Chapter 10
A Glimpse of Parallel Computing

It should not be too difficult to imagine that applications of scientific computing in the real world can require huge amounts of computation. This can be due to a combination of advanced mathematical models, sophisticated numerical algorithms, and high accuracy requirements. Such large-scale applications easily involve millions (or more) of data entities, and thousands (or more) of time steps in the case of time-dependent problems. All these will translate into a huge number of floating-point arithmetic operations and enormous data structures on a computer. However, a serial computer that has only one CPU will have trouble getting all these computations done quickly enough and/or fitting needed data into its memory. The remedy to this capacity problem is to use parallel computing, for which the present chapter aims to provide a gentle introduction.

10.1 Motivations for Parallel Computing

Let us start with motivating the use of multiple-processor computers by looking at the limitations of serial computers from the perspectives of computing speed and memory size.

10.1.1 From the Perspective of Speed

The development of computing hardware was well predicted by the famous *Moore's law* [22], which says that the number of transistors that can be put in an integrated circuit grows exponentially at a roughly fixed rate. Consequently, the speed of a top serial computer, in terms of floating-point operations per second (FLOPS), kept growing exponentially. This trend held from the 1950s until the beginning of the twenty-first century, but it then showed signs of flattening out. A future serial computer that can solve a problem that is too large for today's serial computer is unlikely. Combing the forces of many serial computers in some way, i.e., parallel computing, thus becomes a reasonable approach.

The most prominent reason for adopting parallel computing is the need to finish a large computational task more quickly. To illustrate this point, let us consider a simplified three-dimensional diffusion equation:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}, \tag{10.1}$$

for which a more general form was introduced in Sect. 7.2.

We want to develop an explicit numerical scheme that is based on finite differences for the above 3D equation. This will be done in the same fashion as for the 1D version described in Sect. 7.4. Suppose superscript ℓ denotes a discrete time level, subscripts i, j, k denote a spatial grid point, Δt denotes the time step size, and Δx, Δy, and Δz denote the spatial grid spacing. Then, the temporal derivative term in (10.1) is discretized as

$$\frac{\partial u}{\partial t} \approx \frac{u^{\ell+1}_{i,j,k} - u^{\ell}_{i,j,k}}{\Delta t}.$$

For the spatial derivative term in the x-direction, the finite difference approximation is

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u^{\ell}_{i-1,j,k} - 2u^{\ell}_{i,j,k} + u^{\ell}_{i+1,j,k}}{\Delta x^2}.$$

The other two spatial derivative terms $\partial^2 u/\partial y^2$ and $\partial^2 u/\partial z^2$ can be discretized similarly.

Combining all the finite differences together, the explicit numerical scheme for the 3D diffusion equation (10.1) arises as

$$\begin{aligned}\frac{u^{\ell+1}_{i,j,k} - u^{\ell}_{i,j,k}}{\Delta t} = & \frac{u^{\ell}_{i-1,j,k} - 2u^{\ell}_{i,j,k} + u^{\ell}_{i+1,j,k}}{\Delta x^2} \\ & + \frac{u^{\ell}_{i,j-1,k} - 2u^{\ell}_{i,j,k} + u^{\ell}_{i,j+1,k}}{\Delta y^2} \\ & + \frac{u^{\ell}_{i,j,k-1} - 2u^{\ell}_{i,j,k} + u^{\ell}_{i,j,k+1}}{\Delta z^2}.\end{aligned} \tag{10.2}$$

For simplicity we assume that the solution domain is the unit cube with a Dirichlet boundary condition, e.g., $u = 0$. The values of $u^0_{i,j,k}$ are prescribed by some initial condition $u(x, y, z, 0) = I(x, y, z)$. If the spatial grid spacing is the same in all three directions, i.e., $\Delta x = \Delta y = \Delta z = h = 1/n$, the explicit numerical scheme will have the following formula representing the main computational work per time step:

$$u_{i,j,k}^{\ell+1} = \alpha u_{i,j,k}^{\ell} + \beta \left(u_{i-1,j,k}^{\ell} + u_{i,j-1,k}^{\ell} + u_{i,j,k-1}^{\ell} + u_{i+1,j,k}^{\ell} + u_{i,j+1,k}^{\ell} + u_{i,j,k+1}^{\ell} \right) \tag{10.3}$$

for $1 \leq i, j, k \leq n-1$, where $\alpha = 1 - 6\Delta t/h^2$ and $\beta = \Delta t/h^2$.

The above formula requires eight floating-point operations per inner grid point: two multiplications and six additions. That is, the total number of floating-point operations per time step is $8(n-1)^3$. Recall from Sect. 7.4.5 that explicit numerical schemes often have a strict restriction on the maximum time step size, which is

$$\Delta t \leq \frac{1}{6}h^2$$

for this particular 3D case. The minimum number of time steps N needed for solving (10.1) between $t = 0$ and $t = 1$ is consequently $N \geq \frac{6}{h^2} = 6n^2$. Therefore, the total number of floating-point operations for the entire computation is

$$6n^2 \times 8(n-1)^3 = 48n^2(n-1)^3 \approx 48n^5.$$

If we have $n = 1{,}000$, then the entire computation requires 48×10^{15} floating-point operations. How much CPU time does it need to carry out these operations on a serial computer? Let us assume that an extremely fast serial computer has a peak performance of 48 GFLOPS, i.e., 48×10^9 FLOPS; then the total computation will require 10^6 s, i.e., 278 h. This may not sound like an alarmingly long time. However, the sustainable performance of numerical schemes of type (10.3), which are computer-memory intensive, is normally far below the theoretical peak performance. This is due to the increasing gap between the processor speed and memory speed on modern microprocessors, commonly referred to as the "memory wall" problem [21]. Moreover, our simple model equation (10.1) has not considered variable coefficients, difficult boundary conditions, or source terms. Therefore, it is fair to say that a realistic 3D diffusion problem can require a lot more than the above theoretical CPU usage, making a serial computer totally unfit for the explicit scheme to work on a $1{,}000 \times 1{,}000 \times 1{,}000$ mesh.

