Contents

15. Parallel Computations

Parallel computations is concerned with solving a problem faster by using multiple processors in parallel. These processors may belong to a single machine, or to different machines that communicate through a network. In either case, the use of parallelism requires to split the problem into tasks that can be solved simultaneously.

In the following, we will take a brief look at the history of parallel computing, and then discuss reasons why parallel computing is harder than sequential computing. We explain differences from the related subjects of distributed and concurrent computing, and mention typical application areas. Finally, we outline the rest of this chapter.

Although the history of parallel computing can be followed back even longer, the first parallel computer is commonly said to be Illiac IV, an experimental 64-processor machine that became operational in 1972. The parallel computing area boomed in the late 80s and early 90s when several new companies were founded to build parallel machines of various types. Unfortunately, software was difficult to develop and non-portable at that time. Therefore, the machines were only adopted in the most compute-intensive areas of science and engineering, a market too small to commence for the high development costs. Thus many of the companies had to give up.

On the positive side, people soon discovered that cheap parallel computers can be built by interconnecting standard PCs and workstations. As networks became faster, these so-called *clusters* soon achieved speeds of the same order as the special-purpose machines. At present, the Top 500 list, a regularly updated survey of the most powerful computers worldwide, contains 42% clusters. Parallel computing also profits from the increasing use of multiprocessor machines which, while designed as servers for web etc., can as well be deployed in parallel computing. Finally, software portability problems have been solved by establishing widely used standards for parallel programming. The most important standards, MPI and OpenMP, will be explained in Subsections 15.3.1 and 15.3.2 of this book.

In summary, there is now an affordable hardware basis for parallel computing. Nevertheless, the area has not yet entered the mainstream, which is largely due to difficulties in developing parallel software. Whereas writing a sequential program requires to find an algorithm, that is, a sequence of elementary operations that solves the problem, and to formulate the algorithm in a programming language, parallel computing poses additional challenges:

- Elementary operations must be grouped into tasks that can be solved concurrently.
- The tasks must be scheduled onto processors.
- Depending on the architecture, data must be distributed to memory modules.
- Processes and threads must be managed, i.e., started, stopped and so on.
- Communication and synchronisation must be organised.

Of course, it is not sufficient to find any grouping, schedule etc. that work, but it is necessary to find solutions that lead to fast programs. Performance measures and general approaches to performance optimisation will be discussed in Section 15.2, where we will also elaborate on the items above. Unlike in sequential computing, different parallel architectures and programming models favour different algorithms.

In consequence, the design of parallel algorithms is more complex than the design of sequential algorithms. To cope with this complexity, algorithm designers often use simplified models. For instance, the Parallel Random Access Machine (see Subsection 15.4.1) provides a model in which opportunities and limitations of parallelisation can be studied, but it ignores communication and synchronisation costs.

We will now contrast parallel computing with the related fields of distributed and concurrent computing. Like parallel computing, *distributed computing* uses interconnected processors and divides a problem into tasks, but the purpose of division is different. Whereas in parallel computing, tasks are executed *at the same time*, in distributed computing tasks are executed *at different locations*, using *different resources*. These goals overlap, and many applications can be classified as both parallel and distributed, but the focus is different. Parallel computing emphasises homogeneous architectures, and aims at speeding up applications, whereas distributed computing deals with heterogeneity and openness, so that applications profit from the inclusion of different kinds of resources. Parallel applications are typically standalone and predictable, whereas distributed applications consist of components that are brought together at runtime.

Concurrent computing is not bound to the existence of multiple processors, but emphasises the fact that several sub-computations are in progress at the same time. The most important issue is guaranteeing correctness for any execution order, which can be parallel or interleaved. Thus, the relation between concurrency and parallelism is comparable to the situation of reading several books at a time. Reading the books concurrently corresponds to having a bookmark in each of them and to keep track of all stories while switching between books. Reading the books in parallel, in contrast, requires to look into all books at the same time (which is probably impossible in practice). Thus, a concurrent computation may or may not be parallel, but a parallel computation is almost always concurrent. An exception is data parallelism, in which the instructions of a single program are applied to different data in parallel. This approach is followed by SIMD architectures, as described below.

For the emphasis on speed, typical application areas of parallel computing are science and engineering, especially numerical solvers and simulations. These applications tend to have high and increasing computational demands, since more computing power allows one to work with more detailed models that yield more accurate results. A second reason for using parallel machines is their higher memory capacity, due to which more data fit into a fast memory level such as cache.

The rest of this chapter is organised as follows: In Section 15.1, we give a brief overview and classification of current parallel architectures. Then, we introduce basic concepts such as task and process, and discuss performance measures and general approaches to the improvement of efficiency in Section 15.2. Next, Section 15.3 describes parallel programming models, with focus on the popular MPI and OpenMP standards. After having given this general background, the rest of the chapter delves into the subject of parallel algorithms from a more theoretical perspective. Based on example algorithms, techniques for parallel algorithm design are introduced. Unlike in sequential computing, there is no universally accepted model for parallel algorithm design and analysis, but various models are used depending on purpose. Each of the models represents a different compromise between the conflicting goals of accurately reflecting the structure of real architectures on one hand, and keeping algorithm design and analysis simple on the other. Section 15.4 gives an overview of the models, Section 15.5 introduces the basic concepts of parallel algorithmics, Sections 15.6 and 15.7 explain deterministic example algorithms for PRAM and mesh computational model.

15.1. Parallel architectures

A simple, but well-known classification of parallel architectures has been given in 1972 by Michael Flynn. He distinguishes computers into four classes: SISD, SIMD, MISD, and MIMD architectures, as follows:

- SI stands for "single instruction", that is, the machine carries out a single instruction at a time.
- MI stands for "multiple instruction", that is, different processors may carry out different instructions at a time.
- SD stands for "single data", that is, only one data item is processed at a time.
- MD stands for "multiple data", that is, multiple data items may be processed at a time.

SISD computers are von-Neumann machines. MISD computers have probably never been built. Early parallel computers were SIMD, but today most parallel computers are MIMD. Although the scheme is of limited classification power, the abbreviations are widely used.

The following more detailed classification distinguishes parallel machines into SIMD, SMP, ccNUMA, nccNUMA, NORMA, clusters, and grids.

15.1.1. SIMD architectures

As depicted in Figure 15.1, a SIMD computer is composed of a powerful control processor and several less powerful processing elements (PEs). The PEs are typically arranged as a mesh so that each PE can communicate with its immediate neighbours.

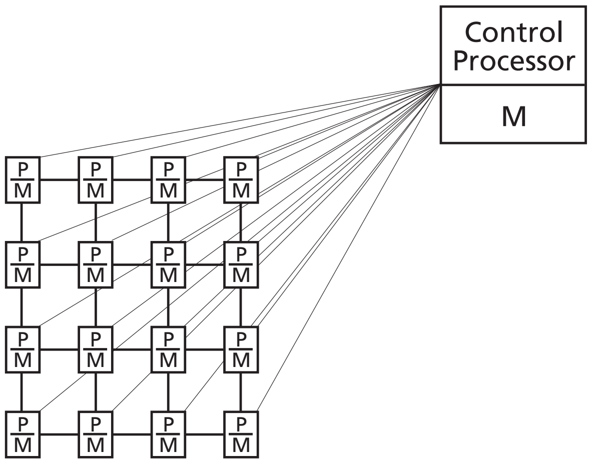


Figure 15.1 SIMD architecture.

A program is a single thread of instructions. The control processor, like the processor of a sequential machine, repeatedly reads a next instruction and decodes it. If the instruction is sequential, the control processor carries out the instruction on data in its own memory. If the instruction is parallel, the control processor broadcasts the instruction to the various PEs, and these simultaneously apply the instruction to different data in their respective memories. As an example, let the instruction be `LD reg, 100`. Then, all processors load the contents of memory address 100 to `reg`, but memory address 100 is physically different for each of them. Thus, all processors carry out the same instruction, but read different values (therefore "SIMD"). For a statement of the form `if` *test* `then` *if_branch* `else` *else_branch*, first all processors carry out the test simultaneously, then some carry out *if_branch* while the rest sits idle, and finally the rest carries out *else_branch* while the formers sit idle. In consequence, SIMD computers are only suited for applications with a regular structure. The architectures have been important historically, but nowadays have almost disappeared.

15.1.2. Symmetric multiprocessors

Symmetric multiprocessors (SMP) contain multiple processors that are connected to a single memory. Each processor may access each memory location through standard load/store operations of the hardware. Therefore, programs, including the operating system, must only be stored once. The memory can be physically divided into modules, but the access time is the same for each pair of a processor and a memory module (therefore "symmetric"). The processors are connected to the memory by a bus (see Figure 15.2), by a crossbar, or by a network of switches. In either case, there is a delay for memory accesses which, partially due to competition for network resources, grows with the number of processors.

In addition to main memory, each processor has one or several levels of cache

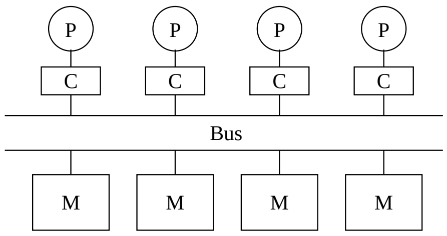


Figure 15.2 Bus-based SMP architecture.

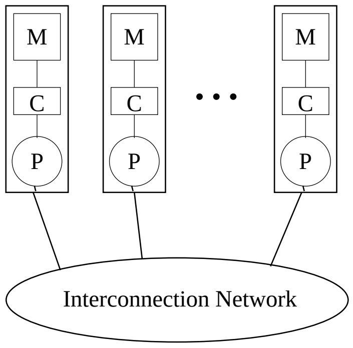


Figure 15.3 ccNUMA architecture.

with faster access. Between memory and cache, data are moved in units of cache lines. Storing a data item in multiple caches (and writing to it) gives rise to coherency problems. In particular, we speak of *false sharing* if several processors access the same cache line, but use different portions of it. Since coherency mechanisms work at the granularity of cache lines, each processor assumes that the other would have updated its data, and therefore the cache line is sent back and forth.

15.1.3. Cache-coherent NUMA architectures:

NUMA stands for **N**on-**U**niform **M**emory **A**ccess, and contrasts with the symmetry property of the previous class. The general structure of ccNUMA architectures is depicted in Figure 15.3. As shown in the figure, each processor owns a *local memory*, which can be accessed faster than the rest called *remote memory*. All memory is accessed through standard load/store operations, and hence programs, including the operating system, must only be stored once. As in SMPs, each processor owns one or several levels of cache; cache coherency is taken care of by the hardware.

15.1.4. Non-cache-coherent NUMA architectures:

nccNUMA (**n**on **c**ache **c**oherent **N**on-**U**uniform **M**emory **A**ccess) architectures differ from ccNUMA architectures in that the hardware puts into a processor's cache only data from local memory. Access to remote memory can still be accomplished through standard load/store operations, but it is now up to the operating system to first move the corresponding page to local memory. This difference simplifies hardware design, and thus nccNUMA machines scale to higher processor numbers. On the backside, the operating system gets more complicated, and the access time to remote memory grows. The overall structure of Figure 15.3 applies to nccNUMA architectures as well.

15.1.5. No remote memory access architectures

NORMA (**NO** **R**emote **M**emory **A**cess) architectures differ from the previous class in that the remote memory must be accessed through slower I/O operations as opposed to load/store operations. Each node, consisting of processor, cache and local memory, as depicted in Figure 15.3, holds an own copy of the operating system, or at least of central parts thereof. Whereas SMP, ccNUMA, and nccNUMA architectures are commonly classified as shared memory machines, SIMD architectures, NORMA architectures, clusters, and grids (see below) fall under the heading of distributed memory.

15.1.6. Clusters

According to Pfister, a cluster is a type of parallel or distributed system that consists of a collection of interconnected whole computers that are used as a single, unified computing resource. Here, the term “whole computer” denotes a PC, workstation or, increasingly important, SMP, that is, a node that consists of processor(s), memory, possibly peripheries, and operating system. The use as a single, unified computing resource is also denoted as single system image SSI. For instance, we speak of SSI if it is possible to login into the system instead of into individual nodes, or if there is a single file system. Obviously, the SSI property is gradual, and hence the borderline to distributed systems is fuzzy. The borderline to NORMA architectures is fuzzy as well, where the classification depends on the degree to which the system is designed as a whole instead of built from individual components.

Clusters can be classified according to their use for parallel computing, high throughput computing, or high availability. Parallel computing clusters can be further divided into *dedicated clusters*, which are solely built for the use as parallel machines, and *campus-wide clusters*, which are distributed systems with part-time use as a cluster. Dedicated clusters typically do not contain peripheries in their nodes, and are interconnected through a high-speed network. Nodes of campus-wide clusters, in contrast, are often desktop PCs, and the standard network is used for intra-cluster communication.

15.1.7. Grids

A grid is a hardware/software infrastructure for shared usage of resources and problem solution. Grids enable coordinated access to resources such as processors, memories, data, devices, and so on. Parallel computing is one out of several emerging application areas. Grids differ from other parallel architectures in that they are large, heterogeneous, and dynamic. Management is complicated by the fact that grids cross organisational boundaries.

15.2. Performance in practice

As explained in the introduction, parallel computing splits a problem into *tasks* that are solved independently. The tasks are implemented as either *processes* or *threads*. A detailed discussion of these concepts can be found in operating system textbooks such as Tanenbaum. Briefly stated, processes are programs in execution. For each process, information about resources such as memory segments, files, and signals is stored, whereas threads exist within processes such that multiple threads share resources. In particular, threads of a process have access to shared memory, while processes (usually) communicate through explicit message exchange. Each thread owns a separate PC and other register values, as well as a stack for local variables. Processes can be considered as units for resource usage, whereas threads are units for execution on the CPU. As less information needs to be stored, it is faster to create, destroy and switch between threads than it is for processes.

Whether threads or processes are used, depends on the architecture. On shared-memory machines, threads are usually faster, although processes may be used for program portability. On distributed memory machines, only processes are a priori available. Threads can be used if there is a software layer (distributed shared memory) that implements a shared memory abstraction, but these threads have higher communication costs.

Whereas the notion of tasks is problem-related, the notions of processes and threads refer to implementation. When designing an algorithm, one typically identifies a large number of tasks that can potentially be run in parallel, and then maps several of them onto the same process or thread.

Parallel programs can be written in two styles that can also be mixed: With ***data parallelism,*** the same operation is applied to different data at a time. The operation may be a machine instruction, as in SIMD architectures, or a complex operation such as a function application. In the latter case, different processors carry out different instructions at a time. With ***task parallelism,*** in contrast, the processes/threads carry out different tasks. Since a function may have an `if` or `case` statement as the outermost construct, the borderline between data parallelism and task parallelism is fuzzy.

Parallel programs that are implemented with processes can be further classified as using *Single Program Multiple Data* (SPMD) or *Multiple Program Multiple Data* (MPMD) coding styles. With SPMD, all processes run the same program, whereas with MPMD they run different programs. MPMD programs are task-parallel, whereas SPMD programs may be either task-parallel or data-parallel. In SPMD mode, task parallelism is expressed through conditional statements.

As the central goal of parallel computing is to run programs faster, performance measures play an important role in the field. An obvious measure is execution time, yet more frequently the derived measure of speedup is used. For a given problem,

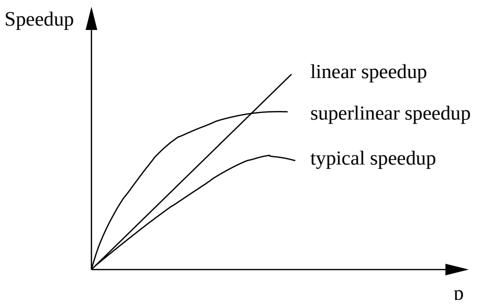


Figure 15.4 Ideal, typical, and super-linear speedup curves.

speedup is defined by

$$speedup(p) = \frac{T_1}{T_p} \ ,$$

where T_1 denotes the running time of the fastest sequential algorithm, and T_p denotes the running time of the parallel algorithm on p processors. Depending on context, speedup may alternatively refer to using p processes or threads instead of p processors. A related, but less frequently used measure is efficiency, defined by

$$efficiency(p) = \frac{speedup(p)}{p} \ .$$

Unrelated to this definition, the term efficiency is also used informally as a synonym for good performance.

