CHAPTER 2

Parallel Systems and Programming

This chapter reviews parallel systems and their programming. The intention is to establish the necessary background and terminology for the following chapters. It begins with the basis of parallel computing—parallel systems—and discusses their architectures and communication networks. In this context, it also addresses programming models for parallel systems. The second part of the chapter is devoted to the parallelization process of parallel programming. A general overview presents the three components of the parallelization process: subtask decomposition, dependence analysis, and scheduling. The subsequent sections discuss subtask decomposition and dependence analysis, which build the foundation for task scheduling.

2.1 PARALLEL ARCHITECTURES

Informally, a parallel computer can be characterized as a system where multiple processing elements cooperate in executing one or more tasks. This is in contrast to the von Neumann model of a sequential computer, where only one processor executes the task. The numerous existing parallel architectures and their different approaches require some kind of classification.

2.1.1 Flynn's Taxonomy

In a frequently referenced article by Flynn [67], the design of a computer is characterized by the flow (or stream) of instructions and the flow (or stream) of data. The taxonomy classifies according to the multiplicity of the instruction and the data flows. The resulting four possible combinations are shown in Table 2.1.

The SISD (single instruction single data) architecture corresponds to the conventional sequential computer. One instruction is executed at a time on one data item. Although the combination MISD (multiple instruction single data) does not seem to be meaningful, pipeline architectures, as found in all modern processors, can be considered MISD (Cosnard and Trystram [45]).

Task Scheduling for Parallel Systems, by Oliver Sinnen

Table 2.1. Flynn's Taxonomy

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

In SIMD (single instruction multiple data) architectures, which are also called data parallel or vector architectures, multiple processing elements (PEs) execute the same instruction on different data items. Figure 2.1(*a*) shows the SIMD structure with one central control unit and multiple processing elements. The central control unit issues the same instruction stream to each PE, which works on its own data set. Especially for regular computations from the area of science and engineering (e.g., signal processing), where computations can be expressed as vector and matrix operations, the SIMD architecture is well adapted.

There are only a few examples of systems that have a pure SIMD architecture, for instance, the early vector machines (e.g., the Cray-1 or the Hitachi S-3600) (van der Steen and Dongarra [193]). Today, the SIMD architecture is often encountered within a vector processor, that is, one chip consisting of the central control unit together with multiple processing elements. A parallel system can be built from multiple vector processors and examples for such systems are given later. Also, most of today's mainstream processor architectures feature an SIMD processing unit, for example, the MMX/SSE unit in the Intel Pentium processor line or the AltiVec unit in the PowerPC processor architecture.

The second parallel architecture in the taxonomy has MIMD (multiple instruction multiple data) streams, depicted in Figure 2.1(*b*). In contrast to the SIMD structure, every PE has its own control unit (CU). Therefore, the processor elements operate

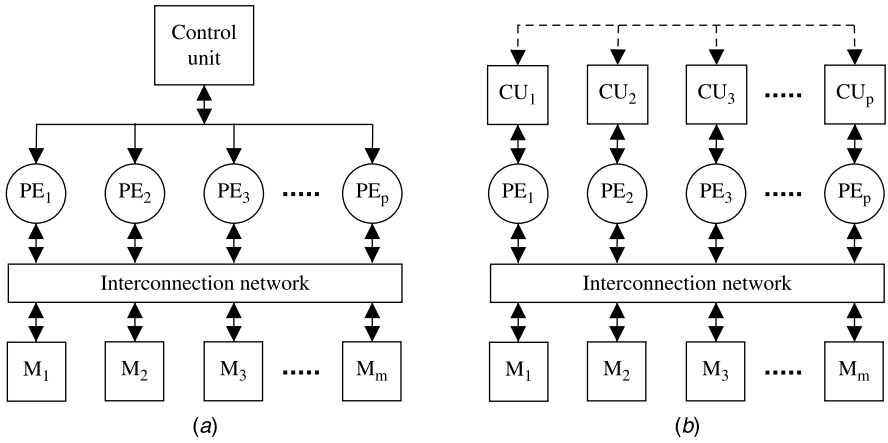


Figure 2.1. SIMD (*a*) and MIMD (*b*) architecture.

independently of each other and execute independent instructions on different data streams. A parallel execution of a global task (i.e., the collaboration of the processing elements) is achieved through synchronization and data exchange between the PEs via the interconnection network. Examples for MIMD architectures are given in the following discussion of memory architectures.

An MIMD architecture can simulate an SIMD architecture by executing the same program on all the processors, which is called SPMD (single program multiple data) mode. In general, however, executing the same program on all processors is not the same as executing the same instruction stream, since processors might execute different parts of the same program depending on their processor identification numbers. The term *processor* stands here for the combination of control unit plus processing element. From now on, this definition of a processor shall be used, if not otherwise stated.

As mentioned earlier, modern parallel systems often consist of multiple vector processors, for example, the Cray J90 or the NEC SX-6 (van der Steen and Dongarra [193]). The Earth Simulator, a NEC SX-6 based system, was the world's fastest computer in 2002–2004 [186]. Within Flynn's taxonomy, these systems can be considered to have an MIMD architecture with an SIMD subarchitecture.

2.1.2 Memory Architectures

It is generally agreed that not all aspects of parallel architectures are taken into account by Flynn's taxonomy. For both the design and the programming model of a parallel system, the memory organization is a very important issue not considered by that classification.

The memory organization of a parallel system can be divided into two aspects: the location and the access policy of the memory. Regarding the location, memory is either *centralized* or *distributed* with the processors. For both cases, systems with a common memory, distributed or not, to which all processors have full access, are called *shared-memory* machines. In systems where there is no such shared memory, processors have to use explicit means of communication like *message passing*. With these two aspects of memory organization in mind, the three most common memory organizations can be examined.

Centralized Memory In a centralized memory multiprocessor, illustrated in Figure 2.2, memory is organized as a central resource for all processors. This typically results in a *uniform memory access* (UMA) characteristic, where the access time to any memory location is identical for every processor. Since the common memory can be accessed by all processors, these systems are called *centralized shared-memory multiprocessors* (Hennessy and Patterson [88]). Due to the UMA characteristic, systems with this architecture are also often called symmetric multiprocessors (SMP).

Distributed Memory The alternative to a centralized architecture is an architecture where the memory is physically distributed with the processors. These systems can be further distinguished according to their memory access policy.

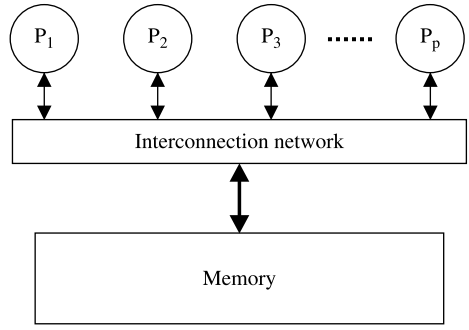


Figure 2.2. Centralized memory multiprocessor.

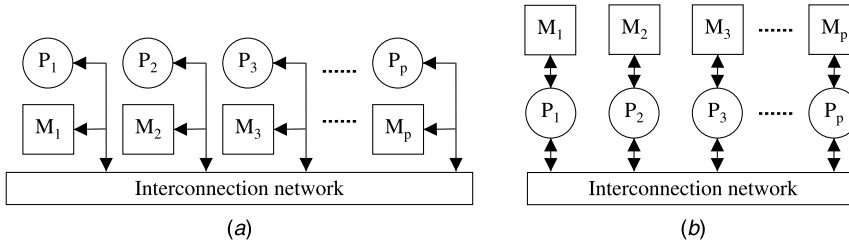


Figure 2.3. Distributed memory multiprocessors: (*a*) shared-memory and (*b*) message passing (memory access goes through processors).

A *distributed shared-memory multiprocessor* (Hennessy and Patterson [88]), as illustrated in Figure 2.3(*a*), integrates the distributed memories into a global address space. Every processor has full access to the memory; however, in general with a *nonuniform memory access* (NUMA) characteristic, as reading from or writing to, local memory is faster than from and to remote memory.

Systems without shared memory are called *distributed memory multiprocessors* or, according to the way the processors communicate, *message passing architectures*[1] (Culler and Singh [48]). Figure 2.3(*b*) displays a distributed memory multiprocessor without shared memory. The difference to the distributed shared-memory multiprocessor is that the local memories are only accessible through the respective processors.

Memory Hierarchy Some distributed memory systems, especially large systems with shared memory, often use some kind of hierarchy for the memory organization. A common example is that a small number of processors (2–4) share one central memory—the processors and the memory comprise a computing node—and

[1]To distinguish the two distributed memory architectures, the supplements shared-memory or message passing shall be used. Otherwise both types are meant.

multiples of these nodes are interconnected on a higher level. Examples of systems with such a hierarchical memory architecture are the Sequent NUMA-Q, the SGI Origin 2000/3000 series, and the IBM Blue Gene/L (van der Steen and Dongarra [193]), which is the world's fastest computer at the time this is written [186].

Motivations The following paragraphs briefly look at the motivations for the different architectures.

