Overview of Parallel Systems

2

Chapter Summary
In this chapter we overview the most important basic notions, concepts, and theoretical results concerning parallel computation. We describe three basic models of parallel computation, then focus on various topologies for interconnection of parallel computer nodes. After and a brief introduction to analysis of parallel computation complexity, we finally explain two important laws of parallel computation, the Amdahl's law and Brents's theorem.

2.1 History of Parallel Computing, Systems and Programming

Let Π be an arbitrary computational problem which is to be solved by a computer. Usually our first objective is to design an *algorithm* for solving Π. Clearly, the class of all algorithms is infinite, but we can partition it into two subclasses, the class of all sequential algorithms and the class of all parallel algorithms.[1] While a *sequential algorithm* performs one operation in each step, a **parallel algorithm** may perform multiple operations in a single step. In this book, we will be mainly interested in parallel algorithms. So, our objective is to design a parallel algorithm for Π.

Let P be an arbitrary parallel algorithm. We say that there is **parallelism** in P. The parallelism in P can be exploited by various *kinds* of **parallel computers**. For instance, multiple operations of P may be executed simultaneously by multiple **processing units** of a parallel computer C_1; or, perhaps, they may be executed by multiple pipelined functional units of a single-processor computer C_2. After all, P

[1] There are also other divisions that partition the class of all algorithms according to other criteria, such as *exact* and *non-exact* algorithms; or *deterministic* and *non-deterministic* algorithms. However, in this book we will not divide algorithms systematically according to these criteria.

R. Trobec et al., *Introduction to Parallel Computing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-98833-7_2

can always be *sequentially* executed on a single-processor computer C_3, simply by executing P's potentially parallel operations one by one in succession.

Let $C(p)$ be a parallel computer of the kind C which contains p processing units. Naturally, we expect the **performance** of P on $C(p)$ to depend both on C and p. We must, therefore, clearly distinguish between the **potential parallelism** in P on the one side, and the **actual capability** of $C(p)$ to execute, in parallel, multiple operations of P, on the other side. So the performance of the algorithm P on the parallel computer $C(p)$ depends on $C(p)$'s capability to exploit P's potential parallelism.

Before we continue, we must unambiguously define what we really mean by the term "performance" of a parallel algorithm P. Intuitively, the "performance" might mean the time required to execute P on $C(p)$; this is called the **parallel execution time** (or, **parallel runtime**) of P on $C(p)$, which we will denote by

$$T_{\text{par}}.$$

Alternatively, we might choose the "performance" to mean how many times is the parallel execution of P on $C(p)$ faster than the sequential execution of P; this is called the **speedup** of P on $C(p)$,

$$S \stackrel{\text{def}}{=} \frac{T_{\text{seq}}}{T_{\text{par}}}.$$

So parallel execution of P on $C(p)$ is S-times faster than sequential execution of P. Next, we might be interested in how much of the speedup S is, on average, due to each of the processing units. Put differently, the term "performance" might be understood as the average contribution of each of the p processing units of $C(p)$ to the speedup; this is called the **efficiency** of P on $C(p)$,

$$E \stackrel{\text{def}}{=} \frac{S}{p}.$$

Since $T_{\text{par}} \leqslant T_{\text{seq}} \leqslant p \cdot T_{\text{par}}$, it follows that speedup is bounded above by p and efficiency is bounded above by

$$E \leqslant 1.$$

This means that, for *any* C *and* p, the parallel execution of P on $C(p)$ can be at most p times faster than the execution of P on a single processor. And the efficiency of the parallel execution of P on $C(p)$ can be at most 1. (This is when each processing unit is continually engaged in the execution of P, thus contributing $\frac{1}{p}$-th to its speedup.) Later, in Sect. 2.5, we will involve one more parameter to these definitions.

From the above definitions we see that both speedup and efficiency depend on T_{par}, the parallel execution time of P on $C(p)$. This raises new questions:

How do we determine T_{par}?
How does T_{par} depend on C (the kind of a parallel computer) ?
Which properties of C must we take into account in order to determine T_{par}?

These are important general questions about parallel computation which must be answered prior to embarking on a practical design and analysis of parallel algorithms. The way to answer these questions is to appropriately **model** parallel computation.

2.2 Modeling Parallel Computation

Parallel computers vary greatly in their organization. We will see in the next section that their processing units may or may not be directly connected one to another; some of the processing units may share a common memory while the others may only own local (private) memories; the operation of the processing units may be synchronized by a common clock, or they may run each at its own pace. Furthermore, usually there are architectural details and hardware specifics of the components, all of which show up during the actual design and use of a computer. And finally, there are technological differences, which manifest in different clock rates, memory access times etc. Hence, the following question arises:

Which properties of parallel computers **must be considered**
and which **may be ignored** *in the design and analysis of parallel algorithms?*

To answer the question, we apply ideas similar to those discovered in the case of sequential computation. There, various **models of computation** were discovered.[2] In short, the intention of each of these models was to *abstract the relevant* properties of the (sequential) computation *from the irrelevant* ones.

In our case, a model called the **Random Access Machine (RAM)** is particularly attractive. Why? The reason is that RAM distills the important properties of the general-purpose sequential computers, which are still extensively used today, and which have actually been taken as the *conceptual basis for modeling of parallel computing and parallel computers*. Figure 2.1 shows the structure of the RAM.

[2] Some of these models of computation are the μ-recursive functions, recursive functions, λ-calculus, Turing machine, Post machine, Markov algorithms, and RAM.

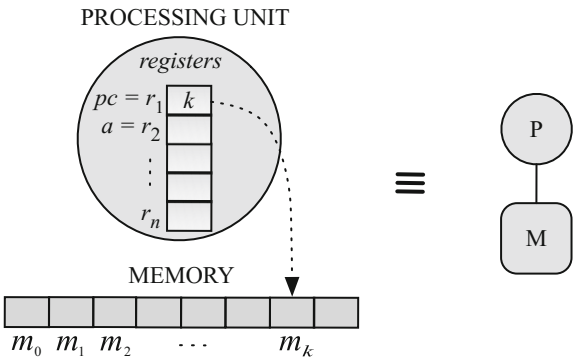

Fig. 2.1 The RAM model of computation has a memory M (containing program instructions and data) and a processing unit P (executing instructions on data)

Here is a brief description of RAM:

- The RAM consists of a **processing unit** and a **memory**. The *memory* is a potentially infinite sequence of equally sized locations $m_0, m_1, \ldots$. The index i is called the *address* of m_i. Each location is directly accessible by the processing unit: given an arbitrary i, reading from m_i or writing to m_i is accomplished in constant time. *Registers* are a sequence $r_1 \ldots r_n$ of locations in the processing unit. Registers are directly accessible. Two of them have special roles. *Program counter pc* $(=r_1)$ contains the address of the location in the memory which contains the instruction to be executed next. *Accumulator a* $(=r_2)$ is involved in the execution of each instruction. Other registers are given roles as needed. The *program* is a finite sequence of instructions (similar to those in real computers).
- Before the RAM is started, the following is done: (a) a program is loaded into successive locations of the memory starting with, say, m_0; (b) input data are written into empty memory locations, say after the last program instruction.
- From now on, the RAM operates independently in a mechanical stepwise fashion as instructed by the program. Let $pc = k$ at the beginning of a step. (Initially, $k = 0$.) From the location m_k, the instruction `I` is read and started. At the same time, pc is incremented. So, when `I` is completed, the next instruction to be executed is in m_{k+1}, unless `I` was one of the instructions that change pc (e.g. jump instructions).

So the above question boils down to the following question:

What is the **appropriate model** *of parallel computation?*

It turned out that finding an answer to this question is substantially more challenging than it was in the case of sequential computation. Why? Since there are many ways to organize parallel computers, there are also many ways to model them; and what is difficult is to select a *single* model that will be appropriate for *all* parallel computers.

As a result, in the last decades, researchers proposed *several* models of parallel computation. However, no common agreement has been reached about which is the right one. In the following, we describe those that are based on RAM.[3]

2.3 Multiprocessor Models

A **multiprocessor model** is a model of parallel computation that builds on the RAM model of computation; that is, it generalizes the RAM. How does it do that?

It turns out that the generalization can be done in three essentially different ways resulting in three different multiprocessor models. Each of the three models has some number p ($\geqslant 2$) of processing units, but the models differ in the organization of their memories and in the way the processing units access the memories.

The models are called the

- *Parallel Random Access Machine* (*PRAM*),
- *Local Memory Machine* (*LMM*), and
- *Modular Memory Machine* (*MMM*).

Let us describe them.

2.3.1 The Parallel Random Access Machine

The **Parallel Random Access Machine**, in short **PRAM** model, has p processing units that are all connected to a common *unbounded shared memory* (Fig. 2.2). Each processing unit can, in *one* step, access *any* location (word) in the shared memory by issuing a memory request *directly* to the shared memory.

The PRAM model of parallel computation is idealized in several respects. First, there is no limit on the number p of processing units, except that p is finite. Next, also idealistic is the assumption that a processing unit can access any location in the shared memory in one single step. Finally, for words in the shared memory it is only assumed that they are of the same size; otherwise they can be of arbitrary finite size.

Note that in this model there is no interconnection network for transferring memory requests and data back and forth between processing units and shared memory. (This will radically change in the other two models, the LMM (see Sect. 2.3.2) and the MMM (see Sect. 2.3.3)).

However, the assumption that any processing unit can access any memory location in one step is **unrealistic**. To see why, suppose that processing units P_i and P_j

[3] In fact, currently the research is being pursued also in other, non-conventional directions, which do not build on RAM or any other conventional computational models (listed in previous footnote). Such are, for example, *dataflow computation* and *quantum computation*.

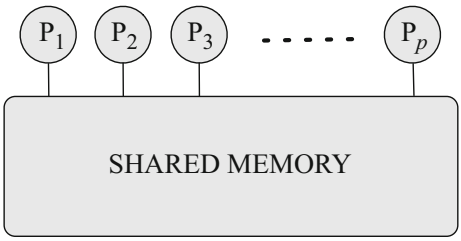

Fig. 2.2 The PRAM model of parallel computation: p processing units share an unbounded memory. Each processing unit can in one step access any memory location

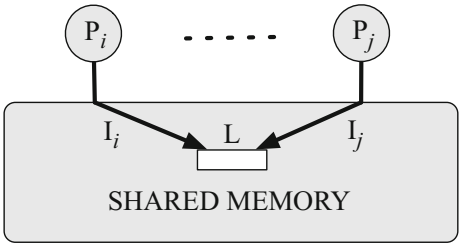

Fig. 2.3 Hazards of simultaneous access to a location. Two processing units simultaneously issue instructions each of which needs to access the same location L

simultaneously issue instructions I_i and I_j where both instructions intend to access (for reading from or writing to) the *same* memory location L (see Fig. 2.3).

