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} (≈thousand) times, then 2^{20} (≈million) times, and finally 2^{40} (≈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$(\pi) = 8$

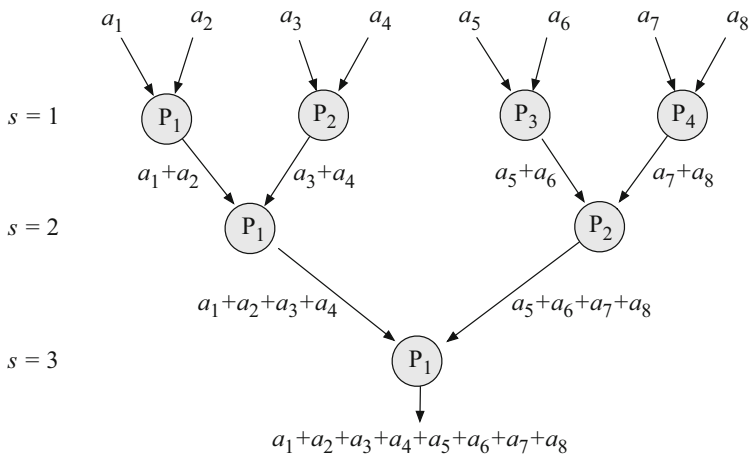

Fig. 2.16 Adding eight numbers in parallel with four processing units

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

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

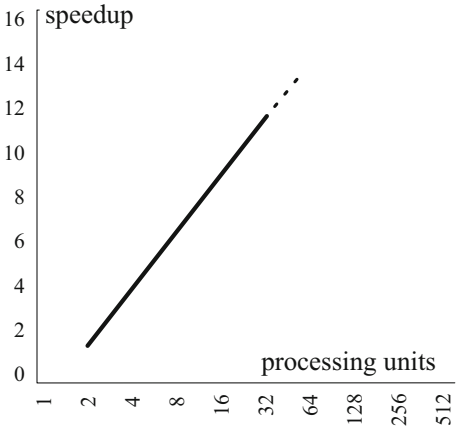

Fig. 2.17 Expected (linear) speedup as a function of the number of processing units

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).

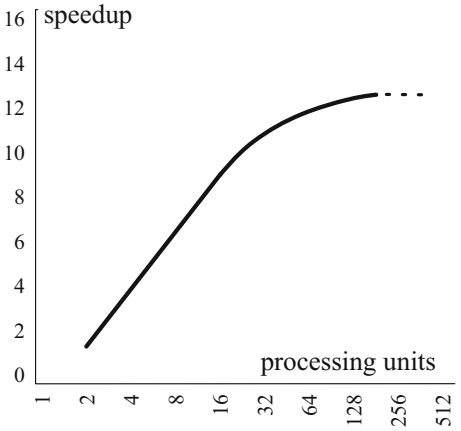

Fig. 2.18 Actual speedup as a function of the number of processing units

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.

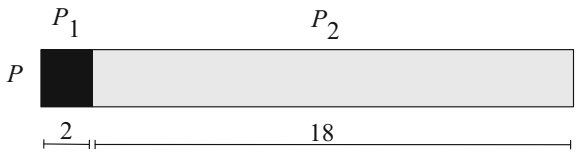

Fig. 2.19 P consists of a non-parallelizable P_1 and a parallelizable P_2. On one processing unit, P_1 runs 2 min and P_2 runs 18 min

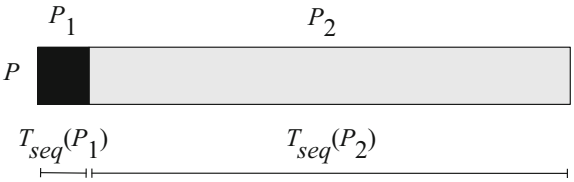

Fig. 2.20 P consists of a non-parallelizable P_1 and a parallelizable P_2. On a single processing unit P_1 requires $T_{\text{seq}}(P_1)$ time and P_2 requires $T_{\text{seq}}(P_2)$ time to complete

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

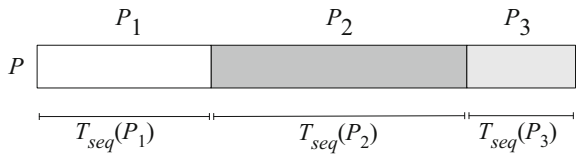

Fig. 2.21 P consists of three differently parallelizable parts

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].