Figure 15.4 shows ideal, typical, and super-linear speedup curves. The ideal curve reflects the assumption that an execution that uses twice as many processors requires half of the time. Hence, ideal speedup corresponds to an efficiency of one. Super-linear speedup may arise due to cache effects, that is, the use of multiple processors increases the total cache size, and thus more data accesses can be served from cache instead of from slower main memory.

Typical speedup stays below ideal speedup, and grows up to some number of processors. Beyond that, use of more processors slows down the program. The difference between typical and ideal speedups has several reasons:

- *Amdahl's law* states that each program contains a serial portion s that is not amenable to parallelisation. Hence, $T_p > s$, and thus $speedup(p) < T_1/s$, that is, the speedup is bounded from above by a constant. Fortunately, another observation, called *Gustafson-Barsis law* reduces the practical impact of Amdahl's law. It states that in typical applications, the parallel variant does not speed up a fixed problem, but runs larger instances thereof. In this case, s may grow slower than T_1, so that T_1/s is no longer constant.
- Task management, that is, the starting, stopping, interrupting and scheduling of processes and threads, induces a certain overhead. Moreover, it is usually impossible, to evenly balance the load among the processes/threads.

- Communication and synchronisation slow down the program. Communication denotes the exchange of data, and synchronisation denotes other types of coordination such as the guarantee of mutual exclusion. Even with high-speed networks, communication and synchronisation costs are orders of magnitude higher than computation costs. Apart from physical transmission costs, this is due to protocol overhead and delays from competition for network resources.

Performance can be improved by minimising the impact of the factors listed above. Amdahl's law is hard to circumvent, except that a different algorithm with smaller s may be devised, possibly at the price of larger T_1. Algorithmic techniques will be covered in later sections; for the moment, we concentrate on the other performance factors.

As explained in the previous section, tasks are implemented as processes or threads such that a process/thread typically carries out multiple tasks. For high performance, the granularity of processes/threads should be chosen in relation to the architecture. Too many processes/threads unnecessarily increase the costs of task management, whereas too few processes/threads lead to poor machine usage. It is useful to map several processes/threads onto the same processor, since the processor can switch when it has to wait for I/O or other delays. Large-granularity processes/threads have the additional advantage of a better communication-to-computation ratio, whereas fine-granularity processes/threads are more amenable to load balancing.

Load balancing can be accomplished with static or dynamic schemes. If the running time of the tasks can be estimated in advance, static schemes are preferable. In these schemes, the programmer assigns to each process/thread some number of tasks with about the same total costs. An example of a dynamic scheme is master/slave. In this scheme, first a master process assigns one task to each slave process. Then, repeatedly, whenever a slave finishes a task, it reports to the master and is assigned a next task, until all tasks have been processed. This scheme achieves good load balancing at the price of overhead for task management.

The highest impact on performance usually comes from reducing communication/synchronisation costs. Obvious improvements result from changes in the architecture or system software, in particular from reducing ***latency,*** that is, the delay for accessing a remote data item, and ***bandwidth,*** that is, the amount of data that can be transferred per unit of time.

The algorithm designer or application programmer can reduce communication/synchronisation costs by minimising the number of interactions. An important approach to achieve this minimisation is locality optimisation. ***Locality,*** a property of (sequential or parallel) programs, reflects the degree of temporal and spatial concentration of accesses to the same data. In distributed-memory architectures, for instance, data should be stored at the processor that uses the data. Locality can be improved by code transformations, data transformations, or a combination thereof.

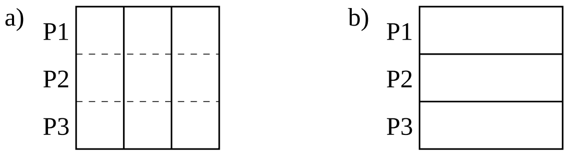


Figure 15.5 Locality optimisation by data transformation.

As an example, consider the following program fragment to be executed on three processors:

```
for (i=0; i<N; i++) in parallel
  for (j=0; j<N; j++)
    f(A[i][j]);
```

Here, the keyword "in parallel" means that the iterations are evenly distributed among the processors so that P_0 runs iterations $i = 0, \ldots, N/3$, P_1 runs iterations $i = N/3 + 1, \ldots, 2N/3$, and P_2 runs iterations $i = 2N/3 + 1, \ldots, N - 1$ (rounded if necessary). The function f is supposed to be free of side effects.

With the data distribution of Figure 15.5a), locality is poor, since many accesses refer to remote memory. Locality can be improved by changing the data distribution to that of Figure 15.5b) or, alternatively, by changing the program into

```
for (j=0; j<N; j++) in parallel
  for (i=0; i<N; i++)
    f(A[i][j]);
```

The second alternative, code transformations, has the advantage of being applicable selectively to a portion of code, whereas data transformations influence the whole program so that an improvement in one part may slow down another. Data distributions are always correct, whereas code transformations must respect *data dependencies*, which are ordering constraints between statements. For instance, in

```
a = 3;          (1)
b = a;          (2)
```

a data dependence occurs between statements (1) and (2). Exchanging the statements would lead to an incorrect program.

On shared-memory architectures, a programmer does not the specify data distribution, but locality has a high impact on performance, as well. Programs run faster if data that are used together are stored in the same cache line. On shared-memory architectures, the data layout is chosen by the compiler, e.g. row-wise in C. The programmer has only indirect influence through the manner in which he or she declares data structures.

Another opportunity to reduce communication costs is replication. For instance, it pays off to store frequently used data at multiple processors, or to repeat short computations instead of communicating the result.

Synchronisations are necessary for correctness, but they slow down program execution, first because of their own execution costs, and second because they cause processes to wait for each other. Therefore, excessive use of synchronisation should be avoided. In particular, critical sections (in which processes/threads require exclusive access to some resource) should be kept at a minimum. We speak of ***sequentialisation*** if only one process is active at a time while the others are waiting.

Finally, performance can be improved by latency hiding, that is, parallelism between computation and communication. For instance, a process can start a remote read some time before it needs the result (prefetching), or write data to remote memory in parallel to the following computations.

Exercises

15.2-1 For standard matrix multiplication, identify tasks that can be solved in parallel. Try to identify as many tasks as possible. Then, suggest different opportunities for mapping the tasks onto (a smaller number of) threads, and compare these mappings with respect to their efficiency on a shared-memory architecture.

15.2-2 Consider a parallel program that takes as input a number n and computes as output the number of primes in range $2 \leq p \leq n$. Task T_i of the program should determine whether i is a prime, by systematically trying out all potential factors, that is, dividing by 2, $\ldots$, $\sqrt{i}$. The program is to be implemented with a fixed number of processes or threads. Suggest different opportunities for this implementation and discuss their pros and cons. Take into account both static and dynamic load balancing schemes.

15.2-3 Determine the data dependencies of the following stencil code:

```
for (t=0; t<tmax; t++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      a[i][j] += a[i-1][j] + a[i][j-1]
```

Restructure the code so that it can be parallelised.

15.2-4 Formulate and prove the bounds of the speedup known as Amdahl law and Gustafson-Barsis law. Explain the virtual contradiction between these laws. What can you say on the practical speedup?

15.3. Parallel programming

Partly due to the use of different architectures and the novelty of the field, a large number of parallel programming models has been proposed. The most popular models today are message passing as specified in the Message Passing Interface standard (MPI), and structured shared-memory programming as specified in the OpenMP standard. These programming models are discussed in Subsections 15.3.1 and 15.3.2, respectively. Other important models such as threads programming, data parallelism, and automatic parallelisation are outlined in Subsection 15.3.3.

15.3.1. MPI programming

As the name says, MPI is based on the programming model of message passing. In this model, several processes run in parallel and communicate with each other by sending and receiving messages. The processes do not have access to a shared memory, but accomplish all communication through explicit message exchange. A communication involves exactly two processes: one that executes a send operation, and another that executes a receive operation. Beyond message passing, MPI includes collective operations and other communication mechanisms.

Message passing is asymmetric in that the sender must state the identity of the receiver, whereas the receiver may either state the identity of the sender, or declare its willingness to receive data from any source. As both sender and receiver must actively take part in a communication, the programmer must plan in advance when a particular pair of processes will communicate. Messages can be exchanged for several purposes:

- exchange of data with details such as the size and types of data having been planned in advance by the programmer
- exchange of control information that concerns a subsequent message exchange, and
- synchronisation that is achieved since an incoming message informs the receiver about the sender's progress. Additionally, the sender may be informed about the receiver's progress, as will be seen later. Note that synchronisation is a special case of communication.

The MPI standard has been introduced in 1994 by the MPI forum, a group of hardware and software vendors, research laboratories, and universities. A significantly extended version, MPI-2, appeared in 1997. MPI-2 has about the same core functionality as MPI-1, but introduces additional classes of functions.

MPI describes a set of library functions with language binding to C, C++, and Fortran. With notable exceptions in MPI-2, most MPI functions deal with interprocess communication, leaving issues of process management such as facilities to start and stop processes, open. Such facilities must be added outside the standard, and are consequently not portable. For this and other reasons, MPI programs typically use a fixed set of processes that are started together at the beginning of a program run. Programs can be coded in SPMD or MPMD styles. It is possible to write parallel programs using only six base functions:

- MPI_Init must be called before any other MPI function.
- MPI_Finalize must be called after the last MPI function.
- MPI_Comm_size yields the total number of processes in the program.
- MPI_Comm_rank yields the number of the calling process, with processes being numbered starting from 0.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main (int argc, char **argv) {
  char msg[20];
  int me, total, tag=99;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  MPI_Comm_size(MPI_COMM_WORLD, &total);

  if (me==0) {
    strcpy(msg, "Hello");
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag,
             MPI_COMM_WORLD);
  }
  else if (me==1) {
    MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
             &status);
    printf("Received: %s \n", msg);
  }
  MPI_Finalize();
  return 0;
}
```

Figure 15.6 A simple MPI program.

- MPI_Send sends a message. The function has the following parameters:
  - address, size, and data type of the message,
  - number of the receiver,
  - message tag, which is a number that characterises the message in a similar way like the subject characterises an email,
  - communicator, which is a group of processes as explained below.
- MPI_Recv receives a message. The function has the same parameters as MPI_Send, except that only an upper bound is required for the message size, a wildcard may be used for the sender, and an additional parameter called status returns information about the received message, e.g. sender, size, and tag.

Figure 15.6 depicts an example MPI program.

Although the above functions are sufficient to write simple programs, many more functions help to improve the efficiency and/or structure MPI programs. In particular, MPI-1 supports the following classes of functions:

- *Alternative functions for pairwise communication:* The base MPI_Send function, also called standard mode send, returns if either the message has been delivered to the receiver, or the message has been buffered by the system. This decision is left to MPI. Variants of MPI_Send enforce one of the alternatives: In synchronous mode, the send function only returns when the receiver has started receiving the message, thus synchronising in both directions. In buffered mode, the system is required to store the message if the receiver has not yet issued MPI_Recv.

  On both the sender and receiver sides, the functions for standard, synchronous, and buffered modes each come in blocking and nonblocking variants. Blocking variants have been described above. Nonblocking variants return immediately after having been called, to let the sender/receiver continue with program execution while the system accomplishes communication in the background. Nonblocking communications must be completed by a call to MPI_Wait or MPI_Test to make sure the communication has finished and the buffer may be reused. Variants of the completion functions allow to wait for multiple outstanding requests.

  MPI programs can deadlock, for instance if a process P_0 first issues a send to process P_1 and then a receive from P_1; and P_1 does the same with respect to P_0. As a possible way-out, MPI supports a combined send/receive function.

  In many programs, a pair of processes repeatedly exchanges data with the same buffers. To reduce communication overhead in these cases, a kind of address labels can be used, called persistent communication. Finally, MPI functions MPI_Probe and MPI_Iprobe allow to first inspect the size and other characteristics of a message before receiving it.

- *Functions for Datatype Handling:* In simple forms of message passing, an array of equally-typed data (e.g. float) is exchanged. Beyond that, MPI allows to combine data of different types in a single message, and to send data from non-contiguous buffers such as every second element of an array. For these purposes, MPI defines two alternative classes of functions: user-defined data types describe a pattern of data positions/types, whereas packaging functions help to put several data into a single buffer. MPI supports heterogeneity by automatically converting data if necessary.

- *Collective communication functions:* These functions support frequent patterns of communication such as broadcast (one process sends a data item to all other processes). Although any pattern can be implemented by a sequence of sends/receives, collective functions should be preferred since they improve program compactness/understandability, and often have an optimised implementation. Moreover, implementations can exploit specifics of an architecture, and so a program that is ported to another machine may run efficiently on the new machine as well, by using the optimised implementation of that machine.

- *Group and communicator management functions:* As mentioned above, the send and receive functions contain a communicator argument that describes a group of processes. Technically, a communicator is a distributed data structure that tells each process how to reach the other processes of its group, and contains additional information called attributes. The same group may be described by different communicators. A message exchange only takes place if the communicator arguments of MPI_Send and MPI_Recv match. Hence, the use of communicators partitions the messages of a program into disjoint sets that do not influence each other. This way, communicators help structuring programs, and contribute to correctness. For libraries that are implemented with MPI, communicators allow to separate library traffic from traffic of the application program. Groups/communicators are necessary to express collective communications. The attributes in the data structure may contain application-specific information such as an error handler. In addition to the (intra)communicators described so far, MPI supports intercommunicators for communication between different process groups.

MPI-2 adds four major groups of functions:

- *Dynamic process management functions:* With these functions, new MPI processes can be started during a program run. Additionally, independently started MPI programs (each consisting of multiple processes) can get into contact with each other through a client/server mechanism.
- *One-sided communication functions:* One-sided communication is a type of shared-memory communication in which a group of processes agrees to use part of their private address spaces as a common resource. Communication is accomplished by writing into and reading from that shared memory. One-sided communication differs from other shared-memory programming models such as OpenMP in that explicit function calls are required for the memory access.
- *Parallel I/O functions:* A large set of functions allows multiple processes to collectively read from or write to the same file.
- *Collective communication functions for intercommunicators:* These functions generalise the concept of collective communication to intercommunicators. For instance, a process of one group may broadcast a message to all processes of another group.

15.3.2. OpenMP programming

OpenMP derives its name from being an open standard for multiprocessing, that is for architectures with a shared memory. Because of the shared memory, we speak of threads (as opposed to processes) in this section.

Shared-memory communication is fundamentally different from message passing: Whereas message passing immediately involves two processes, shared-memory communication uncouples the processes by inserting a medium in-between. We speak of read/write instead of send/receive, that is, a thread writes into memory, and another thread later reads from it. The threads need not know each other, and a written value may be read by several threads. Reading and writing may be separated by an arbitrary amount of time. Unlike in message passing, synchronisation must be organised explicitly, to let a reader know when the writing has finished, and to avoid concurrent manipulation of the same data by different threads.

OpenMP is one type of shared-memory programming, while others include one-sided communication as outlined in Subsection 15.3.1, and threads programming as

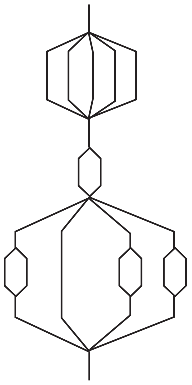


Figure 15.7 Structure of an OpenMP program.

outlined in Subsection 15.3.3. OpenMP differs from other models in that it enforces a fork-join structure, which is depicted in Figure 15.7. A program starts execution as a single thread, called master thread, and later creates a team of threads in a so-called parallel region. The master thread is part of the team. Parallel regions may be nested, but the threads of a team must finish together. As shown in the figure, a program may contain several parallel regions in sequence, with possibly different numbers of threads.

As another characteristic, OpenMP uses compiler directives as opposed to library functions. Compiler directives are hints that a compiler may or may not take into account. In particular, a sequential compiler ignores the directives. OpenMP supports incremental parallelisation, in which one starts from a sequential program, inserts directives at the most performance-critical sections of code, later inserts more directives if necessary, and so on.

OpenMP has been introduced in 1998, version 2.0 appeared in 2002. In addition to compiler directives, OpenMP uses a few library functions and environment variables. The standard is available for C, C++, and Fortran.

Programming OpenMP is easier than programming MPI since the compiler does part of the work. An OpenMP programmer chooses the number of threads, and then specifies work sharing in one of the following ways:

- *Explicitly:* A thread can request its own number by calling the library function `omp_get_thread_num`. Then, a conditional statement evaluating this number explicitly assigns tasks to the threads, similar as in SPMD-style MPI programs.

- *Parallel loop:* The compiler directive `#pragma omp parallel for` indicates that the following `for` loop may be executed in parallel so that each thread carries out several iterations (tasks). An example is given in Figure 15.8. The programmer can influence the work sharing by specifying parameters such as `schedule(static)` or `schedule(dynamic)`. Static scheduling means that each thread gets an about equal-sized block of consecutive iterations. Dynamic scheduling means that first each thread is assigned one iteration, and then, repeatedly, a thread that has fin-

