Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems

Maciej Drozdowski

Instytut Informatyki
Politechnika Poznańska
Poznań, 1997

Contents

Summary

Contemporary computer systems are multiprocessor or multicomputer machines. Their efficiency depends on good methods of administering the executed works. Fast processing of a parallel application is possible only when its parts are appropriately ordered in time and space. This calls for efficient scheduling policies in parallel computer systems.

In this work deterministic problems of scheduling are considered. The classical scheduling theory assumed that the application in any moment of time is executed by only one processor. This assumption has been weakened recently, especially in the context of parallel and distributed computer systems. This monograph is devoted to problems of deterministic scheduling applications (or tasks according to the scheduling terminology) requiring more than one processor simultaneously. We name such applications multiprocessor tasks. In this work the complexity of open multiprocessor task scheduling problems has been established. Algorithms for scheduling multiprocessor tasks on parallel and dedicated processors are proposed. For a special case of applications with regular structure which allow for dividing it into parts of arbitrary size processed independently in parallel, a method of finding optimal scattering of work in a distributed computer system is proposed. The applications with such regular characteristics are called divisible tasks. The concept of a divisible task enables creation of tractable computation models in a wide class of computer architectures such as chains, stars, meshes, hypercubes, multistage networks. Divisible task method gives rise to the evaluation of computer system performance. Examples of such performance evaluation are presented.

This presentation summarizes earlier works of the author as well as contains new original results. The results are presented in a unified form in the context of the current state-of-the-art in the analyzed field. The results obtained point out further research directions.

Chapter 1

Introduction

1.1 Scheduling tasks in multiprocessor computer systems

The increase of the computer speed and their ability to solve bigger and bigger problems is an everlasting challenge for the designers. As computer systems grow more complex and their speed increases the problems that must be overcome to further increase the speed and the "capacity" seem to grow even faster. The difficulties follow physical phenomena at the foundations of computer devices technology. For example consider a processor technology. A limited yield of the current sources in the integrated circuits for the fixed clock period limits the maximum length of buses and internal connections in the circuit. And vice versa for the given size of the connections the frequency of the clock is limited. Thus, to increase the speed the yield of the current sources must be higher or the size of the devices must be smaller. Furthermore, in order to minimize the number of defected circuits in one piece of silicon, the chips are reduced in size. This, and growing complexity of the processors results in increasing density of power dissipation. Yet, it cannot grow to infinity. Moreover, since the photolithography methods are limited by the light wave length further miniaturization becomes slower and more costly than in the recent years. Hence, it seems that unless new ways [197] are found to overcome the existing technological problems the development of processors will be slower and prohibitively expensive [123].

A solution to this problem can be in exploiting potential simultaneity in execution of some independent program fragments. In other worlds, exploiting parallelism of computations can be the answer. It can be verified that even in commonly performed engineering and scientific computations there is a great potential for parallel computations [140]. The idea of reducing computation time by concurrent execution of some parts of a program is over a hundred and fifty years old [139]. Despite that, concurrent computations are not so common in contemporary programs. There can be at least two reasons for this situation: limited technology and difficulties in creating correct and efficient parallel applications. With the advent of relatively cheap and powerful microprocessors the first reason became easier to overcome and many vendors started to offer multiprocessor systems. Furthermore, some parallelizing methods have been implemented in contemporary microprocessors [4, 115, 156] (multiple instruction issue, out of order instruction execution). The second reason seems to be much more significant. It appears that developing an efficient and correct parallel application is not a trivial task. An important issue is that parts of the application must be executed in a proper sequence and should not wait for their data more than necessary. Thus, feasible and efficient scheduling[1] parts of a parallel application is very important. Consequently, the field of scheduling for multiprocessor systems is significant in the design of libraries and compilers [11, 108, 170, 175]. Contemporary parallel computer systems are valuable assets shared by many users. The access to the shared resources must by managed by the operating system. Hence, scheduling of tasks is important also for the designers and administrators of operating systems [69, 185]. In hard-real-time environment, where programs must be completed before deadlines, scheduling is particularly important element of the system design [114, 179, 184, 192, 212]. Scheduling is also one of the main areas of contemporary mathematics [193] as a branch of combinatorial optimization. The origin of scheduling lies in the operations research [10, 30, 43, 68, 143] mainly in production and project management. Only later were these results applied in the management and control of computer systems.

When building a schedule it is an objective to build the one which is the best possible in the sense of some criterion, e.g. the shortest schedule. On the other hand, for practicality reasons the time spent on constructing such a schedule cannot be long. In particular, the time should be polynomially bounded - i.e. growing polynomially, not exponentially, with the growth of the problem size. Satisfying these two requirements is sometimes difficult. When the scheduling problem is computationally hard (precisely **NP**-hard)

[1]The notions used in this section in an intuitive sense will be defined more rigorously in the following sections of this work

then according to the current state of the knowledge polynomial optimization algorithm should not be expected. Thus, it is a crucial problem to indicate which problems are solvable "fast" (in polynomial time). Determining, that a problem is not computationally hard is equivalent with demonstrating a useful algorithm solving the problem. Proving that the problem is computationally hard is a qualitative indication that it is hard to expect an algorithm which is both polynomial and always builds optimal solutions. In such a case it can be advantageous to use fast heuristic algorithm which gives feasible solution, but not necessarily an optimal one. The computational complexity theory supplies methods for the analysis of the problems from the point of view of the necessary computational costs as well as presents the methods of dealing with special classes of problems. Analysis of the algorithms results not only in the algorithms building schedules, but also in the qualitative directions for the design of computer systems. For example, in some architectures it is possible to determine when it is better to execute an application on all available processors and when it is more efficient to use only one processor [172] (without intermediate possibilities).

The deterministic scheduling originated as a branch of operations research and as such has over fifty years of history, and a wide range of theoretical and practical results. The domain is so immense that its systematic presentation is beyond the size of this work. However, many important aspects of scheduling in parallel computer systems were not considered by the classical scheduling theory. This work is devoted to the presentation and analysis of such problems - the problems of scheduling in multiprocessor computer systems.

1.2 The Goal and the Scope of This Work

In the sequel we consider the problems *deterministically*. This means that all the parameters describing the tasks and the computer systems are fixed values (uncertainty is not considered). This approach is justified in many practical situations and in the worst-case analysis. For these reasons it is widely applied when considering scheduling problems. The deterministic character of task parameters has been discussed in many earlier works [10, 30, 43, 68, 143]. In the context of this work it is necessary to explain deterministic character of such parameters of the task as the number of required processors or the set of required processors. Parallel applications are often prepared for a precisely known number of processors. The choice of the actual number of processors can be done by the programmer, by a compiler or by the operating system at the loading time. If changing the number of processors executing an application is possible at the run-time, then still there exist a number of processors which can be most efficiently exploited. In the case of dedicated processors not a number but a set of processors is required. In such dedicated environment the application has a predetermined set of processors necessary for its execution. Thus, the number of required processors or the set of required processors can be considered as known deterministically. This issue is further analyzed in Sections 2.2, 3.1, and 5.1. According to the taxonomy of [56] the scheduling we consider is global and static. In other words, we assume that decisions on scheduling are centralized, the used policies remain constant, and all the required knowledge about the workload is available.

The domain of scheduling in parallel computer systems cannot be considered independently from the architectural constraints and from the programming environment. Hence, the features of contemporary multiprocessor systems important for this work will be presented.

There are many alternative approaches to achieving efficiency of parallel computer systems. Some of them concentrate on a particular element of the system, other try to optimize the system as a whole. The examples can be *allocation, load balancing, routing* etc. which often differ only very slightly from the classical scheduling. It appears that such partial approaches separated from the issues of scheduling have a limited influence on the efficiency of the computer system [133, 213]. Thus, it seems impossible to have an efficient computer system without satisfactory scheduling algorithms. On the other hand, from the practical viewpoint it is impossible to use only the scheduling models. It is a consequence of intractability of design and implementation of scheduling algorithms tackling every possible aspect of a parallel computer system. Hence, in complex systems cooperation between scheduling and the algorithms optimizing other elements of the computer system seems required. Furthermore, a growing number of researchers attempt to incorporate communication constraints in the scheduling models [42]. Consequently, the related approaches and their links with scheduling will be presented.

The classical scheduling theory assumes that a task for its execution requires only one processor at a time. This assumption is disregarded recently, especially in the context of parallel applications in the multiprocessor computer systems. The tasks requiring more than one processor at the same moment of time will be called *multiprocessor tasks.* This work is dedicated to scheduling multiprocessor tasks. The problems of multiprocessor task scheduling can be divided into two classes: scheduling on parallel processors and scheduling on dedicated processors. The computational complexity analysis of open multiprocessor scheduling problems will be conducted. For selected problems polynomial-time algorithms will be presented.

There exists a class of computational tasks which have a very regular linear structure, e.g. processing measurement data [60], some problems of linear algebra [27]. Such computational tasks can be divided into parts of (almost) arbitrary sizes. The parts can be solved independently in parallel by different processors. The transmission times and the processing times for the parts are proportional to the sizes of the parts. Tasks with such characteristic will be called *divisible*. The concept of divisible tasks allows for a creation of simple models of communication and computation processes for a wide class of computer architectures. This enables finding an optimal distribution of the computational task and evaluating the performance of a computer system.

For analyzed problems the previously existing results will be shown using a unified notation. The proposed notation is an attempt to unify communication aspects of computer system with the scheduling problems. This work comprises the results obtained by the author, collected in the context of the current state of research, which allows for pointing out further research directions.

The organization of the work is the following. In Chapter 2 important features of contemporary parallel computer systems will be described. In Chapter 3 basic definitions of scheduling theory, computational complexity analysis, parallel algorithm performance evaluation will be introduced. In Chapter 4 an overview of the problems related to scheduling in multiprocessor computer system will be presented. Chapter 5 addresses scheduling of multiprocessor tasks. Chapter 6 considers divisible task scheduling. Chapter 7 contains final remarks and conclusions.

Chapter 2

Parallel Computer Systems

2.1 Hardware

The field of parallel computing is immense. Thus, we introduce here only basic concepts referred to in the further sections.

It is common to start a description of parallel systems with an attempt of classifying types of parallelism and types of parallel machines. A useful view on parallelism types is distinguishing between *data parallelism* and *code parallelism* (cf. Fig 2.1). Data parallelism is a situation in which the same operations are performed simultaneously on the data structures of the same type, whereas in code parallelism different operations are performed in parallel. Another view of parallel processing classification considers *granularity* of parallelism. Granularity is a measure of the synchronizations frequency among independent threads of parallel execution [98]. Granularity can be also viewed as a size of the units by which work is assigned to processing elements [76]. When granularity is fine the synchronizations are frequent, e.g. every instruction. When granularity is coarse the synchronizations are rare, e.g. every 10^6 instructions.

Classical computers execute instructions in the order dictated by the sequence in the program code. This approach is called control-driven or von Neumann architecture. A different approach where instructions are executed as soon as their operands become available is called data-driven or data flow. Using this concept *data flow machines* were built like Manchester Dataflow or LDF100 [76, 111].

In [96] control-driven computers have been divided into four classes: SISD (single instruction stream, single data stream), SIMD (single instruc-

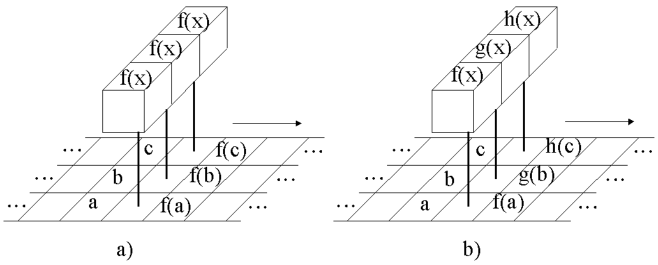

Figure 2.1: Illustration of a) data parallelism and b) code parallelism.

tion stream, multiple data streams), MISD (multiple instruction streams, single data stream), MIMD (multiple instruction streams, multiple data streams). SIMD and MIMD are currently regarded as the classes of parallel computer systems. A variation of SIMD is SPMD (single program multiple data streams). MISD can be a model for machines with pipelined computations. The division into SIMD (resp. SPMD) and MIMD coincides with distinguishing data and code parallelism. Another classification divides parallel computers into *multiprocessors* and *multicomputers*. A multiprocessor is a computer with processors communicating via a shared memory (e.g. CRAY X-MP, Y-MP, IBM 3090 [9]). A multicomputer consists of a set of processors with local memories, interconnected by some kind of network. We will name by *processing element* (PE) a processor with local memory and a network interface. When a processor has a local memory which is not accessible for other processors, then only by passing messages can some other processor access the contents of nonlocal memory. Thus, the above classification coincides with the division into *message-passing* architectures (multicomputers) and *shared-memory* architectures (multiprocessors). The message-passing computers can be divided into two classes: tightly-coupled and distributed. Distributed systems are (usually) heterogeneous computers with different operating systems, connected by (usually) heterogeneous topology Local/ Metropolitan/ Wide Area Networks (LANS, MANs, WANs). This class provides a relatively low cost parallel computing environment which recently became very popular and was successfully applied (e.g. [196]). As the opposite, tightly-coupled systems can be characterized by: homogeneous PEs, uniform interconnection, uniform operating system, single vendor and a single boxing.

Tightly-coupled computers can be further differentiated by the type of PE interconnection. In this work we limit considered interconnection types to: bus(es), point-to-point networks (called also single-stage networks), and multistage networks. *Bus* interconnection is a classical concept in which processors (or PEs) communicate over a shared bus. Machines like C.mmp, Cm*, Alliant FX/8, LDF100, Sequent Balance, SGI Power and Challenge are based on the bus concept [9, 98, 186]. Furthermore, majority of contemporary microprocessors are able to use buses. *Point-to-point* networks link pairs of processing elements (or switches to which PEs are attached). A path between two arbitrary processors in the network may require several hops at the intermediate processors or their network switches. In the class of point-to-point interconnections we distinguish: hypercube (machines: Cosmic Cube, nCUBE1, nCUBE2, CM-2, FPS T, Intel iPSC/2, iPSC/i860), and mesh (possibly torus) (examples: MPP, AP1000, DASH, Alewife, J - Machine, Paragon, Cray T3D) [9, 47, 55, 75, 76]. A *hypercube* ([180]) of dimension d consists of 2^d PEs which can be labeled using d-bit long binary string. Two PEs connected by a link have labels differing on exactly one bit. Transputer interconnections are examples of point-to-point networks, but the topology varies depending on the actual system.

In *multistage networks* PEs are connected by several layers of switches while the internal layer switches have no PEs attached. Multistage networks are divided here into: trees and multistage cube network [161]. In the tree networks a message to reach the destination must go up and down the hierarchy of switches. This kind of networks include fat-tree [146] (e.g. CM-5, Meiko CS-2), and hierarchy of rings (KSR1, KSR2) [47] (cf. Fig. 2.2). The fat-tree is a binary tree with PEs at the leaves and the number of links growing while moving down to the root of the tree. The root is connected to the "external world". The multistage cube network (MCN) is a representative of a wider class of interconnections in which processors are linked by several layers of switches where each layer has the same number of switches (e.g.: BBN Butterfly, SP1 and SP2). When computations are performed by systems consisting of hundreds or more nodes we will say that this is massively parallel processing (MPP). In this work when talking about MPP systems we will mean mainly tightly-coupled systems.

The classification of point-to-point architectures is not full without describing the communication subsystem. When a PE has no specialized communication hardware (e.g. bare T800 Transputers without external switches) the processor must perform communication and routing functions. Hence, it is not able to communicate and compute in parallel. PEs in majority of modern computers are equipped with communication hardware and the overlap of computation by communication is possible. Another element of the architecture is the maximum number of active ports per PE. If communication

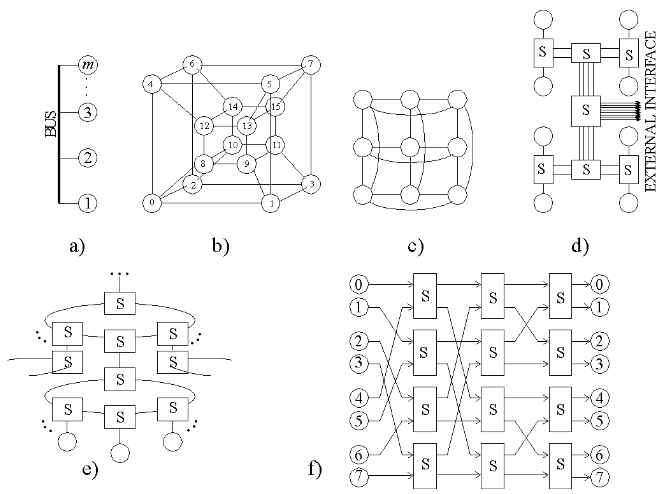

Figure 2.2: Illustration of the interconnection types: a)bus, b)hypercube $d = 4$, c)2-dimensional torus mesh $m = 9$, d)fat-tree $m = 8$, e)hierarchy of rings, f)multistage cube network $m = 8$ (S - switch).

over only one link at a time is possible we call PEs *1-port*. In the opposite case PEs are said to be *p-port* (where p stands for the maximum number of links at a PE which can communicate simultaneously).

The next element of the architecture is the commutation mode. The commutation mode is a physical protocol for message routing. We describe commutation modes here because routing functions are increasingly executed by dedicated hardware. The methods we refer to in this section are also called switching or routing techniques. This should not be mixed with the routing problems alluded to in Section 4.5. Among various commutation (or routing) modes we distinguish *store-and-forward*, *circuit-switched* and *packet-switched* [125, 163]. In the following, distance d is the number of links between the sender and the receiver. For all the commutation modes the communication time between two neighbors is equal to $T_{com} = S + LC$, where S corresponds to the communication start-up time (message packing, routing decision, circuit setting-up), C represents the transmission rate (time units per data unit), and L is the message length.

In the *store-and-forward* mode when a PE sends a message to another PE located at distance d, the message (either as whole or in packet pieces) is sent to the closest PE on the path and it is stored there. Then, this intermediate node sends the message to the next node on the path, and so on until the message reaches its destination. In this mode, the distance is a crucial parameter in the communication delay:

$$T_{com} = d(S + LC)$$

In *the circuit-switched* mode from the transmitter to the receiver a *header* of the message is sent which reserves all the links of a communication path to form a circuit between both PEs. Then, the remaining part of the message is sent in one step. The message is not stored in any intermediate node along the path. The communication delay is:

$$T_{com} = S + d\delta + LC,$$

where δ represents the time needed to commute a switch. Parameter $\delta \ll S$ ($\delta \approx (0.1\% \ldots 1\%)$ of S), and can be neglected. Hence, for this mode, the communication delay does not depend significantly on the distance. This observation is confirmed by experiments [163].

In the *packet-switched* modes the message is split into packets which consist of *flits*. Flits are also called flow of control digits. These are words passed over a link in one control cycle (e.g. clock cycle, or hand-shake cycle). The first flit plays the role of the header, the rest of flits follow it immediately, the last one releases the communication "pipe". The model of the communication delay is the same as in the previous mode. Among the packet-switched modes three sub-types can be identified: wormhole, virtual-cut-through and buffered-wormhole modes. These modes differ in the behavior of flits and packets when the packet cannot move forward (e.g. there is no free link).

• In the *wormhole* mode, the progressing of the message in the pipe is stopped. All the flits remain in the intermediate buffers thus blocking the links.

• In the *virtual-cut-through* mode flits continue progressing on their way until reaching the site where the first flit is stopped. There a whole packet is waiting for release of the link. This mode assumes infinite capacity of buffers.

• In the *buffered-wormhole* mode, flits of some packets move until they reach the stopped flit, then the whole packet is stored there. Yet, the number of packets that can be stored is limited by buffers capacity.

Since the communication delay time can be described in the same way for packet-switched and circuit-switched modes, in this work we distinguish only store-and-forward and circuit-switched modes (circuit-switched includes packet-switched and circuit-switched modes). In Table 2.1 we give examples of the timing parameters for some existing machines [52, 118]. Though a detailed analysis of communication delay time shows that T_{com} is a more

Table 2.1: Example communication parameters

Machine Name	Interconnection	Commutation	S μs	C $\frac{\mu s}{byte}$
Intel iPSC/2	Hypercube	Circuit-Switched	136	0.384
Intel iPSC/i860	Hypercube	Circuit-Switched	350	0.2
Meiko CS-1	Mesh	Store-and-Forward	250	1.000
Think. Mach. CM-5	Fat-tree	Wormhole	73	0.1
Intel Paragon	2D-mesh	Wormhole	100	0.005
Meiko CS-2	Fat-tree	Wormhole	12	0.02
Cray T3D	3D-torus-mesh	Wormhole	8.57	0.0033
IBM SP-1/SP-2	Multistage	(Buffered)Wormhole	39	0.0125
Parastation	Mesh	Wormhole	3083	1.04
Cray C-90	Shared memory	-	0.108	0.0001

complex function [88, 117], in this work we adopt the above models of T_{com} for their simplicity and satisfactory accuracy.

The introduced classification is summarized in Fig. 2.3. Note that the upper and the bottom parts of the figure present exclusive differentiation, while the division of message passing branch is not exclusive, e.g. some hypercube-interconnected computer can use both store-and-forward, 1-port and overlapped communication. The classification we use is not intended to be ultimate, rather than that we wanted to show basic ways of differentiating among the parallel computers. It is not difficult to point out its limitations. The division into control-driven and data-driven computers is not so obvious when considering out-of-order instruction execution by modern microprocessors [115]. Superscalar processors are internally MIMD but for the "outside world" are SISD. Division into shared-memory and message-passing computers also becomes fuzzy when we realize that the memory can be logically shared but physically distributed. Because of the software-hardware dualism the hardware support for distributed shared memory will probably grow in the future and the two classes can converge. Moreover, the same computer system may include several different interconnections at different levels of hardware (e.g. CS-2 has omega network (a kind of multistage network) in the switch and fat-tree of switches) or used for different purposes (e.g. CM-5 has fat-tree as the data network and binary-tree as the control network). Finally, the performance is the ultimate goal for building parallel systems. Hence, the above classes can converge in the future to some yet unpredictable efficient blend of different ideas.

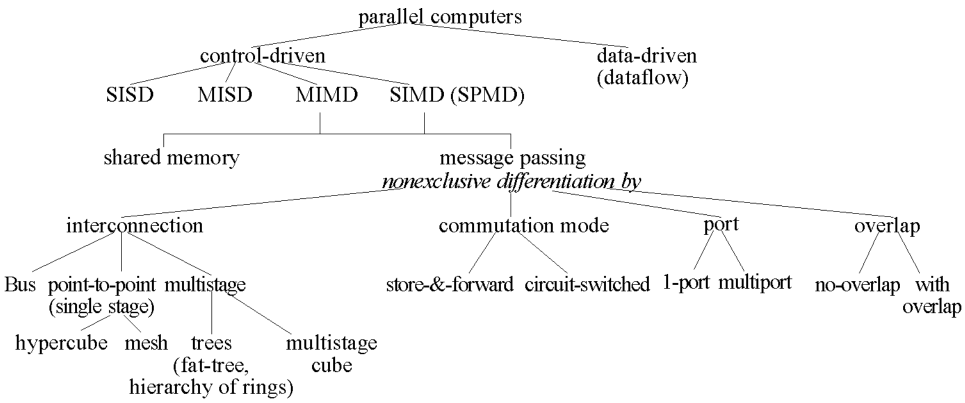

Figure 2.3: The classification of parallel computers.

2.2 Software

In many common applications (programs) great potential parallelism can be found [140]. Thus, programs can be executed via many concurrent *threads* (mutual relations between the notions of an application, a thread and a task, are precisely defined in Section 3.1). Computer systems should provide support for implementing parallelism of an application including the issues posed by scheduling. In this section we introduce some aspects of programming models. Then, we consider operating system support for parallel applications. We pay special attention to the methods of scheduling parallel computations.

Shared memory parallel systems are mature programming platforms. For such architectures extensions handling parallelism have been proposed in popular programming languages [9], especially, in the parallelism of loops (cf. Section 4.4). Unfortunately, it seems that shared-memory architecture does not offer good scalability perspectives (here scalability means potential for increasing the number of cooperating processors - cf. Section 3.3). On the other hand, distributed memory systems offer almost unbounded scalability opportunities. Though many message-passing environments were offered and successfully applied (e.g. PVM [196], Express[165], NX/2, Parmacs[47], MPI [204] etc.), the underlying message-passing architecture is difficult to program. Thus, a concept of distributed memory which is logically shared has been coined to hide the hardware from the view of the programmer. Implementing distributed shared memory rises many issues like data consistency, performance, scalability etc. Several approaches to the implementation of distributed shared memory in tightly-coupled message-passing systems have been applied including, for example, virtual shared memory, latency hiding by use of multiple threads, cache-based distributed shared memory [101]. Also in loosely-coupled distributed systems the concept of logically shared memory has been realized in the form of associative memory offered by Linda environment [54]. Observe that since the underlying communication architecture is based on message-passing, for the programmer there is little difference between tightly-coupled system and distributed system.

Parallel processing imposes different requirements on operating systems to the standard general-purpose single-processor machines. The performance is a prerequisite of parallel systems existence. For that reason performance should not be sacrificed for the functionality of the operating system [178]. Thus, the parallel application must not be punished by unnecessary system functions which are not used but still contribute to memory occupation and latencies. For example, it was pointed out in [118], (cf. Table 2.1) that even tightly-coupled message-passing systems are outperformed by shared-memory systems as far as communication parameters are considered. It is observed in [178] that the communication startup time consists in up to 74% of the processing by the micro-kernel of the operating system. Thus, the functionality of the operating system, and scheduling in particular, should be tightly tailored to the needs of the application (as it is, for example, in QNX operating system [173]). Now, we put to the scrutiny the way applications are scheduled.

Parallel operating systems are evolving from previously existing systems and many ideas have been "naturally" inherited. Based on acceptable response time two load types have been distinguished in single-processor systems [185]: *terminal* (or *interactive*) and *batch* load. Since batch tasks are submitted to the computer system far earlier than their actual execution begins, deterministic scheduling algorithms can be applied. For the terminal load which requires immediate response, access to processors is granted on the basis of FCFS, Round-Robin, multi-level priority queues etc. The concept of a "single" queue has been inherited by the parallel systems: ready processors are assigned work from a system-wide queue. This approach has been applied e.g. in Cray X-MP [9]. Unfortunately, actual parallel execution of application threads depends on the machine loading and there is no guarantee of simultaneous execution on several processors. It is explained in Section 5.1 that such a situation can lead to a significant performance deterioration. To enable simultaneous execution of the application threads, an idea of *coscheduling* (sometimes called *gang scheduling*) has been applied in Medusa operating system of Cm* [98]. The coscheduled threads are assigned to processors in the same time quantum. Coscheduling is often implemented in shared-memory systems (e.g. Alliant FX/8, SGI Challenge). For example, in the IRIX 5.1 operating system running on SGI Challenge multiprocessors coscheduling of threads is possible when requested by a parallel application. IRIX 5.1 insures parallel running of the threads by increasing the priority of the threads when the first thread of the application is scheduled. All threads of the application are expected to be running in at most 10ms (60ms is the time quantum). IRIX 5.1 ensures that even in the heavy load conditions 72% of the coscheduled application run time is truly parallel.

A different approach is based on the concept of *partitions*. All the processors of the computer are divided into separated partitions. The application is granted simultaneous access to all the processors in a partition. Partitions provide means of restricting access to portions of the processor set for particular users, types of jobs and a way to specify various scheduling characteristics on different parts of the computer. For example, there are often different partitions for batch and interactive load. Partitions have been applied e.g. in Paragon, Cray T3D, KSR2, CM-5. In Paragon, Cray T3D, KSR-2 the user can specify the size of the required partition. In Cray T3D two kinds of partitions are distinguished: hardware and software partitions. In the former special hardware protects boundaries of the partition, while in the latter the microkernel of operating system ensures isolation of applications in different partitions. In CM-5 size of the partition is managed by control processor to which the user has interactive access. Time sharing of a partition by different applications is possible e.g. in Paragon and CM-5, but is not e.g. in Cray T3D. Since the application startup is time-consuming, the time quanta are very long (minutes to hours).

Existing computer resources are often not fully exploited [160]. Idle cycles of workstations and personal computers can be a cheap source of computing power. For such systems software is developed which identifies idle workstations, manages access to them, schedules tasks on such resources, supports process migration etc. [80, 104, 107, 162]. Furthermore, computers are often connected into clusters controlled by one scheduler. In such systems programs can be submitted for batch processing. Usually while submitting a task, a user can specify (among other features) the number of required computers, their architecture, maximum processing time. Network Queuing System (Cray), Load Sharing Facility (Convex), Load Leveler (IBM), Prospero Resource Manager [162], Condor [104] are examples of such systems.

Most of them claim support for parallel tasks, i.e. tasks requiring many machines simultaneously. Probably a final form could be called a *metasystem* or *metacomputer* [107, 194] - a single computing resource composed of heterogeneous distributed computers.

To this end let us remark on some analogies between processor scheduling and memory management. The simplest form of memory management is a situation in which one program uses all the available memory. Analogously, a single application on all processors gives the most of possible flexibility and performance. To allow for multiprogramming memory has been partitioned and the parts were used by different programs. In contemporary computers processor partitions are introduced. This rises similar problems as in the optimization of memory utilization: internal and external partition, recognizing and compacting idle processor partitions. Finally, a virtual memory allowed for almost unlimited size of program memory which is mapped into real memory by operating system. Currently parallel applications must gear to the available number of processors which is usually constant throughout application lifetime. Yet, it is not inconceivable to allow for using as many processors as the application needs and to change this number in the run-time. For this purpose *virtual processors* have been introduced which are mapped by operating system into real processors (e.g. by time sharing). Furthermore, the idea of *processor working set* [98, 99] is almost immediate analogy between virtual memory and processor allocation. The processor working set is the number of processors which must be granted simultaneously to the application to enable acceptable progress in computation. Still, it is disputable if the idea of virtual processors will be widely accepted because it is conceptually close to processes (possibly with allowing for process migration to idle processors). Moreover, this increase in functionality must be paid for in reduction of performance. There are also significant differences between memory and processor allocation. The processor allocation strategy must take into account the interconnection topology and the communication pattern of the application. Processors may be equipped with differing hardware and software. Thus, processors are not as uniform as memory units. Moreover, processors are not as easily partitionable as memory space. Some architectures are well partitionable (e.g. meshes, hypercubes) some other are not well suited for partitioning (e.g. multistage cubes). Finally, these days the number of processors is much smaller than the number of page units (yet, this may change in the future). Thus, although there are analogies between memory and processor management, different algorithms must be used for these problems.

Chapter 3

Notions and Definitions

3.1 Deterministic Scheduling Theory

In this section basic notions of scheduling theory will be defined. Extensions necessary to deal with multiprocessor and divisible tasks will be introduced. We will propose a notation to describe considered scheduling problems.

When analyzing scheduling problems three elements must be determined: (i) computing environment comprising processor set $\mathcal{P}$, communication system and other resources $\mathcal{R}$, (ii) task system $\mathcal{T}$, (iii) optimality criterion.

We assume that processor set $\mathcal{P}=\{P_1,\ldots,P_m\}$ consists of m elements. Two classes of processors can be distinguished. *Dedicated processors* and *parallel processors*. Dedicated processors are specialized devices performing differing functions. For example, we often say that specialized processors such as I/O, arithmetic, vector, graphic, signal processors are dedicated. Moreover, even identical processors can be considered as dedicated in certain situations. A multiprocessor task can be considered as executed by a dedicated processor also for the preallocation reasons. For a certain communication pattern among the parts of the parallel application and for a given communication network it can be advantageous to map tasks to processors in some fixed way. Changing the preallocation may increase the communication overhead (due to dilatation, congestion etc., we say more about allocation in Section 4.1). Since the costs of filling a pipeline, vector registers or a cache are high it is disadvantageous to transfer tasks to new sites frequently. Hence, there is a kind of affinity between tasks and processors [155] and parallel processors may behave as dedicated devices. In production systems machines are regarded as dedicated rather than as parallel. In dedicated environment a multiprocessor task requires certain processors not just some number of them. Hence, a *set* of processors is required simultaneously in this case. Alternatively, a task may be executed by some family of alternative processor sets. As it is in the classical scheduling theory [43, 68] multiprocessor tasks may consist of operations. In such a case we distinguish three types of dedicated processor systems: flow-shop, open-shop and job-shop. In the flow-shop all tasks have the same number of operations which are performed sequentially and require the same sets of processors. In the open-shop the order among the operations is immaterial. For the job-shop, the sequence of operations and the sets of required processors are defined for each task separately.

In the case of parallel processors each processor can execute any task. Hence, a task requires some *number* of arbitrary processors. As in the classical scheduling theory parallel processors are divided into three classes: *identical* processors - provided that all tasks are executed on all processors with the same speed, *uniform* processors - if the execution speed differs from processor to processor, and *unrelated* processors - for which execution speed depends on the processor and on the task. In each of the above cases speed of the processor can be determined. However, for the purposes of this work it is more convenient to use processing rate which is reciprocal of the speed. Processing rate is expressed in the units of time per unit of work. The rate of identical processors will be denoted A, of uniform processor P_i: A_i, and in the case of unrelated processors A_{ij} for processor P_i processing task number j. When processors are uniform or unrelated the slowest processor determines speed of processing the whole multiprocessor task.

An important element of the multiprocessor computer is its communication system. The classical scheduling theory originated in the time when multiprocessor systems with few tightly coupled processors dominated. In such systems processors could be considered as fully connected and the communication time was negligible. Nowadays, parallel systems are often multicomputers (cf. Section 2.1) comprising many *processing elements* connected via some kind of network. To be precise we should talk about processing elements rather than about processors. Yet, in this work these two names are equivalent. There is also a great variety of interconnection architectures. Each processing element is characterized by its ability (or inability) to communicate and compute simultaneously. If simultaneous computing and communication is possible then there must be some specialized *network processor* in each PE which performs all network communication functions and off-loads the computing processor. In such a case we say that the communication can *overlap* computation (a system with *overlap* in short). In the opposite case the communication system is without overlap. Another characteristic is ability (or inability) of a processing element to simultaneously communicate by several ports. If it is possible to communicate by p ports simultaneously we say that the system is *p-port*. In the opposite case only one port can communicate at a time and the system is 1-port. According to Section 2.1 we distinguish two basic ways of transferring the messages: store-&-forward and circuit-switched routing. Furthermore, the communication links between processors will be described by communication startup time S and transmission rate C (cf. Section 2.1) when the links are identical, if the links differ we will denote for link i startup and transmission rate S_i, C_i, respectively.

Apart from the processors there can be also a set $\mathcal{R} = \{R_1, \ldots, R_k\}$ of additional resources, each available in $| R_i |$ units $(i = 1, ..., k)$.

The second element of the scheduling problem is the task system. We will explain now the relations between the notions of an application, a thread and a task. An *application* (or a program) can be executed (at least potentially) by many processors working concurrently. A *thread* is the basic unit of processor utilization [125, 185]. Thus, any thread is executed by a single processor. A thread is equivalent to a program stream with independent instruction counter running within the environment of an application. Hence, threads within the application are not isolated from each other. In this work, any activity inherently running on a single processor will be considered as equivalent, from the scheduling viewpoint, to a thread. Analogously, activities which can be performed on many processors (even only potentially) will be considered as applications. A *task* is the basic scheduling unit. Depending on the scheduling model, the task can be equivalent either to an application or to a thread. In the classical scheduling models the application consists of some (potentially) concurrent activities which are subject to scheduling. Thus, in this case tasks correspond to threads. In the case of multiprocessor tasks, where the application is considered without its internal structure, the task corresponds to the application.

We assume that the set of tasks $\mathcal{T}$ consists of n tasks $T_1, \ldots, T_n$. For the whole task system it is possible to determine such features as preemptability (or nonpreemptability) and existence (or unexistence) of precedence constraints. These characteristics are defined as in the classical scheduling theory [43, 68]. Tasks are *preemptable* when each task can be interrupted and restarted later without incurring additional costs. In such a case the schedules are called to be *preemptive*. Otherwise, tasks are *nonpreemptable* and schedules *nonpreemptive*. Tasks are *dependent* if some task T_j must be completed before starting some other task T_i, which we denote $T_j \prec T_i$. Precedence constraints are represented as directed acyclic graphs (DAGs). In the opposite case tasks are independent. New features are *variable profile* and *divisibility* of tasks. A profile of the multiprocessor task is fixed when the number (for parallel processors) or the set (for dedicated processors) of used processors does not change during the execution of a task. A profile is variable if it is possible to change within the schedule the number (of parallel) or the set (of dedicated) processors used by a task. Unless otherwise stated, for variable profile tasks we assume that the cost of expanding a task to a new (different) processor is negligible and that not granting a processor to a task does not increase total amount of work. A task is divisible when it is possible to divide it into parts of arbitrary size and execute these parts independently in parallel on different processors. Note that divisibility and changing task profile resemble preemption. A preemptable task can be interrupted at any moment, hence it can be also divided into chunks of arbitrary size. However, divisibility requires additionally that there are no precedence constraints (and thus no communication) among the copies of the task running in parallel, which is not necessarily the case of preemptable tasks. When a task profile is changing it means that the task appears and disappears on some processor(s), this is thinkable when it is possible to interrupt processing and restart a task. While considering allocation problem (cf. Section 4.1) dependencies among tasks are often represented as *task graphs*. These are weighted graphs representing communications among tasks (nodes) by weights of the edges and processing times of the tasks by weights of the nodes.

Each task separately T_j $(j = 1, \ldots, n)$ is described by a number of parameters. We enumerate them in the following.

1. *Number of operations* n_j. This parameter is given for tasks scheduled on dedicated processors. $n_j > 1$ implies that task T_j consists of operations $\{O_{j1}, \ldots, O_{jn_j}\}$.

2. The *set of simultaneously required processors* fix_j or the *family of alternative processors,* set_j. These parameters are defined only in the case of dedicated processors. The multiprocessor task requires for its processing a set fix_j of dedicated processors simultaneously. It is also possible that more than one set of processors can execute a task. Such a set of alternative processor ensembles will be called a family of alternative processors set_j. We will use the concept of a family of alternative processors only when $\mid set_j \mid > 1$. Analogously, for flow-shop, open-shop, and job-shop set fix_{ji} or

family set_{ji} is defined for operation O_{ji}.

3. The *number of simultaneously required processors* $size_j$, or the *maximum number of usable processors* δ_j, or the *set of usable numbers of processors* any_j. These parameters are defined in the case of parallel processors. When the first parameter is given (task *size* or task *width* in short) the task can be executed only on $size_j$ processors required simultaneously. The multiprocessor task can be executed by some number of processors from the range $[1, \delta_j]$ if the second parameter is given. In the case of the third parameter the task can be executed by various numbers of processors, enumerated in set any_j. We assume that elements of any_j are ordered according to the increasing values. Note that in the case of uniprocessor tasks $size_j = \delta_j = 1$. When $size_j$ is given then $\mid any_j \mid= 1$, when δ_j is defined then $\mid any_j \mid= \delta_j$. When the number of processors executing a task is not restricted, then any_j includes all processor numbers from 1 till m. We will denote $\Delta = \max_{T_j \in \mathcal{T}}\{\delta_j\}$ (for tasks executed by only one number of processors $\Delta = \max_{T_j \in \mathcal{T}}\{size_j\}$).

4. *Execution time.* In the case of scheduling the task on set fix_j of dedicated processors the execution time will be denoted $t_j^{fix_j}$. When task T_j can be executed by a family of alternative processors set_j then for each $fix_{ji} \in set_j$ the processing time is defined and denoted by $t_j^{fix_{ji}}$. In the case of preemptable tasks it is necessary to determine how long a task must be processed in many intervals (possibly by different processor sets) to consider it as finished. Analogously to the classical scheduling on uniform and unrelated processors we assume that task T_j executed in l different time intervals of lengths τ_i, by processors in various sets fix_{ji} $(i = 1, \dots, l)$, is finished when $\sum_{i=1}^{l} \frac{\tau_i}{t_j^{fix_{ji}}} \geq 1$. For a task consisting of n_j operations the execution time $t_{ji}^{fix_{ji}}$ is defined for each operation O_{ji} requiring processors in set fix_{ji}. Analogously, for operation O_{ji} with family set_{ji} of alternative processors execution time $t_{ji}^{fix_{jil}}$ is defined for each $fix_{jil} \in set_{ji}$.

Situation is different in the case of parallel processors. Let us examine identical processors first. When task T_j can be executed only by $size_j$ processors, its execution time is $t_j^{size_j}$. If it can be executed by various numbers of processors, then for each feasible number k of processors execution time t_j^k is defined. There is a number of models describing relation between execution time and the number of used processors. This relation is called *parallelism signature* (cf. Section 3.3). In the literature it is:

• an arbitrary discrete function [90],

- an inversely proportional function (i.e. $t_j^k = \frac{t_j^1}{k}$) [202],
- an inversely proportional function up to $k = \delta_j$ [205],
- a function inversely proportional to k^α (i.e. $t^k = \frac{t_j^1}{k^\alpha}$) where $0 < \alpha < 1$ [170].
- an arbitrary continuous function [208].

Analogously to the case of dedicated processors, we consider preemptable task T_j executed in l intervals of length τ_i on k_i processors ($i = 1, \ldots, l$) as being finished when $\sum_{i=1}^{l} \frac{\tau_i}{t_j^{k_i}} \geq 1$. To calculate execution time of a task on uniform and unrelated processors one has to take into account speeds of the processors. We assume here that the slowest processor determines processing speed of the whole task.

For divisible tasks actual execution time depends not only on the processor speed but also on the speed of communication medium, and scattering algorithm. Hence, it is more convenient to express the required amount of work by the volume of data that must be processed. For the problems with one task only symbol V will denote this volume, and V_j ($j = 1, \ldots, n$) for the problems with more than one task.

5. *Ready time* r_j. A task can be executed only after r_j.

6. *Due-date* or *deadline* d_j. A task should be finished not later than by d_j. If the task must be finished before d_j then this moment is called a deadline.

7. *Weight* or priority w_j. It can be interpreted also as the cost of remaining of T_j in the computer system.

8. *Resource requirements* R_{ji}. The task may additionally require R_{ji} units of resource R_i.

Before describing the third element of a scheduling problem formulation we define a schedule.

Definition 3.1 Schedule *is an assignment in time of tasks to processors (and resources) satisfying the following requirements:*
- Each processor executes at most one task at a time.
- In the case of dedicated processors, multiprocessor task T_j requiring processors in set fix_j is granted all these processors throughout all its execution time. Task T_j with family set_j of alternative processors is executed by exactly those processors which are specified in the used set(s) $fix_{ji} \in set_j$. Operation O_{ji} receives all processors required in fix_{ji} or when $\mid set_{ji} \mid > 1$ all processors included in the used set(s) $fix_{jil} \in set_{ji}$.
- In the case of parallel processors, a multiprocessor task which can be executed by only one number $size_j$ of processors is granted that number of processors simultaneously throughout all its execution time. When maximum

number δ_j of usable processors is specified, in no moment of time is the task executed by more than δ_j processors simultaneously.
- Tasks with fixed profile, when executed on parallel processors use always the same number of processors, and when executed on dedicated processors use always the same set of processors.
- Task T_j is not executed before r_j $(j = 1, \ldots, n)$.
- For each pair $T_j \prec T_i$, task T_j is completed before T_i starts.
- All tasks are executed.
- Nonpreemptable tasks are not interrupted, and preemptable tasks are interrupted a limited number of times.

Given a schedule one can determine for task T_j:
- completion time c_j,
- flow time $f_j = c_j - r_j$,
- lateness $l_j = c_j - d_j$,
- tardiness $\tau_j = \max\{0, c_j - d_j\}$
- whether it is late: $U_j = 1$ if $c_j > d_j$, $U_j = 0$ otherwise.

The optimality criteria constituting the third element of a scheduling problem are:

Schedule length (makespan) $C_{max} = \max_{1 \le j \le n}\{c_j\}$.

Maximum lateness $L_{max} = \max_{1 \le j \le n}\{l_j\}$.

Mean flow time $\overline{F} = \frac{1}{n}\sum_{j=1}^{n} f_j$. Note that it is equivalent to *total completion time* $\sum_{j=1}^{n} c_j$.

Mean weighted flow time $\overline{F_w} = \frac{\sum_{j=1}^{n} w_j f_j}{\sum_{j=1}^{n} w_j}$. It is equivalent to *total weighted completion time* $\sum_{j=1}^{n} w_j c_j$.

Number of late tasks $U = \mid \{T_j : U_j = 1\} \mid$.

Weighted number of late tasks $\sum_{j=1}^{n} w_j U_j$.

Mean tardiness $\frac{1}{n}\sum_{j=1}^{n} \tau_j$, which is equivalent to $\sum_{j=1}^{n} \tau_j$.

Notation

To denote the analyzed scheduling problems we will use standard three -field notation $\alpha \mid \beta \mid \gamma$ proposed in [105] with extensions introduced in [44, 199]. $\alpha \mid \beta \mid \gamma$ scheme in its three fields describes processor system (α), task system (β), optimality criterion (γ). Since the modifications proposed in [199] are not satisfactory to describe the variety of the considered scheduling problems, we propose further expansion of the notation. In the sequel we concentrate on the new elements of the notation. Symbol $\circ$ will denote empty (nonprintable) character which in the problem notation is skipped.

The first field contains symbols $\alpha_1, \ldots, \alpha_7$. The first two symbols are the standard ones:

$\alpha_1 \in \{1, P, \overline{P}, Q, R, O, F, J\}$ - describes the type of processors ($\overline{P}$ - means that the number of identical processors is not bounded).

$\alpha_2 \in \{k, \circ\}$ - denotes the number of processors fixed to k or not fixed ($\circ$) by the definition of the problem.

The third symbol $\alpha_3 \in \{win, \circ\}$ - denotes, respectively, that processors are available in time windows or always available.

Symbol α_4 describes the processor interconnection architecture $\alpha_4 \in \{\circ, conn, chain, star, tree, bus, mesh, hypercube, multistage\}$ (cf. Section 2). The following values of α_4 denote:

- $\alpha_4 = \circ$ interconnection is irrelevant because communication delays
(i) are negligible, or
(ii) for the considered communication system have been included into the execution time of multiprocessor tasks, or
(iii) for the considered communication system have been included in the communication time required to transfer data/results among two dependent tasks allocated to different processors (cf. β_7).
- $\alpha_4 = conn$ interconnection of an arbitrary type;
- $\alpha_4 = chain$ chain of processors;
- $\alpha_4 = star$ star of processors (i.e. single-level tree);
- $\alpha_4 = tree$ tree-type interconnection;
- $\alpha_4 = bus$ bus interconnection;
- $\alpha_4 = mesh$ regular rectangular mesh (possibly $3D{-}mesh, 2D{-}mesh$ for three- and two-dimensional meshes);
- $\alpha_4 = hypercube$ hypercube;
- $\alpha_4 = multistage$ multistage interconnect.

Symbols $\alpha_5, \alpha_6, \alpha_7$ are defined only when interconnection is explicitly considered, i.e. when $\alpha_4 \neq \circ$. Symbol $\alpha_5 \in \{no-overlap, \circ\}$ denotes:

- $\alpha_5 = no{-}overlap$ on no PE can computation overlap communication;
- $\alpha_5 = \circ$ simultaneous communication and computation is possible.

Symbol $\alpha_6 \in \{s\&f, csw\}$ denotes two basic types of routing:

- $\alpha_6 = s\&f$ - store-and-forward;
- $\alpha_6 = csw$ - all types of routing for which the communication delay can be reduced to a single startup time and a term linearly dependent on the volume of transferred data, i.e. wormhole routing, circuit switching, virtual-cut-through etc.

Symbol $\alpha_7 \in \{p-port, \circ\}$ denotes:

-$\alpha_7 = p-port$ - simultaneous communication by at most p ports of a PE is possible;
-$\alpha_7 = \circ$ - all ports of a PE can communicate simultaneously.

The second field $\beta = \beta_1, \ldots, \beta_{10}$ defines the task system.

$\beta_1 \in \{spdp-lin, spdp-lin-\delta_j, spdp-any, size_j, cube_j, fix_j, fix_{ij}, set_j, set_{ij}, \circ\}$ - describes the type of the multiprocessor task.
- $\beta_1 = spdp–lin$ - denotes that t_j^k is inversely proportional to k, in other words, speedup is linear (cf. Section 3.3).
- $\beta_1 = spdp–lin–\delta_j$ - describes a situation similar to the previous one, but the task cannot use more than δ_j processors simultaneously.
- $\beta_1 = spdp–any$ - execution time t_j^k is an arbitrary function of k.
- $\beta_1 = size_j$ - a task can be executed by only one number $size_j$ of processors.
- $\beta_1 = cube_j$ - is a special case of $size_j$ demanding that tasks be executed by numbers of processors being powers of 2 (1, 2, 4, 8,... etc. processors). This situation refers to scheduling on hypercubes.
- $\beta_1 = fix_j$ - denotes that tasks can be executed by only one set fix_j of simultaneously required dedicated processors. In the case of a multiprocessor task comprising a number of operations we use $\beta_1 = fix_{ij}$.
- $\beta_1 = set_j$ - means that tasks have families of alternative dedicated processors which can execute them. In the case of tasks with operations we will use $\beta_1 = set_{ij}$.
- $\beta_1 = \circ$ - stands for standard uniprocessor tasks.

According to the current value of β_1 we will say that tasks require processors according to model $spdp–lin$, $spdp–lin–\delta_j$ etc.

$\beta_2 \in \{div, pmtn, var, \circ\}$ - denotes divisibility, preemptability, variable profile or their absence.
- $\beta_2 = div$ - tasks are divisible.
- $\beta_2 = var$ - denotes variable profile. Note that $\beta_2 = var$ implies $\beta_1 \in \{spdp–lin, spdp–lin–\delta_j, spdp–any, set_j, set_{ij}\}$.
- $\beta_2 = pmtn$ - tasks are preemptable, but the profile is fixed.
- $\beta_2 = \circ$ - denotes that tasks are nonpreemptable and their profiles are fixed.

The rest of the notation for the task system is classical:

$\beta_3 \in \{prec, tree, chain, \circ\}$ - describes the type of precedence constraints.