Even if a truly simultaneous physical access to L had been possible, such an access could have resulted in unpredictable contents of L. Imagine what would be the contents of L after simultaneously writing 3 and 5 into it. Thus, it is reasonable to assume that, eventually, actual accesses of I_i and I_j to L are somehow, on the fly serialized (sequentialized) by hardware so that I_i and I_j physically access L *one after the other*.

Does such an implicit serialization neutralize all hazards of simultaneous access to the same location? Unfortunately not so. The reason is that the order of physical accesses of I_i and I_j to L is *unpredictable*: after the serialization, we cannot know whether I_i will physically access L *before* or *after* I_j.

Consequently, also the effects of instructions I_i and I_j are unpredictable (Fig. 2.3). Why? If *both* P_i and P_j want to *read* simultaneously from L, the instructions I_i and I_j will both read the same contents of L, regardless of their serialization, so both processing units will receive the same contents of L—as expected. However, if *one* of the processing units wants to *read* from L and the *other* simultaneously wants to *write* to L, then the data received by the reading processing unit will depend on whether the reading instruction has been serialized before or after the writing instruction. Moreover, if both P_i and P_j simultaneously attempt to *write* to L, the resulting contents of L will depend on how I_i and I_j have been serialized, i.e., which of I_i and I_j was the last to physically write to L.

In sum, simultaneous access to the same location may end in unpredictable data in the accessing processing units as well as in the accessed location.

In view of these findings it is natural to ask: Does this unpredictability make the PRAM model useless? The answer is no, as we will see shortly.

The Variants of PRAM
The above issues led researchers to define several variations of PRAM that differ in

(i) which sorts of simultaneous accesses to the same location are allowed; and
(ii) the way in which unpredictability is avoided when simultaneously accessing the same location.

The variations are called the

- *Exclusive Read Exclusive Write PRAM* (*EREW-PRAM*),
- *Concurrent Read Exclusive Write PRAM* (*CREW-PRAM*), and
- *Concurrent Read Concurrent Write PRAM* (*CRCW-PRAM*).

We now describe them into more detail:

- **EREW**-PRAM. This is the most realistic of the three variations of the PRAM model. The EREW-PRAM model does not support simultaneous accessing to the same memory location; if such an attempt is made, the model stops executing its program. Accordingly, the implicit assumption is that programs running on EREW-PRAM never issue instructions that would simultaneously access the same location; that is, any access to any memory location must be *exclusive*. So the construction of such programs is the responsibility of algorithm designers.
- **CREW**-PRAM. This model supports simultaneous reads from the same memory location but requires exclusive writes to it. Again, the burden of constructing such programs is on the algorithm designer.
- **CRCW**-PRAM. This is the least realistic of the three versions of the PRAM model. The CRCW-PRAM model allows simultaneous reads from the same memory location, simultaneous writes to the same memory location, and simultaneous reads from and writes to the same memory location. However, to avoid unpredictable effects, different additional restrictions are imposed on simultaneous writes. This yields the following versions of the model CRCW-PRAM:
  - **CONSISTENT**-CRCW-PRAM. Processing units may simultaneously attempt to write to L, but it is assumed that they *all need to write the same value* to L. To guarantee that is, of course, the responsibility of the algorithm designer.
  - **ARBITRARY**-CRCW-PRAM. Processing units may simultaneously attempt to write to L (not necessarily the same value), but it is assumed that *only one of them will succeed*. Which processing unit will succeed is not predictable, so the programmer must take this into account when designing the algorithm.
  - **PRIORITY**-CRCW-PRAM. There is a priority order imposed on the processing units; e.g., the processing unit with smaller index has higher priority. Processing units may simultaneously attempt to write to L, but it is assumed that only the one with *the highest priority will succeed*. Again, algorithm designer must foresee and mind every possible situation during the execution.

– **FUSION**-CRCW-PRAM. Processing units may simultaneously attempt to write to L, but it is assumed that

  ◇ first a particular operation, denoted by $\circ$, will be applied on-the-fly to all the values $v_1, v_2, \ldots, v_k$ to be written to L, and
  ◇ only then the result $v_1 \circ v_2 \circ \cdots \circ v_k$ of the operation $\circ$ will be written to L.

  The operation $\circ$ is assumed to be associative and commutative, so that the value of the expression $v_1 \circ v_2 \circ \cdots \circ v_k$ does not dependent on the order of performing the operations $\circ$. Examples of the operation $\circ$ are the sum (+), product ($\cdot$), maximum (max), minimum (min), logical conjunction ($\wedge$), and logical disjunction ($\vee$).

⋆ The Relative Power of the Variants
As the restrictions of simultaneous access to the same location are relaxed when we pass from EREW-PRAM to CREW-PRAM and then to CRCW-PRAM, the variants of PRAM are becoming less and less realistic. On the other hand, as the restrictions are dropped, it is natural to expect that the variants may be gaining in their power. So we pose the following question:

Do EREW-PRAM, CREW-PRAM and CRCW-PRAM differ in their power?

The answer is *yes, but not too much.* The foggy "too much" is clarified in the next Theorem, where CRCW-PRAM(p) denotes the CRCW-PRAM with p processing units, and similarly for the EREW-PRAM(p). Informally, the theorem tells us that by passing from the EREW-PRAM(p) to the "more powerful" CRCW-PRAM(p) the parallel execution time of a parallel algorithm *may* reduce by some factor; however, this factor is *bounded above* and, indeed, it is at most of the order $O(\log p)$.

Theorem 2.1 *Every algorithm for solving a computational problem Π on the CRCW-PRAM(p) is at most $O(\log p)$-times faster than the fastest algorithm for solving Π on the EREW-PRAM(p).*

Proof Idea We first show that CONSISTENT-CRCW-PRAM(p)'s simultaneous writings to the same location can be performed by EREW-PRAM(p) in $O(\log p)$ steps. Consequently, EREW-PRAM can simulate CRCW-PRAM, with slowdown factor $O(\log p)$. Then we show that this slowdown factor is tight, that is, there *exists* a computational problem Π for which the slowdown factor is actually $\Theta(\log p)$. Such a Π is, for example, the problem of *finding the maximum of n numbers*. □

Relevance of the PRAM Model
We have explained why the PRAM model is unrealistic in the assumption of an immediately addressable, unbounded shared memory. Does this necessarily mean that the PRAM model is *irrelevant* for the purposes of practical implementation of parallel computation? The answer depends on *what we expect from the PRAM model* or, more generally, *how we understand the role of theory.*

When we strive to design an algorithm for solving a problem Π on PRAM, our efforts may not end up with a *practical* algorithm, ready for solving Π. However, the design may reveal something inherent to Π, namely, that Π is *parallelizable*. In other words, the design may detect in Π subproblems some of which could, at least in principle, be solved in parallel. In this case it usually proves that such subproblems are indeed *solvable in parallel* on the most liberal (and unrealistic) PRAM, the CRCW-PRAM.

At this point the importance of Theorem 2.1 becomes apparent: we can replace CRCW-PRAM by the *realistic* EREW-PRAM and solve Π on the latter. (All of that at the cost of a limited degradation in the speed of solving Π).

In sum, the relevance of PRAM is reflected in the following method:

1. *Design* a program P for solving Π on the model CRCW-PRAM(p), where p may depend on the problem Π. Note that the design of P for CRCW-PRAM is expected to be easier than the design for EREW-PRAM, simply because CRCW-PRAM has no simultaneous-access restrictions to be taken into account.
2. *Run* P on EREW-PRAM(p), which is assumed to be able to simulate simultaneous accesses to the same location.
3. *Use* Theorem 2.1 to guarantee that the parallel execution time of P on EREW-PRAM(p) is at most $O(\log p)$-times higher than it would be on the less realistic CRCW-PRAM(p).

2.3.2 The Local-Memory Machine

The **LMM model** has p processing units, each with its own **local memory** (Fig. 2.4). The processing units are connected to a common **interconnection network**. Each processing unit can access its own local memory *directly*. In contrast, it can access a *non*-local memory (i.e., local memory of another processing unit) only by sending a **memory request** through the interconnection network.

The assumption is that all *local operations*, including accessing the local memory, take *unit time*. In contrast, the time required to access a non-local memory depends on

- the capability of the interconnection network and
- the pattern of coincident non-local memory accesses of other processing units as the accesses may congest the interconnection network.

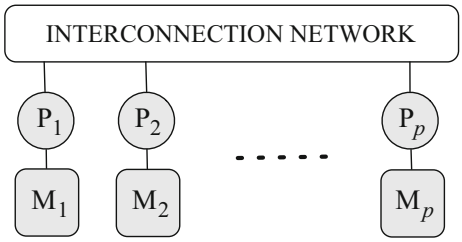

Fig. 2.4 The LMM model of parallel computation has p processing units each with its local memory. Each processing unit directly accesses its local memory and can access other processing unit's local memory via the interconnection network

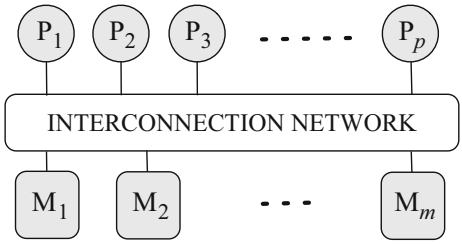

Fig. 2.5 The MMM model of parallel computation has p processing units and m memory modules. Each processing unit can access any memory module via the interconnection network. There are no local memories to the processing units

2.3.3 The Memory-Module Machine

The **MMM model** (Fig. 2.5) consists of p processing units and m **memory modules** each of which can be accessed by any processing unit via a *common interconnection network*. There are no local memories to processing units. A processing unit can access the memory module by sending a *memory request* through the interconnection network.

It is assumed that the processing units and memory modules are arranged in such a way that—when there are no coincident accesses—the time for any processing unit to access any memory module is roughly uniform. However, when there are coincident accesses, the access time depends on

- the capability of the interconnection network and
- the pattern of coincident memory accesses.