```
#include <omp.h>
#define N 100
double a[N][N], b[N], c[N];
int main() {
  int i, j;
  double h;
  /* initialisation omitted */
  omp_set_num_threads(4);
  #pragma omp parallel for shared(a,b,c) private(j)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      c[i] += a[i][j] * b[j];
  /* output omitted */
}
```

Figure 15.8 Matrix-vector multiply in OpenMP using a parallel loop.

ished an iteration gets the next one, as in the master/slave paradigma described before for MPI. Different from master/slave, the compiler decides which thread carries out which tasks, and inserts the necessary communications.

- *Task-parallel sections:* The directive #pragma omp parallel sections allows to specify a list of tasks that are assigned to the available threads.

Threads communicate through shared memory, that is, they write to or read from shared variables. Only part of the variables are shared, while others are private to a particular thread. Whether a variable is private or shared is determined by rules that the programmer can overwrite.

Many OpenMP directives deal with synchronisation that is necessary for mutual exclusion, and to provide a consistent view of shared memory. Some synchronisations are inserted implicitly by the compiler. For instance, at the end of a parallel loop all threads wait for each other before proceeding with a next loop.

15.3.3. Other programming models

While MPI and OpenMP are the most popular models, other approaches have practical importance as well. Here, we outline threads programming, High Performance Fortran, and automatic parallelisation.

Like OpenMP, *threads programming* or by Java threads uses shared memory. Threads operate on a lower abstraction level than OpenMP in that the programmer is responsible for all details of thread management and work sharing. In particular, threads are created explicitly, one at a time, and each thread is assigned a function to be carried out. Threads programming focuses on task parallelism, whereas OpenMP programming focuses on data parallelism. Thread programs may be unstructured, that is, any thread may create and stop any other. OpenMP programs are often compiled into thread programs.

Data parallelism provides for a different programming style that is explicitly supported by languages such as *High Performance Fortran* (HPF). While data parallelism can be expressed in MPI, OpenMP etc., data-parallel languages center on the approach. As one of its major constructs, HPF has a parallel loop whose iterations are carried out independently, that is, without communication. The data-parallel style makes programs easier to understand since there is no need to take care of concurrent activities. On the backside, it may be difficult to force applications into this structure. HPF is targeted at single address space distributed memory architectures, and much of the language deals with expressing data distributions. Whereas MPI programmers distribute data by explicitly sending them to the right place, HPF programmers specify the data distribution on a similar level of abstraction as OpenMP programmers specify the scheduling of parallel loops. Details are left to the compiler. An important concept of OpenMP is the owner-computes rule, according to which the owner of the left-hand side variable of an assignment carries out an operation. Thus, data distribution implies the distribution of computations.

Especially for programs from scientific computing, a significant performance potential comes from parallelising loops. This parallelisation can often be accomplished automatically, by *parallelising compilers*. In particular, these compilers check for data dependencies. that prevent parallelisation. Many programs can be restructured to circumvent the dependence, for instance by exchanging outer and inner loops. Parallelising compilers find these restructuring for important classes of programs.

Exercises

15.3-1 Sketch an MPI program for the prime number problem of Exercise 15.2-3. The program should deploy the master/slave paradigma. Does your program use SPMD style or MPMD style?

15.3-2 Modify your program from Exercise 15.3-1 so that it uses collective communication.

15.3-3 Compare MPI and OpenMP with respect to programmability, that is, give arguments why or to which extent it is easier to program in either MPI or OpenMP.

15.3-4 Sketch an OpenMP program that implements the stencil code example of Exercise 15.2-3.

15.4. Computational models

15.4.1. PRAM

The most popular computational model is the **P**arallel **R**andom **A**ccess **M**achine (PRAM) which is a natural generalisation of the **R**andom **A**ccess **M**achine (RAM).

The PRAM model consists of p synchronised processors $P_1, P_2, \ldots, P_p$, a shared memory with memory cells $M[1]$, $M[2]$, $\ldots$, $M[m]$ and memories of the processors. Figure 15.9. shows processors and the shared random access memory

There are variants of this model. They differ in whether multiple processors are allowed to access the same memory cell in a step, and in how the resulting conflicts are resolved. In particular the following variants are distinguished:

Types on the base of the properties of read/write operations are

- EREW (Exclusive-Read Exclusive-Write) PRAM,

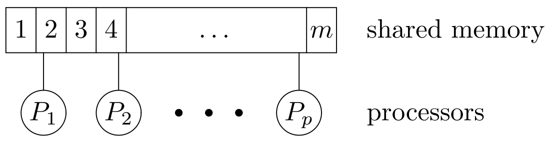


Figure 15.9 Parallel random access machine.

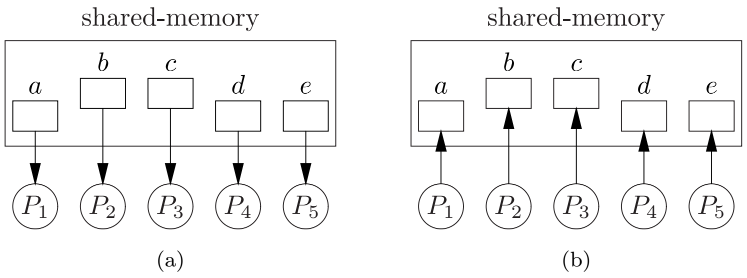
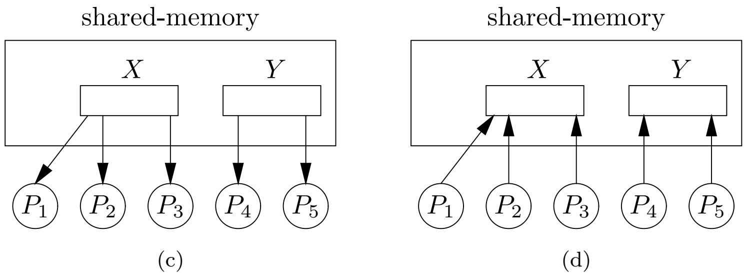



Figure 15.10 Types of parallel random access machines.

- ERCW (Exclusive-Read Concurrent-Write) PRAM,
- CREW (Concurrent-Read Exclusive-Write) PRAM,
- CRCW (Concurrent-Read Concurrent-Write) PRAM.

Figure 15.10(a) shows the case when at most one processor has access a memory cell (ER), and Figure 15.10(d) shows, when multiple processors have access the same cell (CW).

Types of concurrent writing are *common, priority, arbitrary, combined.*

15.4.2. BSP, LogP and QSM

Here we consider the models BSP, LogP and QSM.

Bulk-synchronous Parallel Model (BSP) describes a computer as a collection of nodes, each consisting of a processor and memory. BSP supposes the existence of a router and a barrier synchronisation facility. The router transfers messages between the nodes, the barrier synchronises all or a subset of nodes. According to BSP compu-

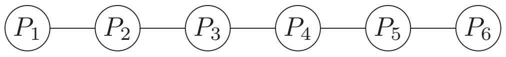


Figure 15.11 A chain consisting of six processors.

tation is partitioned into ***supersteps.*** In a superstep each processor independently performs computations on data in its own memory, and initiates communications with other processors. The communication is guaranteed to complete until the beginning of the next superstep.

g is defined such that gh is the time that is takes to route an h-relation under continuous traffic conditions. An h-relation is a communication pattern in which each processor sends and receives up to h messages.

The cost of a superstep is determined as $x + gh + l$, where x is the maximum number of communications initiated by any processor. The cost of a program is the sum of the costs of the individual supersteps.

BSP contains a cost model that involves three parameters: the number of processors (p), the cost of a barrier synchronisation (l) and a characteristics of the available bandwidth (g).

LogP model was motivated by inaccuracies of BSP and the restrictive requirement to follow the superstep structure.

While LogP improves on BSP with respect to reflectivity, QSM improves on it with respect to simplicity. In contrast to BSP, QSM is a shared-memory model. As in BSP, the computation is structured into supersteps, and each processor has its own local memory. In a superstep, a processor performs computations on values in the local memory, and initiates read/write operations to the shared memory. All shared-memory accesses complete until the beginning of the next superstep. QSM allows for concurrent reads and writes. Let the maximum number of accesses to any cell in a superstep be k. Then QSM charges costs $\max(x, gh, k)$, with x, g, and h being defined in BSP.

15.4.3. Mesh, hypercube and butterfly

Mesh also is a popular computational model. A d-dimensional mesh is an $a_1 \times a_2 \times \cdots \times a_d$ sized grid having a processor in each grid point. The edges are the communication lines, working in two directions. Processors are labelled by d-tuples, as $P_{i_1,i_2,\ldots,i_d}$.

Each processor is a RAM, having a local memory. The local memory of the processor $P_{i_1,i_2,\ldots,i_d}$ is $M[i_1,\ldots,i_d,1],\ldots,M[i_1,\ldots,i_d,m]$. Each processor can execute in one step such basic operations as adding, subtraction, multiplication, division, comparison, read and write from/into the local memory, etc. Processors work in synchronised way, according to a global clock.

The simplest mesh is the ***chain,*** belonging to the value $d = 1$. Figure 15.11 shows a chain consisting of 6 processors.

The processors of a chain are $P_1, \ldots, P_p$. P_1 is connected with P_{p-1}, P_p is connected with P_{p-1}, the remaining processors P_i are connected with P_{i-1} and P_{i+1}.

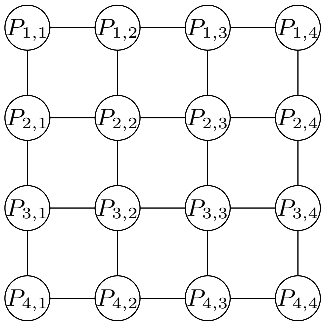


Figure 15.12 A square of size 4×4.

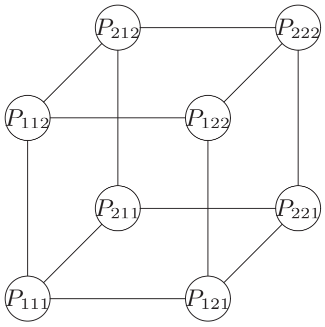


Figure 15.13 A 3-dimensional cube of size $2 \times 2 \times 2$.

If $d = 2$, then we get a rectangle. If now $a_1 = a_2 = \sqrt{p}$, then we get a ***square.*** Figure 15.12 shows a square of size 4×4.

A square contains several chains consisting of a processors. The processors having identical first index, form a ***row of processors,*** and the processors having the same second index form a ***column of processors.*** Algorithms running on a square often consists of such operations, executed only by processors of some rows or columns.

If $d = 3$, then the corresponding mesh is a brick. In the special case $a_1 = a_2 = a_3 =^3 \sqrt{p}$ the mesh is called ***cube.*** Figure 15.13 shows a cube of size $2 \times 2 \times 2$.

The next model of computation is the ***d-dimensional hypercube*** $\mathcal{H}_d$. This model can be considered as the generalisation of the square and cube: the square represented on Figure 15.12 is a 2-dimensional, and the cube, represented on Figure 15.13 is a 3-dimensional hypercube. The processors of $\mathcal{H}_d$ can be labelled by a binary number consisting of d bits. Two processors of $\mathcal{H}_d$ are connected iff the Hamming-distance of their labels equals to 1. Therefore each processors of $\mathcal{H}_d$ has d neighbours, and the of $\mathcal{H}_d$ is d. Figure 15.14 represents $\mathcal{H}_4$.

The butterfly model $\mathcal{B}_d$ consists of $p = (d + 1)2^d$ processors and $2d^{d+1}$ edges. The processors can be labelled by a pair $\langle r, l \rangle$, where r is the ***columnindex*** and l is

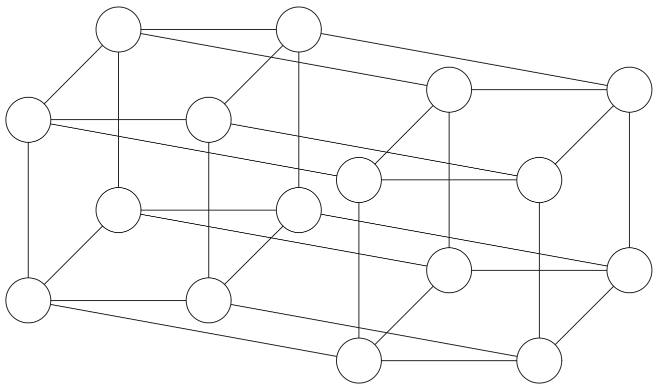


Figure 15.14 A 4-dimensional hypercube $\mathcal{H}_4$.

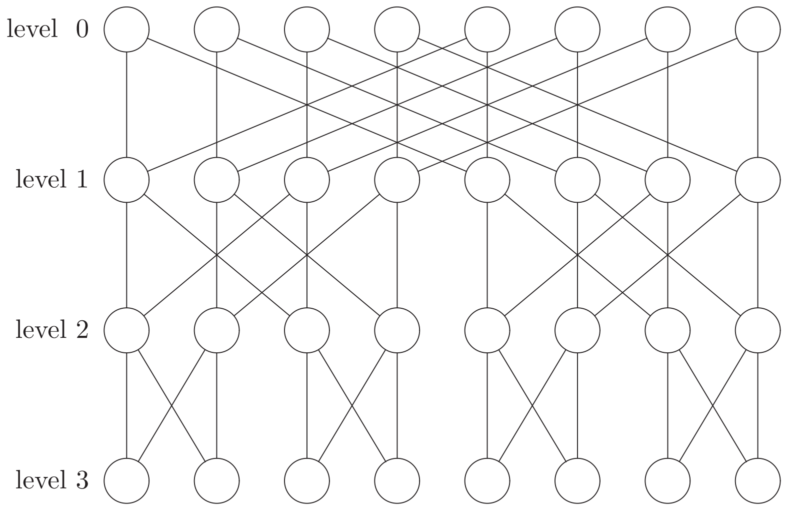


Figure 15.15 A butterfly model.

the ***level*** of the given processor. Figure 15.15 shows a ***butterfly*** model $\mathcal{B}_3$ containing 32 processors in 8 columns and in 4 levels.

Finally Figure 15.16 shows a ***ring*** containing 6 processors.

15.5. Performance in theory

In the previous section we considered the performance measures used in the practice.

In the theoretical investigations the algorithms are tested using abstract computers called computation models.

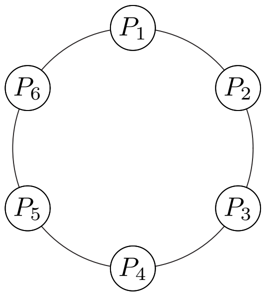


Figure 15.16 A ring consisting of 6 processors.

The required quantity of resources can be characterised using *absolute* and *relative* measures.

Let $W(n, \pi, \mathrm{A})$, resp. $W(n, \pi, p, \mathrm{P})$ denote the time necessary in *worst case* to solve the problem π of size n by the sequential algorithm A, resp. parallel algorithm P (using p processors).

In a similar way let $B(n, \pi, \mathrm{A})$, resp. $B(n, \pi, p, \mathrm{P})$ the time necessary for algorithm A, resp. P in *best case* to solve the problem π of size n (algorithm P can use p processors).

Let $N(n, \pi)$, resp. $N(n, \pi, p)$ the time needed by any sequential, resp. parallel algorithm to solve problem π of size n (algorithm P can use p processors). These times represent a ***lower bound*** of the corresponding running time.

Let suppose the distribution function $D(n, \pi)$ of the problem π of size n is given. Then let $E(n, \pi, \mathrm{A})$, resp. $E(n, \pi, p, \mathrm{P})$ the expected value of the time necessary for algorithm A, resp. P to solve problem π of size n (algorithm P uses p processors).

In the analysis it is often supposed that the input data of equal size have equal probability. For such cases we use the notation $A(n, \mathrm{A})$, resp. $A(n, \mathrm{P}, p)$ and termin ***average running time.***

The value of the performance measures W, B, N, E and A depend on the used computation model too. For the simplicity of notations we suppose that the algorithms determine the computation model.

Usually the context shows in a unique way the investigated problem. If so, then the parameter π is omitted.

Among these performance measures hold the following inequalities:

$$\begin{aligned} N(n) &\leq B(n, \mathrm{A}) && (15.1) \\ &\leq E(n, \mathrm{A}) && (15.2) \\ &\leq W(n, \mathrm{A}) . && (15.3) \end{aligned}$$

In a similar way for the characteristic data of the parallel algorithms the following inequalities are true:

$$\begin{aligned} N(n,p) &\leq B(n,\mathrm{P},p) && (15.4)\\ &\leq E(n,\mathrm{P},p) && (15.5)\\ &\leq W(n,\mathrm{P},p)\ . && (15.6)\end{aligned}$$

For the expected running time we have

$$\begin{aligned} B(n,\mathrm{A}) &\leq A(n,\mathrm{A}) && (15.7)\\ &\leq W(n,\mathrm{A})\ , && (15.8)\end{aligned}$$

and

$$\begin{aligned} B(n,\mathrm{P},p) &\leq A(n,\mathrm{P},p) && (15.9)\\ &\leq W(n,\mathrm{P},p)\ . && (15.10)\end{aligned}$$

These notations can be used not only for the running time, but also for any other resource, as memory requirement, number of messages, etc.

Now we define some ***relative performance measures.***

Speedup shows, how many times is smaller the running time of a parallel algorithm, than the running time of the parallel algorithm solving the same problem.

The ***speedup*** (or *relative number of steps* or *relative speed*) of a given parallel algorithm P, comparing it with a given sequential algorithm A, is defined as

$$g(n,\mathrm{A},\mathrm{P}) = \frac{W(n,\mathrm{A})}{W(n,\mathrm{P},p)}\ . \tag{15.11}$$

If for a sequential algorithm A and a parallel algorithm P holds

$$\frac{W(n,\mathrm{A})}{W(n,p,\mathrm{P})} = \Theta(p)\ , \tag{15.12}$$

then the speedup of P comparing with A is ***linear,*** if

$$\frac{W(n,\mathrm{A})}{W(n,\mathrm{P},p)} = o(p)\ , \tag{15.13}$$

then the speedup of P comparing with A is ***sublinear,*** and if

$$\frac{W(n,\mathrm{A})}{W(n,\mathrm{P},p)} = \omega(p)\ , \tag{15.14}$$

then the speedup of P comparing with A is ***superlinear.***

In the case of parallel algorithms it is a very important performance measure the ***work*** $w(n,p,\mathrm{P})$, defined by the product of the running time and the number of the used processors:

$$w(n,p,\mathrm{P}) = pW(n,\mathrm{P},p)\ . \tag{15.15}$$

This definition is used even then if some processors work only in a small fraction of the running time. Therefore the real work can be much smaller, then given by the formula 15.15).

The ***efficiency*** $h(n, p, \mathrm{P}, \mathrm{A})$ is a measure of the fraction of time for which the processors are usefully employed; it is defined as the ratio of the work of the sequential algorithm to the work of the parallel algorithm P:

$$e(n, p, \mathrm{P}, \mathrm{A}) = \frac{W(n, A)}{pW(n, \mathrm{P}, p)} \ . \tag{15.16}$$

One can observe, that the ratio of the speedup and the number of the used parallel processors results the same value. If the parallel work is not less than the sequential one, then efficiency is between zero and one, and the relatively large values are beneficial.

In connection with the analysis of the parallel algorithms the work-efficiency is a central concept. If for a parallel algorithm P and sequential algorithm A holds

$$pW(n, \mathrm{P}, p) = O(W(n, \mathrm{A})) \ , \tag{15.17}$$

then algorithm P ***work-optimal*** comparing with A.

This definition is equivalent with the equality

$$\frac{pW(n, \mathrm{P}, p)}{W(n, \mathrm{A})} = O(1). \tag{15.18}$$

According to this definition a parallel algorithm is work-optimal only if the order of its total work is not greater, than the order of the total work of the considered sequential algorithm.

A weaker requirement is the following. If there exists a finite positive integer k such that

$$pW(n, \mathrm{P}, p) = O(W(n, \mathrm{A}(\lg n)^k) \ , \tag{15.19}$$

then algorithm P is ***work-efficient*** comparing with A.

If a sequential algorithm A, resp. a parallel algorithm P uses only $O(N(n))$, resp. $O(N(n, p))$ units of a given resource, then A, resp. P is called—for the given resource and the considered model of computation—***asymptotically optimal.***

If an A sequential or a P parallel algorithm uses only the necessary amount of some resource for all possible size $n \geq 1$ of the input, that is $N(n, \mathrm{A})$, resp. $N(n, p, \mathrm{A})$ units, and so we have

$$W(n, \mathrm{A}) = N(n, \mathrm{A}) \ , \tag{15.20}$$

for A and

$$W(n, \mathrm{P}, p) = N(n, \mathrm{P}, p) \ , \tag{15.21}$$

for P, then we say, that the given algorithm is ***absolute optimal*** for the given resource and the given computation model. In this case we say, that $W(n, \mathrm{P}, p) = N(n, \mathrm{P}, p)$ is the ***accurate complexity*** of the given problem.

Comparing two algorithms and having

$$W(n, \mathrm{A}) = \Theta(W(n, \mathrm{B})) \tag{15.22}$$

we say, that the speeds of the growths of algorithms A and B ***asymptotically have the same order.***

Comparing the running times of two algorithms A and B (e.g. in worst case) sometime the estimation depends on n: for some values of n algorithm A, while for other values of n algorithm B is the better. A possible formal definition is as follows. If the functions $f(n)$ and $g(n)$ are defined for all positive integer n, and for some positive integer v hold

1. $f(v) = g(v)$;
2. $(f(v-1) - g(v-1))(f(v+1) - g(v+1)) < 0$,

then the number v is called ***crossover point*** of the functions $f(n)$ and $g(n)$.

For example multiplying two matrices according to the definition and algorithm of Strassen we get one crossover point, whose value is about 20.

Exercises

15.5-1 Suppose that the parallel algorithms P and Q solve the selection problem. Algorithm P uses $n^{0.5}$ processors and its running time is $W(n, \mathrm{P}, p) = \Theta(n^{0.5})$. Algorithm Q uses n processors and its running time is $W(n, \mathrm{P}, p) = \Theta(\lg n)$. Determine the work, speedup and efficiency for both algorithms. Are these algorithms work-optimal or at least work-efficient?

15.5-2 Analyse the following two assertions.

a) Running time of algorithm P is at least $O(n^2)$.

b) Since the running time of algorithm P is $O(n^2)$, and the running time of algorithm B is $O(n \lg n)$, therefore algorithm B is more efficient.

15.5-3 Extend the definition of the crossover point to noninteger v values and parallel algorithms.

15.6. PRAM algorithms

In this section we consider parallel algorithms solving simple problems as prefix calculation, ranking of the elements of an array, merging, selection and sorting.

In the analysis of the algorithms we try to give the accurate order of the running time in the worst case and try to decide whether the presented algorithm is work-optimal or at least work-efficient or not. When parallel algorithms are compared with sequential algorithms, always the best known sequential algorithm is chosen.

To describe these algorithms we use the following pseudocode conventions.