$\beta_4 \in \{p_j = 1, p_{ij} = 1, \circ\}$ - means, respectively, that processing times of tasks are equal, processing times of operations are equal, processing times are arbitrary.

$\beta_5 \in \{r_j, \circ\}$ - tasks have different (r_j) or identical ($\circ$) ready times.

$\beta_6 \in \{res\,\lambda\sigma\rho, \circ\}$, where $\lambda\sigma\rho \in \{k, \cdot\}$ - denotes the type of additional resource requirements ($res\lambda\sigma\rho$) or absence of such requirements ($\circ$).

$\lambda \in \{k, \cdot\}$ - denotes the number of resource types fixed to k or arbitrary $(\cdot)$;
$\sigma \in \{k, \cdot\}$ - means that each resource has either fixed number of k units or $(\cdot)$ the numbers of resources' units are given in the instance of the problem;
$\rho \in \{k, \cdot\}$ - implies that for any resource the maximum number of its units required by any task is fixed to k or $(\cdot)$ that it is arbitrary.

$\beta_7 \in \{com, c_{jk}, c_{j*}, c_{*j}, c, c = 1, \circ\}$ - denotes communication delays appearing when two dependent tasks are executed by different processors.
- $\beta_7 = com$ - the communication delays depend on the volume of transferred data, the function binding time and volume is arbitrary;
- $\beta_7 = c_{jk}$ - the communication delay is defined for each pair of dependent tasks;
- $\beta_7 = c_{j*}$ - the communication delay depends on the transmitter only;
- $\beta_7 = c_{*j}$ - the communication delay depends on the receiver only;
- $\beta_7 = c$ - all the communication delays last c units of time;
- $\beta_7 = c = 1$ - all the communication delays last one unit of time;
- $\beta_7 = \circ$ - no communication delay takes place.

$\beta_8 \in \{dup, \circ\}$
- $\beta_8 = dup$ - tasks can be duplicated (to avoid communication delays) i.e. multiple copies of the same task can be executed independently,
- $\beta_8 = \circ$ - duplication is not allowed.

$\beta_9 \in \{n = 1, \circ\}$ - denotes that only one task is considered $(n = 1)$ or the number of tasks is arbitrary $(\circ)$.

$\beta_{10} \in \{d_j, \circ\}$ - marks either that deadlines are imposed on the tasks (d_j), or $(\circ)$ that no deadlines are considered (still, due-dates can be defined for a due-date involving optimality criterion).

The third field $\gamma = \gamma_1$ where $\gamma_1 \in \{C_{max}, L_{max}, U, \sum c_j, \sum w_j c_j, \sum w_j U_j, \sum \tau_j, -, X\}$ denotes the optimality criterion. Symbol "$-$" indicates testing for existence of a feasible schedule. When a non-standard optimality criterion is considered we denoted such a case by X.

3.2 Complexity Theory

In this section we present basic concepts of computational complexity analysis for combinatorial problems. The description is only a rough outline of the complexity theory. More comprehensive treatment of this subject can be found in [28, 97, 137]. The complexity analysis enables the determination of the computational complexity class of a considered problem, and gives directions for dealing with problems in certain classes.

Among the combinatorial problems we can distinguish decision problems and optimization ones. Decision problems consist in answering "yes" or "no" to some question. Optimization problems require extremalization of some objective function. Each optimization problem has its decision version (but not vice versa) which is not computationally harder than the original version. Hence, it is possible to analyze computational hardness of optimization problems, such as scheduling problems, by considering only their decision counterparts (the links between decision and optimization versions are even tighter [28, 97]). To classify inherent computational complexity of various problems two reliable measures are necessary: a measure of the problem size, and a measure of the execution time which is the considered computational expense. As a measure of size of problem instance I, the length $N(I)$ of a string encoding its data is used. All encoding schemes are equivalent for purposes of complexity analysis provided that:
(i) numbers are encoded using counting system with base greater than 1,
(ii) encoding is not redundant,
(iii) the encoded string can be decoded.
As a measure of the execution time for algorithm A and problem size n the maximum number of elementary steps taken by a computer for any instance $I \in D_\pi$ is used, where D_π is the domain of problem π and $N(I) = n$. Such a measure of execution time is called algorithm complexity function (in short: *algorithm complexity*). When the number of steps can be bounded from above by a polynomial in the problem size we say that the algorithm is *polynomial time* (or *polynomial* in short). When the complexity function cannot be bounded in this way the algorithm is called *exponential time* (*exponential* in short). Observe that this definition of exponential algorithms includes also complexity functions which are not considered exponential and name *nonpolynomial* seems more precise. Yet, we stick to a traditional term introduced in [28, 97]. We will say that algorithm A has complexity $O(f(n))$ when the complexity function $g_A(n)$ satisfies: $\exists C$ such that for almost all $I \in D_\pi : g_A(N(I)) \leq Cf(N(I))$. Still, the execution time cannot be reliably measured without establishing a model of the computer system. We distinguish two types of computer system models: realistic and unrealistic. The class of *realistic models* comprise such models of computers as: Deterministic Turing Machine (DTM), k-tape Deterministic Turing Machine, Random Access Machine. All the above models are equivalent for the task of classifying the complexity of considered problems because algorithms polynomial on one of the three models remain polynomial on any other realistic machine. The class of unrealistic models includes e.g.: Nondeterministic Turing Machine

(NDTM), Oracle Turing Machine (OTM). Unrealistic models are capable of performing computations nondeterministically, which can be interpreted as ability of executing unbounded number of computations in a unit of time.

A crucial element of computational complexity analysis is establishing the complexity class which the considered problem belongs to. In this presentation we use only three basic classes of computational complexity (for decision problems): **P**, a class of **NP**-complete problems (**NP**c in short), and a class of problems **NP**-complete in the strong sense (s**NP**c).

Definition 3.2 *Class* **P** *includes all problems solvable in polynomial time on DTM.*

To define the remaining two classes we have to introduce additional notions.

Definition 3.3 *Class* **NP** *includes all problems solvable in polynomial time on NDTM.*

From these definitions (and definitions of DTM, NDTM [28, 97]) it can be concluded that $\mathbf{P} \subseteq \mathbf{NP}$. Yet, it has neither been proved that $\mathbf{P} \neq \mathbf{NP}$ nor that $\mathbf{P} = \mathbf{NP}$. It is only known that DTM can simulate NDTM in exponential time.

Definition 3.4 *By a* polynomial transformation *of problem π_2 to problem π_1 (which is denoted $\pi_2 \propto \pi_1$) we call a function $f : D_{\pi_2} \to D_{\pi_1}$ such that:*
(i) $\forall_{I_2 \in D_{\pi_2}}$ the answer is "yes" if and only if it is "yes" for $I_1 = f(I_2)$,
(ii) $\forall_{I_2 \in D_{\pi_2}}$ function f can be calculated in time polynomial in $N(I_2)$.

Definition 3.5 *Decision problem π_1 is* **NP**-complete *if $\pi_1 \in \mathbf{NP}$ and $\forall_{\pi_2 \in \mathbf{NP}}\ \pi_2 \propto \pi_1$.*

Hence, if there existed a polynomial algorithm for any **NP**-complete problem then any problem in **NP** would be solvable in polynomial time. Though it has not been shown that $\mathbf{P} \neq \mathbf{NP}$ for years, no polynomial time algorithm is known for any **NP**c problem. Furthermore, this class includes many computationally difficult combinatorial problems. From the definition it can be concluded, that to prove **NP**-completeness of some problem it is enough to show that some **NP**c problem polynomially transforms to the analyzed problem. The first problem proved to be **NP**c is SATISFIABILITY. Nowadays, it is known that class **NP**c includes thousands of problems and subproblems from many fields of combinatorial optimization. For example, in [143] it is

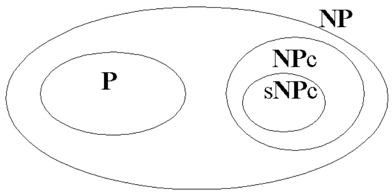

Figure 3.1: Relation between complexity classes provided that **P**$\neq$**NP**.

said that (decision versions of) 417 scheduling problems are in **P**, 3821 in **NP**c, and for 298 problems complexity is not known.

Although no polynomial algorithms were found for problems in **NP**c, for some of them algorithms have been found with complexity bounded by polynomial in the instance size $N(I)$ and the maximum numerical value $Max(I)$. Such algorithms are called *pseudopolynomial*. Mind that these are not polynomial algorithms. On the other hand, for some problems pseudopolynomial time algorithms were hard to be found. Strongly **NP**-complete problems are the ones for which no pseudopolynomial algorithms exist (unless **P**=**NP**). This class is defined as follows.

Definition 3.6 *Let π_p for problem π and some polynomial p denote subproblem of π obtained by restricting D_π to instances such that $Max(I) \leq p(N(I))$. Problem π is* strongly **NP**-complete *when $\pi \in$ **NP** and $\pi_p \in$ **NP**c.*

Showing strong **NP**-completeness using this definition is not very convenient. The following definition and theorem give a simpler way of doing this.

Definition 3.7 *By a* pseudopolynomial transformation *of problem π_2 to π_1 we call function $f : D_{\pi_2} \to D_{\pi_1}$ such that:*
(i) $\forall_{I_2 \in D_{\pi_2}}$ the answer is "yes" if and only if it is "yes" for $f(I_2)$,
(ii) $\forall_{I_2 \in D_{\pi_2}}$ function f can be calculated in time polynomial in $N_2(I_2)$ and $Max_2(I_2)$,
(iii) there exists polynomial q_1 such that $\forall_{I_2 \in D_{\pi_2}} q_1(N_1(f(I_2))) \geq N_2(I_2)$,
(iiii) there exists polynomial q_2 such that $\forall_{I_2 \in D_{\pi_2}} Max_1(f(I_2)) \leq q_2(Max_2(I_2), N_2(I_2))$.

Theorem 3.1 [97] *If $\pi_2 \in s$**NP**c, $\pi_1 \in$ **NP**, and π_2 can be transformed pseudopolynomially to π_1 then $\pi_1 \in s$**NP**c.*

The relations between the defined classes are presented in Fig. 3.1.

Now, we show how to apply the above notions to analyze optimization problems. For optimization problems an equivalent of the class of **NP**c problems is the class of **NP**-hard problems and for the class of s**NP**c problems is the class of strongly **NP**-hard problems (in short **NP**h and s**NP**h, respectively). The optimization problems can be represented as search problems consisting of the domain D_π and the set of feasible solutions $Z_\pi(I)$ for each $I \in D_\pi$.

Definition 3.8 *$R(\pi, e) = \{(x, y) : x$ is a string encoding $I \in D_\pi$ according to coding rule e, y is a string encoding $Z_\pi(I)$ using rule $e\}$.*

The search problem π for encoding rule e is solvable in polynomial time when there exists some program for DTM solving relation $R(\pi, e)$, i.e. for a string encoding instance $I \in D_\pi$ the program finds a string encoding a solution from $Z_\pi(I)$ if such a solution exists (if $Z_\pi(I) = \emptyset$ empty string is returned as a solution).

Definition 3.9 *By a* polynomial Turing transformation *of problem π_1 to π_2 (denoted $\pi_1 \propto_T \pi_2$) we mean algorithm A solving problem π_1 on DTM by use of some hypothetical procedure P solving problem π_2. A is polynomial-time provided that P can be executed in polynomial time by DTM.*

Definition 3.10 *$R(\pi, e)$ is* **NP***h when there exists some* **NP***c language L such that $L \propto_T R(\pi, e)$. The search problem π is* **NP***h when $R(\pi, e)$ is* **NP***h.*

More informally, the above definitions can be summarized in the following way. Language L is equivalent to decision problem π_1: for the given string encoding $I \in D_{\pi_1}$ does I belong to L? Hence, there exists some **NP**c decision problem π_1 which can be solved in polynomial time provided that $R(\pi, e)$ can be solved in polynomial time. Furthermore, **NP**c problem π_1 can be solved in polynomial time when the optimization problem π (represented by $R(\pi, e)$) is solvable in polynomial time. Thus, π is **NP**h when there exists some **NP**c problem π_1 such that $\pi_1 \propto_T \pi$. Note that formulating decision version π_1 of optimization problem π is immediately a proof of $\pi_1 \propto_T \pi$. Hence, **NP**-completeness of a decision version of some problem implies **NP**-hardness of its optimization version. Analogously to strong **NP**-completeness, strong **NP**-hardness is defined. To prove s**NP**h of some problem it is enough to prove s**NP**c of the decision version. Observe that an **NP**h problem cannot be solved in polynomial time unless **P**=**NP**. For many optimization problems it can be also shown [28, 97] that if **P**=**NP** these problems would be solvable in polynomial time. Thus, the complexity classes introduced for decision problems are useful in the analysis of optimization problems.

There are practical consequences of determining the complexity class of a considered problem. When the problem belongs to class **P** then it is solvable in polynomial time which in practice means that it can be solved "fast" (i.e. in reasonable time). Further analysis of such problem complexity consists in the search for the lowest complexity algorithm. On the contrary, **NP**-hardness of some problem (or **NP**-completeness of its decision version) results in the combinatorial explosion when the optimal solution is searched for. Thus, only exponential *optimization* algorithms (i.e. the ones finding the optimal solutions) have been proposed for **NP**h problems. For **NP**h problems which are not s**NP**h, pseudopolynomial algorithms, like dynamic programming, can be proposed. When the optimality of the solution is not as important as the time in which the solution is obtained, heuristics can be used. A *heuristic* is an algorithm which finds a feasible solution of the problem. However, there is no guarantee of optimality. A prerequisite of using some method as a heuristic is its low-order polynomial execution time. Heuristics which give solutions close to the optimum (on average, in the worst case) are obviously preferred. To evaluate the worst-case performance of some heuristic H we will use the worst-case *performance ratio* (performance ratio in short): $S_H = \inf\{r \geq 1 : \forall_{I \in D} \frac{f_H(I)}{OPT(I)} \leq r\}$, where I - the instance, D - the problem domain, $f_H(I)$ - the value of solution generated by H on I, $OPT(I)$ - the optimal value of the criterion for I. The way of proceeding with analysis of the problem complexity and the resulting solution methods are summarized in Fig. 3.2. The the running lines show ways of establishing the complexity class of the problem, while dashed lines indicate possible ways of proceeding with construction of an algorithm solving the problem.

3.3 Performance of Parallel Applications

Many factors contribute to the efficiency of a parallel application (task). Among them are scheduling policies assumed while mapping an algorithm (in its pure mathematical sense) to a real application in a particular computer architecture. In this section we introduce basic notions of the parallel application performance description.

The expected outcome of parallelization is reduction of the execution time. Hence, application *execution time* is the base for majority of efficiency descriptions. The other counterpart of efficiency measures are costs at which the low execution time can be achieved. The most important resource which we must pay with for the reduction of execution time are processors. From

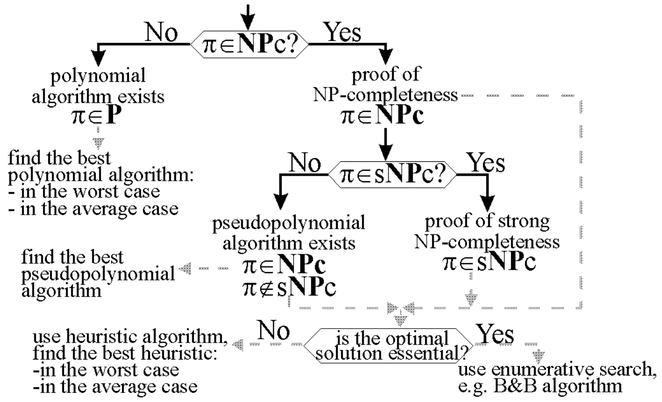

Figure 3.2: An outline of problem π complexity analysis.

these two parameters *speedup* is calculated [125]:

$$S_k = \frac{t^1}{t^k}$$

where k is the actual number of processors assigned to execute a task, t^1 is sequential execution time (i.e. on one processor), t^k is execution time on k processors. Two famous theoretical laws link speedup with the number of processors used: Amdahl's law [5]

$$S_k = \frac{1}{s + (1-s)/k}$$

and Gustafson's law [112]

$$S_k = s' + k(1 - s')$$

where s is the ratio of time spent in immanently sequential part of the code to the whole sequential execution time, s' is the ratio of time spent in sequential part to a total parallel execution time. In the Amdahl's law it is assumed that the size of the problem is fixed, while in the Gustafson's law it is assumed that the size of the problem grows linearly with the number of used processors. It was pointed out by other researchers (e.g. [82]) that this two views do not exploit the whole variety of possible situations. The required behavior of an application is that the speedup be linear, which means that increasing the number of processors reduces the execution time (inversely) proportionally. In most situations speedup is less than linear, but in certain cases speedup greater than linear can be obtained [98]. More precisely the parallel execution time in relation to the number of assigned processors is characterized by a function named *parallelism signature* [99]. The parallelism signature is often modeled by the function:

$$t(k) = a + \frac{b}{k} + c(k)$$

where a is the time spent in the sequential part of the program, b is the amount of work which is unboundedly parallelizable, $c(k)$ is the overhead (e.g. communication) introduced by adding more processors; $c(k)$ is a function increasing with k (linearly in the simplest case). Observe that the above measures (speedup, parallelism signature) are equivalent as based on t^k.

The number of processors used by an application can change over time while for measuring speedup the number of used processors is bounded from above. Hence, speedup may not represent parallelism in the application perfectly. A parameter which does not have such limitations is *execution profile* [99, 140, 181]. Execution profile is a function of the number of processors used vs. time, measured on a computer with unbounded number of processors. In other words, execution profile represents parallelism of the application in time when the number of available processors does not limit the application. From execution profile parameters like average parallelism, maximum parallelism can be derived.

The above measures determine application *scalability*, i.e. its ability to effectively utilize processors [141]. The application has good scalability when increasing the number of processors reduces proportionately the execution time. When increasing the number of processors returns diminishing reductions of execution time the application scalability is bad. Thus, a good scalability means that speedup is linear in a wide range of circumstances [215].

Chapter 4

Overview of Related Problems

The field of scheduling in parallel computer systems is very diverse. In this chapter we describe main approaches related to the problems considered in this work. Since in Chapter 6 we analyze some communication aspects in scheduling, a subsection on communication optimization is included here. Though the methods presented in this chapter are not directly applicable further in the work, we describe them for the completeness of the presentation in the work on scheduling in parallel computer systems. Some methods described here and in the further sections are based on similar assumptions. Some other methods are based on different assumptions and thus are hard to compare. Hence, in the last part of this chapter we point out differences in foundations of the scheduling concepts considered in this work.

4.1 Allocation

Some researchers consider scheduling as comprising two components: allocation and sequencing. In some situations sequencing of activities is not so important. For example, consider an application with clearly separated computation and communication phases. In such a program all processors interchange data, then compute, interchange data again, etc. If all communications can be performed in parallel (e.g. only the neighboring processors have to communicate, buffers have sufficient capacity) then detailed analysis of sequencing each operation in relation to other operations may be unnecessary. The allocation problem in itself consists in determining where to execute a task while disregarding the sequencing of the tasks. An allocator (i.e. an algorithm solving the allocation problem) can be a part of a two-level scheduling system in which modules are allocated to processors first, and then tasks are sequenced on the processors. Hence, the allocation problem can be considered as a relaxed version of the scheduling problem.

Now, we describe the allocation problem more precisely. Consider a parallel application whose modules are communicating with each other. The application is to be executed on a set of processors connected by some network. For fast execution of the computations processors should work in parallel. On the other hand, physical distribution of modules causes communication delays. Thus, to minimize communication overhead it can be advantageous to allocate frequently communicating modules close to each other (possibly on the same processor). Since the actual sequencing of the activities is not taken into account, the precedence constraint graph is reduced to an acyclic graph representing interactions among the tasks "integrated" over run-time. Such a graph is called a *task graph.*

The task allocation problem can be formulated as follows. Define:

$x_{ik} = 1$ when task i is executed on processor k, otherwise $x_{ik} = 0$;

c_{ij} - the number of data units transferred from task i to task j;

d_{kl} - is an interconnection-related communication cost of moving one unit of data between processors k and l (for example, by setting $d_{kl} = \infty$ it is possible to forbid allocating communicating tasks to processors k and l which are not directly connected);

t_{ik} - cost of executing task i on processor k.

The problem is to minimize:

$$\sum_{k=1}^{m}\sum_{i=1}^{n}(t_{ik}x_{ik} + \sum_{l=1}^{m}\sum_{j=1}^{n} c_{ij}d_{kl}x_{ik}x_{jl}) \tag{4.1}$$

subject to

$$\sum_{k=1}^{m} x_{ik} = 1 \quad \text{for} \quad i = 1, \ldots, n$$

The above formulation (with minor variations) was used in e.g. [57, 62, 152, 153, 158]. Observe tight links to quadratic assignment problem [187]. The above objective function is reasonable for unrelated processors. It can be observed that for identical and uniform processors its minimum is achieved when all the work is performed on the fastest processor. Thus, in [73, 182, 210] instead of function (4.1) the working time of the most loaded processor was used as the objective function to be minimized, i.e.:

$$\max_k\{\sum_{i=1}^{n}(t_{ik}x_{ik}+\sum_{l=1}^{m}\sum_{j=1}^{n}c_{ij}d_{kl}x_{ik}x_{jl})\}$$

The allocation problem was solved using enumerative search (e.g. branch-and-bound) [62, 152, 153, 182], heuristics tailored to the problem [73, 57, 210] and metaheuristics [158]. Observe that matrix of x_{ik} binary variables defines a mapping of processes to processors.

The mapping of a task graph into the processors and interconnection network is also called *embedding*. However, when talking about embedding a slightly different problem is considered in the literature than the allocation problem defined above. Let $G = (V, E)$ be a task graph and let $H = (\mathcal{P}, N)$ be a graph consisting of nodes representing processors, and set of edges N representing interconnection network. More precisely the embedding $< f, b >$ of G into H is one-to-one mapping f of the nodes from G to processors of H with mapping b of every edge $e = (u, v) \in E$ onto path $b(e)$ connecting $f(u)$ and $f(v)$. There are three main embedding cost functions considered in the literature [92, 110, 129]: *dilatation* which is the maximum length of any $b(e)$, *expansion* equal to $\frac{|\mathcal{P}|}{|V|}$, and *congestion* which is the maximum over $e' \in N$ of $\mid \{e \in E : e' \in b(e)\} \mid$ (i.e. the number of different paths $b(e)$ using the same edge e'). Embedding is a computationally hard problem in general. As one may note embedding a cycle graph G (a ring) into a graph H with $\mid V \mid = \mid \mathcal{P} \mid$ and congestion 1 is equivalent to HAMILTONIAN CIRCUIT [97] (cf. also [129]).

4.2 Load Balancing

Allocation of computations to processors described in the previous section is done off-line assuming knowledge of computation and communication costs. *Load balancing* is an approach which attempts to minimize application execution time by distributing the computations evenly among the processors. Furthermore, load balancing inherently considers on-line case in which full knowledge of the incoming task cannot be assumed. A task appearing (i.e. created) during the computation produces additional load which must be distributed to the processors. Note that there exists no on-line method producing a solution which is optimal off-line (i.e. with the full knowledge of tasks parameters) [192]. A first step to load balancing is *load sharing* whose goal is supplying each processor with at least some load.

Before implementing any load balancing method several problem areas must be addressed. A reliable and accurate measure of the processor load is required. For example, it can be the number of branch-and-bound tree nodes to be processed by the processor. It can be some estimate of the expected number of search tree nodes (or time) which can emerge from the nodes already assigned to the processor [151]. In [211] CPU utilization, memory utilization or average response time are suggested as load measures. On the other hand, it is demonstrated in [142] that sophisticated load measures are not more useful than the simple ones. Thus, in the following we assume that the number of data units assigned to a PE for processing is its load. Another problem are data dependencies. When there is little dependency between load elements it is possible to move them in an arbitrary fashion. However, in many practical applications such dependencies exist, for instance, while solving partial differential equations [209]. In such cases provisions must be made to avoid sparsely distributing related (e.g. mutually communicating) load elements.

Load balancing methods can be differentiated by the initiator of load balancing [104, 106]. It can be initiated by a processor which ran out of work (this is demand-driven approach) or it can be initiated by a processor on which a new load appeared (supply-driven method). Next, the decision about moving the load can be done globally using information about the whole system status [211], or this decision can be done locally in a distributed manner based on the locally available information [151]. Also intermediate forms are possible [122, 211]. The global approach has bad scalability and the central "load balancer" can easily become a bottleneck. On the other hand, due to the lack of information distributed approaches can result in imbalance. Finally, the amount of load to be transferred must be determined. As far as distributed methods are considered, there are two dominant ways of calculating this value: *nearest neighbor averaging* and *diffusion*. Let l_i denote a load of processor P_i before load balancing, l'_i after load balancing, and Δ_i the set of its neighbors. The nearest neighbor averaging intends to change the load such that it is equal to the mean load of the processor and its neighbors, i.e. $l'_i = \frac{l_i + \sum_{j\in\Delta_i} l_j}{|\Delta_i|+1}$. To achieve this goal processor P_i transfers to processor P_j amount of data equal to $(l_i - l'_i)\frac{q_j}{\sum_{j\in\Delta_i} q_j}$, where $q_j = \max\{0, l'_i - l_j\}$. In the diffusion approach processor P_i sends to P_j amount $\alpha(l_i - l_j)$ of data units, where $\alpha \in (0, 1)$. Thus, after a load balancing step processor P_i has $l'_i = l_i + \alpha \sum_{j\in\Delta_i}(l_i - l_j)$ data units.

4.3 Scheduling with Communication Delays

The class of problems we will call *scheduling with communication delays* can be denoted $P \mid prec, c_{ij} \mid C_{max}$ or $P \mid prec, c_{ij}, dup \mid C_{max}$. The considered problems can be described as follows. Given is a task set $\mathcal{T}$ with precedence constraints among its elements. Two tasks T_i, T_j linked by precedences communicate (e.g. the successor uses results of the predecessor). When both tasks are allocated to the same processor the communication time (data transfer time) can be neglected. Otherwise, the successor can be started only after communication delay c_{ij} following the completion of the predecessor. The optimality criterion is the schedule length. It can be allowed to *duplicate* tasks, i.e. to execute more than one instance of the same task on different processors. When duplication is allowed it is possible to avoid communication delays by producing data for the successors on multiple processors. For example, when the number of processors is not bounded, duplication allows building schedules without communication delays in $O(n)$ time for problem $\overline{P} \mid out-tree, c_{ij}, dup \mid C_{max}$ [66]. A special form of scheduling with communication delays is $P \mid prec, p_j = 1, c = 1 \mid C_{max}$ called UETUCT scheduling which stands for Unit Execution Time, Unit Communication Time.

Unfortunately, even for very restricted cases scheduling with communication delays is computationally hard. Not many polynomially solvable cases have been identified. This directs the research to efficient approximation algorithms. To our knowledge, there is no approximation algorithm with performance ratio better than 2 in the general case. Furthermore, it has been shown in [198] that no polynomial approximation scheme exists (unless **P=NP**) neither for $P \mid prec, p_j = 1, c = 1 \mid C_{max}$ nor for $\overline{P} \mid prec, p_j = 1, c = 1 \mid C_{max}$. The majority of works on scheduling with communication delays considered idealistic fully connected computer system which can transfer unlimited number of messages simultaneously. A particular interconnection type is considered in [94], but even here the scheduling problems are computationally hard. In Table 4.1 we present some important results in scheduling with communication delays. Yet, this branch of scheduling is rapidly evolving and the contents of the table can be found rather limited by an expert. More comprehensive study of the problem can be found in [67, 108].

Table 4.1: Results in scheduling with communication delays

Problem	Result	Reference
$P \mid prec, p_j = 1, c = 1 \mid C_{max}$	s**NP**h	[175]
$P \mid prec, p_j = 1, c = 1 \mid C_{max}$	$C^g_{max} \leq (3-\frac{2}{m})C^*_{max} - (1-\frac{1}{m})$	[175]
$P \mid pmtn, c = 1 \mid C_{max}$	$O(n)$	[174]
$P \mid pmtn, c \geq 2 \mid C_{max}$	s**NP**h	[174]
$P \mid prec, com \mid C_{max}$	$C^{ETF}_{max} \leq (2-\frac{1}{n})C'_{max} + C_{com}$	[124]
$\overline{P} \mid in{-}tree, c_{jk} \mid C_{max}$		
$c_{jk} \leq \min_j p_j$	$O(n)$	[64]
$\overline{P} \mid prec, p_j = 1, c, dup \mid C_{max}$	s**NP**h	[164]
$\overline{P} \mid prec, c_{j*}, dup \mid C_{max}$	$S \leq 2$	[164]
$\overline{P} \mid prec, c_{ij}, dup \mid C_{max}$		
"short" communication times	$O(n^2)$	[72]
$\overline{P} \mid in - tree, p_j = 1, c \mid C_{max}$	s**NP**h	[127]
$\overline{P} \mid in - tree, p_j = 1, c \mid C_{max}$		
complete k-ary intree	$O(n^2 \log n)$	[127]
$\overline{P} \mid prec, c_{ij} \mid C_{max}$	**NP**h	[65]
$\overline{P} \mid tree, c_{ij} \mid C_{max}$ unit depth tree	$O(n^2)$	[65]
$P \mid prec, p_j = 1, c = 1 \mid C_{max} = 4$		
bipartite precedence graph	s**NP**h	[198]
$P \mid prec, p_j = 1, c = 1 \mid C_{max} = 3$	polynomial	[198]
$\overline{P} \mid prec, p_j = 1, c = 1 \mid C_{max} = 6$	s**NP**h	[198]
$\overline{P} \mid prec, p_j = 1, c = 1 \mid C_{max} = 5$	polynomial	[198]
$P \mid prec, p_j = 1, c = 1 \mid C_{max}$		
fixed width of precedence graph	polynomial	[198]
$P \mid in{-}tree, p_j = 1, c = 1 \mid C_{max}$	s**NP**h	[198, 147]
$P2 \mid in{-}tree, p_j = 1, c = 1 \mid C_{max}$	$O(n)$	[109, 147]
$\overline{P} \mid out{-}tree, c_{ij}, dup \mid C_{max}$	$O(n)$	[66]
$\overline{P} \mid prec, c \leq 1 \mid C_{max}$	s**NP**h	[168]
$\overline{P} \mid prec, c \leq 1 \mid C_{max}$	$S_{SCT} = 1 + r$	[168]
$\overline{P} \mid out{-}tree, c_{jk} \mid C_{max}$	**NP**h	[67]
$P, bus \mid prec, p_j = 1, c = 1 \mid C_{max}$	s**NP**h	[94]
$Q2 \mid in{-}tree, p_j = 1, c = 1 \mid C_{max}$		
complete k-ary intree	$O(n)$	[29]

Used notation:
C^g_{max} - length of any greedy schedule, C^*_{max} - optimal length of the schedule, C'_{max} - length of the schedule without communication delays, $C_{com} \leq C'_{max} - 1$ - communication requirement over some chain of precedences, $r = \frac{\max_{i,j \in \mathcal{T}}\{c_{ij}\}}{\min_{i,j \in \mathcal{T}}\{t_j\}}$.

4.4 Loop Scheduling

The idea of a divisible task is tightly linked to parallelism of loops. Thus, we present the most important concepts for the case of loop scheduling. Moreover, loops are considered as the largest and most natural source of parallelism in many applications [101, 139, 155]. In many cases loops can be executed in parallel by different processors. The key problem is determining of the *chunk* size, i.e. the size of the load portion assigned to a processor in one step of data distribution. Two important factors must be taken into account: unpredictability of the actual loop execution time, and overhead related to the access to the work "distributor" (i.e. loop scheduler, note that loop index is a critical section). In the following we denote by t the total number of loops and m the number of processors. Below we present the most widely known ways of loop scheduling [128, 155].

Static Chunk

Each processor is assigned t/m loops to execute. This results in low overhead in accessing the loop scheduler but in bad load-balance among processors.

Self-Scheduling

Each processor is assigned one loop at a time and fetches a new iteration to perform when it becomes idle. This results in good load balance, but the overhead due to accessing scheduler is significant (proportional to the number of loops). A variation of self-scheduling is *chunk self-scheduling* in which a processor is assigned k loops at a time.

Guided Self-Scheduling

A processor requesting for a work is assigned $1/m$ of the remaining unassigned loops. This results in good balance and low overhead if the loops are uniform. When the loops are not so uniform assigning t/m loops to the first requesting processors may result in load imbalance. Furthermore, at the end of the computation processors are assigned one loop at a time which may result in contention while accessing the scheduler.

Trapezoid Self-Scheduling

The first assigned chunk of work has size N_s. The following chunks are decreasing linearly by some step d to the final size N_f. Example values of N_s and N_f can be $N/(2m)$ and 1, respectively. A disadvantage of this method is, that when m is big the difference between the chunks assigned to the consecutive processors as the first ones can be as big as md.

Factoring

Factoring is intended to achieve balanced workload. For this purpose, at each successive allocation the algorithm evenly distributes among the processors

half of the remaining iterations. Thus, $c_i = \lceil R_i/(2m) \rceil$ iterations are assigned to each processor in step i, where $R_1 = t$ and $R_{i+1} = R_i - mc_i$.
Affinity scheduling [155]
Affinity scheduling tries to take advantage of using local memory or cache. In contrast to the previous methods each processor has its own work queue. Thus, the need for synchronization is minimized. Initially, loops are divided into chunks of size $\lceil t/m \rceil$ and appended to each processor's queue. A processor executes $1/k$ (k can be m) of the loops remaining in its queue. When the queue becomes empty, the processor finds the most loaded processor (i.e. with the longest queue) removes $1/k$ of the iterations from that processor's queue and executes them.
Safe Self-Scheduling [128]
This scheme assigns statically the main portion of the loops and then balances the load by assigning the so-called smallest critical chores. More precisely, the processors are assigned statically $\alpha t/m$ (amount computed at the compile time) loops in the first batch. At runtime, the ith processor fetching some load is assigned $\max\{k, (1-\alpha)^{\lceil i/m \rceil} t\alpha/m\}$ loops. α is a crucial allocation factor. It is proposed to calculate it from the equation:

$$\alpha = \frac{1 + prob(E_{max}) + prob(E_{min})E_{min}/E_{max}}{2}$$

where E_{min} - minimum execution time of a loop, E_{max} - maximum execution time of a loop, $prob(x)$ - probability of executing a loop with time x. Thus, to use this method the execution time distribution must be known.

The methods of divisible task scheduling presented in Chapter 6 can be viewed as scheduling loops in distributed environment.

4.5 Communication Optimization

The basic commutation methods such as store-&-forward, circuit-switched, wormhole, virtual-cut-through have been described in Section 2.1. The field of communication optimization considers design of efficient algorithms of message *routing* geared to the considered data exchange operations and communication networks. A routing algorithm is a method of finding the way for a message in the network. In the following we describe what are the common data exchange operations and what is meant by efficiency of routing.

Most of commonly considered communications problems which are typical of many applications involve the following data exchange operations [116, 145, 177]:

One-to-one
There is at most one message to be sent from each processor and at most one message to be sent to each processor.
Broadcasting
Moving the same data unit from one processor to all others.
Gossiping
Moving a data unit from each processor to every other processor.
Scattering/Gathering
Scattering involves moving data from one processor to all others. Gathering consists in collecting data in one processor from all other processors. In scattering (gathering) every recipient (seder) receives (sends) a different piece of data.
Multiscattering/Multigathering
This operation consists in scattering (gathering) from (in) every node.

The routing algorithms should be best possible. Thus, an optimality criterion must be defined. A natural criterion is the total time required to route the messages to their destinations. A special form of this can be minimization of the number of data transfers, especially in networks with big startup times. Yet, there are also other criteria [145]. In communication network a deadlock or a leavelock (starvation) may arise. A deadlock is a situation in which some messages cannot move because they mutually block required resources (e.g. buffers or channels). A leavelock is a situation in which messages can move, but some messages cannot make progress toward their destination (e.g. are repetitively blocked or discarded). Avoiding deadlocks and leavelocks is a prerequisite to the routing algorithm feasibility. When messages are blocked or can be dropped, one may want to maximize the number of messages successfully transferred in a given period of time (maximizing throughput). In applications where hot spots (locations exceptionally often visited) can arise, e.g. involving gathering, the impact of routing to hot spots on messages with other destinations should be minimized. When messages are buffered one would minimize the size of the buffers used. Finally, we may want to minimize the size of the network (number and capacity of switches, wires etc.) and make the routing algorithm reliable enough to respond to network faults.

The communication problems most often considered in Chapter 6 are scattering and gathering.

4.6 Problems in Implementing Scheduling Models

The above approaches to scheduling in parallel computer systems, despite many successful implementations, have some disadvantages. The drawbacks are results of different assumptions made for each method. In a system not satisfying the presupposed assumptions some approaches can be hardly applied or the results can be far from optimal. The assumptions are related to the following issues:
- who is scheduling: the application or the operating system,
- when is scheduling done: in the pre-runtime or during the runtime,
- how much information is necessary for scheduling.

However, these differentiations have deeper origins. In our opinion, these are assumptions on necessary data about the application. Two features are related to data: availability, and complexity.

When availability of extensive knowledge about the application(s) (i.e. task system) is assumed then only the programmer or compiler are able to collect such data. Hence, the operating system has no necessary knowledge and scheduling is made by the application itself. For example, allocation models require extensive information about execution times and communication patterns of the program modules. In the case of scheduling with communication delays the precedence DAG is highly data-dependent. Thus, the actual DAG is known rather after execution than before. Hence, schedules or allocations based on the information collected in some previous experimental runs can miss the optimum when the new data sets are very different. It can be observed that the above two approaches are rather application-oriented and give little help in scheduling by operating system.

Another problem related to available data is complexity. This has two aspects: complexity of data structures and schedule optimization complexity (cf. Section 3.2). For instance, task graphs and precedence DAGs for realistic problems are huge and can comprise thousands of nodes. The operating system scheduler cannot afford wasting its space and time for dealing with so big and complex structures. Furthermore, detailed setting of the problem often results in its **NP**-hardness. For such problems optimal solutions cannot be expected in low-order polynomial time and only small size problems can be solved in short time. Hence, such problem settings can be analyzed and optimized rather off-line in the pre-runtime scheduling than during runtime.

The approaches to scheduling presented in this work are compared in

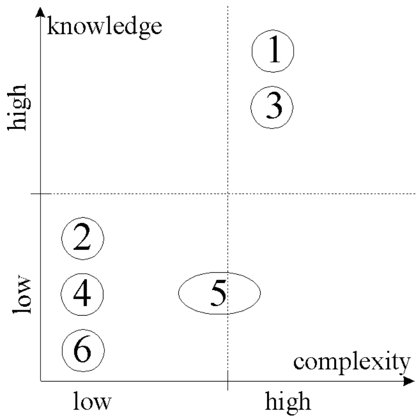

Figure 4.1: Comparison of scheduling methods. The numbers denote: 1-allocation, 2-load balancing, 3-scheduling with communication delays, 4-loop scheduling, 5-multiprocessor task scheduling, 6-divisible task scheduling.

Fig. 4.1 with regard to the described data features. This qualitative comparison reflects only author's opinion on general characteristics of these methods, not all their possible settings. A method was considered as a high-knowledge when precedence DAG or task graph of application is essential to applying it. Complexity was considered low when low-order complexity algorithms (class **P**) can be applied and data structures are small. Complexity was considered high when the method is high-knowledge and the problem setting is **NP**h.

Finally, let us observe that required precision of data is a result of simplifications made in viewing the scheduling problem. Of course, precision contributes to the complexity and availability of data. Note that all the above methods address (successfully) some restricted areas, while disregard other. For instance, allocation models do not consider sequencing of tasks and communications. Thus, a deadlock in communication system can be unpredictable with these models. Load balancing rarely considers restrictions imposed by communication between the balanced elements of the computation. Scheduling loops, though addresses an important area of parallelism, is restricted to a particular class of applications and architectures. The methods we present in the following sections, undoubtfully, are not panacea. Yet, we hope that they improve presentation and solvability of real problems.

Chapter 5

Multiprocessor Tasks

In this chapter we consider scheduling of multiprocessor tasks. Section 5.1 motivates the use of multiprocessor tasks. Section 5.2 is devoted to scheduling multiprocessor tasks on parallel processors, and Section 5.3 to scheduling on dedicated processors.

5.1 Why Multiprocessor Tasks?

A proper scheduling strategy is an indispensable element of an efficient parallel computer system. As it was observed in [133] it cannot be substituted for. Many distributed and multiprocessor computer systems offer some kinds of parallelism. It is not so evident, however, that the concurrency of the application execution is guaranteed, especially in the extreme load conditions. In this section we are going to explain that for many reasons real concurrency should be provided.

Consider a general purpose computer system with time sharing. A parallel application which consists of many concurrent threads is run on a number of processors. The access to a critical section is guarded by a lock which must be acquired by threads using the section. Imagine a situation in which one of the threads captures the lock. It must compete for the processors with an uncertain number of other threads. Soon it can lose its processor. Then, other threads (of the same application) must busy-wait for the release of the lock. This, however, will not happen as long as the thread which is in the possession of the lock is not running. Thus, a bad decision about scheduling such a critical thread results in a significant performance degradation. One conclusion from this example is that busy-waiting threads should not be executed in time quanta when the thread holding a lock is not running.

Now, consider a different situation. The threads of the same application communicate with each other but are run in different time quanta. One thread tries to communicate with some other thread. It sends the data, but since the other thread is descheduled, it must wait at least until the end of the time quantum. Hence, big part of the first thread time quantum is lost for busy-waiting. In one of the following time quanta the receiving thread obtains the data, processes it and sends back the results. The results must still wait for the first thread to start running in order to receive the data.

In both of the above examples the progress in computation depends on the speed of context switching rather than on the raw speed of the processors or the communication system. *Coscheduling* is a scheduling policy proposed to avoid these difficulties [98]. Coscheduling consists in granting simultaneously (in the same time quantum) the processors to the threads of the same application. It has been demonstrated in [213] that coscheduling performs quite well in a wide range of conditions and for various models of parallel applications. In [93] the performance of a parallel application using barrier synchronization was studied. The coscheduling policy (called here *gang scheduling*) has been compared, both theoretically and in practice, with blocking. In blocking the thread releases a processor as soon as it completes its share of the work. For coarse-grain parallelism blocking performs well. For fine-grain parallel application coscheduling is better. Thus, coscheduling is postulated in parallel systems. Observe that coscheduled applications occupy several processors at the same moment of time and as such are multiprocessor tasks.

The parallel applications are very often represented by DAGs. Yet, this kind of representation has a limited applicability for the operating system. Contemporary parallel applications have DAGs with thousands of nodes. At the current state of technology, it is hard to imagine a scheduler (one of the most often executed parts of the operating system) able to handle, analyze and optimize so big structures. Furthermore, threads of an application are indistinguishable for the scheduler. Hence, without additional information from the application the scheduler is not able to give a priority to important threads (e.g. holding a lock) [213]. Note that since the DAG is highly data-dependent it can be precisely known after the execution rather than before. From the above we conclude that it would be reasonable for the operating system to control only the number of processors granted to a parallel application and leave the control of the threads to the application (i.e. to the compiler and the programmer).

Following these propositions a number of massively parallel computer systems divide their processors into *partitions* [47, 75] (cf. Section 2.2). The main idea of a partition is to give an exclusive access to a number of processors to one application only. The operating system is responsible for managing the partitions, granting the access to them etc. Note that from the viewpoint of the partition manager the applications are multiprocessor tasks because they occupy all the processors within the partition at the same moment of time.

In computer control systems a high level of reliability is often achieved by executing redundant copies of the program on different processors and voting on the final control decision [8, 98, 121]. Applications of this kind are multiprocessor tasks because more than one processor is simultaneously occupied.

In the preceding discussion we concentrated on parallel processors. Now, we are going to demonstrate that multiprocessor tasks scheduling is also applicable in the case of dedicated processors. The main idea behind dedication of processors is their specialization. Hence, in massively parallel computers there are processing nodes equipped with communication and other I/O hardware (e.g. disks), while other processing nodes are not equipped with such devices. There can be nodes with arithmetic, vector, graphic, signal processing facilities, while there can be other nodes without them. It is not inconceivable to present parallel applications requiring a little bit of all these facilities. Thus, a multiprocessor task can be considered also in the case of dedicated processors.

The multiprocessor task concept originated from scheduling tests in multiprocessor computer systems [130], testing VLSI chips [74] or other devices [81]. Testing of processors by one another requires at least two processors simultaneously. Due to the fact that the graph of mutual tests cannot be arbitrary to guarantee testability of the system [113, 171] one may regard a test as a dedicated task. A similar situation takes place in testing VLSI chips where some functional units are required to test other units [74]. Another application for multiprocessor task scheduling in dedicated environment can be scheduling of file transfers [70]. A file transfer requires at least two "processing" elements simultaneously: the sender and the receiver. Simultaneous transfers on multiple buses can be also considered as multiprocessor tasks [126]. Finally, simultaneous execution of multiple instructions in a superscalar processor requires matching instructions in such a way that the sets of simultaneously required processor units do not intersect. Scheduling in this case is performed by a compiler or by the hardware of the processor. Not only can the specialization of the processing elements justify considering processors as dedicated. A multiprocessor task can be regarded as executed by a dedicated processor also for the preallocation reasons. For a certain communication pattern among the tasks of a parallel application and a given communication network it can be advantageous to map tasks to processors in some fixed way. Changing the preallocation may increase the communication overhead (due to dilatation, congestion etc.). In some cases even parallel processors may behave as dedicated devices. For example, since the costs of filling a pipeline, vector registers or a cache are high it is disadvantageous to frequently transfer threads to new sites. Hence, there is a kind of affinity between tasks and processors [155].

Though we introduced multiprocessor tasks in the computer context it is not difficult to find application for this kind of scheduling in production systems. In fact, the first papers considering an idea of simultaneous execution of a task by many processors dealt with scheduling operations in chemical plants [48] and project scheduling [200].

5.2 Parallel Processors

In this section we first review the subject literature including earlier works of the author. Then, we present new results especially for scheduling with variable profile. Table 5.1 summarizes the results mentioned in this section.

5.2.1 Overview of Earlier Results

We survey here scheduling multiprocessor tasks on parallel processors (cf. also [87, 199]). One of the first papers considering multiprocessor task scheduling was [149] in which Unit Execution Time (UET) tasks were considered. It was shown that problems $P \mid size_j, p_j = 1 \mid C_{max}$ and problem $P3 \mid size_j, p_j = 1, prec \mid C_{max}$ with $size_j \in \{1, 2\}$ are s**NP**h (reduction from $P \,||\, C_{max}$ and $P \mid p_j = 1, prec \mid C_{max}$, respectively). The performance of any list scheduling heuristic (LS in short) has been proved to be bounded from above by $(2m - \Delta)/(m - \Delta + 1)$ and the ratio of $\lfloor(2m - \Delta)/(m - \Delta + 1)\rfloor$ has been achieved. For problem $P2 \mid size_j, p_j = 1, prec \mid C_{max}$ an algorithm with complexity $O(n^{\log_2 7})$ has been proposed. It is based on building the transitive closure of precedence constraints graph (hence the complexity [95]) and Coffman-Graham algorithm [71].

Problem $R \mid spdp\!-\!lin, pmtn, var, r_j, d_j \mid -$, i.e. a decision problem consisting in verifying existence of a feasible schedule, has been reduced in [200] to solving a linear program with $O(n^3)$ variables and $O(n^2)$ constraints.

In [203] problem $P \mid spdp{-}lin, pmtn, var, r_j, d_j \mid X$ is analyzed. For each task a deadline is given. All the tasks must be completed in time. To achieve such goal any processor is capable of increasing its speed. The optimality criterion is not standard. Firstly, the optimality criterion is minimizing the maximum speed necessary for processing the tasks. Secondly, when processing at some speed is unavoidable then the period of processing at such a speed is minimized. In [202] a similar problem is considered, but the optimality criterion is the total intensity cost, where the cost function is convex.

Problem $P \mid spdp{-}lin{-}\delta_j, pmtn, var, r_j \mid L_{max}$ has been considered in [201]. This problem can be solved by a reduction to a sequence of equivalent maximum network flow problems which check the existence of a feasible schedule for a given value of L_{max}. Let the events in the task system (i.e. r_j and $d_j + L_{max}$ for $j = 1, \dots, n$) be sorted such that event $e_i \leq e_{i+1} (i = 0, \dots, 2n-1)$. In the network a vertex representing the task is connected with a vertex representing interval $[e_i, e_{i+1}]$ when the task can be executed in this interval. The capacity of such an edge is $(e_{i+1} - e_i)\delta_j$. The task vertices are connected with the source vertex by an edge with capacity t_j^1, and the interval vertices are connected to the sink vertex by edges of capacity $m(e_{i+1} - e_i)$. The network comprises $O(n)$ vertices. It is shown in [202] that $O(\sum_{i=1} \delta_j) \leq O(nm)$ calls to $O(n^3)$ network flow algorithm are required. This results in a total complexity of $O(n^4 m)$.

In [32] preemptive and nonpreemptive scheduling of multiprocessor tasks is considered. This paper extends preliminary results of [46]. For problem $P \mid size_j, p_j = 1 \mid C_{max}$ and $size_j \in \{1, \Delta\}$ an $O(n)$ algorithm has been proposed. When the numbers of simultaneously required processors are in the set $\{1, \dots, \Delta\}$ and Δ is fixed the above problem can be solved in $O(n)$ time by integer linear programming (ILP) with fixed number of variables. Problem $P \mid size_j, p_j = 1 \mid C_{max}$ has been shown to be s**NP**h in general (reduction from 3-Partition). It has been shown in [32] that among the optimal schedules for problem $P \mid size_j, pmtn \mid C_{max}$ where $size_j \in \{1, \Delta\}$ there must be a so-called *A-schedule*. In the A-schedule tasks with $size_j = \Delta$ are assigned in the interval $[0, C_{max}]$ using McNaughton's wrap-around-rule [157], and tasks with $size_j = 1$ are assigned in the same way in the remaining part of the schedule. Then, an algorithm with complexity $O(n)$ building *A-schedules* has been proposed for problem $P \mid size_j, pmtn \mid C_{max}$ where $size_j \in \{1, \Delta\}$. When the numbers of simultaneously required processors are from larger than two-element set, an algorithm based on linear programming and the concept of a *processor feasible set* has been proposed. The processor feasible set is a set of tasks that can be executed in parallel on a given number of processors. Note that for n tasks and m processors there are $O(n^m)$ processor feasible sets. We can denote by x_l the processing time of l−th processor feasible set. Then, problem $Pm \mid size_j, pmtn \mid C_{max}$ boils down to solving a linear program: minimize the sum of execution times of all processor feasible sets, subject to the sum of processing times for processor feasible sets containing T_j being not smaller than $t_j^{size_j}$ $(j = 1, \ldots, n)$. Such a formulation has $O(n^m)$ variables, n constraints, and can be formulated and solved in polynomial time, provided m is fixed.

