8.2 Types of Parallelization

The degree of problem (program) concurrency is defined as the number of subproblems extracted in the problem that can potentially be solved in parallel, or as the number of tasks executed simultaneously in the parallel or concurrent program. The greater the degree of problem concurrency, the better it is suited for parallel execution.

Designing an efficient parallel program is also easier when relationships between tasks, that is the frequency of communication and the synchronization of their operation, are small. This is the case when the tasks are independent of each other and in the course of execution they do not need to communicate. It is enough to convey to tasks the data to be processed and then collect and combine the computed results.

8.2.1 Functional Parallelization

Subdivision of the problem into tasks (or subproblems) is referred to as *functional parallelization*. Subdivision into separate functions is used in the following example in which each task is designed to implement a separate function whose goal is to strike off composite numbers that are multiples of a given prime.

The tasks are dependent on each other, as in some of them the data computed is used by other tasks. These dependencies between tasks, requiring certain tasks to be executed before other tasks, can be represented by an acyclic-directed graph called a *task dependence graph*.

The algorithm or method of Sieve of Eratosthenes can be used to find the primes in the interval [2…n] for some natural n.

The first number 2 is recorded as the first prime number and then the multiples of 2 are struck off the list (sieved) starting from the square of this number, that is from $2^2 = 4$, since these multiples are not prime.

After removal of 4, a nonstruck-off number greater than the last prime (that is 2) is 3. It is recorded as the next prime and the multiples of 3 starting from $3^2 = 9$ are struck off. The tasks of striking off the multiples of 2 and 3 from the interval [2…n] can be carried out simultaneously.

After removal of 9, a nonstruck-off number greater than the last prime (that is 3) is 5. It is recorded as the next prime and the multiples of 5 starting from 25 are struck off. The tasks of striking off the multiples of 2, 3, and 5 from the interval [2…n] can be carried out simultaneously.

The Sieve of Eratosthenes ends when all the tasks are finished and the penultimate task finds a prime satisfying the condition $r_k^2 > n$ or else $r_k > \sqrt{n}$. The result of solving the problem is all numbers left in the interval [2…n].

Figure 8.2a shows a task-dependency graph (TDG) for the problem of finding primes. In task A_i for $i = 1, 2, \ldots, k-1$ a consecutive prime r_{i+1}($r_2 = 3$, $r_3 = 5$, and so on) is evaluated by striking off the multiple of r_i. The prime found is sent to task A_{i+1}. A task A_k checks whether $r_k^2 > n$, while task S collects information about the completion of all tasks and decides whether to finish computation.

Figure 8.2a shows that the maximum degree of problem concurrency, equal to the number of A_i tasks, increases with n. This degree is governed by an index k derived from

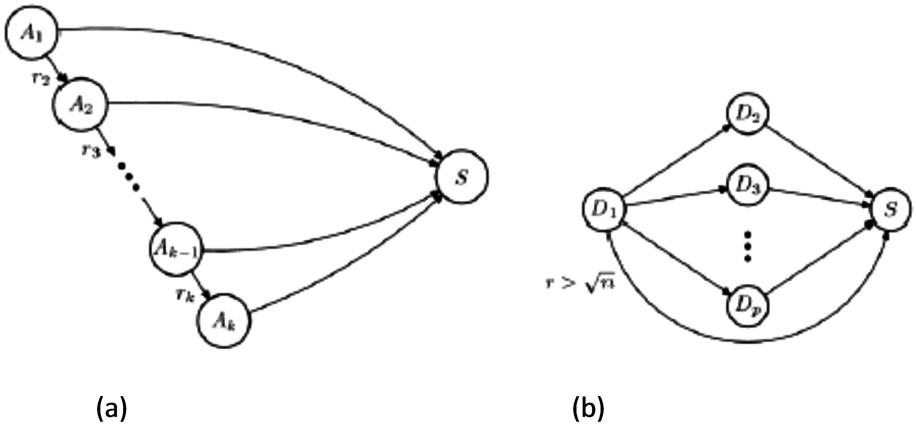

FIGURE 8.2
(a) Task-dependency graph for functional parallelization and (b) task-dependency graph for data parallelization.

inequality $r_k^2 > n$. For a fixed n, the number of tasks that can be performed in parallel is limited. Consequently, the speedup that can be achieved by applying the discussed algorithm is also limited.