```
P_i  in  parallel for $i \leftarrow 1$ to $p$
     do  ⟨ command 1 ⟩
         ⟨ command 2 ⟩
         .
         .
         .
         ⟨ command u ⟩
```

For m^2 PRAM ordered into a square grid of size $m \times m$ the instruction begin with

```
$P_{i,j}$ in parallel for $i \leftarrow 1$ to $m$, $j \leftarrow 1$ to $m$
          do
```

For a k-dimensional mesh of size $m_1 \times \cdots m_k$ the similar instruction begins with

```
$P_{i_1,i_2,\ldots,i_k}$ in parallel for $i_1 \leftarrow 1$ to $m_1$, ..., $i_k \leftarrow 1$ to $m_k$
          do
```

It is allowed that in this commands P_i represents a group of processors.

15.6.1. Prefix

Let $\oplus$ be a binary associative operator defined over a set Σ. We suppose that the operator needs only one set and the set is closed for this operation.

A binary operation $\oplus$ is *associative* on a Σ set, if for all $x, y, z \in \Sigma$ holds

$$((x \oplus y) \oplus z) = (x \oplus (y \oplus z)) \ . \tag{15.23}$$

Let the elements of the sequence $X = x_1, x_2, \ldots, x_p$ be elements of the set Σ. Then the input data are the elements of the sequence X, and the ***prefix problem*** is the computation of the elements $x_1,\ x_1 \oplus x_2,\ \ldots, x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_p$. These elements are called ***prefixes.***

It is worth to remark that in other topics of parallel computations the starting sequences $x_1, x_2, \ldots, x_k$ of the sequence X are called prefixes.

Example 15.1 *Associative operations.* If Σ is the set of integer numbers, $\oplus$ means addition and the sequence of the input data is $X = 3, -5, 8, 2, 5, 4$, then the sequence of the prefixes is $Y = 3, -2, 6, 8, 13, 17$. If the alphabet and the input data are the same, but the operation is the multiplication, then $Y = 3, -15, -120, -240, -1200, -4800$. If the operation is the minimum (it is also an associative operation), then $Y = 3, -5, -5, -5, -5, -5$. In this case the last prefix is the minimum of the input data.

The prefix problem can be solved by sequential algorithms in $O(p)$ time. Any sequential algorithm A requires $\Omega(p)$ time to solve the prefix problem. There are parallel algorithms for different models of computation resulting a work-optimal solution of the prefix problem.

In this subsection at first the algorithm CREW-Prefix is introduced, which solves the prefix problem in $\Theta(\lg p)$ time, using p CREW PRAM processors.

Next is algorithm EREW-Prefix, having similar quantitative characteristics, but requiring only EREW PRAM processors.

These algorithms solve the prefix problem quicker, then the sequential algorithms, but the order of the necessary work is larger.

Therefore interesting is algorithm Optimal-Prefix, which uses only $\lceil p/\lg p \rceil$ CREW PRAM processors, and makes only $\Theta(\lg p)$ steps. The work of this algorithm is only $\Theta(p)$, therefore its efficiency is $\Theta(1)$, and so it is work-optimal. The speedup of this algorithm equals to $\Theta(n/\lg n)$.

For the sake of simplicity in the further we write usually $p/\lg p$ instead of $\lceil p/\lg p \rceil$.

A CREW PRAM algorithm. As first parallel algorithm a recursive algorithm is presented, which runs on CREW PRAM model of computation, uses p processors and $\Theta(\lg p)$ time. Designing parallel algorithm it is often used the principle *divide-and-conquer*, as we we will see in the case of the next algorithm too

Input is the number of processors (p) and the array $X[1 \mathinner{.\,.} p]$, output data are the array $Y[1 \mathinner{.\,.} p]$. We suppose p is a power of 2. Since we use the algorithms always with the same number of processors, therefore we omit the number of processors from the list of input parameters. In the mathematical descriptions we prefer to consider X and Y as sequences, while in the pseudocodes sometimes as arrays.

```
CREW-Prefix($X$)

1  if $p = 1$
2     then $y_1 \leftarrow x_1$
3          return $Y$
4  if $p > 1$
5     then $P_i$ in parallel for $i \leftarrow 1$ to $p/2$
                do compute recursive $y_1, y_2, \ldots, y_{p/2}$,
                   the prefixes, belonging to $x_1, x_2, \ldots, x_{p/2}$
           $P_i$ in parallel for $i \leftarrow p/2 + 1$ to $p$
                do compute recursive $y_{p/2+1}, y_{p/2+2}, \ldots, y_p$
                   the prefixes, belonging to $x_{p/2+1}, x_{p/2+2}, \ldots, x_p$
6          $P_i$ in parallel for $p/2 + 1 \leq i \leq p$
                do read $y_{p/2}$ from the global memory and compute $y_{p/2} \oplus y_{p/2+i}$
7  return $Y$
```

Example 15.2 *Calculation of prefixes of 8 elements on 8 processors.* Let $n = 8$ and $p = 8$. The input data of the prefix calculation are 12, 3, 6, 8, 11, 4, 5 and 7, the associative operation is the addition.

The run of the *recursive algorithm* consists of *rounds*. In the first round (step 4) the first four processors get the input data 12, 3, 6, 8, and compute recursively the prefixes 12, 15, 21, 29 as output. At the same time the other four processors get the input data 11, 4, 5, 7, and compute the prefixes 11, 15, 20, 27.

According to the recursive structure $P_1, P_2,$ P_3 and P_4 work as follows. P_1 and P_2 get x_1 and x_2, resp. P_3 and P_4 get x_3 and x_4 as input. Recursivity mean for P_1 and P_2, that P_1 gets x_1 and P_2 gets x_2, computing at first $y_1 = x_1$ and $y_2 = x_2$, then P_2 updates $y_2 = y_1 \oplus y_2$. After this P_3 computes $y_3 = y_2 \oplus y_3$ and $y_4 = y_4 \oplus y_4$.

While $P_1, P_2,$ P_3 and P_4, according to step 4, compute the final values $y_1,$ $y_2,$ y_3 and y_4, $P_5, P_6,$ P_7 and P_8 compute the local provisional values of $y_5,$ $y_6,$ y_7 and y_8.

In the second round (step 5) the first four processors stay, the second four processors compute the final values of $y_5,$ $y_6,$ y_7 and y_8, adding $y_4 = 29$ to the provisional values 11, 15, 20 and 27 and receiving 40, 44, 49 and 56.

In the remaining part of the section we use the notation $W(n)$ instead of $W(n,p)$

and give the number of used processors in verbal form. If $p = n$, then we usually prefer to use p.

Theorem 15.1 *Algorithm* CREW-Prefix *uses* $\Theta(\lg p)$ *time on* p CREW PRAM *processors to compute the prefixes of* p *elements.*

Proof The lines 4–6 require $W(p/2)$ steps, the line 7 does $\Theta(1)$ steps. So we get the following recurrence:

$$W(p) = W(p/2) + \Theta(1). \tag{15.24}$$

Solution of this recursive equation is $W(p) = \Theta(\lg p)$. ■

CREW-prefix is not work-optimal, since its work is $\Theta(p \lg p)$ and we know sequential algorithm requiring only $O(p)$ time, but it is work-effective, since all sequential prefix algorithms require $\Omega(p)$ time.

An EREW PRAM algorithm. In the following algorithm we use exclusive write instead of the parallel one, therefore it can be implemented on the EREW PRAM model. Its input is the number of processors p and the sequence $X = x_1,\ x_2,\ \ldots,\ x_p$, and its output is the sequence $Y = y_1,\ y_2,\ \ldots,\ y_p$ containing the prefixes.

