The innovation of parallel computing has added a new dimension to the design of algorithms. Thus, parallel processing consists of parallel architectures and parallel algorithms, and recent interests in parallel computers has motivated the development of parallel algorithms to solve many types of problems. Many aspects of parallel algorithm design are presented in [Smith 93, Cosnard 95, Miller 96]. Algorithms in which several operations may be executed simultaneously are referred to as parallel algorithms. In general, a parallel algorithm can be defined as a set of processes or tasks that may be executed simultaneously and may communicate with each other in order to solve a given problem. The term task or process may be defined as a part of a program that can be executed on a processor.

In both parallel architectures and parallel algorithms, there are many design choices for which there are no direct counterparts in conventional sequential processing. In parallel architectures, examples of such choices include the number of processors, local versus global memory organization, synchronous versus asynchronous execution, and interconnection network topologies. In parallel algorithms, issues that do not arise in sequential algorithms include determination of the number of processors needed for a computation, data allocation across memories, whether synchronization is beneficial or necessary, and interprocess communication requirements.

In general, parallelism has introduced new degrees of freedom to both the architecture and algorithm design approaches. For effective use of parallel systems, it is essential to obtain a good match between algorithm requirements and architecture capabilities. A parallel algorithm can be viewed as a collection of independent task modules, some of which can be executed concurrently utilizing the parallel computer. Information that captures the relationship between sequential algorithms and parallel algorithms can be of use in a number of different ways. In general, sequential algorithms may have several parallel versions, depending on

- how data can be accessed.
- how data can be partitioned into tasks.
- how those tasks are allocated to processes.
- how the processes are synchronized.
In practice, there are several principles in the design of parallel algorithms:

**The Brent Scheduling Principle:** This principle makes it possible to reduce the number of processors used in existing parallel algorithms, without increasing the total execution time. In general, the execution time of the algorithms increases somewhat when the number of processors is reduced, but not by an amount that increases the total execution time. In other words, if an algorithm has an execution time of $O(\log n)$, then the total execution time might increase by a constant factor.

**The Pipelining Principle:** Pipelining can be used in situations in which we want to perform several operations in a sequence $\{P_1, ..., P_n\}$, where those operations have the property that some steps of $P_{i+1}$ can be carried out before operation $P_i$ is finished. In a parallel algorithm, it is often possible to overlap these steps and decrease total execution time. Although this technique is most often used in MIMD algorithms, many SIMD algorithms are also able to take the advantage of this principle. Several algorithms in this book illustrate this principle.

**The Divide and Conquer Principle:** This is the principle of splitting a problem into several small independent components and solving them in parallel. There are many examples of this technique in this book: the parallel prefix, the cycles in a graph, the shortest paths, and matrix multiplication. One of the most basic principles in MIMD algorithm design is to analyze the computations to be performed and determine the parallelism, meaning the dependency graph of the computation.

**The Dependency Graph Principle:** We create a directed graph in which the nodes represent blocks of independent operations and the edges represent situations in which one block of operation depends on the outcome of performing other blocks. To design an MIMD algorithm sometimes involves the conversion of a good SIMD algorithm. This conversion requires the computations to be synchronized, which is the principle that arises in the design of MIMD algorithms. This principle is identical to those involved in concurrent programming and is called the race condition principle.

It is worth noting that the main difference between MIMD algorithm design and concurrent algorithm design involves questions of when and why one creates multiple processes. When designing concurrent algorithms to run on a single-processor computer, we create processes in order to handle asynchronous events, because we expect little real concurrency to occur. An example is input and output operations, because there is no real concurrency on a single-processor computer. In contrast, when designing MIMD algorithms to run on a parallel computer, we try to maximize the amount of concurrency, and we look for operations that can be carried out simultaneously and try to create multiple processes to handle this situation.

**The Race Condition Principle:** If two processes try to access the same shared data, they may interfere with each other.

Example: Suppose two processes update a shared linked list simultaneously such that the head of the list is pointed to by a pointer variable named `head` and