As another consideration, numerical simulators are frequently used as an experimental tool. Many different runs of the same simulator are typically needed, requiring the computing time of each simulation to be within e.g. an hour, or ideally minutes.

It should be mentioned that there exist more computationally efficient methods for solving (10.1) than the above explicit scheme. For example, a numerical method with no stability constraint can use dramatically fewer time steps, but with much more work per step. Nevertheless, the above simple example suffices to show that serial computers clearly have a limit in computing speed. The bad news is that the speed of a single CPU core is not expected to grow anymore in the future. Also, as will be shown in the following text, the memory limit of a serial computer is equally prohibitive for large-scale computations.

10.1.2 From the Perspective of Memory

The technical terms of megabyte (MB) and gigabyte (GB) should sound familiar to anyone who has used a computer. But how much data can be held in 1 MB or 1 GB of random access memory exactly? According to the standard binary definition, 1 MB is $1024^2 = 1{,}048{,}576$ bytes and 1 GB is $1024^3 = 1{,}073{,}741{,}824$ bytes. Scientific computations typically use double-precision numbers, each occupying eight bytes of memory on a digital computer. In other words, 1 MB can hold 131,072 double-precision numbers, and 1 GB can hold 134,217,728 double-precision numbers.

We mentioned in the above text that time savings are the most prominent reason for adopting parallel computing. But it is certainly not the only reason. Another equally important reason is to solve larger problems. Let us consider the example of an 8 GB memory, which is decently large for a single-CPU computer. By simple calculation, we know that 8 GB can hold 1024^3 double-precision values. For our simple numerical scheme (10.3), however, 8 GB is not enough for the case of $n = 1{,}000$, because two double-precision arrays of length $(n+1)^3$ are needed for storing u^ℓ and $u^{\ell+1}$.

Considering the fact that a mesh resolution of $1{,}000 \times 1{,}000 \times 1{,}000$ is insufficient for many problems, and that a typical simulation can require dozens or hundreds of large 3D arrays in the data structure, single-CPU computers are clearly far from being capable of very large-scale computations, with respect to both computing speed and memory size.

10.1.3 Parallel Computers

In fact, at the time of this writing, serial computers will soon become history. Every new PC now has more than one processor core, where each core is an independent processing unit capable of doing the tasks of a conventional processor. It is likely that all future computers will be parallel in some form. Knowledge about designing parallel algorithms and writing parallel codes thus becomes essential.

A parallel computer can be roughly defined as a computing system that allows multiple processors to work concurrently to solve one computational problem. The most common way to categorize modern parallel computers is by looking at the memory layout. A *shared-memory* system means that the processors have no private memory but can all access a single global memory. Symmetric multiprocessors (SMPs) were the earliest shared-memory machines, where the memory access time is uniform for all the processors. Later, shared-memory computers adopted the architecture of non-uniform memory access to incorporate more processors. The recent multicore chips can be considered a revival of SMP, equipped with some level of shared cache among the processor cores plus tighter coupling to the global memory.

In the category of *distributed-memory* systems, all processors have a private local memory that is inaccessible by others. The processors have some form of interconnection between them, ranging from dedicated networks with high throughput and low latency to the relatively slow Ethernet. The processors communicate with each other by explicitly sending and receiving messages, which are arrays of data values in a programming language. Two typical categories of distributed-memory systems are proprietary massively parallel computers and cost-effective PC clusters. For example, Ethernet-connected serial computers in a computer lab fall into the latter category. We refer to Fig. 10.1 for a schematic overview of the shared-memory and distributed-memory parallel architectures.

There are, of course, parallel systems that fall in between the two main categories. For example, a cluster of SMP machines is a *hybrid* system. A multicore-based PC cluster is, strictly speaking, also a hybrid system where memory is distributed among the PC nodes, while one or several multicore chips share the memory within each node. Furthermore, the cores inside a node can be inhomogeneous, e.g., general-purpose graphics processing units can be combined with regular CPU cores to accelerate parallel computations. For a review of the world's most powerful parallel computers, we refer the reader to the Top500 List [2].

Note also that parallel computing does not necessarily involve multiple processors. Actually, modern microprocessors have long exploited hardware parallelism within one processor, as in instruction pipelining, multiple execution units, and so on. Development of this compiler-automated parallelism is also part of the reason why single-CPU computing speed kept up with Moore's law for half a century. However, the present chapter will only address parallel computations that are enabled by using appropriate software on multiple processors.

10.2 More About Parallel Computing

A parallel computer provides the technical possibility, but whether or not parallel computing can be applied to a particular computational problem depends on the existence of parallelism and how it can be exploited in a form suitable for the parallel

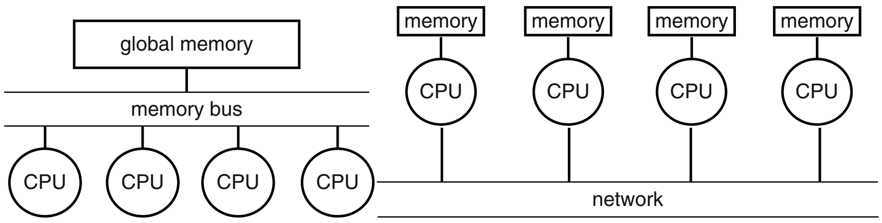

Fig. 10.1 A schematic layout of the shared-memory (*left*) and distributed-memory (*right*) parallel architectures

hardware. This section introduces the basic idea of parallel computing and its main ingredients.

