on processors $P-1, \ldots, P_m$ respectively. Decreasing the load on P_1 shortens the schedule by $x = \gamma_1 V(A+V)$. For $P_l (l = 2, \ldots, m)$ the computations finish by $x = \gamma_l V(A+V) + VC \sum_{i=1}^{l-1} \gamma_i$ time units earlier. Where $VC \sum_{i=1}^{l-1} \gamma_i$ is the result of starting the communication to P_l earlier, because processors $P_1, \ldots, P_{l-1}$ receive less load. Comparing the formulae expressing the decrease of the schedule length X on $P-1$ and $P_2, \ldots, P_m$ we get $\gamma_i = \gamma_1 (\frac{A}{A+C})^{i-1}$. Note that the load γ_0 removed from originator P_0 is equal to the load removed from processor P_m. In the last stage the load on $P_0, P_1, \ldots, P_m$ decreased by

$$\begin{aligned} \sum_{i=0}^{m} \gamma_i &= \gamma_1 \sum_{i=1}^{m} (\frac{A}{A+C})^{i-1} + \gamma_1 (\frac{A}{A+C})^{m-1} = \\ &= \gamma_1 \frac{A+C}{C} (1 - (\frac{A}{A+C})^m) + \gamma_1 (\frac{A}{A+C})^{m-1} \end{aligned} \tag{6.1}$$

The load removed on processors $P_0, P_1, \ldots, P_m$ must compensate for the load $\beta_j = \frac{I}{V(A+C)}$ shifted to processor P_j earlier in the schedule to remove the idle interval in the computations of length I. Thus we have $\sum_{i=0}^{m} \gamma_i$
$beta_j$, and

$$\gamma_1 = \frac{\beta_j}{\frac{A+C}{C}(1 - (\frac{A}{A+C})^m) + \gamma_1 (\frac{A}{A+C})^{m-1}}. \tag{6.2}$$

The schedule length decreased by

$$x = \gamma_1 V(A+C) = \frac{\beta_j V(A+C)}{\frac{A+C}{C}(1-(\frac{A}{A+C})^m)+(\frac{A}{A+C})^{m-1}}. \tag{6.3}$$

On the other hand, adding load earlier in the schedule increased the lengths of the communications to P_j and delayed all the later communications and computations by $\beta_j VC$. Thus, the total reduction of the schedule length is

$$\begin{aligned} L &= \frac{\beta_j V(A+C)}{\frac{A+C}{C}(1-(\frac{A}{A+C})^m)+(\frac{A}{A+C})^{m-1}} - \beta_j VC = \\ &= \beta_j V \frac{(A+C-(A+C)(1-(\frac{A}{A+C})^m)+(\frac{A}{A+C})^{m-1})}{\frac{A+C}{C}(1-(\frac{A}{A+C})^m)+\gamma_1(\frac{A}{A+C})^{m-1}} = \\ &= \beta_j V \frac{(A+C)(\frac{A}{A+C})^m - C(\frac{A}{A+C})^{m-1}}{\frac{A+C}{C}(1-(\frac{A}{A+C})^m)+\gamma_1(\frac{A}{A+C})^{m-1}} = \\ &= \frac{I(A-C)\frac{A^{m-1}}{(A+C)^m}}{\frac{A+C}{C}(1-(\frac{A}{A+C})^m)+\gamma_1(\frac{A}{A+C})^{m-1}} \end{aligned} \tag{6.4}$$

Let us note, that $A > C$ because otherwise parallel processing makes no sense and a single processor would compute any load before it arrives at any remote processors. Hence, $L > 0$. We demonstrated that the optimum schedule with and idle interval in the computations on some processor can be shortened, and thus it was not optimal schedule. Thus, we have a contradiction. We conclude that an optimal schedule should have no idle time in the computations. □

Lemma 6.2. *In the optimal multi-installment divisible job processing on a star with unlimited memory sizes there should be no idle times between the transmissions of consecutive parts of the load to the processors.*

Proof. We will prove constructively that any schedule with an idle time in the communications before the last stage can be shortened by the removal of such an idle interval. The communication gap can be closed by borrowing some load from the preceding stage to fill the communication gap in the considered stage. This method cannot be applied in the first stage. However the communication gap in the first stage can be closed by shifting some load to the end of the schedule.

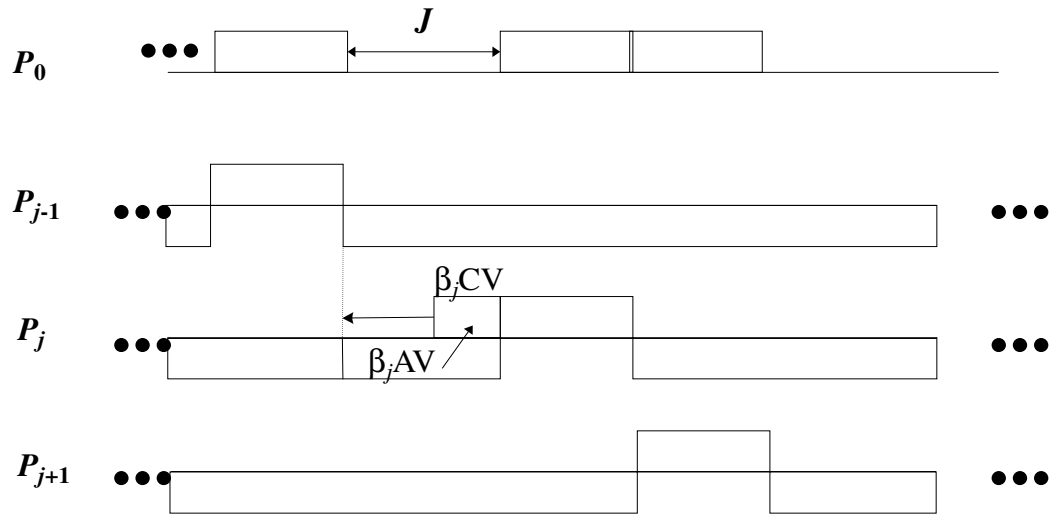


Figure 6.2: Multi-installment processing with communication idle time.

Suppose that in some stage k there is a gap between the communication to P_{j-1} and P_j for $j > m$ (cf. Fig 6.2. If there is an idle interval in computations on P_j immediately before the stage k communications which is sending the load to P_j, then one can shift both the communication and the computations on P_j in stage k to the left, such that there is no idle time either in the computations or in the communications. If an idle intervals remains in the computations then by Lemma 6.1 we know that such an interval can be removed with the benefit of decreasing the schedule length. Therefore we can consider only a gap remaining in communications.

The gap in the communications to P_j can be closed by shifting some load from processor P_j in the preceding stage. This will create a gap in the communications to P_{j+1} in stage k, but also this gap, and the following ones, can be recursively closed by borrowing load from the same processor in the preceding installment. In this way the gaps in the communications can be closed for all the stages except for the first one.

Suppose there is an interval of length J in the first stage when the originator waits before sending the load to processor $1 \leq j \leq m$. This gap can be removed by decreasing the load sent to P_j by $\beta_j VC$. Suppose we shift to the right (i.e.

delay) the communication to P_j and processing the load. Thus, the beginning of the communication to P_j is delayed by $\beta_j V(C+A)$. This would result in an idle interval between the message sent to P_{j-1} and P_j. Thus, also the amount of load sent to P_{j-1} must be decreased by β_{j-1}, such that $\beta_{j-1}VA = \beta_j V(C+A)$. Analogously we have $\beta_i VA = \beta_{i+1}V(C+A)$ for $i = 1, \ldots, j-1$. From this we obtain $\beta_i = \beta_j(1+\frac{C}{A})^{j-i}$. Summing up the load removed in order to close the communication gap we have $\sum_{i=1}^{j}\beta_i = \beta_j \sum_{i=0}^{j-1}(1+\frac{C}{A})^i = \frac{J}{VA}\frac{1-(1+\frac{C}{A})^j}{1-(1+\frac{C}{A})} = \frac{J}{VC}((1+\frac{C}{A})^j - 1)$. Assume we shift all the communication and computations of the current stage to the right. The communications to $P_1, \ldots, P_j$ can be started $L_1 = VC(\beta_1 + \beta_2 + \cdots + \beta_j) + J = J(1+\frac{C}{A})^j$ units of time later. Thus the schedule is made shorter by the same amount of time. On the other hand, $\sum_{i=1}^{j}\beta_i$ units of load must be processed by extending the last stage od the schedule by some time x on each processor. Suppose $\gamma_i, \ldots, \gamma_m$ is the increase of the load on $P_1, \ldots, P_m$ at the end of the schedule. Using equation (6.1) we obtain

$$\begin{aligned}\sum_{i=1}^{m}\gamma_i &= \gamma_1\frac{A+C}{C}(1-(\frac{A}{A+C})^m) + \gamma_1(\frac{A}{A+C})^{m-1} = \\ &= \frac{J}{VC}((1+\frac{C}{A})^j - 1) = \sum_{i=1}^{j}\beta_i.\end{aligned} \tag{6.5}$$

Thus,

$$\gamma_1 = \frac{J((1+\frac{C}{A})^j - 1)}{V(A+C)(1-(\frac{A}{A+C})^m + \frac{CA^{m-1}}{(A+C)^m})}. \tag{6.6}$$

The increase of the schedule length in the last stage is $x = \gamma_1 V(A+C)$ (cf. proof of Lemma 6.1). Hence, schedule got shorter by

$$L_1 - x = J(1+\frac{C}{A})^j - \frac{J((1+\frac{C}{A})^j - 1)}{1-(\frac{A}{A+C})^m + \frac{CA^{m-1}}{(A+C)^m}} = \frac{J(1-(\frac{A}{A+C})^{m-j} + \frac{CA^{m-j-1}}{(A+C)^{m-j}})}{1-(\frac{A}{A+C})^m + \frac{CA^{m-1}}{(A+C)^m}} \tag{6.7}$$

Let us note, that $L_1 - x$ is positive and schedule length can be shortened. Hence, the optimum schedule with and idle interval in the computations on some processor

can be shortened, and thus it was not optimal schedule. Thus, we have a contradiction. We conclude that an optimal schedule should have no idle time in the communications. □

From the above Lemmas we can conclude, that there should be idle times neither in the computation nor in the transmissions. This observation allows to formulate a set of recursive equations to find the sizes of the load distributed to processors in the optimal schedule.

6.2 The maximum gain from multi-installment processing

In this section the maximum possible reduction of the schedule length obtained by multi-installment processing is calculated. Let us note that when the load is sent only once to each processor, the processors do not start computing at the time $t = 0$, but remain idle waiting for transmission. Particularly, the last processor is busy only during a short time at the end of the processing. If the load is divided into many small pieces, the processors would receive the first portion of data sooner and therefore their idle time would be shorter. It is possible to assess what the maximum possible gain is if dividing the load into many pieces. The following reasoning concerns processors without front-ends. A similar reasoning for the processors with front-ends can be applied. In the case of infinite number of pieces, processor P_k starts processing at the time $t_k = kS$ (where k is the sequential number of the processor). If each processor receives data only once, the schedule length is $C^1_{max} = CV + mS + Aa_0V$ (cf. Fig 2.7). We can calculate α_i values from the set of recursive equations:

$$\alpha_{i+1} = (\frac{C}{A} + 1)\alpha_i + \frac{S}{AV} \tag{6.8}$$

$$\alpha_1 = \alpha_0 \tag{6.9}$$

$$\sum_{i=1}^{m} \alpha_i = 1 \tag{6.10}$$

We are interested in calculating the maximum possible gain from multi-installment processing. Therefore, we assume an ideal case $S = 0, n = \infty$ in which startups are eliminated and pieces are as small as desired. From equations (6.8)-(6.10) we can calculate the value of α_1.

$$1 = \sum_{i=0}^{m} \alpha_i = \alpha_0 + \alpha_1 + (\frac{C}{A} + 1)\alpha_1 + (\frac{C}{A} + 1)^2\alpha_1 + \ldots = \alpha_1(2 + \sum_{i=1}^{m-1}(\frac{1}{A'} + 1)^i) \tag{6.11}$$

$$\alpha_1 = \frac{1}{A'(\frac{A'+1}{A'})^m - A' + 1} \qquad \text{where } A' = \frac{A}{C} \tag{6.12}$$

Now let us return to the multi-installment processing. The smaller the pieces are, the sooner all processors start processing. But if the number of communication grows the time spent in communications startups also grows. If the load is transmitted in many pieces the processing time cannot be less than CV. Two cases should be taken into account:

1. $AV \leq (m-1)CV$. In this case the processing time is equal to CV. It results from the fact, that in every moment during the interval of length CV, one of the processors is busy with transmission and the remaining $m-1$ processors can compute. Because $AV \leq (m-1)CV$ it is possible to process all data in time $CV + Aa_1$. The last pieces a_1 and a_0 must be computed after the whole transmission is done, but their size is almost 0.