Centralized shared-memory architectures are an intuitive extension of a single processor architecture; however, the contention for the communication resources to the central memory significantly limits the scalability of these machines. Bus-based systems therefore have a small number of processors (usually ≤ 8), for example, systems with $\times 86$ processors, and only more sophisticated interconnection networks allow these systems to scale up to 64 processors. For example, in the Sun Enterprise series (van der Steen and Dongarra [193]), systems with low model numbers (e.g., 3000) are connected by a bus, whereas the high-end model, the Sun Enterprise 10000, uses a crossbar (see Section 2.2) for up to 64 processors.

In contrast, distributed memory architectures with message passing allow a much simpler system design, but their programming becomes more complicated. In fact, commodity PCs can be connected via a commodity network (e.g., Ethernet) to build a so-called cluster of workstations or PCs (Patterson [147], Sterling et al. [181]). The big advantage of distributed memory systems is their much better scalability. Hence, it is no surprise that the massively parallel processors (MPPs) are distributed memory systems using message passing with up to hundred thousand processors (e.g., Thinking Machines CM-5, Intel Paragon, and IBM Blue Gene/L).

Distributed shared-memory architectures try to integrate both approaches. They provide the ease of the shared-memory programming paradigm and benefit from the scalability of distributed memory systems, for instance, Cray T3D/T3E, the SGI Origin 2000/3000 series, or the HP SuperDome series (van der Steen and Dongarra [193]). Yet, shared-memory programming of these architectures can have limited efficiency in as much as the heterogeneous access times to memory are often hidden from the programmer.

2.1.3 Programming Paradigms and Models

Shared-Memory Versus Message Passing Programming The programming paradigms for parallel systems have a strong correspondence to the memory access policies of multiprocessors. Fundamentally, one can distinguish between shared-memory and message passing programming. In the former paradigm, every processor has full access to the shared memory, and communication between the parallel processors is done implicitly via the memory. Only concurrent access to the same memory location needs explicit synchronization of the processors. In message passing programming, every exchange of data among processors must be explicitly expressed with send and receive commands.

It must be noted that the employed programming paradigm does not always correspond to the underlying memory organization of the target system. Message passing

can be utilized on both shared-memory and message passing architectures. In a shared-memory system, the passing of a message is often implemented as a simple memory copy. Even distributed shared memory can be emulated on message passing machines with an additional software layer (e.g., survey by Protić et al. [154]).

Parallel Random Access Machine (PRAM) PRAM (Fortune and Wyllie [68]) is a popular machine model for algorithm design and complexity analysis. Essentially, the simple model assumes an ideal centralized shared-memory machine with synchronously working processors. PRAMs can be further classified according to how one memory cell can be accessed: only exclusively by one processor or concurrently by various processors. Memory access to different cells by different processors can always be performed concurrently.

The advantage of PRAM is its simplicity and its similarity to the sequential von Neumann model. Yet, owing to the increasing gap between processing and communication speed, it has become more and more unrealistic.

LogP With the proposal of the LogP model, Culler et al. [46, 47] recognized the fact that the widely used PRAM model is unrealistic due to its assumption of cost-free interprocessor communication, especially for distributed systems. The LogP model gained its name from the parameters that are used to describe a parallel system:

- L: Upper bound on the *latency*, or delay, incurred in communicating a message from a source to a destination processor.
- o: *Overhead*—time during which a processor is engaged in sending or receiving a message; during this time the processor cannot perform other operations.
- g: *Gap*—minimal time between consecutive message transmissions or between consecutive message receptions; the reciprocal of g corresponds to the per-processor bandwidth.
- P: Number of processors.

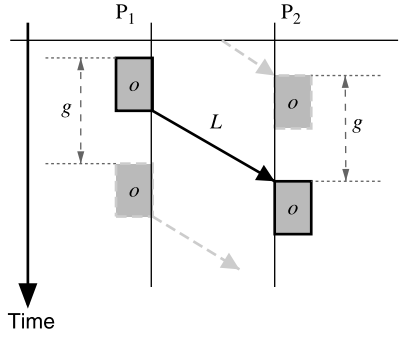


Figure 2.4. Interprocessor communication in LogP.

Furthermore, the structure of the processor network is not described by LogP, but its capacity is limited to $\lceil L/g \rceil$ simultaneous message transfers between all processors. An implicit parameter is the message size M, which consists of one or a small number of words. Based on this parameter, an interprocessor message transfer in the LogP model proceeds as illustrated in Figure 2.4. In contrast to PRAM, LogP is an asynchronous model.

2.2 COMMUNICATION NETWORKS

Fast communication is crucial for an efficient parallel system. A determining aspect of the communication behavior is the network and its topology. In the previous section, some kind of interconnection network for communication among the units of the parallel system was supposed. This section reviews the principal network types, of which each offers a different trade-off between cost and performance.

Initially, interconnection networks can be classified into *static* and *dynamic networks*. Static networks have fixed connections between the units of the system with point-to-point communication links. In a dynamic network, the connections between units of the parallel system are established dynamically through switches when requested. Based on this difference, static, and dynamic networks are sometimes referred to as *direct* and *indirect* networks, respectively (Grama et al. [82], Quinn [156]).

2.2.1 Static Networks

The essential characteristic of a static network is its topology, as the interconnections between the units of the parallel system are fixed. Most static networks are processor networks used in distributed memory systems, where every processor has its own local memory.

Processor network topologies are usually represented as undirected graphs[2]: a vertex represents a processor, together with its local memory and a switch, and an undirected edge represents a communication link between two processors (Cosnard and Trystram [45], Culler and Singh [48], Grama et al. [82]). Figure 2.5(b) depicts an example for a network graph consisting of four processors. Figure 2.5(a) illustrates the implicit association of memory and a switch with each processor.

Once a topology is modeled as an undirected graph, terminology from graph theory can be utilized for its characterization. The *degree* of a vertex is defined as the number of its incident edges, denoted by δ. The eccentricity of a vertex is the largest distance, in terms of the number of edges, from that vertex to any other vertex. Furthermore, the *diameter* of an undirected graph, denoted by D, is defined as the maximum eccentricity of all vertices of the graph. Another notable indicator for a network is its *bisection width*. It is defined as the minimum number of edges that have to be removed to

[2]Basic graph concepts are introduced in Section 3.1 and the undirected graph model of topologies will be defined more formally in Section 7.1. For the current purpose, this informal definition suffices.

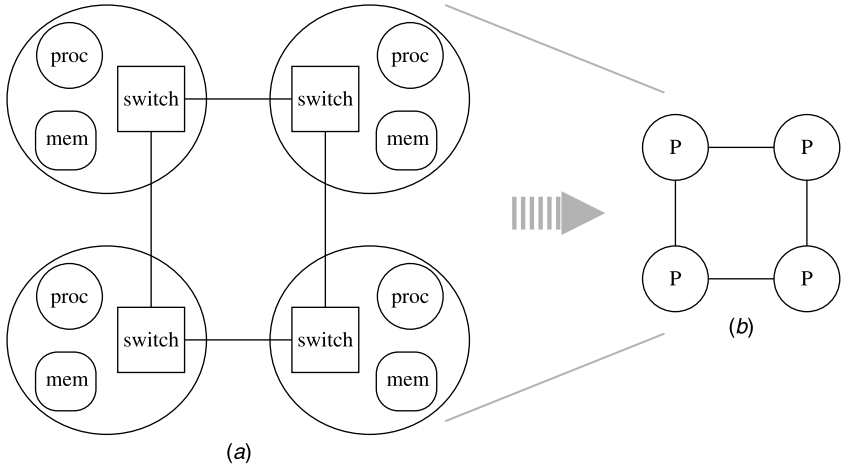


Figure 2.5. An undirected graph representing a processor network (*b*); (*a*) an illustration that the switch and the memory associated with each processor are implicit in the common representation of processor networks.

partition the network into two equal halves. In other words, it is the number of edges that cross a cut of a network into two equal halves.

To achieve a network with a small maximum communication time, the goal is to have a small diameter D. At the same time, the mean degree $\bar{\delta}$ of the network should be small, since it determines the hardware costs. Last but not least, the network should have a large bisection width, because it lowers the contention for the communication links. However, the bisection width is also a measure for the network costs, as it provides a lower bound on the area or volume of its packaging (Grama et al. [82]).

Fully Connected Networks A network in which every processor has a direct link to any other processor, as depicted in Figure 2.6, is called fully connected. It has

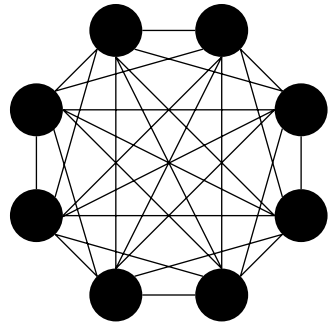


Figure 2.6. An 8-processor fully connected network.

the nice property that it is *nonblocking*; that is, the communication of two processors does not block the connection of any other two processors in the network. All other static networks studied in this section are blocking networks.

Obviously, the degree of each vertex is $\delta = p - 1$, for a network of p processors, the diameter is $D = 1$ and the bisection width is $p^2/4$. Fully connected networks are extreme in both ways: they provide the smallest possible diameter and the highest possible degree and bisection width. In practice, the quadratic growth of the number of links—p processors are fully connected by $p(p-1)/2$ links—renders a fully connected network unemployable for medium to large numbers of processors. However, the fully connected network is important in terms of theoretical work as it is often used as a model.