2.4 The Impact of Communication

We have seen that both LMM model and MMM model explicitly use interconnection networks to convey memory requests to the non-local memories (see Figs. 2.4 and 2.5). In this section we focus on the role of an interconnection network in a

multiprocessor model and its impact on the the parallel time complexity of parallel algorithms.

2.4.1 Interconnection Networks

Since the dawn of parallel computing, the major hallmark of a parallel system have been the type of *the central processing unit* (CPU) and the *interconnection network*. This is now changing. Recent experiments have shown that execution times of most real world parallel applications are becoming more and more dependent on the *communication time* rather than on the calculation time. So, as the number of cooperating processing units or computers increases, the performance of interconnection networks is becoming more important than the performance of the processing unit. Specifically, the interconnection network has great impact on the *efficiency* and *scalability* of a parallel computer on most real world parallel applications. In other words, high performance of an interconnection network may ultimately reflect in higher speedups, because such an interconnection network can shorten the overall parallel execution time as well as increase the number of processing units that can be efficiently exploited.

The performance of an interconnection network depends on several factors. Three of the most important are the *routing*, the *flow-control algorithms*, and the *network topology*. Here **routing** is the process of selecting a path for traffic in an interconnection network; **flow control** is the process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver; and **network topology** is the arrangement of the various elements, such as communication nodes and channels, of an interconnection network.

For the routing and flow-control algorithms efficient techniques are already known and used. In contrast, network topologies haven't been adjusting to changes in technological trends as promptly as the routing and flow-control algorithms. This is one reason that many network topologies which were discovered soon after the very birth of parallel computing are still being widely used. Another reason is the freedom that end users have when they are choosing the appropriate network topology for the anticipated usage. (Due to modern standards, there is no such freedom in picking or altering routing or flow-control algorithms). As a consequence, a further step in performance increase can be expected to come from the improvements in the topology of interconnection networks. For example, such improvements should enable interconnection networks to dynamically adapt to the current application in some optimal way.

2.4.2 Basic Properties of Interconnection Networks

We can classify interconnection networks in many ways and characterize them by various parameters. For defining most of these parameters, *graph theory* is the most elegant mathematical framework. More specifically, an interconnection network can

be modeled as a **graph** $G(N, C)$, where N is a set of **communication nodes** and C is a set of **communication links** (or, **channels**) between the communication nodes. Based on this graph-theoretical view of interconnection networks, we can define parameters that represent both **topological properties** and **performance properties** of interconnection networks. Let us describe both kinds of properties.

Topological Properties of Interconnection Networks
The most important topological properties of interconnection networks, defined by graph-theoretical notions, are the

- node degree,
- regularity,
- symmetry,
- diameter,
- path diversity, and
- expansion scalability.

In the following we define each of them and give comments where appropriate:

- The **node degree** is the number d of channels through which a communication node is connected to other communication nodes. Notice, that node degree includes only the *ports for the network communication*, although a communication node also needs ports for the connection to the processing element(s) and ports for service or maintenance channels.
- An interconnection network is said to be **regular** if all communication nodes have the same node degree; that is, there is a $d > 0$ such that every communication node has node degree d.
- An interconnection network is said to be **symmetric** if all communication nodes possess the "same view" of the network; that is, there is a homomorphism that maps any communication node to any other communication node. In a symmetric interconnection network, the load can be evenly distributed through all communication nodes, thus reducing congestion problems. Many real implementations of interconnection networks are based on *symmetric regular* graphs because of their fruitful topological properties that lead to a simple routing and fair load balancing under the uniform traffic.
- In order to move from a source node to a destination node, a packet must traverse through a series of elements, such as routers or switches, that together comprise a *path* (or, *route*) between the source and the destination node. The number of communication nodes traversed by the packet along this path is called the **hop count**. In the best case, two nodes communicate through the path which has the **minimum hop count**, l, taken over all paths between the two nodes. Since l may vary with the source and destination nodes, we also use the **average distance**, l_{avg}, which is average l taken over all possible pairs of nodes. An important characteristic of any topology is the **diameter**, l_{max}, which is the maximum of all the minimum hop counts, taken over all pairs of source and destination nodes.

- In an interconnection network, there may exist multiple paths between two nodes. In such case, the nodes can be connected in many ways. A packet starting at source node will have at its disposal multiple routes to reach the destination node. The packet can take different routes (or even different continuations of a traversed part of a route) depending on the current situation in the network. An interconnection network that has high **path diversity** offers more alternatives when packets need to seek their destinations and/or avoid obstacles.
- **Scalability** is (i) the capability of a system to handle a growing amount of work, or (ii) the potential of the system to be enlarged to accommodate that growth. The scalability is important at every level. For example, the basic building block must be easily connected to other blocks in a uniform way. Moreover, the same building block must be used to build interconnection networks of different sizes, with only a small performance degradation for the maximum-size parallel computer. Interconnection networks have important impact on scalability of parallel computers that are based on the LMM or MMM multiprocessor model. To appreciate that, note that scalability is limited if node degree is fixed.

Performance Properties of Interconnection Networks
The main performance properties of interconnection networks are the

- channel bandwidth,
- bisection bandwidth, and
- latency.

We now define each of them and give comments where appropriate:

- **Channel bandwidth**, in short bandwidth, is the amount of data that is, or theoretically could be, communicated through a channel in a given amount of time. In most cases, the channel bandwidth can be adequately determined by using a simple **model of communication** which advocates that the **communication time** t_{comm}, needed to communicate given data through the channel, is the sum $t_{\text{s}} + t_{\text{d}}$ of the **start-up time** t_{s}, needed to set-up the channel's software and hardware, and the **data transfer time** t_{d}, where $t_{\text{d}} = mt_{\text{w}}$, the product of the number of words making up the data, m, and the **transfer time per one word**, t_{w}. Then the channel bandwidth is $1/t_{\text{w}}$.
- A given interconnection network can be *cut* into two (almost) equal-sized components. Generally, this can be done in many ways. Given a cut of the interconnection network, the *cut-bandwidth* is the sum of channel bandwidths of all channels connecting the two components. The smallest cut-bandwidth is called the **bisection bandwidth** (BBW) of the interconnection network. The corresponding cut is the *worst-case cut* of the interconnection network. Occasionally, the **bisection bandwidth per node** (BBWN) is needed; we define it as BBW divided by $|N|$, the number of nodes in the network. Of course, both BBW and BBWN depend on the topology of the network and the channel bandwidths. All in all, increasing

the bandwidth of the interconnection network can have as beneficial effects as increasing the CPU clock (recall Sect. 2.4.1).
- **Latency** is the time required for a packet to travel from the source node to the destination node. Many applications, especially those using short messages, are latency sensitive in the sense that efficiencies of these applications strongly depend on the latency. For such applications, their software overhead may become a major factor that influences the latency. Ultimately, the latency is bounded below by the time in which light traverses the physical distance between two nodes.

The transfer of data from a source node to a destination node is measured in terms of various units which are defined as follows:

- **packet**, the smallest amount of data that can be transferred by hardware,
- **FLIT** (flow control digit), the amount of data used to allocate the buffer space in some flow-control techniques;
- **PHIT** (physical digit), the amount of data that can be transferred in a single cycle.

These units are closely related to the bandwidth and to the latency of the network.

Mapping Interconnection Networks into Real Space
An interconnection network of any given topology, even if defined in an abstract higher-dimensional space, eventually has to be **mapped** into the physical, three-dimensional (3D) space. This means that all the chips and printed-circuit boards making up the interconnection network must be allocated physical places.

Unfortunately, this is not a trivial task. The reason is that mapping usually has to optimize certain, often contradicting, criteria while at the same time respecting various restrictions. Here are some examples:

- One such restriction is that the numbers of I/O pins per chip or per printed-circuit board are bounded above. A usual optimization criterion is that, in order to prevent the decrease of data rate, cables be as short as possible. But due to significant sizes of hardware components and due to physical limitations of 3D-space, mapping may considerably stretch certain paths, i.e., nodes that are close in higher-dimensional space may be mapped to distant locations in 3D-space.
- We may want to map processing units that communicate intensively as close together as possibly, ideally on the same chip. In this way we may minimize the impact of communication. Unfortunately, the construction of such optimal mappings is NP-hard optimization problem.
- An additional criterion may be that the power consumption is minimized.

2.4.3 Classification of Interconnection Networks

Interconnection networks can be classified into *direct* and *indirect* networks. Here are the main properties of each kind.

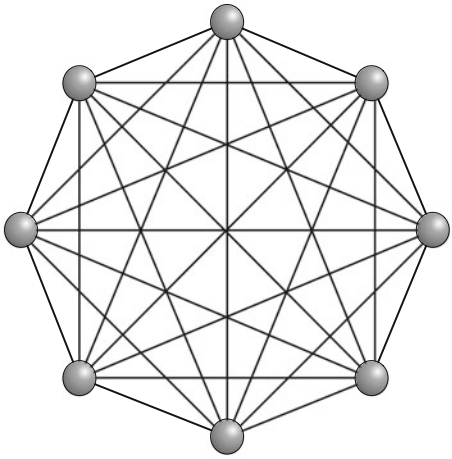

Fig. 2.6 A fully connected network with $n = 8$ nodes

Direct Networks

A network is said to be **direct** when each node is *directly connected* to its neighbors. How many neighbors can a node have? In a **fully connected network**, each of the $n = |N|$ nodes is directly connected to *all* the other nodes, so each node has $n - 1$ neighbors. (See Fig. 2.6).

Since such a network has $\frac{1}{2}n(n-1) = \Theta(n^2)$ direct connections, it can only be used for building systems with small numbers n of nodes. When n is large, each node is directly connected to a *proper subset* of other nodes, while the communication to the remaining nodes is achieved by routing messages through intermediate nodes. An example of such a direct interconnection network is the hypercube; see Fig. 2.13 on p. 28.

Indirect Networks

An **indirect** network connects the nodes through **switches**. Usually, it connects processing units on one end of the network and memory modules on the other end of the network. The simplest circuit for connecting processing units to memory modules is the **fully connected crossbar switch** (Fig. 2.7). Its advantage is that it can establish a connection between processing units and memory modules in an arbitrary way.

At each intersection of a horizontal and vertical line is a crosspoint. A crosspoint is a small switch that can be electrically opened (○) or closed (●), depending on whether the horizontal and vertical lines are to be connected or not. In Fig. 2.7 we see eight crosspoints closed simultaneously, allowing connections between the pairs (P_1, M_1), (P_2, M_3), (P_3, M_5), (P_4, M_4), (P_5, M_2), (P_6, M_6), (P_7, M_8) and (P_8, M_7) at the same time. Many other combinations are also possible.