2. $AV \geq (m-1)CV$. In this case, after the transmission of the last part of the load, all processors still have some load to process. The size of the load processed in the time CV is $\frac{(m-1)CV}{A}$. The remaining $V - \frac{(m-1)CV}{A}$ units of the load can be processed in parallel by m processors. Therefore, the time needed for this is $\frac{AV-(m-1)CV}{m}$.

Let $G = \frac{C^{\infty}_{max}}{C^{1}_{max}}$ be the ratio of the schedule length C^{∞}_{max} for $n = \infty$ and C^{1}_{max} for $n = 1$. The value of G shows possible gain from using multi-installment processing. Smaller values of G denote better gain, which means that the time needed for processing the load is shorter. From the above discussion we have the following formulae expressing G:

$$G = \frac{CV}{CV + A\alpha_1 V} = \frac{1}{1 + A'\alpha_1} \qquad \text{for } A' \leq m-1 \tag{6.13}$$

$$G = \frac{CV + \frac{AV-(m-1)CV}{m}}{CV + A\alpha_1 V} = \frac{1 + \frac{A'-(m-1)}{m}}{1 + A'\alpha_1} \qquad \text{for } A' \geq m-1 \tag{6.14}$$

After inserting equation (6.12) into equations (6.13)-(6.14) we obtain:

$$G = \frac{A'(\frac{A'+1}{A'})^m - A' + 1}{A'(\frac{A'+1}{A'})^m + 1} \qquad \text{for } A \leq m-1 \tag{6.15}$$

$$G = \frac{(A'(\frac{A'+1}{A'})^m - A' + 1)(A'+1)}{m(A'(\frac{A'+1}{A'})^m + 1)} \qquad \text{for } A \geq m-1 \tag{6.16}$$

Let us note that the best gain can be achieved if the value of G is the minimum possible. The function defined by equation (6.15) is monotonicly decreasing when A' takes values from interval 0 to $m-1$. For $A' \geq m-1$ the function defined by equation (6.16) is monotonicly increasing. Thus we can conclude that the G is minimum when $A' = m-1$. Hence we have:

$$G = \frac{(m-1)(\frac{m}{m-1})^m - m + 2}{(m-1)(\frac{m}{m-1})^m + 1} \tag{6.17}$$

The greater m the better gain G can be achieved. To find the best gain that can be achieved we calculate the limit of G when the m tends to infinity. Let us note that $\lim_{m\to\infty}(\frac{m}{m-1})^{m-1} = \lim_{m\to\infty}(1 + \frac{1}{m-1})^{m-1} = e$. Hence:

$$\begin{aligned}\lim_{m\to\infty} G &= \lim_{m\to\infty} \frac{(m-1)(\frac{m}{m-1})^m - m + 2}{(m-1)(\frac{m}{m-1})^m + 1} \\ &= \lim_{m\to\infty} \frac{(\frac{m}{m-1})^{m-1} m - m + 2}{(\frac{m}{m-1})^{m-1} m + 1} \\ &= \lim_{m\to\infty} \frac{(\frac{m}{m-1})^{m-1} - 1 + \frac{2}{m}}{(\frac{m}{m-1})^{m-1} + \frac{1}{m}} \\ &= \frac{e-1}{e} \approx 0.632\end{aligned} \tag{6.18}$$

This means that multi-installment processing can shorten the time needed for processing data to 63.2% of the time used for single-installment processing.

Corollary 6.3. *The minimum schedule length obtained by using multi-installment divisible load processing is* $C^{\infty}_{max} = \frac{e-1}{e} C^{1}_{max}$.

6.3 Processors without front-end

In this section we analyze processing elements without communication front-end using the results of Lemma 6.1 and Lemma 6.2. It is possible to formulate recursive equations determining the values of α_i variables (cf. Fig 2.7).

$$A\alpha_i V = (C + A)\alpha_{i-1} V + S \qquad \text{for } i = 2, \ldots, m \tag{6.19}$$

$$A\alpha_i V = CV \sum_{k=1}^{m-1} \alpha_{i-k} + (m-1)S \qquad \text{for } i = m+1, \ldots, mn \tag{6.20}$$

$$\sum_{i=1}^{mn} \alpha_i = 1 \tag{6.21}$$

$$\alpha_0 = \alpha_1 \tag{6.22}$$

Let us note that equation (6.20) indirectly prevents the accumulation of the load on processors. Equation (6.22) shows that the size of the piece processed by the originator is equal to the size of the last piece sent to processor P_m. All processors

stop processing at the same time. Therefore, we can derive the following equation for the schedule length as the total communication and computing time of the originator:

$$C^n_{max} = C(1-\alpha_0)V + mnS + A\alpha_0 V \tag{6.23}$$

From the recursive equations (6.19) and (6.20) we can find values α_i as the functions of α_1. After expressing values α_i as linear functions of α_1 or α_0 we can find α_1 and α_0 from the equation (6.21), and use it to calculate time C^n_{max} from equation (6.23). There are six parameters A, C, S, V, m, n in the above equations. It is hard to analyze the performance of a computer system described by so many parameters. Hence, we reduce the number of independent parameters by dividing equations (6.19), (6.20) and (6.23) by CV. Thus, we replace A by $A' = \frac{A}{C}$, S by $S' = \frac{S}{CV}$ and C^n_{max} by $C'^n_{max} = \frac{C^n_{max}}{CV}$. The new form of the equations follows:

$$A'\alpha_i = (1+A')\alpha_{i-1} + S' \qquad \text{for } i = 2, \ldots, m \tag{6.24}$$

$$A'\alpha_i = \sum_{k=1}^{m-1} \alpha_{i-k} + (m-1)S' \qquad \text{for } i = m+1, \ldots, mn \tag{6.25}$$

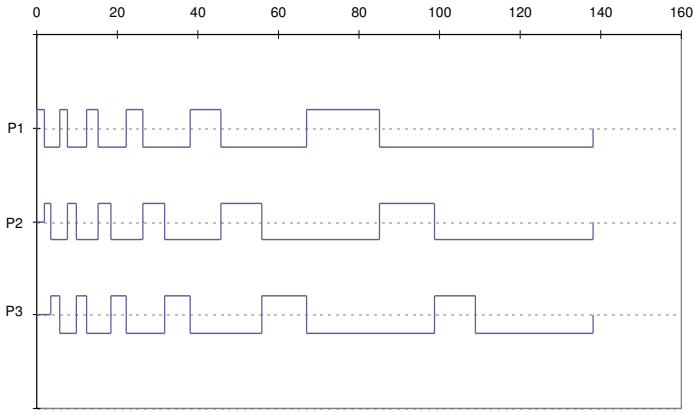
$$\sum_{i=1}^{mn} \alpha_i = 1 \tag{6.26}$$

$$C'^n_{max} = (1-\alpha_0) + mnS' + A'\alpha_0 \tag{6.27}$$

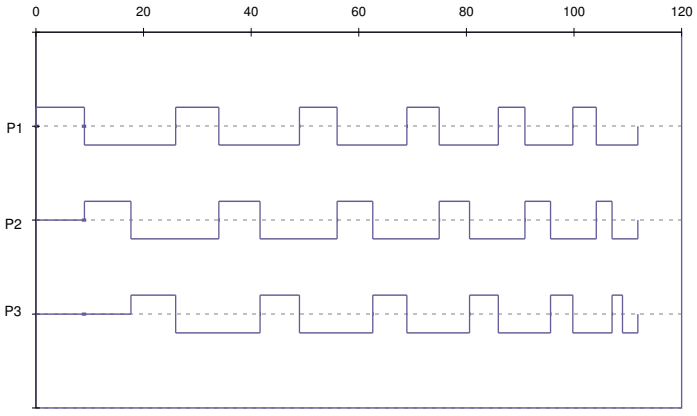
In the following paragraphs we present the results of the simulations determining the relationship between parameters A', S', m, n and the system performance.

An interesting phenomenon has been observed in the systems without the front-ends. If $A/C \approx m-1$ the sizes of the consecutive pieces were almost the same. For $A/C > m-1$ every next piece was larger then the previous one (Fig. 6.3(a)) and for $A/C < m-1$ sizes of the pieces decrease (Fig. 6.3(b)). In Fig. 6.3 the pieces of the line below the dotted lines represent the computing periods and the pieces above the dotted lines are the communications.

Fig. 6.4 presents the optimal number of stages n^* for the given values of A' and



(a) $A/C > m - 1$



(b) $A/C < m - 1$

Figure 6.3: Gantt chart for multi-installment processing without front-ends.

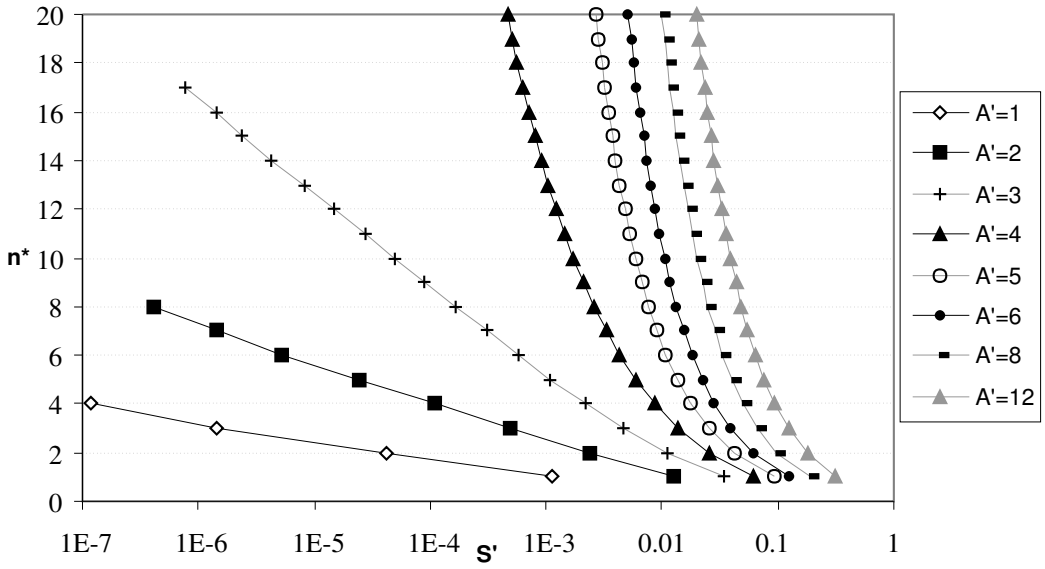
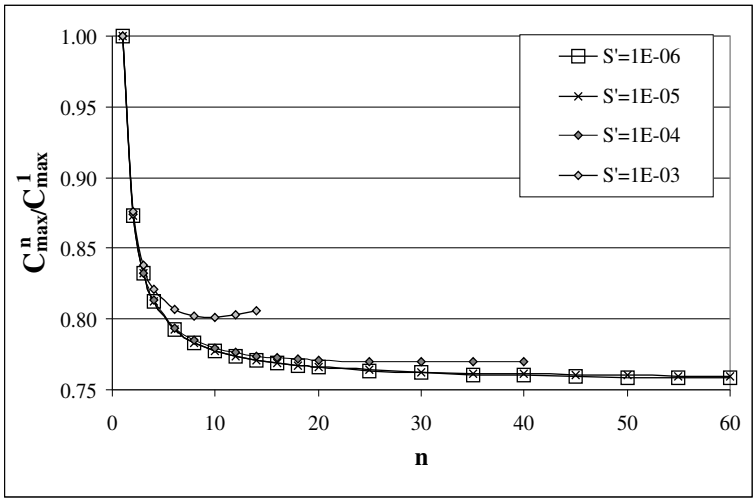


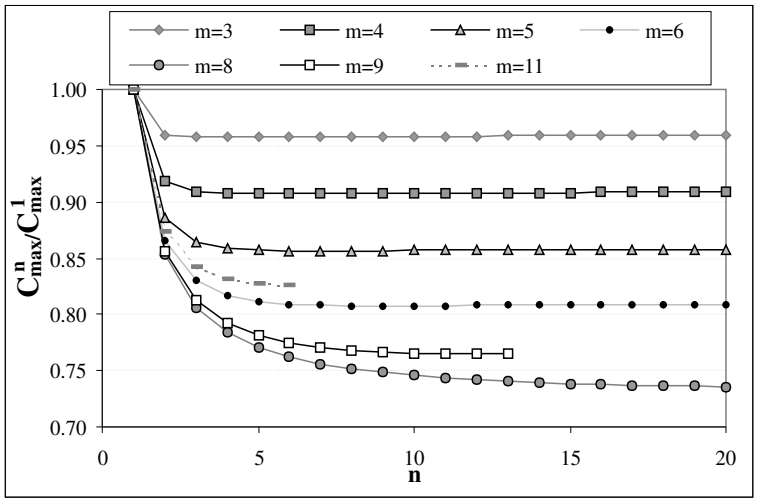
Figure 6.4: Optimal number of installments for $m = 5$ processors.