```
EREW-Prefix($X$)
1  $Y[1] \leftarrow X[1]$
2  $P_i$ in parallel for $i \leftarrow 2$ to $p$
3     do $Y[i] \leftarrow X[i-1] \oplus X[i]$
4  $k \leftarrow 2$
5  while $k < p$
6          do $P_i$ in parallel for $i \leftarrow k+1$ to $p$
7              do $Y[i] \leftarrow Y[i-k] \oplus Y[i]$
8                  $k \leftarrow k + k$
9  return $Y$
```

Theorem 15.2 *Algorithm* EREW-Prefix *computes the prefixes of* p *elements on* p EREW PRAM *processors in* $\Theta(\lg p)$ *time.*

Proof The commands in lines 1–3 and 9 are executed in $O(1)$ time. Lines 4–7 are executed so many times as the assignment in line 8, that is $\Theta(p)$ times. ■

A work-optimal algorithm. Next we consider a recursive work-optimal algorithm, which uses $p/\lg p$ CREW PRAM processors. Input is the length of the input sequence (p) and the sequence $X = x_1,\ x_2,\ \ldots,\ x_p$, output is the sequence $Y = y_1,\ y_2,\ \ldots,\ y_p$, containing the computed prefixes.

```
Optimal-Prefix($p, X$)
1  $P_i$ in parallel for $i \leftarrow 1$ to $p/\lg p$
2      do compute recursive $z_{(i-1)\lg p+1}, z_{(i-1)\lg p+2}, \ldots, z_{i\lg p}$,
           the prefixes of the following $\lg p$ input data
           $x_{(i-1)\lg p+1}, x_{(i-1)\lg p+2}, \ldots, x_{i\lg p}$
3  $P_i$ in parallel for $i \leftarrow 1$ to $p/\lg p$
4      do using CREW-Prefix compute $w_{\lg p}, w_{2\lg p}, w_{3\lg p}, \ldots, w_p$,
           the prefixes of the following $p/\lg p$ elements:
           $z_{\lg p}, z_{2\lg p}, z_{3\lg p}, \ldots, , z_p$
5  $P_i$ in parallel for $i \leftarrow 2$ to $p/\lg p$
6      do for $j \leftarrow 1$ to $p$
7                do $Y[(i-1)\lg p + j] \leftarrow w_{(i-1)\lg p} \oplus z_{(i-1)\lg p+j}$
8  $P_1$ for $j \leftarrow 1$ to $p$
9            do $Y[j] \leftarrow z_j$
10 return $Y$
```

This algorithm runs in logarithmic time. The following two formulas help to show it:

$$z_{(i-1)\lg p+k} = \sum_{j=(i-1)\lg p+1}^{i\lg p} x_j \quad (k = 1,\ 2,\ \ldots,\ \lg p) \tag{15.25}$$

and

$$w_{i\lg p} = \sum_{j=1}^{i} z_{j\lg p} \ (i = 1,\ 2,\ \ldots,), \tag{15.26}$$

where summing goes using the corresponding associative operation.

Theorem 15.3 (parallel prefix computation in $\Theta(\lg p)$ time). *Algorithm* Optimal-Prefix *computes the prefixes of p elements on $p/\lg p$* CREW PRAM *processors in $\Theta(\lg p)$ time.*

Proof Line 1 runs in $\Theta(\lg p)$ time, line 2 runs $O(\lg(p/\lg p)) = O(\lg p)$ time, line 3 runs $\Theta(\lg p)$ time. ■

This theorem imply that the work of Optimal-Prefix is $\Theta(p)$, therefore Optimal-Prefix is a work-optimal algorithm.

Let the elements of the sequence $X = x_1, x_2, \ldots, x_p$ be the elements of the alphabet Σ. Then the input data of the prefix computation are the elements of the sequence X, and the ***prefix problem*** is the computation of the elements $x_1,\ x_1 \oplus x_2,\ \ldots, x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_p$. These computable elements are called ***prefixes.***

We remark, that in some books on parallel programming often the elements of the sequence X are called prefixes.

Example 15.3 Associative operations. If Σ is the set of integers, $\oplus$ denotes the addition and the sequence of the input data is 3, -5, 8, 2, 5, 4, then the prefixes are 3, -2, 6, 8, 13, 17. If the alphabet and the input data are the same, the operation is the multiplication, then the output data (prefixes) are 3, -15, -120, -240, -1200, -4800. If the operation is the

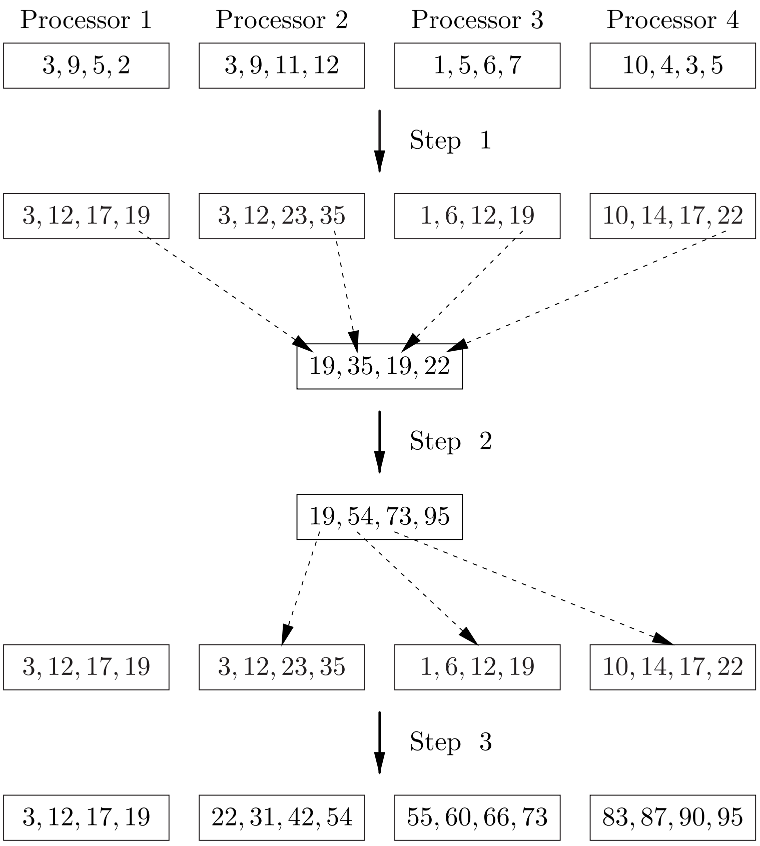


Figure 15.17 Computation of prefixes of 16 elements using OPTIMAL-PREFIX.

minimum (it is also associative), then the prefixes are 3, -5, -5, -5, -5, -5. The last prefix equals to the smallest input data.

Sequential prefix calculation can be solved in $O(p)$ time. Any A sequential algorithm needs $N(p, \mathrm{A}) = \Omega(\mathrm{n})$ time. There exist work-effective parallel algorithms solving the prefix problem.

Our first parallel algorithm is CREW-PREFIX, which uses p CREW PRAM processors and requires $\Theta(\lg p)$ time. Then we continue with algorithm EREW-PREFIX, having similar qualitative characteristics, but running on EREW PRAM model too.

These algorithms solve the prefix problem quicker, than the sequential algorithms, but the order of their work is larger.

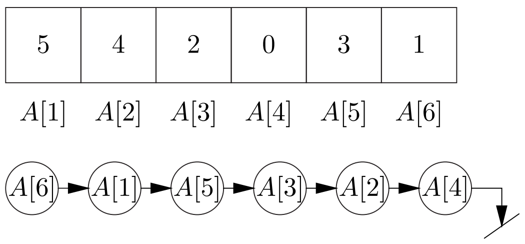


Figure 15.18 Input data of array ranking and the the result of the ranking.

Algorithm Optimal-Prefix requires only $\lceil p/\lg p\rceil$ CREW PRAM processors and in spite of the reduced numbers of processors requires only $O(\lg p)$ time. So its work is $O(p)$, therefore its efficiency is $\Theta(1)$ and is work-effective. The speedup of the algorithm is $\Theta(n/\lg n)$.

15.6.2. Ranking

The input of the ***list ranking problem*** is a list represented by an array $A[1\,..\,p]$: each element contains the index of its ***right neighbour*** (and maybe further data). The task is to determine the rank of the elements. The ***rank*** is defined as the number of the right neighbours of the given element.

Since the further data are not necessary to find the solution, for the simplicity we suppose that the elements of the array contain only the index of the right neighbour. This index is called ***pointer.*** The pointer of the rightmost element equals to zero.

Example 15.4 *Input of list ranking.* Let $A[1\,..\,6]$ be the array represented in the first row of Figure 15.18. Then the right neighbour of the element $A[1]$ is $A[5]$, the right neighbour of $A[2]$ is $A[4]$. $A[4]$ is the last element, therefore its rank is 0. The rank of $A[2]$ is 1, since only one element, $A[4]$ is to right from it. The rank of $A[1]$ is 4, since the elements $A[5], A[3], A[2]$ and $A[4]$ are right from it. The second row of Figure 15.18 shows the elements of A in decreasing order of their ranks.

The list ranking problem can be solved in linear time using a sequential algorithm. At first we determine ***the head of the list*** which is the unique $A[i]$ having the property that does not exist an index j $(1 \leq j \leq p)$ with $A[j] = i$. In our case the head of A is $A[6]$. The head of the list has the rank $p-1$, its right neighbour has a rank $p-2, \ldots$ and finally the rank of the last element is zero.

In this subsection we present a deterministic list ranking algorithm, which uses p EREW PRAM processors and in worst case $\Theta(\lg p)$ time. The pseudocode of algorithm Det-Ranking is as follows.

The input of the algorithm is the number of the elements to be ranked (p), the array $N[1\,..\,p]$ containing the index of the right neighbour of the elements of A, output is the array $R[1\,..\,p]$ containing the computed ranks.

neighbour						*rank*						
5	4	2	0	3	1	1	1	1	0	1	1	(initial state)
3	0	4	0	2	5	2	1	2	0	2	2	$q = 1$
4	0	0	0	0	2	4	1	2	0	3	4	$q = 2$
0	0	0	0	0	0	4	1	2	0	3	5	$q = 3$

Figure 15.19 Work of algorithm Det-Ranking on the data of Example 15.4.

```
Det-Ranking($p, N$)
$P_i$ in parallel for $i \leftarrow 1$ to $p$
   do if $N[i] = 0$
        then $R[i] \leftarrow 0$
        else $R[i] \leftarrow 1$
for $j \leftarrow 1$ to $\lceil \lg p \rceil$
    do $P_i$ in parallel for $i \leftarrow 1$ to $p$
           do if $N[i] \neq 0$
                 then $R[i] \leftarrow R[i] + R[N[i]]$
                      $N[i] \leftarrow N[N[i]]$
return $R$
```

The basic idea behind the algorithm Det-Ranking is the ***pointer jumping.*** According to this algorithm at the beginning each element contains the index of its right neighbour, and accordingly its provisional rank equal to 1 (with exception of the last element of the list, whose rank equals to zero). This initial state is represented in the first row of Figure 15.19.

Then the algorithm modifies the element so, that each element points to the right neighbour of its right neighbour (if it exist, otherwise to the end of the list). This state is represented in the second row of Figure 15.19.

If we have p processors, then it can be done in $O(1)$ time.

After this each element (with exception of the last one) shows to the element whose distance was originally two. In the next step of the pointer jumping the elements will show to such other element whose distance was originally 4 (if there is no such element, then to the last one), as it is shown in the third row of Figure 15.19.

In the next step the pointer part of the elements points to the neighbour of distance 8 (or to the last element, if there is no element of distance 8), according to the last row of Figure 15.19.

In each step of the algorithm each element updates the information on the number of elements between itself and the element pointed by the pointer. Let $R[i]$, resp. $N[i]$ the rank, resp. neighbour field of the element $A[i]$. The initial value of $R[i]$ is 1 for the majority of the elements, but is 0 for the rightmost element ($R(4) = 0$ in the first line of Figure 15.19). During the pointer jumping $R[i]$ gets the new value (if $N[i] \neq 0$) gets the new value $R[i] + R[N[i]]$, if $N[i] \neq 0$. E.g. in the second row of Figure 15.19) $R[1] = 1 + 1 = 2$, since its previous rank is 1, and the rank of its right neighbour is also 1. After this $N[i]$ will be modified to point to $N[N[i]]$. E.g. in the second row of Figure 15.19 $N[1] = 3$, since the right neighbour of the right neighbour of $A[1]$ is $A[3]$.

Theorem 15.4 *Algorithm* Det-Ranking *computes the ranks of an array consisting of p elements on p* EREW PRAM *processors in $\Theta(\lg p)$ time.*

Since the work of Det-Ranking is $\Theta(p \lg p)$, this algorithm is not work-optimal, but it is work-efficient.

The list ranking problem corresponds to a list prefix problem, where each element is 1, but the last element of the list is 0. One can easily modify Det-Ranking to get a prefix algorithm.

15.6.3. Merge

The input of the ***merging problem*** is two sorted sequences X_1 and X_2 and the output is one sorted sequence Y containing the elements of the input.

If the length of the input sequences is p, then the merging problem can be solved in $O(p)$ time using a sequential processor. Since we have to investigate all elements and write them into the corresponding element of Y, the running time of any algorithm is $\Omega(p)$. We get this lower bound even in the case when we count only the number of necessary comparisons.

Merge in logarithmic time. Let $X_1 = x_1, x_2, \ldots, x_m$ and $X_2 = x_{m+1}, x_{m+2}, \ldots, x_{2m}$ be the input sequences. For the shake of simplicity let m be the power of two and let the elements be different.

To merge two sequences of length m it is enough to know the ranks of the keys, since then we can write the keys—using $p = 2m$ processors—into the corresponding memory locations with one parallel write operation. The running time of the following algorithm is a logarithmic, therefore it is called Logarithmic-Merge.

Theorem 15.5 *Algorithm* Logarithmic-Merge *merges two sequences of length m on $2m$* CREW PRAM *processors in $\Theta(\lg m)$ time.*

Proof Let the rank of element x be r_1 (r_2) in X_1 (in X_2). If $x = x_j \in X_1$, then let $r_1 = j$. If we assign a single processor P to the element x, then it can determine, using binary search, the number q of elements in X_2, which are smaller than x. If q is known, then P computes the rank r_j in the union of X_1 and X_2, as $j + q$. If x belongs to X_2, the method is the same.

Summarising the time requirements we get, that using one CREW PRAM processor per element, that is totally $2m$ processors the running time is $\Theta(\lg m)$. ■

This algorithm is not work-optimal, only work-efficient.

Odd-even merging algorithm. This following recursive algorithm ODD-EVEN-MERGE follows the classical *divide-and-conquer* principle.

Let $X_1 = x_1, x_2, \ldots, x_m$ and $X_2 = x_{m+1}, x_{m+2}, \ldots, x_{2m}$ be the two input sequences. We suppose that m is a power of 2 and the elements of the arrays are different. The output of the algorithm is the sequence $Y = y_1, \ldots, y_{2m}$, containing the merged elements. This algorithm requires $2m$ EREW PRAM processors.