Unfortunately, the fully connected crossbar has too large complexity to be used for connecting large numbers of input and output ports. Specifically, the number of crosspoints grows as pm, where p and m are the numbers of processing units and memory modules, respectively. For $p = m = 1000$ this amounts to a million crosspoints which is not feasible. (Nevertheless, for medium-sized systems, a crossbar

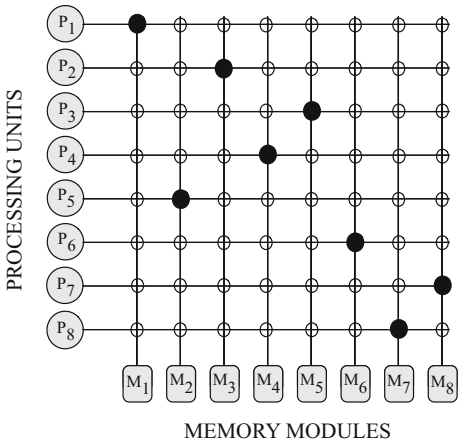

Fig. 2.7 A fully connected crossbar switch connecting 8 nodes to 8 nodes

design is workable, and small fully connected crossbar switches are used as basic building blocks within larger switches and routers).

This is why indirect networks connect the nodes through many switches. The switches themselves are usually connected to each other in **stages**, using a regular connection pattern between the stages. Such indirect networks are called the *multi-stage* interconnection networks; we will describe them in more detail on p. 29.

Indirect networks can be further classified as follows:

- A **non-blocking** network can connect any idle source to any idle destination, regardless of the connections already established across the network. This is due to the network topology which ensures the existence of multiple paths between the source and destination.
- A **blocking rearrangeable** networks can rearrange the connections that have already been established across the network in such a way that a new connection can be established. Such a network can establish all possible connections between inputs and outputs.
- In a **blocking** network, a connection that has been established across the network may block the establishment of a new connection between a source and destination, even if the source and destination are both free. Such a network cannot always provide a connection between a source and an arbitrary free destination.

The distinction between direct and indirect networks is less clear nowadays. Every direct network can be represented as an indirect network since every node in the direct network can be represented as a router with its own processing element connected to other routers. However, for both direct and indirect interconnection networks, the full crossbar, as an ideal switch, is the heart of the communications.

2.4.4 Topologies of Interconnection Networks

It is not hard to see that there exist many network topologies capable of interconnecting p processing units and m memory modules (see Exercises). However, not every network topology is capable of conveying memory requests quickly enough to *efficiently* back up parallel computation. Moreover, it turns out that the network topology has a large influence on the *performance* of the interconnection network and, consequently, of parallel computation. In addition, network topology may incur considerable difficulties in the actual construction of the network and its cost.

In the last few decades, researchers have proposed, analyzed, constructed, tested, and used various network topologies. We now give an overview of the most notable or popular ones: the *bus*, the *mesh*, the *3D-mesh*, the *torus*, the *hypercube*, the *multistage network* and the *fat tree*.

The Bus

This is the simplest network topology. See Fig. 2.8. It can be used in both local-memory machines (LMMs) and memory-module machines (MMMs). In either case, all processing units and memory modules are connected to a single bus. In each step, at most one piece of data can be written onto the bus. This can be a request from a processing unit to read or write a memory value, or it can be the response from the processing unit or memory module that holds the value.

When in a memory-module machine a processing unit wants to read a memory word, it must first check to see if the bus is busy. If the bus is idle, the processing unit puts the address of the desired word on the bus, issues the necessary control signals, and waits until the memory puts the desired word on the bus. If, however, the bus is busy when a processing unit wants to read or write memory, the processing unit must *wait until the bus becomes idle*. This is where drawbacks of the bus topology become apparent. If there is a small number of processing units, say two or three, the contention for the bus is manageable; but for larger numbers of processing units, say 32, the contention becomes unbearable because most of the processing units will wait most of the time.

To solve this problem we add a *local cache to each processing unit*. The cache can be located on the processing unit board, next to the processing unit chip, inside the processing unit chip, or some combination of all three. In general, caching is not done on an individual word basis but on the basis of blocks that consist of, say, 64 bytes. When a word is referenced by a processing unit, the word's entire block

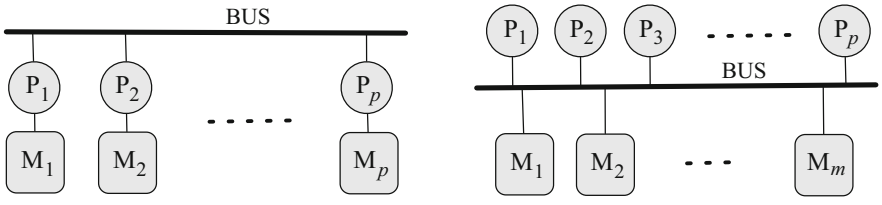

Fig. 2.8 The bus is the simplest network topology

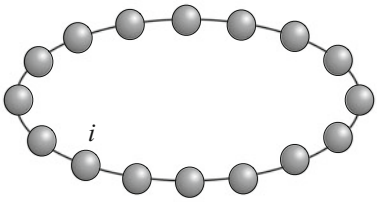

Fig. 2.9 A ring. Each node represents a processor unit with local memory

is fetched into the local cache of the processing unit. After that many reads can be satisfied out of the local cache. As a result, there will be less bus traffic, and the system will be able to support more processing units.

We see, that the practical advantages of using buses are that (i) they are simple to build, and (ii) it is relatively easy to develop protocols that allow processing units to cache memory values locally (because all processing units and memory modules can observe the traffic on the bus). The obvious disadvantage of using a bus is that the processing units must take turns accessing the bus. This implies that as more processing units are added to a bus, the average time to perform a memory access grows proportionately with the number of processing units.

The Ring

The ring is among the simplest and the oldest interconnection networks. Given n nodes, they are arranged in linear fashion so that each node has a distinct label i, where $0 \leqslant i \leqslant n-1$. Every node is connected to two neighbors, one to the left and one to the right. Thus, a node labeled i is connected to the nodes labeled $i+1 \bmod n$ and $i-1 \bmod n$ (see Fig. 2.9). The ring is used in local-memory machines (LMMs).

2D-Mesh

A two-dimensional mesh is an interconnection network that can be arranged in rectangular fashion, so that each switch in the mesh has a distinct label (i, j), where $0 \leqslant i \leqslant X-1$ and $0 \leqslant j \leqslant Y-1$. (See Fig. 2.10). The values X and Y determine the lengths of the sides of the mesh. Thus, the number of switches in a mesh is XY. Every switch, except those on the sides of the mesh, is connected to six neighbors: one to the north, one to the south, one to the east, and one to the west. So a switch

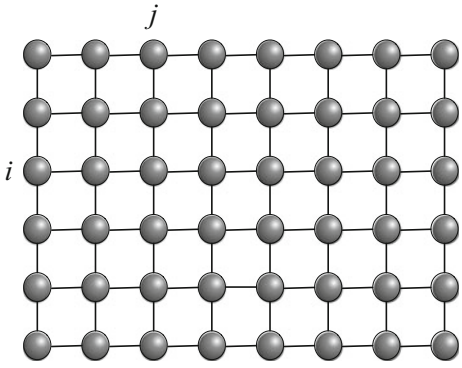

Fig. 2.10 A 2D-mesh. Each node represents a processor unit with local memory

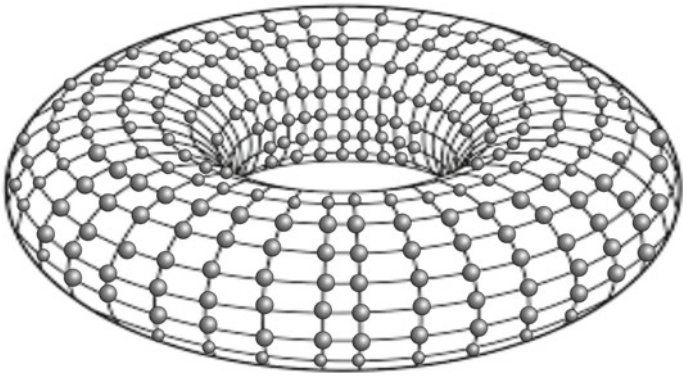

Fig. 2.11 A 2D-torus. Each node represents a processor unit with local memory

labeled (i, j), where $0 < i < X - 1$ and $0 < j < Y - 1$, is connected to the switches labeled $(i, j + 1)$, $(i, j - 1)$, $(i + 1, j)$, and $(i - 1, j)$.

Meshes typically appear in local-memory machines (LMMs): a processing unit (along with its local memory) is connected to each switch, so that remote memory accesses are made by routing messages through the mesh.

2D-Torus (Toroidal 2D-Mesh)

In the 2D-mesh, the switches on the sides have no connections to the switches on the opposite sides. The interconnection network that compensates for this is called the toroidal mesh, or just torus when $d = 2$. (See Fig. 2.11). Thus, in torus every switch located at (i, j) is connected to four other switches, which are located at $(i, j + 1 \bmod Y)$, $(i, j - 1 \bmod Y)$, $(i + 1 \bmod X, j)$ and $(i - 1 \bmod X, j)$.

Toruses appear in local-memory machines (LMMs): to each switch is connected a processing unit with its local memory. Each processing unit can access any remote memory by routing messages through the torus.

3D-Mesh and 3D-Torus

A three-dimensional mesh is similar to two-dimensional. (See Fig. 2.12). Now each switch in a mesh has a distinct label (i, j, k), where $0 \leqslant i \leqslant X - 1, 0 \leqslant j \leqslant Y - 1$, and $0 \leqslant k \leqslant Z - 1$. The values X, Y and Z determine the lengths of the sides of the mesh, so the number of switches in it is XYZ. Every switch, except those on the sides of the mesh, is now connected to six neighbors: one to the north, one to the south, one to the east, one to the west, one up, and one down. Thus, a switch labeled (i, j, k), where $0 < i < X - 1, 0 < j < Y - 1$ and $0 < k < Z - 1$, is connected to the switches $(i, j + 1, k)$, $(i, j - 1, k)$, $(i + 1, j, k)$, $(i - 1, j, k)$, $(i, j, k + 1)$ and $(i, j, k - 1)$. Such meshes typically appear in LMMs.