S' for $m = 5$ processors. For the optimal number n^* of stages the schedule length is the shortest possible. There is a simple iterative method of determining n^*. For the number of pieces less than optimal, the schedule length is greater, and for the number of pieces greater than optimal the set of equations (6.24)-(6.27) is infeasible because some of the load fractions α_i are negative. The dependencies of n^* on S' for other numbers of processors m are very similar. We can observe in Fig. 6.4 that multi-installment divisible job processing is particularly effective for small values of S' and big values of A'.

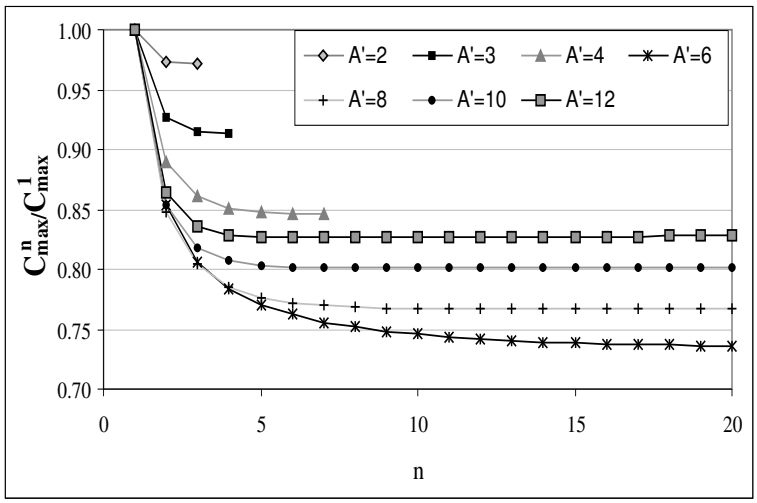
The ratio of the processing time C^n_{max} for n installments to the processing time C^1_{max} for a single installment is shown in the Fig. 6.5. Fig. 6.5(a) depict the results for $m = 6$ processors with $A' = 4$. Fig. 6.5(b) shows the results for processing data with $A' = 6$ and $S' = 1E - 4$. Fig. 6.5(c) presents the results for processing on 8 processors with parameter $S' = 1E - 4$. We can notice that the best values can be achieved when $A' = m - 2$. This empirical rule applies not only to specific values $A' = 6$ and $m = 8$, but to all tested values of A' and m. This means that for the systems with the given A' value we should use $m = A' + 2$ processors to optimize the



(a) $A' = 4, m = 6$



(b) $A' = 6, S' = 1E - 4$



(c) $m = 8, S' = 1E - 4$

Figure 6.5: Gain in processing time from sending data in n pieces.

utilization of the resources. We can also observe that for small S' and large n results are better. It seems reasonable because with smaller S' less time is used for setting up a communication. On the other hand, we cannot choose very large n, because when n is too big, it is impossible to solve the equations (6.24)-(6.27). Fig. 6.6 shows the benefit of multi-installment scheduling with the maximum possible value of n. Fig. 6.6(a) shows the gain for different values of A' and n. Fig. 6.6(b) shows the gain for maximum admissible value of n as a function of A'. In both figures the value of m was equal to $A' + 2$. It can be seen that it is hard to get the processing time ratio less than approximately 0.64. This confirms the results from Section 6.2.

6.4 Processors with front-end

In this section we analyze processing elements with communication front-end. Similarly to the classical DLT also here all processors stop processing at the same time. It is possible to formulate recursive equations determining the sizes of α_i of the load sent to the processors.

$$A\alpha_i V = (C + A)\alpha_{i-1} V + S \qquad \text{for } i = 2, \ldots, m \tag{6.28}$$

$$A\alpha_i V = CV \sum_{k=1}^{m} \alpha_{i-k} + mS \qquad \text{for } i = m+1, \ldots, mn \tag{6.29}$$

$$\sum_{i=0}^{mn} \alpha_i = 1 \tag{6.30}$$

$$A\alpha_0 V = CV(1 - \alpha_0) + mnS + A\alpha_1 V \tag{6.31}$$

Considering activities of the originator, we can derive the following equation for the schedule length:

$$C_{max}^n = A\alpha_0 V = CV(1 - \alpha_0) + mnS + A\alpha_1 V \tag{6.32}$$

Similarly to the procedure applied in the previous section we can divide equations

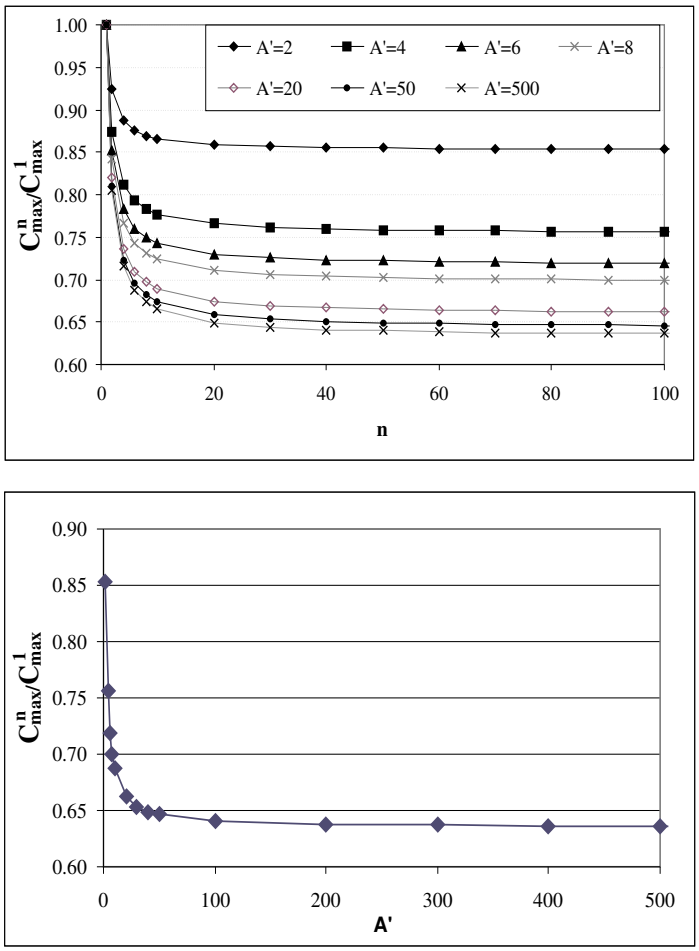


Figure 6.6: Benefit from multi-installment processing with the maximum admissible value of n.

(6.28), (6.29), (6.31) by CV, in order to reduce the number of independent variables and to simplify the further analysis.

$$A'\alpha_i = (1 + A')\alpha_{i-1} + S' \qquad \text{for } i = 2, \ldots, m \tag{6.33}$$

$$A'\alpha_i = \sum_{k=1}^{m} \alpha_{i-k} + mS' \qquad \text{for } i = m+1, \ldots, mn \tag{6.34}$$

$$\sum_{i=0}^{mn} \alpha_i = 1 \tag{6.35}$$

$$C'^{n}_{max} = (1 - \alpha_0) + mnS' + A'\alpha_1 \tag{6.36}$$

We examined the dependence of the system performance on the parameters A', S', n. Fig. 6.7 presents the optimal number n^* of stages for the given values of A', S' and $m = 6$ processors. The number of installment n^* is optimal in the sense of the shortest schedule. Also for the processors with the front-ends there is a simple method of determining n^*. Schedule length decreases with n up to n^*. For the number of stages greater than n^* the set of equations (6.33)-(6.36) is infeasible, because some processor would have $\alpha_i < 0$. We can observe that the multi-installment processing is particularly justified for small values of S' and big values of A'. Let us note, that Fig.6.7 is similar to Fig.6.4. It means that using a different communication equipment has little influence on efficiency here. The dependance of n^* on A', ad S' is very similar also for other number of processors.

Fig. 6.8 shows the ratio $C'^{n}_{max}/C'^{1}_{max}$. Fig.6.8(a) shows the results for $m = 6$ processors with $A' = 4$. Fig. 6.8(b) presents the results for processing data with parameter $A' = 6$ and $S' = 1E-4$. Fig. 6.8(c) depicts the results for processing data using $m = 8$ processors with parameter $S' = 1E-4$. Comparing Fig. 6.5 and Fig. 6.8 we can conclude that using processors with front-ends we can gain only a few percent in the reduction of the schedule length over the processors without front-ends.

It can be observed in Fig 6.8 that the best values are achieved when $A' = m - 1$. This empirical rule applies not only to the specific values $A' = 6$ and $m = 8$, but to all values of A' and m tested. It means that for a system with given A' we should use

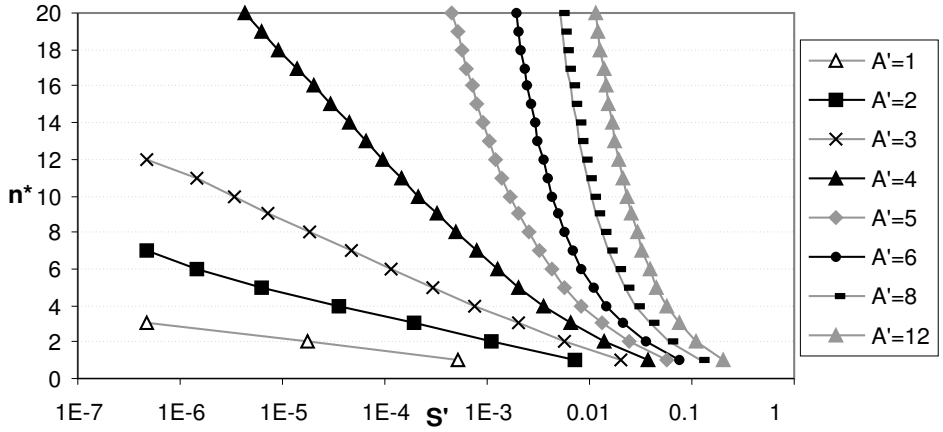
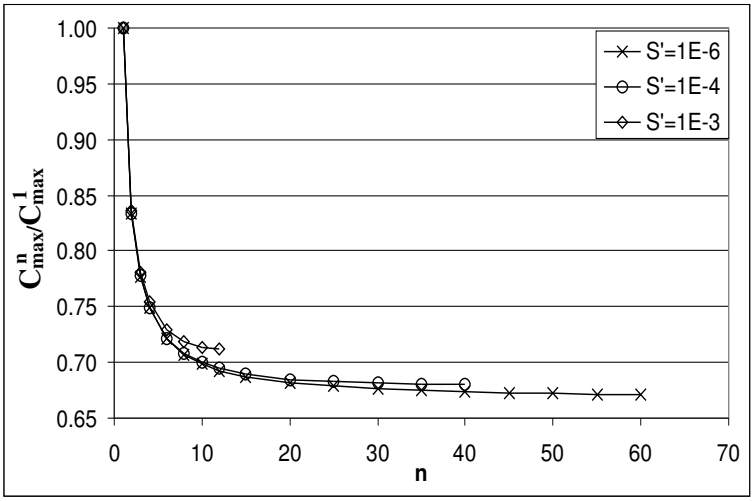


Figure 6.7: Optimal number of installments for 6 processors with front-end

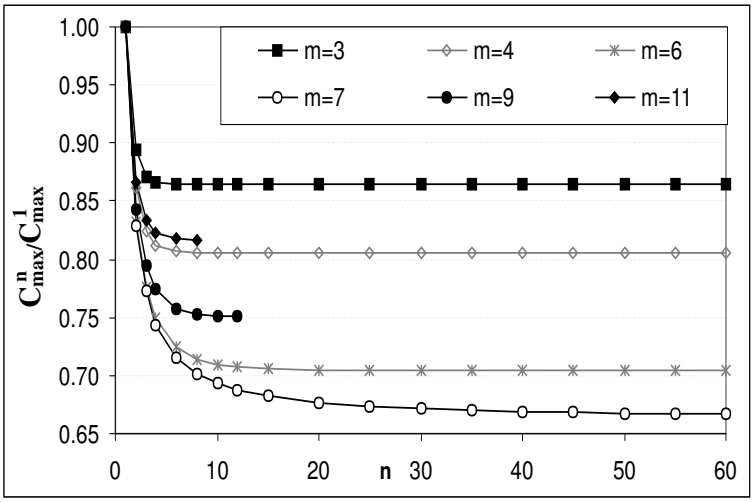
$m = A' + 1$ processors to optimize the gain from multi-installment processing. Let us observe that for small S' and large n the results are better. It is intuitively natural because with smaller S' less time is used for setting up a communication. Yet, we cannot choose very large n, because when n is too large, the solution of the equations (6.33)-(6.36) is infeasible. It is the case when some of α_i are negative. In Fig. 6.9 benefits from multi-installment scheduling with the maximum admissible value of n are presented for different values of A' and n. The value of m is equal to $A' + 1$ in Fig. 6.9. The reduction of processing time is not more that approximately 0.64 C^1_{max}. Thus predictions from Section 6.2 are confirmed.