```
ODD-EVEN-MERGE($X_1, X_2$)

 1  if $m = 1$
 2     then get $Y$ by merging $x_1$ and $x_2$ with one comparison
 3     return $Y$
 4  if $m > 1$
 5     then $P_i$ in parallel for $i \leftarrow 1$ to $m$
 6               do merge recursively $X_1^{odd} = x_1, x_3, \ldots, x_{m-1}$ and
 7                  $X_2^{odd} = x_{m+1}, x_{m+3}, \ldots, x_{2m-1}$ to get $L_1 = l_1, l_2, \ldots, l_m$
 8          $P_i$ in parallel for $1 \leftarrow m+1$ to $2m$
 9               do merge recursively $X_1^{even} = x_2, x_4, \ldots, x_m$ and
10                  $X_2^{even} = x_{m+2}, x_{m+4}, \ldots, x_{2m}$ to get $L_2 = l_{m+1}, l_{m+2}, \ldots, l_{2m}$
11          $P_i$ in parallel for $i \leftarrow 1$ to $m$
12               do $y_{2i-1} \leftarrow l_i$
13                  $y_{2i} \leftarrow l_{m+i}$
14                  if $y[2i] > y[2i+1]$
13                     then $z \leftarrow y[2i]$
14                          $y[2i] \leftarrow y[2i+1]$
15                          $y[2i+1] \leftarrow z$
15  return $Y$
```

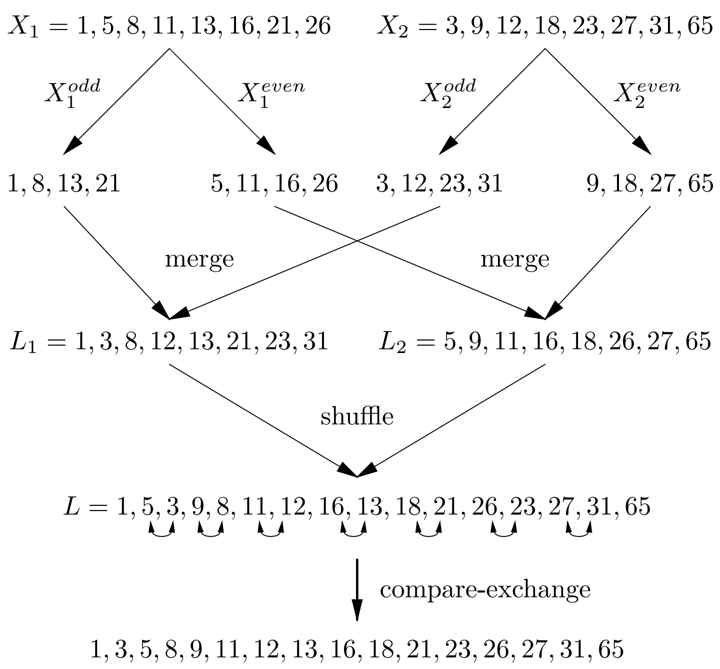
Example 15.5 *Merge of twice eight numbers.* Let X_1 = 1, 5, 8, 11, 13, 16, 21, 26 and X_2 = 3, 9, 12, 18, 23, 27, 31, 65. Figure 15.20 shows the sort of 16 numbers.

At first elements of X_1 with odd indices form the sequence X_1^{odd} and elements with even indices form the sequence X_1^{even}, and in the same way we get the sequences X_2^{odd} and X_2^{even}. Then comes the recursive merge of the two odd sequences resulting L_1 and the recursive merge of the even sequences resulting L_2.

After this ODD-EVEN-MERGE shuffles L_1 and L_2, resulting the sequence $Y = y_1, \ldots, y_{2m}$: the elements of Y with odd indices come from L_1 and the elements with even indices come from L_2.

Finally we compare the elements of Y with even index and the next element (that is $Y[2]$ with $Y[3]$, $Y[4]$ with $Y[5]$ etc.) and if necessary (that is they are not in the good order) they are changed.

Theorem 15.6 (merging in $\Theta(\lg m)$ time). *Algorithm* ODD-EVEN-MERGE *merges two sequences of length m elements in $\Theta(\lg m)$ time using* $2m$ EREW PRAM *processors.*



$X_1 = 1, 5, 8, 11, 13, 16, 21, 26$
$X_2 = 3, 9, 12, 18, 23, 27, 31, 65$
X_1^{odd}
X_1^{even}
X_2^{odd}
X_2^{even}
$1, 8, 13, 21$
$5, 11, 16, 26$
$3, 12, 23, 31$
$9, 18, 27, 65$
merge
merge
$L_1 = 1, 3, 8, 12, 13, 21, 23, 31$
$L_2 = 5, 9, 11, 16, 18, 26, 27, 65$
shuffle
$L = 1, 5, 3, 9, 8, 11, 12, 16, 13, 18, 21, 26, 23, 27, 31, 65$
compare-exchange
$1, 3, 5, 8, 9, 11, 12, 13, 16, 18, 21, 23, 26, 27, 31, 65$

Figure 15.20 Sorting of 16 numbers by algorithm Odd-Even-Merge.

Proof Let denote the running time of the algorithm by $W(m)$. Step 1 requires $\Theta(1)$ time, Step 2 $m/2$ time. Therefore we get the recursive equation

$$W(m) = W(m/2) + \Theta(1), \tag{15.27}$$

having the solution $W(m) = \Theta(\lg m)$. ■

We prove the correctness of this algorithm using the ***zero-one principle.***

A comparison-based sorting algorithm is ***oblivious,*** if the sequence of comparisons is fixed (elements of the comparison do not depend on the results of the earlier comparisons). This definition means, that the sequence of the pairs of elements to be compared (i_1, j_1), $(i_2, j_2), \ldots,$ (i_m, j_m) is given.

Theorem 15.7 (zero-one principle). *If a simple comparison-based sorting algorithm correctly sorts an arbitrary 0-1 sequence of length n, then it sorts also correctly any sequence of length n consisting of arbitrary keys.*

Proof Let A be a comparison-based oblivious sorting algorithm and let S be such a sequence of elements, sorted incorrectly by A. Let suppose A sorts in increasing order the elements of S. Then the incorrectly sorted sequence S' contains an element x on the i-th $(1 \leq i \leq n-1)$ position in spite of the fact that S contains at least i keys smaller than x.

Let x be the first (having the smallest index) such element of S. Substitute in the input sequence the elements smaller than x by 0's and the remaining elements by 1's. This modified sequence is a 0-1 sequence therefore A sorts it correctly. This observation implies that in the sorted 0-1 sequence at least i 0's precede the 1, written on the place of x.

Now denote the elements of the input sequence smaller than x by red colour, and the remaining elements by blue colour (in the original and the transformed sequence too). We can show by induction, that the coloured sequences are identical at the start and remain identical after each comparison. According to colours we have three types of comparisons: blue-blue, red-red and blue-red. If the compared elements have the same colour, in both cases (after a change or not-change) the colours remain unchanged. If we compare elements of different colours, then in both sequences the red element occupy the position with smaller index. So finally we get a contradiction, proving the assertion of the theorem. ■

Example 15.6 *A non comparison-based sorting algorithm.* Let $x_1, x_2, \ldots, x_n$ be a bit sequence. We can sort this sequence simply counting the zeros, and if we count z zeros, then write z zeros, then $n - z$ ones. Of course, the general correctness of this algorithm is not guaranteed. Since this algorithm is not comparison-based, therefore this fact does not contradict to the zero-one principle.

But merge is sorting, and Odd-Even-Merge is an oblivious sorting algorithm.

Theorem 15.8 *Algorithm* Odd-Even-Merge *sorts correctly sequences consisting of arbitrary numbers.*

Proof Let X_1 and X_2 sorted 0-1 sequences of length m. Let q_1 (q_2) the number of zeros at the beginning of X_1 (X_2). Then the number of zeros in X_1^{odd} equals to $\lceil q_1/2 \rceil$, while the number of zeros in X_1^{even} is $\lfloor q_1/2 \rfloor$. Therefore the number of zeros in L_1 equals to $z_1 = \lceil q_1/2 \rceil + \lceil q_2/2 \rceil$ and the number of zeros in L_2 equals to $z_2 = \lfloor q_1/2 \rfloor + \lfloor q_2/2 \rfloor$.

The difference of z_1 and z_2 is at most 2. This difference is exactly then 2, if q_1 and q_2 are both odd numbers. Otherwise the difference is at most 1. Let suppose, that $|z_1 - z_2| = 2$ (the proof in the other cases is similar). In this cases L_1 contains two additional zeros. When the algorithm shuffles L_1 and L_2, L begins with an even number of zeros, end an even number of ones, and between the zeros and ones is a short “dirty" part, 0, 1. After the comparison and change in the last step of the algorithm the whole sequence become sorted. ■

A work-optimal merge algorithm. Algorithm Work-Optimal-Merge uses only $\lceil 2m/\lg m \rceil$ processors, but solves the merging in logarithmic time. This algorithm divides the original problem into $m/\lg m$ parts so, that each part contains approximately $\lg m$ elements.

Let $X_1 = x_1, x_2, \ldots, x_m$ and $X_2 = x_{m+1}, x_{m+2}, \ldots, x_{m+m}$ be the input sequences. Divide X_1 into $M = \lceil m/\lg m \rceil$ parts so, that each part contain at most

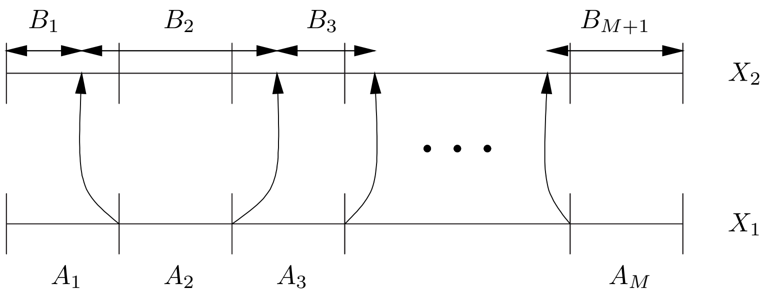


Figure 15.21 A work-optimal merge algorithm Optimal-Merge.

$\lceil \lg m \rceil$ elements. Let the parts be denoted by $A_1, A_2, \ldots, A_M$. Let the largest element in A_1 be l_i $(i = 1, 2, \ldots, M)$.

Assign a processor to each l_i element. These processors determine (by binary search) the correct place (according to the sorting) of l_i in X_2. These places divide X_2 to $M + 1$ parts (some of these parts can be empty). Let denote these parts by B_1, B_2, $\ldots$, B_{M+1}. We call B_i the subset corresponding to A_i in X_2 (see Figure 15.21).

The algorithm gets the merged sequence merging at first A_1 with B_1, A_2 with B_2 and so on, and then joining these merged sequences.

Theorem 15.9 *Algorithm* Optimal-Merging *merges two sorted sequences of length m in $O(\lg m)$ time on* $\lceil 2m/\lg m \rceil$ CREW PRAM *processors.*

Proof We use the previous algorithm.

The length of the parts A_i is $\lg m$, but the length of the parts B_i can be much larger. Therefore we repeat the partition. Let A_i, B_i an arbitrary pair. If $|B_i| = O(\lg m)$, then A_i and B_i can be merged using one processor in $O(\lg m)$ time. But if $|B_i| = \omega(\lg m)$, then divide B_i into $|B_i|/\lg m$ parts—then each part contains at most $\lg m$ keys. Assign a processor to each part. This assigned processor finds the subset corresponding to this subsequence in A_i: $O(\lg\lg m)$ time is sufficient to do this. So the merge of A_i and B_i can be reduced to $|B_i|/\lg m$ subproblems, where each subproblem is the merge of two sequences of $O(\lg m)$ length.

The number of the used processors is $\sum_{i=1}^{M} \lceil |B_i|/\lg m \rceil$, and this is at most $m/\lg m + M$, what is not larger then $2M$. ■

This theorem imply, that Optimal-Merging is work-optimal.

Corollary 15.10 Optimal-Merging *is work-optimal.*

15.6.4. Selection

In the ***selection problem*** $n \geq 2$ elements and a positive integer i $(1 \leq i \leq n)$ are given and the i-th smallest element is to be selected.

Since selection requires the investigation of all elements, and our operations can handle at most two elements, so $N(n) = \Omega(n)$.

Since it is known sequential algorithm A requiring only $W(n, \mathcal{A}) = O(n)$ time, so A is asymptotically optimal.

The ***search problem*** is similar: in that problem the algorithm has to decide, whether a given element appears in the given sequence, and if yes, then where. Here negative answer is also possible and the features of any element decide, whether it corresponds the requirements or not.

We investigate three special cases and work-efficient algorithms to solve them.

Selection in constant time using n^2 processors. Let $i = n$, that is we wish to select the largest key. Algorithm Quadratic-Select solves this task in $\Theta(1)$ time using n^2 CRCW processors.

The input (n different keys) is the sequence $X = x_1, x_2, \ldots, x_n$, and the selected largest element is returned as y.

Quadratic-Select(X)

```
 1  if $n = 1$
 2     then $y \leftarrow x_1$
 3           return $y$
 4  $P_{ij}$ in parallel for $i \leftarrow 1$ to $n$, $j \leftarrow 1$ to $n$
        do if $k_i < k_j$
 5            then $x_{i,j} \leftarrow$ FALSE
 6            else  $x_{i,j} \leftarrow$ TRUE
 7  $P_{i1}$ in parallel for $i \leftarrow 1$ to $n$
 8      do $L_i \leftarrow$ TRUE
 9  $P_{ij}$ in parallel for $i \leftarrow 1$ to $n$, $j \leftarrow 1$ to $n$
10      if $x_{i,j}$ = FALSE
11         then $L_i \leftarrow$ FALSE
12  $P_{i1}$ in parallel for $i \leftarrow 1$ to $n$
13      do if $L_i$ = TRUE
14            then $y \leftarrow x_i$
15  return $y$
```

In the first round (lines 4–6) the keys are compared in parallel manner, using all the n^2 processors. P_{ij} $(1 \leq i, j \leq n)$ so, that processor P_{ij} computes the logical value $x_{i,j} = x_i < x_j$. We suppose that the keys are different. If the elements are not different, then we can use instead of x_i the pair (x_i, i) (this solution requires an additional number of length $(\lg n)$ bits. Since there is a unique key for which all comparison result FALSE, this unique key can be found with a logical OR operation is lines 7–11.

Theorem 15.11 (selection in $\Theta(1)$ time). *Algorithm* Quadratic-Select *determines the largest key of n different keys in $\Theta(1)$ time using n^2* CRCW *common* PRAM *processors.*

Proof First and third rounds require unit time, the second round requires $\Theta(1)$ time, so the total running time is $\Theta(1)$. ■

The speedup of this algorithm is $\Theta(n)$. The work of the algorithm is $w = \Theta(n^2)$. So the efficiency is $E = \Theta(n)/\Theta(n^2) = \Theta(1/n)$. It follows that this algorithm is not work-optimal, even it is not work-effective.

Selection in logarithmic time on n processors. Now we show that the maximal element among n keys can be found, using even only n common CRCW PRAM processors and $\Theta(\lg \lg n)$ time. The used technique is the *divide-and-conquer*. For the simplicity let n be a square number.

The input and the output are the same as at the previous algorithm.