We can expand a 3D-mesh into a **toroidal 3D-mesh** by adding edges that connect nodes located at the opposite sides of the 3D-mesh. (Picture omitted). A switch labeled (i, j, k) is connected to the switches $(i + 1 \bmod X, j, k)$, $(i - 1 \bmod X, j, k)$, $(i, j + 1 \bmod Y, k)$, $(i, j - 1 \bmod Y, k)$, $(i, j, k + 1 \bmod Z)$ and $(i, j, k - 1 \bmod Z)$.

3D-meshes and toroidal 3D-meshes are used in local-memory machines (LMMs).

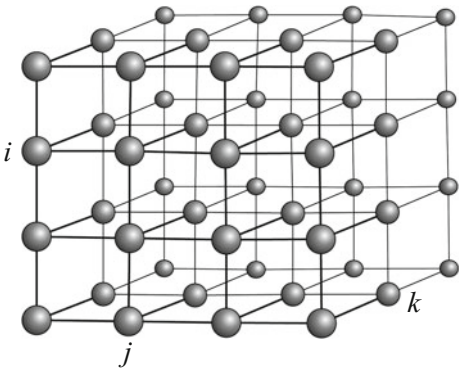

Fig. 2.12 A 3D-mesh. Each node represents a processor unit with local memory

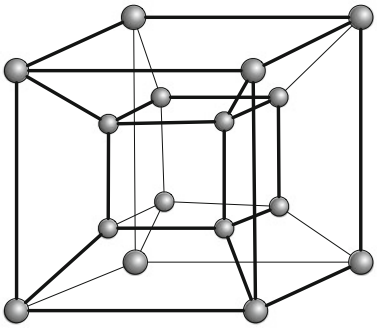

Fig. 2.13 A hypercube. Each node represents a processor unit with local memory

Hypercube
A hypercube is an interconnection network that has $n = 2^b$ nodes, for some $b \geqslant 0$. (See Fig. 2.13). Each node has a distinct label consisting of b bits. Two nodes are connected by a communication link if an only if their labels differ in precisely one bit location. Hence, each node of a hypercube has $b = \log_2 n$ neighbors.

Hypercubes are used in local-memory machines (LMMs).

⋆ The k-ary d-Cube Family of Network Topologies
Interestingly, the ring, the 2D-torus, the 3D-torus, the hypercube, and many other topologies all belong to one larger family of k-**ary** d-**cube** topologies.

Given $k \geqslant 1$ and $d \geqslant 1$, the k-ary d-cube topology is a *family* of certain "gridlike" topologies that share the fashion in which they are constructed. In other words, the k-ary d-cube topology is a *generalization* of certain topologies. The parameter d is called the **dimension** of these topologies and k is their **side length**, the number of nodes along each of the d directions. The fashion in which the k-ary d-cube topology is constructed is defined *inductively* (on the dimension d):

A k-ary d-cube is constructed from k other k-ary $(d-1)$-cubes
by connecting the nodes with identical positions into rings.

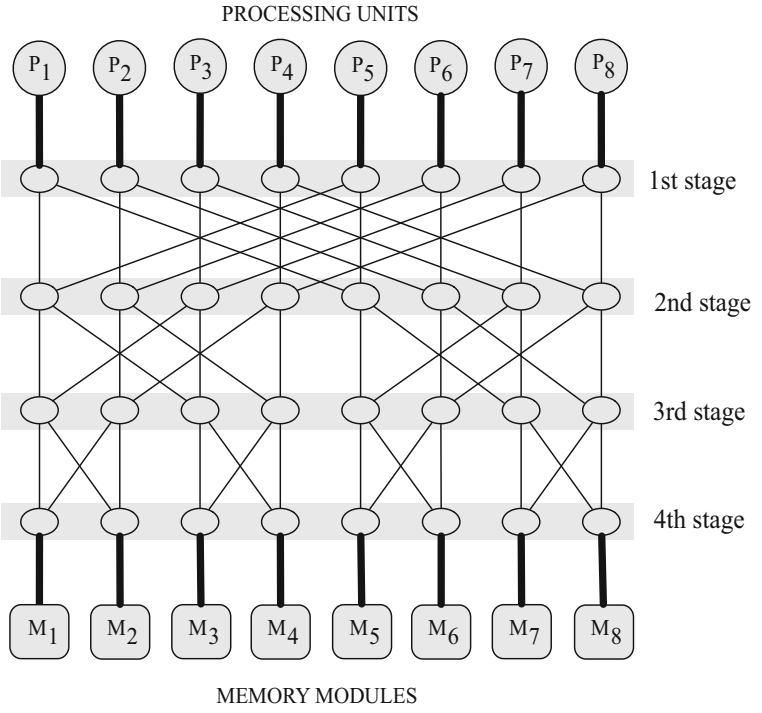

Fig. 2.14 A 4-stage interconnection network capable of connecting 8 processing units to 8 memory modules. Each switch ∘ can establish a connection between arbitrary pair of input and output channels

This inductive definition enables us to systematically construct actual k-ary d-cube topologies and analyze their topological and performance properties. For instance, we can deduce that a k-ary d-cube topology contains $n = k^d$ communication nodes and $c = dn = dk^d$ communication links, while the diameter is $l_{max} = \frac{dk}{2}$ and the average distance between two nodes is $l_{avg} = \frac{l_{max}}{2}$ (if k even) or $l_{avg} = d(\frac{k}{4} - \frac{1}{4k})$ (if k odd). Unfortunately, in spite of their simple recursive structure, the k-ary d-cubes have a poor expansion scalability.

Multistage Network

A multistage network connects one set of switches, called the *input switches*, to another set, called the *output switches*. The network achieves this through a sequence of *stages*, where each stage consists of switches. (See Fig. 2.14). In particular, the input switches form the first stage, and the output switches form the last stage. The number d of stages is called the *depth* of the multistage network. Usually, a multistage network allows to send a piece of data from any input switch to any output switch. This is done along a path that traverses all the stages of the network in order from 1 to d. There are many different multistage network topologies.

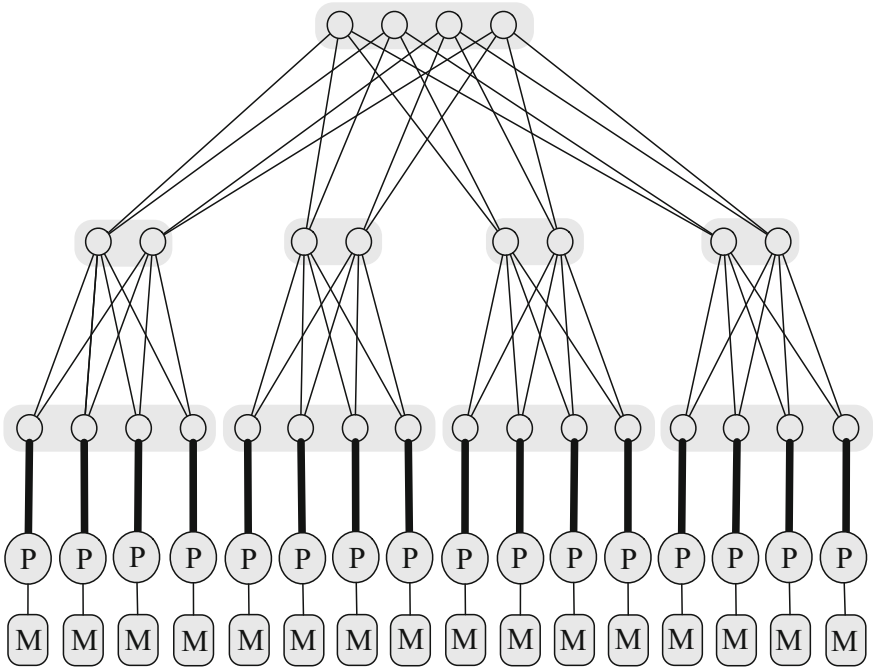

Fig. 2.15 A fat-tree. Each switch ○ can establish a connection between arbitrary pair of incident channels

Multistage networks are frequently used in memory-module machines (MMMs); there, processing units are attached to input switches, and memory modules are attached to output switches.

Fat Tree

A fat tree is a network whose structure is based on that of a tree. (See Fig. 2.15). However, in contrast to the usual tree where edges have the same thickness, in a fat tree, edges that are nearer the root of the tree are "fatter" (thicker) than edges that are further down the tree. The idea is that each node of a fat tree may represent many network switches, and each edge may represent many communication channels. The more channels an edge represents, the larger is its capacity and the fatter is the edge. So the capacities of the edges near the root of the fat tree are much larger than the capacities of the edges near the leaves.

Fat trees can be used to construct local-memory machines (LMMs): processing units along with their local memories are connected to the leaves of the fat tree, so that a message from one processing unit to another first travels up the tree to the least common ancestor of the two processing units and then down the tree to the destination processing unit.

2.5 Parallel Computational Complexity

In order to examine the complexity of computational problems and their parallel algorithms, we need some new basic notions. We will now introduce a few of these.

2.5.1 Problem Instances and Their Sizes

Let Π be a computational problem. In practice we are usually confronted with a particular *instance* of the problem Π. The instance is obtained from Π by replacing the variables in the definition of Π with actual data. Since this can be done in many ways, each way resulting in a different instance of Π, we see that the problem Π can be viewed as a *set* of all the possible instances of Π.

To each instance π of Π we can associate a natural number which we call the **size of the instance** π and denote by

$$\text{size}(\pi).$$

Informally, size(π) is roughly the amount of space needed to represent π in some way accessible to a computer and, in practice, depends on the problem Π.

For example, if we choose $\Pi \equiv$ "sort a given finite sequence of numbers," then $\pi \equiv$ "sort 0 9 2 7 4 5 6 3" is an instance of Π and size(π) = 8, the number of numbers to be sorted. If, however, $\Pi \equiv$ "Is n a prime number?", then "Is 17 a prime number?" is an instance π of Π with size(π) = 5, the number of bits in the binary representation of 17. And if Π is a problem about graphs, then the size of an instance of Π is often defined as the number of nodes in the actual graph.

Why do we need sizes of instances? When we examine how fast an algorithm A for a problem Π is, we usually want to know how A's execution time depends on the size of instances of Π that are input to A. More precisely, we want to find a function

$$T(n)$$

whose value at n will represent the execution time of A on instances of size n. As a matter of fact, we are mostly interested in the **rate of growth** of $T(n)$, that is, *how quickly $T(n)$ grows when n grows*.