6.5 Model Comparison

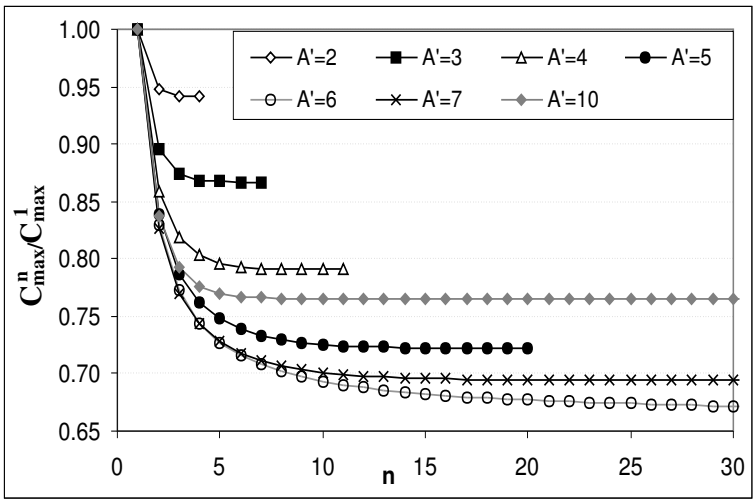
The results of the simulations for processors with and without front-ends are very similar. On both types of processors it is possible to obtain similar performance. The significant difference is the optimal number of processors one should use to achieve the best gain from the multi-installment processing. For processors with a front-end we can save one processor compared to the processors without a front-end. It is the case because the originator is processing data during communication and can be treated as an additional processor. Another difference is that for processors without



(a) $A' = 4, m = 6$



(b) $A' = 6, S' = 1E - 4$



(c) $m = 8, S' = 1E - 4$

Figure 6.8: Gain in processing time from sending data in n pieces.

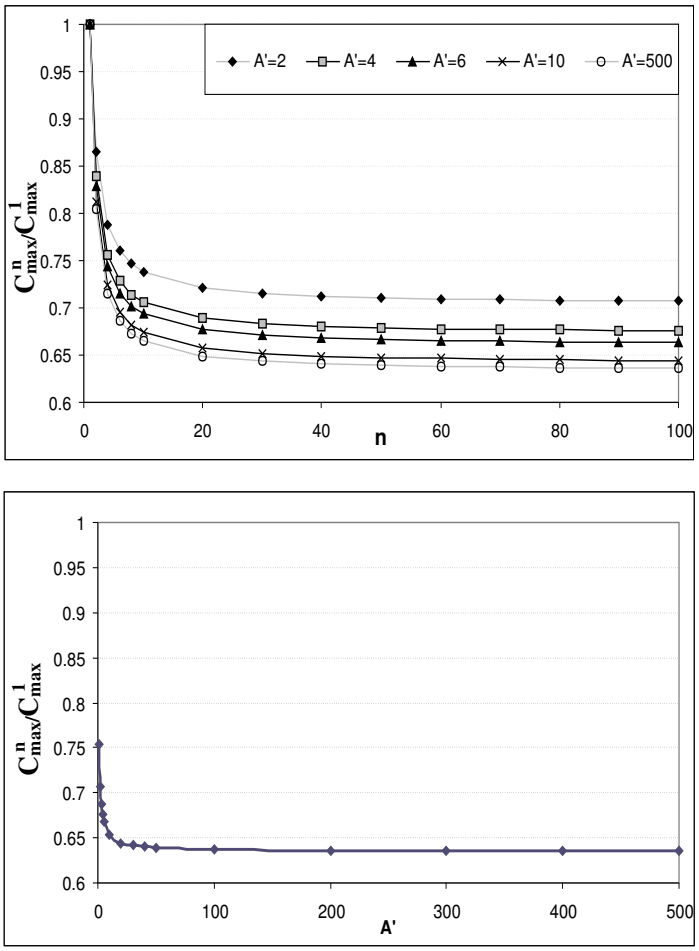


Figure 6.9: Benefit from multi-installment processing with the maximum possible value of n.

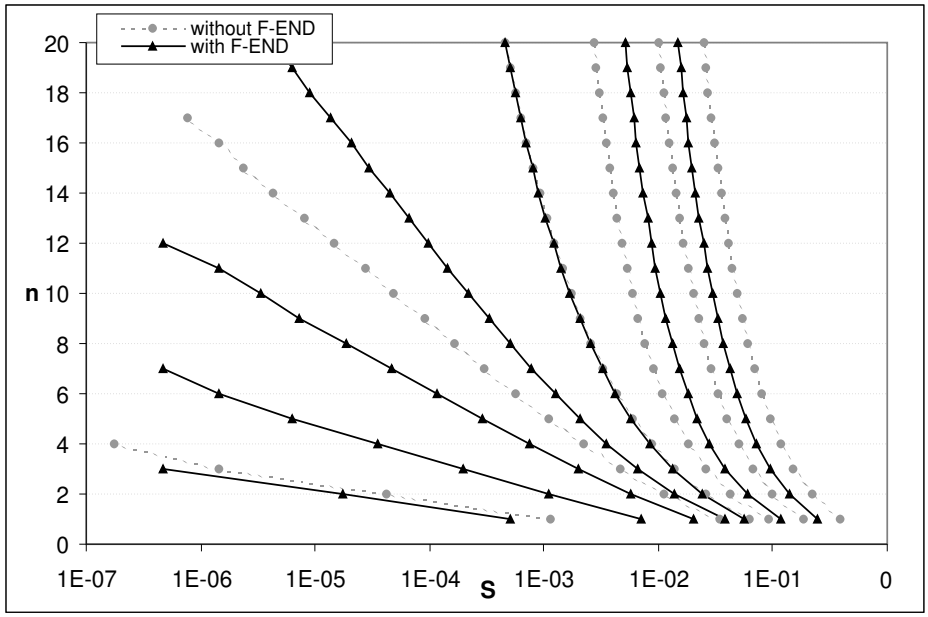


Figure 6.10: Optimal number of pieces for different processor type.s

front-end it is possible to use more pieces n for the same value of S'. The results in Fig. 6.4 and Fig. 6.7 are compared in Fig. 6.10. We can see that the lines are shifted horizontally. Thus, on the system without front-ends it is optimal to use the same number of pieces as on the system with front-ends at S' that is 1.5-2 times larger. For similar values of S' it is optimal to use less stages on the system without front-ends than on the system with front-ends for the same parameters A', m.

On the other hand, it should not be forgotten that by using more processors it is possible to shorten the schedule length, but the gain of multi-installment processing is smaller. Consequently, the processor utilization is also smaller. In other words, it is more efficient to process two jobs, each requiring m processors in parallel, than process the same jobs sequentially using $2m$ processors. Fig. 6.11 shows the schedule length for different values of A' and m. Let us note that more processors can be activates in slow systems, i.e. for big A'.

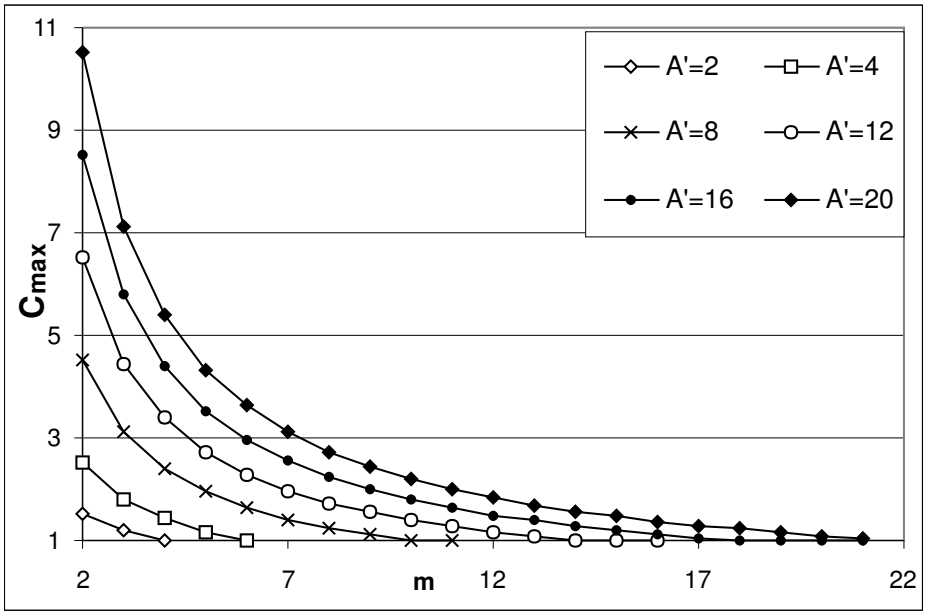


Figure 6.11: Schedule length for different values of A' and m

6.6 Memory utilization in multi-installment processing

In this section we consider memory requirements in multi-installment processing. In the processing type presented in the previous section all data chunks could have different sizes. The size of the largest part of the load determined the minimal memory buffer for the processors. On the other hand it is also possible to divide the volume of data into parts of equal sizes. In this type of processing the idle times in the transmission or in the processing can appear. Therefore, the schedule length will be greater. Since all chunks have the same size the memory requirements can be smaller than in the previous case. This method of processing is also simpler to implement than the former one. We can formulate the equation describing this model: Let T_s denote the time between transmissions of consecutive data chunks to the same processor, and δ denote sizes of the load sent to the processors $P_1, \ldots, P_n$

$$T_s = \max\{S + \delta(C + A), m(S + C\delta)\} \tag{6.37}$$

$$C_{max} = (n-1) * T_s + m(S + C\delta) + A\delta \tag{6.38}$$

If $T_s = S + \delta(C + A)$ processing step is computation bound and an idle time in the communication appears. If $T_s = m(S + C\delta)$ processing step is communication bound and an idle time in the computation appears. In both cases, after transmitting all data, the originator can process the load of size δ. Hence:

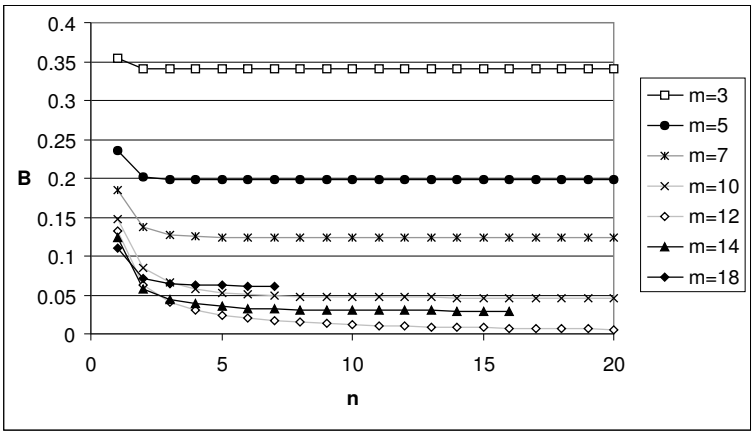
$$\alpha_i = \delta = V/(mn + 1) \leq B \quad \text{for } i = 0, \ldots, m \tag{6.39}$$

On the other hand if the size B of the memory buffers is given it is possible to find the minimal number of installments required, so that the load chunks fit into the memory buffers.

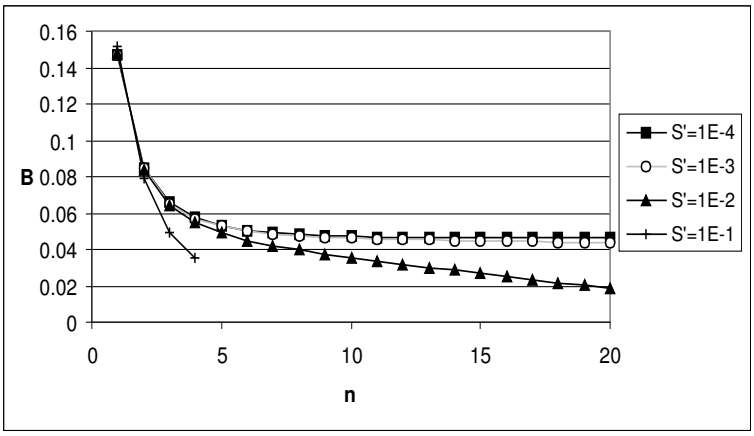
$$n \geq \lceil (V - B)/(Bm) \rceil \tag{6.40}$$

The minimal memory buffer sizes for processing with variable sizes of the load (as discussed in the preceding sections) are shown in Fig 6.12 and the minimal memory buffer required sizes for processing with fixed sizes of the load chunks are shown in Fig 6.13. In both cases the buffer sizes are expressed as parts of the total volume size.

We can notice that also for multi-installment processing with different sizes of data pieces the best values can be achieved when $A' = m - 2$ (cf. Fig. 6.12). This means that for the systems with given A' value we should use $m = A' + 2$ processors in order to minimize memory requirements. For multi-installment processing with equal sizes of data chunks we can observe that the schedule length depends only on the number of processors and processing steps (cf. Fig. 6.13). It is the obvious result of equation (6.39). For this kind of processing memory buffers can be smaller than for the processing with different piece sizes. But because of equal pieces we can expect that processing time will be longer than in the first method. Schedule length reduction for multi-installment processing with equal sizes of the load is presented in Fig. 6.14. The analysis was done for $A' = 10$ and $S' = 1E - 5$. We can observe that for some values of the parameters the schedule length is even longer then for single-installment processing. It is because the computation time dominates the schedule length and severe load imbalance appears..