In the functional subdivision used in the problem of finding primes using the Sieve of Eratosthenes, each of the extracted tasks (functions) performs sieving of composite numbers that are multiples of a given prime.

8.2.2 Data Parallelization

Data subdivision is the most common method to extract tasks that can be executed in parallel. It is frequently used when large amounts of data are processed. The benefits of conducting parallel computing are then significant and sometimes only this type of computing guarantees processing data in reasonable time.

Subdivision is concerned with

- Input data
- Intermediate data
- Output data

When designing a parallel algorithm, all possible partitions of data should be analyzed in order to choose the one that leads to efficient computing.

- Input data

The subdivision of input data is based on partitioning this data into a number of equal parts. Each of these parts is processed by a separate task; typically, the tasks perform the same computation on the assigned parts of the data and evaluate on the basis of its own part a certain amount of output data. The subdivision of input data is used in the following example in which the tasks find the primes in the assigned parts of input data of sizes $[n/p]$.

Given n input items, they can be broken down into p parts for $p < n$ of size at most $[n/p]$ and assigned to particular tasks. In each of the p tasks, all the necessary functions on portions of data assigned to a task are carried out.

In the problem of finding prime numbers employing the Sieve of Eratosthenes, an interval [2…p] can be divided into p subintervals: [2…[n/p]], [[n/p]+2…[n/p]],…, [($p-1$)[n/p]…n].

In each task associated with a given subinterval the multiples of consecutive primes, that is, 2, 3, 5, and so on, are struck off.

If we assume that p is sufficiently smaller than $\sqrt{n}$, then the primes less than $\sqrt{n}$ and the first prime greater than $\sqrt{n}$ will be found by the task assigned to subinterval [2…[n/p]].

The task D_1 sends consecutively determined primes to other tasks (D_2, D_3,…, D_p) and it also sends a signal to task S about the end of computation when a prime greater than $\sqrt{n}$ is detected (Figure 8.2b). The problem solution is nonstruck-off numbers in the subintervals processed by all tasks.

The degree of concurrency of the parallel algorithm constructed using data parallelization is p, for p satisfying $1<p<n$.

- Output data

The subdivision of output data assumes that the respective parts of this data are computed by the tasks in parallel. This is possible if each element of the output data can be determined independently as a function of the input data.

The problem of multiplication of matrices a and b of sizes $n\times n$, that is, $c=ab$ can be obtained by performing multiplication of submatrices of size approximately $[n/2]\times[n/2]$, as follows:

$$\left[\begin{array}{c|c} a_{1,1} & a_{1,2} \\ \hline a_{2,1} & a_{2,2} \end{array}\right] \times \left[\begin{array}{c|c} b_{1,1} & b_{1,2} \\ \hline b_{2,1} & b_{2,2} \end{array}\right] = \left[\begin{array}{c|c} c_{1,1} & c_{1,2} \\ \hline c_{2,1} & c_{2,2} \end{array}\right]$$

The output data, that is matrix c, is composed of four submatrices, each of which can be computed by a separate task. The subdivision of the problem into four tasks is based on the subdivision of the output data as indicated below:

A disadvantage of both subdivisions is a relatively low degree of concurrency equal to 4. It is easy to point out that decomposition can also be done on the basis of *intermediate data*, that is, taking into account the products of the submatrices appearing earlier. These products can be evaluated by separate tasks in the first stage, and in the second stage the selected pairs of intermediate results can be combined into the final one. Such a subdivision based on the intermediate data has a higher degree of concurrency equal to 8.

However, in this case the need for synchronization of computation arises due to the dependencies between groups of tasks for the two stages.

8.2.3 Recursive Parallelization

Recursive subdivision is used in cases where the problem is solved by the *divide-and-conquer* method. In every step of this method, the problem is divided into a number of independent subproblems, or tasks. In most circumstances, the tasks are smaller instances of the original problem. The tasks are executed in a similar manner and their division into subsequent subtasks ends when they are so simple that they can be resolved in a trivial way. When all the tasks are executed, their results are combined into the solution to the original problem.

An example of the application of the divide-and-conquer method is a recursive algorithm to find both the minimum and maximum elements in a given array a. In line 10,