Meshes In practical terms, the most popular class of static networks are meshes. Processors are arranged in a linear order in one or more dimensions, whereby only neighboring processors are interconnected by communication links. According to their dimension, meshes can be grouped into linear networks or rings (1D), grids (2D), and tori (3D). Figure 2.7 visualizes meshes up to three dimensions.

A distinction can be made between acyclic and cyclic meshes. For the latter, the links are wrapped around at the end of the respective dimension to its beginning. The few additional links in comparison with acyclic meshes reduce the diameter significantly, namely, by half in each dimension. Thus, in practice, meshes are usually cyclic.

Table 2.2 summarizes the properties of acyclic and cyclic meshes up to three dimensions. In general, an n-dimensional cyclic mesh with q_k processors on each

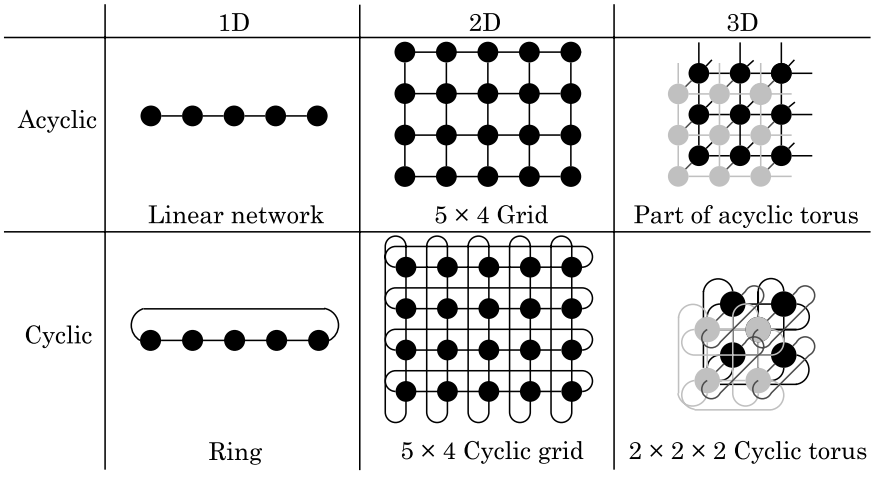


Figure 2.7. Meshes.

Table 2.2. Properties of Static Networks[a]

Network	Processors	Links	Mean Degree $\bar{\delta}$	Diameter D	Bisection Width
Fully connected	p	$p(p-1)/2$	$p-1$	1	$p^2/4$
Linear network	p	$p-1$	$2-2/p$	$p-1$	1
Ring	p	p	2	$\lfloor p/2 \rfloor$	2
Grid	$q_1 \times q_2$	$q_1(q_2-1)+q_2(q_1-1)$	$4-2/q_1-2/q_2$	q_1+q_2-2	$\min(q_1,q_2)$
Cyclic grid	$q_1 \times q_2$	$2q_1q_2$	4	$\lfloor q_1/2 \rfloor + \lfloor q_2/2 \rfloor$	$\min(2q_1,2q_2)$
Torus	$q_1 \times q_2 \times q_3$	$q_2q_3(q_1-1)+q_1q_3(q_2-1)$ $+q_1q_2(q_3-1)$	$6-2/q_1-2/q_2$ $-2/q_3$	$q_1+q_2+q_3-3$	$\min(q_1q_2,q_2q_3,q_1q_3)$
Cyclic torus	$q_1 \times q_2 \times q_3$	$3q_1q_2q_3$	6	$\lfloor q_1/2 \rfloor + \lfloor q_2/2 \rfloor + \lfloor q_3/2 \rfloor$	$\min(2q_1q_2,2q_2q_3,2q_1q_3)$
Hypercube	$p=2^d$	$(p/2)\log p = d2^{d-1}$	$\log p = d$	$\log p = d$	$p/2 = 2^{d-1}$

[a] q_k denotes the number of processors in dimension k; d is the dimension of the hypercube.

dimension k (with $k = 1, 2, \ldots, n$) has the following diameter and mean degree:

$$D = \sum_{k=1}^{n} \left\lfloor \frac{q_k}{2} \right\rfloor \quad \text{and} \quad \bar{\delta} = 2n. \tag{2.1}$$

Its bisection width is

$$\min_i \frac{2 \prod_{k=1}^{n} q_k}{q_i}, \tag{2.2}$$

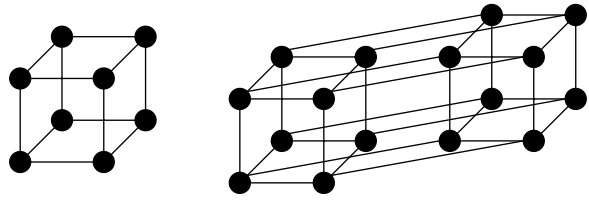
assuming the number of processors in the dimension q_i, through which the bisection cut is made, is even.

The mesh topologies can be very attractive for scientific and engineering computations, since they correspond to data structures like vectors and matrices, which are heavily used for this kind of computations. In the ideal case, the data structures can be distributed uniformly among the processors.

Mesh networks can be found in a large variety of systems. The scalable coherent interface (SCI) standard specifies a ring-based network. A two-dimensional grid is used in the Intel Paragon. Three-dimensional tori are often employed in MPP systems, for example, the Cray T3E or the IBM Blue Gene/L.

Hypercubes A hypercube is a network with the interesting property that the degree equals the diameter, $\delta = D = d$, where d is the dimension of the hypercube. The number of processors is given by $p = 2^d$. A hypercube of dimension d can be recursively constructed from two $(d-1)$-cubes, in which each processor of one cube is connected to the processor at the same position in the other cube. Figure 2.8 shows how this is performed for a 4D cube using two 3D cubes. This construction procedure also leads intuitively to the bisection width of $p/2 = 2^{d-1}$, which is the number of edges that connect the two identical $(d-1)$-cubes.

Hypercubes have the nice property that the diameter and the degree only grow logarithmically with the number of processors. Furthermore, routing can be implemented with very little effort using Gray codes to denote the processors (Cosnard and Trystram [45]). However, these theoretical advantages are opposed by practical shortcomings. The number of processors must be a power of 2. Moreover, building a large hypercube in hardware is difficult, as the links differ much in length. An example of systems employing a hypercube is the SGI Origin 2000/3000 series.



3D 4D

Figure 2.8. Hypercubes.

Summary Table 2.2 summarizes the properties of the static networks analyzed so far, including acyclic and cyclic meshes with up to three dimensions.

Many other static network topologies have been proposed in the past: most notably, topologies that are constructed modularly by substituting the nodes of one topology with subtopologies. A detailed study of more sophisticated topologies and further reading can be found in Cosnard and Trystram [45], Culler and Singh [48], Grama et al. [82], and Quinn [156].

2.2.2 Dynamic Networks

In contrast to static networks, dynamic networks establish the connection between the units of the parallel system on demand. The connections are made through internal network switches that are not associated with any processor—hence the alternative name of indirect network. This is in contrast to static networks, where every switch is implicitly associated with one processor; see Figure 2.5. A dynamic network is constituted by a set of switches, which route the communication through the network from its source to its destination.

Crossbars The most powerful dynamic network is a crossbar. It employs a grid of nm switches to connect n units on one side with m units on the other side, as illustrated in Figure 2.9. Through this simple approach, it is a *nonblocking* network; that is, a connection between unit i on the left and unit j at the top does not block the connection of any other unit on the left with any other unit at the top. An example of this nonblocking behavior is shown in Figure 2.9(b). From the static networks considered in Section 2.2.1, only the fully connected network has the same nonblocking property. However, it also comes at a similar cost, as the number of switches in a crossbar has the same complexity as the number of links in a fully connected network, namely, $O(p^2)$

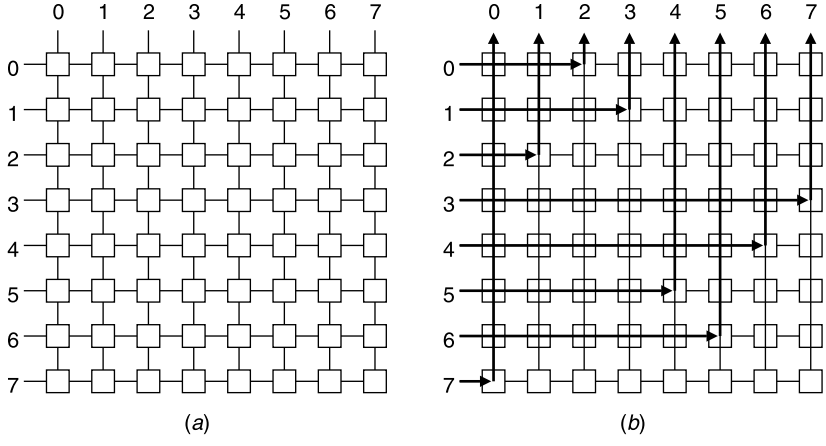


Figure 2.9. An 8 × 8 crossbar; (a) the source and destination units are connected to the numbered lines; (b) nonblocking communication routing in the crossbar. The squares represent the switches.