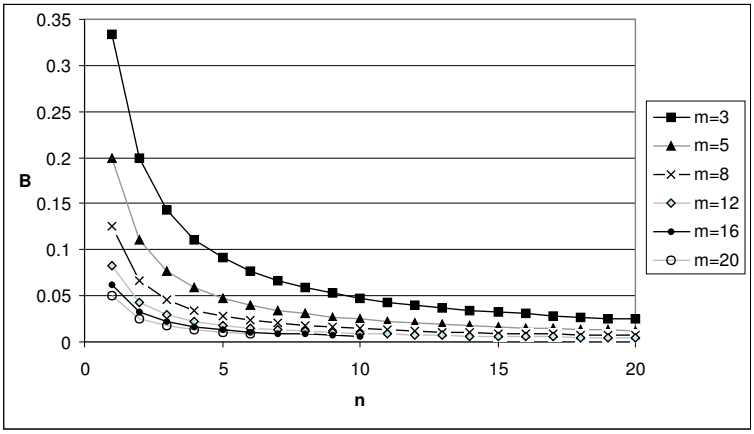


(a) $A' = 10, S' = 1E - 5$

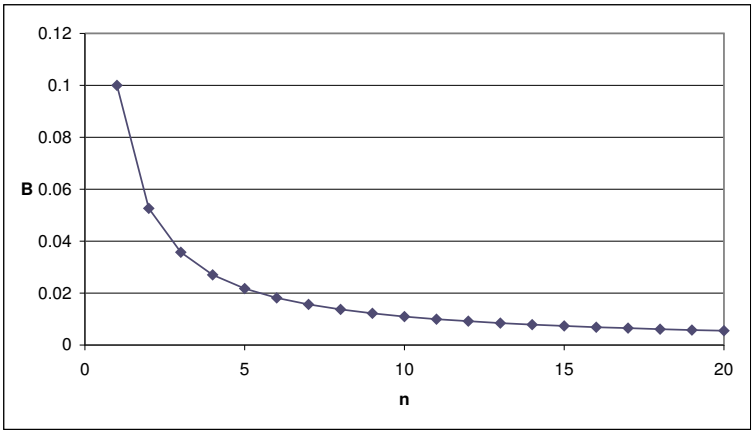


(b) $m = 10, A' = 10$

Figure 6.12: Decrease of memory requirements for multi-installment processing with variable sizes of the load



(a) $A' = 10, S' = 1E - 5$



(b) $m = 10, A' = 10$

Figure 6.13: Decrease of memory requirements for multi-installment processing with equal sizes of the load

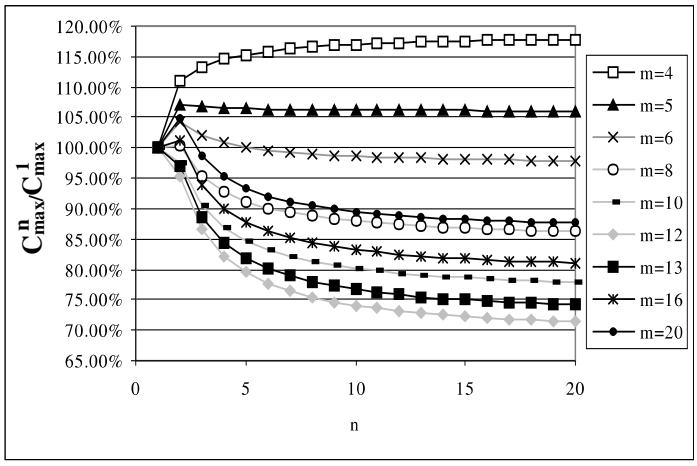


Figure 6.14: Reduction in schedule length for multi-installment processing with equal chunks.

The comparison of the processing time gain of the two considered processing types is presented in Fig. 6.15. On the vertical axis the absolute time is shown here instead of the time gain as in the previous figures. The multi-installment processing with equal sizes of the pieces is always worse than the other method. We can observe that for larger number of pieces the difference between both methods decreases. The difference is about 15% for small number of pieces and about 5% for larger number of pieces.

The comparison of the memory requirements of both processing types is presented in Fig. 6.16. The memory requirement for the processing with equal sizes of the load is much smaller. The smallest difference $(20 - 40\%)$ can be observed for $A' = m - 2$. This is because while processing with variable sizes of pieces the sizes of pieces are roughly similar (cf. Fig. 6.3. But for other values of A' and m processing with equal sizes of pieces requires even several times less memory buffers and this difference increases with the increasing number of pieces.

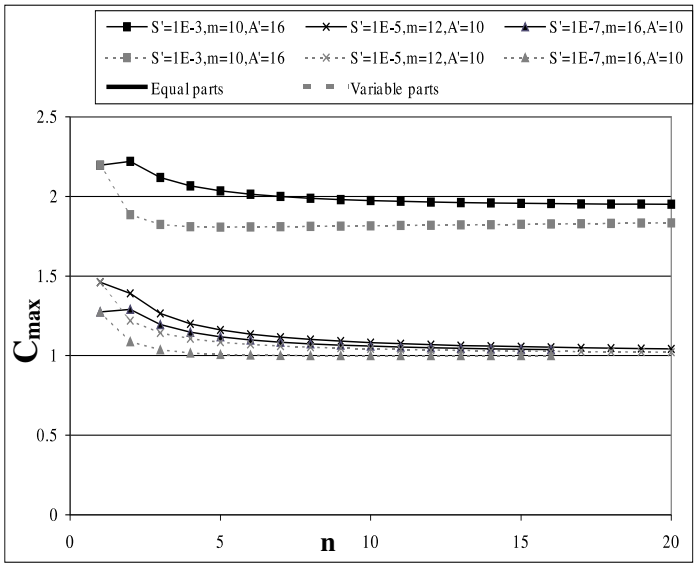


Figure 6.15: Schedule length for two multi-installment processing types

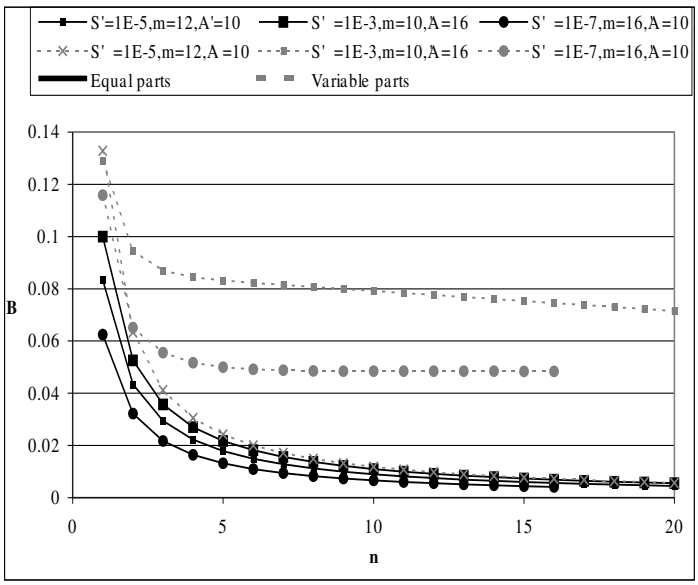


Figure 6.16: Minimal memory buffer sizes for two multi-installment processing types.

6.7 Conclusions

The above results show that it is advantageous to use multi-installment divisible job processing, but the gain on the schedule length C_{max} is limited. It was demonstrated that the processing time can be reduced approximately by one third. The number of optimal processing stages n^*, depends on the values A' and S'. If S' is large then the optimal processing requires fewer stages. The results show that the value of A' determines the number of processors we should use to have minimum $\frac{C^n_{max}}{C^1_{max}}$ and better memory utilization. On the other hand the minimum $\frac{C^n_{max}}{C^1_{max}}$ is not equivalent to minimum C^n_{max}. Multi-installment processing allows also to use processors with smaller memory buffers. If the memory buffers are important it is recommended to divide data into equal pieces, but in such a case there is no gain on processing time for certain values of parameters. From Fig. 6.6 and Fig. 6.9 we can conclude that using a large number of stages can reduce the schedule length only slightly. It is reasonable to divide the load only to a few installment (for example no more than 10) because for a larger number of pieces the benefit is small. Moreover, in the real computing systems, there can be problems with synchronization between the processors communications, a small temporary variation of parameters can cause a significant delay in schedule.

Chapter 7

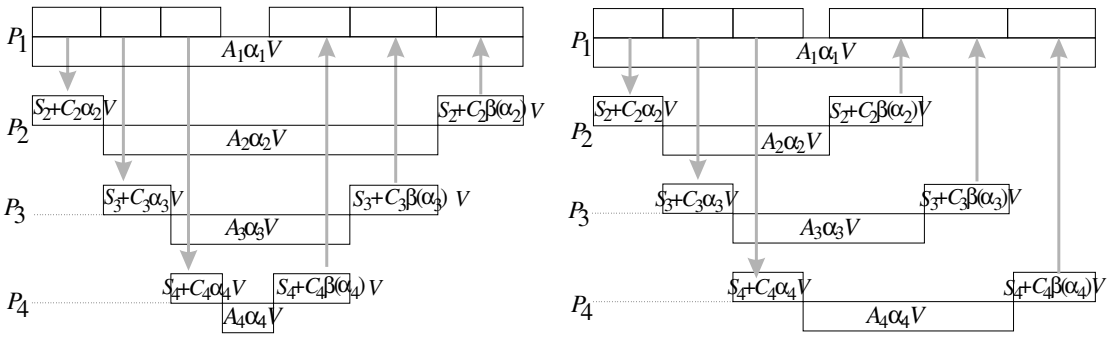
Practice of Divisible Job Processing

In this chapter we present result of practical verification of the divisible load theory predictions.

7.1 Method of experimenting

In this chapter we presents a series of experiment in parallel processing conducted on various cluster of workstation platforms. During the experiments real application were run. Because the results were collected two different modes of returning results can be distinguished. Amount of the returned results is expressed as a function of the size of received data $\beta = f(\alpha)$. Processors can return results in the same order they received data (FIFO case) or in the reverse order (LIFO case). The Gantt charts for both types of communication are presented in Fig. 7.1. All experiments employed single-installment processing.

The goal is to distribute computations, i.e. find α_i, such that the duration of all communications and computations is minimal. Let us observe (cf. Fig. 7.1(a)) that in the LIFO case processing on the processor activated earlier lasts as long as sending to the next processor, computing on it and returning the results. Using this observation we can formulate a set of linear equations from which distribution of the load can be found:



(a) LIFO case (b) FIFO case

Figure 7.1: Communications and computations in star.

$$\alpha_i A_i = 2S_{i+1} + C_{i+1}(\alpha_{i+1} + \beta(\alpha_{i+1})) + A_{i+1}\alpha_{i+1} \qquad i = 0, \ldots, m-1 \tag{7.1}$$

$$V = \sum_{i=0}^{m} \alpha_i \tag{7.2}$$

P_0 denotes the originator. In the FIFO case (cf. Fig. 7.1(b)) the time of processing on P_i and returning results the from processor P_i is equal to the time of sending to P_{i+1} and processing on P_{i+1}. Hence, distribution of the work can be calculated from equations:

$$\alpha_i A_i + S_i + \beta(\alpha_i)C_i = S_{i+1} + \alpha_{i+1}(C_{i+1} + A_{i+1}) \qquad i = 1, \ldots, m-1 \tag{7.3}$$

$$\alpha_0 A_0 = \sum_{i=1}^{m}(S_i + \alpha_i C_i) + \alpha_m A_m + S_m + C_m\beta(\alpha_m) \tag{7.4}$$

$$V = \sum_{i=0}^{m} \alpha_i \tag{7.5}$$

Due to the specific structure the above two equation systems can be solved in $O(m)$ time. However, they may have no feasible solution (because some $\alpha_i < 0$) when volume V is too small and not all m processors are able to take part in the computation. In this case less processors should be used.

7.2 Test applications

Search for a pattern

The problem consists in verifying whether some given sequence X of characters contains substring x. If it is the case the position of the first character in X matching x must be returned as a result. Having calculated quantity α_i of data the originator sends to processor P_i amount of $\alpha_i V + \texttt{strlen}(x) - 1$ bytes from the sequence X starting at position $\sum_{j=1}^{i-1} \alpha_j + 1$. The chunks overlap in order to avoid cutting substring x placed across the border of two different chunks. As the files for the tests were known the amount of returned results was also known. $\beta(x) \approx 0.005x$ which is typical of search in databases holding e.g. surnames and personal data. This application was run also with different numbers of patterns. The idea was to check accuracy of divisible job processing for different A/C ratios.