In [41] problem $P \mid size_j, pmtn, res1 \cdot 1 \mid C_{max}$ was considered. Additionally, it was assumed that $size_j \in \{1, 2\}$, all tasks with $size_j = 1$ required a unit of the resource, while only some tasks with $size_j = 2$ required the resource. It was shown that among the optimal schedules there must exist an A-schedule (called here *normalized*). An $O(n \log n)$ algorithm has been proposed. This problem was further analyzed in [43] for $\Delta > 2$ and tasks with $size_j = 1$ requiring a unit or no resource. An $O(nm)$ algorithm was given. For problem $Pm \mid size_j, pmtn, res \cdot \cdot \mid C_{max}$ a method based on linear programming and feasible sets of tasks was proposed.

The problem of preemptive scheduling multiprocessor tasks on hypercube multicomputer, i.e. problem $P \mid cube_j, pmtn \mid C_{max}$, has been tackled in [58]. An $O(n^2)$ algorithm has been proposed to test whether a feasible schedule of length T exists. The algorithm builds a *stair-like* schedules. A schedule is stair-like when (i) each processor P_j is busy before time $f(P_j)$ and idle after $f(P_j)$, (ii) f is nonincreasing function of processor number. Thus, the highest step is at processor 0 (the most loaded machine), the steps gradually decrease till the lowest step at the least loaded processor(s). Tasks are scheduled in the order of decreasing $size_j$. A task is executed in such a way that it ends at the common deadline T, steps of the stair-like schedule are consecutively filled from left to right and no sooner is the new (less loaded) step used than the current one is completely full. This results in $O(n^2)$ preemptions. The testing algorithm can be applied in time $O(n^2(\log n + \log \max_j\{t_j^{size_j}\}))$ to find the optimal schedule. Note that in such a case C_{max} is calculated with unit granularity.

In [90] the authors proved that problems $P2 \mid size_j, chain \mid C_{max}$, $P5 \mid size_j \mid C_{max}$ are s**NP**h. Each schedule for $P2 \mid size_j \mid C_{max}$, $P3 \mid size_j \mid C_{max}$ can be transformed into a *canonical schedule* and dynamic programming method can be used to obtain an optimal schedule. The complexity of problem $P4 \mid size_j \mid C_{max}$ remains open. The preemptive scheduling has been shown in [90] to be **NP**h for $P2 \mid spdp\text{–}any, pmtn \mid C_{max}$ and s**NP**h for $P \mid spdp\text{–}any, pmtn \mid C_{max}$. For $Pm \mid spdp-any, pmtn \mid C_{max}$, i.e. when the number of processors is fixed, a dynamic program has been given.

In [119] an $O(n \log n)$ algorithm testing the existence of a schedule for problem $P \mid cube_j, pmtn \mid C_{max}$ has been proposed. The algorithm differs from the one from [58] in the use of *pseudo-stairlike* schedules. In the pseudo-stairlike schedule a task is not filling "steps" one by one, but fills at most two subcubes ("steps") one from the moment it becomes available till the end of the schedule and possibly one more but only partially. This results in a lower number of $O(n)$ preemptions. The algorithm can be applied in time $O(n \log n(\log n + \log \max_j\{t_j^{size_j}\}))$ to find the optimal schedule by a binary search. A similar approach has been proposed independently in [2].

In [208] problem $P \mid spdp-any, pmtn, r_j \mid L_{max}$ is considered. The number of processors is assumed to be big enough to deal with them as with a continuous medium. Each task is described by a continuous function binding the processing speed and the number (amount) of assigned processors. Qualitative conclusions are derived. The problem is reduced to a set of nonlinear equations.

Problem $P \mid cube_j, pmtn, r_j, d_j \mid -$, (i.e. verification of the existence of a feasible schedule) was reduced in [169] to a linear program with $O(mn^2)$ variables and $O(n^2m^2)$ constraints.

In [37] problem $Q \mid size_j, pmtn \mid C_{max}$ of scheduling on uniform processors was considered. It was also assumed that $size_j \in \{1, 2\}$ and that processors form pairs of equal speed. A proposed $O(n \log n + nm)$ algorithm was inspired by [103]. First, a lower bound of the schedule length is calculated. Next, tasks with $size_j = 2$ are scheduled in the order of decreasing processing times. Then, in the remaining free intervals tasks with $size_j = 1$ are scheduled. When the schedule is to short to accommodate all the tasks it is extended by a calculated amount of time. The idea of the above algorithm was extended to solve problem $Q \mid size_j, pmtn \mid C_{max}$ with $size_j \in \{1, \Delta\}$ in [39], and to solve problem $Q \mid cube_j, pmtn \mid C_{max}$ in [38]. In the latter problem tasks are scheduled in the order of decreasing $size_j$ and tasks with the same $size_j$ in the order of decreasing $t_j^{size_j}$. In [84] results of computational experiment on the above algorithm for $Q \mid cube_j, pmtn \mid C_{max}$ have been reported. For problem $Qm \mid size_j, pmtn \mid C_{max}$ with arbitrary $size_j$ a solution based on linear programming and feasible sets has been proposed in [38].

Heuristics for scheduling $P \mid spdp-lin\text{–}\,\delta_j, prec \mid C_{max}$ have been proposed in [205]. It has been proved that any LS algorithm has tight performance ratio $\Delta+\frac{m-\Delta}{m}$. The standard LS algorithm assigns a task to the first available processor. However, it can be advantageous to delay the start of a task until a moment when more processors are available. The Earliest Completion Time (ECT) heuristic is an LS algorithm which assigns tasks to ready processors in a manner minimizing their completion times. The worst case performance ratio of ECT has been shown to be not worse than $\ln \Delta + 1$. Further analysis of ECT in [206] proved that the performance is bounded by $3 - \frac{2}{m}$ and an instance with 2.5 performance ratio was demonstrated. The idea of ECT algorithm has been independently proposed in [7] to find the set of processors executing a computationally intensive task on a distributed workstation system. Processors become available at different moments after completion of earlier task(s) (this is a variation of problem $Q, win \mid spdp\text{–}any \mid C_{max}$ with $n = 1$). An algorithm with complexity $O(m^2)$ has been proposed in [7] and with complexity $O(m \log m)$ in [91].

Preemptive scheduling on a hypercube was considered again in [183]. A feasibility testing algorithm of [2] was modified to obtain complexity $O(nm)$. By the observation that in the optimal schedule at least one task must use all the time remaining up to the end of the schedule, an $O(n^2m^2)$ algorithm finding optimal schedule for $P \mid cube_j, pmtn \mid C_{max}$ was given. The above algorithm uses for each subcube a parametric representation of the remaining processing time as a linear function of some (hypothetical) common deadline T. The parameters are modified as a result of building partial schedule.

For nonpreemptive scheduling on a hypercube a Largest Dimension Longest Processing Time (LDLPT) heuristic has been analyzed in [59]. The LDLPT heuristic is a LS method assigning task to processors in the order of (primarily) decreasing $size_j$ and (secondarily) decreasing processing time. The tight performance ratio of LDLPT is $2 - 2/m$.

A variation of $P \mid spdp\text{–}any \mid C_{max}$ was considered in [131]. It was assumed that $n \leq m$, all tasks (at least initially) are executed in parallel, and minimization of C_{max} was achieved by changing the number of processors used by the tasks. An approximation algorithm proposed in that paper was successively increasing the number of processors used by the longest task until δ_j was achieved or all m processors were occupied. We call this algorithm Varying Size (VS). The tight performance ratio of VS is $\min\{n, R/(1 - m/n)\}$, where R is the maximum of the ratio of two successive acceptable sizes of any task. A similar idea was used in an approximate algorithm with performance ratio R for problem $P \mid spdp\text{–}any, pmtn \mid C_{max}$ [132].

In [40] problem $Pm \mid size_j, pmtn \mid L_{max}$ for $size_j \in \{1, \Delta\}$ was considered. A linear program based on the processor feasible sets was proposed. Since this method requires an LP with big number of variables an approximation method based on tabu search and linear programming has been proposed. The reported good experimental results for the second method have been explained by a particular topology of the criterial function.

In [214] the feasibility algorithm of [2] has been modified to obtain $O(n^2 \log^2 n)$ algorithm finding optimal schedule for $P \mid cube_j, pmtn \mid C_{max}$. Again, the observation is used that in the optimal schedule some task must be scheduled exploiting all the remaining processing time on some subcube. To calculate the testing values of C_{max} a parametric representation of the remaining free processing time on processors is used (cf. [183]). The optimum C_{max} is found by considering tasks in the order of decreasing $size_j$ and testing the calculated schedule lengths for successively increasing subcube numbers. For nonpreemptive scheduling the authors propose a Largest Dimension First (LDF) heuristic with tight performance ratio $2 - 1/m$. For on-line scheduling (i.e. the set of tasks is not known a priori) an instance is demonstrated for which LDF has performance ration greater than $1 + \sqrt{6}/2$.

In [133] an experimental study is reported for on-line scheduling for problem $P \mid cube_j \mid \sum c_j$. It is observed that even sophisticated processor allocation strategies alone cannot guarantee good performance. A set of *Scan* strategies is proposed which combine the simple buddy allocation scheme with clustering tasks according to their $size_j$. Tasks with the same $size_j$ are appended to one queue. Queues with different size tasks are scanned in the direction of increasing (or decreasing) size. This strategy effectively overcomes the shortcomings (e.g. weak ability to recognize idle subcubes) of the buddy allocator.

In [148] application of LPT heuristic to problem $P \mid size_j \mid C_{max}$ is considered. The performance ratio is proved to be $\frac{4}{3}k - \frac{k(k+1)}{6m}$, where k is the number of different task sizes. For problem $P \mid cube_j \mid C_{max}$ LPT has performance ratio not greater than $2 - 1/m$ and performance $2 - 2/m$ has been demonstrated.

In [181] various special cases solvable in polynomial time for $P \mid spdp-any \mid \sum c_j$ are analyzed. For $P \mid spdp-lin \mid \sum c_j$ an SPT rule is proved to be optimal.

Problem $P \mid spdp\text{–}any, prec \mid C_{max}$ was considered in [170]. An algorithm for the determining of $size_j$ and sequencing of multiprocessor tasks being elements of an arbitrary DAG has been proposed. Processors are considered here as a continuous medium which behaves like electrical charge passing from one task to another in the DAG. The optimality conditions impose a set of nonlinear equations on the flow of processing power (processors) and on the completion times of independent paths of execution. These equations are analogous to Kirchhoff's laws of electrical circuit theory. An algorithm based on conjugate gradient method has been proposed. The complexity is $O(e^2+ne+I(n+e))$, where e - the number of edges in the precedence graph and I - number of iterations in the algorithm.

A similar problem motivated by computer vision application is considered in [63] (a variation of $P \mid spdp\text{–}any, prec \mid C_{max}$). The important difference is that the computations are pipelined and the tasks constantly coexist on the processor set. Long sequences of data sets undergo processing by a collection of tasks forming series-parallel DAG. The throughput defined as the longest execution of a single task in the DAG is the interval between obtaining results for two consecutive data sets (a "clock" of the pipe). Two problems are posed: for the given throughput find minimal response time, and for the given response time find maximal throughput. Heuristic algorithms are proposed with complexity $O(nm^2)$ for the first problem and $O(nm^2 \log m)$ for the second one.

In [85] problem $P \mid size_j, pmtn \mid C_{max}$ is proved to be **NP**h, but it is an open problem whether it is s**NP**h. Also problem $P4 \mid size_j \mid C_{max}$ is proved to be solvable in pseudpolynomial time provided that no task is uniprocessor.

In [89] a special case of problem $P \mid spdp-lin-\delta_j, var, chain \mid C_{max}$ is considered. It is assumed that tasks form chains of three elements (denoted $\mid chain \mid= 3$): a sequential head ($size_j = \delta_j = 1$), parallel central part with an unbounded linear speedup ($spdp-lin, \delta_j > m$) and a tail which is sequential again. This model was motivated by a master-slave model of computations. It was shown that the above preemptive scheduling problem is s**NP**h. This result leads to a conclusion that preemptive scheduling of tasks with linear speedup and a given execution profile (cf. Section 3.3) is s**NP**h. Yet, the optimal schedule for the $m-1$ longest tasks can be extended to an optimal schedule for all the tasks. Furthermore, when m is fixed such a schedule can be obtained in polynomial time. When the chain of tasks consists of two elements of one type only, e.g. there are only heads and a central (parallel) part (denoted $\mid chain \mid= 2$), then the optimal solution can be found in $O(n \log n)$ time. Three approximation algorithms have been proposed with tight performance bounds $3, 2, 2$, respectively.

In [45] problem $P3 \mid size_j, p_j = 1, chain \mid C_{max}$ is proved to be s**NP**h. Low-order polynomial time algorithms are proposed in special cases. When the chain consists of two concatenated subchains: a leading chain of multiprocessor tasks with $size_j = \Delta$ and trailing chain of uniprocessor tasks an $O(n \log n)$ algorithm can be applied when $2\Delta > m$. Chains of this kind are called monotonically decreasing chains (m.d.-chains in short). For uniform chains (u.-chains in short) consisting either of multiprocessor tasks with $size_j = \Delta$ or uniprocessor tasks the optimal schedule can be found in $O(n \log n)$ time.

The problem of nonpreemptive scheduling multiprocessor tasks on two processors for $\sum w_j c_j$, $\sum c_j$ and L_{max} criteria is considered in [144]. It is shown that $P2 \mid size_j \mid \sum w_j c_j$ is s**NP**h. A dynamic programming procedure is proposed for a fixed number of duoprocessor tasks. Problem $P2 \mid size_j \mid \sum c_j$ is **NP**h even when there is only one duoprocessor task. A heuristic is proposed for the first problem with tight performance bound 2 which decreases to $\frac{3}{2}$ for problem $P2 \mid size_j \mid \sum c_j$. $P2 \mid size_j \mid L_{max}$ is shown to be s**NP**h and a dynamic program is proposed for a fixed number of duoprocessor tasks. $P2 \mid p_j = 1, size_j \mid L_{max}$ is shown to be polynomially solvable by an extension of EDD (Earliest Due Date) rule. Using this rule for arbitrary processing times gives an algorithm with bound $L^{H2}_{max} \leq L^*_{max} + \frac{1}{2}\sum_{j \in \mathcal{T}^2} t_j$, where $\mathcal{T}^2$ is the set of duoprocessor tasks.

5.2.2 $P \mid spdp\!-\!lin\!-\!\delta_j, var \mid C_{max}$

Let us start this section with an observation that $P \mid spdp\!-\!lin, var \mid C_{max}$, unlike problem $P \mid spdp\!-\!any, pmtn \mid C_{max}$, is trivially solvable in $O(n)$ by executing tasks on all available processors. Now, we show that problem $P \mid spdp\!-\!lin\!-\!\delta_j, var \mid C_{max}$ is solvable in polynomial time.

Theorem 5.1 *Problem $P \mid spdp\!-\!lin\!-\!\delta_j, var \mid C_{max}$ can be solved in $O(n)$ time.*

Proof The algorithm for this problem is an extension of McNaughton's wrap-around-rule [157]. First, we calculate the length of the schedule

$$C^*_{max} = \max\left\{ \max_{T_j \in \mathcal{T}}\{\frac{t^1_j}{\delta_j}\}, \frac{\sum_{j=1}^n t^1_j}{m} \right\} \tag{5.1}$$

Note that the schedule cannot be shorter because the first term denotes the longest execution time of a single task, and the second term is the total amount of work evenly distributed among the processors. We will show that a feasible schedule of length C^*_{max} exists. The schedule can be built by applying McNaughton's wrap-around-rule: Task T_1 is scheduled starting at time 0 on processor P_1. When $t_1^1 \leq C^*_{max}$, processing of the task T_1 finishes at time t_1^1 on processor P_1. In the opposite case amount $t_1^1 - C^*_{max}$ of the remaining work is executed on processor P_2. If $t_1^1 - C^*_{max} > C^*_{max}$ the procedure is repeated and the excess of the work is processed by processor(s) $P_3, (P_4, \ldots)$. Right after the completion of task T_1 processing of T_2 starts. When the task does not fit completely on the processor where it was started the rest is processed by the following processor(s). This procedure of wrapping-around is repeated until the last task. Now, let us consider the feasibility of the schedule. The second term determining C^*_{max} guarantees that the capacity of the box of m processors in time C^*_{max} is not exceeded. We have to ensure that no task uses more than δ_j processors. Assume that task T_{j-1} finished at time x on processor P_i. For T_0, $x = 0$ and $i = 1$. Suppose task T_j uses more than δ_j processors. If this is true then in the whole schedule length task T_j uses at least δ_j processors and at moment x^+ (infinitesimally after x), the task uses $\delta_j + 1$ (or more) processors. This means that T_j occupies more than $\delta_j C^*_{max}$ which contradicts $C^*_{max} \geq \max_{T_j \in \mathcal{T}}\{\frac{t_j^1}{\delta_j}\}$. Hence, the schedule is feasible. Let us analyze the time in which the schedule can be obtained. Task T_j finishes on processor $\lfloor (t_j^1 + x)/C^*_{max} \rfloor + i$. The completion time of T_j is $x + t_j^1 - C^*_{max}\lfloor (t_j^1 + x)/C^*_{max} \rfloor$. The processors used by a task and the intervals of the task processing can be found in constant time. Thus, the whole schedule can be built in $O(n)$ time. □

5.2.3 $P \mid spdp{-}lin{-}\delta_j, var, r_j \mid C_{max}$

In this section we present an $O(n^2)$ algorithm for problem $P \mid spdp{-}lin{-}\delta_j, var, r_j \mid C_{max}$ [86]. The procedure we propose uses some concepts of the Muntz and Coffman algorithm [159] solving problem $P \mid pmtn, in-tree \mid C_{max}$. That algorithm schedules tasks according to their level which is the time required to finish all the tasks along the path from the given task to the root of the tree. Furthermore, an idea of *processing capabilities* was introduced in [159]. Processing capabilities are real numbers representing a fraction of all m processors which is assigned to process a task for some time. Processing capabilities can be considered as speeds of processing tasks. We will use a similar method to assign processors to tasks. A technique proposed in the previous section for problem $P \mid spdp{-}lin{-}\delta_j, var \mid C_{max}$, is applied to schedule pieces of tasks. The main idea behind the algorithm is to build a schedule starting from the interval where only one task is ready and ending with the last interval where all tasks are ready. The more tasks are executed before the last interval, the smaller C_{max} is. Now, we introduce some additional notation.

Height $h(j)$ of task T_j is the shortest time required to complete T_j. $h(j)$ is equal to the required remaining processing divided by δ_j. $h(j)$ is $\frac{t_j^1}{\delta_j}$ initially, and decreases while processing T_j. Hence, $h(j) = 0$ means that T_j is finished. Two tasks are said to be equal if their heights are equal. We assume that there are $l \le n$ different values of ready times, and $r_1 = 0 < r_2 < \ldots < r_l$. We introduce also $r_{l+1} = \infty$. In the algorithm we will denote by:

k - index of interval $[r_k, r_{k+1}]$, $k = 1, \ldots, l$,
Q_k - set of tasks ready in interval k,
τ - length of the current processing capabilities assignment,
t - beginning time of the current processing capabilities assignment;
$\overline{\beta}$ - a vector of n processing capabilities for tasks $T_1, \ldots, T_n$.

An Algorithm for $P \mid spdp{-}lin{-}\delta_j, var, r_j \mid C_{max}$
1: $t := 0$; group tasks with ready time r_k in set Q_k; order tasks in Q_k according to nonincreasing heights, $k = 1, \ldots, l$;
2: **for** $k := 1$ **to** l **do**
begin
2.1: order tasks in Q_k according to nonincreasing values of $h(j)$ for $T_j \in Q_k$;
2.2: **while** $(r_{k+1} > t)$ **and** $(\exists_{T_j \in Q_k} h(j) > 0)$ **do**
begin
2.2.1: CAPABILITIES$(Q_k, \overline{\beta})$;
2.2.2: calculate times:
if $\exists_{T_j, T_{j+1} \in Q_k} h(j) > h(j+1)$ **then**
$\tau' := \min_{T_j, T_{j+1} \in Q_k} \{ \frac{h(j) - h(j+1)}{\frac{\beta_j}{\delta_j} - \frac{\beta_{j+1}}{\delta_{j+1}}} : \frac{\beta_j}{\delta_j} \ne \frac{\beta_{j+1}}{\delta_{j+1}}, h(j) > h(j+1) \}$
else $\tau' := \infty$
- the shortest time required for two tasks T_j, T_{j+1} with different heights to become equal;
$\tau'' := \frac{h(|Q_k|)}{\delta_{|Q_k|}}$ - the time to the earliest completion of any task;
2.2.3: $\tau := \min\{\tau', \tau'', r_{k+1} - t\}$;
2.2.4: schedule $\tau\beta_j$ piece of task T_j in interval $[t, t+\tau]$ according to the algorithm for $P \mid spdp{-}lin{-}\delta_j, var \mid C_{max}$, for $T_j \in Q_k$;
2.2.5: $h(j) := h(j) - \frac{\tau\beta_j}{\delta_j}$ for $T_j \in Q_k$;
2.2.6: $t := t + \tau$;
end;
2.3: **if** $(\exists_{T_j \in Q_k} h(j) > 0)$ **then** $Q_{k+1} := Q_{k+1} \cup \{T_j : T_j \in Q_k, h(j) > 0\}$;
end; (* end of the algorithm *)

procedure CAPABILITIES(**in:**X;**out:**$\overline{\beta}$); (* X - a set of tasks *)
  begin
3.1: $\overline{\beta} := \overline{0}$; $avail := m$; (* $avail$ is the number of free processors *)
3.2: **while** $avail > 0$ **and** $\mid X \mid > 0$ **do**
    begin
3.2.1: construct set Y of the highest tasks in X with $h(j) > 0$;
3.2.2: **if** $\sum_{T_j \in Y} \delta_j > avail$ **then**
      begin
3.2.3: $\beta_j := \delta_j \frac{avail}{\sum_{T_j \in Y} \delta_j}$ for task $T_j \in Y$; $avail$:=0;
      end
    else (* tasks in Y can use at most $avail$ processors *)
      begin
3.2.4: $\beta_j := \delta_j$ for $T_j \in Y$; $avail := avail - \sum_{T_j \in Y} \delta_j$;
      end;
3.2.5: $X := X - Y$;
    end; (* of **while** loop *)
  end; (* of procedure CAPABILITIES *)

High level description. Intervals $[r_k, r_{k+1}]$ are considered consecutively in lines 2-2.3. In these intervals, subintervals are created in lines 2.2-2.2.6 where processing capabilities assignment remains constant. Tasks are assigned processors in line 2.2.1 analogously to the method proposed in [159]. High tasks are given preference (line 3.2.1). If there are more processors than can be simultaneously required by the ready tasks, a maximal possible number of processors is assigned in line 3.2.4. Otherwise, processors are shared (line 3.2.3) by equal tasks such that their heights decrease at the same pace (cf. line 2.2.5). The length of the current assignment is calculated in line 2.2.3. The assignment of processors to tasks changes in three cases: $h(j)$ for some initially higher task becomes equal to $h(j+1)$ of some initially lower task (calculated as τ' in line 2.2.2), or the lowest task in Q_k finishes (τ''), or else the end of the interval is encountered and tasks in Q_{k+1} must be considered. In line 2.3 tasks from Q_k not completed by the end of interval k are added to Q_{k+1} to be considered also in the next interval.

Lemma 5.1 *The algorithm for $P \mid spdp-lin-\delta_j, var, r_j \mid C_{max}$ is correct.*

Proof First, we prove that the algorithm halts. Procedure CAPABILITIES stops because in each execution of while loop in lines 3.2-3.2.5 at least one task is removed from X. Equal tasks reduce their heights with the same speed (cf. line 2.2.5). Hence, when two tasks become equal they remain equal until their completion. Height of an initially higher task cannot fall below a height of any initially lower task, which is guaranteed by calculation of τ' in line 2.2.2. Thus, two tasks can become equal at most $n-1$ times. We conclude that while loop of lines 2.2-2.2.6 can be executed only a limited number of times. Therefore, the algorithm stops.

Now, consider feasibility of the schedule. Tasks are not scheduled before their ready times because any task released at r_k can be considered in sets $Q_k,\dots,Q_l$, not $Q_1,\dots,Q_{k-1}$. No task T_j is ruled out from consideration as long as $h(j)\neq 0$. This means that each task receives required processing. Finally, in each subinterval built in lines 2.2-2.2.6 equation (5.1) is satisfied and a feasible schedule can be built by the algorithm introduced in Section 5.2.2. This is because:
(i) The sum of processing requirements of tasks is $\sum_{T_j\in Q_k}\tau\beta_j=\tau(\sum_{T_j\in Q'_k}\delta_j+\sum_{T_j\in Q_k-Q'_k}\frac{\delta_j(m-\sum_{T_i\in Q'_k}\delta_i)}{\sum_{T_i\in Q_k-Q'_k}\delta_i})\le\tau\, m$, where $Q'_k\subseteq Q_k$ is a set of tasks which received processing capabilities in line 3.2.4 of procedure CAPABILITIES. Thus, the sum of processing requirements does not exceed the subinterval capacity.
(ii) Task T_j $(j=1,\dots,n)$ is assigned at most δ_j processing capabilities which results from lines 3.2.3 and 3.2.4 in procedure CAPABILITIES. Hence, $\tau\ge\frac{\tau\beta_j}{\delta_j}$. □

Theorem 5.2 *The above algorithm builds an optimal schedule in $O(n^2)$ time.*

Proof In each subinterval created in lines 2.2-2.2.6 either all m processors are occupied, or as many processors are occupied as possible. Hence, capacity of interval $[r_k,r_{k+1}]$, $k=1,\dots,l-1$, is maximally exploited. Thus, the total processing requirement moved to Q_{k+1} is minimal possible. Since tasks with the longest expected execution time are preferred, also $\max_{j\in Q_k}h(j)$ is maximally decreased. The above arguments hold inductively for intervals $[r_k,r_{k+1}]$ $(k=1,\dots,l-1)$. Thus, also Q_l has tasks with the lowest possible $\max_{j\in Q_l}h(j)$ and their total processing requirement $\sum_{j\in Q_l}h(j)\delta_j$ is minimal. Using arguments of Theorem 5.1 proof (cf. equation (5.1)) we conclude that the schedule is optimal.

The complexity of the algorithm can be determined as follows. Grouping tasks according to their ready times in line 1 can be implemented in $O(n\log n)$ time. There are $O(n)$ values of index k considered in loop 2-2.3. Ordering tasks according to their heights is equivalent to sorting and requires $O(n \log n)$ time in line 1 and $O(n)$ time in line 2.1 (merging of Q_k and Q_{k-1}). Procedure CAPABILITIES can be executed in $O(n)$ time because it assigns processing capabilities to at most n tasks. Lines 2.2.2-2.2.6 require $O(n)$ time. Thus, the total complexity is $O(n^2)$. □

Observe that to schedule tasks with ready times at most equal r_k, the value of r_{k+1} is needed rather than information about tasks in $Q_{k+1}, \ldots, Q_l$. Hence, the above algorithm can be run on-line, i.e. it builds optimal schedules using only the information about tasks that have been already released and about the time when the new tasks will arrive. In the above algorithm it was assumed that the cost of preemption (or a context switch) is negligible. The cost of context switching is related to the number of preemptions. The number of preemptions can be determined by the number of subintervals where processing capabilities are constant and the preemptions within such subintervals. There can be at most $3n$ subintervals. Any task adds at most one preemption on one processor within the subinterval. The end of a subinterval can add one more preemption on each processor. Hence, there are at most $3n^2 + 3nm$ preemptions in the schedule.

Now, consider problem $P \mid spdp\text{–}lin\text{–}\delta_j, var \mid L_{max}$. There are l different due-dates: $d_1 < d_2 < \ldots < d_l$. This problem can be solved by a modification of the above algorithm. For problem $P \mid spdp\text{–}lin\text{–}\delta_j, var \mid L_{max}$ we have to guarantee that task T_j is feasibly executed in interval $[0, d_j + L_{max}]$ and L_{max} is minimal possible. For problem $P \mid spdp\text{–}lin\text{–}\delta_j, var, r_j \mid C_{max}$ we have to schedule task T_j in interval $[r_j, C_{max}]$ and minimize C_{max}. Thus, for instance I of $P \mid spdp\text{–}lin\text{–}\delta_j, var \mid L_{max}$ we can construct equivalent instance I' of problem $P \mid spdp\text{–}lin\text{–}\delta_j, var, r_j \mid C_{max}$ by assuming $r'_j = d_l - d_{l-j+1}$ for $j = 1, \ldots, l$. Schedule S' for I' should be read from the end at C'_{max} to the beginning to be schedule S for $P \mid spdp\text{–}lin\text{–}\delta_j, var \mid L_{max}$, i.e. each time instant t' in S' has equivalent $t = C'_{max} - t'$ in S. We conclude:

Corollary 5.1 *$P \mid spdp\text{–}lin\text{–}\delta_j, var \mid L_{max}$ is solvable in $O(n^2)$ time.*

5.2.4 $P2 \mid size_j, pmtn, r_j \mid C_{max}$

We use here methods designed in the previous section to solve the problem of preemptive scheduling on two processors of multiprocessor tasks ready at different moments with a fixed number of used processors, for the schedule length criterion. The notation is analogous to the one in the previous section. *Height* $h(j)$ of task T_j is the time required to complete it. There are $l \leq n$ different values of ready times $r_1 = 0 < r_2 < \ldots < r_l < r_{l+1} = \infty$. Procedure

CAPABILITIES was presented in Section 5.2.3 and we do not repeat it here. Beyond the notation of Section 5.2.3 we use:

Q_k - the set of uniprocessor tasks ready in interval k,

S_k - the set of duoprocessor tasks ready in interval k.

An Algorithm for $P2 \mid size_j, pmtn, r_j \mid C_{max}$

1: $t := 0$; group duoprocessor tasks with ready time r_k in S_k,
  and uniprocessor tasks with ready time r_k in set Q_k,
  order tasks in Q_k according to nonincreasing heights, $k = 1, \dots, l$;
2: **for** $k := 1$ **to** l **do**
  begin
2.1: order tasks in Q_k according to nonincreasing values of $h(j)$ for $T_j \in Q_k$;
2.2: $\tau := \min\{\sum_{T_j \in S_k} t_j^2, r_{k+1} - r_k\}$;
2.3: schedule τ units of duoprocessor tasks from S_k in interval $[r_k, r_k + \tau]$;
  reduce $h(j)$ of scheduled duoprocessor task T_j by the received amount
  of processing for $T_j \in S_k$;
2.4: $t := t + \tau$;
2.5: **while** $(r_{k+1} > t)$ **and** $(\exists_{T_j \in Q_k} h(j) > 0)$ **do**
  begin
2.5.1: CAPABILITIES$(Q_k, \overline{\beta})$;
2.5.2: calculate times:
  if $\exists_{T_j, T_{j+1} \in Q_k} h(j) > h(j+1)$ **then**
  $\tau' := \min_{T_j, T_{j+1} \in Q_k}\{\frac{h(j)-h(j+1)}{\beta_j - \beta_{j+1}} : \beta_j \neq \beta_{j+1}, h(j) > h(j+1)\}$
  else $\tau' := \infty$
  - the shortest time required for two tasks T_j, T_{j+1} with different heights
  to become equal;
  $\tau'' := h(\mid Q_k \mid)$ - the time to the earliest completion of any task;
2.5.3: $\tau := \min\{\tau', \tau'', r_{k+1} - t\}$;
2.5.4: schedule $\tau\beta_j$ piece of task T_j in interval $[t, t + \tau]$ according to
  McNaughton rule, for $T_j \in Q_k$;
2.5.5: $h(j) := h(j) - \tau\beta_j$ for $T_j \in Q_k$;
2.5.6: $t := t + \tau$;
  end;
2.6: **if** $(\exists_{T_j \in Q_k} h(j) > 0)$ **then** $Q_{k+1} := Q_{k+1} \cup \{T_j : T_j \in Q_k, h(j) > 0\}$;
2.7: **if** $(\exists_{T_j \in S_k} h(j) > 0)$ **then** $S_{k+1} := S_{k+1} \cup \{T_j : T_j \in S_k, h(j) > 0\}$;
  end; (* end of the algorithm *)

High level description. As in the previous algorithm intervals $[r_k, r_{k+1}]$ are considered in lines 2-2.7. First, duoprocessor tasks are scheduled in these intervals such that either all duoprocessor tasks are scheduled in the interval or the interval is completely filled by these tasks (lines 2.2-2.4). Then, within intervals $[r_k, r_{k+1}]$ subintervals are created in lines 2.5-2.5.6 where assignment of processing capabilities to tasks in Q_k remains constant. Tasks are assigned processors in line 2.5.1 as in [159] and Section 5.2.3. Note that $\delta_j = size_j = 1$ for all tasks passed to procedure CAPABILITIES. The current processor assignment of uniprocessor tasks changes in three cases (calculated in line 2.5.3): $h(j)$ for some initially higher task becomes equal to $h(j+1)$ of some initially lower task (calculated as τ' in line 2.5.2), the lowest task in Q_k finishes (τ''), the end of the interval is encountered. In line 2.6 uniprocessor tasks from Q_k not completed by the end of interval k are added to Q_{k+1} to be considered in the next interval. The same applies to duoprocessor tasks in line 2.7.

Lemma 5.2 *The algorithm for* $P2 \mid size_j, pmtn, r_j \mid C_{max}$ *is correct.*

Proof First, we prove that the algorithm stops. Procedure CAPABILITIES stops as explained in Lemma 5.1. Equal tasks reduce their heights with the same speed (cf. line 2.5.5). Hence, when two uniprocessor tasks become equal they remain equal until their completion. Height of initially higher task cannot fall below the height of an initially lower task, which is guaranteed by calculation of τ' in line 2.5.2. Thus, two tasks can become equal at most $n-1$ times. We conclude that while loop of lines 2.5-2.5.6 can be executed only a limited number of times and the algorithm stops.

Now, consider feasibility of the schedule. Tasks are not scheduled before their ready times because any uniprocessor (duoprocessor) task released at r_k can be considered in sets $Q_k, \ldots, Q_l$ $(S_k, \ldots, S_l)$. No task is ruled out from consideration as long as $h(j) \neq 0$. Hence, each task is completed. In each subinterval built in lines 2.5-2.5.6:
(i) no task is assigned more than processing capability 1, which results from lines 3.2.3 and 3.2.4 in procedure CAPABILITIES. Hence, each uniprocessor task fits in the subinterval.
(ii) the sum of processing requirements of tasks is $\sum_{T_j \in Q_k} \tau\beta_j = \tau(\mid Q'_k \mid + \mid Q_k - Q'_k \mid \frac{m-|Q'_k|}{|Q_k - Q'_k|}) \leq \tau\, m$, where $Q'_k \subseteq Q_k$ is a set of tasks which received processing capability 1 in line 3.2.4 of procedure CAPABILITIES. Thus, the sum of processing requirements for the subinterval does not exceed its capacity.

(i) and (ii) ensure that in the subinterval created in lines 2.5-2.5.6 a feasible schedule can be obtained by McNaughton's wrap-around rule. □

Theorem 5.3 *The above algorithm builds an optimal schedule in $O(n^2)$ time.*

Proof Observe that by swapping pieces of tasks on both processors simultaneously any feasible schedule can be converted to a schedule with the same length in which duoprocessor tasks are executed consecutively from their ready time. Thus, among optimal schedules there is one where duoprocessor tasks are executed first in intervals $[r_k, r_{k+1}]$ $(k = 1, \dots, l)$.

The completion time for tasks in interval $[r_l, r_{l+1}]$, and hence of the whole schedule, is $C_{max} = r_l + \sum_{j \in S_l} h(j) + \max\{\max_{j \in Q_l} h(j), \frac{1}{2}\sum_{j \in Q_l} h(j)\}$ which is the amount of processing required by duoprocessor tasks plus either the longest uniprocessor task or the mean loading of the processors by uniprocessor tasks. Thus, the length of the schedule, depends on the amount of duoprocessor tasks shifted from $[r_{l-1}, r_l]$, amount of shifted uniprocessor tasks, and the longest shifted piece of a uniprocessor task. In each subinterval created in lines 2.5-2.5.6 either all m processors are occupied, or as many processors are occupied as possible. Hence, capacity of interval $[r_k, r_{k+1}]$, $k = 1, \dots, l-1$ is maximally exploited, and the total processing requirement moved to Q_{k+1} is minimal possible. Tasks with the longest expected execution time are preferred in procedure CAPABILITIES, thus $\max_{j \in Q_k} h(j)$ is maximally decreased. These arguments hold inductively for intervals $[r_k, r_{k+1}]$ $(k = 1, \dots, l-1)$. Thus, Q_l has tasks with the lowest possible $\max_{j \in Q_l} h(j)$, $\sum_{j \in Q_l} h(j)$, and $\sum_{j \in S_l} h(j)$ is minimal. Hence, the schedule is optimal.

Grouping tasks according to their ready times in line 1 requires $O(n \log n)$ time. There are $O(n)$ values of index k considered in loop 2-2.7. Ordering tasks according to their heights is equivalent to sorting and requires $O(n \log n)$ time in line 1 and $O(n)$ time in line 2.1 (merging of Q_k and Q_{k-1}). Procedure CAPABILITIES can be executed in $O(n)$. Lines 2.5.2-2.5.6 require $O(n)$ time. Thus, the total complexity is $O(n^2)$. □

Note that the above algorithm can be extended to any number of processors provided that $\forall_{j \in \mathcal{T}} size_j \in \{1, m\}$. Furthermore, it can be used to solve $P2 \mid size_j, pmtn \mid L_{max}$ as it was possible to use an algorithm for $P \mid spdp-lin-\delta_j, var, r_j \mid C_{max}$ to solve $P \mid spdp-lin-\delta_j, var \mid L_{max}$.

Corollary 5.2 *$P2 \mid size_j, pmtn \mid L_{max}$ is solvable in $O(n^2)$ time.*

In the following Table 5.1 we summarize results, from the literature as well as presented in this work, for scheduling multiprocessor tasks on parallel processors.

Table 5.1: Scheduling multiprocessor tasks on parallel processors

Problem	Result	Reference
Nonpreemptive scheduling		
$P \mid size_j, p_j = 1 \mid C_{max}$	s**NP**h	[149]
$P3 \mid size_j, p_j = 1, chain \mid C_{max}$ and $size_j \in \{1, 2\}$	s**NP**h	[45]
$P \mid size_j, p_j = 1, prec \mid C_{max}$	$S_{LS} = \frac{2m-\Delta}{m-\Delta+1}$	[149]
$P2 \mid size_j, p_j = 1, prec \mid C_{max}$	$O(n^{\log_2 7})$	[149]
$P \vert size_j, p_j = 1 \vert C_{max}$ and $size_j \in \{1, \Delta\}$	$O(n)$	[32]
$P \mid size_j, p_j = 1 \mid C_{max}$ and $size_j \in \{1, \ldots, \Delta\}$	$O(n)$ ILP	[32]
$P \mid size_j, p_j = 1 \mid C_{max}$	s**NP**h	[32]
$P2 \mid size_j \mid C_{max}$, $P3 \mid size_j \mid C_{max}$	**NP**h,pseudopoly.	[90]
$P4 \mid size_j \mid C_{max}$	?	[90]
$P5 \mid size_j \mid C_{max}$	s**NP**h	[90]
$P2 \mid size_j, chain \mid C_{max}$	s**NP**h	[90]
$P \mid size_j \mid C_{max}$	$S_{LPT} \leq \frac{4\Delta}{3} - \frac{\Delta(\Delta+1)}{6m}$	[148]
$P4 \mid size_j \mid C_{max}$ and $size_j \neq 1$	pseudopoly.	[85]
$P2 \mid size_j \mid \sum w_j c_j$	s**NP**h,$S_{H1} \leq 2$	[144]
$P2 \mid size_j \mid \sum w_j c_j$	dyn.prog. for special case	[144]
$P2 \mid size_j \mid \sum c_j$	**NP**h, $S_{H1} \leq \frac{3}{2}$	[144]
$P2 \mid size_j \mid L_{max}$	s**NP**h	[144]
$P2 \mid size_j \mid L_{max}$	$L^{H2}_{max} \leq L^*_{max} + \frac{1}{2}\sum_{j\in\mathcal{T}^2} t_j$	[144]
$P2 \mid size_j, p_j = 1 \mid L_{max}$	$O(n \log n)$ EDD	[144]
$P \mid size_j, p_j = 1, m.d.{-}chains \mid C_{max}$ and $size_j \in \{1, \Delta\}$,$2\Delta > m$	$O(n \log n)$	[45]
$P \mid size_j, p_j = 1, u.{-}chains \mid C_{max}$ and $size_j \in \{1, \Delta\}$	$O(n \log n)$	[45]
$P \mid cube_j \mid C_{max}$	$S_{LDLPT} = 2 - \frac{2}{m}$	[59]
$P \mid cube_j \mid C_{max}$	$S_{LDF} = 2 - \frac{1}{m}$	[214]
$P \mid cube_j \mid \sum c_j$	experimental study	[133]
$P \mid cube_j \mid C_{max}$	$S_{LPT} \leq 2 - \frac{1}{m}$	[148]
$P \mid spdp{-}lin{-}\delta_j, prec \mid C_{max}$	$S_{LS} = \Delta + \frac{m-\Delta}{m}$	[205]
$P \mid spdp{-}lin{-}\delta_j, prec \mid C_{max}$	$S_{ECT} < \ln \Delta + 2$	[205]
$P \mid spdp{-}lin{-}\delta_j, prec \mid C_{max}$	$S_{ECT} < 3 - \frac{2}{m}$	[206]
$P \mid spdp{-}any \mid C_{max}$ and $n \leq m$	$S_{VS} = \min\{n, \frac{R}{1-\frac{m}{n}}\}$	[131]
$Q, win^{1)} \mid spdp{-}any, n=1 \mid C_{max}$	$O(m^2)$	[7]
$P \mid spdp{-}lin \mid \sum c_j$	SPT is optimal	[181]
$P \mid spdp{-}any \mid \sum c_j$	special cases analyzed	[181]
$P \mid spdp{-}any, prec \mid C_{max}$	heuristic	[63]
$P \mid spdp{-}any, prec \mid C_{max}$	$O(e^2 + en + I(e + n))$	[170]
$Q, win^{1)} \mid spdp{-}any, n=1 \mid C_{max}$	$O(m \log m)$	[91]

1) Processors become continuously available after different moments of time.

Problem	Result	Reference
Preemptive scheduling		
$P \mid size_j, pmtn \mid C_{max}$ and $size_j \in \{1, \Delta\}$	$O(n)$	[32]
$Pm \mid size_j, pmtn \mid C_{max}$	LP	[32]
$P \mid size_j, pmtn, res1 \cdot 1 \mid C_{max}$ and $size_j \in \{1,2\}$	$O(n \log n)$	[41]
$Q \mid size_j, pmtn \mid C_{max}$ and $size_j \in \{1,2\}$	$O(n \log n + nm)$	[37]
$Q \mid size_j, pmtn \mid C_{max}$ and $size_j \in \{1, \Delta\}$	$O(n \log n + nm)$	[39]
$Qm \mid size_j, pmtn \mid C_{max}$	LP	[38]
$Pm \mid size_j, pmtn \mid L_{max}$ and $size_j \in \{1, \Delta\}$	LP or tabu search+LP	[40]
$P \mid size_j, pmtn, res1 \cdot 1 \mid C_{max}$	$O(nm)$	[43]
$Pm \mid size_j, pmtn, res \cdot \cdot \mid C_{max}$	LP	[43]
$P \mid size_j, pmtn \mid C_{max}$	**NP**h, ?	[85]
$P2 \mid size_j, pmtn, r_j \mid C_{max}$	$O(n^2)$	Th.5.3
$P2 \mid size_j, pmtn \mid L_{max}$	$O(n^2)$	Coro.5.2
$P \mid cube_j, pmtn \mid C_{max}$	$O(n^2(\log n + \log \max_j\{t_j^{size_j}\}))$	[58]
$P \mid cube_j, pmtn \mid C_{max}$	$O(n \log n(\log n + \max_j\{t_j^{size_j}\}))$	[119, 2]
$P \mid cube_j, pmtn, r_j, d_j \mid -$	LP	[169]
$Q \mid cube_j, pmtn \mid C_{max}$	$O(n \log n + nm)$	[38, 84]
$P \mid cube_j, pmtn \mid C_{max}$	$O(n^2 m^2)$	[183]
$P \mid cube_j, pmtn \mid C_{max}$	$O(n^2 \log^2 n)$	[214]
$R \mid spdp\text{–}lin, var, r_j, d_j \mid -$	LP	[200]
$P \mid spdp\text{–}lin, var, r_j, d_j \mid X$	$O(n^2)$	[203]
$P \mid spdp\text{–}lin, var, r_j, d_j \mid X$	$O(n^2)$	[202]
$P \mid spdp\text{–}lin\text{–}\delta_j, var, r_j \mid L_{max}$	$O(n^4 m)$	[201]
$P \mid spdp-any, var, r_j \mid L_{max}$	continuous processors	[208]
$Pm \mid spdp\text{–}any, pmtn \mid C_{max}$ and $m \geq 2$	**NP**h,pseudopoly.	[90]
$P \mid spdp\text{–}any, pmtn \mid C_{max}$	s**NP**h	[90]
$P \mid spdp\text{–}lin\text{–}\delta_j, var, chain \mid C_{max}$ and $\mid chain \mid = 3$	s**NP**h, $S_{H1} = 3$ $S_{H2} = S_{H3} = 2$	[89]
$Pm \mid spdp\text{–}lin\text{–}\delta_j, var, chain \mid C_{max}$ and $\mid chain \mid = 3$	polynomially solvable	[89]
$P \mid spdp\text{–}lin\text{–}\delta_j, var, chain \mid C_{max}$ and $\mid chain \mid = 2$	$O(n \log n)$	[89]
$P \mid spdp\text{–}any, pmtn \mid C_{max}$	$S_{VS} \leq R$	[132]
$P \mid spdp-lin-\delta_j, var \mid C_{max}$	$O(n)$	Th.5.1
$P \mid spdp\text{–}lin\text{–}\delta_j, var, r_j \mid C_{max}$	$O(n^2)$	Th.5.2
$P \mid spdp\text{–}lin\text{–}\delta_j, var \mid L_{max}$	$O(n^2)$	Coro.5.1

5.3 Dedicated Processors

This section considers scheduling multiprocessor tasks on dedicated processors. First, we review the existing literature of this field (including earlier author's works). Then, from Section 5.3.2 on we present new results. Part of them was prepared in cooperation with other researchers [24, 20]. In Section 5.3.2 we present analysis of low order complexity algorithms based on Earliest Due-Date rule. In Section 5.3.3 we consider scheduling in time windows.

5.3.1 Overview of Earlier Results

The first paper considering multiprocessor scheduling seems to be [48] in which branch and bound (B&B) algorithm is proposed for scheduling in chemical plants. A concept of *compatibility* and *incompatibility* of tasks has been introduced. Two tasks T_i and T_j are compatible if $fix_i \cap fix_j = \emptyset$. The two tasks are incompatible when $fix_i \cap fix_j \neq \emptyset$. This gives way to definition of incompatibility graph in which nodes represent tasks and edges link pairs of tasks which cannot be processed together. To bound the search tree a Maximum Degree of Incompatibility (MDI) was used to prefer executing some tasks over the others.

In [130] scheduling of diagnostic tests is analyzed. The tests to be performed are represented by a *diagnostic graph* in which nodes represent processors and edges - tasks. An edge has weight - processing time of a task. Two processors connected by an edge are simultaneously required to test each other. We will call such kind of representation *scheduling graph* (following [134]). The considered problem $P \mid fix_j \mid C_{max}$ with $\forall_j \mid fix_j \mid= 2$ is proved in [130] to be **NP**h. An LPT heuristic is analyzed, and worst case performance bound $4(d-1)/d$ is demonstrated, where d is the maximum degree of any vertex. For graphs with $d \leq 5$ this bound is tightened to 3, and for binomial graphs with integral ratio of the weights to 2.

In [70] the problem of scheduling file transfers is considered. A file transfer involves two computers/communication centers. Each computer may be able to use multiple ports to execute simultaneous file transfers. Let p denote maximum number of ports. The transfers to be performed are described by a scheduling graph in which vertices are communicating nodes and edges are files to transfer. The problem is analyzed for the case with central controller as well as for the distributed case. The complexity of the problem is analyzed in 10 theorems using (mainly) edge coloring model. In this way complexity of 43 special cases is established (including general graphs, bipartite graphs, trees, paths, even/odd cycles, one-port, arbitrary number of ports, single edges, multiple edges). The performance ratio of LS heuristics is analyzed. It is proved that in the worst case $4/3 < S_{LS} \leq 3$. This bound can be tightened for special forms of the scheduling graph, e.g. $S_{LS} \leq 2$ for $p \leq 2$. LPT heuristic has performance ratio $5/2 - 1/p$ when $p \geq 2$. Finally, two distributed protocols are proposed to schedule file transfers. For the first (called Demand Protocol 1) it is proved that $C_{max}^{DP1} \leq 3C_{max}^{*} + e\varepsilon$, where C_{max}^{DP1} is the length of the schedule, C_{max}^{*} the length of the optimal schedule, e is the number of edges in the scheduling graph, ε is the maximum time to initiate some file transfer. For the second protocol similar bounds have been obtained. These bounds were tightened in special cases.