For example, if we find that $T(n) = n$, then A's execution time is a **linear** function of n, so if we double the size of problem instances, A's execution time doubles too. More generally, if we find that $T(n) = n^{const}$ ($const \geqslant 1$), then A's execution time is a **polynomial** function of n; if we now double the size of problem instances, then A's execution time multiplies by 2^{const}. If, however, we find that $T(n) = 2^n$, which is an **exponential** function of n, then things become dramatic: doubling the size n of problem instances causes A to run 2^n-times longer! So, doubling the size from 10 to 20 and then to 40 and 80, the execution time of A increases 2^{10} ($\approx$thousand) times, then 2^{20} ($\approx$million) times, and finally 2^{40} ($\approx$thousand billion) times.

2.5.2 Number of Processing Units Versus Size of Problem Instances

In Sect. 2.1, we defined the parallel execution time T_{par}, speedup S, and efficiency E of a parallel program P for solving a problem Π on a computer $C(p)$ with p processing units. Let us augment these definitions so that they will involve the size n of the instances of Π. As before, the program P for solving Π and the computer $C(p)$ are tacitly understood, so we omit the corresponding indexes to simplify the notation. We obtain the parallel execution time $T_{\text{par}}(n)$, speedup $S(n)$, and efficiency $E(n)$ of *solving* Π's *instances of size* n:

$$S(n) \stackrel{\text{def}}{=} \frac{T_{\text{seq}}(n)}{T_{\text{par}}(n)},$$

$$E(n) \stackrel{\text{def}}{=} \frac{S(n)}{p}.$$

So let us pick an arbitrary n and suppose that we are only interested in solving instances of Π whose size is n. Now, if there are *too few* processing units in $C(p)$, i.e., p is too small, the potential parallelism in the program P will not be fully exploited during the execution of P on $C(p)$, and this will reflect in low speedup $S(n)$ of P. Likewise, if $C(p)$ has *too many* processing units, i.e., p is too large, some of the processing units will be idling during the execution of the program P, and again this will reflect in low speedup of P. This raises the following question that obviously deserves further consideration:

How many processing units p should have $C(p)$,
so that, for all instances of Π of size n, the speedup of P will be maximal?

It is reasonable to expect that the answer will depend somehow on the type of C, that is, on the multiprocessor model (see Sect. 2.3) underlying the parallel computer C. Until we choose the multiprocessor model, we may not be able to obtain answers of practical value to the above question. Nevertheless, we *can* make some general observations that hold for any type of C. First observe that, in general, if we let n grow then p must grow too; otherwise, p would eventually become too small relative to n, thus making $C(p)$ incapable of fully exploiting the potential parallelism of P. Consequently, we may view p, the number of processing units that are needed to maximize speedup, to be some function of n, the size of the problem instance at hand. In addition, intuition and practice tell us that a larger instance of a problem requires at least as many processing units as required by a smaller one. In sum, we can set

$$p = f(n),$$

where $f : \mathbb{N} \to \mathbb{N}$ is some *nondecreasing* function, i.e., $f(n) \leqslant f(n+1)$, for all n.

Second, let us examine how quickly can $f(n)$ grow as n grows? Suppose that $f(n)$ grows **exponentially.** Well, researchers have proved that if there are exponentially many processing units in a parallel computer then this necessarily incurs long communication paths between some of them. Since some communicating processing units become exponentially distant from each other, the communication times between them increase correspondingly and, eventually, blemish the theoretically achievable speedup. The reason for all of that is essentially in our real, 3-dimensional space, because

- each processing unit and each communication link occupies some non-zero volume of space, and
- the diameter of the smallest sphere containing exponentially many processing units and communication links is also exponential.

In sum, exponential number of processing units is impractical and leads to theoretically tricky situations.

Suppose now that $f(n)$ grows **polynomially**, i.e., f is a polynomial function of n. Calculus tells us that if $\text{poly}(n)$ and $\exp(n)$ are a polynomial and an exponential function, respectively, then there is an $n' > 0$ so that $\text{poly}(n) < \exp(n)$ for all $n > n'$; that is, $\text{poly}(n)$ is eventually dominated by $\exp(n)$. In other words, we say that a polynomial function $\text{poly}(n)$ **asymptotically grows slower** than an exponential function $\exp(n)$. Note that $\text{poly}(n)$ and $\exp(n)$ are two *arbitrary* functions of n.

So we have $f(n) = \text{poly}(n)$ and consequently the number of processing units is

$$p = \text{poly}(n),$$

where $\text{poly}(n)$ is a polynomial function of n. Here we tacitly discard polynomial functions of "unreasonably" large degrees, e.g. n^{100}. Indeed, we are hoping for much lower degrees, such as 2, 3, 4 or so, which will yield realistic and affordable numbers p of processing units.

In summary, we have obtained an answer to the question above which—because of the generality of C and Π, and due to restrictions imposed by nature and economy—falls short of our expectation. Nevertheless, the answer tells us that p must be some polynomial function (of a moderate degree) of n.

We will apply this to Theorem 2.1 (p. 16) right away in the next section.

2.5.3 The Class NC of Efficiently Parallelizable Problems

Let P be an algorithm for solving a problem Π on CRCW-PRAM(p). According to Theorem 2.1, the execution of P on EREW-PRAM(p) will be at most $O(\log p)$-times slower than on CRCW-PRAM(p). Let us use the observations from previous section and require that $p = \text{poly}(n)$. It follows that $\log p = \log \text{poly}(n) = O(\log n)$. To appreciate why, see Exercises in Sect. 2.7.

Combined with Theorem 2.1 this means that for $p = \text{poly}(n)$ the execution of P on EREW-PRAM(p) will be at most $O(\log n)$-times slower than on CRCW-PRAM(p).

But this also tells us that, when $p = \text{poly}(n)$, choosing a model from the models CRCW-PRAM(p), CREW-PRAM(p), and EREW-PRAM(p) to execute a program affects the execution time of the program by a factor of the order $O(\log n)$, where n is the size of the problem instances to be solved. In other words:

The execution time of a program does not vary too much as we choose the variant of PRAM that will execute it.

This motivates us to introduce a *class of computational problems* containing all the problems that have "fast" parallel algorithms requiring "reasonable" numbers of processing units. But what do "fast" and "reasonable" really mean? We have seen in previous section that the number of processing units is reasonable if it is *polynomial* in n. As for the meaning of "fast", a parallel algorithm is considered to be fast if its parallel execution time is *polylogarithmic* in n. That is fine, but what does now "polylogarithmic" mean? Here is the definition.

Definition 2.1 A function is **polylogarithmic** in n if it is polynomial in $\log n$, i.e., if it is $a_k(\log n)^k + a_{k-1}(\log n)^{k-1} + \cdots + a_1(\log n)^1 + a_0$, for some $k \geqslant 1$.

We usually write $\log^i n$ instead of $(\log n)^i$ to avoid clustering of parentheses. The sum $a_k \log^k n + a_{k-1}\log^{k-1} n + \cdots + a_0$ is asymptotically bounded above by $O(\log^k n)$. To see why, consider Exercises in Sect. 2.7.

We are ready to formally introduce the class of problems we are interested in.

Definition 2.2 Let **NC** be the class of computational problems solvable in polylogarithmic time on PRAM with polynomial number of processing units.

If a problem Π is in the class NC, then it is solvable in polylogarithmic parallel time with polynomially many processing units *regardless of the variant* of PRAM used to solve Π. In other words, the class NC is **robust**, insensitive to the variations of PRAM. How can we see that? If we replace one variant of PRAM with another, then by Theorem 2.1 Π's parallel execution time $O(\log^k n)$ can only increase by a factor $O(\log n)$ to $O(\log^{k+1} n)$ which is still polylogarithmic.

In sum, NC is the class of **efficiently parallelizable** computational problems.

Example 2.1 Suppose that we are given the problem $\Pi \equiv$ "add n given numbers." Then $\pi \equiv$ "add numbers 10, 20, 30, 40, 50, 60, 70, 80" is an instance of size$(\pi) = 8$

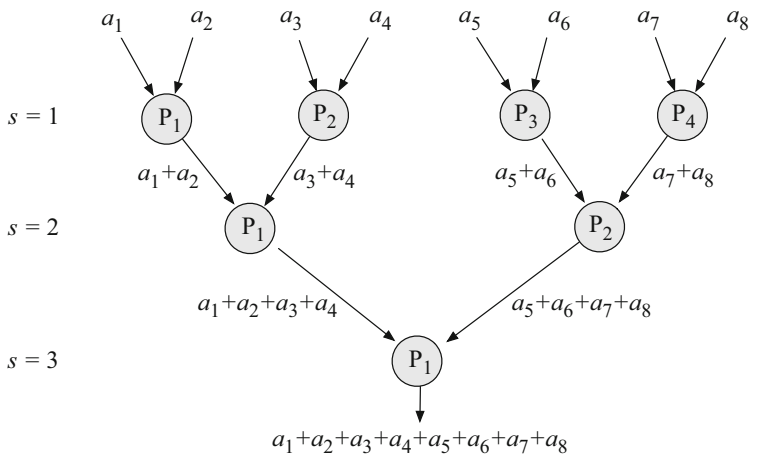

Fig. 2.16 Adding eight numbers in parallel with four processing units

of the problem Π. Let us now focus on *all* instances of size 8, that is, instances of the form $\pi \equiv$ "add numbers $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$."

The fastest sequential algorithm for computing the sum $a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8$ requires $T_{\text{seq}}(8) = 7$ steps, with each step adding the next number to the sum of the previous ones.

In parallel, however, the numbers $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$ can be summed in just $T_{\text{par}}(8) = 3$ parallel steps using $\frac{8}{2} = 4$ processing units which communicate in a tree-like pattern as depicted in Fig. 2.16. In the first step, $s = 1$, each processing unit adds two adjacent input numbers. In each next step, $s \geqslant 2$, two adjacent previous partial results are added to produce a new, combined partial result. This combining of partial results in a tree-like manner continues until $2^{s+1} > 8$. In the first step, $s = 1$, all of the four processing units are engaged in computation; in step $s = 2$, two processing units (P_3 and P_4) start idling; and in step $s = 3$, three processing units (P_2, P_3 and P_4) are idle.