Compression

In this application originator sends parts of a file to the processors. The part sent to processor P_i has size α_i. Each of the processors compresses the obtained data using LZW [68, 70] compression algorithm. The resulting compressed strings are returned to the originator and appended to one output file. The original file can be obtained by decompressing each part in turn. The achieved compression ratio determines the amount of the returned results. It was measured that $\beta(x) = 0.55x$. The compression ratio and speed depend on the contents and size of the input. In order to eliminate (or at least minimize) this dependence only parts of at least 10kB were sent to the processors for remote compression.

Join

Join is a fundamental operation in relational databases. Suppose there are two databases: X e.g. with a list of supplier identifiers, names, addresses etc., and Y with a list of products with names, prices, etc. and supplier identifier. The result of

join operation on X and Y should be one file with a list of suppliers (names, addresses, etc.) and the products the respective supplier provides. The join algorithm can be understood as calculation of cartesian product $X \times Y$ of the two initial databases. $X \times Y$ can be viewed as a 2-dimensional array in which one row corresponds to one record x_j from file X and one column corresponds to one record y_k from database Y. On the intersection of row x_j and column y_k pair (x_j, y_k) is created. (x_j, y_k) is transferred to the output file only if the fields of the supplier identifier match. In our implementation of distributed join, one of the databases (say X) was transmitted to all processors first. Then, the second database (Y) was cut into parts Y_i according to the calculated volumes α_i, and sent to processors $P_i (i = 1, \ldots, m)$. Each of the processors calculated join on X and Y_i and the results were returned to the originator. Databases X and Y were artificially and randomly generated, therefore the amount of results was known. β expressed the ratio of the amount of results and database Y size.

Graph coloring and genetic search

Consider graph $G(V, E)$, where V is a set of vertices, and $E = \{\{v_i, v_j\} : v_i, v_j \in V\}$ is a set of edges. Node coloring problem consists in assigning colors to the nodes such that no two adjacent nodes v_i, v_j have the same color. More precisely, node coloring is a mapping $f : V \to \{1, \ldots, k\}$, where $\{v_i, v_j\} \in E \Rightarrow f(v_i) \neq f(v_j)$. Find minimum k, i.e. chromatic number χ_G.

Determining graph chromatic number is a hard combinatorial problem, therefore heuristic algorithms are suitable for solving it approximately. Genetic search [55] is one of the metaheuristics applicable with this respect. In our implementation of the genetic search each solution is a gene represented by a string of colors assigned to the consecutive nodes. Good solutions from the initial population are combined using genetic operators to obtain a new population. The measure of solution quality is called fitness function which in our case was the number of the colors used plus the number of infeasibly colored nodes. Two genetic operators were used to obtain new 'individuals':

crossover and mutation. Crossover is a binary operator exchanging tails of the strings in two genes starting at a randomly selected place. The effect of the crossover and selecting good solutions to produce offsprings is that certain properties of the initial solutions can be implicitly identified and combined. Mutation operator makes random changes in the individuals and diversifies the population. Solutions were selected to produce offsprings with probability increasing proportionally to decreasing of the fitness function (note that we have minimization). Originator generated the initial population of 1000 random solutions (genes). This population was distributed among the processors according to the calculated values of α_i's. Each processor created a fixed number of new generations and returned final population to the originator. Thus, $\beta(x) = x$. The gene with feasible coloring and the smallest number of colors used was selected as a final solution.

7.3 The results

In this section we outline results obtained in the experiments (cf. also [39, 49, 58]). We experimented on several different hardware and software platforms. Due to time and workforce limitations not all applications were performed on every considered platform. In Table 7.1 we summarize which application was tested on which platform. Abbreviation ded. stands for dedicated network segment interconnection, and pub. for public network available for other traffic in the experiments time.

The main goal of the experiments was to apply divisible task model in practice and to verify correctness of its predictions. The verification was done by comparing the real and the predicted execution times of some application when data is distributed in chunks of sizes (α_i's) calculated from equations (7.1)-(7.5). To formulate any of the above equations we needed data, i.e. parameters A_j, C_j, S_j for $j = 1, \ldots, m$. Therefore, we had to measure these parameters first. The communication parameters were measured by a ping-pong test. Originator sent to a processor some amount of data. The processor immediately returned these data. A symmetry of the communication

Table 7.1: Platforms vs applications

application→ platform	search for a pattern	com-pression	join	coloring
A: heterogeneous Sun workstations: PVM, ded. Eth	yes			
B: heterogeneous PCs: Linux, PVM, pub. Eth.	yes	yes		
C: nodes of IBM SP2, PVM, ded. HPS		yes		
D: homogeneous PCs: WinNT, MPI, ded. Eth.	yes	yes	yes	
E: heterogeneous PCs: Win98, Java, pub. Eth.				yes
F: homogeneous PCs: Linux, Java, ded. Eth.				yes
G: heterogeneous supercomputers (Cray, SGI): PVM, pub. FDDI	yes			

links was assumed and half of the total bidirectional communication time was taken as the unidirectional communication time. The time of the communication and the amount of data were stored. After collecting a number of such pairs (for various sizes of the message), parameters S_j, C_j were calculated using linear regression. Processing rate A_j was measured as an average of the ratios of the computation time and the quantity of data processed. The method of obtaining $\beta(x)$ has been explained in the previous section. The measured communication parameters are presented in Table 7.2. Standard deviations are reported after the $\pm$ sign.

Table 7.2 requires some comment and explanation. Firstly, these numbers may differ from system to system and from implementation to implementation. Thus, they should be understood rather as indicators than the ultimate truth about communication performance. The values of parameters can significantly depend on the implemented method of communication. Therefore they cannot be taken from hardware specification but should be found using the same method of communication as implemented in the application. The measurements were taken on unloaded computers (no other user applications were running). The values represent one pair of

Table 7.2: Typical values of communication parameters

platform	$C_j[\mu s/B]$	$S_j[\mu s]$
A: various Sun workstations	70.7±0.3	636000±86000
B: various PCs 1	7031±13	2861±9312
various PCs 2	185±20	348±206
C: IBM SP2	68.6±0.1	205±144
D: homogeneous PCs	1.04±0.13	6200±7200
E: various PCs	111±4	116000±10000
F: homogeneous PCs	6±240	8770±240
G: various supercomputers		
Cray SV1	368±3	4761±1066
Cray T3E 900	233±12	10880±4105
Origin 3000	1283±12	13600±4175

communicating computers.

In Table 7.3 examples of typical processing rates (A_j) are given. Note, that these values not only depend on the raw speed of the hardware or the operating system, but also on the application, its implementation, and run-time environment. All results refer to a single computer. Though values A_j are often greater than C_j it does not mean that parallel processing is less efficient than sequential execution because in most of such cases a single computer is not able to hold equivalent quantity of data as distributed computers and process it with the same speed as for small amounts of data (we discussed it in Section 4).

The contents of the following diagrams is organized as follows. Difference between the expected execution time and the measurement divided by the expected execution time (i.e. relative error) is presented on the vertical axis. The horizontal axis shows the size of the problem. In Fig. 7.2 results of the "search for a pattern" application on platform D are shown. In all cases real running time was longer than the expectation. For platform A the results were similar. The difference is stable and around 35% in LIFO case. In the FIFO case the difference has bigger variation, and grows slightly with V from approx. 25% to 30%. In Fig. 7.3 results of the "compression" application on platform C are shown. Real running time was longer than the expectation. The

Table 7.3: Examples of processing rates (A_j)

platform	application	$A_j[\mu s/B]$
A: Sun SLC, PVM	search for a pattern	6.99±0.03
B1: various PC Linux, PVM	compression	1500±20
B2: various PC Linux, PVM	compression	523±466
C: IBM SP2, PVM	compression	650±60
D: homogeneous PCs WinNT, MPI	search for a pattern	0.838±0.007
D: homogeneous PCs WinNT, MPI	join	1176±6
E: various PCs Win95, Java	coloring	25±76
F: homogeneous PCs Linux, Java	coloring	26±2
G: supercomputers Cray SV1 Cray T3E 900 Origin 3000	search for a pattern	1114±12 139±4 143±2

LIFO/FIFO orders of returning results have similar behavior as in the previous application. LIFO case is more stable and real execution times oscillates around 10.5%. In FIFO case difference is growing with the size of the problem from approx. 6% to approx. 13%. For the same application on platform D relative error was decreasing from approx. 55% to approx. 7% with V increasing.

In Fig. 7.4 relative error for "join" application on platform D are displayed. In both LIFO and FIFO cases the difference decreases from approx. 40% to less than 0.5%. Intuitively, it seems reasonable that there should be a good coincidence between the expectation and the measurement for big values of V, because processing and communication times are long and transient (or "noisy") effects are compensated for. In Fig. 7.5 relative difference between the model and experiment for "coloring" application on platform F is shown. As it can be seen with growing volume V the

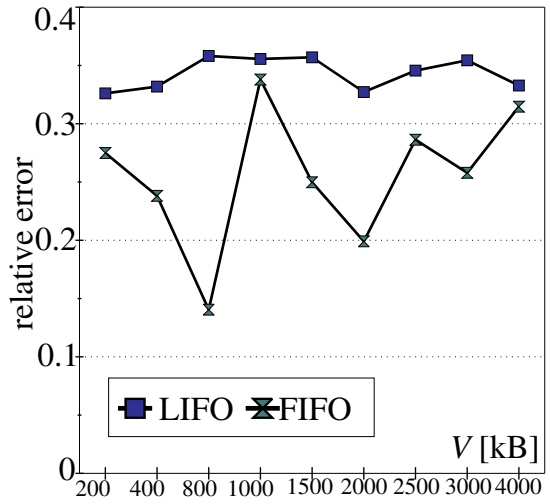


Figure 7.2: Difference between model and measurement on platform D in "search for a pattern" application.

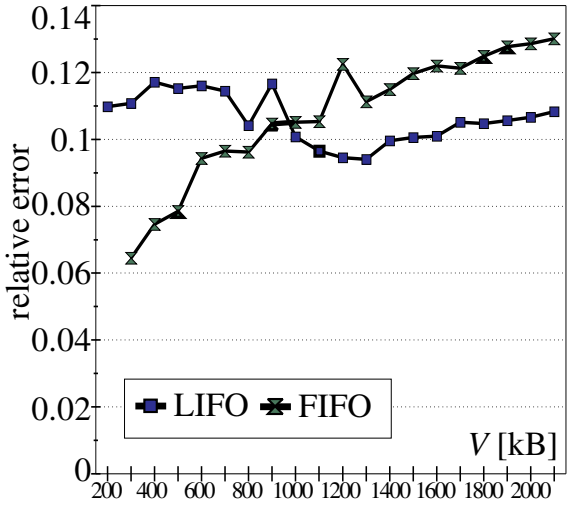


Figure 7.3: Difference between model and measurement on platform C in "compression" application.

relative error decreases from approx. 30% to less than 5% and then increases to approx. 30%. Real execution time was longer up to 30kB, and from 40kB on it was shorter than the expectation. As these results differ from the previous ones further critical verification is needed.

In Fig. 7.6 and Fig. 7.6 relative errors for search for pattern application are presented. In these experiments not only the size of the load was changing but also the number of patterns was changing which resulted in different A/C ratio. In Fig. 7.6 results for platform G are shown. Relative errors grow up to 13% with growing volume V and with the increasing number of patterns. This experiments were run on "live" supercomputer system with dynamically changing load so predicted computation time could not be accurate. Therefore when the computation dominates over communication (for larger number of patterns) the relative error grows but also stabilizes. Negative errors for smaller data sizes are results of rounding error in measurement of very short amounts of time. In Fig. 7.7 results for the same application for platform B2 are shown. This time computers and the network were dedicated to the experiment. We can notice than the size of the data volume have no influence

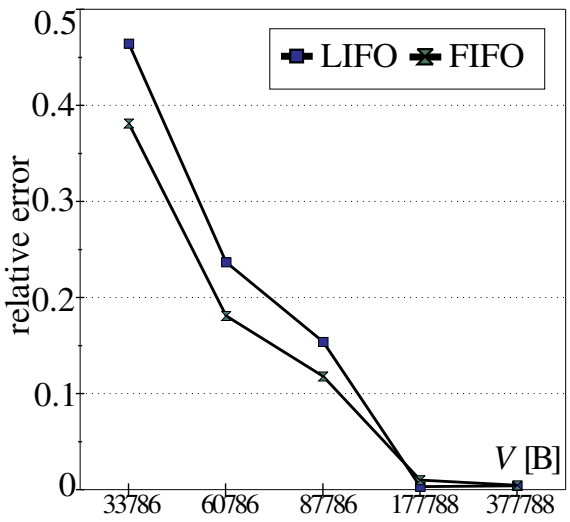


Figure 7.4: Difference between model and measurement on platform D in "join" application.