10.2.1 Inspirations from Parallel Processing in Real Life

In everyday life, it is very common to see several people join forces to work on something together. Let us look at a scenario where two workers are set to paint a wall. Of course, each worker should first have a brush and a can of paint. Then, the two workers need to negotiate which area of the wall is to be painted by whom. When this work division is settled, each painter can start painting his or her assigned area, without having to interact with the other. The only assumption is that the two painters do not fight with each other when painting the boundary zone. In case more workers are available, the situation is similar except that the wall needs to be divided into more and smaller areas.

Compared with letting one worker paint the wall, employing several workers will normally speed up the project of wall painting. It has to be assumed, though, that the work division is reasonable, i.e., all the workers are able to finish their assigned areas using roughly the same amount of time. Otherwise the entire project of wall painting is not done until the slowest worker finishes his part. A second observation is that the initial negotiation of work division, which is not necessary for the single-worker case, should not cost too much time overhead. A third observation is that you cannot assign too many workers to paint a wall. That is, each worker should have a sufficiently large area to paint, without frequently coming over to others' areas and thereby slowing down each other. All in all, a speedup of the work for this very simple case is possible to achieve, but requires a large enough wall plus caution with respect to the work division.

There is, however, another important factor that is not addressed in the preceding example: Collaboration between the workers is often needed while the work is being carried out. Let us consider the example of bricklaying. Suppose a number of masons are hired to set up a brick wall. A possible means of work division is to let each mason have a vertical strip, inside of which she can lay the bricks layer-wise from bottom to top. For reasons of structural strength, the masons should preferably always work on the same layer of bricks. In addition, the positioning of bricks by neighboring masons should match up, without leaving large gaps between sections of bricks. The masons therefore need to collaborate, i.e., adopt a synchronized pace and exchange ideas frequently.

10.2.2 From Serial Computing to Parallel Computing

Not unlike the above cases of parallel processing in real life, parallel computing is only meaningful when an entire computational problem can be somehow divided evenly among the processors, which are to work concurrently, with coordination in the form of information exchange and synchronization. The purpose of information exchange is to provide private data to other processors that need them, whereas synchronization has the purpose of keeping the processors at same pace when desired. Both forms of coordination require the processors to communicate with each other. Moreover, for parallel computing to be beneficial, it is necessary that all the processors have a sufficient work load, and that the extra work caused by parallelization is limited.

Parallelism and Work Division

Parallelism in a computational problem arises from the existence of computations that can be carried out concurrently. The two most common forms of parallelism in scientific computing are *task parallelism* and *data parallelism.* Task parallelism arises from a set of standalone computational tasks that have a clear distinction between each other. Some of the tasks may need to follow the completion of other tasks. The possibility of parallel computing in this context arises from the concurrent execution of multiple tasks, possibly with some interaction between each other.

A simple example of task parallelism is parameter analysis: for example, a coefficient of a partial differential equation (PDE) is perturbed systematically to study its impact on the solution of the PDE. Here, each different value of the coefficient requires a new solution of the PDE, and all these different PDE-solving processes can proceed totally independently of each other. Therefore, one-PDE solving process, i.e., a computational task, can be assigned to one processor.

In the case that a certain dependency exists among (some of) the tasks, it is necessary to set up a dependency graph involving all the tasks. Parallel execution starts with the tasks that depend on nobody else and progressively includes more and more tasks that become ready to execute. When there are more waiting tasks than available processors, a dynamically updated job queue needs to be set up and possibly administered by a master processor. The worker processors can then each take one task off the queue, work on the assigned task, and come back for more until the queue is empty. Also, the worker processors may need to communicate with each other, if the assigned tasks require interaction.

When the number of tasks is small and the number of processors is large, an extra difficulty arises, because multiple processors have to share the work of one task. This is a more challenging case for parallel computing and requires further work division. The answer to this problem is data parallelism, which is a much more widespread source of parallelism in scientific computing than task parallelism. Data parallelism has many types and loosely means that identical or similar operations can be carried out concurrently on different sections of a data structure. Data parallelism can be considered as extreme task parallelism, because the operations applied on each unit of the data structure can be considered a small task. Data parallelism can thus produce, if needed, fine-grain parallelization, whereas task parallelism normally results in coarse-grain parallelization. The somewhat vague distinction between the two forms of parallelism lies in the size and type of the tasks. It is also worth noticing that some people restrict the concept of data parallelism to loop-level parallelism, but we in this chapter we will refer data parallelism more broadly as parallelism that arises from dividing some global data structure and the associated computing operations. Unlike in the case of task parallelism, where the tasks can be large and independent of each other, the different pieces in data parallelism are logically connected through an underlying global data structure. Work division related to data parallelism, which is the first step of transforming a serial task into its parallel counterpart, can be non-trivial. So will be the interaction between the processors. Our focus is therefore on data parallelism in the following text.

10.2.3 Example 1 of Data Parallelism

Let us consider a simple example of evaluating a uni-variable function $f(x)$ for a set of x values. In a typical computer program the x values and evaluation results are stored in one-dimensional arrays, and the computational work is implemented as the following `for`-loop in C/C++ syntax:

```
for (i=0; i<n; i++)
  y[i] = f(x[i]);
```

An important observation is that the evaluations of f for different x values are independent of each other. If each function evaluation $f(x)$ is considered as a separate task, this example can be categorized as task parallel. However, it is more common to consider the present example as data parallel, because the same function is applied to an array of numerical values. Parallelism arises from dividing the x values into subsets, and one subset is given to one processor. Suppose all the evaluations of f are equally expensive and all processors are equally powerful. Fairness thus requires that each processor get the same number of x values. (For cases of different costs of the evaluations and inhomogeneous processors, we refer to Exercises 10.2 and 10.3.)