In general, instances $\pi(n)$ of Π can be solved in parallel time $T_{\text{par}} = \lceil \log n \rceil = O(\log n)$ with $\lceil \frac{n}{2} \rceil = O(n)$ processing units communicating in similar tree-like patterns. Hence, $\Pi \in$ NC and the associated speedup is $S(n) = \frac{T_{\text{seq}}(n)}{T_{\text{par}}(n)} = O(\frac{n}{\log n})$. □

Notice that, in the above example, the efficiency of the tree-like parallel addition of n numbers is quite low, $E(n) = O(\frac{1}{\log n})$. The reason for this is obvious: only half of the processing units engaged in a parallel step s will be engaged in the next parallel step $s + 1$, while all the other processing units will be idling until the end of computation. This issue will be addressed in the next section by Brent's Theorem.

2.6 Laws and Theorems of Parallel Computation

In this section we describe the Brent's theorem, which is useful in estimating the lower bound on the number of processing units that are needed to keep a given parallel time complexity. Then we focus on the Amdahl's law, which is used for predicting the theoretical speedup of a parallel program whose different parts allow different speedups.

2.6.1 Brent's Theorem

Brent's theorem enables us to quantify the performance of a parallel program when the number of processing units is reduced.

Let M be a PRAM of an arbitrary type and containing unspecified number of processing units. More specifically, we assume that the number of processing units is always sufficient to cover all the needs of any parallel program.

When a parallel program P is run on M, different numbers of operations of P are performed, at each step, by different processing units of M. Suppose that a total of

$$W$$

operations are performed during the parallel execution of P on M (W is also called the **work** of P), and denote the parallel runtime of P on M by

$$T_{\text{par, M}}(P).$$

Let us now reduce the number of processing units of M to some fixed number

$$p$$

and denote the obtained machine with the reduced number of processing units by

$$R.$$

R is a PRAM of the same type as M which can use, in every step of its operation, at most p processing units.

Let us now run P on R. If p processing units cannot support, in every step of the execution, all the potential parallelism of P, then the parallel runtime of P on R,

$$T_{\text{par, R}}(P),$$

may be larger than $T_{\text{par, M}}(P)$. Now the question raises: Can we quantify $T_{\text{par, R}}(P)$?

The answer is given by **Brent's Theorem** which states that

$$T_{\text{par, R}}(P) = O\left(\frac{W}{p} + T_{\text{par, M}}(P)\right).$$

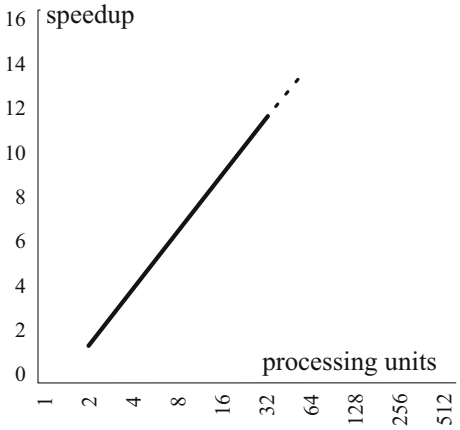

Fig. 2.17 Expected (linear) speedup as a function of the number of processing units

Proof Let W_i be the number of P's operations performed by M in ith step and $T := T_{\text{par, M}}(P)$. Then $\sum_{i=1}^{T} W_i = W$. To perform the W_i operations of the ith step of M, R needs $\lceil \frac{W_i}{p} \rceil$ steps. So the number of steps which R makes during its execution of P is $T_{\text{par, R}}(P) = \sum_{i=1}^{T} \lceil \frac{W_i}{p} \rceil \leqslant \sum_{i=1}^{T} \left(\frac{W_i}{p} + 1 \right) \leqslant \frac{1}{p} \sum_{i=1}^{T} W_i + T = \frac{W}{p} + T_{\text{par, M}}(P)$. □

Applications of Brent's Theorem
Brent's Theorem is useful when we want to reduce the number of processing units as much as possible while keeping the parallel time complexity. For example, we have seen in Example 2.1 on p. 34 that we can sum up n numbers in parallel time $O(\log n)$ with $O(n)$ processing units. Can we do the same with asymptotically less processing units? Yes, we can. Brent's Theorem tells us that $O(n/\log n)$ processing units suffice to sum up n numbers in $O(\log n)$ parallel time. See Exercises in Sect. 2.7.

2.6.2 Amdahl's Law

Intuitively, we would expect that *doubling* the number of processing units should *halve* the parallel execution time; and doubling the number of processing units again should halve the parallel execution time once more. In other words, we would expect that the speedup from parallelization is a linear function of the number of processing units (see Fig. 2.17).

However, linear speedup from parallelization is just a desirable optimum which is not very likely to become a reality. Indeed, in reality very few parallel algorithms achieve it. Most of parallel programs have a speedup which is *near-linear* for *small* numbers of processing elements, and then flattens out into a *constant* value for *large* numbers of processing elements (see Fig. 2.18).

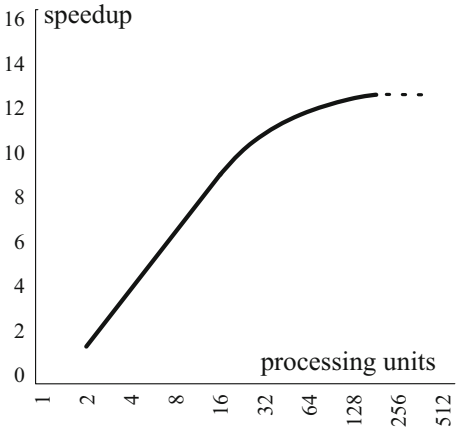

Fig. 2.18 Actual speedup as a function of the number of processing units

Setting the Stage

How can we explain this unexpected behavior? The clues for the answer will be obtained from two simple examples.

- *Example 1.* Let P be a sequential program processing files from disk as follows:
  - P is a sequence of two parts, $P = P_1 P_2$;
  - P_1 scans the directory of the disk, creates a list of file names, and hands the list over to P_2;
  - P_2 passes each file from the list to the processing unit for further processing.

  Note: P_1 *cannot* be sped up by adding new processing units, because scanning the disk directory is intrinsically sequential process. In contrast, P_2 *can* be sped up by adding new processing units; for example, each file can be passed to a separate processing unit. In sum, *a sequential program can be viewed as a sequence of two parts that differ in their parallelizability*, i.e., amenability to parallelization.
- *Example 2.* Let P be as above. Suppose that the (sequential) execution of P takes 20 min, where the following holds (see Fig. 2.19):
  - the non-parallelizable P_1 runs *2* min;
  - the parallelizable P_2 runs *18* min.

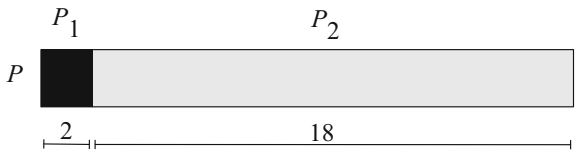

Fig. 2.19 P consists of a non-parallelizable P_1 and a parallelizable P_2. On one processing unit, P_1 runs 2 min and P_2 runs 18 min

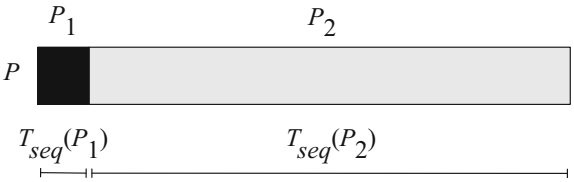

Fig. 2.20 P consists of a non-parallelizable P_1 and a parallelizable P_2. On a single processing unit P_1 requires $T_{\text{seq}}(P_1)$ time and P_2 requires $T_{\text{seq}}(P_2)$ time to complete

Note: since only P_2 can benefit from additional processing units, the parallel execution time $T_{\text{seq}}(P)$ of the whole P cannot be less than the time $T_{\text{seq}}(P_1)$ taken by the non-parallelizable part P_1 (that is, 2 min), regardless of the number of additional processing units engaged in the parallel execution of P. In sum, if parts of a sequential program differ in their potential parallelisms, they differ in their potential speedups from the increased number of processing units, so the speedup of the whole program will depend on their sequential runtimes.
The clues that the above examples brought to light are recapitulated as follows: In general, a program P executed by a parallel computer can be split into two parts,

- part P_1 which *does not* benefit from multiple processing units, and
- part P_2 which *does benefit* from multiple processing units;
- besides P_2's benefit, also the sequential execution times of P_1 and P_2 influence the parallel execution time of the whole P (and, consequently, P's speedup).

Derivation
We will now assess quantitatively how the speedup of P depends on P_1's and P_2's sequential execution times and their amenability to parallelization and exploitation of multiple processing units.

Let $T_{\text{seq}}(P)$ be the sequential execution time of P. Because $P = P_1 P_2$, a sequence of parts P_1 and P_2, we have

$$T_{\text{seq}}(P) = T_{\text{seq}}(P_1) + T_{\text{seq}}(P_2),$$

where $T_{\text{seq}}(P_1)$ and $T_{\text{seq}}(P_2)$ are the sequential execution times of P_1 and P_2, respectively (see Fig. 2.20).

When we actually employ additional processing units in the parallel execution of P, it is the execution of P_2 that is sped up by some factor $s > 1$, while the execution of P_1 does not benefit from additional processing units. In other words, the execution time of P_2 is reduced from $T_{\text{seq}}(P_2)$ to $\frac{1}{s} T_{\text{seq}}(P_2)$, while the execution time of P_1 remains the same, $T_{\text{seq}}(P_1)$. So, after the employment of additional processing units the parallel execution time $T_{\text{par}}(P)$ of the whole program P is

$$T_{\text{par}}(P) = T_{\text{seq}}(P_1) + \frac{1}{s} T_{\text{seq}}(P_2).$$

The speedup $S(P)$ of the whole program P can now be computed from definition,

$$S(P) = \frac{T_{\text{seq}}(P)}{T_{\text{par}}(P)}.$$

We could stop here; however, it is usual to express $S(P)$ in terms of b, the *fraction* of $T_{\text{seq}}(P)$ during which parallelization of P is beneficial. In our case

$$b = \frac{T_{\text{seq}}(P_2)}{T_{\text{seq}}(P)}.$$

Plugging this in the expression for $S(P)$, we finally obtain the **Amdahl's Law**

$$S(P) = \frac{1}{1 - b + \frac{b}{s}}.$$

Some Comments on Amdahl's Law
Strictly speaking, the speedup in the Amdahl's Law is a function of three variables, P, b and s, so it would be more appropriately denoted by $S(P, b, s)$. Here b is the fraction of the time during which the sequential execution of P *can* benefit from multiple processing units. If multiple processing units are actually available and exploited by P, the part of P that exploits them is sped up by the factor $s > 1$. Since s is only the speedup of a *part* of the program P, the speedup of the *whole* P cannot be larger than s; specifically, it is given by $S(P)$ of the Amdahl's Law.