In [134] problem $P \mid fix_j \mid C_{max}$ where $\mid fix_j \mid \in \{1,2\}$ is analyzed. Uniprocessor tasks are represented in the scheduling graph as loops. The above problem is **NP**h even if the scheduling graph is caterpillar with one loop or a star with a loop at each noncentral vertex. References are given to other works establishing the complexity of 16 subcases.

In [74] the analysis of problems $P \mid fix_j \mid C_{max}$ and $P \mid fix_j, p_j = 1 \mid C_{max}$ is motivated by scheduling of built-in tests for VLSI circuits. An *incompatibility graph* is a model of dependencies among the tasks. Three algorithms based on Maximum Degree of Incompatibility are proposed.

In [79] problem $P \mid fix_j \mid \sum w_j c_j$ is considered. For $P2 \mid fix_j, p_j = 1 \mid \sum w_j c_j$ optimization algorithm with complexity $O(n \log n)$ is given. For the general version of the problem integer linear programming formulation was given. Two relaxation methods and two heuristics were presented. Computational results are reported.

In [135] preemptive scheduling is considered. By reduction of edge multicoloring problem $P \mid fix_j, pmtn \mid C_{max}$ with $\mid fix_j \mid = 2$ is proved to be s**NP**h (via complexity equivalence with $P \mid fix_j, p_j = 1 \mid C_{max}$). For problem $Pm \mid fix_j, pmtn \mid C_{max}$, i.e. when the number of processors is fixed an algorithm based on linear programming and processor feasible sets is given.

In [31] the case of nonpreemptive scheduling on three processors is analyzed. The complexity of this problem is established by the following theorem.

Theorem 5.4 *Problem $P3 \mid fix_j \mid C_{max}$ is s**NP**h in general* [31].

Proof We prove strong **NP**-hardness by reduction of 3-PARTITION to a decision version of our problem. 3-PARTITION is defined as follows.

3-PARTITION
INSTANCE: Set A of $3q$ numbers a_j $(j = 1, \ldots, 3q)$, such that $\sum_{j=1}^{3q} a_j = Bq$

and $B/4 < a_j < B/2$ for $j = 1, \ldots, 3q$. Without loss of generality we assume that $B > q$.
QUESTION: Can A be partitioned into q disjoint subsets $A_1, \ldots, A_q$ such that $\sum_{a_j \in A_i} a_j = B$ for $i = 1, \ldots, q$?

The above problem can be transformed into problem $P3 \mid fix_j \mid C_{max}$ as follows: $n = 12q - 1$, $\mathcal{T} = U \cup V \cup W \cup X \cup Y_{12} \cup Y_{23} \cup Y_{13}$. The above sets of tasks are defined in the following table:

Task set	fix_j	j - task indices	processing time	
U	$\{P_1\}$	1	$B^5 + B^4$	k_2
		$2, \ldots, 2q$	$B^7 + B^5 + B^4$ for $(j \bmod 2) = 1$	k_9
			$B^6 + B$ for $(j \bmod 2) = 0$	k_6
V	$\{P_2\}$	$2q+1, \ldots, 4q-1$	B^5 for $(j \bmod 2) = 1$	k_3
			$B^7+B^6+B^2+B$ for $(j \bmod 2)=0$	k_7
		$4q$	$B^6 + B^2 + B$	k_{11}
W	$\{P_3\}$	$4q+1, \ldots, 6q-1$	$B^6 + B^5 + B^3$ for $(j \bmod 2) = 1$	k_4
			B^7 for $(j \bmod 2) = 0$	k_{10}
X	$\{P_3\}$	$6q, \ldots, 9q-1$	a_{j-6q+1}	-
Y_{12}	$\{P_1, P_2\}$	$9q, \ldots, 10q-1$	B^3	k_5
Y_{23}	$\{P_2, P_3\}$	$10q, \ldots, 11q-1$	B^4	k_1
Y_{13}	$\{P_1, P_3\}$	$11q, \ldots, 12q-1$	B^2	k_8

$y = q(B^7 + B^6 + B^5 + B^4 + B^3 + B^2 + B) - B^7$.
We ask whether for the above task set a schedule of length at most y exists. Suppose the answer to 3-PARTITION is positive, then a feasible schedule of length y looks like the one in Fig. 5.1.

Assume now that a feasible schedule not longer than y exists for problem $P3 \mid fix_j \mid C_{max}$. Note that processing requirements for all processors are equal to the schedule length. Hence, no idle time is allowed in a feasible schedule. To prove a positive answer for 3-PARTITION we will examine the numbers of various type tasks scheduled between tasks from sets Y_{12}, Y_{23}, Y_{13}. The notation of the task numbers for each defined task type is presented in the last column of the above table (cf. also Fig. 5.1).

1. Tasks preceding some duoprocessor task $T_j \in Y_{13}$ must finish simultaneously on P_1 and P_3. Thus, $k_2(B^5+B^4)+k_5B^3+k_6(B^6+B)+k_8B^2+k_9(B^7+B^5+B^4) = k_1B^4+k_4(B^6+B^5+B^3)+k_8B^2+k_{10}B^7+\sum_{T_i \in L} a_{i-6q+1}$, where L is the set of tasks from X executed before the considered task $T_j \in Y_{13}$. Coefficients at the same power of B must be equal on both sides of the above equation, we have (equations bounding the coefficients are presented along with B at the appropriate power): $(B :)\, k_6B = \sum_{T_i \in L} a_{i-6q+1}$,

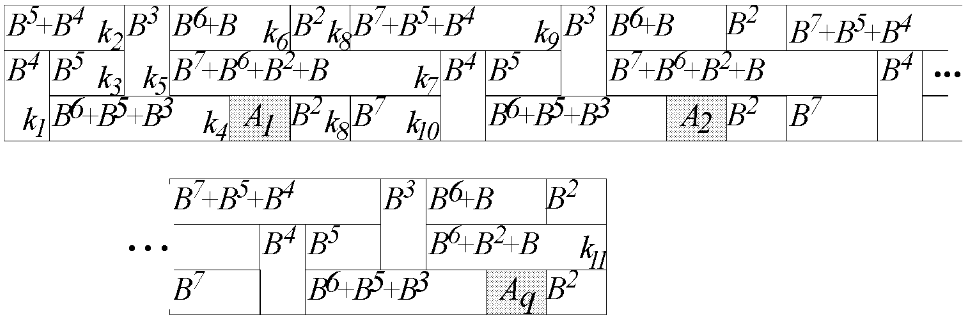

Figure 5.1: A schedule for the proof of Theorem 5.4. Symbols used to denote the numbers of the given type tasks are placed in the lower-right corners.

$(B^3 :)\, k_4 = k_5$, $(B^4 :)\, k_2 + k_9 = k_1$, $(B^5 :)\, k_2 + k_9 = k_4$, $(B^6 :)\, k_6 = k_4$, $(B^7 :)\, k_9 = k_{10}$, and from this

$$k_1 = k_4 = k_5 = k_6 = k_2 + k_9, \;\; k_{10} = k_9 \tag{5.2}$$

We conclude that the same number of tasks from sets Y_{12}, Y_{23} must precede (and follow) in the schedule any task from Y_{13} because $k_1 = k_5$. $\sum_{T_i \in L} a_{i-6q+1}$ must be a multiple of B.

2. Tasks preceding some duoprocessor task $T_j \in Y_{23}$ must finish simultaneously on P_2 and P_3. Therefore, $k_3 B^5 + k_5 B^3 + k_7(B^7 + B^6 + B^2 + B) + k_{11}(B^6 + B^2 + B) = k_4(B^6 + B^5 + B^3) + k_8 B^2 + k_{10} B^7 + \sum_{T_i \in L} a_{i-6q+1}$, from which we have $(B :)\, (k_7 + k_{11})B = \sum_{T_i \in L} a_{i-6q+1}$, $(B^2 :)\, k_7 + k_{11} = k_8$, $(B^3 :)\, k_5 = k_4$, $(B^5 :)\, k_3 = k_4$, $(B^6 :)\, k_7 + k_{11} = k_4$, $(B^7 :)\, k_{10} = k_7$. From the above

$$k_3 = k_4 = k_5 = k_8 = k_7 + k_{11} = \sum_{T_i \in L} a_{i-6q+1}/B, \;\; k_7 = k_{10} \tag{5.3}$$

Since $k_5 = k_8$ the number of tasks from Y_{12} which precede (and follow) some task from Y_{23} must be equal to the number of tasks from Y_{13}.

3. Tasks preceding some duoprocessor task from $T_j \in Y_{12}$ must finish simultaneously on P_1 and P_2. Hence, $k_2(B^5+B^4)+k_5 B^3+k_6(B^6+B)+k_8 B^2 + k_9(B^7+B^5+B^4) = k_1 B^4 + k_3 B^5 + k_7(B^7+B^6+B^2+B) + k_5 B^3 + k_{11}(B^6 + B^2 + B)$. From this we obtain: $(B :)\, k_6 = k_{11} + k_7$, $(B^2 :)\, k_8 = k_7 + k_{11}$, $(B^4 :)\, k_2 + k_9 = k_1$, $(B^5 :)\, k_2 + k_9 = k_3$, $(B^6 :)\, k_6 = k_7 + k_{11}$, $(B^7 :)\, k_9 = k_7$ and from this

$$k_6 = k_8 = k_7 + k_{11}, \;\; k_1 = k_3 = k_2 + k_9, \;\; k_7 = k_9 \tag{5.4}$$

Suppose some $T_j \in Y_{12}$ (i.e. with $fix_j = \{P_1, P_2\}$) is the first (or the last) scheduled task from Y_{12} an it is executed after (before) both T_1 and T_{4q}. From this assumption we get $k_2 = k_{11} = 1$ and from (5.4) $k_1 = k_3 = 1 + k_9 = 1 + k_7 = k_6 = k_8$. Thus, $k_1 = k_8 \geq 1$. From (5.3) it is known that the same number of tasks from Y_{12} and from Y_{13} must precede any task from Y_{23}. But here we would have $k_8 \geq 1$ tasks from Y_{13} and no task from Y_{12}, which is a contradiction.

Suppose some $T_j \in Y_{12}$ is the first (the last) scheduled task from Y_{12} and there is neither T_1 nor T_{4q} before (after) it. From (5.4) we obtain $k_1 = k_8$. When $k_1 = k_8 \geq 1$ the same arguments as in the previous paragraph can be applied. On the other hand, $k_1 = k_8 = 0$ implies that it is impossible to be in agreement with (5.2) and (5.3) and schedule any task from Y_{23} or Y_{13} after the considered T_j. This means that such a schedule cannot exist. Conclusion: the first executed task from Y_{12} must be preceded either by T_1 or T_{4q}.

Assume T_1 precedes the first task from Y_{12}, then $k_2 = 1, k_{11} = 0, k_1 = k_3 = 1 + k_9 = 1 + k_7 = 1 + k_8 = 1 + k_6$, and there is one more task from Y_{23} before any task from Y_{12} than the number of the tasks from Y_{13}.

When T_{4q} precedes the first task from Y_{12} then $k_2 = 0, k_{11} = 1, k_1 = k_3 = k_7 = k_9 = k_8 - 1 = k_6 - 1$, and there is one less task from Y_{23} before any task from Y_{12} than the number of tasks from Y_{13}. Hence, the schedule cannot be started with a task from Y_{12}.

4. Now, we will examine whether the schedule can be started by three uniprocessor tasks. Suppose it is possible.

4a. $T_j \in Y_{12}$ is the first duoprocessor task in the schedule, then $k_2 + k_9 + k_6 > 0$, $k_{11} + k_3 + k_7 > 0$, $k_1 = k_8 = 0$ on the other hand, from (5.4) $k_2 + k_9 = k_3$, $k_{11} + k_7 = k_6$, $k_1 = k_3$, $k_6 = k_8$ from which we get a contradiction: $0 = k_1 + k_8 = k_3 + k_6 > 0$.

4b. $T_j \in Y_{13}$ is the first duoprocessor task in the schedule, then $k_2 + k_9 + k_6 > 0$, $k_1 = k_5 = 0$, and from (5.2) $k_2 + k_9 = k_6$, $k_1 = k_6$, from which we obtain a contradiction: $0 < 2k_6 = 2k_1 = 0$.

4c. $T_j \in Y_{23}$ is the first duoprocessor task in the schedule, but there is no combination of tasks from sets V (requiring P_2) and W (requiring P_3) which would compensate B^2 and B^3 without duoprocessor tasks. Hence, such a schedule is infeasible.

Conclusion: schedule may not start with uniprocessor tasks on all three processors. Following conclusion of Point **3**, schedule must start with a task either from Y_{23} or from Y_{13}.

5. Let us analyze how many tasks of various types precede the first task

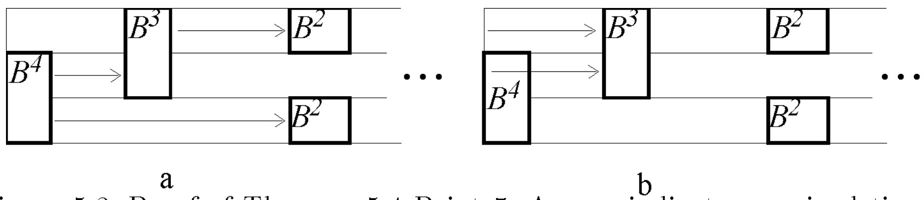

Figure 5.2: Proof of Theorem 5.4 Point **5**. Arrows indicate examined times intervals.

$T_j \in Y_{13}$. Without loss of generality we assume that the first duoprocessor task in the schedule belongs to Y_{23} (i.e. $k_1 \geq 1$). Consider the time interval between the task from Y_{23} preceding the first task from Y_{13} (cf. Fig. 5.2a). We have: $k_2(B^5+B^4)+k_6(B^6+B)+k_9(B^7+B^5+B^4)+k_5B^3+k_3B^5+k_7(B^7+B^6+B^2+B)+k_{11}(B^6+B^2+B)+k_1B^4 = \sum_{T_i \in L} a_{i-6q+1} + k_4(B^6+B^5+B^3)+k_{10}B^7+k_1B^4$. From the above $(B :)\ \sum_{T_i \in L} a_{i-6q+1}/B = k_{11}+k_6$, $(B^2 :)\ k_{11}+k_7=0$, $(B^3 :)\ k_5=k_4$, $(B^4 :)\ k_2+k_9=0$, $(B^5 :)\ k_2+k_3+k_9=k_4$, $(B^6 :)\ k_6+k_7+k_{11}=k_4$, $(B^7 :)\ k_9+k_7=k_{10}$, and from this

$$k_2=k_7=k_9=k_{10}=k_{11}=0,\ k_3=k_4=k_5=k_6=\sum_{T_i \in L} a_{i-6q+1}/B \quad (5.5)$$

Now, analyze the time from the start of the schedule to the first task $T_i \in Y_{12}$ (cf. Fig. 5.2b) which precedes $T_j \in Y_{13}$. We have $k'_2(B^5+B^4)+k'_6(B^6+B)+k'_9(B^7+B^5+B^4) = k_3B^5+k_1B^4$ (note that k_1, k_3 are the same numbers as in equation (5.5)). From this we obtain: $(B^7 :)\ k'_9=0, (B^5 :)\ k'_9+k'_2=k_3, (B^4 :)\ k'_9+k'_2=k_1$ and thus $k'_2=k_1=k_3$. Since $k_1 \geq 1$ and $k'_2 \in \{0,1\}$ then $k'_2=k_1=k_3=k_4=k_5=k_6=1$. Hence, the schedule before the first task from Y_{13} must look like in Fig. 5.1.

6. One can examine now what tasks are present between the first (second, third, etc.) task from Y_{12}, and the first task from Y_{23} which is following it. Analogously, the time between the first (second, third, etc.) task from Y_{13} and the task from Y_{12} following it. From such an analysis it can be inferred that the schedule must have a form like the one in Fig. 5.1. There are boxes B time long on P_3 between consecutive tasks from Y_{23} where tasks from set X must be executed. Hence, the answer to the 3-PARTITION must be positive. Observe that the schedule ends with a task from set Y_{13} and task T_{4q}. If we assumed in Point **5** that the schedule starts with a task from set Y_{23} we would obtain the same schedule read from the end. □

In [31] normal schedules (NS) for problem $P3 \mid fix_j \mid C_{max}$ are analyzed. A normal schedule is the one in which task requiring two processors simultaneously are executed in parallel with tasks requiring the third processor. Three special cases are identified when normal schedules are optimal. In general case performance ratio of normal schedules is shown to be less than 4/3. The same problem is further analyzed in [78]. It is shown that normal schedules guarantee performance 5/4 and this bound is tight. On the contrary LPT and SPT rules have tight worst-case performance ratio 3. For a certain distribution of instances it is shown that over 95% of them are recognized as solvable in polynomial time. A better approximation algorithm with tight performance ratio $S_{18} = 7/6$ has been proposed in [102] (we call it *18* for it chooses the best out of 18 schedules).

In [19] preemptive scheduling is considered. For problems $Pm|fix_j, pmtn|$ L_{max}, $Pm \mid set_j, pmtn \mid L_{max}$, $Pm \mid fix_j, pmtn, r_j|$ L_{max}, $Pm \mid set_j, pmtn, r_j|$ L_{max}, are solved in polynomial time by the use of processor feasible sets and linear programming.

In [50] many open-, flow- and job-shop scheduling problems with multiprocessor tasks are considered. For the open-shop it is assumed that the same number operations of different tasks require the same set of processors. In some of the considered problems the number of stages is fixed, i.e. for each task the number of operations can be fixed. These problems are further pursued in [49]. In some cases the number of task types was fixed to R.

In [21] $O(n)$ complexity algorithms are given for problems $P2|fix_j, pmtn|$ C_{max}, $P3 \mid fix_j, pmtn \mid C_{max}$, $P4 \mid fix_j, pmtn \mid C_{max}$, $P4 \mid fix_j, pmtn, res1 \cdot 1 \mid$ C_{max}.

Problem $P \mid fix_j \mid C_{max}$ is considered in [25]. For special cases $P2, 3, 4 \mid$ $fix_j, p_j = 1 \mid C_{max}$ linear time algorithms are given, for $P5 \mid fix_j, p_j = 1 \mid$ C_{max}, $O(n^{2.5})$ algorithm is given. The general case is analyzed on the base of incompatibility graph. A special easy case is identified: when incompatibility graph is a comparability graph, the problem is solvable in polynomial time. For a general case B&B algorithm is proposed. The idea of augmenting incompatibility graph to a comparability graph is further used in [77] to examine problem $P \mid fix_j, prec \mid C_{max}$.

In [120] computational complexity of a group of multiprocessor task scheduling problems for C_{max} and $\sum c_j$ criteria is considered (the work is known since 1992).

Scheduling file transfers in time windows (i.e. $P, win \mid fix_j \mid C_{max}$ and $\mid fix_j \mid = 2$) is analyzed in [136]. The problem is shown to be **NP**h. Lower and upper bounds on the optimal schedule length are proposed. Three polynomially solvable cases are identified.

In [22] polynomially solvable cases of scheduling unit-execution time tasks are considered. Linear time algorithm is given for problem $P2 \mid fix_j, p_j = 1 \mid L_{max}$.

In [85] problem $P2 \mid fix_j \mid L_{max}$ is shown to be s**NP**h.

Scheduling according to model set_j is tackled in [23]. Dynamic programming formulations are given for $P2 \mid set_j \mid C_{max}$ and for $P3 \mid set_j \mid C_{max}$ in the absence of one of the three duoprocessor task types. For $P \mid set_j \mid C_{max}$ heuristic scheduling tasks in the shortest processing time mode (SPTM) is proposed. Its tight performance ratio is m. For $P \mid set_j, pmtn \mid C_{max}$ a polynomial time algorithm based on processor feasible sets and linear programming is proposed.

In [26] the complexity of problem $P \mid set_j \mid C_{max}$ is analyzed. Methods of calculating lower and upper bounds on the length of the schedule are proposed. A heuristic method solving iteratively separate subproblems: the assignment (selection of processing mode) and scheduling problem, is proposed.

Article [138] analyzes the complexity of a wide range of preemptive and nonpreemptive scheduling problems with $\mid fix_j \mid = 2$.

In [51] for problem $P2 \mid fix_j, pmtn \mid \sum c_j$ an $O(n \log n)$ optimization algorithm is given. $P2 \mid fix_j \mid \sum c_j$ is proved to be s**NP**h, and a heuristic with performance bound 2 is proposed.

The new results presented in this section as well as the previously existing ones are summarized in Table 5.2.

5.3.2 Low Complexity Algorithms for Maximum Lateness

We assume that there are s different values of due-dates: $d_1 < d_2 < \ldots < d_s$. We will say that tasks with due-dates equal to d_i must finish in the i-th interval, because they must not be finished later than in interval $[d_{i-1} + L_{max}, d_i + L_{max}]$ for $i = 1, \ldots, s$, where $d_0 = -L_{max}$. Without loss of generality we assume in this section that there are no two tasks with the same due-date and the same set of required processors (such tasks can be analyzed as one task with execution time equal to the sum of execution times). The task with due-date d_i and requiring set D of processors will be denoted $T^{D,i}$ and $t^{D,i}$ will denote its processing time. Moreover, we assume that tasks are ordered according to nondecreasing values of their due-dates.

$P2 \mid fix_j, pmtn \mid L_{max}$

In this section we give a formulation of the algorithm for the case of tasks requiring either one of the two processors or both of them. The following theorem establishes conditions under which a feasible schedule with value L of lateness can be built.

Theorem 5.5 *For existence of a feasible schedule for problem $P2|fix_j,pmtn|L_{max}$ with maximum lateness value equal to L it is necessary and sufficient to guarantee that the following set of inequalities holds:*

$$\sum_{j=1}^{i}(t^{12,j}+t^{1,j}) \leq d_i + L \quad \text{for} \quad i=1,\ldots,s \tag{5.6}$$

$$\sum_{j=1}^{i}(t^{12,j}+t^{2,j}) \leq d_i + L \quad \text{for} \quad i=1,\ldots,s \tag{5.7}$$

Proof Inequalities (5.6), (5.7) establish necessary conditions for schedule feasibility because there are processing requirements of processors P_1 and P_2 on their left-hand sides. On the right-hand sides, there are processing capacities of processors in periods $[0, d_i + L]$ $(i = 1, \ldots, s)$. Hence, no schedule with smaller value of lateness can exist. Now, we will show that when conditions (5.6) and (5.7) are satisfied then a feasible schedule exists. The proof is given by induction over index i of the interval.

For $i = 1$ the inequalities (5.6), (5.7) have the form:

$$\begin{aligned} t^{12,1}+t^{1,1} &\leq d_1 + L \\ t^{12,1}+t^{2,1} &\leq d_1 + L \end{aligned}$$

From [21] we know that for problem $P2 \mid fix_j, pmtn \mid C_{max}$ the optimal *length* of the schedule is equal to $C^1_{max} = \max\{t^{12,1}+t^{1,1}, t^{12,1}+t^{2,1}\}$. From (5.6), (5.7) we get $d_1 + L \geq C^1_{max}$, and a feasible schedule can be built in the first interval for the given L (cf. Fig. 5.3a).

Now let us assume, that a feasible schedule for tasks finishing in intervals $1, \ldots, i$ exists and inequalities (5.6), (5.7) are satisfied for $1, \ldots, i+1$. Then, a feasible schedule for tasks with due-date d_{i+1} must also exist. Suppose no feasible schedule for the tasks finishing in the interval $i+1$ exists. This means that one of the following inequalities must hold (cf. Fig. 5.3b):

$$t^{12,i+1}+t^{1,i+1} > d_{i+1} + L - \sum_{j=1}^{i}(t^{12,j}+t^{1,j}) \tag{5.8}$$

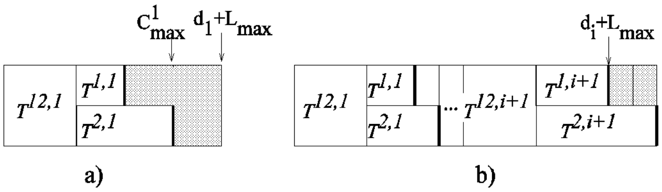

Figure 5.3: Partial schedule for $P2 \mid fix_j, pmtn \mid L_{max}$.

$$t^{12,i+1} + t^{2,i+1} \quad > \quad d_{i+1} + L - \sum_{j=1}^{i}(t^{12,j} + t^{2,j}) \tag{5.9}$$

But (5.8) is in contradiction with (5.6), and (5.9) with (5.7). We conclude that also for the tasks finishing in interval $i+1$ a feasible schedule must exist. Induction on i completes the proof. □

From the above theorem we conclude that the optimal lateness L^*_{max} can be found as a minimal value of L which is satisfying inequalities (5.6), (5.7). Since there are $O(n)$ inequalities in (5.6),(5.7), L^*_{max} can be found in $O(n)$ time. The optimal schedule can be built following the scheme presented in Fig. 5.3. $T^{12,i+1}$ is scheduled as soon as tasks with the due-date d_i are finished. Tasks $T^{1,i+1}$ and $T^{2,i+1}$ are shifted to the left as far as possible. Then, tasks from interval $(i+2)$ follow immediately. We will name this method *interval scheduling*. The schedule can be built in $O(n)$ time. In order to achieve this, the search for free time slots must be completed in $O(n)$ time for all n tasks. It is possible when the scheduling algorithm holds a list of free time slots. The time spent on finding appropriate time slots is proportional to the number of considered slots. Since no time slot is considered after it is completely allocated, and there are at most $\lceil n/2 \rceil$ free time slots on one processor, the schedule can be constructed in $O(n)$ time.

$P3 \mid fix_j, pmtn \mid L_{max}$

In this section we consider three processor case. The problem can be solved in linear time for the instances with the following property which will be called *accommodation property*:

$$\begin{aligned} \sum_{j=1}^{i} t^{1,j} > \sum_{j=1}^{i} t^{23,j} \quad &\Rightarrow \quad t^{1,i+1} > t^{23,i+1} \text{ for } i = 1,\dots,s \\ \sum_{j=1}^{i} t^{2,j} > \sum_{j=1}^{i} t^{13,j} \quad &\Rightarrow \quad t^{2,i+1} > t^{13,i+1} \text{ for } i = 1,\dots,s \end{aligned} \tag{5.10}$$

$$\sum_{j=1}^{i} t^{3,j} > \sum_{j=1}^{i} t^{12,j} \quad \Rightarrow \quad t^{3,i+1} > t^{12,i+1} \text{ for } i = 1, \ldots, s.$$

This means that if in some interval uniprocessor tasks are executed longer than duoprocessor tasks requiring the remaining two processors, then also in the following intervals this situation takes place. The following theorem states necessary and sufficient conditions for the existence of a schedule with the given value of maximum lateness.

Theorem 5.6 *For existence of a feasible schedule for problem $P3 \mid fix_j$, $pmtn \mid L_{max}$ with maximum lateness equal to L and instance with accommodation property, it is necessary and sufficient that the following set of inequalities holds:*

$$\begin{aligned}
\sum_{j=1}^{i}(t^{123,j} + t^{12,j} + t^{13,j} + t^{1,j}) &\leq d_i + L \text{ for } i = 1, \ldots, s \\
\sum_{j=1}^{i}(t^{123,j} + t^{12,j} + t^{23,j} + t^{2,j}) &\leq d_i + L \text{ for } i = 1, \ldots, s \\
\sum_{j=1}^{i}(t^{123,j} + t^{13,j} + t^{23,j} + t^{3,j}) &\leq d_i + L \text{ for } i = 1, \ldots, s \\
\sum_{j=1}^{i}(t^{123,j} + t^{12,j} + t^{13,j} + t^{23,j}) &\leq d_i + L \text{ for } i = 1, \ldots, s.
\end{aligned} \tag{5.11}$$

Proof Observe that no schedule with maximum lateness smaller than L satisfying (5.11) can exist. Otherwise, tasks would have to overlap. We will show by induction over interval number i that for L satisfying inequalities (5.11) a feasible schedule exists.

Let us analyze $i = 1$. According to [21], where problem $P3 \mid fix_j, pmtn \mid C_{max}$ has been analyzed, the shortest schedule in the first interval has length $C^1_{max} = t^{123,1} + \max\{t^{12,1} + t^{13,1} + t^{1,1}, t^{12,1} + t^{23,1} + t^{2,1}, t^{13,1} + t^{23,1} + t^{3,1}, t^{12,1} + t^{13,1} + t^{23,1}\}$.
From (5.11) we have $C^1_{max} \leq d_1 + L$, and a feasible schedule can be built in the first interval.

Next, assume that for tasks finishing in intervals $1, \ldots, i$ a feasible schedule exists and inequalities (5.11) are satisfied for intervals $1, \ldots, i+1$. We show that a feasible schedule for tasks with due-date d_{i+1} must also exist.

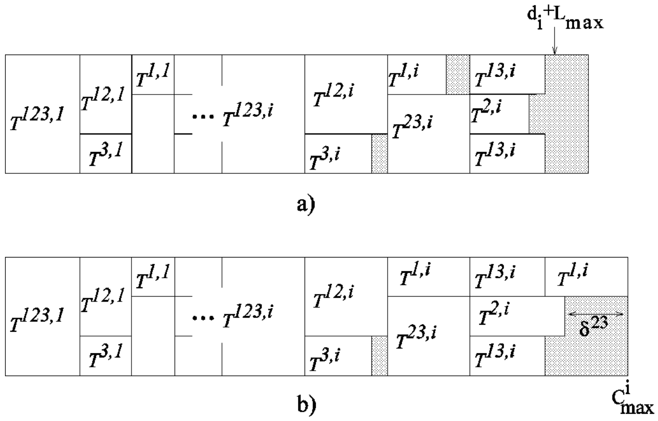

Figure 5.4: Partial schedule for $P3 \mid fix_j, pmtn \mid L_{max}$.

Suppose no feasible schedule for tasks finishing in interval $i+1$ exists. We will analyze each type of tasks according to the number of used processors.

Case A. Some uniprocessor task(s) with due-date d_{i+1} cannot be scheduled feasibly. Without loss of generality let it be task $T^{1,i+1}$. This means that
$t^{123,i+1}+t^{12,i+1}+t^{13,i+1}+t^{1,i+1} > d_{i+1}+L-\sum_{j=1}^{i}(t^{123,j}+t^{12,j}+t^{13,j}+t^{1,j})$,
which contradicts (5.11). We conclude that task $T^{1,i+1}$ can be scheduled feasibly. In the same manner one can prove the existence of feasible schedules for tasks $T^{2,i+1}$ and $T^{3,i+1}$.

Case B. Some duoprocessor task(s) with due-date d_{i+1} cannot be scheduled. We will analyze two subcases: *B.1* - length of the schedule for tasks finishing in the intervals $1,\ldots,i$ (denoted C^i_{max}) was established by the processing times of tripleprocessor or duoprocessor tasks (cf. Fig. 5.4a); *B.2* - C^i_{max} was imposed by processing time on a single processor (cf. Fig. 5.4b).

Subcase B.1 Assume task(s) from $T^{12,i+1}$ cannot be scheduled. This means that one of the three inequalities must be satisfied:
$t^{123,i+1}+t^{12,i+1}+t^{13,i+1}+t^{1,i+1} > d_{i+1}+L-\sum_{j=1}^{i}(t^{123,j}+t^{12,j}+t^{13,j}+t^{1,j})$
$t^{123,i+1}+t^{12,i+1}+t^{23,i+1}+t^{2,i+1} > d_{i+1}+L-\sum_{j=1}^{i}(t^{123,j}+t^{12,j}+t^{23,j}+t^{2,j})$
$t^{123,i+1}+t^{12,i+1}+t^{13,i+1}+t^{23,i+1} > d_{i+1}+L-\sum_{j=1}^{i}(t^{123,j}+t^{12,j}+t^{13,j}+t^{23,j})$.
The former two inequalities can be excluded from further analysis because also some uniprocessor task would not be scheduled feasibly, which is impossible according to *Case A*. The latter inequality contradicts (5.11). Hence, $T^{12,i+1}$ can be scheduled feasibly in this subcase. Analogous proof can be given for $T^{13,i+1}$ and $T^{23,i+1}$.

Subcase B.2 C^i_{max} was imposed by a single processor. Without loss of generality let it be P_1. Suppose $T^{12,i+1}$ cannot be scheduled. We exclude at this point *Case A* (i.e. the fact that $T^{12,i+1}$ cannot be scheduled due to some uniprocessor task). Denote by δ^{23} the length of the interval in which $T^{23,i+1}$ can be processed before moment C^i_{max}. Hence,
$\delta^{23} = C^i_{max} - \max\{\sum_{j=1}^i (t^{123,j} + t^{12,j} + t^{13,j} + t^{23,j}), \sum_{j=1}^i (t^{123,j} + t^{12,j} + t^{23,j} + t^{2,j}), \sum_{j=1}^i (t^{123,j} + t^{13,j} + t^{23,j} + t^{3,j})\}$.
Since $T^{12,i+1}$ cannot be scheduled feasibly, we have:

$$t^{123,i+1} + t^{12,i+1} + t^{13,i+1} + \max\{0, t^{23,i+1} - \delta^{23}\} > d_{i+1} + L - C^i_{max}. \quad (5.12)$$

Suppose that $t^{23,i+1} \geq \delta^{23}$ then by substituting δ^{23} in (5.12) we get:

$$\begin{aligned} t^{123,i+1} + t^{12,i+1} + t^{13,i+1} + t^{23,i+1} \\ -C^i_{max} + \max\{\sum_{j=1}^{i} (t^{123,j} + t^{12,j} + t^{13,j} + t^{23,j}), \\ \sum_{j=1}^{i} (t^{123,j} + t^{12,j} + t^{23,j} + t^{2,j}), \sum_{j=1}^{i} (t^{123,j} + t^{13,j} + t^{23,j} + t^{3,j})\} > \\ d_{i+1} + L - C^i_{max}. \quad (5.13) \end{aligned}$$

Assume that the first term in the *max* component of the above inequality is grater, then we obtain

$$\sum_{j=1}^{i+1} (t^{123,j} + t^{12,j} + t^{13,j} + t^{23,j}) > d_{i+1} + L$$

which contradicts (5.11). Consider the second term of the *max* component in (5.13) as maximum. This may happen only if $\sum_{j=1}^i t^{13,j} < \sum_{j=1}^i t^{2,j}$. Then, we have

$$\begin{aligned} t^{123,i+1} + t^{12,i+1} + t^{13,i+1} + t^{23,i+1} + \\ \sum_{j=1}^{i} (t^{123,j} + t^{12,j} + t^{23,j} + t^{2,j}) > \\ d_{i+1} + L. \end{aligned}$$

And from this

$$\sum_{j=1}^{i+1} (t^{123,j} + t^{12,j} + t^{23,j} + t^{2,j}) - t^{2,i+1} + t^{13,i+1} > d_{i+1} + L.$$

From (5.11) we have $d_{i+1} + L \geq \sum_{j=1}^{i+1}(t^{123,j} + t^{12,j} + t^{23,j} + t^{2,j})$ and the above two inequalities together give $t^{13,i+1} > t^{2,i+1}$, which contradicts accommodation property. For the last component of max term in (5.13) the reasoning is analogous.

Now, suppose that $t^{23,i+1} < \delta^{23}$. Then, (5.12) takes the form:

$$t^{123,i+1} + t^{12,i+1} + t^{13,i+1} > d_{i+1} + L - C^i_{max}.$$

Since the length of the schedule for tasks finishing in the intervals $1, \ldots, i$ was imposed by a uniprocessor task using processor P_1, C^i_{max} is

$$C^i_{max} = \sum_{j=1}^{i}(t^{123,j} + t^{12,j} + t^{13,j} + t^{1,j}).$$

From the above two formulations we get
$t^{123,i+1} + t^{12,i+1} + t^{13,i+1} > d_{i+1} + L - \sum_{j=1}^{i}(t^{123,j} + t^{12,j} + t^{13,j} + t^{1,j})$, which contradicts (5.11). Thus, a feasible schedule for $T^{12,i+1}$ must exist. The same reasoning can be applied to $T^{13,i+1}$ and $T^{23,i+1}$ because inequality (5.12) must hold when $T^{13,i+1}$ and $T^{23,i+1}$ cannot be scheduled. This completes *Subcase B.2* and *Case B*.

Case C. Suppose some tripleprocessor task(s) cannot be scheduled. We can exclude from further analysis the case when a tripleprocessor task cannot be scheduled with uniprocessor and/or duoprocessor task(s), because these cases have already been analyzed (*Case A, B*). Hence, we get:

$$t^{123,i+1} > d_{i+1} + L_{max} - C^i_{max}$$

which implies (5.12). Thus, a feasible schedule for tripleprocessor tasks must exist. This proves the existence of a feasible schedule for tasks finishing in interval $i+1$. Induction on i completes the proof. □

From the above theorem we conclude that when inequalities (5.10) hold, the optimal schedule can be obtained in $O(n)$ time. L^*_{max} can be found as a minimal value L satisfying inequalities (5.11). Task $T^{123,i+1}$ must be executed immediately after all tasks from interval i are finished. Duoprocessor tasks from interval $i+1$ are shifted to the left as much as possible. Finally, uniprocessor tasks follow shifted as much as possible to the left. In the next interval tasks are scheduled in the same manner. Again, we will name this method *interval scheduling*.

When conditions (5.10) do not hold, it is possible that inequalities (5.11) are not sufficient to reflect interactions between tasks in consecutive intervals. For example, task $T^{2,i}$ influences completion time of tasks $T^{13,i+1}$, though they never appear together in (5.11). In such a situation it is easier to apply linear programming approach proposed in [19] or Section 5.3.3. One more explanation why the interval scheduling algorithm does not guarantee optimality is the following. In each partial schedule of tasks from intervals $1,\dots,i$ there are free time slots which provide processing capacity for uni- and duoprocessor tasks. When there is no longer free space for uniprocessor task it must be allocated in the slots accessible for duo- and triple-processor tasks. Depending on the choice of the slot free time intervals for duoprocessor tasks are consumed. Since the kind of duoprocessor tasks that follow in the next interval(s) is not considered during the construction of a partial schedule, it is not possible to build in this way an optimal schedule for all cases.

We will show now that even though interval scheduling does not build optimal schedules in all cases, it is still delivering solutions of good quality. Namely, we will show that in the worst case, the relative difference between optimum L^*_{max} and maximum lateness L^{IS}_{max} of the schedule built by the interval scheduling algorithm is bounded. Let us denote by C^i_{max} the completion time of the last task from interval i.

Theorem 5.7 *Every schedule built by the interval scheduling algorithm for problem $P3 \mid fix_j, pmtn \mid L_{max}$ satisfies*

$$\frac{L^{IS}_{max} - d_j}{L^*_{max}} \le 2, \quad \text{where } j \text{ satisfies } C^j_{max} = L^{IS}_{max} + d_j.$$

Proof Firstly, an upper bound on L^{IS}_{max} will be calculated. Note that $L^{IS}_{max} = \max_i\{C^i_{max} - d_i\}$. Completion time C^i_{max} can be bounded from above by $\sum_{j=1}^{i}(t^{123,j} + t^{12,j} + t^{23,j} + t^{13,j} + t^{1,j} + t^{2,j} + t^{3,j})$ in which we assume that no tasks are executed in parallel. The lower bound for $L^*_{max} + d_i$ is the length of the shortest feasible schedule of tasks from intervals $1,\dots,i$. According to [21] it is

$$\max_i\left\{\begin{array}{l} \sum_{j=1}^{i}(t^{123,j} + t^{12,j} + t^{13,j} + t^{1,j}) \\ \sum_{j=1}^{i}(t^{123,j} + t^{12,j} + t^{23,j} + t^{2,j}) \\ \sum_{j=1}^{i}(t^{123,j} + t^{13,j} + t^{23,j} + t^{3,j}) \\ \sum_{j=1}^{i}(t^{123,j} + t^{12,j} + t^{13,j} + t^{23,j}) \end{array}\right\} \tag{5.14}$$

Suppose the last term is maximum, then from comparing it with the previous three terms we have $\sum_{j=1}^{i} t^{1,j} \le \sum_{j=1}^{i} t^{23,j}$ and $\sum_{j=1}^{i} t^{2,j} \le \sum_{j=1}^{i} t^{13,j}$ and $\sum_{j=1}^{i} t^{3,j} \le \sum_{j=1}^{i} t^{12,j}$. Thus, we get

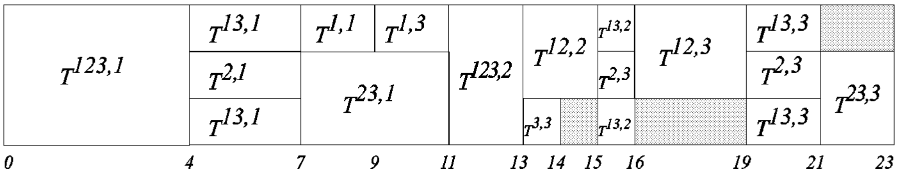

Figure 5.5: Example schedule for problem $P3 \mid fix_j, pmtn \mid L_{max}$.

$$\frac{C^i_{max}}{L^*_{max}+d_i} < \frac{\sum_{j=1}^{i}(t^{123,j}+t^{12,j}+t^{23,j}+t^{13,j}+t^{1,j}+t^{2,j}+t^{3,j})}{\sum_{j=1}^{i}(t^{123,j}+t^{12,j}+t^{13,j}+t^{23,j})} \leq$$

$$1+\frac{\sum_{j=1}^{i}(t^{1,j}+t^{2,j}+t^{3,j})}{\sum_{j=1}^{i}(t^{123,j}+t^{12,j}+t^{13,j}+t^{23,j})} \leq 1+\frac{\sum_{j=1}^{i}(t^{1,j}+t^{2,j}+t^{3,j})}{\sum_{j=1}^{i}(t^{123,j}+t^{1,j}+t^{2,j}+t^{3,j})} \leq 2.$$

If any other term in a given interval is maximum in (5.14), similar arguments follow. Next, there must exist at least one interval j satisfying $C^j_{max} = L^{IS}_{max}+d_j$. Hence, $C^j_{max} = L^{IS}_{max}+d_j \leq 2(L^*_{max}+d_j)$ and from this $\frac{L^{IS}_{max}-d_j}{L^*_{max}} \leq 2$. $\square$

We complete this section with an example.

Example

We are given 14 tasks. Tasks with due-date $d_1 = 2$ (we enumerate only processing times): $t^{123,1} = 4, t^{13,1} = 3, t^{23,1} = 4, t^{1,1} = 2, t^{2,1} = 3$; tasks with due-date $d_2 = 4$: $t^{123,2} = 2, t^{12,2} = 2, t^{13,2} = 1$; tasks with due-date $d_3 = 6$: $t^{12,3} = 3, t^{13,3} = 2, t^{23,3} = 2, t^{1,3} = 2, t^{2,3} = 3, t^{3,3} = 1$. As it can be verified, this instance has accommodation property. Inequalities (5.11) are satisfied by values of $L_{max} \geq 17$. The optimal schedule is presented in Fig. 5.5.

$P4 \mid fix_j, pmtn \mid L_{max}$

In this section scheduling on four processors will be considered. Let us introduce some additional notation. By $\mathcal{T}^{D,i}$ we will denote the set of tasks in the intervals $1,\dots,i$ requiring processors from set D, i.e. $\mathcal{T}^{D,i} = \cup_{j=1}^{i} T^{D,j}$. A *competition graph* is a graph in which nodes correspond to task types and edges connect nodes (i.e. task types) which cannot be executed in parallel. Consider a competition graph built for tasks from intervals $1,\dots,i$. One can distinguish in such a graph twelve cliques - groups of tasks that must not be executed in parallel. These are:

$A^i = \{\mathcal{T}^{1,i}, \mathcal{T}^{12,i}, \mathcal{T}^{13,i}, \mathcal{T}^{14,i}, \mathcal{T}^{123,i}, \mathcal{T}^{124,i}, \mathcal{T}^{134,i}, \mathcal{T}^{1234,i}\}$,
$B^i = \{\mathcal{T}^{2,i}, \mathcal{T}^{12,i}, \mathcal{T}^{23,i}, \mathcal{T}^{24,i}, \mathcal{T}^{123,i}, \mathcal{T}^{124,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\}$,
$C^i = \{\mathcal{T}^{3,i}, \mathcal{T}^{13,i}, \mathcal{T}^{23,i}, \mathcal{T}^{34,i}, \mathcal{T}^{123,i}, \mathcal{T}^{134,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\}$,
$D^i = \{\mathcal{T}^{4,i}, \mathcal{T}^{14,i}, \mathcal{T}^{24,i}, \mathcal{T}^{34,i}, \mathcal{T}^{124,i}, \mathcal{T}^{134,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\}$,

$$
\begin{aligned}
&E^i = \{\mathcal{T}^{12,i}, \mathcal{T}^{13,i}, \mathcal{T}^{23,i}, \mathcal{T}^{123,i}, \mathcal{T}^{124,i}, \mathcal{T}^{134,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\},\\
&F^i = \{\mathcal{T}^{12,i}, \mathcal{T}^{14,i}, \mathcal{T}^{24,i}, \mathcal{T}^{123,i}, \mathcal{T}^{124,i}, \mathcal{T}^{134,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\},\\
&G^i = \{\mathcal{T}^{13,i}, \mathcal{T}^{14,i}, \mathcal{T}^{34,i}, \mathcal{T}^{123,i}, \mathcal{T}^{124,i}, \mathcal{T}^{134,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\},\\
&H^i = \{\mathcal{T}^{23,i}, \mathcal{T}^{24,i}, \mathcal{T}^{34,i}, \mathcal{T}^{123,i}, \mathcal{T}^{124,i}, \mathcal{T}^{134,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\},\\
&I^i = \{\mathcal{T}^{12,i}, \mathcal{T}^{13,i}, \mathcal{T}^{14,i}, \mathcal{T}^{123,i}, \mathcal{T}^{124,i}, \mathcal{T}^{134,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\},\\
&J^i = \{\mathcal{T}^{12,i}, \mathcal{T}^{23,i}, \mathcal{T}^{24,i}, \mathcal{T}^{123,i}, \mathcal{T}^{124,i}, \mathcal{T}^{134,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\},\\
&K^i = \{\mathcal{T}^{13,i}, \mathcal{T}^{23,i}, \mathcal{T}^{34,i}, \mathcal{T}^{123,i}, \mathcal{T}^{124,i}, \mathcal{T}^{134,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\},\\
&L^i = \{\mathcal{T}^{14,i}, \mathcal{T}^{24,i}, \mathcal{T}^{34,i}, \mathcal{T}^{123,i}, \mathcal{T}^{124,i}, \mathcal{T}^{134,i}, \mathcal{T}^{234,i}, \mathcal{T}^{1234,i}\}.
\end{aligned}
$$

To guarantee optimality of the schedule for four processors, built in the same way as for two and three processors, the instance of the problem must satisfy more restrictive conditions:

$$
\begin{aligned}
t^{12,i} = t^{34,i}, t^{13,i} = t^{24,i}, t^{14,i} = t^{23,i} &\qquad \text{for } i = 1, \ldots, s\\
\sum_{j=1}^{i} t^{1,j} \le \sum_{j=1}^{i} t^{234,j} &\qquad \text{for } i = 1, \ldots, s\\
\sum_{j=1}^{i} t^{2,j} \le \sum_{j=1}^{i} t^{134,j} &\qquad \text{for } i = 1, \ldots, s \qquad (5.15)\\
\sum_{j=1}^{i} t^{3,j} \le \sum_{j=1}^{i} t^{124,j} &\qquad \text{for } i = 1, \ldots, s\\
\sum_{j=1}^{i} t^{4,j} \le \sum_{j=1}^{i} t^{123,j} &\qquad \text{for } i = 1, \ldots, s.
\end{aligned}
$$

As before our problem can be solved by analysis of a set of inequalities.

Theorem 5.8 *For the instances of problem $P4 \mid fix_j, pmtn \mid L_{max}$ satisfying conditions (5.15) a feasible schedule with maximum lateness equal to L exists if and only if the following set of inequalities holds:*

$$
\max_{\mathcal{J} \in \{A^i, \ldots, L^i\}} \{ \sum_{S \in \mathcal{J}, T_j^K \in S} t_j^K \} \le d_i + L \ \text{ for } \ i = 1, \ldots, s. \qquad (5.16)
$$

Proof $A^i, \ldots, L^i$ are cliques of tasks which means that tasks in each of these sets must be executed sequentially, and no schedule with maximum lateness smaller than the value satisfying (5.16) can exist. We will show by induction on interval number i that as long as inequalities (5.15),(5.16) hold, a feasible schedule must exist.

Consider the first interval ($i = 1$). From (5.16) and [21] where length C^1_{max} of optimal schedule for problem $P4 \mid fix_j, pmtn \mid C_{max}$ has been established, we have

$$d_1 + L \geq C^1_{max} = \max_{\mathcal{J} \in \{A^1,\dots,L^1\}} \{ \sum_{S \in \mathcal{J}, T^K_j \in S} t^K_j \}$$

and a feasible schedule can be built in the first interval.

Assume now, that a feasible schedule for tasks from the intervals $1, \dots, i$ exists and the inequalities (5.16) are satisfied for the intervals $1, \dots, i+1$. We will show that a feasible schedule must also exist for interval $i+1$. On the contrary, suppose that some task(s) cannot be scheduled. We will analyze task types according to the number of processors used.

Case A. Some uniprocessor task(s) cannot be scheduled. Let it be $T^{1,i+1}$, without loss of generality. Then, the following inequality must hold:
$t^{1,i+1} + t^{12,i+1} + t^{13,i+1} + t^{14,i+1} + t^{123,i+1} + t^{124,i+1} + t^{134,i+1} + t^{1234,i+1} >$
$d_{i+1} + L_{max} - \sum_{j=1}^{i}(t^{1,j} + t^{12,j} + t^{13,j} + t^{14,j} + t^{123,j} + t^{124,j} + t^{134,j} + t^{1234,j})$, which contradicts (5.16). We conclude that a feasible schedule for $T^{1,i+1}$ must exist. For other uniprocessor tasks types reasoning is similar.