Let P denote the number of processors and n the number of x values; then each processor should take n/P values. In case n is not divisible by P, the following formula provides a fair work division:

$$n_p = \left\lfloor \frac{n}{P} \right\rfloor + \begin{cases} 1 & \text{if } p < \text{mod}(n, P), \\ 0 & \text{else}, \end{cases} \tag{10.4}$$

where p denotes the processor id, which by convention starts from 0 and stops at $P - 1$. In (10.4) the symbol $\lfloor \cdot \rfloor$ denotes the floor function that gives the same result as integer division in a computer language, and $\text{mod}(n, P)$ denotes the remainder of the integer division n/P. It is clear that the maximum difference between the different n_p values is one, i.e., the fairest possible partitioning.

Example 10.1. If P is 6 and n is 63, then we have

$$\left\lfloor \frac{n}{P} \right\rfloor = \left\lfloor \frac{63}{6} \right\rfloor = 10 \quad \text{and} \quad \text{mod}(n, P) = \text{mod}(63, 6) = 3.$$

This means that processors with id $p = 0, 1, 2$ will be assigned with $n_p = 11$, whereas the other processors get $n_p = 10$.

■

Finding the number of x values for each processor, e.g., by using (10.4), does not complete the work division yet. The next information is about which x values should be assigned to each processor. There can be many different solutions to this partitioning problem, and its impact on the resulting parallel performance is normally larger for distributed-memory systems than for the shared-memory counterparts. Often, letting each processor handle a contiguous piece of memory is performance friendly. In this spirit, the index set $\{0, 1, \dots, n-1\}$ corresponding to the array entries can be segmented into P pieces, where the start position for processor p is

$$i_{\text{start},p} = p \left\lfloor \frac{n}{P} \right\rfloor + \min(p, \text{mod}(n, P)). \tag{10.5}$$

Example 10.2. Following the above example, where $P = 6$ and $n = 63$, we can use (10.5) to find

$$\begin{aligned}
i_{\text{start},0} &= 0 \times \left\lfloor \frac{63}{6} \right\rfloor + \min(0, \text{mod}(63, 6)) = 0, \\
i_{\text{start},1} &= 1 \times \left\lfloor \frac{63}{6} \right\rfloor + \min(1, \text{mod}(63, 6)) = 11, \\
i_{\text{start},2} &= 2 \times \left\lfloor \frac{63}{6} \right\rfloor + \min(2, \text{mod}(63, 6)) = 22, \\
i_{\text{start},3} &= 3 \times \left\lfloor \frac{63}{6} \right\rfloor + \min(3, \text{mod}(63, 6)) = 33, \\
i_{\text{start},4} &= 4 \times \left\lfloor \frac{63}{6} \right\rfloor + \min(4, \text{mod}(63, 6)) = 43, \\
i_{\text{start},5} &= 5 \times \left\lfloor \frac{63}{6} \right\rfloor + \min(5, \text{mod}(63, 6)) = 53.
\end{aligned}$$

■

When the work division is ready, i.e., n_p and $i_{\text{start},p}$ are computed, the parallelized computation can be implemented simply as follows:

```
for (i=i_start_p; i<i_start_p+n_p; i++)
  y[i] = f(x[i]);
```

The above code segment implicitly assumes that all processors can access the entire `x` and `y` arrays in memory, although each processor is only assigned to work on a distinct segment. This assumption fits very well with a shared-memory architecture, i.e., arrays x and y are shared among all processors. Such shared-memory programming closely resembles standard serial programming, but there are differences that require special attention. For example, the index integer i can not be shared between processors, because each processor needs to use its own i index to traverse an assigned segment of x and y.

In the case of distributed memory, the rule of the thumb is to avoid allocating global data structures if possible. Therefore, each processor typically only allocates two local arrays, x_p and y_p, which are of length n_p and correspond to the assigned piece of the global x and y arrays. Distributed-memory programming thus has to consider more details, such as local data allocation and mapping between local and global indices. However, the pure computing part on a distributed-memory architecture is rather simple, as follows:

```
for (i=0; i<n_p; i++)
  y_p[i] = f(x_p[i]);
```

10.2.4 Example 2 of Data Parallelism

Parallelizing the previous example of function evaluation is very simple, because the processors can work completely independently of each other. However, such embarrassingly parallel examples with no collaboration between the processors are rare in scientific computing. Let us now look at another example where inter-processor collaboration is needed.

The composite trapezoidal rule of numerical integration was derived in Chap. 1, where the original formula was given as (1.16). Its purpose is to approximate the integral $\int_a^b f(x)dx$ using $n + 1$ equally spaced samples. Before we discuss its parallelization, let us first recall the computationally more efficient formula of this numerical integration rule, which was given earlier as (6.2):

$$\int_a^b f(x)dx \approx h\left(\frac{1}{2}(f(a) + f(b)) + \sum_{i=1}^{n-1} f(a + ih)\right), \quad h = \frac{b-a}{n}. \tag{10.6}$$

Looking at (10.6), we see that it is quite similar to the previous example of function evaluation. The difference is that the evaluated function values need to be summed up, where the two end-points receive a half weight, in comparison with the $n - 1$ inner points. An important observation is that summation of a large set of values can be achieved by letting each processor sum up a distinct subset and then adding up all the partial sums. Similar to the previous example, parallelization starts with dividing the $n-1$ inner-point function evaluations on P processors, which also carry out a subsequent partial summation. More specifically, the sampling points $x_1, x_2, \ldots, x_{n-1}$, where $x_i = a + ih$, are segmented into P equal pieces. Then each processor can compute a partial sum in the form:

$$s_p = \sum_{i=i_{\text{start},p}}^{i_{\text{start},p}+n_p-1} f(x_i), \tag{10.7}$$

where, for this example, assuming equally expensive function evaluations and homogeneous processors, we have

$$n_p = \left\lfloor \frac{n-1}{P} \right\rfloor + \begin{cases} 1 & \text{if } p < \text{mod}(n-1, P), \\ 0 & \text{else,} \end{cases} \tag{10.8}$$

and

$$i_{\text{start},p} = 1 + p \left\lfloor \frac{n-1}{P} \right\rfloor + \min(p, \text{mod}(n-1, P)). \tag{10.9}$$

Example 10.3. Suppose $n = 200$ and $P = 3$; then we have from (10.8) $n_0 = 67$, $n_1 = n_2 = 66$. Following (10.9), we have $i_{\text{start},0} = 1$, $i_{\text{start},1} = 68$, and $i_{\text{start},2} = 134$. More specifically, processor 0 is responsible for inner points from x_1 until x_{67}, processor 1 works for inner points from x_{68} until x_{133}, and processor 2 is assigned to inner points from x_{134} until x_{199}.