(assuming $n = m = p$). Even though the diameter grows linearly with the number of processors, $D = 2p$, this is often not an issue, since crossbars are usually implemented in a very compact form. In any case, the quadratically growing number of switches is more limiting to the scalability. On the upside, the bisection width grows linearly with the number of processors.

In conclusion, crossbars are not very scalable costwise and are only used for small to medium sized networks. They are found in larger shared-memory architectures (Section 2.1.2), for example, Sun Enterprise 10000, Sun Fire E25K, and on the node level in the SGI Origin 2000/3000 series and the HP SuperDome series.

Multistage Networks In comparison to crossbars, multistage networks employ a reduced number of switches. This makes them more scalable in terms of cost but also makes them *blocking* networks; that is, certain combinations of source–destination connections block each other. Yet, the aim of their design is to keep the number of conflicts low in practice.

A common multistage network is the *butterfly network*. Figure 2.10 visualizes a three-dimensional butterfly network with 8 source units and 8 destination units. Every communication from a source unit (left side of Figure 2.10(a)) travels through switches (squares) until it reaches its destination unit (right side of Figure 2.10(b)). So this network consists of three stages. As noted earlier, conflicts can arise between certain combinations of source–destination connections. For example, the source–destination connections 2–2 and 6–3 conflict, since both utilize the same link between the second and the third stage (Figure 2.10(b)).

Butterfly networks are equivalent to *omega networks* and one can be turned into the other by relabeling/reordering the switches (Cosnard and Trystram [45]). The omega network uses the so-called perfect shuffle pattern (Cosnard and Trystram [45]) to interconnect the switches. An example for an omega network is found in the IBM RS/6000 SP-SMP.

The number of switches in a butterfly or omega network is $(p/2)\log p$ (switches-per-stage $\times$ stages), or $2^{d-1}d$ with $p = 2^d$, which is significantly less compared to the p^2 switches in a crossbar. Furthermore, the diameter grows only logarithmically with the number of the processors, $D = \log p + 1 = d + 1$, and the

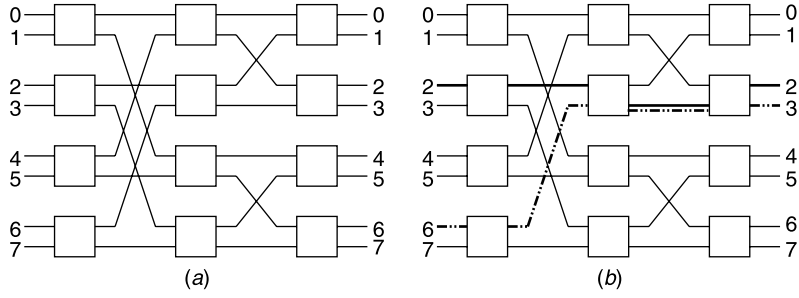


Figure 2.10. A 3D butterfly network with 3 stages; (a) the source and destination units are connected to the numbered lines; (b) example for blocking communication conflict.

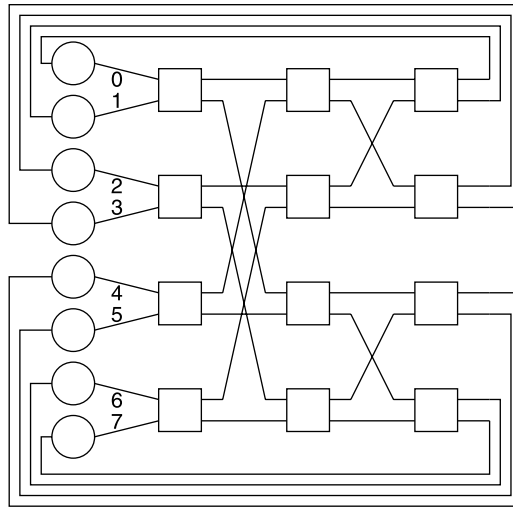


Figure 2.11. A 3D butterfly connecting 8 processors by wrapping around the destination to the source.

bisection width is $p/2$ (in Figure 2.10 the bisection cut is done horizontally between lines 3 and 4).

Dynamic networks are often employed in larger scale centralized shared-memory architectures (Section 2.1.2) as a connection between the processors on the one side and the memory banks on the other (e.g., Convex Exemplar, Sun Enterprise 10000).

Dynamic networks are also utilized to interconnect processors, or processing nodes consisting of several processors, for example, in the IBM RS/6000 SP-SMP, the HP SuperDome series, or the Sun Fire E25K. As in the static networks discussed in Section 2.2.1, memory is usually associated with each processor or processing node, consisting of several processors. For crossbars and multistage networks, the processor network is built by wrapping around the connection lines on the destination side back to the source side (Hennessy and Patterson [87]). In other words, destination line i is connected directly to the processor at source line i. Using this method, the butterfly in Figure 2.10 can interconnect 8 processors (circles) with each other, as illustrated in Figure 2.11.

Tree Networks In tree structured dynamic networks, the leaf nodes (i.e., the nodes without children) are the processors and the intermediate nodes are the switches. An example of such a network, a binary tree network, is shown in Figure 2.12. To send a communication from one processor to another, the message must travel upward until it reaches the root node of the subtree containing the destination processor, from which it descends to it. A binary tree network for $p = 2^d$ processors has $p - 1$ switches. Hence, it has less switches than the discussed multistage networks, while it has also a logarithmically growing diameter, $D = 2 \log p$. The big shortcoming of

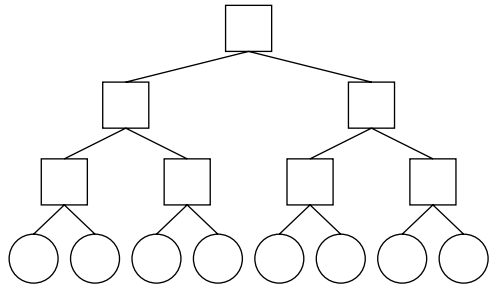


Figure 2.12. Binary tree network with 8 processors (circles).

the tree is its bisection width of 1. The links at higher levels are easily congested with communications. All communications between the processors on the left side and the processors on the right side of a tree go through the root node and its links. Fat trees alleviate this problem by having more, parallel links in higher levels of the tree. In other words, the bandwidth between the switches is increased on the way to the root.

Trees are especially interesting for certain types of communication, for example, a broadcast, where one message is sent to all processors. Once the message is at the root, all processors receive the message at the same time, with a logarithmic delay in terms of the number of processors. For this reason, the IBM Blue Gene/L employs, in addition to its torus network, a tree network.

Summary Table 2.3 summarizes the properties of the dynamic networks considered so far. The expressions are given for processor networks; that is, only processors are connected to the network, not separate memory banks. For the crossbar and the butterfly/omega network, this implies that the destination lines are wrapped around to the processors which are connected at the source lines, as shown in Figure 2.11 for an 8-processor butterfly. As a consequence, the expressions for the number of links, the diameter, and the bisection width include those links that connect the processors with the dynamic network. This makes these expressions more comparable to each other

Table 2.3. Properties of Dynamic Networks Connecting p Processors

Network	Processors	Switches[a]	Links	Maximum Links per Switch	Diameter D	Bisection Width
Crossbar	p	p^2	p^2	4	$2p$	$3p/2$
Butterfly/ omega	$p = 2^d$	$(p/2)\log p$ $= d2^{d-1}$	$p(\log p + 1)$ $= 2^d(d+1)$	4	$\log p + 1$ $= d + 1$	$p/2$
Binary tree	$p = 2^d$	$p - 1$	$p - 1$	3	$2\log p = 2d$	1

[a] Switches—the number of switches not associated with a processor.

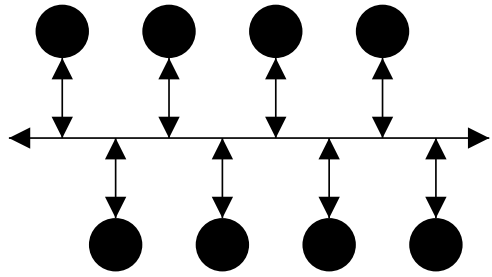


Figure 2.13. An 8-processor bus topology.

and to static networks. Instead of the mean degree $\bar{\delta}$ (Section 2.2.1), the maximum number of links per switch is given.

Bus One particular dynamic network, the bus, deserves special attention, since it is found in many small-sized parallel systems and also as a building block in larger scale computers. Conceptually, it is quite different from the dynamic networks analyzed so far. The units of the parallel system are grouped around the bus, which is shared among them for their communications, as shown in Figure 2.13. A controller assigns the bus to a unit on request and during the assigned time the unit can use the bus exclusively.

In terms of performance scalability, it is situated on the opposite side of the spectrum, when compared with the crossbar. Already one communication between two processors blocks the bus for any other connection. As a consequence, the bus is usually time-shared among the connected units in practice. While its simplicity makes the bus very attractive, it very quickly becomes the performance bottleneck of a parallel system. Consequently, the number of connected units is small (usually ≤ 8).

2.3 PARALLELIZATION

The objective of a parallel system is the fast execution of applications, faster than they can be executed on single processor systems. For this aim, the application's formulation as a program must be in a form that allows it to benefit from the multiple processing units. Unfortunately, parallel programming (i.e., the formulation of an application as a program for a parallel system) is significantly more complex than sequential programming. The additional procedures involved in parallel programming are often called the *parallelization* of the program, especially if the parallel version of the program is derived from an existing sequential implementation.