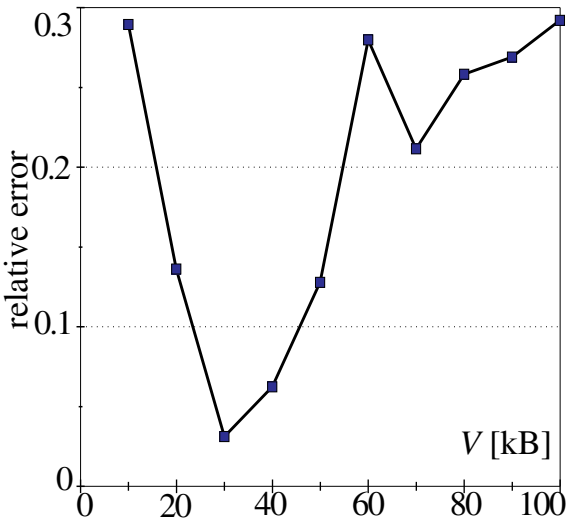


Figure 7.5: Difference between model and measurement on platform F in "coloring" application LIFO case.

on the relative error which is less than 15%. For small A/C ratio (few patterns) the processing times were very short so rounding errors could have important influence on the relative error. Negative error for 35 and 40 patterns are more difficult to explain. Probably it is a result of not accurate measurement of processing ratio A for this cases. Even in dedicated environment some CPU time can be used in unpredictable way by the operating system to run some maintenance processes, daemons etc.

Results for another kind of experiments are shown in Fig. 7.8 and Fig. 7.8. During these experiments different communication methods were used to test the influence of communication middleware on processing time and relative error. The tested application was searching for patterns and communication middleware were MPI, PVM, shmem library and vendor specific socket implementation. Tests were conducted in Poznan Supercomputing and Networking Center using SGI Origin 3000 and IBM SP2. In all tests 7 processes were used. Fig. 7.8 presents relative errors for Origin 3000 supercomputer. Relative errors are different for different communication methods and best results (1.8-3.6%) were achieved for MPI. For shmem library relatives

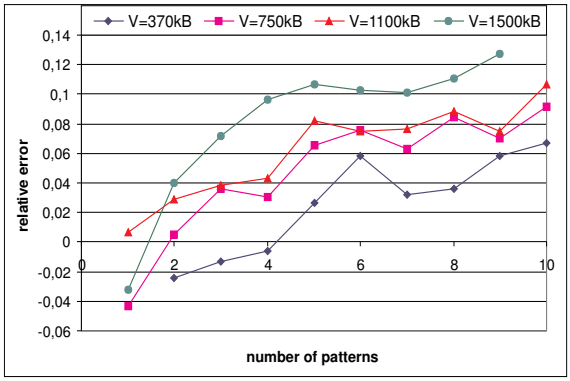


Figure 7.6: Difference between model and measurement on platform G in search for pattern application.

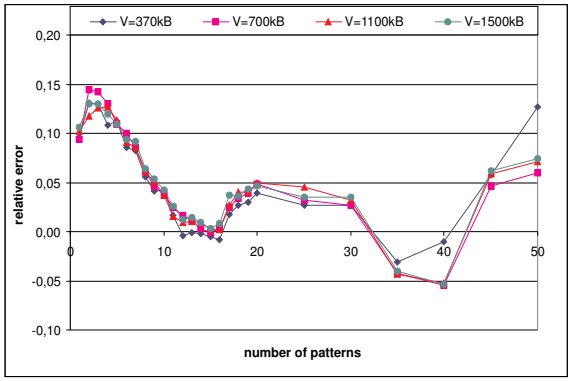


Figure 7.7: Difference between model and measurement on platform B2 in search for pattern application.

error were 19-36% but it resulted from imprecision of communication speed measurement. Origin 3000 is a shared memory machine therefore communication over shmem library is very fast and takes about an order of magnitude less than in other methods. Because communication between processes take only few milliseconds it is not possible to measure it precisely. Fig. 7.9 presents relative errors for IBM SP2 supercomputer. The shmem library is not available for this platform because IBM SP platform is based on the idea of a cluster of workstation. Also for this machine the best results were achieved for MPI. Accuracy for this method is outstanding and the error is less than 1%. Socket and PVM implementation were also quite precise and relative errors were in range from 3 to 8%.

7.4 Discussion and conclusions

Let us observe that in most of the cases relative difference between the model and the measurement is $\approx 30\%$ and less. We believe that the coincidence of the model and experimental results can be improved if more effort is devoted to better understanding the computing environment, and more carefully setting up the experiments (e.g. if we have more control on the computer software suite). On the other hand differences below 10% (cf. Fig. 7.3, Fig. 7.4, Fig. 7.8, Fig. 7.9) indicate that there are applications

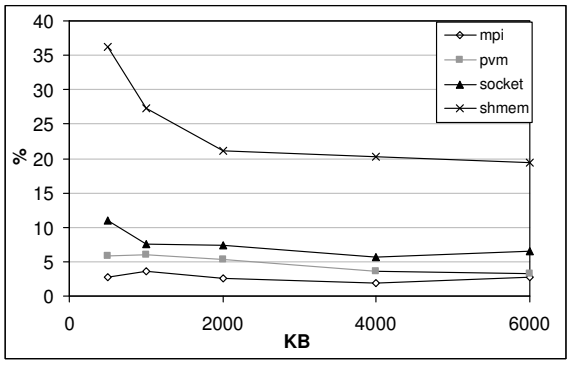


Figure 7.8: Difference between model and measurement on Origin 3000.

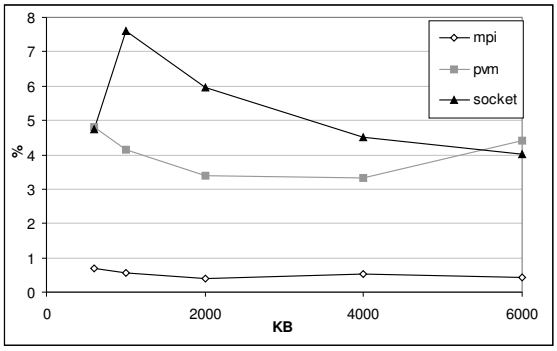


Figure 7.9: Difference between model and measurement on IBM SP2.

and platforms where the divisible task model is accurate.

It can be observed that the more uniform system we used and the more the system was dedicated to our experiments, the better the coincidence with the model was. Basing on the experience we gained it can be observed that calling operating system and runtime environment services is one of error sources in our results. For example, references to disk files or memory allocation procedures introduces great amount of uncertainty and dependence on the behavior of other software using the computer. This was also the case for long messages for which the efficiency of communication decreased as soon as the message size exceeded free core memory size. Virtual memory was used by the operating system to hold big data volumes to be communicated. In such situations the assumption about the linear dependence of the communication time on the volume of data was not fulfilled, and communication speed decreased with data size. This observation applies also to the dependence of processing time on the volume of data: in wide ranges of data sizes the assumption on linearity of this function may be violated. Distribution of the results can be another reason for disagreement of the real running time and the expectation. This applies e.g. to "search for a pattern" and "join" applications. In the model distribution of the results is uniform and any fraction of the total volume of data induces some results. This may not be the case in reality because records or text patterns may be abundant in data for one processor, and may be absent from the data for another processor. Our

experiments were performed on Ethernet network. The access time to this kind of network is not deterministic. Also the software running in parallel with our programs (e.g. interrupt drivers, operating system in general) causes that processing speed is not stable. As a result both communication and computing parameters include some amount of uncertainty, which can be estimated by the value of these parameters standard deviation. The standard deviation of C_j and A_j parameters on most of the platforms was less than approx. 0.01 of the parameter value. Deviation of startup time parameters (S_j) is much bigger, even as much as 3.3 in case of platform B1.

In this chapter we presented results of experiments on applying divisible task theory in practice. It has been demonstrated that the model is capable of accurately describing the reality. There are also cases when the predictions of the model are not satisfactory yet and these cases should need further analysis. The collected experience gives rise to a better understanding of distributed processing environments and may help improve the theoretical models. Another conclusion from the experiments is that the parameters A, C and S depend on the specific implementation of application and on chosen communication platform. Therefore these parameters cannot be taken from hardware or application specification but should be measured with the benchmarks implemented in the same way as the applications.

Chapter 8

Summary

In this work we examined various aspects of divisible job processing. We focused on the impact of memory limits on the schedule length. Linear programming methods were used to model various computing systems with different kinds of memory limits. The results of simulations were presented and discussed. Divisible Load Theory (DLT) was used to find the shortest possible schedule length and to minimize the resource requirements. The results also show versatility of DLT which establishes a link between scheduling and communication optimization.

In Chapter 3 we considered systems with single level of limited memory. It was shown that scheduling divisible load in systems with limited memory is NP-hard. Interconnection topologies of a star, and binomial tree under two different distribution strategies were studied. It was shown that in homogeneous systems and big computationally intensive applications performance of the systems is limited mainly by the processor and communication speeds limit. For such a systems the memory limitations does not have significant influence on the schedule length. The optimum processor activation order for a star-network of heterogeneous processors was also considered. Exponential optimization algorithm and polynomial-time heuristics have been presented and compared. It appeared that the shortest schedule length could be achieved with heuristics based on communication transfer rate. However, in certain situations, also other parameters, such as computing rate and memory buffers may be

important. It was also shown that IBS algorithm proposed in [53] is only a heuristic. It was possible to shorten the schedule length by using linear programming approach.

In Chapter 4 we studied systems with hierarchical memory. We demonstrated the origin of superlinear speedup arising from memory hierarchy in distributed systems. The influence of the model parameters on the performance of the computing system has been studied. Efficiency of distributed processing divisible loads in multi-installment and out-of-core modes were compared. Multi-installment processing appears to be advantageous for reasonably selected load chunks sizes.

Chapter 5 was devoted to the systems with limited communication buffers. The impact of communication buffer size D on the performance of divisible load processing in various distributed networks was examined. Interactions of several scattering algorithms with computations under limited communication buffer size have been analyzed. We observed severe performance limitations incurred by the tree structures. It was shown that influence of the limited communication buffer size manifests in several ways. It was concluded, that even for big memory buffers it is recommended to implement the limited size of the messages.

In Chapter 6 the regular multi-installment divisible job processing was considered. We studied the influence of the number of installments on the schedule length. Two kinds of processors were considered, processors with front-end and without front-end. It was shown that multi-installment processing can shorten the schedule length up to $\frac{e-1}{e}$ compared to the schedule length achieved in single-installment processing. The influence of the number of installments on memory requirements was also shown. Two methods of the load distribution were presented, with variable and fixed size of the load pieces. It was shown that there is a trade-off between the schedule length and memory requirements.

Finally in Chapter 7 in a sequence of experiments we demonstrated the viability and practicability of the divisible load model.

Streszczenie w języku polskim

Wzrastające zapotrzebowanie na moc obliczeniową zaowocowało wzrostem poplarności równoległych i rozproszonych modeli przetwarzania. Obecnie praktycznie żaden pojedynczy komputer nie jest w stanie zapewnić mocy obliczeniowej portrzebnej do przetwarzania współczesnych problemów obliczeniowych. Dlatego potrzebne są algorytmy i aplikacje, które potrafią wykorzystać jednocześnie wiele procesorów. Opracowanie szybkich algorytmów równoległych wymaga jednak odpowiedniego modelu przetwarzania. Szczegółowe modele, mimo ich dokładności, są nieprzydatne ze względu na ich złożoność i trudność analizy. Dlatego istenieje potrzeba opracowania dla wybranych zastosowań prostych modeli, które będą poprawne i łatwe do analizy.

W tej pracy rozważamy model przetwarzania zadań jednorodnych. Dla zadań jednorodnych dane mogą zostać podzielone na dowolną liczbę fragmentów i każdy fragment może być przetwarzany niezależnie. Model zadania jednorodnego znajduje wiele zastosowań w przetwarzaniu dużych zbiorów danych pomiarowych, przeszukiwaniu baz danych, w wybranych problemach algebry liniowej, symulacjach, czy w przetwarzaniu obrazów. Model zadania jednorodnego można zastosować do modelowania różnych topologii połączeń w systemach równoległych takich jak łańcuch procesorów, gwiazda, pierścień, drzewa, kraty czy hiperkostki.

Celem pracy jest zbadanie wpływu różnego rodzaju ograniczeń pamięciowych na wydajność przetwarzania zadań jednorodnych. Pod uwagę wzięto trzy rodzaje ograniczeń pamięciowych: pamięć hierarchiczną, jednopoziomową pamięć o ograniczonym rozmiarze oraz system z ograniczonym rozmiarem buforów komnikacyjnych. Przedstawione zostały sformułowania programowania liniowego dla różnych systemów

obliczeniowych. Ich rozwiązaniem jest taki przydział części obliczeñ do poszczególnych procesorów, aby czas przetwarzania zadania był jak najmniejszy. Przedstawione i omówione zostały symulacje wpływu wielu różnych parametrów systemów na jakość rozwiązań .

Przedstawiono również wyniki rzeczywistych eksperymentów, które pokazują, że model zadania jednorodnego dobrze nadaje się do modelowania pewnej klasy aplikacji, a przewidywania modelu są zgodne z rzeczywistym czasem obliczeń.

Bibliography

[1] http://www.top500.org.