Case B. Some duoprocessor task(s) cannot be scheduled feasibly. We can exclude from further analysis the case for which duoprocessor tasks cannot be scheduled due to some uniprocessor task(s) - since this is *Case A*. Hence, we can also exclude from further analysis violation of the schedule feasibility by the tasks forming cliques of type $A^{i+1}, \dots, D^{i+1}$. The rest of the proof for *Case B* has two parts *Subcase B.1* - when C^i_{max} was imposed by tripleprocessor or duoprocessor tasks and *Subcase B.2* - C^i_{max} was imposed by uniprocessor tasks.

Subcase B.1. C^i_{max} was imposed by duoprocessor or tripleprocessor tasks, thus it is a sum of processing requirements of one of cliques $E^i, \dots, L^i$. For instances satisfying equations (5.15), sums of processing times of tasks in cliques $E^i, \dots, L^i$ are the same. Hence,

$$C^i_{max} = \sum_{S \in E^i, T^{K,j} \in S} t^{K,j}.$$

Suppose some duoprocessor task cannot be scheduled. This means that for some clique with duoprocessor and tripleprocessor tasks schedule is infeasible. Let it be a clique of the E type ($E^{i+1} - E^i$, to be precise) for example. Then, we have
$t^{1234,i+1} + t^{123,i+1} + t^{124,i+1} + t^{134,i+1} + t^{234,i+1} + t^{12,i+1} + t^{13,i+1} + t^{23,i+1} >$
$d_{i+1} + L - C^i_{max} =$

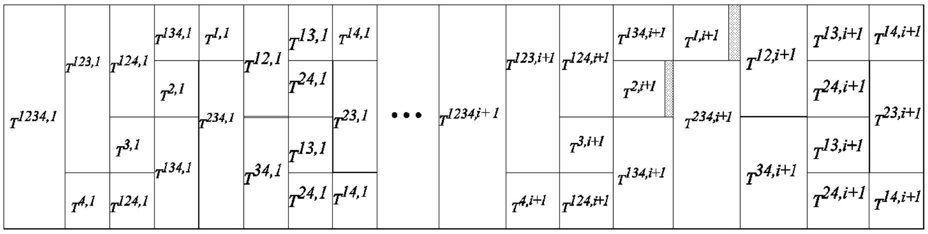

Figure 5.6: Partial schedule for the proof of Theorem 5.8.

$d_{i+1} + L - \sum_{j=1}^{i}(t^{1234,j} + t^{123,j} + t^{124,j} + t^{134,j} + t^{234,j} + t^{12,j} + t^{13,j} + t^{23,j})$
$\geq t^{1234,i+1} + t^{123,i+1} + t^{124,i+1} + t^{134,i+1} + t^{234,i+1} + t^{12,i+1} + t^{13,i+1} + t^{23,i+1}$
which contradicts (5.16). For other clique types, the proof is analogous.

Subcase B.2. This subcase cannot happen when inequalities (5.15) hold. Hence, duoprocessor tasks can be feasibly executed in the interval $i+1$.

Case C. Suppose some tripleprocessor task(s) cannot be scheduled. We exclude situations that tripleprocessor task(s) cannot be scheduled due to some uni- or duoprocessor task(s) since these are *Case A* or *Case B*, respectively. Thus, the following inequality must hold (cf. Fig. 5.6):
$t^{1234,i+1} + t^{123,i+1} + t^{124,i+1} + t^{134,i+1} + t^{234,i+1} > d_{i+1} + L - C^{i}_{max}$
But from (5.15)
$C^{i}_{max} = \sum_{j=1}^{i}(t^{1234,j} + t^{123,j} + t^{124,j} + t^{134,j} + t^{234,j} + t^{12,j} + t^{13,j} + t^{23,j})$
and we have a contradiction with (5.16). Hence, a feasible schedule for tripleprocessor task(s) must also exist.

Case D. Some four-processor task(s) cannot be scheduled. If we exclude the cases caused by some uni- or duo- or triple-processor tasks, then the remaining situations contradict (5.16). Hence, the theorem follows. □

The optimal value L^{*}_{max} can be found in linear time as minimal L satisfying inequalities (5.16). The optimal schedule has form presented in Fig. 5.6: four-processor tasks $T^{1234,i+1}$ are scheduled as soon as tasks from interval i are finished, then tripleprocessor tasks are shifted as much to the left as possible. Next, duoprocessor tasks are executed. Finally, uniprocessor tasks follow. After scheduling uniprocessor tasks there is no idle time to the left from scheduled uniprocessor tasks.

One may ask what would happen if inequalities (5.15) were not satisfied. A schedule built in the above way would not be optimal in general and inequalities (5.16) would not deliver L^{*}_{max}. Without (5.15) it is difficult to give a simple (and independent of the instance) set of rules which would guarantee optimality of the above algorithm. In such situations it is simpler to apply linear programming approach ([19] or Section 5.3.3). As in the previous section we will prove that the worst case solutions generated by interval scheduling algorithm have maximum lateness (L^{IS}_{max}) within some bounded vicinity of the optimum (L^{*}_{max}).

Theorem 5.9 *For any schedule generated by the interval scheduling algorithm for problem $P4 \mid fix_j, pmtn \mid L_{max}$ the following holds*

$$\frac{L^{IS}_{max} - 3d_j}{L^{*}_{max}} \leq 4, \quad \text{where } j \text{ satisfies } C^{j}_{max} = L^{IS}_{max} + d_j.$$

Proof In each interval a lower bound on $L^{*}_{max} + d_i$ is the sum of processing times of tasks forming cliques. In this proof we will distinguish three cliques A^i, E^i and I^i as representatives for $A^i, \ldots, L^i$. For other cliques the proof is similar.

Case A. Clique A^i is maximal in interval i. Processing time of tasks in this clique is a lower bound on $L^{*}_{max} + d_i$. The upper bound on C^{i}_{max} can be calculated assuming that all tasks from $T^{D,j}$ are executed before tasks from $T^{D,j+1}$ and after tasks from $T^{D,j-1}$ ($j < i$). Furthermore, the period of executing tasks from $T^{D,j}$ can be calculated as the sum of processing times of tasks in the $A^j - A^{j-1}$ clique plus some excess of the processing time which cannot be scheduled in parallel with clique $A^j - A^{j-1}$. Thus, the upper bound on C^{i}_{max} is

$$\sum_{j=1}^{i}(t^{1234,j} + t^{123,j} + t^{124,j} + t^{134,j} + t^{12,j} + t^{13,j} + t^{14,j} + t^{1,j} +$$
$$\max\begin{cases} t^{234,j} + t^{23,j} + t^{24,j} + t^{2,j} - t^{134,j} - t^{1,j} - t^{13,j} - t^{14,j} \\ t^{234,j} + t^{23,j} + t^{34,j} + t^{3,j} - t^{124,j} - t^{1,j} - t^{12,j} - t^{14,j} \\ t^{234,j} + t^{24,j} + t^{34,j} + t^{4,j} - t^{123,j} - t^{1,j} - t^{12,j} - t^{13,j} \\ t^{234,j} + t^{23,j} + t^{24,j} + t^{34,j} - t^{1,j} - t^{12,j} - t^{13,j} - t^{14,j} \\ 0 \end{cases}) \leq$$
$$\sum_{j=1}^{i}(t^{1234,j} + t^{123,j} + t^{124,j} + t^{134,j} + t^{12,j} + t^{13,j} + t^{14,j} + t^{1,j} + t^{234,j} + t^{23,j} + t^{24,j} + t^{34,j} + t^{2,j} + t^{3,j} + t^{4,j}).$$

On the other hand, comparing A^i with B^i, C^i, D^i yields:

$\sum_{j=1}^{i}(t^{134,j} + t^{14,j} + t^{13,j} + t^{1,j}) \geq \sum_{j=1}^{i}(t^{234,j} + t^{23,j} + t^{2,j})$,
$\sum_{j=1}^{i}(t^{1,j} + t^{12,j} + t^{14,j} + t^{124,j}) \geq \sum_{j=1}^{i}(t^{34,j} + t^{3,j})$,
$\sum_{j=1}^{i}(t^{1,j} + t^{13,j} + t^{12,j} + t^{123,j}) \geq \sum_{j=1}^{i}(t^{4,j} + t^{24,j})$.

Thus, we have

$$\frac{C^{i}_{max}}{L^{*}_{max}+d_i} \leq$$
$$\frac{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{12,j}+t^{13,j}+t^{14,j}+t^{1,j}+t^{234,j}+t^{23,j}+t^{24,j}+t^{34,j}+t^{2,j}+t^{3,j}+t^{4,j})}{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{12,j}+t^{13,j}+t^{14,j}+t^{1,j})} \leq$$

$1+\frac{\sum_{j=1}^{i}(t^{234,j}+t^{23,j}+t^{24,j}+t^{34,j}+t^{2,j}+t^{3,j}+t^{4,j})}{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{12,j}+t^{13,j}+t^{14,j}+t^{1,j})} \leq$
$1+\frac{\sum_{j=1}^{i}(t^{123,j}+t^{124,j}+t^{134,j}+2t^{12,j}+2t^{13,j}+2t^{14,j}+3t^{1,j})}{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{12,j}+t^{13,j}+t^{14,j}+t^{1,j})} \leq 4.$

Case B. Suppose clique E^i is maximal. By analyzing this case as *Case A* we obtain an upper bound for C^i_{max} (for the sake of simplicity we dropped subtraction in the *max* term):

$$\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{23,j}+ \max\begin{cases} t^{24,j}+t^{34,j}+t^{14,j}+t^{4,j} \\ t^{34,j}+t^{3,j} \\ t^{24,j}+t^{2,j} \\ t^{14,j}+t^{1,j} \\ 0 \end{cases}) \leq$$

$\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{23,j}+t^{34,j}+t^{24,j}+t^{14,j}+t^{1,j}+t^{2,j}+t^{3,j}+t^{4,j})$.

From comparing E^i with $A^i, \ldots, D^i$ we get
$\sum_{j=1}^{i}(t^{234,j}+t^{23,j}) \geq \sum_{j=1}^{i}(t^{14,j}+t^{1,j})$,
$\sum_{j=1}^{i}(t^{134,j}+t^{13,j}) \geq \sum_{j=1}^{i}(t^{24,j}+t^{2,j})$,
$\sum_{j=1}^{i}(t^{124,j}+t^{12,j}) \geq \sum_{j=1}^{i}(t^{34,j}+t^{,j})$,
$\sum_{j=1}^{i}(t^{123,j}+t^{12,j}+t^{13,j}+t^{23,j}) \geq \sum_{j=1}^{i}(t^{14,j}+t^{24,j}+t^{34,j}+t^{4,j})$.

Thus, we obtain

$\frac{C^i_{max}}{L^*_{max}+d_i} \leq$
$\frac{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{23,j}+t^{34,j}+t^{24,j}+t^{14,j}+t^{1,j}+t^{2,j}+t^{3,j})}{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{23,j}} \leq$
$1+\frac{\sum_{j=1}^{i}(t^{34,j}+t^{24,j}+t^{14,j}+t^{1,j}+t^{2,j}+t^{3,j})}{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{23,j})} \leq$
$1+\frac{\sum_{j=1}^{i}(t^{124,j}+2t^{13,j}+t^{134,j}+2t^{13,j}+t^{234,j}+2t^{23,j}+t^{123,j})}{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{23,j})} \leq 3$

Case C. Suppose clique I^i is maximal. Then, as for previous cases C^i_{max} can be bounded from above by

$$\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{14,j}+ \max\begin{cases} t^{24,j}+t^{34,j}+t^{4,j} \\ t^{23,j}+t^{34,j}+t^{3,j} \\ t^{23,j}+t^{24,j}+t^{2,j} \\ t^{1,j} \end{cases}) \leq$$

$\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{14,j}+t^{23,j}+t^{24,j}+t^{34,j}+t^{4,j}+t^{3,j}+t^{2,j}+t^{1,j})$

By comparing I^i with $A^i, \ldots, D^i$ we obtain
$\sum_{j=1}^{i}(t^{234,j}) \geq \sum_{j=1}^{i}(t^{1,j})$,
$\sum_{j=1}^{i}(t^{134,j} + t^{13,j} + t^{14,j}) \geq \sum_{j=1}^{i}(t^{23,j} + t^{24,j} + t^{2,j})$,
$\sum_{j=1}^{i}(t^{124,j} + t^{12,j} + t^{14,j}) \geq \sum_{j=1}^{i}(t^{23,j} + t^{34,j} + t^{3,j})$,
$\sum_{j=1}^{i}(t^{123,j} + t^{12,j} + t^{13,j}) \geq \sum_{j=1}^{i}(t^{24,j} + t^{34,j} + t^{4,j})$.
From this we get

$$\frac{C^i_{max}}{L^*_{max}+d_i} \leq \frac{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{14,j}+t^{23,j}+t^{24,j}+t^{34,j}+t^{4,j}+t^{3,j}+t^{2,j}+t^{1,j})}{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{14,j})} \leq$$

$$1 + \frac{\sum_{j=1}^{i}(t^{23,j}+t^{24,j}+t^{34,j}+t^{4,j}+t^{3,j}+t^{2,j}+t^{1,j})}{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{14,j})} \leq$$

$$1 + \frac{\sum_{j=1}^{i}(t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+2t^{12,j}+2t^{13,j}+2t^{14,j})}{\sum_{j=1}^{i}(t^{1234,j}+t^{123,j}+t^{124,j}+t^{134,j}+t^{234,j}+t^{12,j}+t^{13,j}+t^{14,j})} \leq 3.$$

We have shown that $C^i_{max} \leq 4(L^*_{max} + d_i)$. The rest of the proof is analogous to the proof of Theorem 5.7. □

Computational Experiments

We describe here results of computational experiments on the interval scheduling algorithm for problem $P4 \mid fix_j, pmtn \mid L_{max}$. The schedules generated by interval scheduling have been compared with the optimal schedules computed by the method presented in [19]. Simulation software has been written in Borland Pascal version 7 using a simulator described in [83] and run on IBM-AT 386. Parameters describing tasks have been generated pseudo-randomly with a uniform probability distribution: processing times were in range (0,10], due-dates in range [0,5], the number of required processors and their indices were generated from interval [1,4]. We tested instances from 2 till 100 tasks but due to the limitations of our LP-solver only the solutions with up to 20 tasks have been compared with the optimal solution. Fig. 5.7 through Fig. 5.9 collect the results of over 2000 experiments.

In Fig. 5.7 the average execution times of the two methods are compared. The lowest curve is the execution time of the pure interval scheduling algorithm as it has been presented in the previous sections. The middle curve is the execution time of the interval scheduling algorithm with the time needed to sort tasks in the order of nonincreasing due-dates and group them according to their types. The highest curve is the execution time of the optimization algorithm ([19]). The approximation algorithm outranks the optimization algorithm. For example, the interval scheduling algorithm schedules twenty tasks in dozens of milliseconds while the optimization algorithm requires about a minute. Thus, the difference is three orders of magnitude. In Fig. 5.8 memory requirements of the two methods are juxtaposed. The approximation algorithm requires about 3kB of memory to schedule 100 tasks while the optimization algorithm needs about 90kB to schedule 20 tasks. In Fig. 5.9 the distance of the solution generated by the interval scheduling algorithm from the optimum (i.e. L^{IS}_{max}/L^{*}_{max}) versus the number of tasks is depicted. The upper curve is the worst case observed, the lower one is the average from over 90 experiments for each point. It can be seen that the average distance is about 2-3%. The worst case distance for all observed cases is below 50%. This figure demonstrates that the worst-case expectations of Theorem 5.9 overestimate the average case. For instances with more than 8 tasks the worst observed case distance is decreasing. We also analyzed the quality of the solution generated by the interval scheduling as a function of the aggregated distance from the cases for which (5.15) holds. The "aggregated distance" is a rough measure reflecting how far the instance is from satisfying (5.15). In practice, it was the sum for all intervals of the absolute deviation from the equations and the inequalities (5.15) divided by the number of tasks. No correlation between the solution quality and the distance from (5.15) has been observed. We conclude that the interval scheduling algorithm is quite efficient.

5.3.3 Scheduling in Time Windows

The case of processors available in time windows is quite common in real situations. For example, tasks have different priorities. Urgent real-time tasks are prescheduled on processors and executed in fixed time intervals which create free time windows for lower priority tasks. Breakdowns of processors can be modeled as time windows. We will present low order polynomial time algorithms for simple cases of the problem, then an algorithm solving the problem for any fixed number of processors we will be presented. Before presenting the results let us introduce some auxiliary notation. The number of time windows is p. Time window i is an interval $[b_i, e_i]$ with a nonempty set of available processors. Two neighboring intervals differ in the set of available processors. Windows with one, two etc. available processor are called 1-windows, 2-windows etc. There are s different values of due-dates: $d_1 \leq d_2 \leq \ldots \leq d_s$, and l different values of ready times $0 = r_1 \leq r_2 \leq \ldots \leq r_l$, where $s, l \leq n$. Unlike in the previous section, $T^{D,i}$ denotes the set of tasks released at r_i, requiring set D of processors simultaneously, while $t^{D,i}$ is the sum of their

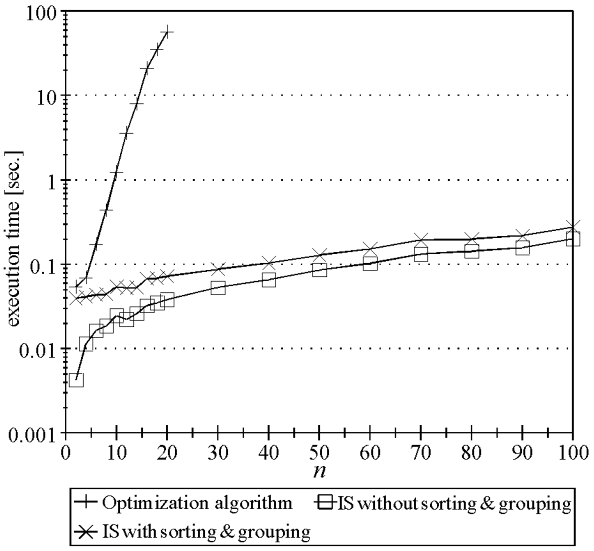

Figure 5.7: Execution time vs. n for problem $P4 \mid fix_j, pmtn \mid L_{max}$.

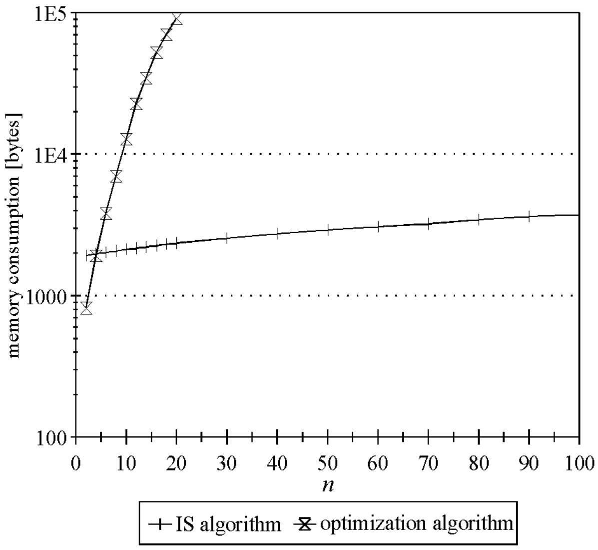

Figure 5.8: Memory consumption vs. n for problem $P4 \mid fix_j, pmtn \mid L_{max}$.

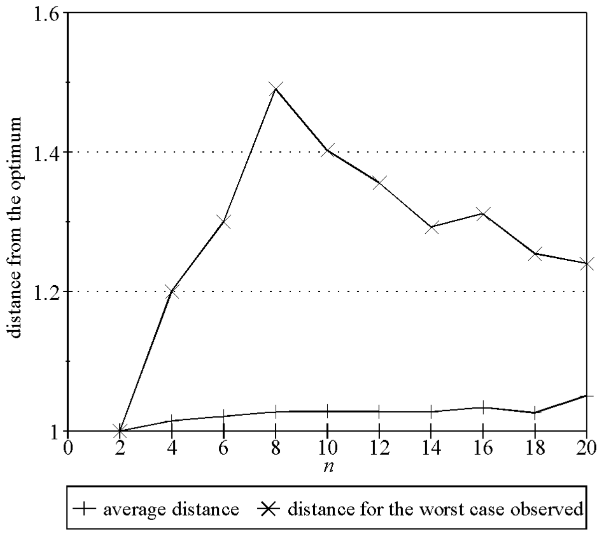

Figure 5.9: Quality of solutions generated by the IS algorithm for problem $P4 \mid fix_j, pmtn \mid L_{max}$.

processing times. Before examining the preemptive case, observe that the nonpreemptive version is s**NP**h even for one processor, (i.e. $1, win \mid\mid C_{max}$). To prove this, observe that reduction from 3-PARTITION requires only that time windows created boxes where triplets from 3-PARTITION problem must fit.

$P2, win \mid fix_j, pmtn \mid C_{max}$

The algorithm for problem $P2, win \mid fix_j, pmtn \mid C_{max}$ is as follows.

1: Shift duoprocessor tasks in 2-windows to the left as far as possible.
2: Shift to the left uniprocessor tasks in the remaining free intervals so that there is no idle time between time 0 and the completion of the last uniprocessor task.

Optimality of this algorithm follows from the following observations: duoprocessor tasks cannot be finished earlier, there is no idle time from the beginning of the schedule until the completion of uniprocessor tasks on each of the processors. The complexity of the algorithm is $O(n + p)$, where p is the number of time windows.

$P2, win \mid fix_j, pmtn, r_j \mid C_{max}$

The algorithm for problem $P2, win \mid fix_j, pmtn, r_j \mid C_{max}$ is a combination of the algorithms for problems $P2 \mid fix_j, pmtn, r_j \mid C_{max}$ and $P2, win \mid fix_j, pmtn \mid C_{max}$. We introduce the algorithm for the problem without time windows first.

Algorithm for $P2 \mid fix_j, pmtn, r_j \mid C_{max}$
On arrival of tasks:

1: Suspend processing of uniprocessor tasks, if there are any.
Schedule duoprocessor tasks first.
2: On completion of duoprocessor tasks immediately start processing of the remaining uniprocessor tasks or their parts.

Now, we examine optimality of this algorithm. When a set of tasks appears at time r_j, it can be either finished before the next ready time r_{j+1}, or it can be necessary to execute tasks from both ready times together. Only in the latter case may the tasks released at r_j influence the schedule length. Consequently, tasks released at $r_j, r_{j+1}, \ldots, r_{l-1}$ will have their contribution to C_{max} if there is no idle time on at least one of the processors in the interval $[r_j, C_{max}]$. Hence, C_{max} can be found from the formula:

$$C_{max} = \max_{1 \le i \le l} \{r_i + \sum_{j=i}^{l} t^{12,j} + \max\{\sum_{j=i}^{l} t^{1,j}, \sum_{j=i}^{p} t^{2,j}\}\}$$

On the other hand, there can be no shorter schedule because the above equation represents processing requirements of tasks released at certain time moments to be processed on a given processor. Hence, the schedule is optimal. The algorithm can be implemented to run in $O(n)$ time.

Now, we return to problem $P2, win \mid fix_j, pmtn, r_j \mid C_{max}$. Adaptation of the previous algorithm for this case consists in scheduling uniprocessor tasks in 1-windows whenever ready uniprocessor tasks exist, while giving preference to duoprocessor tasks in 2-windows. Thus, duoprocessor tasks are scheduled as soon as they appear and cannot be finished earlier. Uniprocessor tasks are shifted to the left, so that the idle times are avoided and ready times are observed. Uniprocessor tasks can be finished earlier only by delaying some duoprocessor task(s). This, however, does not reduce the length of the schedule. We conclude that no shorter schedule can exist. Observe that to apply this algorithm the only required information is which (ready) task requires what set of processors. The complexity of the algorithm is $O(n+p)$.

$P3, win \mid fix_j, pmtn \mid C_{max}$

The problem with three processors requires more careful treatment because different types of duoprocessor tasks cannot be executed in parallel. The order of executing duoprocessor tasks in 3-windows is important because appropriate 2-window for some duoprocessor task can be found somewhere later in the schedule. Hence, 3-window space should be preserved for the duoprocessor tasks that have no appropriate 2-windows.

Now, we describe the rationale behind the algorithm presented below. Without loss of generality we assume that for each set D of required processors there is only one multiprocessor task T^D with processing time t^D. Shifting tripleprocessor task to the left in 3-windows produces the shortest possible schedule for this task. In the remaining schedule there are 2-windows which comprise the following sets of processors: $\{P_1, P_2\}$, $\{P_1, P_3\}$, or $\{P_2, P_3\}$. 3-windows are available for all types of duoprocessor tasks. This creates a *processing capacity profile* consisting of the amount of processing time available for each of duoprocessor task types separately and the processing time available in 3-windows for all types of duoprocessor tasks together. For interval $[0, t]$ there are 2-windows with processing capacity $pc_{12}(t), pc_{13}(t), pc_{23}(t)$ on processors $\{P_1,P_2\}$, $\{P_1, P_3\}$, $\{P_2, P_3\}$, respectively, and 3-windows with capacity $pc_{123}(t)$. The shortest schedule for duoprocessor tasks is defined by the minimal time at which the processing capacity profile accommodates requirements of the tasks:

$$C_{max}^{\text{duoproc.tasks}} = \min\{t : \min\{0, t^{12} - pc_{12}(t)\} + \min\{0, t^{13} - pc_{13}(t)\} + \min\{0, t^{23} - pc_{23}(t)\} \leq pc_{123}(t)\}$$

Thus, while scheduling duoprocessor tasks, 2-windows can be immediately allocated to appropriate duoprocessor tasks, because no other type of duoprocessor task can use it. Allocation in 3-windows should be postponed until the final allotment of duoprocessor tasks to 2-windows is known. Uniprocessor tasks should be shifted to the left so that there is no idle time before the end of the last uniprocessor task on the given processor. Hence, the algorithm consists in three steps of scheduling tripleprocessor, duoprocessor, and finally, uniprocessor tasks. For simplicity of the presentation we assume that all tasks fit in p time windows.

An Algorithm for $P3, win \mid fix_j, pmtn \mid C_{max}$

1: Schedule tripleprocessor task shifted to the left in 3-windows; remove allocated 3-windows from data structures holding free time windows;
$j := 1$; $pc_{123} := 0$;

2: **while** $(t^{12}\geq 0)$ **or** $(t^{13}\geq 0)$ **or** $(t^{23}\geq 0)$ **do**
begin
2.1: **if** j is 2-window comprising processors from set D **and** $t^D > 0$ **then**
begin
2.1.1: $tmp := \min\{t^D, e_j - b_j\}$;
2.1.2: **if** $t^{12} + t^{13} + t^{23} - tmp \leq pc_{123}$ **then** (* enough pc found *)
begin
2.1.2.1: $tmp := t^D - (t^{12} + t^{13} + t^{23} - pc_{123})$;
2.1.2.2: schedule tmp units of T^D task in interval $[b_j, b_j + tmp]$;
2.1.2.3: $t^D := t^D - tmp$;
2.1.2.4: schedule remaining duoprocessor tasks or their parts
in the previously memorized 3-windows;
update data structures holding free time windows;
2.1.2.5: $t^{12} := t^{13} := t^{23} := 0$;
end
else (* not enough pc to schedule all duoprocessor tasks *)
begin
2.1.2.6: schedule tmp units of T^D task in interval $[b_j, b_j + tmp]$;
update data structures holding free time windows;
2.1.2.7: $t^D := t^D - tmp$;
end;
end;
2.2: **if** window j is 3-window **then**
begin
2.2.1: **if** $t^{12} + t^{13} + t^{23} \leq pc_{123} + (e_j - b_j)$ **then** (* enough pc found *)
begin
2.2.1.1: schedule remaining duoprocessor tasks or their parts in 3-window j
and previously memorized 3-windows finishing at $b_j + t^{12} + t^{13} + t^{23} - pc_{123}$;
update data structures holding free time windows;
2.2.1.2: $t^{12} := t^{13} := t^{23} := 0$;
end
else (*still not enough pc has been found*)
begin
2.2.1.3: $pc_{123} := pc_{123} + (e_j - b_j)$; memorize 3-window j for future use;
end;
end;
2.3: $j := j + 1$; (* analyze the next window *)
end;
3: Schedule uniprocessor tasks in the remaining time windows such that

there is no idle time before the completion time of the last uniprocessor task on each of the processors;

end.

High level description. Tripleprocessor task is scheduled in line 1, duoprocessor tasks in lines 2-2.3, and uniprocessor tasks in line 3. While scheduling duoprocessor tasks, windows are analyzed one by one. 2-windows are considered in lines 2.1-2.1.2.7, 3-windows in lines 2.2-2.2.1.3. Final schedule for duoprocessor tasks is built in lines 2.1.2.1-2.1.2.5 when the last used window has 2 processors and in lines 2.2.1-2.2.1.2 if it is 3-window. When the final schedule cannot be built finishing in the current window, a piece of duoprocessor task is scheduled in 2-window (lines 2.1.2.6-2.1.2.7), or in the case of 3-window information about it is stored for future use (2.2.1.3).

Optimality of the above algorithm is a result of the following facts: Tripleprocessor task cannot be finished earlier. A duoprocessor task can be finished earlier only by using time interval of some other duoprocessor task or of some tripleprocessor task. This is not reducing the length of a schedule (but can increase). The same applies to uniprocessor tasks. The complexity of the algorithm is $O(n+p)$.

It is hard to extend this approach to solve the cases with greater number of processors. For instance, in $P4, win \mid fix_j, pmtn \mid C_{max}$, duoprocessor task T^{12}, beside appropriate 2-windows, can be scheduled in 3-windows comprising processors $\{P_1, P_2, P_3\}, \{P_1, P_2, P_4\}$, and in 4-windows. The decision where the tasks from T^{12} are scheduled influences the completion time of the other duoprocessor task types. Hence, a stronger tool seems to be necessary to solve such problems.

Observe that for integer values of processing times and time window intervals all preemptions take place at integer values of time. Thus, the above algorithms can be applied also for unit execution time tasks. We conclude:

Corollary 5.3 *Problems $P2, win \mid fix_j, p_j = 1 \mid C_{max}$, $P3, win \mid fix_j, p_j = 1 \mid C_{max}$ can be solved in $O(n+p)$ time.*

$Pm, win \mid fix_j, pmtn, r_j \mid C_{max}$

When the number of processors is fixed the problem can be solved in polynomial time using linear programming, processor feasible sets and binary search. Observe that since tasks are released at different moments and since C_{max} can be smaller than the beginning time of some time window, the processor feasible sets change with time and C_{max}. Only tasks which are already released can be included in processor feasible sets. Assume that it is possible to schedule feasibly all tasks in u time windows. Then we will apply a binary search to find the smallest possible u. We denote by:

M_i the number of processor feasible sets in time window i,

x_{ij} for $i = 1, \ldots, M_j, j = 1, \ldots, u$, a variable denoting processing time of the i-th feasible set in window j,

A^i_j the set of processor feasible sets indices in window i which include task T_j,

S_i the set of tasks released in window i,

f_{iq} the index of the task released in window i as q-th task,

U_{iq} the set of all processor feasible set indices in window i which include tasks released in window i as q-th, $q+1$-th, $\ldots, \mid S_j \mid$-th task, i.e. $U_{iq} = \cup_{j=q}^{|S_i|} A^i_{f_{ij}}$.

For the considered u linear program $LP_1(u)$ is as follows:

minimize C_{max}

subject to

$$\sum_{i=1}^{M_j} x_{ij} \leq e_j - b_j \quad \text{for } j = 1, \ldots, u-1 \tag{5.17}$$

$$\sum_{i=1}^{M_u} x_{iu} \leq C_{max} - b_u \tag{5.18}$$

$$\sum_{h \in U_{jq}} x_{hj} \leq e_j - r_{f_{jq}} \quad \text{for } q = 1, \ldots, \mid S_j \mid, j = 1, \ldots, u-1 \tag{5.19}$$

$$\sum_{h \in U_{uq}} x_{hu} \leq C_{max} - r_{f_{uq}} \quad \text{for } q = 1, \ldots, \mid S_u \mid \tag{5.20}$$

$$\sum_{i=1}^{u} \sum_{h \in A^i_j} x_{hi} \geq t_j \quad \text{for } j = 1, \ldots, n \tag{5.21}$$

$$b_u \leq C_{max} \leq e_u \tag{5.22}$$

$$x_{ij} \geq 0 \quad \text{for } i = 1, \ldots, M_j \quad j = 1, \ldots, u$$

Inequalities (5.17),(5.18) guarantee that processor feasible sets are not executed beyond the end of their windows. Inequalities (5.19),(5.20) guarantee that tasks released during a time window will not be executed longer than the interval from their ready time to the end of the window or till

C_{max}, respectively. Inequalities (5.21) guarantee that tasks are fully executed and (5.22) guarantee that u is the last window used. The above linear program has $O(pn^m)$ variables. The number of inequalities (5.19) is equal to the number or ready times. Hence, there are $O(n+p)$ constraints in $LP_1(u)$. It can be formulated and solved in polynomial time, provided the number of processors is fixed. When $LP_1(u)$ has a feasible solution it can be verified whether for smaller number of windows a feasible solution exists. On the other hand, when a feasible solution does not exist one may try with bigger number of windows. Thus, using binary search over u the optimal solution can be found by solving $O(\log_2 p)$ linear programs.

$Pm, win \mid set_j, pmtn, r_j \mid C_{max}$

When tasks have alternative modes of execution (set_j model), then while generating processor feasible sets we have to analyze a wider set of possibilities. Furthermore, to guarantee complete execution of the tasks we have to sum the percentages of processing times on alternative sets of processors. The number of processor feasible sets remains polynomially bounded. Assume that each task can be executed on K alternative sets of processors then there are no more than $O((nK)^m)$ processor feasible sets. Such a number could be achieved only if the tasks were executed on two (or more) alternative sets of processors in the same processor feasible set (i.e. the same task would be executed simultaneously on several non-intersecting sets of processors) which is forbidden. Since the number of processors m is fixed, K is $O(2^m)$ and the number of processor feasible sets is polynomially bounded by $O(p(n2^m)^m)$. A linear program for problem $Pm, win \mid set_j, pmtn, r_j \mid C_{max}$ differs from the formulation (5.17)-(5.22) only in the set of inequalities (5.21) which should be replaced by

$$\sum_{i=1}^{u} \sum_{D \in set_j} \sum_{h \in A^i_{D,j}} \frac{x_{hi}}{t_j^D} \geq 1 \quad \text{for} \quad j = 1, \ldots, n \tag{5.23}$$

where $A^i_{D,j}$ is the set of the processor feasible set indices including task T_j executed on processors from set D in interval i. Then, the problem can be solved in polynomial time analogously to the method of solving $Pm, win \mid fix_j, pmtn, r_j \mid C_{max}$.

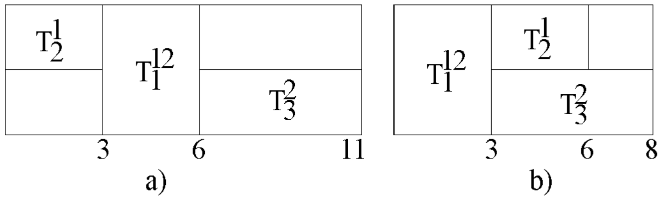

Figure 5.10: Nonexistence of an on-line optimization algorithm for $P2 \mid fix_j, pmtn, r_j \mid L_{max}$.

$Pm, win \mid fix_j, pmtn, r_j \mid L_{max}$

Assuming that "on-line" scheduling is based only on the information about tasks that already arrived we will show that there is no on-line optimization algorithm for this problem. Consider an example.

Example
$fix_1 = \{P_1, P_2\}, fix_2 = \{P_1\}, t_1^{12} = t_2^1 = 3, d_1 = 4, d_2 = 2, r_1 = r_2 = 0$. Since T_1 and T_2 cannot be executed in parallel, they must be executed sequentially. Assume task T_2 is started first and after this task T_1 is executed. Then at moment $r_3 = 3$ arrives task T_3 which has $t_3 = 5, d_3 = 6, fix_3 = \{P_2\}$. The best schedule that can be achieved at that moment is presented in Fig. 5.10a and has $L_{max} = 5$. If T_1 were scheduled first the best schedule could have $L_{max} = 4$ (Fig. 5.10b). Assume an opposite scenario in which T_1 is scheduled first, then at $r_3 = 3$ task T_3 arrives which has $t_3 = 5, d_3 = 9, fix_3 = \{P_2\}$ and the best possible schedule has $L_{max} = 4$ (Fig. 5.10b). If T_2 were scheduled first the schedule could have $L_{max} = 2$ (Fig. 5.10a).

We conclude that whatever the sequence of executing the ready tasks is, a scenario is possible which results in not optimal schedule. Thus, there is no on-line optimization algorithm for problem $P2 \mid fix_j, pmtn, r_j \mid L_{max}$. This applies also in the nonpreemptive case. Next, let us note that EDD (Earliest Due - Date first) rule is not optimal (off-line) for this problem as shown in the following example.

Example
$n = 2, fix_1 = \{P_1, P_2\}, t_1^{12} = 3, d_1 = 4, fix_2 = \{P_1\}, t_2^1 = 3, d_2 = 3, p = 1, b_1 = 3, e_1 = 6$ and it is 1-window with $\{P_1\}$ free. The EDD schedule is presented in Fig. 5.11a. It has $L_{max} = 5$. The optimal schedule in Fig. 5.11b has $L_{max} = 3$.

We conclude that the optimization algorithm requires a global look at the schedule. When the number of processors is fixed, the problem can be solved using linear programming. The algorithm is similar to the one proposed for

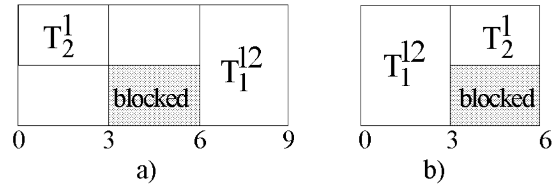

Figure 5.11: Nonoptimality of the EDD rule for $P2 \mid fix_j, pmtn \mid L_{max}$.

problem $Pm, win \mid fix_j, pmtn, r_j \mid C_{max}$. When building processor feasible sets for window i one can use only tasks which are present, i.e. set $\{T_j : r_j < e_i, d_j + L_{max} > b_i\}$. Thus, the processor feasible sets change when value L_{max} passes a point where for some tasks T_{j_1}, T_{j_2} and window i: $d_{j_1} + L_{max} = b_i$ or $r_{j_2} = d_{j_1} + L_{max}$. Hence, there are $O(np+n^2)$ intervals of L_{max} values where the processor feasible sets remain unchanged. Consider minimization of L_{max} in u-th such interval, i.e. in the range $[L_u^b, L_u^e]$ of L_{max} values. In such an interval the sequence of all the events in the system (ready times, due-dates increased by L_{max}, windows beginnings and windows ends) is constant. Let us order all such events according to their appearance. We denote:

$q = 2p + 2n$ - the number of events,

ε_i for $i = 1, \ldots, q$ - the time instant at which event i takes place for releases of tasks, window beginnings and window ends; for a due-date related event ε_i is appropriate due-date (i.e. $\varepsilon_i = d_j$ when event i is related to $d_j + L_{max}$ for some T_j and the current value of L_{max}); $\varepsilon_1 = 0$,

f_i for $i = 1, \ldots, q$ - the function returning 1 if event i is related to a due-date, and returning 0 otherwise,

M_i for $i = 1, \ldots, q-1$ - the number of processor feasible sets between events i and $i+1$,

x_{ij} for $i = 1, \ldots, M_j, j = 1, \ldots, q-1$ - the time of executing tasks in i-th processor feasible set in interval between events j and $j+1$,

A_j^i for $i = 1, \ldots, q-1, j = 1, \ldots, n$ - a set of processor feasible sets indices between event i and $i+1$ which include task T_j.

For the considered interval $[L_u^b, L_u^e]$ the linear program $LP_2(u)$ is as follows:

min L_{max}
subject to

$$\sum_{i=1}^{M_j} x_{ij} \quad \leq \varepsilon_{j+1} - \varepsilon_j + L_{max}(f_{j+1} - f_j) \quad \text{for } j = 1, \ldots, q-1 \tag{5.24}$$

$$\sum_{i=1}^{q-1} \sum_{k \in A_i^j} x_{ki} \geq t_j \qquad \text{for } j = 1, \ldots, n \tag{5.25}$$

$$L_u^b \leq L_{max} \leq L_u^e \tag{5.26}$$

$$x_{ij} \geq 0 \qquad \text{for } i=1, \ldots, M_j, j=1, \ldots, q-1$$

Inequalities (5.24) guarantee that processor feasible sets are executed between appropriate events. Inequalities (5.25) guarantee that tasks are fully executed. Finally, (5.26) guarantees that the order of events is unchanged and processor feasible sets remain valid. There are $O(p+n)$ constraints and $O((p+n)n^m)$ variables. Hence, the above formulation can be solved in polynomial time for fixed m [150]. When a feasible solution exists then a range of smaller L_{max} values can be considered. And vice versa, when no feasible solution exists a rage of greater L_{max} values can be analyzed. Hence, the optimal value L^*_{max} can be found by binary search in $O(\log(np+n^2)LP))$ time, where LP is the complexity of formulating and solving $LP_2(u)$.

$Pk, win \mid set_j, pmtn, r_j \mid L_{max}$

The method from the previous subsection can be extended to set_j model where tasks can be processed by alternative sets of processors. The differences come from a wider range of possible task combinations in feasible sets and from the fact that processing time must be accumulated over alternative sets of processors executing a task. Since the number of processor feasible sets is limited from above by $O((n2^m)^m)$, the number of variables in the linear program is bounded polynomially from above by $O((p+n)(n2^m)^m)$. The linear program differs from $LP_2(u)$ in inequality (5.25) which should be replaced by

$$\sum_{i=1}^{q-1} \sum_{D \in set_j} \sum_{h \in A^i_{D,j}} \frac{x_{hi}}{t_j^D} \geq 1 \text{ for } j = 1, \ldots, n \tag{5.27}$$

where $A^i_{D,j}$ is the set of the processor feasible set indices including task T_j executed on processors from set D in interval i. The method of finding L^*_{max} and computational complexity can be derived analogously.

In Table 5.2 results on scheduling multiprocessor tasks in dedicated processors environment are collected. The following abbreviations denote: B&B - branch and bound algorithm, s.g. - scheduling graph, LP - linear programming, ILP - integer linear programming, pseudopoly. - pseudopolynomial algorithm.

Table 5.2: Scheduling multiprocessor tasks on dedicated processors

Problem	Result	Reference
	Nonpreemptive scheduling	
$P \mid fix_j \mid C_{max}$	B&B	[48]
$P \mid fix_j \mid C_{max}$ and $\mid fix_j \mid = 2$	$\mathbf{NP}$h, $S_{LPT} = \frac{4(d-1)}{d}$	
	$S_{LPT} \leq 3$ when $d \leq 5$	
	$S_{LPT} < 2$ binomial s.g.	[130]
$P \mid fix_j \mid C_{max}$ and $\mid fix_j \mid = 2$	20 cases $\mathbf{NP}$h	
	23 cases polynomial	
	$\frac{4}{3} < S_{LS} \leq 3$	
	$S_{LS} \leq 2$ for $p \leq 2$	
	$S_{LPT} = \frac{5}{2} - \frac{1}{p}$	
	$C_{max}^{DP1} \leq 3C_{max}^{*} + e\varepsilon$	[70]
$P \mid fix_j \mid C_{max}$ and $\mid fix_j \mid \in \{1, 2\}$	9 cases $\mathbf{NP}$h	
	9 polynomial cases	[134]
$P \mid fix_j \mid C_{max}$ and		
$P \mid fix_j, p_j = 1 \mid C_{max}$	experimental study	[74]
$P2 \mid fix_j, p_j = 1 \mid \sum w_j c_j$	$O(n \log n)$	[79]
$P \mid fix_j \mid \sum w_j c_j$	ILP+experiment	[79]
$P3 \mid fix_j \mid C_{max}$	s$\mathbf{NP}$h,	Th.5.4,
		[31, 120]
	$S_{NS} < \frac{4}{3}$	[31]
	$S_{NS} = \frac{5}{4}$	[78]
	$S_{LPT} = S_{SPT} = 3$	[78]
$P \mid fix_j, p_j = 1 \mid C_{max}$	special cases, bounds	[207]
$O \mid fix_{ij} \mid C_{max}$ and $stages = 2$	$O(n)$	[50]
$O \mid fix_{ij} \mid C_{max}$ and $stages = 3$	$\mathbf{NP}$h	[50]
$O \mid fix_{ij}, p_{ij} = 1 \mid C_{max}$ and $stages = r$	polynomial	[50]
$F \mid fix_{ij} \mid C_{max}$ and $stages = 2$	$O(n \log n)$	[50]
$F2 \mid fix_{ij} \mid C_{max}$ and $stages = 3$	s$\mathbf{NP}$h	[50]
$J2 \mid fix_{ij}, p_{ij} = 1 \mid C_{max}$	s$\mathbf{NP}$h	[50]
$J2 \mid fix_{ij} \mid C_{max}$ and $n_j \leq 2$	$O(n \log n)$	[50]
$J \mid fix_{ij} \mid C_{max}$ and $n = 2$	$O(n^2 \log n)$	[50]
$J2 \mid fix_{ij} \mid C_{max}$ and $n = k$	$O(n^{3k})$	[50]
$Pm \mid fix_j, p_j = 1 \mid f$		
and $f \in \{\sum w_j U_j, \sum \tau_j, \sum w_j c_j\}$	$O(R2^R n^{R+1} +$	
and R number of task types	$+2^R(R + m))$	[49]
$Pm \mid fix_j, p_j = 1, r_j \mid f$		
and $f \in \{C_{max}, \sum c_j\}$	$O(R2^R n^{R+1} +$	
and R number of task types	$2^R(R + m))$	[49]
$J \mid fix_{ij}, p_{ij} = 1, prec, r_j \mid f$		
and $n = k, f \in \{max f_j, \sum f_j\}$		
and f_j nondecreasing function of c_j	$O(k2^k m \sum_{j=1}^{k} n_j \prod_{j=1}^{k} n_j)$	[49]

Table 5.2 continued

Problem	Result	Reference
$F \mid fix_{ij}, p_{ij} = 1 \mid f$ and $stages = r$ and $f \in \{\sum w_j c_j, \sum \tau_j, \sum w_j U_j\}$	$O(r^2 2^r n^{r+2} + 2^r(r+m))$	[49]
$F \mid fix_{ij}, p_{ij} = 1, r_j \mid f$ and $stages = r, f \in \{\sum c_j, C_{max}\}$	$O(r^2 2^r n^{r+2} + 2^r(r+m))$	[49]
$O \mid fix_{ij}, p_{ij} = 1 \mid f$ and $stages = r$ and $f \in \{\sum w_j c_j, \sum \tau_j, \sum w_j U_j\}$	$O(r^3(r!)^2 2^r n^{r!(r+1)+1} +$ $+2^r(r+m))$	[49]
$O \mid fix_{ij}, p_{ij} = 1, r_j \mid f$ and $stages = r$ and $f \in \{C_{max}, \sum c_j\}$	$O(r^3(r!)^2 2^r n^{r!(r+1)+1} +$ $+2^r(r+m))$	[49]
$O \mid fix_{ij}, p_{ij}, prec \mid f$ and $n = 2, stages = r$ and $f \in \{max f_j, \sum f_j\}$ and f_j nondecreasing function of c_j	$O(r^{2.5})$	[49]
$P2,3,4 \mid fix_j, p_j = 1 \mid C_{max}$	$O(n)$	[25]
$P5 \mid fix_j, p_j = 1 \mid C_{max}$	$O(n^{2.5})$	[25]
$P \mid fix_j \mid C_{max}$	B&B	[25]
$Pm \mid fix_j, p_j = 1 \mid C_{max}$	ILP	[120]
$P \mid fix_j, p_j = 1 \mid C_{max}$	s**NP**h	[120]
$P2 \mid fix_j, p_j = 1, chain \mid C_{max}$	s**NP**h	[120]
$P2 \mid fix_j, p_j = 1, r_j \mid C_{max}$	s**NP**h	[120]
$Pm \mid fix_j, p_j = 1, r_j \mid C_{max}$	ILP	[120]
$P2 \mid fix_j \mid \sum c_j$	**NP**h	[120]
$P3 \mid fix_j \mid \sum c_j$	s**NP**h	[120]
$P2 \mid fix_j \mid \sum w_j c_j$	s**NP**h	[120]
$P \mid fix_j, p_j = 1 \mid \sum c_j$	s**NP**h	[120]
$P2 \mid fix_j, p_j = 1, chain \mid \sum c_j$	s**NP**h	[120]
$P, win \mid fix_j \mid C_{max}$	**NP**h, 3 polynomial cases	[136]
$P2 \mid fix_j, p_j = 1 \mid L_{max}$	$O(n)$	[22]
$P3 \mid fix_j \mid C_{max}$	$S_{18} = \frac{7}{6}$	[102]
$P2 \mid fix_j \mid L_{max}$	s**NP**h	[85]
$P3 \mid fix_j \mid C_{max}$ and $\mid fix_j \mid = 2$	$O(n)$	[138]
$P3 \mid fix_j, chain \mid C_{max}$ and $\mid fix_j \mid = 2$	$O(n)$	[138]
$P4 \mid fix_j, p_j = 1, chain \mid f$ and $\mid fix_j \mid = 2$ and $f \in \{C_{max}, \sum c_j\}$	s**NP**h	[138]
$P \mid fix_j, p_j = 1 \mid \sum c_j$ and $\mid fix_j \mid = 2$	s**NP**h	[138]
$P3 \mid fix_j, chain \mid \sum c_j$ and $\mid fix_j \mid = 2$	$O(n \log n)$	[138]
$P4 \mid fix_j \mid \sum c_j$ and $\mid fix_j \mid = 2$	**NP**h	[138]
$P \mid fix_j \mid \sum c_j$ and $\mid fix_j \mid = 2$ and s.g. is 2-star	**NP**h	[138]

Table 5.2 continued

Problem	Result	Reference
$P2 \mid set_j \mid C_{max}$	pseudopoly.	[23]
$P3 \mid set_j \mid C_{max}$		
and $\forall_{T_j} fix_j \neq \{P_1, P_3\}$	pseudopoly.	[23]
$P \mid set_j \mid C_{max}$	$S_{SPTM} = m$	[23]
$P \mid set_j \mid C_{max}$	s**NP**h, heuristic	[26]
$P2 \mid fix_j \mid \sum c_j$	s**NP**h,$S_H < 2$	[51]
$P \mid fix_j, prec \mid C_{max}$	special cases	[77]
$P2, win \mid fix_j, p_j = 1 \mid C_{max}$		
and p number of time windows	$O(n+p)$	Coro.5.3
$P3, win \mid fix_j, p_j = 1 \mid C_{max}$	$O(n+p)$	Coro.5.3
Preemptive scheduling		
$P \mid fix_j, pmtn \mid C_{max}$		
and $\mid fix_j \mid = 2$	s**NP**h	[135]
$Pm \mid fix_j, pmtn \mid C_{max}$		
and $\mid fix_j \mid = 2$	LP	[135]
$P2 \mid fix_j, pmtn \mid C_{max}$	$O(n)$	[21]
$P3 \mid fix_j, pmtn \mid C_{max}$	$O(n)$	[21]
$P4 \mid fix_j, pmtn \mid C_{max}$	$O(n)$	[21]
$P4 \mid fix_j, pmtn, res1 \cdot 1 \mid C_{max}$	$O(n)$	[21]
$P2 \mid fix_j, pmtn \mid L_{max}$	$O(n)$	Th.5.5
$P3 \mid fix_j, pmtn \mid L_{max}$	interval scheduling	Th.5.6,5.7
$P4 \mid fix_j, pmtn \mid L_{max}$	interval scheduling	Th.5.8,5.9
$P \mid fix_j, pmtn \mid C_{max}$ and $\mid fix_j \mid = 2$		
s.g. bipartite, unicyclic	$O(n^2)$	[138]
$P \mid fix_j, pmtn \mid C_{max}$ and $\mid fix_j \mid = 2$		
s.g. candy,caterpillar	$O(n)$	[138]
$P4 \mid fix_j, pmtn, chain \mid f$		
and $\mid fix_j \mid = 2$ and $f \in \{C_{max}, \sum c_j\}$	s**NP**h	[138]
$P \mid fix_j, pmtn \mid \sum c_j$ and $\mid fix_j \mid = 2$	s**NP**h	[138]
$P3 \mid fix_j, pmtn, chain \mid \sum c_j$		
and $\mid fix_j \mid = 2$	$O(n \log n)$	[138]
$P \mid fix_j, pmtn \mid C_{max}$		
and $\mid fix_j \mid = 2$ and s.g. 2-star,superstar	$O(n \log n)$	[138]
$P2 \mid fix_j, pmtn \mid \sum c_j$	$O(n \log n)$	[51]
$Pm \mid set_j, var \mid C_{max}$	LP	[23]
$Pm \mid set_j, var, r_j \mid L_{max}$	LP	[19]
$P2, win \mid fix_j, pmtn \mid C_{max}$	$O(n+p)$	Sec.5.3.3
$P2, win \mid fix_j, pmtn, r_j \mid C_{max}$	$O(n+p)$	Sec.5.3.3
$P3, win \mid fix_j, pmtn \mid C_{max}$	$O(n+p)$	Sec.5.3.3
$Pm, win \mid set_j, var, r_j \mid C_{max}$	LP	Sec.5.3.3
$Pm, win \mid set_j, var, r_j \mid L_{max}$	LP	Sec.5.3.3

Chapter 6

Divisible Tasks

6.1 Introduction

In this chapter a new scheduling model applicable in a wide range of parallel architectures and parallel applications is presented. We consider scheduling divisible tasks, i.e. tasks that can be divided into parts of arbitrary size. Furthermore, the parts can be processed in parallel independently of each other. In other words, the parallel application includes no precedence constraints (data dependencies) and granularity of parallelism is fine. Before proceeding to the presentation of the divisible task method, we introduce, in a more informal way, basic founding concepts.