■

So parallel computing in connection with the composite trapezoidal rule arises from the fact that each processor can independently compute its s_p following (10.7). However, when the processors have finished the local computation of s_p, an additional computation of the following form is needed to complete the entire work:

$$\int_a^b f(x)dx \approx h \left(\frac{1}{2}(f(a) + f(b)) + \sum_{p=0}^{P-1} s_p \right). \tag{10.10}$$

Note that the local results s_p are so far only available on the different processors. Therefore, before we can calculate $\sum_{p=0}^{P-1} s_p$, the processors have to somehow share all the s_p values. There are two approaches. In the first approach, we designate one processor as the master and consider all the other processors as slaves. All the slave processors pass their s_p value to the master, which then computes the final result using (10.10). In case the slaves also want to know the final result, the master can send it to all the slaves. In the second approach, all processors have an equal role. The first action on each processor is to pass its own s_p value to all the other $P - 1$ processors, and get in return $P - 1$ different s_p values. The second action on each processor is then to independently carry out the computation of (10.10).

It is worth noting that the processors have to collaborate in calculating $\sum_{p=0}^{P-1} s_p$. In the first approach above, the communication is of the form all-to-one, followed possibly by a subsequent one-to-all communication. In the second approach, the communication is of the form all-to-all. No matter which approach, we can imagine that a considerable amount of communication programming is needed for summing up the different s_p values possessed by the P processors. Luckily, standard parallel programming libraries and languages have efficient built-in implementations for such collective communications with associated calculation (i.e., summation in this example) that are called *reduction operations*. Users thus do not have to program these from scratch. The time cost of these built-in reduction operations is typically on the order $\mathcal{O}(\log_2 P)$.

As an alternative to the above data-parallel approach to the example of the composite trapezoidal integration rule, let us now consider another approach that is more in the style of task parallelism. To this purpose, let us first note the following relation:

$$\int_a^b f(x)dx = \sum_{j=0}^{P-1} \int_{X_j}^{X_{j+1}} f(x)dx, \tag{10.11}$$

where $a = X_0 < X_1 < X_2 < \ldots < X_{P-1} < X_P = b$ is an increasing sequence of x-coordinates that coincide with a subset of the sampling points $\{x_i\}$. In other words, formula (10.11) divides the integral domain $[a, b]$ into P segments. The following new parallelization approach is motivated by the fact that a serial implementation of the composite trapezoidal rule probably already exists, such as Algorithm 6.2. The idea now is to let processor p call the serial trapezoidal(X_p, X_{p+1}, f, n_p) function as an independent task to approximate $\int_{X_p}^{X_{p+1}} f(x)dx$. Afterward, the global sum is obtained by a reduction operation involving all the P processors. We have to ensure that n_p this time gives a division of the n sampling intervals, instead of dividing the $n-1$ inner points. (For computing the X_p values that match the P-way division of the n intervals, we refer the reader to Exercise 10.4.) The advantage is the reuse of an existing serial code, plus a more compact parallel implementation. The actual computation on each processor is done by a single call to the trapezoidal(X_p, X_{p+1}, f, n_p) function instead of a `for`-loop. The disadvantage is a small number of duplicated function evaluations and floating-point arithmetic operations, which arise because computation of $0.5f(X_p)$ is carried out on both processor $p-1$ and processor p. In practical cases, where n is very large, the cost of the duplicated operations can be neglected.

10.2.5 Example 3 of Data Parallelism

The above example of computing the composite trapezoidal rule in parallel only requires one collective communication in the form of a reduction operation at the end of the computation. Let us now look at another example where inter-processor collaboration is more frequent.

Consider an explicit numerical scheme for solving the 1D diffusion problem (7.82)–(7.85) from Sect. 7.4. The finite difference formula for the $n-1$ inner points is, as already given in (7.91),

$$u_i^{\ell+1} = u_i^\ell + \frac{\Delta t}{\Delta x^2}\left(u_{i-1}^\ell - 2u_i^\ell + u_{i+1}^\ell\right) + \Delta t f(x_i, t_\ell) \quad \text{for } i = 1, 2, \ldots, n-1. \tag{10.12}$$

In addition, the Dirichlet boundary condition (7.83), i.e., $u(0,t) = D_0(t)$, is realized as

$$u_0^{\ell+1} = D_0(t_{\ell+1}), \tag{10.13}$$

whereas the Neumann boundary condition (7.84), i.e., $\partial u/\partial x(1,t) = N_1(t)$, is realized as

$$u_n^{\ell+1} = u_n^\ell + 2\frac{\Delta t}{\Delta x^2}\left(u_{n-1}^\ell - u_n^\ell + N_1(t_\ell)\Delta x\right) + \Delta t f(1,t_\ell), \tag{10.14}$$

see Sect. 7.4.3 for the details.