```
Quick-Selection($X, y$)
1  if $p = 1$
2     then $y \leftarrow x_1$
3          return $y$
4  if $p > 1$
5     then divide the input into groups $G_1, G_2, \dots, G_a$ and
           divide the processors into groups $Q_1, Q_2, \dots, Q_a$
6  $Q_i$ in parallel for $i \leftarrow 1$ to $a$
6      do recursively determines the maximal element $M_i$ of the group $G_i$
7  Quadratic-Select($M$)
8  return $y$
```

The algorithm divides the input into $\sqrt{p} = a$ groups $(G_1, G_2, \dots, G_a)$ so, that each group contains a elements $(x_{(i-1)a+1}, x_{(i-1)a+2}, \dots, x_{ia})$, and divides the processors into a groups $(Q_1, Q_2, \dots, Q_a)$ so, that group Q_i contains a processors $P_{(i-1)a+1}, P_{(i-1)a+2}, \dots, P_{ia}$. Then the group of processors Q_i computes recursively the maximum M_i of group G_i. Finally the previous algorithm Quadratic-Select gets as input the sequence $M = M_1, \dots, M_a$ and finds the maximum y of the input sequence X.

Theorem 15.12 (selection in $\Theta(\lg \lg p)$ time). *Algorithm* Quick-Select *determines the largest of p different elements in $O(\lg \lg p)$ time using n common* CRCW PRAM *processors.*

Proof Let the running time of the algorithm denoted by $W(n)$. Step 1 requires $W(\sqrt{n})$ time, step 2 requires $\Theta(1)$ time. Therefore $W(p)$ satisfies the recursive equation

$$W(p) = W(\sqrt{p}) + \Theta(1), \tag{15.28}$$

having the solution $\Theta(\lg \lg p)$. ■

The total work of algorithm Quick-Select is $\Theta(p \lg \lg p)$, so its efficiency is $\Theta(p)/\Theta(p \lg \lg p) = \Theta(1/ \lg \lg p)$, therefore Quick-Select is not work-optimal, it is only work-effective.

Selection from integer numbers. If the problem is to find the maximum of n keys when the keys consist of one bit, then the problem can be solved using a logical

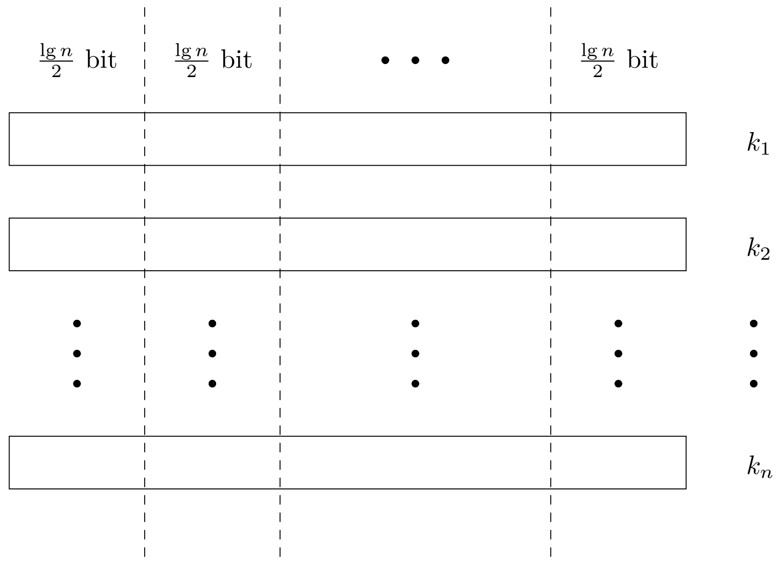


Figure 15.22 Selection of maximal integer number.

OR operation, and so requires only constant time using n processors. Now we try to extend this observation.

Let c be a given positive integer constant, and we suppose the keys are integer numbers, belonging to the interval $[0, n^c]$. Then the keys can be represented using at most $c \lg n$ bits. For the simplicity we suppose that all the keys are given as binary numbers of length $c \lg n$ bits.

The following algorithm Integer-Selection requires only constant time and n CRCW PRAM processors to find the maximum.

The basic idea is to partition the $b_1, b_2, \ldots, b_{2c}$ bits of the numbers into parts of length $(\lg n)/2$. The i-th part contains the bits $b_{(i-1)+1}$, $b_{(i-1)+2}, \ldots, b_{(i-1)+b_{(i-1)+(\lg n)/2}}$, the number of the parts is $2c$. Figure 15.22 shows the partition.

The input of Integer-Selection is the number of processors (n) and the sequence $X = x_1,\ x_2,\ \ldots,\ x_n$ containing different integer numbers, and output is the maximal number y.

Integer-Selection(p, X)

```
1  for $i \leftarrow 1$ to $2c$
2      do compute the maximum ($M$) of the remaining numbers on the base of
          their $i$-th part
3         delete the numbers whose $i$-th part is smaller than $M$
4  $y \leftarrow$ one of the remaining numbers
5  return $y$
```

The algorithm starts with searching the maximum on the base of the first part of the numbers. Then it delete the numbers, whose first part is smaller, than the maximum. Then this is repeated for the second, ..., last part of the numbers. Any of the non deleted numbers is maximal.

Theorem 15.13 (selection from integer numbers). *If the numbers are integers drawn from the interval $[0, n^c]$, then algorithm* Integer-Selection *determines the largest number among n numbers for any positive c in $\Theta(1)$ time using n* CRCW PRAM *processors.*

Proof Let suppose that we start with the selection of numbers, whose $(\lg n)/2$ most significant bits are maximal. Let this maximum in the first part denoted by M. It is sure that the numbers whose first part is smaller than M are not maximal, therefore can be deleted. If we execute this basis operation for all parts (that is $2c$ times), then exactly those numbers will be deleted, what are not maximal, and all maximal element remain.

If a key contains at most $(\lg n)/2$ bits, then its value is at most $\sqrt{n} - 1$. So algorithm Integer-Select in its first step determines the maximum of integer numbers taken from the interval $[0, \sqrt{n} - 1]$. The algorithm assigns a processor to each number and uses $\sqrt{n}$ common memory locations $(M_1, M_2, \ldots, M_{\sqrt{n}-1})$, containing initially $-\infty$. In one step processor P_i writes k_i into M_{k_i}. Later the maximum of all numbers can be determined from $\sqrt{n}$ memory cells using n processors by Theorem 15.11 in constant time. ■

General selection. Let the sequence $X = x_1, x_2, \ldots, x_n$ contain different numbers and the problem is to select the kth smallest element of X. Let we have $p = n^2/\lg n$ CREW processors.

```
General-Selection($X$)
1 divide the $n^2/\lg n$ processors into $n$ groups $G_1, \ldots, G_n$ so, that group $G_i$
    contains the processors $P_{i,1}, P_{i,2}, \ldots, P_{i,n/\lg n}$ and divide
    the $n$ elements into $n/\lg n$ groups $(X_1, X_2, \ldots, X_{n/\lg n})$ so, that group $X_i$
    contains the elements $x_{(i-1)\lg n)+1}, x_{(i-1)\lg n)+2}, \ldots, x_{(i-1)\lg n)+\lg n}$
2 $P_{ij}$ in parallel for $i \leftarrow 1$ to $n$
3     do determine $h_{ij}$ (how many elements of $X_j$ are smaller, than $x_i$)
4 $G_i$ in parallel for $i \leftarrow 1$ to $n$
5     do using Optimal-Prefix determine $s_i$
        (how many elements of $X$ are smaller, than $x_i$)
6 $P_{i,1}$ in parallel for $i \leftarrow 1$ to $n$
7     do if $s_i = k - 1$
8           then return $x_i$
```

Theorem 15.14 (general selection). *The algorithm* General-Selection *determines the i-th smallest of n different numbers in $\Theta(\lg n)$ time using $n^2/\lg n$ processors.*

Proof In lines 2–3 P_{ij} works as a sequential processor, therefore these lines require $\Theta \lg n$ time. Lines 4–5 require $\Theta \lg n$ time according to Theorem 15.3. Lines 6–8 can be executed in constant time, so the total running time is $\Theta(\lg n)$. ■

The work of General-Selection is $\Theta(n^2)$, therefore this algorithm is not work-effective.

15.6.5. Sorting

Given a sequence $X = x_1, x_2, \ldots, x_n$ the ***sorting problem*** is to rearrange the elements of X e.g. in increasing order.

It is well-known that any A sequential comparison-based sorting algorithm needs $N(n, A) = \Omega(n \lg n)$ comparisons, and there are comparison-based sorting algorithms with $O(n \lg n)$ running time.

There are also algorithms, using special operations or sorting numbers with special features, which solve the sorting problem in linear time. If we have to investigate all elements of X and permitted operations can handle at most 2 elements, then we get $N(n) = \Omega(n)$. So it is true, that among the comparison-based and also among the non-comparison-based sorting algorithms are asymptotically optimal sequential algorithms.

In this subsection we consider three different sorting algorithm.

Sorting in logarithmic time using n^2 processors. Using the ideas of algorithms Quadratic-Selection and Optimal-Prefix we can sort n elements using n^2 processors in $\lg n$ time.

```
Quadratic-Sort($K$)

 1  if $n = 1$
 2     then $y \leftarrow x_1$
 3            return $Y$
 4  $P_{ij}$ in parallel for $i \leftarrow 1$ to $n$, $j \leftarrow 1$ to $n$
        do if $x_i < x_j$
 5             then $x_{i,j} \leftarrow 0$
 6             else  $x_{i,j} \leftarrow 1$
 7  divide the processors into $n$ groups $(G_1, G_2, \ldots, G_n)$ so, that group $G_i$ contains
            processors $P_{i,1}, P_{i,2}, \ldots, P_{i,n}$
 8  $G_i$ in parallel for $i \leftarrow 1$ to $n$
 9     do compute $s_i = x_{i,1} + x_{i,2} + \cdots + x_{i,n}$
10  $P_{i1}$ in parallel for $i \leftarrow 1$ to $n$
11      do $y_{s_i+1} \leftarrow x_i$
12  return $Y$
```

In lines 4–7 the algorithm compares all pairs of the elements (as Quadratic-Selection), then in lines 7–9 (in a similar way as Optimal-Prefix works) it counts, how many elements of X is smaller, than the investigated x_i, and finally in lines 10–12 one processor of each group writes the final result into the corresponding memory cell.

Theorem 15.15 (sorting in $\Theta(\lg n)$ time). *Algorithm* Quadratic-Sort *sorts n elements using n^2* CRCW PRAM *processors in $\Theta(\lg n)$ time.*

Proof Lines 8–9 require $\Theta(\lg n)$ time, and the remaining lines require only constant time. ■

Since the work of Quadratic-Sort is $\Theta(n^2 \lg n)$, this algorithm is not work-effective.

Odd-even algorithm with $O(\lg n)$ running time. The next algorithm uses the Odd-Even-Merge algorithm and the classical *divide-and-conquer* principle. The input is the sequence $X = x_1, \ldots, x_p$, containing the numbers to be sorted, and the output is the sequence $Y = y_1, \ldots, y_p$, containing the sorted numbers.

```
Odd-Even-Sort($X$)
 1  if $n = 1$
 2     then $Y \leftarrow X$
 3  if $n > 1$
 4     then let $X_1 = x_1, x_2, \ldots, x_{n/2}$ and $X_2 = x_{n/2+1}, x_{n/2+2}, \ldots, x_n$.
 5     $P_i$ in parallel for $i \leftarrow 1$ to $n/2$
 6         do sort recursively $X_1$ to get $Y_1$
 7     $P_i$ in parallel for $i \leftarrow n/2 + 1$ to $n$
 8         do sort recursively $X_2$ to get $Y_2$
 9     $P_i$ in parallel for $i \leftarrow 1$ to $n$
10         do merge $Y_1$ and $Y_2$ using Odd-Even-Merge($Y_1, Y_2$)
11  return $Y$
```

The running time of this EREW PRAM algorithm is $O(\lg^2 n)$.

Theorem 15.16 (sorting in $\Theta(\lg^2 n)$ time). *Algorithm* Odd-Even-Sort *sorts n elements in $\Theta(\lg^2 n)$ time using n* EREW PRAM *processors.*

Proof Let $W(n)$ be the running time of the algorithm. Lines 3–4 require $\Theta(1)$ time, Lines 5–8 require $W(n/2)$ time, and lines 9–10 require $\Theta(\lg n)$ time, line 11 require $\Theta(1)$ time. Therefore $W(n)$ satisfies the recurrence

$$W(n) = \Theta(1) + W(n/2) + \Theta(\lg n), \tag{15.29}$$

having the solution $W(n) = \Theta(\lg^2 n)$. ■

Example 15.7 *Sorting on 16 processors.* Sort using 16 processors the following numbers: 62, 19, 8, 5, 1, 13, 11, 16, 23, 31, 9, 3, 18, 12, 27, 34. At first we get the odd and even parts, then the first 8 processors gets the sequence $X_1 = 62, 19, 8, 5, 1, 13, 11, 16$, while the other 8 processors get $X_2 = 23, 31, 9, 4, 18, 12, 27, 34$.

The output of the first 8 processors is $Y_1 = 1, 5, 8, 11, 13, 16, 19, 62$, while the output of the second 8 processors is $Y_2 = 3, 9, 12, 18, 23, 27, 31, 34$. The merged final result is $Y = 1, 3, 5, 8, 9, 11, 12, 13, 16, 18, 19, 23, 27, 31, 34, 62$.

The work of the algorithm is $\Theta(n \lg^2 n)$, its efficiency is $\Theta(1/\lg n)$, and its speedup is $\Theta(n/\lg n)$. The algorithm is not work-optimal, but it is work-effective.

Algorithm of Preparata with $\Theta(\lg n)$ running time. If we have more processors, then the running time can be decreased. The following recursive algorithm due to Preparata uses $n \lg n$ CREW PRAM processors and $\lg n$ time. Input is the sequence $X = x_1, x_2, \ldots, x_n$, and the output is the sequence $Y = y_1, y_2, \ldots, y_n$ containing the sorted elements.

```
PREPARATA(X)
 1  if $n \le 20$
 2     then sort $X$ using $n$ processors and ODD-EVEN-SORT
 3     return $Y$
 4  divide the $n$ elements into $\lg n$ parts $(X_1,\ X_2,\ \ldots,\ X_{\lg n})$ so, that each part
    contains $n/\lg n$ elements, and divide the processors into $\lg n$ groups
    $(G_1,\ G_2,\ \ldots,\ G_n)$ so, that each group contains $n$ processors
 5  $G_i$ in parallel for $i \leftarrow 1$ to $\lg n$
 6      do sort the part $X_i$ recursively to get a sorted sequence $S_i$
 7           divide the processors into $(\lg n)^2$ groups $(H_{1,1}, H_{1,2}, \ldots, H_{(\lg n, \lg n)})$
        containing $n/\lg n$ processors
 8  $H_{i,j}$ in parallel for $i \leftarrow 1$ to $\lg n, j \leftarrow 1$ to $\lg n$
 9        do merge $S_i$ and $S_j$
10  divide the processors into $n$ groups $(J_1,\ J_2, \ldots, J_n)$ so, that each group
    contains $\lg n$ processors
11  $J_i$ in parallel for $i \leftarrow 1$ to $n$
12     do determine the ranks of the $x_i$ element in $X$ using the local ranks
            received in line 9 and using the algorithm OPTIMAL-PREFIX
13          $Y_i \leftarrow$ the elements of $X$ having a rank $i$
14  return $Y$
```

This algorithm uses the *divide-and-conquer* principle. It divides the input into $\lg n$ parts, then merges each pair of parts. This merge results local ranks of the elements. The global rank of the elements can be computed summing up these local ranks.

Theorem 15.17 (sorting in $\Theta(\lg n)$ time). *Algorithm* PREPARATA *sorts* n *elements in* $\Theta(\lg n)$ *time using* $n \lg n$ CREW PRAM *processors.*

Proof Let the running time be $W(n)$. Lines 4–6 require $W(n/\lg n)$ time, lines 7–12 together $\Theta(\lg \lg n)$. Therefore $W(n)$ satisfies the equation

$$W(n) = W(n/\lg n) + \Theta(\lg \lg n), \tag{15.30}$$

having the solution $W(n) = \Theta(\lg n)$. ■

The work of Preparata is the same, as the work of Odd-Even-Sort, but the speedup is better: $\Theta(n)$. The efficiency of both algorithms is $\Theta(1/\lg n)$.

Exercises

15.6-1 The memory cell M_1 of the global memory contains some data. Design an algorithm, which copies this data to the memory cells $M_2, M_3, \ldots, M_n$ in $O(\lg n)$ time, using n EREW PRAM processors.

15.6-2 Design an algorithm which solves the previous Exercise 15.6-1 using only $n/\lg n$ EREW PRAM processors saving the $O(\lg n)$ running time.

15.6-3 Design an algorithm having $O(\lg\lg n)$ running time and determining the maximum of n numbers using $n/\lg\lg n$ common CRCW PRAM processors.

15.6-4 Let X be a sequence containing n keys. Design an algorithm to determine the rank of any $k \in X$ key using $n/\lg n$ CREW PRAM processors and $O(\lg n)$ time.

15.6-5 Design an algorithm having $O(1)$ running time, which decides using n common CRCW PRAM processors, whether element 5 is contained by a given array $A[1\,..\,n]$, and if is contained, then gives the largest index i, for which $A[i] = 5$ holds.

15.6-6 Design algorithm to merge two sorted sequence of length m in $O(1)$ time, using n^2 CREW PRAM processors.

15.6-7 Determine the running time, speedup, work, and efficiency of all algorithms, discussed in this section.

15.7. Mesh algorithms

To illustrate another model of computation we present two algorithms solving the prefix problem on meshes.

15.7.1. Prefix on chain

Let suppose that processor P_i $(i = 1,\ 2,\ \ldots,\ p)$ of the chain $\mathcal{L} = \{P_1, P_2, \ldots, P_p\}$ stores element x_i in its local memory, and after the parallel computations the prefix y_i will be stored in the local memory of P_i.

At first we introduce a naive algorithm. Its input is the sequence of elements $X = x_1,\ x_2,\ \ldots,\ x_p$, and its output is the sequence $Y = y_1,\ y_2,\ \ldots, y_p$, containing the prefixes.

Chain-Prefix(X)