Many contemporary parallel applications are divisible tasks. Consider, for example, searching for a record in a huge database (thousands or more records). This can be done by cooperating processors. The database file can be divided into parts with one record granularity. The search can be conducted in each part independently of the other parts. Finally, the results are reported to some master processor. The same method can be applied to searching for a pattern in a text, graphical, audio, etc. file. Similar situation takes place when sorting a database file in a distributed way. Yet, this case is a bit more complex because the sorted file parts must be merged. Analogously, big measurement data files can be divided into parts processed independently in parallel [60]. Further examples of divisible tasks are relevant to data parallelism: simulations of molecular dynamics [3], some problems of linear algebra with the use of big matrices [27], solving partial differential equations by finite element method [209] and many other engineering and scientific problems [53]. Note that similar assumptions on divisibility of the load were made in loop scheduling and load balancing (cf. Sections 4.2,4.4).

Now, we will outline the process of data dissemination and processing. A parallel computer consists of m *processing elements* (PEs), each of which comprises a processor, local memory, and is capable of communicating in the interconnection network (either by independent network processor, or by use of software run on the processor). For simplicity reasons names of processor and processing element are equivalent here. Only when a PE has a network processor is it capable of simultaneous computing and communicating. Initially, the whole volume V of data to be processed resides in one processor called *originator*. The originator intercepts for local processing α_1 data units and sends the rest (i.e. $V - \alpha_1$) to its idle neighbors. Each processing element intercepts for local processing some data from the received volume and sends the rest to the idle neighbors. Thus, PE number i (denoted P_i) intercepts and processes locally α_i data units while sending the rest of the obtained data to its still idle neighbors. P_i will process its share α_i in $\alpha_i A_i$ units of time. Following Section 2.1 the transmission time of x data units over link i joining two processors is $S_i + xC_i$. Our goal is to find such a distribution of task parts (or problem data) that the communications and computations are finished in the shortest possible time. The above description still leaves space for details including, e.g. a communication algorithm tailored to the interconnection. Observe that when no results are returned to the originator, all the processors must stop working at the same moment of time. This observation can be explained intuitively: when P_i finishes earlier then it is possible to off-load other PEs by moving part of the load to P_i. In this way the whole length of a schedule would be reduced. This observation has been proved both for particular interconnections [60, 188] and for a general type of interconnection [33]. The model can be applied also in the case when some results are returned. However, the former case simplifies the presentation. In majority of works on divisible tasks only one application is assumed to be present in the computer system (i.e. $n = 1$). Unless otherwise stated we assume in this chapter that the number of tasks is equal to one. Since the actual processing time of a task depends on speeds of communication channels, speeds of PEs, and distribution of the load it is hard to use a single value of processing time as in the previous sections. Hence, in this chapter we will use volume V of data to be processed as a more natural measure of work to be performed.

The organization of this chapter is as follows. In Section 6.2 we give an overview of existing subject literature. In Section 6.3 we present results of applying the idea of divisible task to scheduling and performance evaluation of distributed systems. Section 6.3 is divided according to the analyzed interconnection architectures. Section 6.4 contains final remarks and conclusions.

6.2 Overview of Earlier Results

To our knowledge, the first work analyzing divisible tasks was [60]. Considering divisible tasks was motivated by the problem of finding the optimal balance between parallelism and necessary communication in a network of intelligent sensors. Linear network of PEs with or without network processor for store-and-forward commutation mode was examined. Thus, the investigated problems can be denoted $Q, chain, s\&f, no-overlap \mid n=1, div \mid C_{max}$ or $Q, chain, s\&f \mid n=1, div \mid C_{max}$. The communication time was assumed to be a linear function of the transferred volume. The startup time was negligible ($S=0$). A solution based on reduction to a set of linear equations can be applied in time proportional to the number of PEs. The same problem was analyzed independently in [154]. Closed-form expressions were presented to find a distribution of the load. It was also shown that in a homogeneous network with the originator located in the network interior, the whole load processing time is the same when the originator sends data to the left first or to the right first.

In [61] scheduling divisible tasks on a tree network of processors is considered. The analyzed problems can be denoted $Q, tree, s\&f, no-overlap \mid div, n=1 \mid C_{max}$ or $Q, tree, s\&f \mid div, n=1 \mid C_{max}$. The sequence of communications is assumed to be known a priori. There is no communication startup time. For such assumptions the problem can be solved by a set of linear equations.

Scheduling a divisible task on a bus interconnected system has been analyzed in [14]. Again, it was assumed that the sequence of communications is known and startup time is negligible. Two cases were distinguished: a system with a master processor which is not computing but is in charge of collecting measurements and handling data communications, and a system without the master processor. PEs had no network processors. The tackled problems can be denoted $Q, bus, no-overlap \mid div, n=1 \mid C_{max}$. The case of PEs with the network processor (i.e. $Q, bus \mid div, n=1 \mid C_{max}$) was analyzed in [13].

In [15] problems $Q, chain, s\&f, no-overlap \mid div, n=1 \mid C_{max}$; $Q, chain, s\&f \mid div, n=1 \mid C_{max}$; $Q, tree, s\&f, no-overlap \mid div, n=1 \mid C_{max}$; $Q, tree, s\&f \mid div, n=1 \mid C_{max}$ are considered. By the use of the concept of an equivalent processor which is a single-processor equivalent of the original multiprocessor system, the ultimate performance limits are calculated. This analysis is further extended in [176]. Also in [100] performance limits of linear networks and star networks are examined (problems $Q, chain, s\&f, no-overlap \mid div, n = 1 \mid C_{max}$; $Q, chain, s\&f \mid div, n = 1 \mid C_{max}$; $Q, star, s\&f, no-overlap \mid div, n = 1 \mid C_{max}$; $Q, star, s\&f \mid div, n = 1 \mid C_{max}$). Closed-form formulae expressing the limit of the performance enhancement obtained by using additional processors are presented. The communication appeared to have a similar effect on speedup as the sequential part of parallel application in the Amdahl's law.

In [16] optimal sequencing of communications in a star network is considered. It is shown that for the case with the network processors the optimal sequence of distributing data is the order of decreasing communication speed, speeds of processors are irrelevant. Thus, when the communication links are identical, the ordering of communications to processors is immaterial. These counterintuitive results are satisfied for $S = 0$.

Closed-form expressions for the optimal load distribution in a bus and tree networks are given in [12]. The performance of symmetric tree networks is analyzed by collapsing the component processors and links into one equivalent processor.

Work [189] analyzes scheduling more than one divisible application in the computer system. The PEs were either equipped with network processors or not so equipped and interconnected by a bus ($R, bus, no-overlap \mid div \mid C_{max}$ and $R, bus \mid div \mid C_{max}$). Tasks were processed in the First-In First-Out fashion.

The problem of scheduling a divisible job on a bus system in the presence of background activities is investigated in [190]. It is assumed that the speed of processors and communication links is inversely proportional to the number of tasks sharing a processor or a link. The arrival of a background task reduces the speed observed by the considered application. The method of computing deterministically the optimal load distribution is given for the case in which the arrival times and departures of background tasks are known. When the above parameters of the background tasks are unknown a probabilistic analysis is presented.

In [34] the idea of divisible job is applied in scheduling and performance analysis for hypercube networks. The startup time was assumed to be negligible ($S = 0$).

The first paper including startup time S in the model of communication time is [33]. For linear networks (chain, ring) and homogeneous hypercube the optimal distribution of the load can be found in low order polynomial

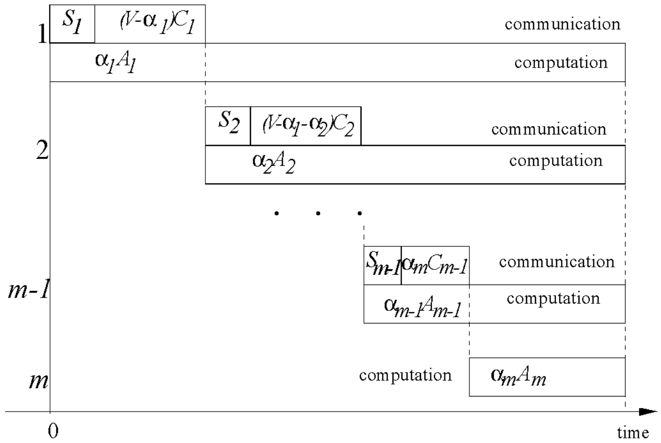

Figure 6.1: Communication and computation in chain interconnection.

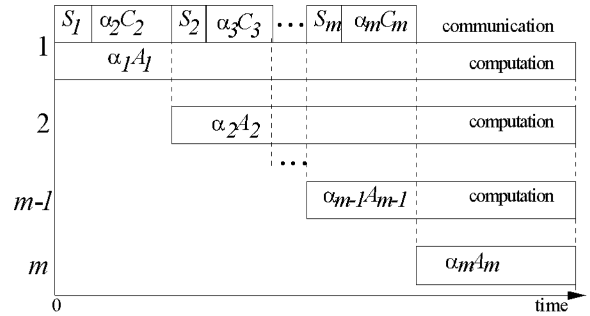

Figure 6.2: Communication and computation in star interconnection.

time. However, in general case scheduling on arbitrary interconnection graph, and on arbitrary bus system in particular, requires determining the optimal sequence of communications which is s**NP**h. For a star network polynomial cases are identified. We will describe in more detail chain and star networks as starting points for considerations in the further sections. The Gantt chart of communications and computations in the chain network and in the star network are depicted in Fig. 6.1 and Fig. 6.2. For simplicity of presentation we assume that no results are returned to the originator. It will be demonstrated later that this restriction can be removed. Assume, that PEs have network processors and the originator is located at the chain's end. In the chain network the part of load which is not processed by the originator is sent to the nearest neighbor. The neighbor divides the received data into a part processed locally and re-sends the rest to the next idle processor. This procedure is repeated until the last processor. Since no data is returned all the processors must stop at the same moment of time [33, 60]. The computing time on the PE sending data lasts as long as communication to and computing on the receiver (cf. Fig. 6.1). Thus, we can find the distribution of the load from the following set of equations:

$$\begin{aligned} \alpha_i A_i &= S_i + (\alpha_{i+1} + \ldots + \alpha_m)C_i + \alpha_{i+1}A_{i+1} \quad i = 1, \ldots, m-1 \qquad (6.1) \\ V &= \alpha_1 + \alpha_2 + \ldots + \alpha_m \\ & \alpha_1, \alpha_2, \ldots, \alpha_m \geq 0 \end{aligned}$$

where S_i, C_i are parameters describing a link joining P_i and P_{i+1}, and A_i is processing rate for P_i. The above equation set can be solved in $O(m)$ time. Yet, it may happen that a feasible solution does not exist [33]. In such a case less than m processors can solve the problem. The maximum number of usable processors can be found by binary search over m. When the results are returned the above equation set must be modified in such a way that while P_i computes, the spare data is sent to P_{i+1}, processed on $P_{i+1}, \ldots, P_m$ and results are returned to P_i. Thus, equations (6.1) have form (for $i = 1, \ldots, m-1$):

$$\alpha_i A_i = 2S_i + (\alpha_{i+1} + \ldots + \alpha_m)C_i + \alpha_{i+1}A_{i+1} + \beta(\alpha_{i+1} + \ldots + \alpha_m)C_i \qquad (6.2)$$

where $\beta(x)$ is the amount of results returned for x units of data (in simple cases $\beta(x)$ is constant or linear function). For the star network it was assumed that the originator is located in the center of the star, each PE has a network processor and no data is returned. Observe that computing on P_i lasts as long as communicating to and processing on P_{i+1} (cf. Fig. 6.2). Hence, we have a set of equations from which optimal distribution of the load can be found:

$$\begin{aligned} \alpha_i A_i &= S_i + \alpha_{i+1}C_{i+1} + \alpha_{i+1}A_{i+1} \quad i = 1, \ldots, m-1 \qquad (6.3) \\ V &= \alpha_1 + \alpha_2 + \ldots + \alpha_m \\ & \alpha_1, \alpha_2, \ldots, \alpha_m \geq 0 \end{aligned}$$

Note that all the communications are performed by the originator in the center of the star. Analogous situation takes place in the bus network because bus cannot be used by multiple communications at the same moment of time. Therefore, the originator sends the data to the consecutive processors in sequel and equation set (6.3) can be applied to find the data distribution in the bus system.

Two-dimensional rectangular mesh network with store-and-forward communication mode is considered in [35] ($P, 2D\text{–}mesh, s\&f \mid div, n = 1 \mid C_{max}$). A better communication algorithm based on circuit-switched communication mode is applied to distribute computation in [36] ($P, 2D\text{–}mesh, csw \mid div, n = 1 \mid C_{max}$). In the former work $S = 0$, and in the latter one $S \geq 0$ were assumed.

The problem of scheduling in a star network is tackled again in [17]. A new data distribution pattern based on pipelining is proposed. The data is distributed in greater number of small chunks rather than in one big chunk to each processor in sequel. This results in improved performance.

In work [18] scheduling in a chain network is considered in which PEs are equipped with 1-port network processors. This means that a PE can communicate only over one link at a time. The originator sends the share of data to be processed directly to a particular PE. The network processors of intermediate PEs facilitate these transfers. Thus, any PE can start computing right after receiving its share of data, without waiting for the load to be re-sent to the following PEs. Again, this improves the performance.

A bus system with network processors investigated in [191]. Two criteria are minimized: computing time and cost $Q, bus \mid div, n = 1 \mid X$. Cost of computation is calculated per unit of load. For the minimal cost PEs should be activated in nondecreasing order of their costs per data unit. Two algorithms are proposed: finding minimal execution time for the given cost, and finding minimal cost for the given execution time.

The results mentioned in this section are collected in Table 6.3.

6.3 Applying Divisible Task Concept

6.3.1 Chain Interconnection

In this section a new data distribution scheme based on circuit-switched communication is proposed and compared with the previously known methods [60]. In the following we assume that all PEs have network processors and all PEs can simultaneously transmit over both ports. Hence, we consider problem $Q, chain, csw \mid div \mid C_{max}$. Moreover, we assume that the originator is located in the center of the chain and results are not returned.

Let us repeat after Section 2.1 that in the circuit-switching routing (unlike in the store-and-forward) the time of data transfer does not depend significantly on the distance between the sender and the receiver. The same situation takes place for packet-switched communication. Accor-

Table 6.1: Number of PEs activated while scattering in a chain.

Step number	Initially active	Activated	Finally active
1	1	2	3
2	3	6	9
3	9	18	27
...	...	...	...
h	3^{h-1}	$2 \cdot 3^{h-1}$	3^h

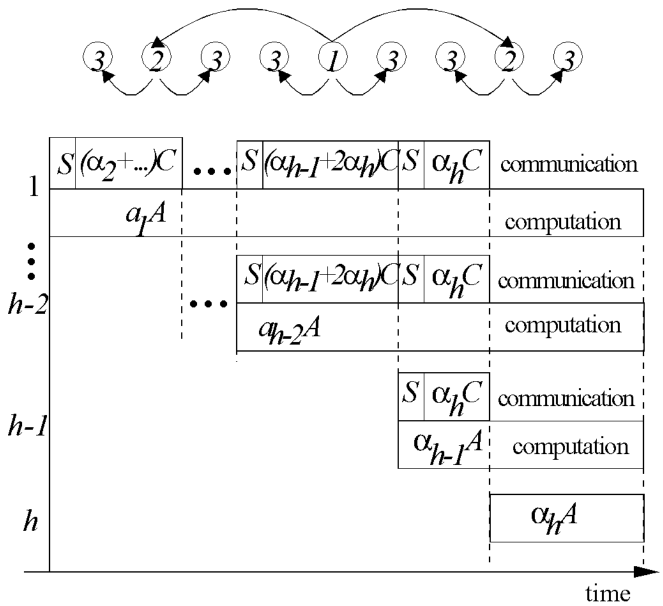

Figure 6.3: Communication and computation in a chain with circuit-switched commutation.

ding to conventions adopted in the previous sections when talking about circuit-switching communication we mean all modes with the above model of communication delay. Since communication delay does not depend on distance it can be advantageous to send some data far ahead and then redistribute it from two (or more) points. Thus, the originator sends data simultaneously to two distant PEs. In the next step both the originator and the two previously activated PEs send data to two new processors. The process is repeated until activating all the PEs after $h = \lceil \log_3 m \rceil$ steps. In Table 6.1 we demonstrate how the number of active (i.e. computing) processors is growing with consecutive steps of data distributing. The process of data distributing is depicted in Fig. 6.3 for $m = 9$. In Fig. 6.3 we present also a diagram of communication and computing in a chain with the above communication algorithm. When no results are returned, all the PEs must finish processing their parts of data at the same moment of time. The correctness of this observation has been demonstrated under very general assumptions in [33]. Here, it can be explained in the following way. Suppose to the contrary that one PE of the two activated from the same "parent" PE finishes earlier than the second one. Then, by balancing the load between the two descendants of the same "parent" PE would reduce the total length of the schedule. This reasoning can be repeated recursively until the originator. The case of non-zero data return time can be easily included as demonstrated in equations (6.2). Before proceeding to the solution of the problem let us remind that we denote:

V - the whole volume of data to be processed,
A - the processing rate of all processors,
C - the communication rate of all links,
S - the startup time of all links,
h - the number of steps in data distribution algorithm,
α_i - the amount of data assigned to PEs activated in step $i=1,\dots,h$.

Observe (cf. Fig. 6.3) that time of computing on the sending PE is equal to time of communicating to and computing on the receiving PE. Thus, the following set of equations can be formulated:

$$
\begin{aligned}
\alpha_{h-1}A &= S + \alpha_h C + \alpha_h A \\
\alpha_{h-2}A &= S + (\alpha_{h-1} + 2\alpha_h)C + \alpha_{h-1}A \\
\alpha_{h-3}A &= S + (\alpha_{h-2} + 2\alpha_{h-1} + 6\alpha_h)C + \alpha_{h-2}A \\
\dots & \\
\alpha_{h-i}A &= S + (\alpha_{h-i+1} + 2\sum_{j=2}^{i} 3^{j-2}\alpha_{h-i+j})C + \alpha_{h-i+1}A \qquad (6.4) \\
\dots & \\
\alpha_1 A &= S + (\alpha_2 + 2\sum_{j=2}^{h-1} 3^{j-2}\alpha_{j+1})C + \alpha_2 A \\
V &= \alpha_1 + 2\sum_{j=2}^{h} 3^{j-2}\alpha_j \\
& \alpha_1, \alpha_2, \dots, \alpha_k \geq 0
\end{aligned}
$$

The above set of equations can be solved in $O(\log\ m)$ time provided a feasible solution exists. The solution method uses the first $h-1$ equations of (6.4)

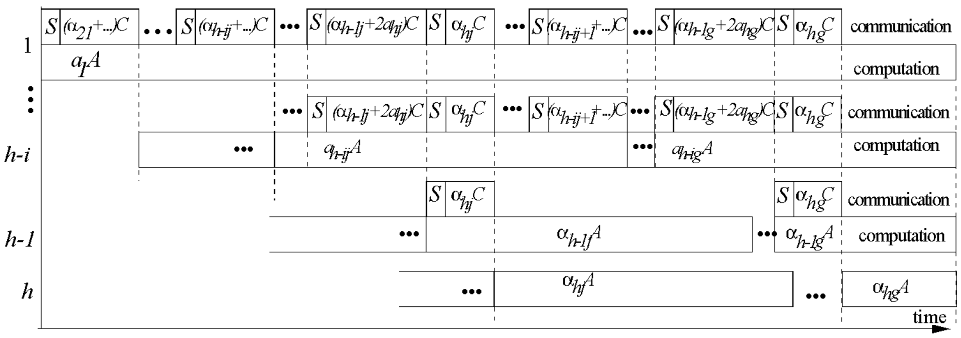

Figure 6.4: Communication and computation diagram in chain with pipelining and circuit-switched commutation.

to reduce α_{h-i} $(i = 1, \dots, h-1)$ to a linear function of α_h, i.e. $\alpha_{h-i} = k_{h-i}\alpha_h + l_{h-i}$. Coefficients k_{h-i}, l_{h-i} are (from (6.4)):

$$k_{h-i} = \frac{C}{A}(k_{h-i+1} + 2\sum_{j=2}^{i} 3^{j-2}k_{h-i+j}) + k_{h-i+1}$$

$$l_{h-i} = \frac{S}{A} + \frac{C}{A}(l_{h-i+1} + 2\sum_{j=2}^{i} 3^{j-2}l_{h-i+j}) + l_{h-i+1}$$

While α_h is found from the last equation of (6.4):

$$\alpha_1 = \frac{V - 2\sum_{i=2}^{h} 3^{i-2}l_i - l_1}{2\sum_{i=2}^{h} 3^{i-2}k_i + k_1}$$

The communication pattern can be further improved by applying pipelining of communications as proposed for star network in [17]. The communication and computation diagram for such a case is presented in Fig. 6.4. Note that now the data distribution consists of g pipeline stages each of which consists (as previously) of h steps. Let us denote by $\alpha_{i,j}$ the amount of data processed by each of PEs activated in step i of data distribution and in pipeline stage j. Since results are not returned, for the last stage it can be observed that all PEs must finish simultaneously. Hence, computing on the sender must last as long as communicating to and computing on the receiver. According to [17] we assume that PEs activated in step i of stage j $(j = 1, \dots, g-1)$ of pipelining must compute as long as sending data in steps $i+1, \dots, h$ of stage j and steps $1, \dots, i$ of stage $j+1$. In other words, PEs are processing the current portion of the load exactly until receiving the next portion of data. This assumption can be motivated as follows: Suppose it receives less, then there is an idle time and the schedule could be made shorter by avoiding the idle time. Suppose it receives more, then the PEs activated in the next step wait for data longer than necessary and start computing later than it is possible, thus increasing the idle time. The originator computes as long as all the communication lasts plus the time of computing the $\alpha_{h,g}$, i.e. when all PEs compute only and no communication takes place. This discussion can be summarized in the set of equations:

$$
\begin{aligned}
\alpha_{h-1,g}A &= S + \alpha_{h,g}C + \alpha_{h,g}A \\
&\dots \\
\alpha_{h-i,g}A &= S + (\alpha_{h-i+1,g} + 2\sum_{j=2}^{i} 3^{j-2}\alpha_{h-i+j,g})C + \alpha_{h-i+1,g}A \\
&\dots \\
\alpha_{2,g}A &= S + (\alpha_{3,g} + 2\sum_{j=2}^{h-2} 3^{j-2}\alpha_{j+1,g})C + \alpha_{3,g}A \\
&\dots \\
\alpha_{h-i,j}A &= \sum_{p=h-i+1}^{h} [S + C(\alpha_{p,j} + 2\sum_{q=1}^{h-p} 3^{q-1}\alpha_{p+q,j})] + \\
&\quad + \sum_{p=2}^{h-i}[S + C(\alpha_{p,j+1} + 2\sum_{q=1}^{h-p} 3^{q-1}\alpha_{p+q,j+1})] \qquad (6.5) \\
&\dots \\
\alpha_1 A &= \sum_{j=2}^{g}\sum_{p=1}^{h}(S + C(\alpha_{p,j} + 2\sum_{q=1}^{h-p} 3^{q-1}\alpha_{p+q,j})) + \alpha_{h,g}A \\
V &= \alpha_1 + 2\sum_{j=1}^{g}\sum_{i=2}^{h} 3^{i-2}\alpha_{i,j} \\
&\alpha_{i,j} \geq 0
\end{aligned}
$$

The above equation set can be solved analogously to (6.4) by expressing all the unknowns as linear functions of $\alpha_{h,g}$. Thus, $\alpha_{i,j} = k_{i,j}\alpha_{h,g} + l_{i,j}$ and for $i = 2, \dots, h$

$$k_{h-i,g} = \frac{C}{A}(k_{h-i+1,g} + 2\sum_{j=2}^{i} 3^{j-2}k_{h-i+j,g}) + k_{h-i+1,g}$$

$$l_{h-i,g} = \frac{S}{A} + \frac{C}{A}(l_{h-i+1,g} + 2\sum_{j=2}^{i} 3^{j-2}l_{h-i+j,g}) + l_{h-i+1,g}$$

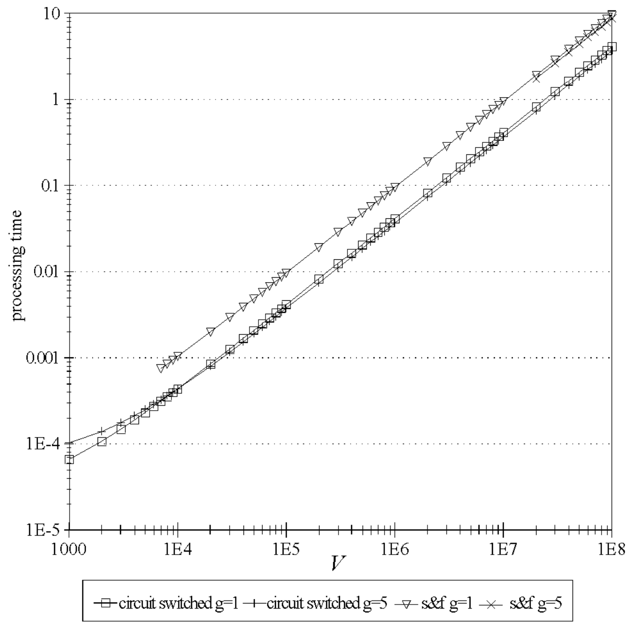

Figure 6.5: Comparison of the algorithms based on store-and-forward and on circuit-switched commutation in a chain network.

and for $i = 2, \ldots, h; j = 1, \ldots, g-1$:

$$k_{h-i,j} = \frac{C}{A} \sum_{p=h-i+1}^{h} (k_{p,j} + 2\sum_{q=2}^{h-p} 3^{q-1} k_{p+q,j}) + \sum_{p=2}^{h-i} (k_{p,j+1} + 2\sum_{q=1}^{h-p} 3^{q-1} k_{p+q,j+1})$$

$$l_{h-i,j} = \sum_{p=h-i+1}^{h} [\frac{S}{A} + \frac{C}{A}(l_{p,j} + 2\sum_{q=2}^{h-p} 3^{q-1} l_{p+q,j})] + \sum_{p=2}^{h-i} [\frac{S}{A} + \frac{C}{A}(l_{p,j+1} + 2\sum_{q=1}^{h-p} 3^{q-1} l_{p+q,j+1})]$$

and

$$k_1 = \frac{C}{A} \sum_{j=1}^{g} \sum_{p=2}^{h} (k_{p,j} + 2\sum_{q=1}^{h-p} 3^{q-1} k_{p+1,j}) + 1$$

$$l_1 = \sum_{j=1}^{g} \sum_{p=2}^{h} (\frac{S}{A} + \frac{C}{A}(l_{p,j} + 2\sum_{q=1}^{h-p} 3^{q-1} l_{p+1,j}))$$

We compared the above two methods based on circuit-switching routing with a method proposed in [60] and based on store-and-forward routing. The results are collected in Fig. 6.5 where times of processing various size tasks on a network of 27 PEs with $A = 1\mu s/byte, C = 0.01\mu s/byte, S = 10\mu s$ are juxtaposed. The figure presents times for store-and-forward routing and for circuit-switched routing. For both commutation methods two pipelining schemes are considered $g = 1$ (no pipelining), and $g = 5$. Not for all sizes the above patterns are feasible. When $g = 5$ only for big volumes computing on all 27 PEs is possible using store-and-forward. For smaller sizes the computation is finished on smaller number of PEs before the furthest ones are activated. Such situations are not included in Fig. 6.5. As it can be seen the algorithm based on circuit-switched routing is significantly better. Pipelining gives much smaller reduction (about 10%) in computing time than replacing the old algorithm using store-and-forward with the new one exploiting circuit-switched commutation (reduction by about 50%).

6.3.2 Star and Bus Interconnections

In this section we consider star and bus interconnections. It has been observed earlier [33] that star and single bus networks can be viewed in the same way. Thus, we will use here only the star name. The star interconnection is an attractive model of distributed computations, e.g. of the PVM master-worker concept. Hence, in the following discussion names originator and master are equivalent. We begin this section with some theoretical points. Next, we analyze some on-line scheduling algorithms. Finally, we practically verify the considered model.

Though the complexity of problem $Q, star \mid div, n = 1 \mid C_{max}$, i.e. scheduling a divisible task on a star is not established when the startup times are nonzero, the optimal solution can be found from the following mixed linear programming formulation.

$$\begin{aligned}
\text{minimize} \quad & C_{max} \\
\text{subject to} \quad & \\
C_{max} &= \alpha_1 A_1 && (6.6)\\
C_{max} &= \sum_{k=2}^{j}\sum_{i=1}^{m} x_{ki}[C_i\alpha_i + S_i] + \sum_{i=1}^{m} x_{ji}A_i\alpha_i \\
& \qquad\qquad j = 2, \ldots, m && (6.7)\\
V &= \sum_{i=1}^{m}\alpha_i && (6.8)
\end{aligned}$$

$$1 = \sum_{i=1}^{m} x_{ij} \qquad j = 1, \ldots, m \tag{6.9}$$

$$1 = \sum_{j=1}^{m} x_{ij} \qquad i = 1, \ldots, m \tag{6.10}$$

$$x_{ij} \in \{0, 1\} \qquad i, j = 1, \ldots, m \tag{6.11}$$

$$\alpha_1, \alpha_2, \ldots, \alpha_m \geq 0$$

In the above formulation $x_{ij} = 1$ denotes that P_j is activated as the ith in the sequence. $x_{ij} = 0$ denotes the opposite situation. Equation (6.6) demands that P_1 computes all the time, equations (6.7) that the communication to the processor activated as the jth immediately follows activating of the previous one. Furthermore, computing on all processors finishes at the end of the schedule. Equations (6.8) impose processing of the whole load, (6.9) that each PE is activated, (6.10) that each position in the activations sequence is filled. The above formulation may have no feasible solution if it is not possible to activate all PEs before processing the whole load on a smaller number of PEs. In such a case one may reduce the value of m in the formulation until a feasible (and optimal) solution is found. We proposed a solution based on mixed linear programming for the case without returning results (cf. equations (6.3)). Yet, it would not be a difficult task to reformulate it and include returning results.

Divisible Task Scheduling in Distributed Batch System

In [7, 91] the problem of finding the optimal set of processors to execute a distributed application is considered. This problem is related, for example, to distributed batch schedulers like NQS, LL, LSF, PRM, etc. The system assumed in [7, 91] allows for executing only one application on a processor at a time. Changing the assigned processors during the execution is not allowed. The application is submitted to a scheduler which runs it on available processors such that the completion time is minimized. The central scheduler "knows" which processors are available and the moments when the busy processors will become available. Immediate starting of the application on small number of available processors may not be optimal. Furthermore, delaying the start time until more processors are available may reduce the completion time. In [7, 91] the set of processors and the starting moment was selected using ECT (Earliest Completion Time) rule. In the above publications no communication overhead was considered. In the following we examine such a computing system using divisible task concept. Let us denote by b_i $(i = 1, \ldots, m)$ the moment of time at which processor P_i becomes available. Without loss of generality we assume $b_1 \le b_2 \le \ldots \le b_{m+1} = \frac{V}{A_1+C_1} + b_1 + S_1$, where b_{m+1} is an upper bound on the schedule length introduced to simplify the presentation. For the sake of simplicity we assume that results are not returned.

Consider the case when before b_i communication to P_i is not possible, after b_i both communication and processing on P_i are possible. An immediate approach to this problem is to follow the lines of [7, 91] and use the ECT rule. The scheduler starts communication to processors $\{P_1, \ldots, P_i\}$ $(i = 1, \ldots, m)$ at the some moment b_i for some selected i. Distribution of the load and the completion time can be calculated using (6.3). This can be done in $O(m)$ time for each i, which results in $O(m^2)$ complexity.

A different approach is allowing for successive activating PEs as they become available. Thus, P_1 would start computing first after receiving its share of data, then P_2 would be activated etc. In this case minimal execution time is determined by the processing capacity available on the activated processors. Let us denote by e_i the time moment at which communication to processor P_i stops and computation begins. At e_i communication to P_{i+1} can begin provided that $b_{i+1} \le e_i$. Hence, P_i can process $\alpha_i = \frac{C_{max} - \max\{e_{i-1}, b_i\} - S_i}{C_i + A_i}$ units of data, where $e_i = \max\{e_{i-1}, b_i\} + \alpha_i C_i + S_i$ $(i = 1, \ldots, m)$ and $e_0 = b_1$. For the given C_{max} the total amount of load that can be processed is $\sum_{i=1}^{j} \alpha_i$, where $b_j < C_{max} \le b_{j+1}$. By binary search over j the earliest interval for which the total amount of load that can be processed is greater than or equal to V and where the optimal length of the schedule is, can be determined in $O(m \log m)$ time. This procedure can be further applied to find the optimal schedule length C^*_{max}. Suppose ε is the precision of C^*_{max} calculation. Then, the complexity of the binary search would be $O(m(\log m + \log \max_j\{b_j - b_{j-1}\} - \log \varepsilon))$. We cannot use equations (6.3) in this case because there are intervals when the processor activated earlier (say P_i) is not communicating with the originator while the processor activated as the next one (P_{i+1}) cannot start communicating because it is not available yet. Thus, to solve the problem with a perfect precision a stronger tool seems necessary. Before going into further details let us assume that the above procedure has been applied and it is determined that $C^*_{max} \in [b_j, b_{j+1}]$. The minimal length of the schedule and distribution of the load can be found from the following linear program.

$$\min C_{max}$$

subject to

$$
\begin{aligned}
C_{max} &\geq e_i + A_i\alpha_i && \text{for } i = 1, \dots, j && (6.12)\\
e_i &\geq e_{i-1} + C_i\alpha_i + S_i && \text{for } i = 2, \dots, j && (6.13)\\
e_i &\geq b_i + C_i\alpha_i + S_i && \text{for } i = 1, \dots, j && (6.14)\\
\sum_{i=1}^{j} \alpha_i &\geq V && && (6.15)
\end{aligned}
$$

$$e_1, \dots, e_j, \alpha_1, \dots, \alpha_j, C_{max} \geq 0$$

In the above formulation equations (6.12) guarantee that computing times fit in the available time intervals, equations (6.13), (6.14) guarantee that the proper sequence of communications is preserved and no communication starts before the processor becomes available. Equation (6.15) ensures complete processing of the task. The above formulation has at most $2m + 1$ variables and $3m$ constraints and can be formulated and solved in polynomial time. Hence, the problem is polynomially solvable.

Now, consider a situation when PEs have network processors with satisfactory buffers allowing for sending new data to processor P_i while it is still computing the pervious task. Then, communication is allowed even before b_i, but processing is allowed only after this time. This case can be solved similarly to the previous one. The amount of load processed by processor P_i depends on the starting time of the computations. The starting time for computations is either b_i if the communication to P_i finishes before the processor becomes available or it is the moment when the communication to P_i finishes, i.e. e_i. In the first case P_i processes $\alpha_i = \frac{C_{max} - b_i}{A_i}$. Since $e_i = e_{i-1} + \alpha_i C_i + S_i \leq b_i$ this case implies $e_{i-1} \leq b_i - \frac{C_i}{A_i}(C_{max} - b_i) - S_i$. In the second case P_i computes $\alpha_i = \frac{C_{max} - e_{i-1} - S_i}{C_i + A_i}$. Thus, for a given C_{max} the number of used processors j can be determined from conditions $b_j < C_{max} \leq b_{j+1}$. Then, the capacity for processing the load can be found as a sum $\sum_{i=1}^{j} \alpha_i$, where α_i are calculated according to the two above cases. This results in $O(m \log m)$ complexity binary search procedure determining the set of required processors for load V. As before, one can extend this procedure to find the optimal length of the schedule with some accuracy. When perfect precision is required one may use linear program (6.12)-(6.15) with equations (6.13) in range $i = 1, \dots, j$ and $e_0 = 0$, while (6.14) should be replaced with

$$e_i \geq b_i \text{ for } i = 1, \dots, j.$$

Thus, also this case can be solved in polynomial time.

On-line Algorithms

The methods of data distribution presented above are well suited for computer systems dedicated to one application only where the processing speed and communication speed are stable. Yet, in distributed computations based on LAN/MAN/WANs stability of these parameters is hard to be guaranteed. Thus, a different adaptive scattering algorithm seems to be required. By an "adaptive" algorithm we mean here a method which makes no assumptions on the speed of computer and communication media. The A_i parameters depend on the background loading of processors and on the application with its current data. Therefore, it is hard to use some standard benchmark to estimate A_is. Note that the same problem appears in loop scheduling (cf. Section 4.4). On the other hand, the application itself is a good benchmark. Thus, we conclude that the best way of calculating A_is is doing it while executing the particular application for the particular data set. The communication algorithm proposed in [61, 100] for processing divisible tasks causes that a lot of data is sent to the PE activated as the first one. Consequently, the first communication time is very long. In the meantime, the other PEs are unnecessarily idle. It is more efficient to send small chunks of data to all PEs and let them start computing earlier. Thus, the communication pattern should be based on pipelining as the one proposed in [17]. With the above observations in mind we propose the following scheduling algorithm.

Distribution Algorithm 1 (DA1)

1: Send to all processors the same initial amount $\alpha_i = \alpha$ of data to process.
2: While there is anything to send, send to idle processor P_i amount $\alpha_i' = \alpha_i \frac{\tau}{\sigma}$ of data to process, where α_i is the amount of data sent in the previous activation of P_i, τ is the required length of the interval between accesses to the originator and σ is the observed interval between two accesses.

In the above algorithm the PE that returns results earlier is sent a bigger chunk of data than the PE that returns the results later. The key idea behind the above algorithm is to obtain a fixed interval τ between the accesses of different PEs to the originator. This reduces the contention in accessing the originator. Below, we analyze the behavior of DA1. Without loss of generality we assume that the amount of returned results is equal to the amount of data to process. Furthermore, we assume that originator is not computing (performs only control and communication functions) and the number of slave processors is m.

Lemma 6.1 *In the contention-free situation and for stable parameters of communication links and PEs, the interval between the accesses to the originator in DA1 converges to* τ.

Proof In the assumed contention-free situation the access to the originator of some processor does not coincide with the access of any other processor. Let us denote by σ, σ', σ'' the time between three consecutive accesses of P_i to the originator. Let α, α', α'' denote the amounts of data sent to P_i in the three respective accesses. Remember, that we assume the amount of returned results being equal to the amount of data. Observe that $\sigma = \alpha(2C_i+A_i)+2S_i$, $\sigma' = \alpha'(2C_i + A_i) + 2S_i$, $\sigma'' = \alpha''(2C_i + A_i) + 2S_i$. Now, let us calculate $\sigma'' - \sigma' = \alpha''(2C_i + A_i) + 2S_i - \alpha'(2C_i + A_i) + 2S_i = (\alpha'' - \alpha')(2C_i + A_i)$. On the other hand, $\alpha' = \alpha\frac{\tau}{\sigma} = \frac{\tau}{2C_i+A_i+\frac{2S_i}{\alpha}}$ and $\alpha'' = \alpha'\frac{\tau}{\sigma'} = \frac{\tau}{2C_i+A_i+\frac{2S_i}{\alpha'}}$. Thus,

$$\sigma'' - \sigma' = \frac{2\tau S_i(2C_i+A_i)(\frac{1}{\alpha}-\frac{1}{\alpha'})}{(2C_i+A_i+\frac{2S_i}{\alpha'})(2C_i+A_i+\frac{2S_i}{\alpha})} = \frac{2\tau S_i(2C_i+A_i)(1-\frac{\sigma}{\tau})}{\alpha(2C_i+A_i+\frac{2S_i}{\alpha'})(2C_i+A_i+\frac{2S_i}{\alpha})}.$$

From the above we can infer about the direction of σ'', σ' changes. Suppose $\frac{\tau}{\sigma} > 1$ then $\alpha' > \alpha$ and $\sigma' > \sigma$. Furthermore, $\sigma'' > \sigma'$ and $\alpha'' > \alpha'$. Thus, the amount of data sent to P_i is increased in two consecutive steps. For $\frac{\tau}{\sigma} < 1$ the amount of data sent to P_i is gradually decreased. This can be inductively extended to all accesses to the originator. Note that direction of changes is constant. Hence, the time between two consecutive accesses cannot become greater than τ if it was smaller than τ initially. And vice versa, if the interval between the accesses was greater than τ initially then it cannot become smaller than τ.

Finally, consider the distance from τ in consecutive data distribution steps. We will calculate how the distance from τ changes. $(\sigma''-\tau)-(\sigma'-\tau) = \sigma'' - \sigma'$. For $\frac{\tau}{\sigma} < 1$ the data chunks are decreasing and the distance from τ is decreasing in consecutive steps. For $\frac{\tau}{\sigma} > 1$ the data chunks are increasing. The time between the accesses is approaching τ from below and distances $(\sigma'-\tau)$ and $(\sigma''-\tau)$ are negative values. Thus, in the latter steps the distance is "less negative" and the absolute value of deviation from τ decreases. We conclude that in the conditions stated above the time between the accesses to the originator monotonically converges to τ. □

In the following lemma we discuss the execution time of an application with DA1 assuming that access intervals of all PEs are equal to τ. This is a slight simplification because the initial adjusting of data chunks is neglected. Furthermore, we assume that communications take place immediately one after another (cf. Fig. 6.6).

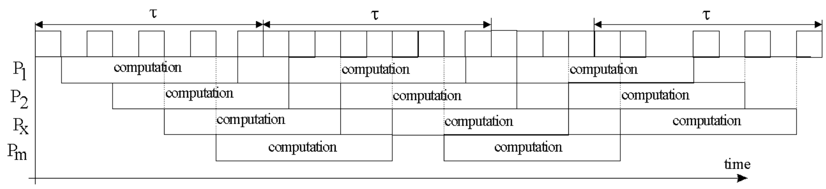

Figure 6.6: Computation - communication diagram for DA1.

Lemma 6.2 *When access intervals for all PEs are equal to τ, where* $V(2C_1+A_1)+2S_1 \geq \tau \geq \frac{\sum_{i=1}^m 2S_i(1-\frac{2C_i}{2C_i+A_i})}{1-\sum_{i=1}^m \frac{2C_i}{2C_i+A_i}}$ *the application execution time is* $\tau\lfloor\frac{V}{\sum_{i=1}^m \frac{\tau-2S_i}{2C_i+A_i}}\rfloor+C_1\frac{\tau-2S_1}{2C_1+A_1}+S_1+\sum_{i=2}^{x-1}(2C_i\frac{\tau-2S_i}{2C_i+A_i}+2S_i)+C_x\frac{\tau-2S_x}{2C_x+A_x}+S_x+\tau$ *where x satisfies* $\sum_{i=1}^x \frac{\tau-2S_i}{2C_i+A_i} \geq V-\lfloor\frac{V}{\sum_{i=1}^m \frac{\tau-2S_i}{2C_i+A_i}}\rfloor\sum_{i=1}^m \frac{\tau-2S_i}{2C_i+A_i}$.

Proof First, let us comment on the conditions set in the lemma. When τ is longer than the execution time on a single processor, i.e. $\tau \geq V(2C_1+A_1)+2S_1$, then the execution time remains constant and equal to $V(2C_1+A_1)+2S_1$. On the other hand, when τ is too short it may be impossible to communicate to all m processors. This is the case when communications are longer than τ. Since access intervals of all PEs are equal to τ, P_i processes amount of load equal to $\alpha_i=\frac{\tau-2S_i}{2C_i+A_i}$. Thus, $\tau \geq \sum_{i=1}^m(2C_i\alpha_i+2S_i)=\sum_{i=1}^m(2C_i\frac{\tau-2S_i}{2C_i+A_i}+2S_i)$, from which we obtain that τ must satisfy $\tau \geq \frac{\sum_{i=1}^m 2S_i(1-\frac{2C_i}{2C_i+A_i})}{1-\sum_{i=1}^m \frac{2C_i}{2C_i+A_i}}$ in order to make DA1 realizable. The execution time of the application under DA1 consists of two phases. In the first one all processors one by one are repetitively accessing the originator. The number of repetitions is $y=\lfloor\frac{V}{\sum_{i=1}^m \alpha_i}\rfloor=\lfloor\frac{V}{\sum_{i=1}^m \frac{\tau-2S_i}{2C_i+A_i}}\rfloor$. Thus, the first phase lasts $\tau\lfloor\frac{V}{\sum_{i=1}^m \frac{\tau-2S_i}{2C_i+A_i}}\rfloor$ units of time. In the second phase only some processors are activated. Let us denote by x the number of processors activated in this phase. It must be big enough to accommodate the remaining volume of data. Hence, $\sum_{i=1}^x \alpha_i \geq V-y\sum_{i=1}^m \alpha_i$. From which we obtain $\sum_{i=1}^x \frac{\tau-2S_i}{2C_i+A_i} \geq V-\lfloor\frac{V}{\sum_{i=1}^m \frac{\tau-2S_i}{2C_i+A_i}}\rfloor\sum_{i=1}^m \frac{\tau-2S_i}{2C_i+A_i}$. In this phase processors $P_2,\ldots,P_m$ return results from the last cycle of the first stage, while only processors $P_1,\ldots,P_x$ are activated for the last time. Thus, the last phase includes the time of sending data to P_1, receiving results from, and sending data to P_2, ... receiving results from, and sending data to P_x, time of processing data on P_x and returning results from P_x. Other communications and data processing activities take place in parallel with the above actions (cf. Fig. 6.6). The last sending to P_x, computing on it and returning results lasts τ units of time. Hence, the total processing time is $\tau\lfloor\frac{V}{\sum_{i=1}^{m}\frac{\tau-2S_i}{2C_i+A_i}}\rfloor+C_1\frac{\tau-2S_1}{2C_1+A_1}+2S_1+\sum_{i=2}^{x-1}(2C_i\frac{\tau-2S_i}{2C_i+A_i}+2S_i)+C_x\frac{\tau-2S_x}{2C_x+A_x}+S_x+\tau$ which is the amount of time specified above. □

From Lemma 6.2 we can conclude that when the number of cycles is big the first phase of DA1 dominates and the first term in the execution time formula dominates. Furthermore, when startup times are small in relation to τ then the total execution time tends to $\frac{V}{\sum_{i=1}^{m}\frac{1}{2C_i+A_i}}$. Hence, under the above conditions the total processing time does not depend on τ which is the only parameter that can be modified in the algorithm.