Parallelism in formula (10.12) is due to the fact that the computations on any two inner points $u_i^{\ell+1}$ and $u_j^{\ell+1}$ are independent of each other. More specifically, to compute the value of $u_i^{\ell+1}$, we rely on three nodal values from the previous time step: u_{i-1}^ℓ, u_i^ℓ, u_{i+1}^ℓ, thus none of the other $u^{\ell+1}$ values. The $n-1$ inner points can actually be updated simultaneously for the same time level $\ell+1$. Work division is the same as partitioning these inner points. If the index set $\{1,2,\ldots,n-1\}$ is segmented into P contiguous pieces, the work division is equivalent to decomposing the solution domain into P subdomains, as is evident in Fig. 10.2. Actually, for many PDE problems, it is more natural and general to let domain decomposition give rise to the work and data division, so that each processor is assigned with a subdomain.

After a work division is decided, the local work of each processor at time step $\ell+1$ is to compute $u_i^{\ell+1}$ using (10.12) for a subset of the i indices $\{1,2,\ldots,n-1\}$. In addition, the processor that is responsible for x_0 has to update $u_0^{\ell+1}$ using (10.13), and similarly the processor responsible for x_n needs to compute $u_n^{\ell+1}$ using (10.14). It should be stressed again that concurrency in formula (10.12) assumes that the inner points on the same time level are updated. In other words, no processor should be allowed to proceed to the next time step before all the other processors have finished the current time step. Otherwise the computational results will be incorrect. Such a coordination among processors can typically be achieved by a built-in synchronization operation called *barrier*, which forces all processors to wait for the slowest one. On a shared-memory system, the barrier operation is the only needed inter-processor communication. All nodal values of u^ℓ are accessible by all processors on a shared-memory architecture, and therefore there is no need to communicate while each processor is computing its assigned portion of $u^{\ell+1}$.

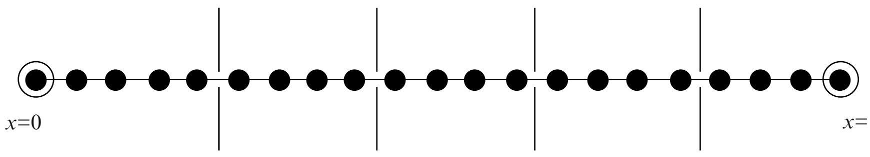

Fig. 10.2 An example of partitioning the 1D computational domain $x \in (0,1)$, The *leftmost* and *rightmost* mesh points are marked for treatment of the physical boundary conditions, whereas the inner points are divided fairly among the processors

On a distributed-memory system, the parallelization is a bit more complex. We recall that a processor with only a local memory should avoid allocating global data structures if possible. So processor p should ideally only operate on two local arrays $\mathbf{u}_p^\ell$ and $\mathbf{u}_p^{\ell+1}$, both of length n_p, which contain, respectively, the assigned segments of the u_i^ℓ and $u_i^{\ell+1}$ values. The data segmentation can use the formulas (10.8) and (10.9). However, as indicated by (10.12), computing the leftmost and rightmost values of $\mathbf{u}_p^{\ell+1}$ requires one u^ℓ value each from the two neighboring subdomains. To avoid costly `if`-tests when computing the leftmost and rightmost values of $\mathbf{u}_p^{\ell+1}$, it is therefore more convenient to extend the two local arrays by one value at both ends. These two additional points are called *ghost points*, whose values participate in the owner subdomain's computations but are provided by the neighboring subdomains through communication. We refer the reader to Fig. 10.3 for an illustration. It should also be noted that on processor 0 the left ghost point coincides with the left physical boundary point $x = 0$, whereas on processor $P - 1$ the right ghost point coincides with the right physical boundary point $x = 1$.

Compared with a corresponding implementation on a shared-memory system, the implementation on a distributed-memory system is different in its inter-processor communications, in addition to using local data arrays. Here, each pair of neighboring subdomains has to explicitly exchange one nodal value per time step. For example, the value of $u_{p,1}^{\ell+1}$ that is computed on processor p needs to be sent to processor $p - 1$, which in return sends back the computed value on its rightmost inner point. A similar data exchange takes place between processors p and $p + 1$. The data exchanges need to be carried out before proceeding to the next time step. The resulting communications involve pairs of processors, commonly known as *one-to-one* communications. In this example, these one-to-one communications implicitly ensure that the computations on the neighboring processors are synchronized. There is therefore no need for a separate barrier operation, which is required in the shared-memory implementation. The complete numerical scheme suitable for a distributed-memory computer is given in Algorithm 10.1.

Remarks

To a beginner, whether or not parallelism exists in a computational problem can seem mysterious. A rule of the thumb is that there must be a sufficient amount of computational work and also that (parts of) the computations must not be dependent on each other.

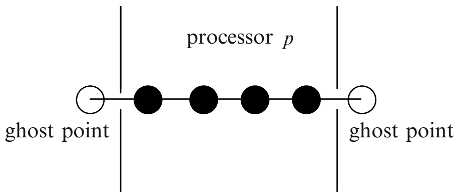

Fig. 10.3 An example of a subdomain that is assigned to processor p. The assigned mesh points consist of two ghost points and a set of inner points