Figure 2.15 illustrates the process of parallel programming and for comparison Figure 2.14 shows that of sequential programming, where the program is directly created from the application specification. For its parallelization, the application must be divided into subtasks, or simply tasks, which in general are not independent. Precedence constraints, caused by the dependence of a task on the result of another task,

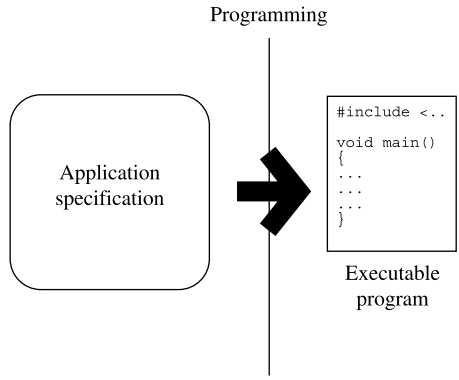


Figure 2.14. Sequential programming.

impose a partial order of execution among the tasks. Thus, a correct execution of the application demands a dependence analysis, on whose basis the tasks must be ordered for execution. Given the division of the application into tasks and the analysis of their dependence relations, the final step of parallelization consists of mapping the tasks onto the processors and the determination of their execution order, which corresponds to the assignment of start times to the tasks. The assignment of start times is called *scheduling* and can only be accomplished with an existing mapping. Therefore, generally both are meant, the spatial (i.e., mapping) and the temporal assignment of the tasks to the processors, when referring to scheduling. Based on the schedule of the tasks, the parallel program can be created. This process of parallelization is depicted in Figure 2.15 from left to right: beginning with the specification of the application, the program undergoes subtask decomposition, dependence analysis, and scheduling. At the end of the parallelization process, the program code must be generated as in sequential programming.

In many practical approaches to parallel programming, the steps of parallelization are not clearly separable. For example, tasks are often grouped into processes or threads (the so-called orchestration (Culler and Singh [48]) or agglomeration

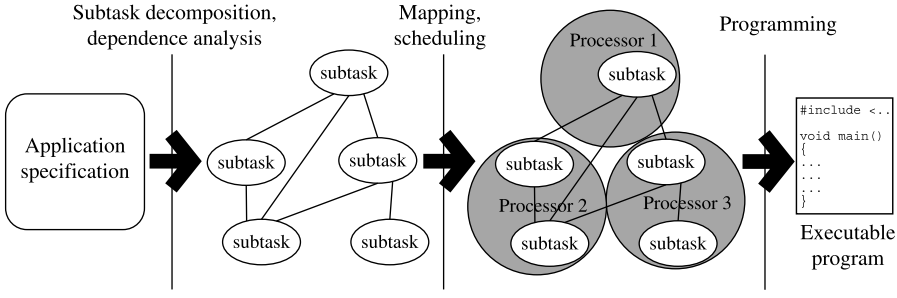


Figure 2.15. Parallel programming—process of parallelization.

(Foster [69])) before the mapping and scheduling. Furthermore, the parallelization part of parallel programming might be repeated several times until a satisfying solution is obtained (Foster [69]), since the decomposition into subtasks determines the precedence constraints among them, which in turn restrict the scheduling on the processors. In other words, different decompositions into subtasks can lead to different precedence constraints and schedules with different efficiencies.

Summarizing, three main areas of the parallelization process are identified:

- Subtask decomposition
- Dependence analysis
- Scheduling

Subtask decomposition and dependence analysis build the foundation for scheduling. The next two sections focus on these areas in order to lay the groundwork for the treatment of scheduling. Examples used in the following sections help one to better comprehend the aspects of parallelization introduced here.

2.4 SUBTASK DECOMPOSITION

In general, the decomposition of a program into subtasks cannot be examined isolated from other aspects of the parallelization process. First, the decomposition is driven by the type of computation, and the programming techniques and language used for the program formulation. Second, the subsequent techniques employed in the parallelization process have a strong influence on the decomposition process. Finally, the architecture of the target system and the corresponding parallel programming paradigm affect the decomposition. Before these three aspects are discussed, the next two subsections look at metrics that characterize a decomposition and common decomposition techniques.

2.4.1 Concurrency and Granularity

The decomposition of an application into subtasks determines its concurrency or parallelism, which is defined as the amount of processing that can be executed simultaneously. More specifically, the *degree of concurrency* is the number of subtasks that can be executed simultaneously. Interesting indicators of a decomposed program are the *maximum* and the *average degree of concurrency* (Grama et al. [82]). The maximum degree is defined as the maximum number of subtasks that can be executed simultaneously at any time during the execution of the program. Correspondingly, the average degree is defined as the average number of subtasks that can be executed simultaneously during the execution of the entire program. Commonly, the maximum degree of concurrency is inferior to the total number of subtasks due to dependences between the tasks. This will become clear in Section 2.5 on dependence analysis. Ideally, the decomposition completely exposes the inherent parallelism or concurrency of an application.

Another notion widely used in the context of parallelism is that of *granularity*. By saying, for example, "a program is coarse grained," it is meant that the distinguishable chunks of the program, in other words the subtasks, are large considering the size of the entire program. Typically, the granularity of a program is informally differentiated between *small/fine*, *medium*, and *coarse grained*. Granularity can also be understood as the relation of the data transfer to the computational work of a program object. For instance, a small amount of computational work in comparison to the data transfered is considered to be small grained. Although all this is a rather informal definition of granularity, its utilization in the literature is quite usual. With the aid of a graph model, however, granularity can be defined formally, which is done in Section 4.4.3. Until then, the notion of granularity will be used in the usual sense of comparing the computational size of objects.

2.4.2 Decomposition Techniques

Grama et al. [82] describe several decomposition techniques: data, recursive, exploratory, and speculative decomposition. While this is not a complete list of decomposition techniques, it comprises commonly used techniques. Data decomposition and recursive decomposition are general-purpose techniques, while exploratory decomposition and speculative decomposition are more specialized and only apply to certain classes of problems. Their structural approach is similar to recursive decomposition. In practice, it also happens that these techniques are combined in what might be called hybrid decomposition.

Data Decomposition In data decomposition, the focus is on the data rather than on the different operations applied to the data. This technique is powerful and commonly used for programs that operate on large data structures, as often found in scientific computing. In terms of Flynn's taxonomy (Section 2.1), this method corresponds to SIMD. The data is partitioned into N usually equally sized parts, whereby N depends on the desired degree of concurrency. If there are no dependences among the operations performed on the different parts, these N parts can be mapped onto N processors and executed in parallel. In other words, the degree of concurrency is N.

Figure 2.16 illustrates the example of a data decomposition applied to the matrix–vector multiplication $A \bullet b = y$, with $N = 4$. The four tasks that result from this decomposition are independent; for example, task 1 calculates $y[i] = \sum_{j=1}^{n} A[i, j]b[j]$ for $1 \leq i \leq n/N$ and task 2 for $n/N + 1 \leq i \leq 2n/N$.

Data decomposition can be differentiated further by the way the data is partitioned. The partitioning can be applied to the output data, input data, intermediate data, or a mixture of them. In the earlier matrix–vector multiplication example, the output data is completely partitioned and the input data is partially partitioned (matrix A is partitioned, vector b is not).

Recursive Decomposition This decomposition technique uses the divide-and-conquer strategy. A problem is solved by first dividing it into a set of independent subproblems. Done recursively, a tree structure of subproblems is created as depicted

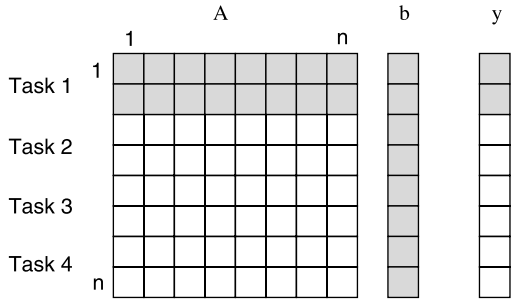


Figure 2.16. Data composition of matrix–vector multiplication $A \bullet b = y$. Data is partitioned into $N = 4$ parts corresponding to 4 tasks; the gray-shaded part shows the data accessed by task 1 (Grama et al. [82]).

in Figure 2.17. The leaves of the tree are basic subproblems that cannot be divided further into smaller subproblems. Each node that is not a leaf node combines the solutions of its subproblems. The result is finally computed in the root of the tree, when its subproblems are combined. Each node of the tree represents one task, and the edges between the nodes reflect their dependences (task dependence and graphs for dependence representation are discussed in Section 2.5 and Chapter 3, respectively). Since all leaves are independent from each other, the maximum degree of concurrency is the number of leaves.

The divide-and-conquer structure can be found in many applications and algorithms, for example, in Mergesort (Cormen et al. [42]).