From the Amdahl's Law we see that

$$S < \frac{1}{1 - b},$$

which tells us that a small part of the program which cannot be parallelized will limit the overall speedup available from parallelization. For example, the overall speedup S that the program P in Fig. 2.19 can possibly achieve by parallelizing the part P_2 is bounded above by $S < \frac{1}{1-\frac{18}{20}} = 10$.

Note that in the derivation of the Amdahl's Law nothing is said about the size of the problem instance solved by the program P. It is implicitly assumed that the problem instance remains the same, and that the only thing we carry out is parallelization of P and then application of the parallelized P on the same problem instance. Thus, Amdahl's law only applies to cases where the size of the problem instance is fixed.

Amdahl's Law at Work
Suppose that 70% of a program execution can be sped up if the program is parallelized and run on 16 processing units instead of one. What is the maximum speedup that can be achieved by the whole program? What is the maximum speedup if we increase the number of processing units to 32, then to 64, and then to 128?

In this case we have $b = 0.7$, the fraction of the sequential execution that can be parallelized; and $1 - b = 0.3$, the fraction of calculation that cannot be parallelized.

The speedup of the parallelizable fraction is s. Of course, $s \leqslant p$, where p is the number of processing units. By Amdahl's Law the speedup of the whole program is

$$S = \frac{1}{1 - b + \frac{b}{s}} = \frac{1}{0.3 + \frac{0.7}{s}} \leqslant \frac{1}{0.3 + \frac{0.7}{16}} = 2.91.$$

If we double the number of processing units to 32 we find that the maximum achievable speedup is 3.11:

$$S = \frac{1}{1 - b + \frac{b}{s}} = \frac{1}{0.3 + \frac{0.7}{s}} \leqslant \frac{1}{0.3 + \frac{0.7}{32}} = 3.11\,,$$

and if we double it once again to 64 processing units, the maximum achievable speedup becomes 3.22:

$$S = \frac{1}{1 - b + \frac{b}{s}} = \frac{1}{0.3 + \frac{0.7}{s}} \leqslant \frac{1}{0.3 + \frac{0.7}{64}} = 3.22.$$

Finally, if we double the number of processing units even to 128, the maximum speedup we can achieve is

$$S = \frac{1}{1 - b + \frac{b}{s}} = \frac{1}{0.3 + \frac{0.7}{s}} \leqslant \frac{1}{0.3 + \frac{0.7}{128}} = 3.27.$$

In this case doubling the processing power only slightly improves the speedup. Therefore, using more processing units is not necessarily the optimal approach.

Note that this complies with actual speedups of realistic programs as we have depicted in Fig. 2.18.

⋆ A Generalization of Amdahl's Law

Until now we assumed that there are just two parts of of a given program, of which one cannot benefit from multiple processing units and the other can. We now assume that the program is a sequence of three parts, each of which could benefit from multiple processing units. Our goal is to derive the speedup of the whole program when the program is executed by multiple processing units.

So let $P = P_1 P_2 P_3$ be a program which is a sequence of three parts P_1, P_2, and P_3. See Fig. 2.21. Let $T_{\text{seq}}(P_1)$ be the time during which the sequential execution of

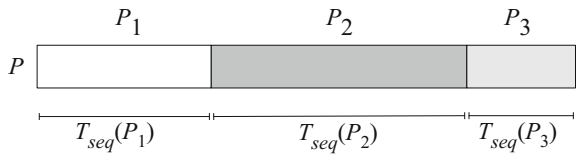

Fig. 2.21 P consists of three differently parallelizable parts

P spends executing part P_1. Similarly we define $T_{\text{seq}}(P_2)$ and $T_{\text{seq}}(P_3)$. Then the sequential execution time of P is

$$T_{\text{seq}}(P) = T_{\text{seq}}(P_1) + T_{\text{seq}}(P_2) + T_{\text{seq}}(P_3).$$

But we want to run P on a parallel computer. Suppose that the analysis of P shows that P_1 could be parallelized and sped up on the parallel machine by factor $s_1 > 1$. Similarly, P_2 and P_3 could be sped up by factors $s_2 > 1$ and $s_3 > 1$, respectively. So we parallelize P by parallelizing each of the three parts P_1, P_2, and P_3, and run P on the parallel machine. The parallel execution of P takes $T_{\text{par}}(P)$ time, where $T_{\text{par}}(P) = T_{\text{par}}(P_1) + T_{\text{par}}(P_2) + T_{\text{par}}(P_3)$. But $T_{\text{par}}(P_1) = \frac{1}{s_1} T_{\text{seq}}(P_1)$, and similarly for $T_{\text{par}}(P_2)$ and $T_{\text{par}}(P_3)$. It follows that

$$T_{\text{par}}(P) = \frac{1}{s_1} T_{\text{seq}}(P_1) + \frac{1}{s_2} T_{\text{seq}}(P_2) + \frac{1}{s_3} T_{\text{seq}}(P_3).$$

Now the speedup of P can easily be computed from its definition, $S(P) = \frac{T_{\text{seq}}(P)}{T_{\text{par}}(P)}$.

We can obtain a more informative expression for $S(P)$. Let b_1 be the *fraction* of $T_{\text{seq}}(P)$ during which the sequential execution of P executes P_1; that is, $b_1 = \frac{T_{\text{seq}}(P_1)}{T_{\text{seq}}(P)}$. Similarly we define b_2 and b_3. Applying this in the definition of $S(P)$ we obtain

$$S(P) = \frac{T_{\text{seq}}(P)}{T_{\text{par}}(P)} = \frac{1}{\frac{b_1}{s_1} + \frac{b_2}{s_2} + \frac{b_3}{s_3}}.$$

Generalization to programs which are sequences of arbitrary number of parts P_i is straightforward. In reality, programs typically consist of several parallelizable parts and several non-parallelizable (serial) parts. We easily handle this by setting $s_i \geqslant 1$.

2.7 Exercises

1. How many pairwise interactions must be computed when solving the n-body problem if we assume that interactions are symmetric?
2. Give an intuitive explanation why $T_{\text{par}} \leqslant T_{\text{seq}} \leqslant p \cdot T_{\text{par}}$, where T_{par} and T_{seq} are the parallel and sequential execution times of a program, respectively, and p is the number of processing units used during the parallel execution.
3. Can you estimate the number of different network topologies capable of interconnecting p processing units P_i and m memory modules M_j? Assume that each topology should provide, for every pair (P_i,M_j), a path between P_i and M_j.
4. Let P be an algorithm for solving a problem Π on CRCW-PRAM(p). According to Theorem 2.1, the execution of P on EREW-PRAM(p) will be at most $O(\log p)$-times slower than on CRCW-PRAM(p). Now suppose that $p = \text{poly}(n)$, where n is the size of a problem instance. Prove that $\log p = O(\log n)$.

5. Prove that the sum $a_k \log^k n + a_{k-1} \log^{k-1} n + \cdots + a_0$ is asymptotically bounded above by $O(\log^k n)$.
6. Prove that $O(n/\log n)$ processing units suffice to sum up n numbers in $O(\log n)$ parallel time. *Hint:* Assume that the numbers are summed up with a tree-like parallel algorithm described in Example 2.1. Use Brent's Theorem with $W = n - 1$ and $T = \log n$ and observe that by reducing the number of processing units to $p := n \log n$, the tree-like parallel algorithm will retain its $O(\log n)$ parallel time complexity.
7. True or false:
   (a) The definition of the parallel execution time is: "execution time = computation time + communication time + idle time."
   (b) A simple model of the communication time is:"communication time = set-up time + data transfer time."
   (c) Suppose that the execution time of a program on a single processor is T_1, and the execution time of the same parallelized program on p processors is T_p. Then, the speedup and efficiency are $S = T_1/T_p$ and $E = S/p$, respectively.
   (d) If speedup $S < p$ then $E > 1$.
8. True or false:
   (a) If processing units are identical, then in order to minimize parallel execution time, the work (or, computational load) of a parallel program should be partitioned into equal parts and distributed among the processing units.
   (b) If processing units differ in their computational power, then in order to minimize parallel execution time, the work (or, computational load) of a parallel program should be distributed evenly among the processing units.
   (c) Searching for such distributions is called **load balancing**.
9. Why must be the load of a parallel program evenly distributed among processors?
10. Determine the bisection bandwidths of 1D-mesh (chain of computers with bidirectional connections), 2D-mesh, 3D-mesh, and the hypercube.
11. Let a program P be composed of a part R that can be ideally parallelized, and of a sequential part S; that is, $P = RS$. On a single processor, S takes 10% of the total execution time and during the remaining 90% of time R could run in parallel.
    (a) What is the maximal speedup reachable with unlimited number of processors?
    (b) How is this law called?
12. **Moore's law** states that computer performance doubles every 1.5 year. Suppose that the current computer performance is Perf $= 10^{13}$. When will be, according to this law, 10 times greater (that is, $10 \times$ Perf)?
13. A problem Π comprises two subproblems, Π_1 and Π_2, which are solved by programs P_1 and P_2, respectively. The program P_1 would run 1000 s on the computer C_1 and 2000 s on the computer C_2, while P_2 would require 2000 and 3000 s on C_1 and C_2, respectively. The computers are connected by a 1000-km long optical fiber link capable of transferring data at 100 MB/sec with 10 msec latency. The programs can execute concurrently but must transfer either (a) 10 MB of data 20,000 times or (b) 1 MB of data twice during the execution. What is the best configuration and approximate runtimes in cases (a) and (b)?

2.8 Bibliographical Notes

In presenting the topics in this Chapter we have strongly leaned on Trobec et al. [26] and Atallah and Blanton [3]. On the computational models of sequental computation see Robič [22]. Interconnection networks are discussed in great detail in Dally and Towles [6], Duato et al. [7], Trobec [25] and Trobec et al. [26]. The dependence of execution times of real world parallel applications on the performance of the interconnection networks is discussed in Grama et al. [12].