Algorithm 10.1

Explicit Scheme for Solving the 1D Diffusion Problem (7.82)–(7.85) on a Distributed-Memory Computer.

```
Given $n+1$ as the total number of global mesh points, $\Delta x = 1/n$,
$\Delta t$ as the time step size, $m$ as the number of time steps, $\alpha = \Delta t/\Delta x^2$,
and $P$ as the total number of processors.
On processor $p$ $(0 \le p \le P-1)$:
    if $p < \mathrm{mod}(n-1, P)$ then
        $n_p = \lfloor \frac{n-1}{P} \rfloor + 1$
    else
        $n_p = \lfloor \frac{n-1}{P} \rfloor$
    $i_{\mathrm{start},p} = p \lfloor \frac{n-1}{P} \rfloor + \min(p, \mathrm{mod}(n-1, P))$
    Allocate local arrays $\mathbf{u}_p^\ell$ and $\mathbf{u}_p^{\ell+1}$ both of length $n_p + 2$
    Initial condition: $u_{p,i}^\ell = I((i_{\mathrm{start},p} + i)\Delta x)$, for $i = 0, 1, \ldots, n_p + 1$
    for $\ell = 0, 1, \ldots, m-1$
        for $i = 1, 2, \ldots, n_p$
            $u_{p,i}^{\ell+1} = u_{p,i}^\ell + \alpha\left(u_{p,i-1}^\ell - 2u_{p,i}^\ell + u_{p,i+1}^\ell\right)$
                    $+\Delta t f((i_{\mathrm{start},p} + i)\Delta x, t_\ell)$
        if $p = 0$ then
            $u_{p,0}^{\ell+1} = D_0(t_{\ell+1})$
        else
            receive value from processor $p-1$ into $u_{p,0}^{\ell+1}$
            send value of $u_{p,1}^{\ell+1}$ to processor $p-1$
        if $p = P-1$ then
            $u_{p,n_p+1}^{\ell+1} = u_{p,n_p+1}^\ell + 2\alpha\left(u_{p,n_p}^\ell - u_{p,n_p+1}^\ell + N_1(t_\ell)\Delta x\right)$
                    $+\Delta t f(1, t_\ell)$
        else
            send value of $u_{p,n_p}^{\ell+1}$ to processor $p+1$
            receive value from processor $p+1$ into $u_{p,n_p+1}^{\ell+1}$
        Data copy before next time step: $\mathbf{u}_p^\ell = \mathbf{u}_p^{\ell+1}$
    end for
```

For instance, an addition operation between two scalar values is obviously not a subject for parallelization, whereas adding two long vectors is embarrassingly parallel. Solving a single ordinary differential equation (ODE), even if a huge number of time steps are employed, is normally serial by nature. This is because each time step contains too little work and the time steps have to be carried out one after another. For a system of ODEs, the situation with respect to parallel computing can be improved, especially when the number of involved ODEs is large. Parallelism can be found within each time step, either because the ODEs are handled separately in an explicit scheme, or because an implicit scheme solves a linear system that has some level of parallelism. Nevertheless, the number of ODEs in a system is normally not as many as the number of available processors, so exploiting such limited parallelism can be challenging. The best situation arises in an ODE–PDE coupled problem, because many numerical methods involve solving the ODE part for every spatial grid point. In this case, when the number of spatial grid points is large, the task of solving all these ODEs is again embarrassingly parallel.

Even when parallelism exists, work division can still be a challenge. Our previous examples are simple, because the work division there is essentially a one-dimensional partitioning applied to structured data entities. In the case of two- or three-dimensional unstructured computational meshes, partitioning not only has to make sure that each processor has approximately the same amount of work, but also has to keep the amount of resulting communication as low as possible. In the case of time-dependent problems that have a dynamically changing work load, dynamic load balancing can be another challenge.

10.2.6 Performance Analysis

The prominent reason for adopting parallel computing, time saving is also the main measure for evaluating the quality of a parallelized code. If we denote by $T(P)$ the computing time for solving a problem using P processors, the very important concept of *speedup* is thus defined as

$$S(P) = \frac{T(1)}{T(P)}, \tag{10.15}$$

where $T(1)$ should be the computing time used by a serial code, and not a parallelized code run on one processor. This is because a parallel implementation typically has some additional operations that are not needed in a serial implementation.

An ideal result of parallelization is that the entire serial code is parallelizable with a perfect work division and that the parallelization-induced communication cost is negligible. In such an ideal situation, we can expect that $T(P)$ is exactly one Pth of $T(1)$, thus $S(P) = P$. The ideal situation is rarely achievable, so we understandably would like the value of $S(P)$ to be close to P; the larger the better quality of the parallelization. An equivalent quality measure, called *parallel efficiency*, can be defined as

$$\eta(P) = \frac{S(P)}{P} = \frac{T(1)}{P\,T(P)}, \tag{10.16}$$

which will have a normalized value between 0 and 1.

Amdahl's Law

The first obstacle to perfect parallelization is that there can exist some bits of a serial code that are inherently sequential. Suppose a serial code has a fraction of size α

that is not parallelizable. Consequently, even if the remaining fraction $1 - \alpha$ is perfectly parallelized, there is a theoretical upper limit on the best possible achievable speedup. Amdahl's law [3] states the following:

$$S(P) \le \frac{T(1)}{\left(\alpha + \dfrac{1-\alpha}{P}\right) T(1)} = \frac{1}{\alpha + \dfrac{1-\alpha}{P}} < \frac{1}{\alpha}. \tag{10.17}$$

For instance, if 1% of a serial code is not parallelizable, the best possible speedup cannot exceed $1/\alpha = 1/0.01 = 100$, no matter how many processors are used.

Amdahl's law was derived in the 1960s, a time when parallel computing was still in its infancy. The theory was in fact used as an argument against this new-born technology. An important observation is that Amdahl's law assumes a fixed problem size, no matter how many processors are used. It in effect says that it does not pay to use too many processors if a threshold of parallel efficiency is to be maintained (see Exercise 10.6). Amdahl's law indeed gives an unnecessarily pessimistic view, because the non-parallelizable fraction of a code typically decreases as the problem size increases. In other words, it pays off to use more processors when the problem becomes larger.