Exploratory Decomposition This technique has the same tree-like structure as the recursive decomposition. Exploratory decomposition is employed when the underlying computation corresponds to a search of a space for solutions. For such a problem, the search space is represented by a tree, where each node represents a point in the search space and each edge a path to the neighboring points. To decompose

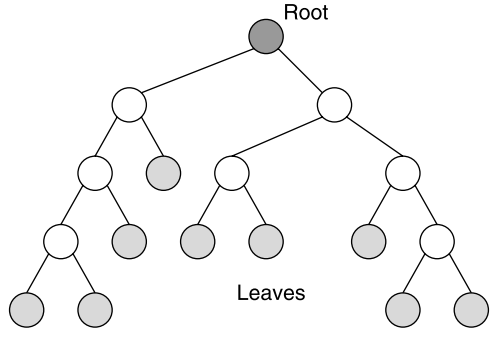


Figure 2.17. Tree created by a recursive decomposition.

such a problem for parallel execution, an initial search tree is generated until there is a sufficiently large number of leaf nodes that can be executed in parallel (i.e., until the required degree of concurrency is met). Each of these leaf nodes corresponds to an independent task, which searches for a solution in the subtree induced below this node. The overall computation can finish as soon as one of the parallel tasks finds a solution. This fact is the main difference between exploratory decomposition and data decomposition, where all subproblems must be solved in their entirety.

Speculative Decomposition Speculative decomposition takes the idea of exploratory decomposition even one step further. In applications where the choice for a certain computation branch depends on the result of an earlier step, a speedup can be achieved by speculatively executing all (or a subset of all) branches. At the time the data for making the choice is finally available, only the result of the correct branch is chosen; the others are discarded. For example, an `if` statement in a programming language has two lines of execution—one if the evaluated expression is true, another if it is false. In speculative decomposition, each of the two branches is made a task and executed concurrently. At the time the expression of the `if` statement can finally be evaluated, the result of the correct branch is chosen and the result of the other branch is discarded.

2.4.3 Computation Type and Program Formulation

Essential for the subtask decomposition is the *granularity*, the *dependence structure*, and the *regularity* inherent in an application and its description as a program. The computation type of an application and its expression as a program may already give a strong indication for its decomposition into subtasks.

Computation Type Most applications belonging to one area might employ certain types of computation and data structures. For instance, an application of signal processing is likely to perform linear algebra on matrices. In this case, data decomposition is probably the right technique. Other application areas might typically use other types of data structures (e.g., lists and graphs) and computations (e.g., matching and search algorithms). For these application areas, recursive or exploratory decomposition can be a good candidate. Furthermore, depending on the computation type, an application might consist of several distinct steps or one iterative block of computation. In the former case, each step can be represented by one task, hence using coarse grained parallelism, while in the latter case the iterative computation suggests a regular, small grained decomposition, potentially based on data decomposition.

The type of the application also limits the inherent degree of concurrency. For example, the matrix–vector multiplication in Figure 2.16 performs n^2 multiplications and $n(n-1)$ additions. All multiplications can be performed concurrently, but the additions must wait for the multiplications to finish. Hence, the degree of concurrency is bounded by n^2. Also, it is often pointless to partition an application into subtasks that strongly depend on each other's results, preventing their parallel execution due to the small degree of concurrency. As will be shown in the next section,

some dependence relations are inherent in an application, while others arise from its actual implementation as a program.

Thus, the application area often defines the type of computation, which in turn largely determines the inherent granularity, dependence structure, and regularity of an application. Yet, the parallelization process often initiates from the application's description as a (sequential) program.

Program Formulation One application can be expressed by many, sometimes substantially, different programs. A regular computation expressed in a loop supports its division into iterations (e.g., using data decomposition)—that is, a fine grained parallelism—while the same computation expressed as a recursive function might be easier divided into a few large tasks based on recursive decomposition. Also, a program, for example, written with a few large functions, is surely a candidate for a coarse grained decomposition, while a program with many small functions is probably better handled in a medium grained approach. This also depends strongly on the parallelization techniques that can be applied to the utilized programming paradigm and language.

2.4.4 Parallelization Techniques

Different parallelization techniques require or favor different subtask decompositions. This is illustrated by the following two examples.

When talking about parallelization techniques, one should not forget that in most cases a program is parallelized manually by its designer. A popular approach for parallel programming is message passing, where communication between the processors of a parallel system is performed by the exchange of messages (Section 2.1.3). It is typically the designer who decides which parts of the program are to be executed concurrently, thereby defining the subtasks of the program. The designer's job is then to insert the message passing directives for the synchronization and communication between those tasks. Such an approach normally favors a coarse grained decomposition.

Parallelizing compilers, on the other hand, focus on the parallelization of loops in programs. They therefore support regular decompositions with rather small or medium granularity.

Furthermore, it is important to determine how dependence relations between tasks can be avoided or eliminated by a transformation of the program.

2.4.5 Target Parallel System

Another dominant factor in the decomposition process is the target parallel system. At the beginning of this chapter different parallel architectures and systems were discussed. Some of those systems are appropriate for small grained parallelism, while others are better suited for coarse grained tasks. The number of processors, the communication speed and the communication overhead are, among others, important aspects for an inclination toward a certain form of parallelism.

A parallel system, whose interprocessor communication is based on message passing, typically performs interprocessor communication with a relatively expensive overhead. Consequently, subtasks should be quite large so that the overhead becomes small when compared with the computation time of the tasks; otherwise the speedup of the parallel program is degraded.

A massively parallel system, with hundreds of processors, needs a high degree of parallelism in order to involve all processors in the execution of the program, which obviously cannot be achieved with only a few large tasks.

These two examples demonstrate the influence of the system architecture on the subtask decomposition of a program. Also, the programming paradigm can have a similar effect. A centralized shared-memory system, which inherently supports fine grained parallelism, can also be programmed with a message passing paradigm. Yet, when doing so, the same applies as for a message passing system; hence, coarser granularity is indicated.

Earlier discussion showed that the decomposition of a program is a complex task that is strongly interwoven with other parts of the parallelization process. As was mentioned at the beginning of this section, it is not possible to treat subtask decomposition separately from its context. Nevertheless, this section gave an overview of the general subject and techniques and referred to the associated problems. In the next section, subtask decomposition will be seen in practice for simple program examples.

2.5 DEPENDENCE ANALYSIS

Dependence analysis (Banerjee et al. [17], Wolfe [204]) distinguishes between two kinds of dependence: *data dependence* and *control dependence.* The latter represents the dependence relations evoked by the control structure of the program, which is established by conditional statements like `if...else`. Section 2.5.3 analyzes this kind of dependence. The discussion starts with data dependence, which reflects the dependence relations between program parts, or tasks, caused by data transfer.

2.5.1 Data Dependence

The best way to build an understanding for dependence is to start with a simple example. Consider the following equation:

$$x = a * 7 + (a + 2). \tag{2.3}$$

Assume now that Eq. (2.3) corresponds to the initial application specification of the parallelization process as described in Section 2.3 (see Figure 2.15). The first step is to divide the equation into subtasks. This can be done by considering every operation as one task. For example, suppose the value 2 is assigned to a, a program can be written from Eq. (2.3) with four tasks as presented in Example 1.

Example 1 Program for $x = a * 7 + (a + 2)$

```
1: a = 2
2: u = a + 2
3: v = a * 7
4: x = u + v
```

Every line of this program is a statement of the form *variable* = *expression*. In the dependence analysis literature, statements are the entities between which dependence relations are defined, especially since classical dependence analysis is strongly linked to cyclic computations (i.e., loops), which will be covered in Section 2.5.2. In scheduling, the notion of a task is preferred, as it reflects a more flexible concept in terms of the size of the represented computation. A task can range from a small statement to basic blocks, loops, and sequences of these. To emphasize the fact that the ideas of this section have general validity, a statement will be referred to as a task. The concepts of dependence presented here are unaffected by the choice of notation.

The *variable* of each statement *variable* = *expression* holds the result of the task and thereby acts as its output, while the variables in the *expression* are the input of the task. In Example 1, the output variable of task 1 (a) is an input variable of tasks 2 and 3, and the output variables of task 2 (u) and task 3 (v) are the input variables of task 4. In other words, tasks 2 and 3 depend on the outcome of task 1, and task 4 on the outcomes of tasks 2 and 3. It is said that tasks 2 and 3 are *flow dependent* on task 1, caused by variable a. Correspondingly, task 4 is flow dependent on tasks 2 and 3, caused by the variables u and v. Consequently, only tasks 2 and 3 are not dependent on each other and can therefore be executed in parallel.

Flow dependence is not the only type of dependence encountered in programs. Consider the equation in Example 2 and its corresponding formulation as a program.

Example 2 Program for $x = a * 7 + (a * 5 + 2)$

```
1: a = 2
2: v = a * 5
3: u = v + 2
4: v = a * 7
5: x = u + v
```

While task 3 is flow dependent on task 2 (caused by variable v), there is a new form of dependence between tasks 3 and 4. In flow dependence, a write occurs before a read, but task 3 reads variable v before task 4 writes it. The supposed input of task 3 is the value computed by task 2 and not that of task 4, but both use variable v for their output. In order to calculate the correct value of x, task 3 must be executed before task 4. Task 4 is said to be *antidependent* on task 3.

Another type of dependence is output dependence. Consider the program extract in Example 3.

Example 3 Output Dependence

```
   ...
2: o = 10
   ...
4: o = 7
   ...
```

Suppose the variable o is only utilized in the two tasks 2 and 4 of the program. After a sequential execution, the value of o is 7. In a parallel execution, however, the value of o is determined by the execution order of the two tasks: if task 4 is executed before task 2, the final value of o is 10! Obviously, the sequential and the parallel versions should produce the same result. Thus, task 4 is said to be *output dependent* on task 2—for a correct semantic of the program task 2 must be executed before task 4.