Corollary 6.1 *When the number of iterations is big and startup times are small the execution time under DA1 does not depend on τ.*

We compared DA1 with an algorithm in which the data chunk is constant for all processors:

Distribution Algorithm 2 (DA2)

While there is anything to send, send fixed amount v of data to a free PE.

In Fig. 6.7 and Fig. 6.8 we compared relative execution times of application executed under DA1 and under DA2 as a function of τ and v, respectively. Relative execution time is a ratio of the actual execution time and the execution time for the case when all PEs receive data in one chunk as described in [61, 100]. The results were calculated in a series of simulations. There were $m = 8$ PEs, the processing rate was $A_1 = \ldots = A_8 = 1$, in the first experiment, $A_1 = \ldots = A_8 = 0.1$ in the second, and $A_1 = \ldots = A_8 = 0.01$ in the third one. The communication links were identical with $C_1 = \ldots = C_8 = 0.001$, $S_1 = \ldots = S_8 = 0.1$ which is a typical relation between transfer rate and startup time (cf. Table 2.1). DA1 guarantees better performance than DA2 in the worst case. Yet, for big values of τ the execution time becomes unstable: for growing τ it first increases, then decreases and increases again (cf. Fig. 6.7). This anomalous behavior is a result of too big value of τ. To keep the interval between the accesses equal to τ one of the processors intercepts a big chunk of load. This results in the load imbalance which dominates the execution time. When τ is big the load remaining for the dominating last chunk is smaller than for small τ, because for big τ the

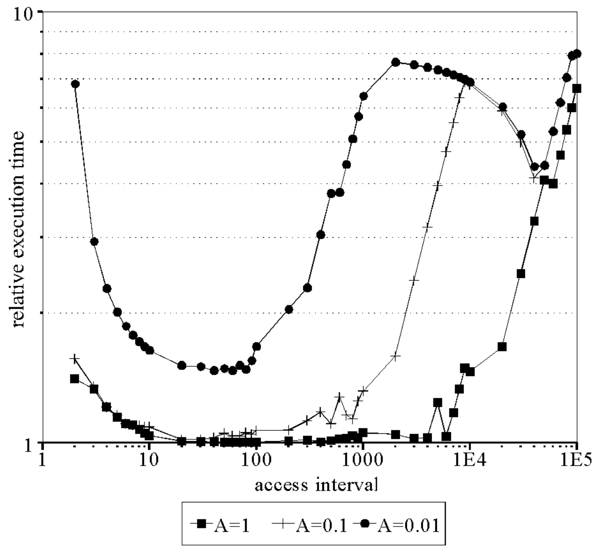

Figure 6.7: Execution time for DA1 - simulation.

chunks sent earlier were also bigger. Thus, increasing τ may reduce execution time. As it can be seen for small chunk sizes, where communication time dominates, the bigger the speed of processors, the worse DA2 is.

In Fig. 6.9 and 6.10 we present results of applying DA1 and DA2 in a cluster of six (including the originator) SUN workstations cooperating by use of PVM [88]. The distributed application consisted in distributed search for a pattern in a text file. The size of the file was V =5760kB. In the observed range of intervals between accesses to the master processor DA1 has better stability than DA2 and better worst-case performance. This situation is in accordance with simulations for small values of τ. As it can be observed DA2 exposes bad performance when chunk size is too small. This behavior is analogous to the one observed in simulations.

Though algorithm DA1 exposes good qualities in certain conditions (cf. Lemma 6.1, Corollary 6.1), it has also weaknesses when the value of τ is chosen badly. The drawbacks of DA1 are visible in Fig. 6.7. These are imbalances which contribute to a longer than necessary execution time. Hence, we proposed a new algorithm which adjusts value of τ.

Distribution Algorithm 3 (DA3)
Apply the DA1 algorithm with two exceptions:
1) if the originator is constantly busy in the second access cycle or later, then set $\tau := \tau * m$;

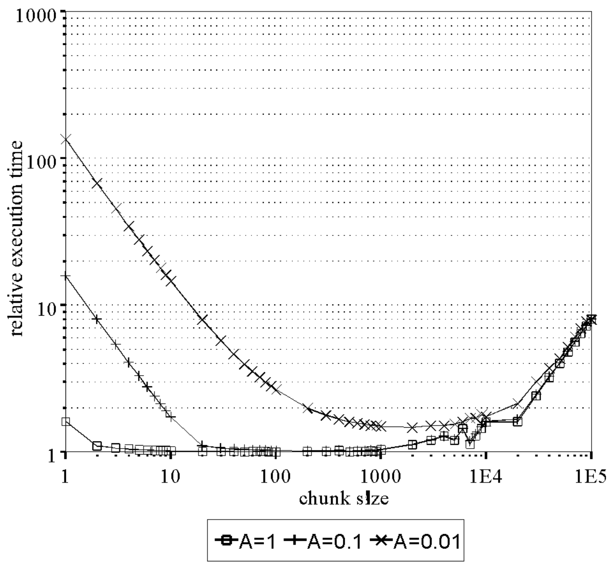

Figure 6.8: Execution time for DA2 - simulation.

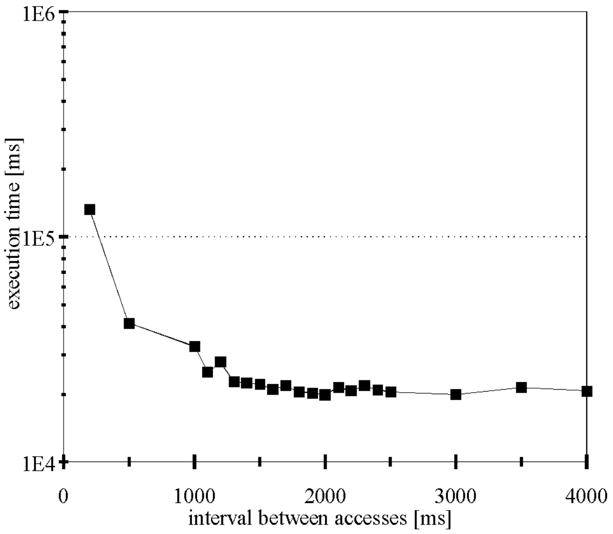

Figure 6.9: Execution time for DA1 in a workstation cluster.

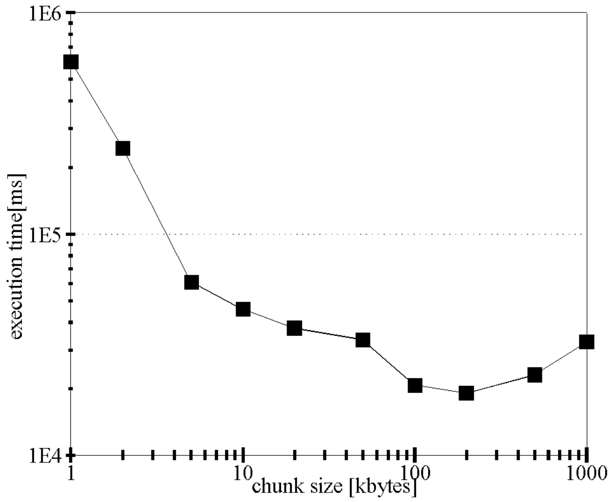

Figure 6.10: Execution time for DA2 in a workstation cluster.

2) if some PE wants to take more than $\frac{1}{m}$ of the remaining load and one of the two is true: master is not continuously occupied, or it is the first data distribution cycle, then contract $\tau := \frac{\tau}{m}$.

Let us comment on DA3. The first exception prevents communications and queuing to the master from dominating the whole execution time. The second exception is intended to prevent the imbalances. Observe that the two exceptions cannot take place simultaneously. We assume that initially master is not idle. The same simulations were performed for DA3 as depicted in Fig. 6.7 for DA1. The results are collected in Fig. 6.11. As it can be observed DA3 has better stability on average. However, for certain combinations of computer system parameters and the history of data distribution the set of conditions included in DA3 to prevent instability is not satisfactory. This results in "glitches" as e.g. in Fig. 6.11 for the initial value of $\tau = 200$.

The three data distribution algorithms are compared in one more way in Fig. 6.12. This chart presents dependence of the variance in processing time on changing speed of PEs. The following parameters were assumed: $m = 8, C_i = 0.001, S_i = 0.1, V = 10^5$. The processing rate A_i remains with uniform probability in the interval $[0.5, 1.5]$. The rate remained unchanged during random number of accesses to the PE. The number of accesses without change in speed was uniformly distributed in interval $[1, 5]$. Each point in the chart presents variation (i.e. standard deviation) of 20 experiments. As it can be observed DA1, DA2 are more stable than DA3 for small chunk

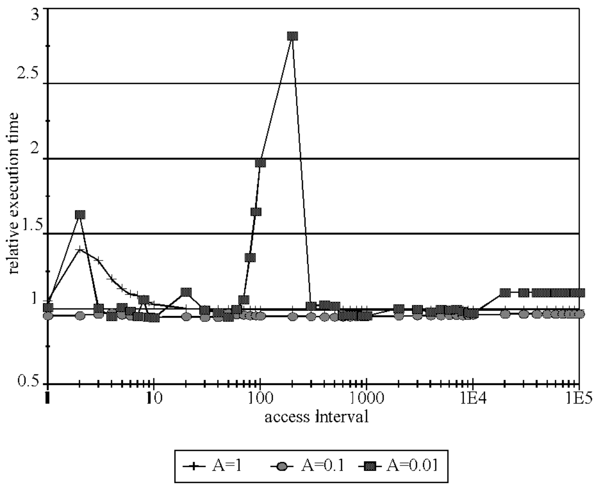

Figure 6.11: Execution time for DA3 vs. initial values of τ - simulation.

size, resp. τ. For big values of these parameters processing dominates in the total execution time, and hence changes of processing rates significantly influence total execution time. Variance of processing time for DA3 is similar over all (initial) values of τ. Hence, it can be claimed that on average DA3 is more impervious to the changes of the processing rate.

Verification in Transputer System

In the following paragraphs we present results of a practical verification of the divisible task concept in a star architecture.

The basic star model of e.g. [61] assumed that the results are returned in the inverted order of sending data. Here, we assumed that the results are returned in the same order as the data was sent (cf. Fig. 6.13). In such a situation the time of processing on processor P_i and returning results from this processor must be equal to the time of sending to P_{i+1} and processing on P_{i+1}. Hence, the basic equation set (6.3) must be modified as follows

$$\begin{aligned} \alpha_i A_i + S_i + \beta(\alpha_i) C_i &= S_{i+1} + \alpha_{i+1}(C_{i+1} + A_{i+1}) \quad i = 1, \ldots, m-1 \qquad (6.16) \\ V &= \alpha_1 + \alpha_2 + \ldots + \alpha_m \\ & \alpha_1, \alpha_2, \ldots, \alpha_m \geq 0 \end{aligned}$$

where $\beta(x)$ is the amount of results returned for x units of data. The above method has been practically applied in a T805 transputer network depicted in Fig. 6.14a. As it can be verified in Fig. 6.14a the underlying topology is

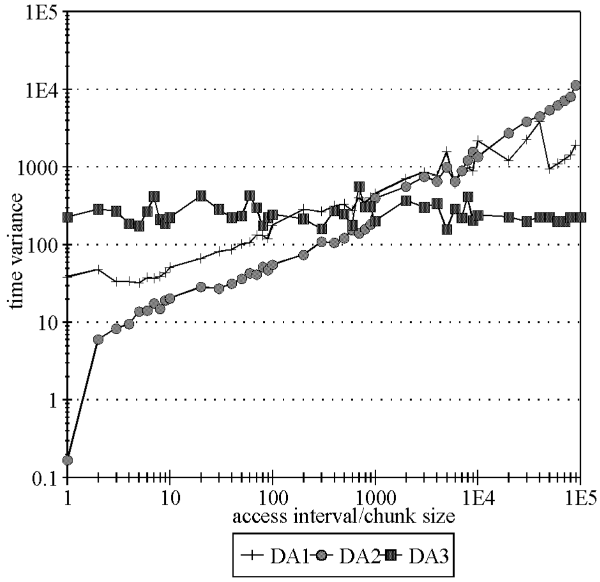

Figure 6.12: Variance of processing time when processing rate is changing.

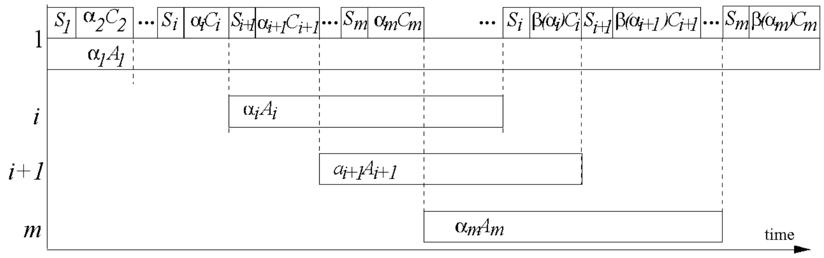

Figure 6.13: Communication and computation in a star. The sequences of data distribution and the results collection are the same.

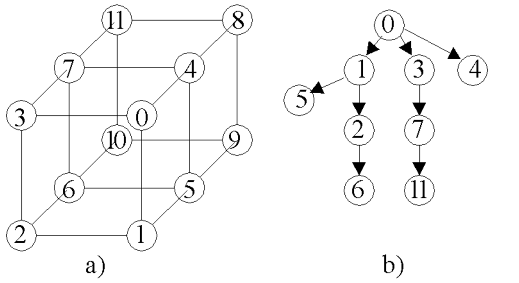

Figure 6.14: The transputer testbed: a) topology b) data distribution paths in the experiment with eight processors.

not a star. Thus, by a star we mean a logical interconnection observed in scattering. The communication algorithm is based on wormhole routing. The considered application was a search for a pattern in a text file. In all experiments the returned results fit in one 1000-byte packet. Hence, $\beta(\alpha_i) = 1000$. For simplicity of the experiment we assume that computations and communications do not overlap. Parameters A_i were measured for each PE as an average of 100 tests consisting in searching in 300000-byte file. Parameters C_i, S_i were calculated using linear regression from a set of transmission time measurements where the originator (labeled 0) sent to P_i messages of size $1, \dots, 100$ packets (which is range $2000, \dots, 102584$ of bytes with step 1016 bytes). The first experiment considered only a pair: the originator plus the PE labeled 11 and consisted in transferring and processing 300000 bytes of data. The difference between execution time measured experimentally and calculated was below 0.5%. In the next experiment we used three processors labeled 6, 9, 11, respectively. For only three computing processors the interconnection can be considered as a star. Fig. 6.15 presents an absolute value of relative difference between the expected and measured execution time. Every point is an average of 100 experiments. As it can be seen in Fig. 6.15 the difference decreases fast and for $V \geq 40000$ it is smaller than 10% while for $V \geq 300000$ it is below 1%. In the following experiment we tried to use eight processors. Yet, it turned out that the construction of a routing table caused that PEs were simultaneously computing and processing. This resulted in approx. 25% difference between the measurement and the expectation. Such a big discrepancy was caused by the fact that parameters A_i no longer reflected the speed of processing because on routing PEs the routing process competed for processing power with the application. We changed the data distribution sequence in accordance to the routing table such that the routing process is not activated together with application process. The topology of data distribution paths is depicted in Fig. 6.14b. We activated the PEs in the following order: 6, 2, 5, 1, 11, 7, 3, 4. As in [61] the results were returned in the inverted order of sending the data. In this way, we avoided simultaneous routing and processing by PEs. In Fig. 6.16 we present the difference between the expected and measured execution time. As it can be verified the difference is in the range $[-1.5\%, 1.5\%]$. We conclude that the practical verification proved viability of the proposed theory.

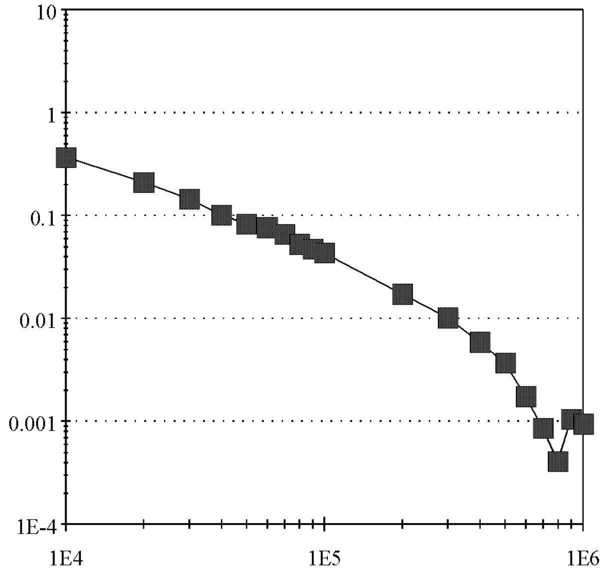

Figure 6.15: A relative difference between the expected and measured execution time for three processors.

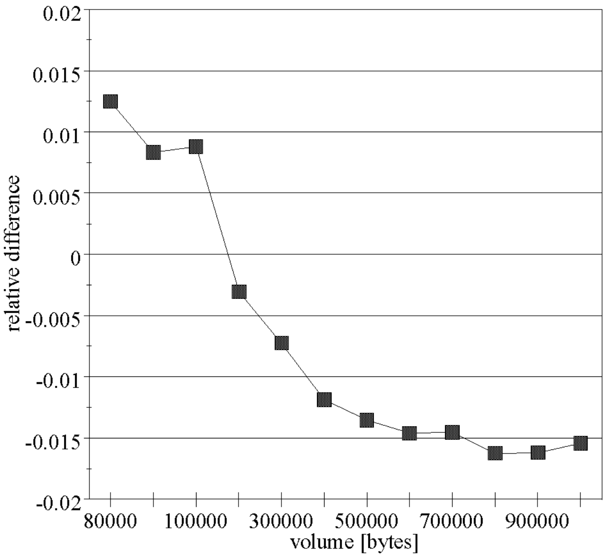

Figure 6.16: A relative difference between the expected and measured execution time for eight processors.

6.3.3 Hypercube

In this section we present data distribution methods based on divisible task concept for hypercube interconnection. Previous works on this architecture [34] assumed no communication startup costs and store-and-forward communication. Here we consider methods using circuit-switched communication for the system where simultaneous communication over all available PE links is possible (i.e. for d-dimensional hypercube PEs are d-port). PEs have network processors so that simultaneous communication and computation is possible. Furthermore, we assume that network processors are able to re-route data stream without additional intervention of the sender after transferring some fixed amount of data. Finally, all PEs and communication links are identical.

Let us remind that PEs of hypercube interconnection can be labeled such that the connected PEs differ in exactly one bit. In the following discussion as a measure of the distance of PEs from the originator we will use Hamming distance of the PE label and the originator label. Let us name by *layer of processors* the set of PEs activated simultaneously in the same step of data distribution. Moreover, as it was in [33, 34], we assume that layers consist of PEs differing in exactly one bit. For quick reference we will number layers according to the Hamming distance from the originator (number of 1 bits in a PE label, in other words). Note that the layer number may not coincide with the sequence of activating PEs. We will try to take advantage of circuit-switched communication for which the costs of sending data to the layers distant from the originator is similar to sending to a very close layer. We will present several methods of data distribution. However, due to space limitation only some of them are described in fine detail. The considered data distribution methods are illustrated in Fig. 6.17, where sequence of communications between layers is depicted.

Hypercube Distribution Algorithm 1 (H1) [33, 34]

In H1 all the data for layers $1, \ldots, d$ is sent from the originator (layer 0) to a layer 1. On the receipt of all that data PEs in layer 1 start processing their share of data and sending the rest of data to the following layers. Data for layers $i, \ldots, d$ ($i = 2, \ldots, d-1$) are sent from layer $i-1$ to layer i. Having received all data for layers $i, \ldots, d$, PEs in layer i start computing their part of data and sending the rest to layer $i+1$. H1 is based store-&-forward communication.

Hypercube Distribution Algorithm 2 (H2)

H2 is a modification of H1. The sequence of activating layers is as in H1. Yet, the PEs in consecutive layers first receive their part of the load and

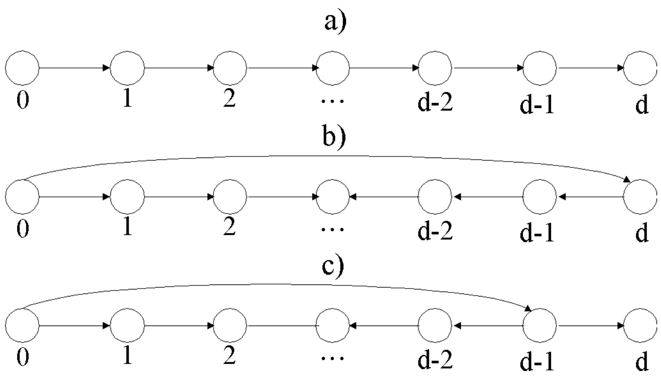

Figure 6.17: Data distribution methods for hypercube: a) H1,H2, b) H3, c) H4.

immediately start computing. The rest of data is re-routed by the network processors to the next layer without storing.

Hypercube Distribution Algorithm 3 (H3)

H3 is a modification of H2. As in H2 the PEs are activated right after receiving their share of data, while the rest of data stream is re-routed to the next layer. Yet, the data for layers $\lceil d/2 \rceil, \ldots, d$ are sent from layer 0 to "antipodes" of the hypercube, i.e. to layer d PE first. Then, the layers are activated from the opposite "ends" of the hypercube.

Hypercube Distribution Algorithm 4 (H4)

H4 is a modification of H3. The data for layers $\lfloor d/2 \rfloor, \ldots, d$ is sent from layer 0 to layer $d-1$. Then, layer d and layers $\lceil \frac{d-1}{2} \rceil, \ldots, d-2$ are fed from layer $d-1$. Layers $1, \ldots, \lfloor \frac{d-1}{2} \rfloor$ are supplied with data via the layers closer to the originator.

Hypercube Distribution Algorithm 5 (H5)

In the H5 method PEs are activated one by one. For each of them data is supplied over d non-intersecting paths. The d paths can be built as follows. Consider a PE with i bits equal to 1 in the label at positions $a_1, a_2, a_3, \ldots, a_i$. The first path goes via PEs with label bits equal to 1 in positions: a_1, then in positions a_1, a_2, next a PE with bits equal to 1 in positions a_1, a_2, a_3; $\ldots$ $a_1, a_2, a_3, \ldots, a_i$. The second path is routed via PEs with bits equal to 1 in positions: a_2; a_2, a_3; a_2, a_3, a_4; $\ldots$ $a_2, a_3, \ldots, a_i$; $a_1, a_2, a_3, \ldots, a_i$. In an analogous way the rest of i differing paths can be routed. Suppose b_i is one of $d-i$ bits equal to 0 in the address of a target PE. The next $d-i$ paths can be routed as follows: Reach the PE with bit b_i equal to 1, then route via PEs with label bits set to 1: b_i, a_1; b_i, a_1, a_2; b_i, a_1, a_2, a_3; $\ldots$ $b_i, a_1, a_2, a_3, \ldots, a_i$. H5 method is logically equivalent to distributing in a star.

Hypercube Distribution Algorithm 6 (H6)

H6 algorithm divides a hypercube into hypercube-shaped subcubes of a smaller dimension. The originator sends data to several neighbors which in turn become originators for distributing in the subcubes. The subcubes are divided into a next level subcubes, etc. This dividing into smaller subcubes continues until activating all the PEs.

In the following we describe the methods for finding the amounts of data to be processed by particular layers. Let us denote by α_i the amount of load to be processed by one PE of layer i, by A processing rate of PEs and by C, S parameters of the communication links. For the sake of simplicity we assume that no data are returned to the originator. Hence, all the PEs must stop computing at the same moment of time. Then, the time of computing on the sending PEs must be equal to the time of data transmission to the receiving PEs plus the computing time on the receiver. The solution method for H1 has been given in [33, 34].

H2: Since the time of computing on the sender PE must be equal to the time of computing on the receiver plus the communication time, we have the following set of equations from which data distribution can be found.

$$\begin{aligned}
A\alpha_0 &= S + \alpha_1\left(\binom{d}{1}\frac{C}{d} + A\right) \\
A\alpha_1 &= S + \alpha_2\left(\binom{d}{2}\frac{C}{d} + A\right) \\
&\cdots \\
A\alpha_{i-1} &= S + \alpha_i\left(\binom{d}{i}\frac{C}{d} + A\right) \qquad (6.17) \\
&\cdots \\
A\alpha_{d-1} &= S + \alpha_d(\frac{C}{d} + A) \\
V &= \sum_{i=0}^{d}\binom{d}{i}\alpha_i
\end{aligned}$$

In the above equations term $\alpha_i \binom{d}{i}\frac{C}{d}$ stands for communication time to layer i from the originator over its d links simultaneously. Note that the first set of d links from layer 0 to layer 1 is limiting the speed of data transfer because in the following layers data is fairly distributed among the links of a PE. The number of links from layer i to layer $i+1$ is $\binom{d}{i}(d-i)$ [33] which at least equals to d. Equations (6.17) can be solved in $O(d)$ time by a method analogous to that used to solve sets of equations (6.4), (6.5). When a solution with $\alpha_i \geq 0$ ($i=0,\ldots,d$) does not exist then the maximum feasible number of layers can be found in $O(d \log d)$ time by a binary search over d .

H3: In this algorithm data distribution starts by sending data from the originator to layer d. Then distribution from the originator and layer d is symmetric because the topology of a hypercube observed from both directions is the same. Hence, we conclude that $\alpha_i = \alpha_{d-i}$ for $i = 1, \ldots, \lfloor d/2 \rfloor$. The distribution of the load can be found from the equation set:

$$\begin{aligned}
A\alpha_0 &= S + \frac{C}{d}\left(\alpha_d + d\alpha_1 + \ldots + \frac{\alpha_{\lfloor d/2 \rfloor}}{2^{d+1 \bmod 2}}\binom{d}{\lfloor d/2 \rfloor}\right) + \alpha_d A \\
A\alpha_d &= S + \alpha_1(C + A) \\
A\alpha_{i-1} &= S + \alpha_i\left(\binom{d}{i}\frac{C}{d} + A\right) \quad \text{for } i = 2, \ldots, \lfloor d/2 \rfloor - 1 \qquad (6.18) \\
A\alpha_{\lfloor d/2 \rfloor - 1} &= S + \alpha_{\lfloor d/2 \rfloor}\left(\binom{d}{\lfloor d/2 \rfloor}\frac{C}{d2^{d+1 \bmod 2}} + A\right) \\
\alpha_i &= \alpha_{d-i} \text{ for } i = 1, \ldots, \lfloor d/2 \rfloor \\
V &= \sum_{i=0}^{d}\binom{d}{i}\alpha_i
\end{aligned}$$

In the first equation of (6.18) and in the equation number $\lfloor d/2 \rfloor + 1$ term $2^{d+1 \bmod 2}$ plays any role only for the hypercubes of even dimension. In such hypercubes the central layer is fed from the originator and from the "antipodes". Since both directions are symmetric, half of data is received from the direction of the originator and half from layer d. (6.18) can be solved analogously to equation sets (6.4), (6.5). Observe that the number of data distribution steps is smaller in H3 than in H1, H2. However, this reduction is at the cost of transferring big part of data to the "antipodes" first which is a potential bottleneck.

H4: The number of data distribution steps is $\lfloor \frac{d-1}{2} \rfloor + 1$ instead of d. The communication pattern for the last activated layer is different for odd dimension d of the hypercube and for even d. For even d there are two separated layers which are activated in the last cycle: layer $\lfloor \frac{d-1}{2} \rfloor$ fed from the originator and layer $\lceil \frac{d-1}{2} \rceil$ fed from layer $d-1$. For odd d there is one layer of PEs which are activated in the last step of data distribution from layers $\frac{d-1}{2}-1$ and $\frac{d-1}{2}+1$. We will describe both cases in the sequel. Let us denote by $p=\lfloor d/2\rfloor$. When d is even α_i $(i=0,\dots,d)$ can be found from the following set of linear equations:

$$
\begin{aligned}
A\alpha_0 &= S+\frac{C}{d}\left(\alpha_d+d\alpha_{d-1}+\ldots+\binom{d}{p}\alpha_p\right)+\alpha_{d-1}A\\
A\alpha_{p-i} &= S+\alpha_{p-i+1}\left(\binom{d}{p-i+1}\frac{C}{d}+A\right)\ \text{for } i=2,\ldots,p-1\\
A\alpha_{p+i} &= S+\alpha_{p+i-1}\left(\binom{d}{p+i-1}\frac{C}{d(d-1)}+A\right)\ \text{for } i=1,\ldots,p-1\\
A\alpha_{d-1} &= S+\alpha_d(\frac{C}{d}+A)\\
A\alpha_{d-1} &= S+\alpha_1(C+A)\\
V &= \sum_{i=0}^{d}\binom{d}{i}\alpha_i
\end{aligned}
\tag{6.19}
$$

Equations (6.19) can be solved in $O(d)$ time similarly to (6.4), (6.5). α_{p+i} for $i=1,\dots,p-1$ can be expressed as a linear function of α_p, i.e. $\alpha_{p+i}=k_{p+i}\alpha_p+l_{p+i}$, where $k_{p+i}=(1+\frac{C}{d(d-1)A}(\binom{d}{p+i-1}))k_{p+i-1}$ and $l_{p+i}=\frac{S}{A}+(1+\frac{C}{d(d-1)A}(\binom{d}{p+i-1}))l_{p+i-1}$. In the same manner α_{p-i} for $i=2,\dots,p-1$ can be expressed as a linear function of α_{p-1}, i.e. $\alpha_{p-i}=k_{p-i}\alpha_{p-1}+l_{p-i}$, where $k_{p-i}=(1+\frac{C}{dA}(\binom{d}{p-i+1}))k_{p-i+1}$ and $l_{p-i}=\frac{S}{A}+(1+\frac{C}{dA}(\binom{d}{p-i+1}))l_{p-i+1}$. From the penultimate equation of (6.19) we get $\alpha_p=\frac{1}{k_{d-1}}(\frac{S}{A}+(\frac{C}{A}+1)l_1-l_{d-1}+\alpha_{p-1}(\frac{C}{A}+1)k_1)$. Using this relation we can express all α_i's as a linear function of α_{p-1} and from the last equation of (6.19) one may find the value of α_{p-1}.

When d is odd, layer p receives data both from layer $p-1$ and from layer $p+1$. Let us denote by α'_p the amount of data received by a PE of layer p from PEs of layer $p-1$ and by α''_p received from layer $p+1$. In the following discussion we assume that only when all its data already arrived can a PE start computation. The subsequent lemma establishes relation between the moments when receiving from layers $p+1$ and $p-1$ should be finished.

Lemma 6.3 *In the case of circuit-switched communication and PEs receiving data asynchronously from many identical links, the data transfers should finish simultaneously.*

Proof Let us consider some PE receiving data from two links asynchronously. In a circuit-switched communication we can change the sizes of messages sent to the considered PE without affecting the PEs activated earlier. Let us consider two cases: one link is finishing communication δ units of time earlier, both links are finishing communication simultaneously. Suppose the total time of communication to the considered PE and computing on it is T. Let C denote transfer rate, S startup, A processing rate, and α_{ij} amount of data received by P_j in case i. Thus, in the first case we get $T = A(\alpha_{11} + \alpha_{12}) + \alpha_{11}C + \delta + S$ and $T = A(\alpha_{11} + \alpha_{12}) + \alpha_{12}C + S$. Hence, $\alpha_{11} + \alpha_{12} = 2(T - S - 2\delta)(2A + C)$. In the second case we have $T = A(\alpha_{21} + \alpha_{22}) + \alpha_{21}C + S$ and $T = A(\alpha_{21} + \alpha_{22}) + \alpha_{22}C + S$. From which we obtain $\alpha_{21} + \alpha_{22} = 2(T - S)/(2A + C)$ which is bigger than $\alpha_{11} + \alpha_{12}$. We infer that in the second case more load can be processed in the given time. Consequently [33], given load is processed in the shortest time when data transfers are finished simultaneously. This reasoning can be applied also to more than two links and to links which are not identical. □

The results of Lemma 6.3 can be used to find distribution of the load as a solution to the following equations:

$$\begin{aligned}
A\alpha_0 &= S + \frac{C}{d}\left(\alpha_d + d\alpha_{d-1} + \ldots + \binom{d}{p}\alpha'_p\right) + \alpha_{d-1}A \\
A\alpha_{p-i} &= S + \alpha_{p-i+1}\left(\binom{d}{p-i+1}\frac{C}{d} + A\right) \quad \text{for } i = 2, \ldots, p-1 \\
A\alpha_{p-1} &= S + \binom{d}{p}\frac{C\alpha'_p}{d} + A(\alpha''_p + \alpha'_p) \\
A\alpha_{p+i} &= S + \alpha_{p+i-1}\left(\binom{d}{p+i-1}\frac{C}{d(d-1)} + A\right) \quad \text{for } i = 2, \ldots, p-1 \\
A\alpha_{p+1} &= S + \binom{d}{p}\frac{C\alpha''_p}{d(d-1)} + A(\alpha''_p + \alpha'_p) \\
A\alpha_{d-1} &= S + \alpha_d\left(\frac{C}{d} + A\right) \\
A\alpha_{d-1} &= S + \alpha_1(C + A)
\end{aligned} \tag{6.20}$$

$$V = \sum_{i=0}^{p-1} \binom{d}{i} \alpha_i + \binom{d}{p} (\alpha'_p + \alpha''_p) + \sum_{i=p+1}^{d} \binom{d}{i} \alpha_i$$

Equation set (6.20) can be solved in $O(d)$ time analogously to equations (6.19). Loads $\alpha_{p+i}, \alpha_{p-i}$ for $i = 1, \ldots, p-1$ can be expressed as a linear functions of α'_p and α''_p. Then, the penultimate equation can be used to find linear relation between α'_p and α''_p. The rest of the procedure is analogous.

H6: The H6 method is based on a recursive dividing a hypercube into subcubes of a smaller dimension. Unfortunately, this approach does not scale well with the dimension of the network. For example, a hypercube of dimension 6 has 4 subcubes of dimension 4, and 8 subcubes of dimension 3. Thus, one can feed from the originator 4 subcubes of dimension 4 using 4 links (out of 6) simultaneously, or feed 8 subcubes of dimension 3 but not simultaneously. In H6 we accepted the first choice, i.e. simultaneous activating of the same size subcubes. Furthermore, in the H6 algorithm it seems impossible to apply re-routing of the messages as, e.g. in H2. This is because the links that were used to feed originator of subcube e.g. Z are needed to distribute in other subcubes in the following scattering steps. Thus, these links cannot be kept occupied with the transfer to the sub-subcubes in Z via the originator of Z. Table 6.2 describes behavior of H6 along with the dimension of the hypercube. The originator first sends data to the number of PEs specified in the second column. Then, these PEs and the originator become sources for distribution in the subcubes which number is in the third column. In each of the following scattering steps all active PEs are sources of data for subcubes of decreasing dimension. The pattern of communications in the subcube of some dimension is the same as for hypercube of this dimension. Observe that in H6 PEs in the same layer may have different distance from the originator. The last column gives the numbers of PEs activated in consecutive steps of scattering. The distribution of the load can be found from the following equation set:

$$A\alpha_i = S + C(\alpha_{i+1} + \sum_{j=i+2}^{h} m_{(i+1)j}\alpha_j) + \alpha_{i+1}A \quad \text{for } i=0,\ldots,h-1 \quad (6.21)$$

$$V = \sum_{i=0}^{h} k_i \alpha_i$$

where:
$m_{ij} = \frac{k_j}{\sum_{l=0}^{i} k_l}$ - the number of PEs in layer j fed from one PE in layers $0,\ldots,i$,

Table 6.2: Numbers of PEs activated by H6

d	No. of originator's sending links	No. of 1st-step subcubes	size of 1st-step subcubes	No. of distribution steps	No. of PEs in layers $0, \ldots, d$
1	1	1	1	1	1,1
2	1	2	2	2	1,1,2
3	3	4	2	2	1,3,4
4	3	4	4	3	1,3,4,8
5	3	4	8	3	1,3,12,16
6	3	4	16	4	1,3,12,16,32
7	7	8	16	4	1,7,24,32,64
8	7	8	32	4	1,7,24,96,128
9	7	8	64	5	1,7,24,96,128,256
10	7	8	128	5	1,7,56,192,256,512
11	7	8	256	5	1,7,56,192,768,1024
12	7	8	512	6	1,7,56,192,768,1024,2048
13	7	8	1024	6	1,7,56,448,1536,2048,4096
14	7	8	2048	6	1,7,56,448,1536,6144,8192
15	15	16	2048	6	1,15,112,896,3072,12288,16384
16	15	16	4096	7	1,15,112,896,7168,12288,32768

h - the total number of layers (the 2nd column from right in Table 6.2),
k_i - the number of PEs in layer i (ith position in the rightmost column (and the proper row) of Table 6.2).

We compared our data distribution algorithms for hypercube in a series of simulations. Fig. 6.18 presents execution times for an application with $V = 10^6$ bytes, for different dimension hypercubes with $A = 1\mu s$, C=100ns/byte, $S = 0.1ms$ (typical values of Table 2.1) executed under the above data distribution algorithms. The presented values are all ratio of the actual execution time and the execution time of H1 for the same d. As it can be observed H2 is the best algorithm. For $d > 10$, however, H2 is not able to feed all the PEs before completing all the computations on smaller number of processors. On the other hand H3, H4 for small dimensions are rather cumbersome, while for bigger dimensions perform better. Furthermore, for $d \geq 12$ H3, H4 are still able to supply with data all the PEs. On the other hand, it cannot be considered as an advantage because it is a sign of longer communications. For big dimension hypercubes H4 is better than H3 because more PEs are activated earlier in H4 than in H3. Both H3 and H4 suffer from the fact that the first step of distribution consists in sending big chunk of data to the "antipodes" of the hypercube. H5 performs satisfactory for small hypercubes. For big ones the startup times of numerous consecutive

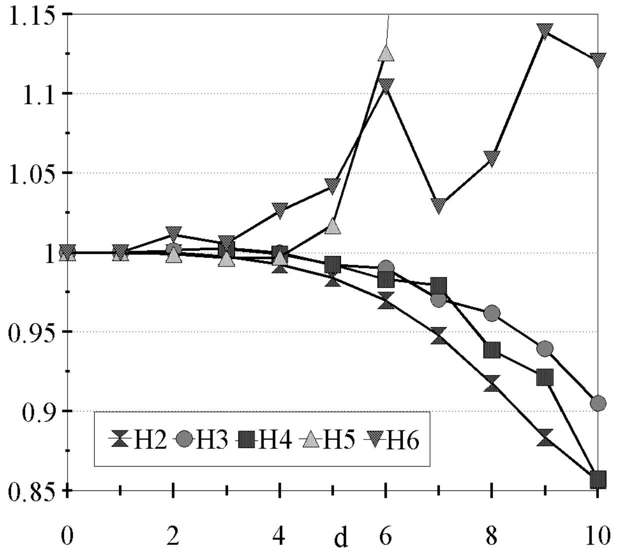

Figure 6.18: Execution times under different data distribution modes.

communications limit applicability of this method. H6 performs worse than H1,...,H4. The curve of the H6 performance is irregular because the numbers of PEs activated in a given step of scattering can change radically with d. Moreover, in Fig. 6.18 a ratio of H6 execution time to H1 execution time is presented. Performance of H6, expressed e.g. as speedup is smoother.

We conclude this section with some remark on practical aspects of applying the above algorithms. The method of finding the routes in H5 has been described before. In the remaining five algorithms it is possible to calculate the amounts of data to be shifted from a PE of one layer to a PE of the next layer over one link. This can be used to partition the load into chunks according to all the different paths reaching some PE. Such chunks, before leaving the originator should be properly grouped. For example in H1 PE of layer 1 receives data for itself and for $d-1$ PEs of layer 2. In the chunk of a layer 2 PE there is data for itself and for $d-2$ PEs of layer 3, etc. At the destination data received from different directions can be (if needed) restored to the original order. This requires also sending additional information to the network processors about the size and destination of the data chunk. Such an approach to applying H1,...,H4,H6, though possible, can be hard to implement in practice for big hypercubes, especially when one takes into account that data comes to the last PE over m different paths. Yet, in many applications it is possible to consider data chunks as "nameless". In such a case no reinstantiating of the original order of data at the destination is necessary. Hence, it is not necessary to establish at the originator the destination of each chunk of data. Only the amount of data to be intercepted in each layer is necessary. Any PE reads from the incoming data streams due amount of data and re-routes the rest in equal shares to the outgoing links. This can be implemented in far less coding.

6.3.4 3D-mesh

In this section we consider divisible job applications executed in three-dimensional meshes or tori (3DMesh in short). In the following we present five methods of recursive distribution (scattering) in such architectures and analyze their performance. The methods differ in the number p of a PE ports used simultaneously. The algorithms are based on repetitive execution of three types of moves in submeshes of decreasing size. The PEs activated in the same move of scattering will be called a layer. The three consecutive moves will be called a step. The PEs activated in the same step of scattering will be said to constitute a *basic cube*. Each step activates all the PEs of the basic cube. Then, each of the active PEs becomes a source of further distribution in the basic cube of a smaller size. The size of the basic cube decreases after each step by $1/(p+1)$. In the proposed methods each move increases the number of activated PEs p times. Note that this is the maximum possible because the number of ports used simultaneously is limited to p. Initially only the originator is active. Thus, after k distribution moves the number of working PEs is $(p+1)^k$. Accordingly, the number of PEs activated by a step of scattering (i.e. three moves) is $(p+1)^3$. In Fig. 6.19, and Fig. 6.20 the data distribution patterns are presented.

Let us describe more precisely the distribution methods. For $p=1$ and $p=2$, each move of a step activates p PEs neighboring along a different dimension. In a 3-port system the originator activates three PEs located in the same two-dimensional cross-section of a basic mesh (say, along the plane $y0z$). Next, the four active PEs send data along the third dimension (x dimension). In the last move each active processor sends data to the neighbors along the hull of basic cube and to one neighbor to the inside of a basic cube. For the 4-port system the first move sends data from the originator to four processors located along one dimension (e.g. z). Then, each PE actives other PEs located in a two-dimensional cross-section of a basic cube (along the plane $x0y$) as it was proposed in [36, 167]. In the 5-port system (cf. Fig. 6.20) the originator located at coordinates (x_0, y_0, z_0)

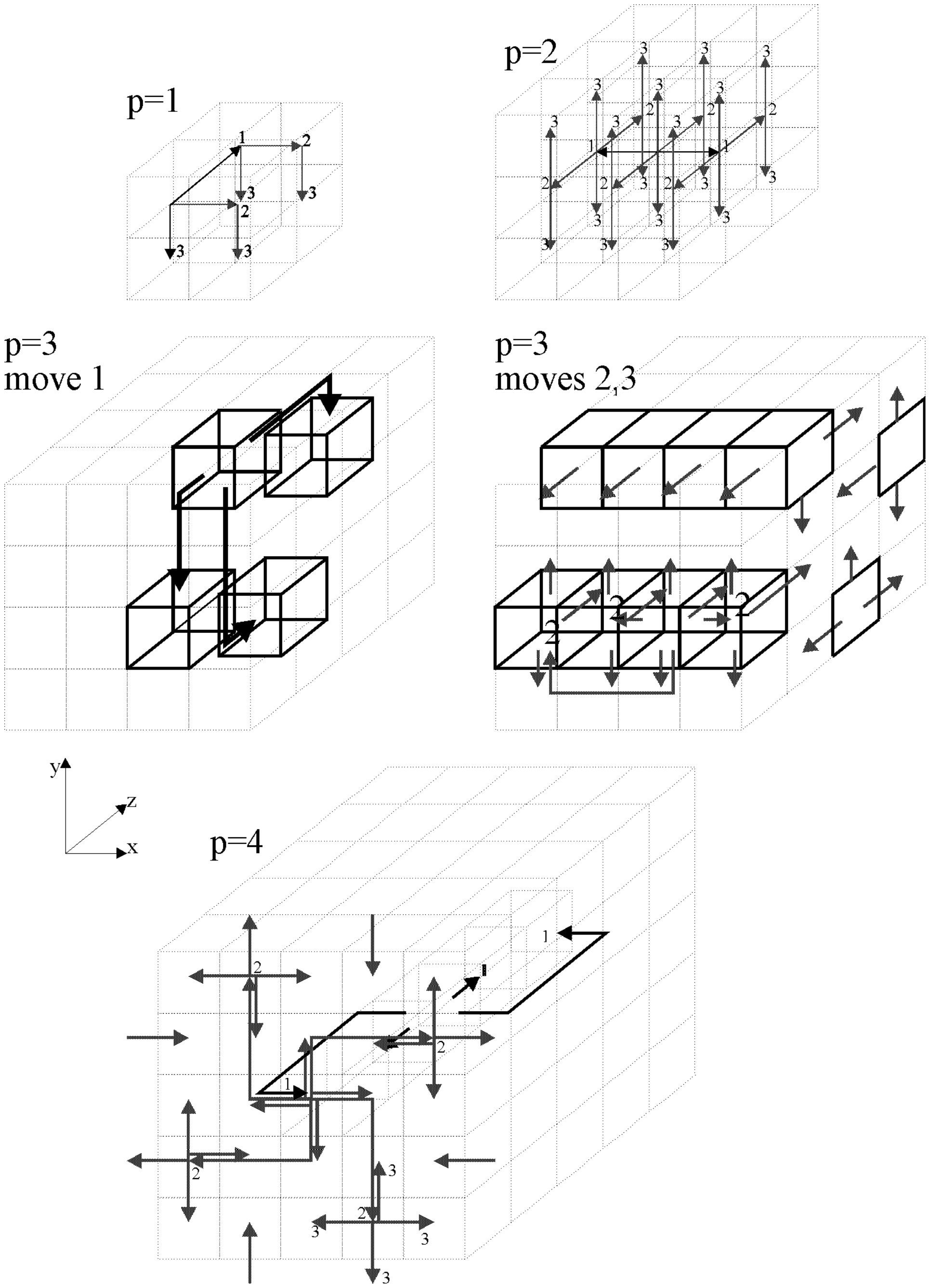

Figure 6.19: Scattering patterns in 3-dimensional meshes or tori for $p = 1, 2, 3, 4$.

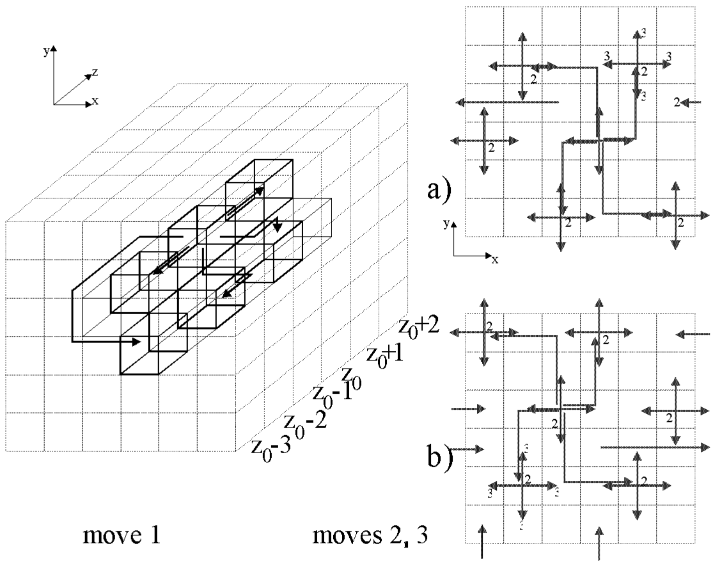

Figure 6.20: Data distribution pattern in 3-dimensional tori for 5-port PEs a) z_0-3, z_0-1, z_0+1 b) z_0-2, z_0, z_0+2.

activates PEs at coordinates (x_0+1, y_0-1, z_0-3), (x_0, y_0, z_0-2), (x_0+1, y_0-1, z_0-1), (x_0+1, y_0-1, z_0+1), (x_0, y_0, z_0+2). The communication systems of PEs at z coordinate values equal, respectively, to z_0-3, z_0-2, and z_0-1, z_0, and z_0+1, z_0+2, cooperate in pairs in the moves two and three. Each of the three pairs perform the same communications. Each PE activated in the first move sends data to five neighbors with the same value of z coordinate. For that purpose four PEs are activated using only links of the PEs in the very one two-dimensional cross-section (the same z coordinate), and one PE is activated using the links of the pairing two-dimensional cross-section. Thus, in each two-dimensional cross-section there are 5 active processors after move two. In the third move each active PE activates 4 more PEs in the same two-dimensional cross-section and one in the pairing two-dimensional cross-section. A method of scattering in 6-port toroidal 3-dimensional mesh has been proposed in [6]. In [166] a broadcasting method for d-dimensional toroidal mesh with $2d$-port PEs and edge size $(2d+1)^k$ has been proposed (where $k \in Z^+$). This method activates $(2d+1)^{kd}$ PEs in kd moves. Thus, the methods proposed in this section can be extended to toroidal meshes of arbitrary dimension when $p = 2d+1$.

Now, we evaluate our data distribution methods. As in the previous sections we assume for simplicity reasons that nothing is returned to the originator. Let us denote by α_i the amount of data to be processed by a PE

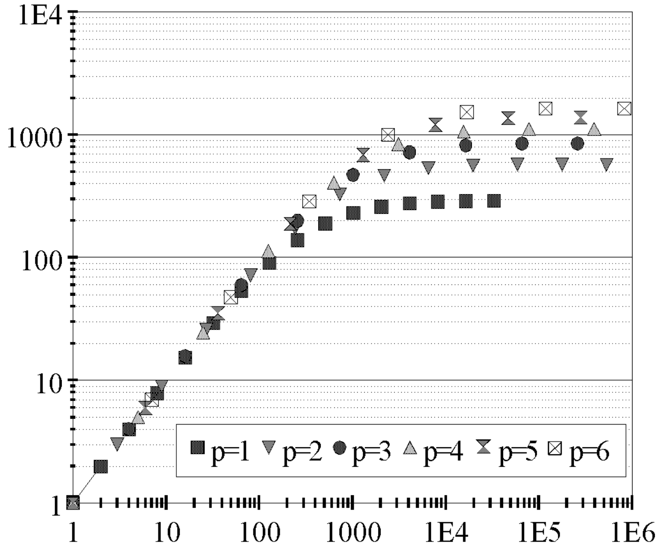

Figure 6.21: Speedup for 3-dimensional tori.

activated in the move i of scattering, and by $k = \log_{p+1} m$ the number of moves. The volumes of data can be found from the following set of equations ($p \in \{1, \dots, 6\}$):

$$\begin{aligned} A\alpha_{k-1} &= S + (A+C)\alpha_k \\ A\alpha_{k-i} &= S + C(\alpha_{k-i+1} + p\sum_{j=2}^{i} \alpha_{k-i+j}(p+1)^{i-2}) + A\alpha_{k-i+1} \\ & \quad \text{for } i = 2, \dots, k \qquad (6.22) \\ V &= \alpha_0 + p\sum_{i=1}^{k}(p+1)^{i-1}\alpha_i \end{aligned}$$

The speedup of systems with 1-port to 6-port PEs have been compared in Fig. 6.21. It was assumed that $A = 1\mu$s/byte, $C = 3.3$ns/byte, $S = 8.57\mu$s, $V = 1$Mb (the communication parameters are typical of CRAY T3D). As it can be observed the bigger the number of ports is the bigger speedup the computer system can sustain. It is because the more ports work in parallel the more data can be transferred in a unit of time, and hence the communication system introduces less overhead.