```
1  $P_1$ sends $y_1 = x_1$ to $P_2$
2  $P_i$ in parallel for $i \leftarrow 2$ to $p-1$
3  for $i \leftarrow 2$top$-1$
4      do gets $y_{i-1}$ from $P_{i-1}$, then computes and stores $y_i \leftarrow y_{i-1} \oplus x_i$
          stores $z_i = z_{p-1} \oplus x_p$, and sends $z_i$ to $P_{i+1}$
5  $P_a$ gets $z_{p-1}$ from $P_{p-1}$, then computes and stores $y_a = y_{a-1} \oplus x_a$
```

Saying the truth, this is not a real parallel algorithm.

Theorem 15.18 *Algorithm* Chain-Prefix *determines the prefixes of p elements using a chain $\mathcal{C}_p$ in $\Theta(p)$ time.*

Proof The cycle in lines 2–5 requires $\Theta(p)$ time, line 1 and line 6 requires $\Theta(1)$ time. ■

Since the prefixes can be determined in $O(p)$ time using a sequential processor, and $w(p, p, \text{Chain-Prefix}) = pW(p, p, \text{Chain-Prefix}) = \Theta(p^2)$, so CHAIN-Prefix is not work-effective.

15.7.2. Prefix on square

An algorithm, similar to Chain-Prefix, can be developed for a square too.

Let us consider a square of size $a \times a$. We need an ***indexing*** of the processors. There are many different indexing schemes, but for the next algorithm Square-Prefix sufficient is the one of the simplest solutions, the ***row-major indexing scheme,*** where processor $P_{i,j}$ gets the index $a(i-1)+j$.

The input and the output are the same, as in the case of Chain-Prefix.

The processors $P_{i-1)a+1}$, $P_{(i-1)a+2)}$, ...$P_{(i-1)a)+a}$ form the processor row $R_i (1 \leq i \leq a)$ and the processors P_{a+j}, P_{2a+j}, $\ldots P_{a(a-1)+j}$ form the processor column C_j $(1 \leq j \leq a)$. The input stored by the processors of row R_i is denoted by X_i, and the similar output is denoted by Y_i.

The algorithm works in 3 rounds. In the first round (lines 1–8) processor rows R_i $(1 \leq i \leq a)$ compute the row-local prefixes (working as processors of Chain-Prefix). In the second round (lines 9–17) the column C_a computes the prefixes using the results of the first round, and the processors of this column P_{ja} $(1 \leq j \leq a-1)$ send the computed prefix to the neighbour $P_{(j+1)a)}$. Finally in the third round the rows R_i $(2 \leq i \leq a)$ determine the final prefixes.

Square-Prefix(X)

```
1  $P_{j,1}$ in parallel for $j \leftarrow 1$ to $a$
2      do sends $y_{j,1} = x_{j,1}$ to $P_{j,2}$
```

```
3   $P_{j,i}$ in parallel for $i \leftarrow 1$ to $a-1$
    4   for $i \leftarrow 2$to$a-1$
5           do gets $y_{j,i-1}$ from $P_{j,i-1}$, then computes and
                stores $y_j, i = y_{j,p-1} \oplus x_{j,p}$, and sends $y_{j,i}$ to $P_{j,i+1}$
7   $P_{j,a}$ in parallel for $j \leftarrow 1$ to $a$
8       do gets $y_{j,a-1}$ from $P_{j,a-1}$, then computes and stores $y_{1,a} = y_{1,a-1} \oplus x_{1,a}$
9   $P_{1,a}$ sends $y_{1,a}$ to $P_{2,a}$
10  $P_{j,a}$ in parallel for $j \leftarrow 2$ to $a-1$
    11  for $j \leftarrow 2$to$a-1$
12      do gets $y_{j-1,a}$ from $P_{j-1,a}$, then computes and stores
            stores $y_{j,a} = y_{j-1,a} \oplus y_{j,a}$, and sends $y_{j,a}$ to $P_{j+1,a}$
13  $P_{a,a}$ gets $y_{a-1,a}$ from $P_{a-1,a}$, then computes and stores $y_{a,a} = y_{a-1,a} \oplus y_{a,a}$
14  $P_{j,a}$ in parallel for $j \leftarrow 1$ to $a-1$
15      do send $y_{j,a}$ to $P_{j+1,a}$
16  $P_{j,a}$ in parallel for $j \leftarrow 2$ to $a$
17      do sends $y_{j,a}$ to $P_{j,a-1}$
18  $P_{j,i}$ in parallel for $i \leftarrow a-1$ downto 2
19      for $j \leftarrow 2$to$a$
20          do gets $y_{j,a}$ from $P_{j,i+1}$, then computes and
21              stores $y_{j,i} = y_{j,i+1} \oplus y_{j,i}$, and sends $y_{j,a}$ to $P_{j,i-1}$
22  $P_{j,1}$ in parallel for $j \leftarrow 2$ to $a-1$
23      do gets $y_{j,a}$ from $P_{j,2}$, then computes and stores $y_{j,1} = y_{j,a} \oplus y_{j,1}$
```

Theorem 15.19 *Algorithm* Square-Prefix *solves the prefix problem using a square of size $a \times a$, major row indexing in $3a + 2 = \Theta(a)$ time.*

Proof In the first round lines 1–2 contain 1 parallel operation, lines 3–6 require $a-1$ operations, and line 8 again 1 operation, that is all together $a+1$ operations. In a similar way in the third round lines 18–23 require $a+1$ time units, and in round 2 lines 9–17 require a time units. The sum of the necessary time units is $3s+2$. ■

Example 15.8 *Prefix computation on square of size* 4×4 Figure 15.23(a) shows 16 input elements. In the first round Square-Prefix computes the row-local prefixes, part (b) of the figure show the results. Then in the second round only the processors of the fourth column work, and determine the column-local prefixes – results are in part (c) of the figure. Finally in the third round algorithm determines the final results shown in part (d) of the figure.

Chapter Notes

Basic sources of this chapter are for architectures and models the book of Leopold [26], and the book of Sima, Fountaine and Kacsuk [31], for parallel programming the

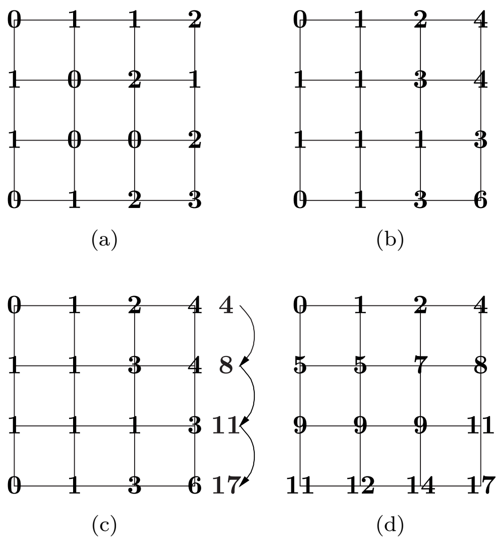


Figure 15.23 Prefix computation on square.

book due to Kumar et al. [13] and [26], for parallel algorithms the books of Berman and Paul, [1] Cormen, Leiserson and Rivest [5], the book written by Horowitz, Sahni and Rajasekaran [16] and the book [18], and the recent book due to Casanova, Legrand and Robert [3].

The website [?] contains the Top 500 list, a regularly updated survey of the most powerful computers worldwide [?]. It contains 42% clusters.

Described classifications of computers are proposed by Flynn [8], and Leopold [26]. The Figures 15.1, 15.2, 15.3, 15.4, 15.5, 15.7 are taken from the book of Leopold [26], the program 15.6 from the book written by Gropp et al. [14].

The clusters are characterised using the book of Pfister [29], grids are presented on the base of the book and manuscript of Foster and Kellerman [9, ?].

With the problems of shared memory deal the book written by Hwang and Xu [17], the book due to Kleiman, Shah, and Smaalders [22], and the textbook of Tanenbaum and van Steen [33].

Details on concepts as tasks, processes and threads can be found in many textbook, e.g. in [30, 32]. Decomposition of the tasks into smaller parts is analysed by Tanenbaum and van Steen [33].

The laws concerning the speedup were described by Amdahl [?], Gustafson-Barsis [15] and Brent [2]. Kandemir, Ramanujam and Choudray review the different methods of the improvement of locality [20]. Wolfe [?] analyses in details the connection between the transformation of the data and the program code. In connection with code optimisation the book published by Kennedy and Allen [21] is a useful source.

The MPI programming model is presented according to Gropp, Snir, Nitzberg, and Lusk [14], while the base of the description of the OpenMP model is the paper due to Chandra, Dragum, Kohr, Dror, McDonald and Menon [4], further a review found on the internet [?].

Lewis and Berg [27] discuss pthreads, while Oaks and Wong [28] the Java threads in details. Description of *High Performance Fortran* can be found in the book Koelbel et al. [23]. Among others Wolfe [?] studied the parallelising compilers.

The concept of PRAM is due to Fortune and Wyllie and is known since 1978 [?]. BSP was proposed in 1990 by Valiant [34]. LogP has been suggested as an alternative of BSP by Culler et al. in 1993 [7]. QSM was introduced in 1999 by Gibbons, Matias and Ramachandran [12].

The majority of the pseudocode conventions used in Section 15.6 and the description of crossover points and comparison of different methods of matrix multiplication can be found in [6].

The Readers interested in further programming models, as skeletons, parallel functional programming, languages of coordination and parallel mobile agents, can find a detailed description in [26]. Further problems and parallel algorithms are analysed in the books of Leighton [24, 25] and in the chapter *Memory Management* of this book [?]. and in the book of Horowitz, Sahni and Rajasekaran [16] A model of scheduling of parallel processes is discussed in [11, 19, 35].

Cost-optimal parallel merge is analysed by Wu and Olariu in [36]. New ideas (as the application of multiple comparisons to get a constant time sorting algoritm) of parallel sorting can be found in the paper of Gararch, Golub, and Kruskal [10].

Bibliography

[1] K. A. Berman, J. L. Paul. *Sequential and Parallel Algorithms*. PWS Publishing Company, 1996. 755

[2] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206, 1974. 755

[3] H. Casanova, A. Legrand, Y. Robert. *Parallel Algorithms*. Chapman & Hall, 2009. 755

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000. 756

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1990. 755

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 2007 (8th corrected printing of 2nd edition). 756

[7] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, T. von Eicken. LogP: A practical model of parallel computation. *Communication of the ACM*, 39(11):78–85, 1996. 756

[8] M. J. Flynn. Very high-speed computer systems. *Proceedings of the IEEE*, 5(6):1901–1909, 1966. 755

[9] I. Foster, C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman Publisher, 2004 (2nd edition). 755

[10] W. Gararch, E. Evan, C. Kruskal. Proofs that yield nothing but their validity or all languages in NP. *Journal of the ACM*, 38(3):691–729, 1991. 756

[11] P. Gács. Compatible sequences and a slow Winkler percolation. *Combinatorics Probability and Computing*, 13(6):815–856, 2004. 756

[12] P. B. Gibbons, Y. Matias, V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation. *Theory of Computing Systems*, 32(3):327–359, 1999. 756

[13] A. Grama, A. Gupta, G. Karypis, V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2003 (2nd edition). 755

[14] W. Gropp, M. Snir, B. Nitzberg, E. Lusk. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. The MIT Press, 1998. 755

[15] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of ACM*, 28(1):532–535, 1988. 755

[16] E. Horowitz, S. Sahni, S. Rajasekaran. *Computer Algorithms*. Computer Science Press, 1998. 755, 756

[17] K. Hwang, Z. Xu. *Scalable Parallel Computing*. McGraw-Hill, 1998. 755

[18] A. Iványi. *Párhuzamos algoritmusok* (*Parallel Algorithms*). ELTE Eötvös Kiadó, 2003. 755

[19] A. Iványi. Density of safe matrices. *Acta Universitatis Sapientiae*, 1(2):121–142, 2009. 756

[20] M. Kandemir, J. Ramanujam, A. Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed-memory machines. *Journal of Parallel and Distributed Computing*, 60:924–965, 2000. 755

[21] K. Kennedy, R. Allen. *Optimizing Compilers for Modern Architectures.* Morgan Kaufman Publishers, 2001. 755

[22] S. Kleiman, D. Shah, B. Smaalders. *Programming with Threads.* Prentice Hall, 1996. 755

[23] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. Steele Jr., M. E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994. 756

[24] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes.* Morgan Kaufman Publishers, 1992. 756

[25] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Algorithms and VSLI.* Morgan Kaufman Publishers, 2001. 756

[26] C. Leopold. *Parallel and Distributed Computing.* Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Copyrights 2001. 754, 755, 756

[27] B. Lewis, D. J. Berg. *Multithreaded Programming with Phtreads.* Prentice Hall, 1998. 756

[28] S. Oaks, H. Wong. *Java Threads.* O'Reilly, 1999. 756

[29] G. F. Pfister. *In Search of Clusters.* Prentice Hall, 1998 (2nd edition). 755

[30] A. Silberschatz, P. Galvin, G. Gagne. *Applied Operating System Concepts.* John Wiley & Sons, 2000. 755

[31] D. Sima, T. Fountain, P. Kacsuk. *Advanced Computer Architectures: a Design Space Approach.* Addison-Wesley Publishing Company, 1998 (2nd edition). 754

[32] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 2001. 755

[33] A. S. Tanenbaum, M. van Steen. *Distributed Systems. Principles and Paradigms.* Prentice Hall, 2002. 755

[34] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. 756

[35] W. Winkler. Dependent percolation and colliding random walks. *Random Structures & Algorithms*, 16(1):58–84, 2000. 756

[36] J. Wu, S.. On cost-optimal merge of two intransitive sorted sequences. *International Journal of Foundations of Computer Science*, 14(1):99–106, 2003. 756

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, "2-3-4-tree" is indexed as if were "two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: *ex* for exercise, *exa* for example, *fig* for figure, *pr* for problem and *fn* for footnote.

The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of her/his birth and death.