The antidependence in Example 2 and the output dependence of Example 3 can be resolved by modifying the programs. For instance, if Example 2 is rewritten as displayed in Example 4, its antidependence is eliminated—task 4 is no longer antidependent on task 3.

Example 4 Eliminated Antidependence from Program in Example 2

```
1: a = 2
2: v = a * 5
3: u = v + 2
4: w = a * 7
5: x = u + w
```

The dependence was eliminated by substituting variable v in tasks 4 and 5 with the new variable w. Likewise, the output dependence of Example 3 is eliminated by the utilization of an additional variable.

Real Dependence In general, antidependence and output dependence are caused by variable (i.e., memory) reuse or reassignment. For this reason, they are sometimes called *false dependence* as flow dependence is called *real dependence* (Wolfe [204]). Flow dependence is inherent in the computation and cannot be eliminated by renaming variables.

Nevertheless, the concepts of output and antidependence are important in dependence analysis of concrete programs, for example, in a parallelizing compiler (Banerjee et al. [13], Polychronopoulos et al. [153]). There, the program is a concrete implementation of an application specification and might contain these types of false dependence. The compiler must either eliminate the dependence relations or arrange the tasks in precedence order.

Dependence in Scheduling As will be seen later in Section 3.5, the graph representation of programs used in task scheduling only addresses real data dependence. Other dependence types are bound to a given implementation and are not inherent in the represented computation. Thus, a graph represents either an application specification with the inherent real data dependence relations or a concrete program, where all false dependence relations have been eliminated. Based on this convention, every data dependence in a graph is equivalent to a data transfer, called communication, between the respective tasks.

2.5.2 Data Dependence in Loops

Data dependence in loops (i.e., iterative (or cyclic) computations) is conceptually identical to the data dependence in linear programs as described previously. For the parallelization of a program, loops are very attractive, since they typically consume the lion's share of the total execution time of the program. They are different from linear programs in two aspects: (1) loops form a regular computation—the same statements are executed multiple times; and (2) loops often contain array variables. Both aspects together have an impact on the dependence analysis of loops. The following discussion of data dependence in loops is restricted to flow dependence, yet the definitions and conclusions are valid for all types of data dependence.

Single Loops Example 5 presents a loop over the index variable i with a loop body, or loop kernel, consisting of the two statements, or tasks, `S` and `T`.

Example 5 Single Loop

```
for i = 2 to 100 do
  S: A(i) = B(i+2) + 7
  T: B(i*2+1) = A(i-1) + C(i+1)
end for
```

During the execution of the loop, i takes $2, 3, \ldots, 100$ as its values. For each value, an instance of the loop body is executed, and this instance is called an iteration of the loop. In each iteration, instances of the tasks `S` and `T` are executed and the instances corresponding to the iteration for $i = j$ are denoted by $\mathtt{S}(j)$ and $\mathtt{T}(j)$. Unlike in the previous discussion where the variables used in the examples are scalars, the variables in the tasks `S` and `T` are elements of the disjoint arrays A, B, and C. Thus, data dependences arise from the reference to the same array element of different task instances. The two tasks read from and write to elements of the arrays A and B, potentially creating a dependence relation. Elements of C are only read in task `T`, therefore not causing any dependence relation.

Within one iteration, that is, between the instances $\mathtt{S}(j)$ and $\mathtt{T}(k)$, with $j = k$, it can easily be verified that there is no dependence: for any value of $i = j$, $\mathtt{S}(j)$ and $\mathtt{T}(j)$ access different elements of A and B. $\mathtt{S}(j)$ writes to $A(j)$ and $\mathtt{T}(j)$ reads from

$A(j-1)$, but i never equals $i-1$ for $i = 2, \ldots, 100$. Likewise for array B, since $i+2 \neq i*2+1$ for $i = 2, \ldots, 100$. If such a dependence exists, it is referred to as *intraiteration dependence*.

Dependence does arise between instances of different iterations. Example 6 shows the first four iterations, similar to the linear programs studied in the previous section.

Example 6 First Iterations of Example 5

```
S(2):  A(2) = B(4) + 7
T(2):  B(5) = A(1) + C(3)

S(3):  A(3) = B(5) + 7
T(3):  B(7) = A(2) + C(4)

S(4):  A(4) = B(6) + 7
T(4):  B(9) = A(3) + C(5)

S(5):  A(5) = B(7) + 7
T(5):  B(11) = A(4) + C(6)
...
```

Thus, it is easy to spot that instance T(3) is flow dependent on S(2): the output variable of task S(2), $A(2)$, is the input of T(3). The same can be observed for the relation between T(4) and S(3), and between T(5) and S(4), caused by other elements of array A. The dependence *distance*, that is, the number of iterations between the two task instances forming a dependence relation, is 1. In general, it can be stated that the instance T($i+1$) is dependent on the instance S(i). Such a dependence is called *uniform* since its distance is constant. Analogous to the intraiteration dependence, this type is referred to as *interiteration dependence*. For flow dependence relations, intra- and interiteration dependences correspond to intra- and interiteration communications, respectively. Note that a task can even depend on itself, under the condition that the dependence distance is greater than 0. Another logical step is to consider intraiteration dependence as a special case of interiteration dependence, namely, with distance 0. In fact, dependence analysis for loops, for example, as presented by Banerjee et al. [17], does not distinguish between intra- and interiteration dependence. However, it will prove useful for the treatment of graph representations in Chapter 3.

In Example 6 one observes further dependence relations. The instance S(3) is flow dependent on T(2), caused by the array element $B(5)$, and S(5) is flow dependent on T(3), caused by $B(7)$. These are two examples of the dependence pattern caused by the output variable $B(i*2+1)$ of T and the input variable $B(i+2)$ of S. Other dependence pairs of this pattern are: S(7) depends on T(4), S(9) depends on T(5), ..., S(99) depends on T(50). Since the distance of these dependence relations is not constant—the minimum distance is 1 and the maximum is 49—it is said to be *nonuniform*.

Double Loops—Loop Nests The next logical step in dependence analysis of loops is to extend the described concepts to double and multiple nested loops. Example 7 displays a double loop over the indices i and j, containing the two statements, or tasks, `S` and `T` in the kernel.

Example 7 Double Loop

```
for i = 0 to 5 do
  for j = 0 to 5 do
    S: A(i+1,j) = B(i,j) + C(i,j)
    T: B(i+1,j+1) = A(i,j) + 1
  end for
end for
```

Suitable for the double loop, the arrays used in the tasks are two dimensional, whereby the index variable i of the outer loop is used only in the subscripts of the arrays' first dimension and the index variable j only in the subscripts of the second dimension. While this is common practice in nested loops, it is neither a guaranteed nor a necessary condition for the dependence analysis in nested loops. The arrays, for example, might only have one dimension, and the subscripts might be functions of more than one index variable. Relevant for a dependence relation is only the reference of two different tasks to the same array element.

What the index variable is to the single loop is now, in a straightforward generalization, an index vector of two dimensions. An instance of the double loop kernel (i.e., an iteration) is determined by the two corresponding values of the index variables i and j. Also, an instance of one of the tasks `S` and `T` is denoted by $\mathtt{S}(i,j)$ and $\mathtt{T}(i,j)$, respectively. The extension to a more general nest of loops follows a similar pattern—every loop simply contributes one dimension to the index vector. In the same way, the index variable of a single loop can be treated as an index vector of one dimension.

By examining the tasks of the loop in Example 7, it becomes apparent that instance $\mathtt{S}(i+1, j+1)$ depends on instance $\mathtt{T}(i, j)$, caused by the references to the elements of array B, and instance $\mathtt{T}(i+1, j)$ depends on $\mathtt{S}(i, j)$, caused by the references to the elements of array A. As a logical consequence of the generalization from the index variable to an index vector, the dependence distance is also expressed as a *distance vector*. For the identified dependence relations in Example 7, the distance vectors are $(1, 1)$, for $\mathtt{S}(i+1, j+1)$ depending on $\mathtt{T}(i, j)$, and $(1, 0)$, for $\mathtt{T}(i+1, j)$ depending on $\mathtt{S}(i, j)$. So there are two uniform dependences, as the distance vector is constant for every dependence.

The determination of the dependence relations and the distance vectors for the loops in Example 5 and Example 7 are relatively simple. In real programs, however, various circumstances can make dependence analysis more complicated and time consuming. Sometimes it might even be impossible to determine the dependence relation of a program: for example, when a subscript of an array, which is read and written in various tasks, is a function of an input variable of the program. In that case, the dependence relations can only be established at runtime. A conservative approach,

in the sense that the discovered dependence structure includes all constraints of the true structure, is then to assume dependence between all respective tasks.

Loop Stride The dependence analysis presented so far is based on the assumption that the loop stride (i.e., increment or decrement of the index variable per iteration) is 1. In order to handle different strides, as they happen to appear in real loops, the loop must be normalized to a stride of 1—at least during the analysis phase. The normalization involves a transformation of all expressions where the original index variable is included (Banerjee et al. [17]).