Finally, let us comment on the communication patterns proposed for the 3DMesh. Observe that the scattering methods for $p = 1$ and $p = 2$ can be extended to more dimensions than three. For d-dimensional mesh d

moves would be necessary to activate basic cube of $(p+1)^d$ PEs. Whether equivalent or more efficient methods exist for $d \geq 4$ and $p \geq 3$ is an open problem. The methods proposed are optimal in that sense that the number of activated processors in the allowed number of steps is the biggest possible. Using message pipelining may result in further reduction of data distribution and total processing times. Note that in the above scattering methods the following equation is satisfied for a basic cube: $(p+1)^k = x^d$, where p is the number of ports per a PE, k is the number of moves per step, x is the length of one edge of the basic cube, and d is the number of dimensions. It is an interesting problem which solutions of this equation lead to a feasible tessellation of moves in a basic cube. It was shown in [166] that for meshes not all solutions lead to a feasible tessellation. A similar problem is existence of moves tessellations for shapes other than cube.

6.3.5 Multistage Interconnection

In this section we apply the divisible task concept in the multistage architecture. As examples we consider multistage cube network (MCN) and IBM SP-2 high performance switch (HPS) network [1, 195].

We will analyze the multistage cube network in the form introduced in Fig. 2.2f. Note that since PEs are 1-port, the distribution of the load could be found from equations (6.22) provided that a feasible scattering method using all ports of all active PEs existed. We present such a method in the following. Let $k = \log_2 m$ denote the number of stages in the network. The method of activating PEs is the following:

Scattering in MCN
1: P_0 activates P_{m-1};
2: $z := m - 2$;
3: **for** $i := k$ **downto** 2 **do**
  begin
4: **for** $j \in \{0 \dots 2^{k-i}-1\} \cup \{m - 2^{k-i} \dots m-1\}$ **paralleldo**
    P_j activates $P_{j\,\mathbf{xor}\,z}$;
5: $z := m$ **xor** 2^{k-i+1};
  end;

Phrase **for** *index* **paralleldo** *body* demands that *body* block be performed in parallel for all values of *index*. For example, when $m = 16$ the following sequence of distributions takes place ($x \rightarrow y$ means x activates y, ';' separates actions not performed simultaneously): $0 \rightarrow 15; 0 \rightarrow 14, 15 \rightarrow 1; 0 \rightarrow 12, 1 \rightarrow$

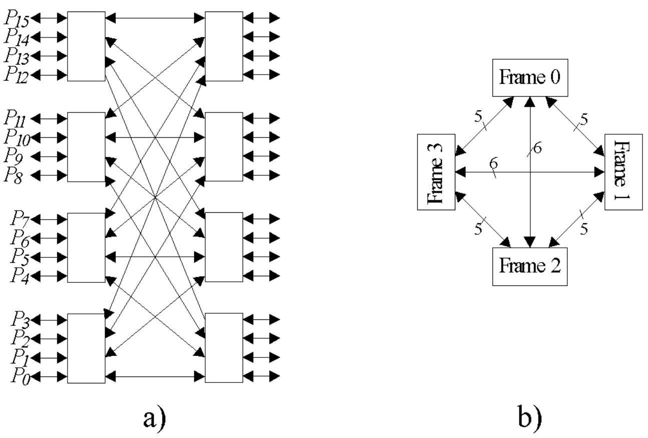

Figure 6.22: An SP-1/-2 interconnection: a) 16-PE frame, b) 64-PE/4-frame.

$13, 15 \rightarrow 3, 14 \rightarrow 2; 0 \rightarrow 8, 1 \rightarrow 9, 2 \rightarrow 10, 3 \rightarrow 11, 15 \rightarrow 7, 14 \rightarrow 6, 13 \rightarrow 5, 12 \rightarrow 4$.

Lemma 6.4 *A multistage cube network of $m = 2^k$ processors can be activated in k steps by the above algorithm of scattering in MCN.*

Proof Note that in each step of the MCN distribution the number of active PEs duplicates. Hence, we have to show only, that all the concurrent communications are contention-free. The message in the MCN is switched "swap" in ith layer switch if the source and the destination differ in their addresses in position i. Otherwise, the message goes "straight". "Straight" means that a message entering the switch at the lower input leaves the switch from the lower output, a message entering at the higher input departures from the higher output (cf. Fig. 2.2f). "Swap" is the opposite case. During the distribution each PE communicates to a PE with the binary address being the sender address xor'ed with z. All the sending PEs addresses are changed in the same bits to obtain the destination. Hence, the messages are switched in the same way in all the respective layers of multistage cube network. A contention may arise only when two messages arrive at the same switch and one message wants to go "straight", while the other needs to be "swapped". This means that the two messages are differently switched in the same layer which contradicts the earlier observations. Thus, communications are contention free. □

Analogous reasoning can be conducted for multistage interconnection of IBM SP-1/SP-2 computers which are using bi-directional switches. The

data distribution pattern differs slightly in this case. 16 PEs interconnected by HPS constitute a frame presented in Fig. 6.22a. Frames can be linked with each other in a variety of ways [195] including e.g. fat-tree networks, SW-Banyan. There are at least 4 usable paths between each pair of nodes, except for node pairs that are directly attached to the same HPS (4-PE groups $P_0 ... P_3$; $P_4 ... P_7$ etc.) In this section we assume that each PE has a single port attachment to the HPS ($p = 1$). Hence, each PE can activate only one additional PE in a distribution step. This pattern can be applied in a multi-frame SP-1/-2 releases. Assuming that f is the number of frames and that each frame can be connected with any other frame by at least one contention-free route, all the frames can be activated in $\log_2 f$ steps. Then, PEs of a frame can be activated starting from a single PE as in the following description. The distribution for a single frame can be 4-step (cf. Fig. 6.22):
1:$P_0 \rightarrow P_8$;
2:$P_0 \rightarrow P_4$, $P_8 \rightarrow P_{12}$;
3:$P_0 \rightarrow P_2$, $P_4 \rightarrow P_6$, $P_8 \rightarrow P_{10}$, $P_{12} \rightarrow P_{14}$;
4:$P_0 \rightarrow P_1$,$P_2 \rightarrow P_3$, $P_4 \rightarrow P_5$, $P_6 \rightarrow P_7$, $P_8 \rightarrow P_9$,$P_{10} \rightarrow P_{11}$, $P_{12} \rightarrow P_{13}$,$P_{14} \rightarrow P_{15}$.

In a multi-frame machine (Fig. 6.22b) the above four steps could be preceded by two more steps:
1:Frame0P_0 :→Frame1P_0;
2:Frame0P_0 :→Frame2P_0, Frame1P_0 :→Frame3P_0.

Then, the distribution of the load can be found analogously to a multistage cube network case using equations (6.22) for $p = 1$.

6.4 Discussion and Conclusions

In this section we comment on the granularity of data, summarize this chapter as well as discuss some possible further extensions of divisible task concept. The results of this chapter are collected in Table 6.3. Column "Result" shows (among the others) computational complexity of the algorithm finding distribution of the load.

Let us observe that the assumption on infinite divisibility can be hard to justify in practice. In real applications data usually has some unit of granularity e.g. record in a database file, a floating point number, etc. It is still possible to calculate bounds on performance of computer systems or to use the above described methods as good approximations. The data distribution calculated from solutions of equations e.g. (6.1), (6.3), (6.4), (6.16) can be rounded up to the nearest unit of granularity. This results in a schedule longer than calculated. Yet, the increase of the schedule length can be bounded from above. For the case of not returning results this bound is equal to $\delta(A + kC)$, where δ is the maximal increase of the load for a PE (equal to the granularity unit), A - is the processing rate of the target PE, C - is the communication rate (homogeneous communication links assumed), k - is the biggest number of times the data chunks including the load for the target PE are transferred from one PE to another PE. k is a constant depending on the scattering method and the communication network. Thus, the deviation from the expectation is bounded from above. When returning the results is considered, parameter k should include also the biggest number of data transfer operations during returning the results.

The analysis performed here usually included two steps: devising a scattering algorithm and solving a set of linear equations. The scattering algorithm included and hid the underlying hardware/software details (i.e. architecture). The sets of linear equations include two types of equations: firstly, equations linking processing time and communication time of the sender and the receiver PEs or of the consecutively activated PEs, and secondly, an equation expressing that all the load must be processed. Although this method has been applied to analyze quite wide range of computer architectures still many questions remain open. One of them is the complexity of problem $Q, star \mid div, n = 1 \mid C_{max}$ or $Q, bus \mid div, n = 1 \mid C_{max}$ when startup times are non-zero. Similar question can be raised for tree networks of processors.

In this publication and in the previous works on divisible task theory it was assumed that the computation time depends linearly on the volume of processed data. But, for example, distributed sorting has nonlinear dependence of the processing time on the size of processed data. This can be included in our equations (e.g. (6.1), (6.4), (6.16), etc.) as a nonlinear processing time function depending on the amount of processed data. In such a case, however, the sets of equations would be by far harder to solve. This problem can be even more difficult than solving a set of nonlinear equations. Consider, for example, distributed multiplication of two matrices. To produce one entry in the resulting matrix a column from one input matrix and a row from another must be known. A possible approach here is to send a column and a row to a PE each time a final entry is calculated. Then, the divisible task concept can be used as presented in the earlier sections. Yet, it is not difficult to observe that this method is not optimal because many columns and rows would be sent several times. The minimum number of communications is achieved when the PEs compute square submatrix of the product matrix. It is because square is a rectangle with minimal length of edges for the given area. Hence, to find an optimal distribution of the load it may be necessary to find partitioning of the product matrix into squares, possibly of different sizes. Note that this is a cutting problem - one of the hardest computational problems.

In the scattering algorithms presented in this chapter PEs received data from one link only. An interesting direction of further research can be considering the case when PEs receive data via several non-identical paths. As far as scattering algorithms are considered, perplexing problem is the question of their optimality. To our knowledge there are no general methods of proving time optimality of the scattering algorithms. To demonstrate superiority of the scattering algorithm direct comparison is applied. Hence, the description of the methods proposed here ended in performance evaluation.

Another interesting issue for further research is including in the model a limited sizes of buffers on the communication paths and at the destination PEs, applying divisible task concept in dynamic distributed load balancing, or considering multiple applications (instead of one) issued by multiple originators.

Observe that divisible task approach can be also applied to analyze the production-transportation systems. In such a system the transportation system is an equivalent of the computer interconnection network, while production facilities are equivalent to processors.

Finally, let us note that divisible task and multiprocessor task concepts can be used together at different levels of scheduling. For example, multiprocessor task scheduling can be applied to assign processors or partitions to applications, while divisible task scheduling can be applied to find the best computation distribution for the application.

Table 6.3: Scheduling divisible tasks

Problem	Result, remarks	Reference
Q,$chain$,$s\&f$,$no-overlap$ $\mid div$,$n=1\mid C_{max}$	$O(m)$	[15, 60, 100] [154, 176]
Q,$chain$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	$O(m)$	[15, 60] [100, 154]
Q,$tree$,$s\&f$,$no-overlap$ $\mid div$,$n=1\mid C_{max}$	$O(m)$	[12, 15, 61]
Q,$tree$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	$O(m)$	[12, 15, 61]
Q,bus, $no-overlap$ $\mid div$,$n=1\mid C_{max}$	$O(m)$	[14]
Q,bus $\mid div$,$n=1\mid C_{max}$	$O(m)$	[12, 13]
Q,$star$,$s\&f$,$no-overlap$ $\mid div$,$n=1\mid C_{max}$	$O(m)$	[16, 100]
Q,$star$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	$O(m)$	[16, 100]
R,bus, $no-overlap$ $\mid div \mid C_{max}$	$O(m)$,FIFO	[189]
R,bus $\mid div \mid C_{max}$	$O(m)$,FIFO	[189]
P,$hypercube$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	$O(\log m)$	[33, 34]
Q,$chain$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	$O(m\log m)$, $S\neq 0$	[33]
Q,$conn$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	**NP**h,$S\neq 0$	[33]
Q,bus $\mid div$,$n=1\mid C_{max}$	**NP**h,$S\neq 0$	[33]
Q,$star$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	polynomial cases,$S\neq 0$	[33]
Q,$hypercube$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	$O(\log m \log\log m)$, $S\neq 0$	[33]
P,$2D-mesh$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	performance bounds	[35]
P,$2D-mesh$, csw $\mid div$,$n=1\mid C_{max}$	$O(\log m \log\log m)$	[36]
Q,$star$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	pipelining	[17]
Q,$chain$, $1-port$, $s\&f$ $\mid div$,$n=1\mid C_{max}$	$O(m)$	[18]
Q,bus,$\mid div$,$n=1\mid X$	two criteria	[191]
P,$chain$, csw $\mid div$,$n=1\mid C_{max}$	performance bounds	Sec. 6.3.1
Q,$star$ $\mid div$, $n=1\mid C_{max}$	MILP	Sec. 6.3.2
Q,$star$, win^1 $\mid div$, $n=1\mid C_{max}$	polynomial,LP	Sec. 6.3.2
P,$hypercube$, csw $\mid div$,$n=1\mid C_{max}$	$O(\log m \log\log m)$,$S\neq 0$	Sec. 6.3.3
P,$3D-mesh$, csw $\mid div$,$n=1\mid C_{max}$	$O(\log m \log\log m)$,$S\neq 0$	Sec. 6.3.4
P,$multistage$, csw $\mid div$,$n=1\mid C_{max}$	$O(\log m \log\log m)$,$S\neq 0$	Sec. 6.3.5

1) Processors become continuously available after certain moments of time.

Chapter 7

Conclusions

In this work we considered selected methods of scheduling in multiprocessor computer systems. With the advent of modern computer systems it turned out that classical scheduling methods in many cases are not satisfactory (cf. Sections 4.6, 5.1). Therefore, two new scheduling models were analyzed here: multiprocessor tasks and divisible tasks.

Multiprocessor tasks require several processors simultaneously, thus allow for expressing task parallelism at high level of abstraction. This results in formulation of more tractable scheduling problems which in classical form are computationally hard. Divisible task model assumes that work is infinitely divisible and parallelizable. This model allows for finding simple solutions of problems which in other setting are again intractable. Moreover, divisible task concept permits introducing computer architecture context which in the classical approach is often highly generalized to make problems manageable. In this way scheduling problems have been combined with communication optimization problems. Thus, a method has been proposed which links these two research areas.

The results of this work are collected, together with the previous publications on the subject, in tables: 5.1,5.2,6.3. To sum up the results of this publication, the most important in our opinion are:

- Formulation of algorithms for scheduling preemptive multiprocessor tasks with linear speedup on parallel processors.
- Formulation of low-order complexity algorithms for scheduling preemptive multiprocessor tasks on dedicated processors with L_{max} criterion.
- Formulation of polynomial-time algorithms for scheduling preemptive

multiprocessor tasks on dedicated processors in time windows.

- Formulation of divisible job scheduling algorithms for a variety of computer architectures (including 3-dimensional meshes, hypercubes, multistage interconnections)

This work has not exhausted the resource of scheduling problems in multiprocessor systems. Further research can include for example:

- Scheduling multiprocessor tasks on parallel processors available in time windows.
- Scheduling multiprocessor tasks with linear speedup for mean completion time criterion.
- Scheduling divisible tasks on other architectures.
- Scheduling multiple divisible tasks in a system with many originators.
- Further practical verifying divisible task concept for other applications and other architectures.

Streszczenie w języku polskim

Nowoczesne systemy komputerowe są systemami wieloprocesorowymi. Ich efektywność zależy od metod zarządzania wykonywanymi pracami. Szybkie wykonanie równoległych aplikacji jest możliwe jedynie wtedy, gdy poszczególne jej elementy są odpowiednio uporządkowane w czasie i przestrzeni. Stąd wynika znaczenie poprawnego szeregowania zadań w wieloprocesorowych systemach komputerowych.

W niniejszej rozprawie rozważne są zagadnienia deterministycznego szeregowania zadań. Klasyczna teoria szeregowania zadań zakłada, że zadanie w jednej chwili czasu wymaga dokładnie jednego procesora. W ostatnich latach założenie to jest podważane, zwłaszcza w kontekście aplikacji dla równoległych i rozproszonych systemów komputerowych. Praca poświęcona jest zagadnieniom szeregowania tego typu aplikacji zwanych dalej zadaniami wieloprocesorowymi. Przedstawiono analizę złożoności obliczeniowej otwartych problemów szeregowania zadań wieloprocesorowych. Zaprezentowano algorytmy szeregowania zadań podzielnych zachowujących liniowe przyspieszenie na procesorach równoległych. Analizie poddano problemy podzielnego szeregowania zadań wieloprocesorowych na procesorach dedykowanych z kryterium maksymalnego opóźnienia.

Wiele aplikacji równoległych ma tak regularną strukturę, że możliwy jest podział obliczeń na części o dowolnych rozmiarach i wykonywanie ich na niezależnych procesorach. Tego typu aplikacje nazwiemy zadaniami jednorodnymi. Dla zadań jednorodnych zaprezentowano metodę znajdowania optymalnego rozdziału obliczeń w rozproszonym systemie wieloprocesorowym. Koncepcja zadań jednorodnych umożliwia prostą analizę obszernej klasy systemów komputerowych (m.in. architektur takich jak 3-wymiarowa krata, hiperkostka, magistrala itd.), uwzględniającą wiele szczegółowych aspektów obliczeń równoległych. Metoda ta pozwala także na ocenę efektywności wieloprocesorowych systemów komputerowych; przedstawiono przykłady takiej oceny. Koncepcja zadania jednorodnego umożliwia uwzględnienie wielu aspektów komunikacyjnych, stanowi tym samym pomost pomiędzy teorią szeregowania zadań a teorią optymalizacji komunikacji w sieciach komputerowych.

Praca zawiera wyniki przedstawione na tle aktualnego stanu badań w rozpatrywanej dziedzinie. Uzyskane rezultaty, przedstawione w ujednoliconej formie, wskazują dalsze kierunki badawcze.

Bibliography

[1] T. Agerwala, J.L. Martin, J.H. Mirza, D.C. Sadler, D.M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2), 1994. http://www-i.almaden.ibm.com/journals/sj/agerw/agerw.html.

[2] M. Ahuja and Y. Zhu. An $O(n \log n)$ feasibility algorithm for preemptive scheduling of n independent jobs on a hypercube. *Information Processing Letters*, 35:7–11, June 1990.

[3] W. Alda, W. Dzwinel, J. Kitowski, J. Mościński, and D.A.Yuen. Penetration mechanics via molecular dynamics. Research Report UMSI 93/58, University of Minessota Supercomputing Institute, April 1993.

[4] D. Alpert and D. Avnon. Architecture of the Pentium microprocessor. *IEEE Micro*, 13(3):11–21, June 1993.

[5] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings (Atlantic City Apr.18-20, 1967)*, volume 30, pages 483–485. AFIPS, April 1967.

[6] T. Armitage and J.G. Peters. Circuit-switched broadcasting in 3-dimensional toroidal meshes. manuscript, 1995.

[7] M.J. Atallah, C.L. Black, D.C. Marinescu, H.J. Siegel, and T.L. Casavant. Models and algorithms for coscheduling compute-intensive tasks on a network of workstations. *Journal of Parallel and Distributed Computing*, 16:319–327, 1992.

[8] A. Avizienis, G.C. Gilley, F.P. Mathur, D.A. Rennels, J.A. Rohr, and D.K. Rubin. The star (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. *IEEE Transactions on Computers*, 20(11):1312–1321, 1971.

[9] R.G. Babb II, editor. *Programming parallel processors*. Addison - Wesley, 1988.

[10] K.R. Baker. *Introduction to sequencing and scheduling*. John Wiley & Sons, 1974.

[11] E. Bampis, J.C. Konig, and D. Trystram. Optimal parallel execution of complete binary trees and grids into most popular interconnection networks. Rapport APACHE 5, Institut IMAG, December 1993.

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

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

[14] S. Bataineh and T.G. Robertazzi. Distributed computation for a bus network with communication delays. In *Proceedings of the 25-th Conference on Information Sciences and Systems, The John Hopkins University, Baltimore*, pages 709–714, March 1991.

[15] S. Bataineh and T.G. Robertazzi. Ultimate performance limits for networks of load sharing processors. CEAS Technical Report 623, State University of New York at Stony Brook, April 1992.

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

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

[18] V. Bharadwaj, D. Ghose, and V.Mani. An efficient load distribution strategy for a distributed linear network of processors with communication delays. *Computers Math. Applic.*, 29(9):95–112, 1995.

[19] L. Bianco, J. Błażewicz, P. Dell'Olmo, and M. Drozdowski. Preemptive scheduling of multiprocessor tasks on the dedicated processors system subject to minimal lateness. *Information Processing Letters*, 46:109–113, 1993.

[20] L. Bianco, J. Błażewicz, P. Dell'Olmo, and M. Drozdowski. Preemptive multiprocessor task scheduling with release times and time windows. Technical Report RA-94/013, Institute of Computing Science, Poznań University of Technology, 1994.

[21] L. Bianco, J. Błażewicz, P. Dell'Olmo, and M. Drozdowski. Scheduling preemptive multiprocessor tasks on dedicated processors. *Performance Evaluation*, 20:361–371, 1994.

[22] L. Bianco, J. Błażewicz, P. Dell'Olmo, and M. Drozdowski. Scheduling UET multiprocessor tasks. *Foundations of Computing and Decision Sciences*, 19(4):273–283, 1994.

[23] L. Bianco, J. Błażewicz, P. Dell'Olmo, and M. Drozdowski. Scheduling multiprocessor tasks on a dynamic configuration of dedicated processors. *Annals of Operations Research*, 58:493–517, 1995.

[24] L. Bianco, J. Błażewicz, P. Dell'Olmo, and M. Drozdowski. Linear algorithms for preemptive scheduling of multiprocessor tasks subject to minimal lateness. *Discrete Applied Mathematics*, 72(1-2):25-46, January 1997.

[25] L. Bianco, P. Dell'Olmo, and M.G. Speranza. Nonpreemptive scheduling of independent tasks with prespecified processor allocations. *Naval Research Logistics Quarterly*, 41:959–971, 1994.

[26] L. Bianco, P. Dell'Olmo, and M.G. Speranza. Scheduling independent tasks with multiple modes. *Discrete Applied Mathematics*, 62:35–50, 1995.

[27] J.-Y. Blanc and D. Trystram. Implementation of parallel numerical routines using broadcast communication schemes. In E. Burkhart, editor, *Lecture Notes in Computer Science 457, CONPAR 90-VAPP IV, Joint International Conference on Vector and Parallel Processing, Proceedings, Zurich, Switzerland, September 10-13, 1990*, pages 467–478, Berlin, September 1990. Springer-Verlag.

[28] J. Błażewicz. *Złożoność obliczeniowa problemów kombinatorycznych.* WNT, Warszawa, 1988.

[29] J. Błażewicz, P.Bouvry, F.Guinand, and D.Trystram. Scheduling complete in-trees on two uniform processors with communication delays. *Information Processing Letters*, 58:255–263, 1996.

[30] J. Błażewicz, W. Cellary, R. Słowiński, and J. Węglarz. *Badania operacyjne dla informatyków.* WNT, Warszawa, 1983.

[31] J. Błażewicz, P. Dell'Olmo, M. Drozdowski, and M.G. Speranza. Scheduling multiprocessor tasks on three dedicated processors. *Information Processing Letters*, 41:275–280, April 1992. Corrigendum: IPL 49, 1994, 269-270.

[32] J. Błażewicz, M. Drabowski, and J. Węglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers*, 35(5):389–393, May 1986.

[33] J. Błażewicz and M. Drozdowski. Scheduling divisible jobs with communication startup costs. *Discrete Applied Mathematics*, 1997, to appear.

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

[35] J. Błażewicz and M. Drozdowski. Performance limits of two-dimensional network of load-sharing processors. *Foundations of Computing and Decision Sciences*, 21(1):3–15, 1996.

[36] J. Błażewicz, M. Drozdowski, F. Guinand, and D. Trystram. Scheduling under architectural constraints. Technical Report RA-003/95, Institute of Computing Science, Poznań University of Technology, 1995.

[37] J. Błażewicz, M. Drozdowski, G. Schmidt, and D.de Werra. Scheduling independent two processor tasks on a uniform duo-processor system. *Discrete Applied Mathematics*, 28:11–20, 1990.

[38] J. Błażewicz, M. Drozdowski, G. Schmidt, and D.de Werra. Scheduling independent multiprocessor tasks on a uniform k-processor system. Technical Report R92/030, Institute of Computing Science, Poznań University of Technology, 1992.

[39] J. Błażewicz, M. Drozdowski, G. Schmidt, and D.de Werra. Scheduling independent multiprocessor tasks on a uniform k-processor system. *Parallel Computing*, 20:15–28, 1994.

[40] J. Błażewicz, M. Drozdowski, D.de Werra, and J. Węglarz. Scheduling independent multiprocessor tasks before deadlines. *Discrete Applied Mathematics*, 65(1-3):81–96, March 1996.

[41] J. Błażewicz and K. Ecker. Scheduling multiprocessor tasks under unit resource constraints. In *Proceedings of International Conference on Optimization Techniques and Applications, Singapore*, pages 161–169, April 1987.

[42] J. Błażewicz and K. Ecker. Scheduling in computer and manufacturing systems. Technical Report 114, Dagstuhl Seminar Report, 1995.

[43] J. Błażewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Węglarz. *Scheduling Computer and Manufacturing Processes*. Springer Verlag, Heidelberg, New York, 1996.

[44] J. Błażewicz, J.K. Lenstra, and A.H.G. Rinnoy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.

[45] J. Błażewicz and Z. Liu. Scheduling multiprocessor tasks with chain constraints. *European Journal of Operational Research*, 94:231–241, 1996.

[46] J. Błażewicz, J. Węglarz, and M. Drabowski. Scheduling independent 2-processor tasks to minimize schedule length. *Information Processing Letters*, 18:267–273, 1984.

[47] T. Bönniger, R. Esser, and D. Krekel. CM-5, KSR2, Paragon XP/S: A comparative description of massively parallel computers. *Parallel Computing*, 21:199–232, 1995.

[48] G.Bozoki and J.-P. Richard. A branch-and-bound algorithm for the continuous-process job-shop scheduling problem. *AIIE Transactions*, 2(3):246–252, September 1970.

[49] P. Brucker and A.Krämer. Polynomial algorithms for resource constrained and multiprocessor task scheduling problems with a fixed number of task types. Osnabrücker Schriften zur Mathematik, Reihe P Preprints Heft 165, Fachbereich Mathematik/Informatik, Universität Osnabrück, May 1994.

[50] P. Brucker and A.Krämer. Shop scheduling problems with multiprocessor tasks and dedicated processors. *Annals of Operations Research: Mathematics of Industrial Systems I*, 57:13–27, 1995.

[51] X. Cai, C.-Y. Lee, and C.-L. Li. Minimizing total flow time in multiprocessor tasks systems with prespecified allocations, private communication, March 1996.

[52] Ch. Calvin. *Optimisation du surcoût des communications dans la parallélisation des algorithmes numériques.* PhD thesis, INPG Grenoble France, 1995.

[53] W.J. Camp, S.J. Plimpton, B.A. Hendrickson, and R.W. Leland. Massively parallel methods for engineering and science problems. *Communications of the ACM*, 37(4):31–40, April 1994.

[54] N.J. Carriero, D. Gelertner, T.G. Mattson, and A.H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20:633–655, 1994.

[55] T.L. Casavant. Architectures for massively parallel computers. In *ISIP-CALA'93 Drafts of Papers*, pages 1–49, 1993.

[56] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[57] V. Chaudhary and J.K. Aggarwal. A generalized scheme for mapping parallel algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):328–346, 1993.

[58] G.-I. Chen and T.-H. Lai. Preemptive scheduling of independent jobs on a hypercube. *Information Processing Letters*, 28:201–206, July 1988.

[59] G.-I. Chen and T.-H. Lai. Scheduling independent jobs on partitionable hypercubes. *Journal of Parallel and Distributed Computing*, 12:74–78, 1991.

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

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

[62] M.-S. Chern, G.H. Chen, and P. Liu. An LC branch-and-bound algorithm for the module assignment problem. *Information Processing Letters*, 32:61–71, 1989.

[63] A.N. Choundhary, B. Narahari, D.M. Nicol, and R. Simha. Optimal processor assignment for a class of pipelined computations. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):439–445, April 1994.

[64] P. Chrétienne. A polynomial algorithm to optimally schedule tasks on a virtual distributed system under tree-like precedence constraints. *European Journal of Operational Research*, 43:225–230, 1989.

[65] P. Chrétienne. Task scheduling with interprocessor communication delays. *European Journal of Operational Research*, 57:348–354, 1992.

[66] P. Chrétienne. Tree scheduling with communication delays. *Discrete Applied Mathematics*, 49:129–141, 1994.

[67] P. Chrétienne and C. Picoleau. Scheduling with communication delays: A survey. In P. Chretienne, E.G. Coffman Jr., J.K. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications.* J. Wiley, 1995.

[68] E.G. Coffman Jr. (editor). *Computer and Job-Shop Scheduling Theory.* Wiley & Sons, New York, 1976.

[69] E.G. Coffman Jr. and P.J. Denning. *Operating Systems Theory.* Prentice-Hall, Englewood Cliffs, 1973.

[70] E.G. Coffman Jr., M.R. Garey, D.S. Johnson, and A.S. LaPaugh. Scheduling file transfers. *SIAM Journal on Computing*, 3(14):744–780, August 1985.

[71] E.G. Coffman Jr. and R.J. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

[72] J.Y. Colin and P. Chretienne. C.p.m scheduling with small communication delays and task duplication. *Operation Research*, 39(4):680–684, July 1991.

[73] D.H. Cornett and M.A. Franklin. Scheduling independent tasks with communications. In *Proceedings of 17th Allerton Confernce*, pages 624–633, 1979.

[74] G.L. Craig, C.R. Kime, and K.K. Saluja. Test scheduling and control for vlsi built-in self-test. *IEEE Transactions on Computers*, 37(9):1099–1109, September 1988.

[75] Cray Research Inc. *Cray T3D, Technical summary*, 1993.

[76] A.L. Decegama. *The Technology of Parallel Processing. Parallel Processing Architectures and VLSI Hardware Volume I.* Prentice-Hall Inc., Englewood Cliffs, 1989.

[77] P. Dell'Olmo and M.G. Speranza. Graph models for multiprocessor scheduling with precedence constraints. *Foundations of Computing and Decision Sciences*, 21(1):17–30, 1996.

[78] P. Dell'Olmo, M.G. Speranza, and Z. Tuza. Easy and hard cases of a scheduling problem on three dedicated processors, private communication, 1993.

[79] G. Dobson and U.S. Karmarkar. Simultaneous resource scheduling to minimize weighted flow times. *Operations Research*, 37(4):592–600, July 1989.

[80] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–785, August 1991.

[81] M. Drabowski. *Szeregowanie zadań w systemach wielomikroprocesorowych.* PhD thesis, Poznań University of Technology, 1985.

[82] M.A. Driscoll and W.R. Daasch. Accurate predictions of parallel program execution time. *Journal of Parallel and Distributed Computing*, 25:16–30, 1995.

[83] M. Drozdowski. *Problemy i algorytmy szeregowania zadań wieloprocesorowych.* PhD thesis, Poznań University of Technology, 1992.

[84] M. Drozdowski. Scheduling multiprocessor tasks on hypercubes. *Bulletin of the Polish Academy of Sciences, Technical Sciences*, 42(3):437–445, 1994.

[85] M. Drozdowski. On complexity of multiprocessor tasks scheduling. *Bulletin of the Polish Academy of Sciences, Technical Sciences*, 43(3):381–392, 1995.

[86] M. Drozdowski. Real-time scheduling of linear speedup parallel tasks. *Information Processing Letters*, 57:35–40, 1996.

[87] M. Drozdowski. Scheduling multiprocessor tasks - an overview. *European Journal of Operational Research*, 94:215–230, 1996.

[88] M. Drozdowski and Z. Kluczyński. Algorytmy szeregowania zadań jednorodnych w systemach rozproszonych. Technical Report RB-96/001, Institute of Computing Science, Poznań University of Technology, 1996.

[89] M. Drozdowski and W. Kubiak. Scheduling parallel tasks with sequential heads and tails. Technical report, Faculty of Business Administration, Memorial University of Newfoundland, 1995.

[90] J. Du and J.Y-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4):472–478, November 1989.

[91] K. Efe and V. Krishnamoorthy. Optimal scheduling of compute-intensive tasks on a network of workstations. *IEEE Transactions on Parallel and Distributed Systems*, 6(6):668–673, 1995.

[92] J.A. Ellis. Embedding rectangular grids into square grids. *IEEE Transactions on Computers*, 40(1):46–52, 1991.

[93] D.G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[94] L. Finta and Z. Liu. Scheduling of parallel programs in single-bus multiprocessor systems. Technical Report 2302, INRIA, Centre Sophia Antipolis, May 1994.

[95] M.J. Fisher and A.R. Mayer. Boolean matrix multiplication and transitive closure. In *Twelfth Ann. Symposium on Switching Automata Theory, East Lansing, Mich.*, pages 129–131, 1971.

[96] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[97] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[98] E.F. Gehringer, D.P. Siewiorek, and Z. Segall. *Parallel Processing: The Cm* Experience*. Digital Press, Bedford, 1987.

[99] D. Ghosal, G. Serazzi, and S. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

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

[101] W.K. Giloi. Parallel supercomputer architectures and their programming models. *Parallel Computing*, 20:1443–1470, 1994.

[102] M.X. Goemans. An approximation algorithm for scheduling on three dedicated processors. *Discrete Applied Mathematics*, 61:49–59, 1995.

[103] T. Gonzalez and S. Sahni. Preemptive scheduling of uniform processor systems. *Journal of the ACM*, 25:92–101, 1978.

[104] A. Goscinski. *Distributed Operating Systems*. Addison-Wesley Publishing Co., Sydney, 1991.

[105] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[106] A.Y. Grama, V. Kumar, and V.N. Rao. Experimental evaluation of load balancing techniques for hypercube. In D.J. Evans, G.R. Joubert, and H. Liddell, editors, *Parallel Computing '91*, pages 497–514. Elsevier Science Publishers B.V., 1992.

[107] A.S. Grimshaw, J.B. Weissman, E.A. West, and E.C. Loyot Jr., Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21:257–270, 1994.

[108] F. Guinand. *Ordonnancement avec communications pour architectures multiprocesseurs dans divers modelés d'exécution.* PhD thesis, LMC-IMAG, Institut National Polytechnique de Grenoble, June 1995.

[109] F. Guinand and D. Trystram. Optimal scheduling of UECT trees on two processors. Apache 3, Institut IMAG, November 1993.

[110] A.K. Gupta and S.E. Hambrusch. Embedding complete binary trees into butterfly networks. *IEEE Transactions on Computers*, 40(7):853–863, July 1991.

[111] J.R. Gurd, C.C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[112] J.L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[113] S. L. Hakimi and A. T. Amin. Characterization of connection assignment of diagnosable systems. *IEEE Transactions on Computers*, 23(1):86–89, 1974.

[114] W.A. Halang and K.M. Sacha. *Real-Time Systems, Implementation of Industrial Computerised Process Automation.* World Scientific Publishing, London, 1992.

[115] T.R. Halfhill. Intel's P6. *Byte*, 20(4):42–58, 1995.

[116] S.M. Hedetniemi, S.T. Hedetniemi, and A.L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.

[117] R.W. Hockney. Performance parameters and benchmarking of supercomputers. In J.J. Dongarra and W. Gentzsch, editors, *Computer Benchmarks*, pages 41–63. Elsevier Science Bv., 1993.

[118] R.W. Hockney. The communication challenge for mpp: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20:389–398, 1994.

[119] C.P.M. van Hoesel. Preemptive scheduling on a hypercube. Technical Report 8963/A, Erasmus University, P.O.Box 1738-3000 Rotterdam, The Netherlands, March 1989.

[120] J.A. Hoogeveen, S.L. van de Velde, and B. Veltman. Complexity of scheduling multiprocessor tasks with prespecified processor allocations. *Discrete Applied Mathematics*, 55:259–272, 1994.

[121] A.L. Hopkins, J.M. Lala, and T.B. Smith. FTMP - a highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10), 1978.

[122] G. Horton. A multi-level diffusion method for dynamic load balancing. *Parallel Computing*, 19:209–218, 1993.

[123] G.D. Hutchenson and J.D. Hutchenson. Technologia i koszty w przemyśle półprzewodnikowym. *Świat Nauki*, (3):32–38, March 1996. (Polish edition of *Scientific American*, January 1996).

[124] J.-J. Hwang, Y.-C. Chow, F.D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[125] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programability*. McGraw-Hill, New York, 1993.

[126] R. Jain, K. Somalwar, J. Werth, and J. C. Browne. Scheduling parallel i/o operations in multiple bus systems. *Journall of Parallel and Distributed Computing*, 16:352–362, 1992.

[127] A. Jakoby and R. Reischuk. The complexity of scheduling problems with communication delays for trees. In O. Nurmi and E. Ukkonen, editors, *Lecture Notes in Computer Science 621. Algorithm Theory, SWAT92*, pages 165–177. Springer Verlag, 1992.

[128] J.Liu, V.A.Saletore, and T.G.Lewis. Safe Self-Scheduling: A parallel loop scheduling scheme for shared-memory multiprocessors. *International Journal of Parallel Programming*, 22(6):589–616, 1994.

[129] Y.M. Kim and T.-H. Lai. The complexity of congestion-1 embedding in a hypercube. *Journal of Algorithms*, 12:246–280, 1991.

[130] H. Krawczyk and M. Kubale. An approximation algorithm for diagnostic tast scheduling in multicomputer systems. *IEEE Transactions on Computers*, 34(9):869–872, September 1985.

[131] R. Krishnamurti and E. Ma. An approximation algorithm for scheduling tasks on varying partition sizes in partitionable, multiprocessor systems. *IEEE Transactions on Computers*, 41(12):1572–1579, December 1992.

[132] R. Krishnamurti and B. Narahari. An approximation algorithm for preemptive scheduling on parallel-task systems. *SIAM Journal on Discrete Mathematics*, 8(4):661–669, 1995.

[133] P. Kruger, T.-H. Lai, and V.A. Dixit-Radiya. Job scheduling is more important than processor allocation for hypercube computers. *IEEE Transaction on Parallel and Distributed Sysytems*, 5(5):488–497, May 1994.

[134] M. Kubale. The complexity of scheduling independent two-processor tasks on dedicated processors. *Information Processing Letters*, 24:141–147, February 1987.

[135] M. Kubale. Podzielne uszeregowania zadań dwuprocesorowych na procesorach dedykowanych. *Zeszyty Naukowe Politechniki Śląskiej, Seria:Automatyka z.100, Nr kol. 1082*, pages 145–153, April 1990.

[136] M. Kubale. Podzielne uszeregowania zadań dwuprocesorowych na procesorach dedykowanych. *Zeszyty Naukowe Politechniki Śląskiej, Seria:Automatyka z.110, Nr kol. 1176*, pages 69–76, 1992.

[137] M. Kubale. *Introduction to Computational Complexity*. Wydawnictwo Politechniki Gdańskiej, Gdańsk, 1994.

[138] M. Kubale. Preemptive versus nonpreemptive scheduling of biprocessor tasks on dedicated processors. *European Journal of Operational Research*, 94:242–251, 1996.

[139] D. J. Kuck. A survey of parallel machine organization and programming. *ACM Computer Surveys*, 9(1):29–59, 1977.

[140] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 9(37):1088–1098, September 1988.

[141] V. Kumar and A. Gupta. Analysis of scalability of parallel algorithms and architectures: A survey. In *Proceedings of 1991 International Conference on Supercomputing 1991*, pages 396–405, New-York, 1991. ACM Press.

[142] T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[143] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnoy Kan, and D.B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In *Handbook in Operation Research and Management Science*. North-Holland, Amsterdam, 1993.

[144] C.-Y. Lee and X. Cai. Scheduling multiprocessor tasks without prespecified allocations, private communication, January 1996.

[145] T. Leighton. Methods for message routing in parallel machines. *Theoretical Computer Science*, 128:31–62, 1994.

[146] C.E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[147] J.K. Lenstra, M. Veldhorst, and B. Veltman. The complexity of scheduling trees with communication delays. *Journal of Algorithms*, 20:157–173, 1996.

[148] J.-F. Lin and S.-J. Chen. Scheduling algorithm for nonpreemptive multiprocessor tasks. *Computers Math. Applic.*, 28(4):85–92, 1994.

[149] E.L. Lloyd. Concurrent task systems. *Operations Research*, 29(1):189–201, 1981.

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

[151] R. Lüling and B. Monien. Load balancing for distributed branch & bound. In *Proceedings of 6th International Parallel Processing Symposium*, pages 543–548, 1992.

[152] P.-Y. R. Ma, E.Y.S. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Trans. on Computers*, 31(1):41–47, 1982.

[153] V.F. Magirou and J.Z. Millis. An algorithm for the multiprocessor assignment problem. *Operations Research Letters*, 8:351–356, 1989.

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

[155] E.P. Markatos and T.J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[156] E. McLellan. The Alpha AXP architecture and 21064 processor. *IEEE Micro*, 13(3):36–47, June 1993.

[157] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1–12, 1959.

[158] T. Muntean and E.-G. Talbi. A parallel genetic algorithm for process - processor mapping. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II*, pages 71–82. Elsevier Science Publishers B.V., 1991.

[159] R.R. Muntz and E.G. Coffman Jr. Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of the ACM*, 17(2):324–338, April 1970.

[160] M.W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Transactions on Software Engineering*, 18(4):319–328, April 1992.

[161] W.G. Nation, G. Saghi, and H.J. Siegel. Properties of interconnection networks for large-scale parallel processing systems. In *ISIPCALA'93 Drafts of Papers*, pages 51–82, 1993.

[162] B.C. Neuman and S. Rao. The Prospero Resource Manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, 6(4):339–335, 1994.

[163] L.M. Ni and P.K. McKinley. A survey of warmhole routing techniques in direct networks. *Computer*, 26(2):62–76, February 1993.

[164] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990.

[165] Parasoft Corp. *Express 3.1, Transputer 3L C*, 1991.

[166] J.L. Park and H. Choi. Circuit-switched broadcasting in torus mesh networks. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):184–190, February 1996.

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

[168] C. Picoleau. New complexity results on scheduling with small communication delays. *Discrete Applied Mathematics*, 60:331–342, 1995.

[169] J. Plehn. Preemptive scheduling of independent jobs with release times and deadlines on a hypercube. *IPL*, 34:161–166, April 1990.

[170] S.G.N. Prasanna and B.R. Musicus. Generalized multiprocessor scheduling for directed acyclic graphs. In *Proceedings of Supercomputing 1994*, pages 237–246. IEEE Press, 1994.

[171] F.P. Preparata, G. Metze, and R.T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, 16(6):848–854, December 1967.

[172] C.C. Price and M. Salama. Optimal task allocation in hypercube multiprocessor ensambles. *Computers & Mathematics with Applications*, 26(12):17–24, 1993.

[173] QNX Software Systems Ltd. *QNX: We work in real time*, 1994.

[174] V.J. Rayward-Smith. The complexity of preemptive scheduling given interprocessor communication delays. *Information Processing Letters*, 25:123–125, 1987.

[175] V.J. Rayward-Smith. UET scheduling with interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[176] T.G. Robertazzi. Processor equivalence for a linear daisy chain of load sharing processors. *IEEE Trans. on Aerospace and Electronic Systems*, 29(4):1216–1221, October 1993.

[177] Y. Saad and M.H. Schultz. Data communication in parallel architectures. *Parallel Computing*, 15(11):131–150, 1989.

[178] W. Schröder-Preikschat. Design principles of parallel operating systems. In *ISIPCALA'93 Drafts of Papers*, pages 51–82, 1993.

[179] K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, August 1992.

[180] Ch.L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[181] K.C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19:107–140, 1994.

[182] C.-C. Shen and W.-H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, 34(3):197–203, 1985.

[183] X. Shen and E.M. Reingold. Scheduling on a hypercube. *Information Processing Letters*, 40:323–328, December 1991.

[184] T. Shepard and J.A. Martin Gagne. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Transactions on Software Engineering*, 17(7):669–677, July 1991.

[185] A. Silberschatz, J.L. Peterson, and P.B. Galvin. *Operating Systems Concepts*. Addison-Wesley Publishing Co., 1991. In Polish: Podstawy systemów operacyjnych, WNT, Warszawa, 1993.

[186] Silicon Graphics Computer Systems. *Symmetric multiprocessing systems*, 1993. Technical Report.

[187] J. Skorin-Kapov. Tabu search applied to the quadratic assignment problem. *ORSA Journal on Computing*, 2(1):33–45, 1990.

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

[189] J. Sohn and T.G. Robertazzi. A muli-job load sharing strategy for divisible jobs on bus networks. Technical Report 697, Department of Electrical Engineering, SUNY at Stony Brook, Stony Brook, New York 11794, August 1994.

[190] J. Sohn and T.G. Robertazzi. An optimum load sharing strategy for divisible jobs with time-varying processor speed and channel speed. Technical Report 706, Department of Electrical Engineering, SUNY at Stony Brook, Stony Brook, New York 11794, January 1995.

[191] J. Sohn, T.G. Robertazzi, and S. Luryi. Optimizing computing costs using divisible load analysis. Technical Report 719, Department of Electrical Engineering, SUNY at Stony Brook, Stony Brook, New York 11794, October 1995.

[192] J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, June 1995.

[193] L.A. Steen (editor). *Mathematics Today, Tweleve Informal Essays*. Springer-Verlag, Berlin, 1979. in Polish: Matematyka współczesna, Dwanaście esejów, WNT, Warszawa, 1983.

[194] M. Stroiński and J. Węglarz. Problemy rozwoju sieci komputerowych z perspektywy globalnej infrastruktury informacyjnej. In *Miejskie Sieci Komputerowe w Nauce, Gospodarce i Administracji, POLMAN'96*, pages 9–16, 1996.

[195] C.B. Stunkel, D.G. Shea, D.G. Grice, P.H. Hochschild, and M. Tsao. The SP1 high-performance switch. In *Proceedings of Scalable High-Performance Computing Conference '94*, pages 150–157, 1994.

[196] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20:531–545, 1994.

[197] T. Thompson. When silicon hits its limits: What's next? *Byte*, 21:44–54, April 1996.

[198] B. Veltman. *Multiprocessor scheduling with communication delays*. PhD thesis, Technical University Eindhoven, 1993.

[199] B. Veltman, B.J. Lageweg, and J.K. Lenstra. Multiprocessor scheduling with communications delays. *Parallel Computing*, 16:173–182, 1990.

[200] V.G. Vizing. About schedules observing dead-lines (in Russian). *Kibernetika*, (1):128–135, 1981.

[201] V.G. Vizing. Minimization of the maximum delay in servicing systems with interruption. *U.S.S.R. Computatioanl Mathematics and Mathematical Physics*, 22(3):227–233, 1982.

[202] V.G. Vizing. Optimal choice of job execution intensities for a convex function of penalties for intensity (in Russian). *Kibernetika (Kiev)*, (3):125–127, 1982.

[203] V.G. Vizing, L.N. Komzakowa, and A.V. Tarchenko. About one algorithm for selecting the intensity of jobs in a schedule (in Russian). *Kibernetika*, (5):71–74, 1981.

[204] D. Walker. The design of a standard message passing interface for disrtibuted memory concurrent computers. *Parallel Computing*, 20:657–673, 1994.

[205] Q. Wang and K.-H. Cheng. List scheduling of parallel tasks. *Information Processing Letters*, 37:291–297, March 1991.

[206] Q. Wang and K.-H. Cheng. A heuristic of scheduling parallel tasks and its analysis. *SIAM Journal on Computing*, 21(2):281–294, April 1992.

[207] D.de Werra, P. Hell, T. Kameda, N. Katoch, Ph. Solot, and M. Yamashita. Graph endpoint coloring and distributed processing. *Networks*, 23:93–98, 1993.

[208] J. Węglarz. Scheduling under continuous performing speed vs. resource amunt activity models. In R. Słowiński and J. Węglarz, editors, *Advances in Project Scheduling*, pages 273–295. Elsevier Science Publisher B.V., Amsterdam, 1989.

[209] R.D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3(5):457–481, October 1991.

[210] C.M. Woodside and G.G. Monforton. Fast allocation of processes in distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):164–174, 1993.

[211] J. Xu and K. Hwang. Heuristic methods for dynamic load balancing in a message-passing multicomputer. *Journal of Parallel and Distributed Computing*, 18:1–13, 1993.

[212] J. Xu and D.L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, January 1993.

[213] J. Zahorjan, E.D. Lazowska, and D.L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–199, April 1991.

[214] Y. Zhu and M. Ahuja. On job scheduling on a hypercube. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):62–69, January 1993.

[215] J.R. Zorbas, D.J.Reble, and R.E. van Kooten. Measuring the scalability of parallel computer systems. In *Proceedings of Supercomputing 1989*, pages 832–841, New York, 1989. ACM Press.

Index