[2] http://setiathome.ssl.berkeley.edu.

[3] http://www.mersenne.org.

[4] http://www.entropia.com.

[5] http://www.distributed.net.

[6] http://folding.stanford.edu.

[7] *Storage, transformation, and recomputation of integrals*, gaussian 98 technical info ed., 2000. http://www.gaussian.com/g98.htm.

[8] Bajaj, C., Pascucci, V., Thompson, D., and Zhang, X. Y. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of IEEE Parallel Visualization and Graphics Symposium* (1999), pp. 97 – 104. http://www.ticam.utexas.edu/CCV/papers/papera1.pdf.

[9] Barlas, G. D. Collection-aware optimum sequencing of operations and closed-form solutions for the distribution of a divisible load on arbitrary processor trees. *IEEE Transactions on Parallel and Distributed Systems 9*, 5 (1998), 429–411.

[10] Bataineh, S., Hsiung, T.-Y., and Robertazzi, T. Closed form solutions for bus and tree networks of processors load sharing a divisible job. *IEEE Transactions on Computers 43*, 10 (October 1994), 1184–1196.

[11] Bataineh, S., and Robertazzi, T. G. Bus-oriented load sharing for a network of sensor driven processors. *IEEE Transactions on Systems, Man and Cybernetics 21*, 5 (1991).

[12] Bataineh, S., and Robertazzi, T. G. Distributed computation for a bus network with communication delay. In *Proceedings of the 25-th Conference on Information Sciences and Systems* (Baltimore, MD, 1991), John Hopkins University, pp. 709–714.

[13] Bataineh, S., and Robertazzi, T. G. Ultimate performance limits for networks of load sharing processors. In *Proceeding of the 1992 Conference on Information Sciences and Systems* (Princeton, NJ, 1992), pp. 794–799.

[14] Berkelaar, M. *lp_solve - Mixed Integer Linear Program Solver*, 1995. ftp://ftp.es.ele.tue.nl/pub/lp_solve.

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

[16] Bharadwaj, V., D.Ghose, and Robertazzi, T. Divisible load theory: A new paradigm for load scheduling in distributed systems. to appear, 2003.

[17] Bharadwaj, V., Ghose, D., and Mani, V. Optimal sequencing and arrangement in distributed single-level tree networks with communication delays. *IEEE Transactions on Parallel and Distributed Systems 5*, 9 (September 1994), 968–976.

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

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

[20] Bharadwaj, V., Li, H. F., and Radhakrishnan, T. Scheduling divisible load in bus networks with arbitrary processor release times. *Computers and Mathematics with Applications 32*, 7 (1996), 55–77.

[21] Bharadwaj, V., Li, X., and Ko, C. C. On the influence of start-up costs in scheduling divisible loads on bus networks. *IEEE Transaction on Parallel and Distributed Systems 11*, 12 (2000), 1288–1305.

[22] Bharadwaj, V., and Viswanadham, N. Sub-optimal solutions using integer approximation techniques for scheduling divisible loads on distributed bus networks. *IEEE Transaction on System, Man and Cybernetics 30*, 6 (2001), 680–691.

[23] Błażewicz, J., and Drozdowski, M. Scheduling divisible jobs on hypercubes. *Parallel Computing 21* (1995), 1945–1956.

[24] Błażewicz, J., and Drozdowski, M. The performance limits of two-dimensional network of load sharing processors. *Foundation of Computing and Decision Sciences 21*, 1 (1996), 3–15.

[25] Błażewicz, J., and Drozdowski, M. Distributed processing of divisible jobs with communication startup costs. *Discrete Applied Mathematics 76* (1997), 21–41.

[26] Błażewicz, J., Drozdowski, M., and Ecker, E. Management of Resources in Parallel Systems. In *Handbook on Parallel and Distributed Processing* (Berlin-Heidelberg, 2000), J. Błażewicz, K. Ecker, B. Plateau, and D. Trystram, Eds., Springer, pp. 263–341.

[27] Błażewicz, J., Drozdowski, M., Guinand, F., and Trystram, D. Scheduling a divisible task in a 2-dimensional mesh. *Discrete Applied Mathematics 94*, 1-3 (June 1999), 35–50.

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

[29] Brezany, P., Bubak, M., Malawski, M., and Zając, K. Irregular and out-of-core parallel computing. In *PPAM 2001, LNCS 2328* (2002), R. W. et al., Ed., Springer-Verlag, pp. 299–306.

[30] Chan, S. K., Bharadwaj, V., and Ghose, D. Experimental study on large size matrix-vector product computations using divisible load paradigm on distributed bus networks. Tech. Rep. TR-OSSL/BV-DLT/01, Open Source Laboratory, Department of Electrical and Computer Engineering, The National University of Singapore, Singapore, November 2000.

[31] Charcranoon, S., and Robertazzi, T. G. Optimizing computing and communication costs in single level tree networks. Tech. Rep. 748, College of Engineering and Applied Science, State University of New York at Stony Brook, September 1997.

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

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

[34] Corrêa, W. T., Klosowski, J. T., and Silva, C. T. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In

Proceedings of Eurographics Workshop on Parallel Graphics and Visualization (2002), D. Bartz, X. Pueyo, and E. Reinhard, Eds. http://www.cs.princeton.edu/omnimedia/papers/piwalk.pdf.

[35] Drozdowski, M. Selected problems of scheduling tasks in multiprocessor computer systems. In *Rozprawy*, no. 321. Poznań University of Technology Press, Poznań, 1997. Also: `http://www.cs.put.poznan.pl/~maciejd/txt/h.ps`.

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

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

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

[39] Drzewiecki, D., and Drozdowski, M. Rozdział obciążeń w rozproszonym systemie komputerowym metodą zadania jednorodnego. Tech. Rep. RB - 008/97, Institute of Computing Science, Poznań University of Technology, Poznań, 1997.

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

[41] Ghose, D., and Kim, H. J. Load partitioning and trade-off study for large matrix-vector computations in multicast bus networks with communication delays. *Journal of Prallel and Distributed Computing 55* (1998), 32–59.

[42] Ghose, D., and Mani, V. Distributed computation in linear networks: Closed-form solutions. *IEEE Transactions on Aerospace and Electronic Systems 30*, 2 (April 1994), 471–483.

[43] Ghose, D., and Mani, V. Distributed computation with communication delays: Asymptotic performance analysis. *Journal of Parallel and Distributed Computing 23* (1994), 293–305.

[44] Głazek, W. Scheduling divisible loads on a chain of processors. In *Proceedings of International Conference on Principles of Distributed Systems OPODIS'97* (Paris, 1997), Editions Hermes, pp. 123–136.

[45] Głazek, W. *Algorithms for Scheduling Divisible Tasks on Parallel Machines.* PhD thesis, Dept. of Electronics, Telecommunication and Computer Science, Technical University of Gdańsk, Gdańsk, 1998.

[46] Głazek, W. Distributed computation in a three-dimensional mesh with communication delays. In *Proceedings of 6-th Euromicro Workshop on Parallel and Distributed Processing* (Madrid, January 1998), IEEE Computer Society, pp. 38–42.

[47] Głazek, W. Szeregowanie zadanń jednorodnych w hiperkostkach dla modelu komunikacji jednoportowej. *Zeszyty Naukowe Politechniki Śl askiej, Seria Automatyka 123* (1998), 145–153, note = in Polish.

[48] Gondzio, J., and Terlaky, T. A computational view of interior-point methods for linear programming. In *Advances in Linear and Integer Programming* (Oxford, 1996), Beasley, Ed., Oxford University Press, pp. 107–146.

[49] Kemp, M. Przetwarzanie w sieci internet metodą zadania jednorodnego. Master's thesis, Institute of Computing Science, Poznań University of technology, Poznań, 1999.

[50] Kim, H. J., Jee, G.-I., and Lee, J. G. Optimal load distribution for tree network processors. *IEEE Transactions on Aerospace and Electronic Systems 32*, 2 (April 1996), 607–612.

[51] Leighton, F. T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[52] Li, K. Managing divisible load in partitionable networks. In *High Performance Computing Systems and Applications* (1998), J. Schaeffer and R. Unrau, Eds., Kluwer Academic Publishers, pp. 217–228.

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

[54] Lustig, I. J., Marsten, R. E., and Shanno, D. F. Interior point methods for linear programming: Computational state of the art. *ORSA J. on Computing 6*, 1 (1994), 1–14.

[55] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Berlin, 1992.

[56] Nemhauser, G. L., and Wolsey, L. A. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1988.

[57] Peters, J. G., and Syska, M. Circuit-switched broadcasting in torus networks. *IEEE Transactions on Parallel and Distributed Systems 7*, 3 (1996), 246–255.

[58] Piotrowski, A. Zastosowanie metody zadnia jednorodnego w rozproszonym systemie komputerowym. Master's thesis, Institute of Computing Science, Poznań University of Technology, Poznań, 1999.

[59] Robertazzi, T. Ten reasons to use divisible job modeling. to appear, 2003.

[60] Robertazzi, T. G. Processor equivalence for a daisy chain of load sharing processors. *IEEE Transaction on Aerospace and Electornic Systems 29*, 4 (1993), 1216–1221.

[61] Robertazzi, T. G., Luryi, S., and Sohn, J. Load sharing controller for optimizing monetary cost. United States Patent no. 5889989, March, 30 1999.

[62] Sohn, J., and Robertazzi, T. G. Optimal load sharing for a divisible job on a bus network. In *Proceedings of the 1993 Conference on Information Sciences and Systems* (Baltimore, MD, 1993), John Hopkins University, pp. 835–840.

[63] Sohn, J., and Robertazzi, T. G. A multi-job load sharing strategy for divisible jobs on bus networks. In *Proceedings of the 1994 Conference on Information Sciences and Systems* (Princeton NJ, March 1994), Princeton University.

[64] Sohn, J., and Robertazzi, T. G. Optimal divisible load sharing on bus networks. *IEEE Transactions on Aerospace and Electronic Systems 1* (1996).

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

[66] Talia, D., and Srimani, P. K., Eds. *Parallel Data-Intensive Algorithms and Applications* (2002), vol. 28 of *special issue of* Parallel Computing.

[67] Toledo, S. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms* (1999), J. M. Abello and J. S. Vitter, Eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, pp. 161–179. http://www.math.tau.ac.il/~stoledo/Pubs/oocsurvey.pdf.

[68] Welch, T. A. A technique for high-performace data compression. *IEEE Computer 17* (1984), 8–19.

[69] Wolniewicz, P. Multi-installment divisible job processing with communication startup cost. *Foundations of Computing and Decision Sciences 27*, 1 (2002), 43–57.

[70] Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24* (1978), 530–536.

Appendix A

Notation Summary

α_i - part of volume V sent to processor for computation,

A - processing rate in a homogeneous system,

$A' = \frac{A}{C}$ - defined only for homogeneous systems,

A_i - processing rate of processor P_i in heterogeneous system,

AL_{i1}, AL_{i2} - coefficients of the linear function describing processing time for the core memory

AH_{i1}, AH_{i2} - coefficients of the linear function describing processing time for virtual memory using the disk storage.

B - size of memory buffer in a homogeneous system,

B_i - size of memory buffer of processor P_i in heterogeneous system,

B' - size of memory buffer in a homogeneous system, relative to the size of the volume V,

C - transfer rate in a homogeneous system,

C_j - transfer rate of link j in heterogeneous system,

C_{max} - schedule length,

C_{max}^{inf} - schedule length in a system with unlimited memory,

C_{max}^{sat} - schedule length in a system with total memory size equal to V,

C_{max}^{1} - schedule length for single-installment processing,

C_{max}^{n} - schedule length for multi-installment processing with n steps ,

$C'^{n}_{max} = \frac{C^{n}_{max}}{CV}$,

C^{∞}_{max} - schedule length for multi-installment processing with $n = \infty$ steps ,

D - size of communication buffer in homogeneous system,

D' - size of communication buffer in a homogeneous system, relative to the size of the volume V,

D_i - size of communication buffer of processor P_i in heterogeneous system,

δ - size of a chunk in simple multi-installment algorithm,

G - gain from multi-installment processing,

h - number of layers in a binomial tree,

m - number of processors,

n - number of steps in multi-installment processing,

n_{min} - minimum admissible number of steps in multi-installment processing,

n^* - optimal number of steps in multi-installment processing to achieve the minimum schedule length,

p - degree of a node in a binomial tree,

P_0 - originator,

P_i - processor number i,

$\mathcal{P}$ - the set of processors

$\rho = \frac{C}{A}$ - defined only for homogeneous systems,

$\sigma = \frac{S}{A}$ - defined only for homogeneous systems,

S - communication startup time in a homogeneous system,

$S' = \frac{S}{CV}$ - defined only for homogeneous systems,

S_j - communication startup time for link j in heterogeneous system,

swp - swap point in a homogeneous system,

swp_i - swap point of processor P_i,

T_s - the time between transmissions of consecutive data chunks to the same processor in multi-installment processing,

V - total size of the load.