Dependence Tests The area of dependence analysis in loops mainly concentrates on array subscripts that are linear functions of the index variables. But even with this restriction, the determination of a dependence can be quite time consuming, as exact solutions are based on integer programming. This, together with the fact that it is sometimes sufficient to know whether there is a dependence or not, led to the utilization of approximation methods. It suggests that these approximation methods often do not determine the distance vector; at most, they compute a direction vector, which is the vector of the signs of the distance vector's components. Some loop transformations only require knowledge of such a direction vector. Many so-called dependence test algorithms have been proposed in the past; consult, for example, Banerjee et al. [15, 16], Blume and Eigenmann [23], Eisenbeis and Sogno [58], Petersen and Padua [148], Pugh [155], Wolfe [202], and Yazici and Terziologu [212].

2.5.3 Control Dependence

In contrast to data dependence, control dependence is not caused by the transfer of data among tasks. Control dependence relations describe the control structure of a program (Banerjee [14], Towle [189]). Consider the sequence of statements in Example 8.

Example 8 Control Dependence

```
1: if u = 0 then
2:    v = w
3: else
4:    v = w + 1
5:    x = x - 1
6: end if
```

The tasks in lines 2, 4, and 5 are *control dependent* on the outcome of the `if` statement in line 1. In other words, tasks 2, 4, and 5 should not be executed until the statement of line 1 is evaluated.

Control dependence can be transformed into data dependence (Banerjee et al. [17]), and in this way the representation and analysis techniques for data dependence can be applied. The transformation proceeds by replacing the `if` statement with a

Boolean variable, say, *b*, to which the result of the `if` statement's argument evaluation is assigned. The control dependent tasks are rewritten, as shown in Example 9, so that they are only executed when *b* is true and false, respectively.

Example 9 Control Dependence of Example 8 Transformed to Data Dependence

```
1: b = [u = 0]
2: v = w when b
3: v = w + 1 when not b
4: x = x - 1 when not b
```

The operator `when` indicates that the statement to its left is only executed when its argument to the right is true. Resulting from the transformation, tasks 2–4 are now flow dependent on task 1, caused by the variable *b*.

Transforming control dependence into data dependence is described by Allen and Kennedy [11] and by Banerjee [14]. As mentioned earlier, the transformation of control dependence into data dependence permits one to unify the treatment of both dependence types. The graph representation of programs can benefit from this and only reflect data dependence relations, as described in Section 3.2.

2.6 CONCLUDING REMARKS

In this chapter, parallel systems and their programming were reviewed. This review focused on the parallel architectures and the communication networks, both of which determine the communication behavior of a parallel system. In order to produce accurate and efficient schedules, a good understanding of the communication behavior is crucial. Scheduling is a crucial part of the parallelization process in parallel programming. The process as a whole was studied and the two steps that precede scheduling—subtask decomposition and dependence analysis—were analyzed in detail. Altogether, this chapter established the foundation, background, and terminology for the following chapters.

Naturally, the discussion of parallel architectures and their networks in Sections 2.1 and 2.2 cannot be complete. For more details and further reading the reader should refer to Cosnard and Trystram [45], Culler and Singh [48], Grama et al. [82], Hamacher et al. [84], Hennessy and Patterson [88], and Kung [109]. Many of the system examples in Section 2.1 are taken from the *Overview of Recent Supercomputers* [193], published yearly since 1996 by van der Steen and Dongarra on the site of the TOP500 Supercomputer Sites [186].

Decomposition techniques are studied in greater detail in Grama et al. [82], on which Section 2.4.2 is based.

The dependence analysis discussed in Section 2.5 is based primarily on the publications by Banerjee et al. [17] and by Wolfe [204]. More on dependence and its analysis, especially in loops, can be found, apart from the references given in the text, in the literature by Allen and Kennedy [12], Banerjee et al. [15, 16], Blume et al. [24], Polychronopoulos [152], and Wolfe [202, 203].

Plenty of general books on the aspects of parallel computing and programming have been published: for example, Culler and Singh [48], El-Rewini and Lewis [64, 122, 123], Foster [69], Grama et al. [82], Hwang and Briggs [95], Kumar et al. [108], Leighton [118], Parhami [143], Polychronopoulos [152], Quinn [156], and Wilkinson and Allen [200].

Before starting the analysis of scheduling in Chapter 4, Chapter 3 will introduce graph models for the representation of parallel programs. Task scheduling is based on such models.

2.7 EXERCISES

2.1 Figure 2.1(*a*) in Section 2.1.1 shows a schematic diagram of an SIMD system. This can be interpreted as a parallel system consisting of one vector processor. Draw the schematic diagram of a system consisting of multiple vector processors, that is, a multiple SIMD system.

2.2 From a memory architecture point of view (Section 2.1.2), there are two fundamentally different approaches to the design of a parallel system: shared-memory architecture and message passing architecture.

(a) Describe the difference between the two architectures.

(b) Which architecture has a cluster of workstations interconnected through a LAN (local area network)?

(c) Which architecture has a single workstation equipped with four processors?

(d) On which of these two systems can the message passing programming model be used? Which of these two systems can share the memory in a global address space?

2.3 In Section 2.1.2 the memory architectures of parallel systems are studied. Conceptually, the distributed shared-memory architecture is the most interesting one, but also the most complex. In a system with such an architecture:

(a) What is meant by local and remote memory access? What is the usual major difference between them?

(b) Is such a system likely to be a UMA or a NUMA system?

(c) What is the danger of NUMA systems for their efficiency?

(d) Why are memory hierarchies used in distributed shared-memory systems?

(e) Can the message passing programming model be used on this system?

2.4 LogP is an advanced model for parallel programming (Section 2.1.3).

(a) Why was the LogP model introduced?

(b) What does "LogP" stand for?

2.5 The communication network is an essential part of a parallel system. Many different network topologies are discussed in Section 2.2. For a network that

connects $p = 128$ processors, calculate the number of links, the diameter, and the bisection width for the following topologies:

(a) Fully connected, cyclic grid (2D mesh), cyclic torus (3D mesh), and hypercube.

(b) Crossbar, butterfly, and binary tree. Also, calculate the number of switches for each network.

2.6 A butterfly network is a blocking network (Section 2.2.2), which means that certain combinations of source–destination connections are mutually exclusive. Figure 2.10 depicts a 3D butterfly network with eight source and destination lines. Which of the following connection pairs block each other: 2–6 and 7–1, 3–4 and 2–7, 5–0 and 0–5, 3–5 and 6–7, 1–1 and 3–0, 4–2 and 1–3?

2.7 Parallel programming implies the parallelization of a problem (Section 2.3). Describe the three main areas of the parallelization process.

2.8 Four quite common decomposition techniques are studied in Section 2.4.2: data decomposition, recursive decomposition, exploratory decomposition, and speculative decomposition. Which of these techniques seems indicated for parallelizing the following algorithms and applications: Quicksort (Cormen et al. [42]), chess game, fast fourier transform (FFT) (Cormen et al. [42]), and path finding algorithm. Briefly describe how you would decompose these problems.

2.9 In Section 2.5.1, three different types of data dependence are discussed: flow dependence, antidependence, and output dependence. Consider the following fragment of a program:

```
1: b = a * 3
2: a = b * 7
3: c = a + b - d
4: a = c - 11
5: d = a - b + c
```

Identify all dependences and classify their types.

2.10 Exercise 2.9 asks you to identify and classify the data dependences in a small code fragment. In loops, data dependence usually arises through the access of array elements (Section 2.5.2). Consider the following loop:

```
for i = 3 to 50 do
  S:  A(i) = B(2*i) + 1
  T:  B(i+1) = A(i-1) + A(i+1)
  U:  C(2*i) = A(i-2) + E(i+2)
  V:  D(i) = C(2*i+1) + B(i+1)
end for
```

(a) Without considering the subscripts of the array variables, which of the array variables $A, B, C, D,$ and E are potentially involved in the dependences among the tasks?

(b) Identify all *intra*iteration dependences and their types.

(c) Identify all *inter*iteration dependences and their types. What is the distance of each dependence? For nonuniform dependences, give the minimum and the maximum distance.

2.11 Dependence analysis of nested loops is a straightforward generalization of the dependence analysis of single loops. Essentially, the index variable becomes an index vector and the dependence distance becomes a dependence vector (Section 2.5.2). Consider the following example of a double loop:

```
for i = 1 to 10 do
  for j = 1 to 10 do
    S: A(i+1,j-1) = B(i,j) + E(i+1,j+1)
    T: B(i,j-1) = A(i,j) + 1
    U: C(2*i+1,j) = A(i+1,j-1) + 4
    V: D(i,j) = C(i,j-1) - 1
  end for
end for
```

(a) Identify all dependences and their types. What is the distance vector of each dependence? For nonuniform dependences, give the minimum and the maximum distance for each component of the vector.

(b) Can the loop order be swapped, that is, the first loop iterates over *j* and the second over *i*?

2.12 Control dependence can be converted into data dependence, using a `when` operator as shown in Section 2.5.3. Convert the control dependences in the following code fragment into data dependences.

```
1: if u > 5 then
2:   v = u - 5
3: else if  u < 2 then
4:   v = u + 2
5: else
6:   w = u - 2
7: end if
```