Gustafson–Barsis's Law

There is another theory, known as Gustafson–Barsis's law [16] that looks at the issue of scaled speedup. The basic idea is that P processors should be used to solve a problem that is P times the problem size on one processor. Suppose the absolute amount of serial computing time does not grow with problem size, and let α_P denote the fraction of serial computing time in $T(P)$; then $T(1)$ would have been $(\alpha_P + P(1 - \alpha_P))\, T(P)$. The scaled speedup can be computed as follows:

$$S(P) = \frac{(\alpha_P + P(1-\alpha_P))\, T(P)}{T(P)} = \alpha_P + P(1-\alpha_P) = P - \alpha_P(P-1). \tag{10.18}$$

An important assumption of Gustafson–Barsis's law is that the fraction of serial computing time becomes increasingly small as the problem size increases, which is true for most parallel codes. The consequence is that it is possible to achieve a scaled speedup very close to P, provided the problem size is large enough.

Gustafson–Barsis's law can also be used to estimate speedup without knowing $T(1)$. For example, if $\alpha_P = 10\%$ is assumed for $P = 1{,}000$, then the scaled speedup of using 1,000 processors is, according to (10.18), $P - \alpha_P(P - 1) = 1000 - 0.1(1000 - 1) = 900.1$. This gives a corresponding parallel efficiency of 90%.

As indicated by Gustafson–Barsis's law, when a desired speedup value is not achievable due to Amdahl's law, it is time to consider enlarging the problem size. After all, solving larger problems is the other important reason for adopting parallel computing.

10.2.7 Overhead Related to Parallelization

Neither Amdahl's law nor Gustafson–Barsis's law has considered the impact of overhead on the achievable speedup. To a great extent, the attention on the fraction of non-parallelizable computation is over emphasized by both theories. In many large-scale problems, however, the actual fraction of non-parallelizable computation is virtually zero. The real obstacles to achieving high speedup values are different types of overhead that are associated with the parallelization.

First of all, a parallelized algorithm can introduce additional computations that were not present in the original serial algorithm. For example, work division often requires a few arithmetic operations, such as using (10.8) and (10.9). The cost of partitioning unstructured computational meshes, in particular, is often considerable. Second, when perfect work division is either impossible or too expensive to achieve, the resulting work load imbalance will decrease the actual speedup. Third, explicit synchronization between processors may be needed from time to time. Such operations can be costly. Fourth, communication overhead can surely not be overlooked in most parallel applications. For a collective communication involving all P processors, the cost is typically of the order $\mathcal{O}\left(\lceil \log_2 P \rceil\right)$, where $\lceil \cdot \rceil$ denotes the ceiling function which gives the smallest integer value that is equal to or larger than $\log_2 P$. For one-to-one communication on distributed-memory systems, an idealized model for the overhead of transferring a message with L bytes from one processor to another is as follows:

$$t_C(L) = \tau + \xi L, \tag{10.19}$$

where τ is the so-called *latency*, which represents the start-up time for communication, and ξ is the cost for transferring one byte of data. By the way, $1/\xi$ is often referred to as the *bandwidth* of the communication network. The parameter values of τ and ξ can vary greatly from system to system. The reason for saying that (10.19) is an idealized model is because it does not consider the possible situation of several processors competing for the network, which can reduce the network's effective capacity. Also, the actual curve of $t_C(L)$ is often of a staircase shape, instead of a straight line with constant slope. (For example, transferring one double-precision value can take the same time as transferring a small number of double-precision values in one message.) Although shared-memory systems seem to be free of one-to-one communications, there is related overhead behind the scene. For example, each processor typically has own private cache. We remark that cache is a small piece of fast memory that dynamically duplicates a small portion of the main memory. The speed of cache is much faster than that of the main memory, and the use of cache is meant to overcome the memory-speed bottleneck. A variable in the shared memory can thus risk being updated differently in several caches. Keeping the different caches coherent between each other can therefore incur costly operations in the parallel system.

It should be mentioned that there are also factors that may be speedup friendly. First, many parallel systems have the capability of carrying out communications at the same time as computations. This allows for the possibility of hiding the communication overhead. However, to enable communication–computation overlap can be a challenging programming task. Second, it sometimes happens that by using many processors on a distributed-memory system, the subproblem per processor suddenly acquires a much better utilization of the local cache, compared with solving the entire serial problem using one CPU and its local cache. This provides the potential of superlinear speedup, i.e., $S(P) > P$, for particular problem sizes.

Example 10.4. Let us analyze the speedup of the parallel composite trapezoidal rule (10.7)–(10.10). As we can see from (10.6), the serial algorithm involves $n+1$ function evaluations plus $\mathcal{O}(n)$ additions and multiplications. If we assume that the function evaluations are much more expensive than the floating-point operations, the computational cost of the serial algorithm is

$$T(1,n) = (n+1)t_f,$$

where t_f denotes the cost of one function evaluation. The computational cost of the parallel algorithm is

$$T(P,n) = \left\lceil \frac{n-1}{P} \right\rceil t_f + C\,\lceil \log_2 P \rceil + 2t_f, \tag{10.20}$$

where the first term corresponds to the cost of computing s_p using (10.7), the second term corresponds to the cost of a reduction operation needed to sum all the s_p values, and the last term corresponds to the cost of computing $f(a)$ and $f(b)$ in the end. Therefore, the speedup will be of the following form:

$$\begin{aligned} S(P,n) = \frac{T(1,n)}{T(P,n)} &= \frac{(n+1)t_f}{\lceil \frac{n-1}{P} \rceil t_f + C\,\lceil \log_2 P \rceil + 2t_f} \\ &= \frac{n+1}{\lceil \frac{n-1}{P} \rceil + 2 + \tilde{C}\,\lceil \log_2 P \rceil}. \end{aligned} \tag{10.21}$$

Of course, to compute the actual values of $S(P,n)$, we need to know the value of the constant $\tilde{C} = C/t_f$. This can be achieved by measuring $T(P,n)$ for a number of different choices of P and n and estimating the value of $\tilde{C}$ by the method of least squares. Even without knowing the exact value of $\tilde{C}$ in (10.21), we can say that the parallel efficiency $\eta(P)$ will get further away from 100% for increasing P. This is because the $\tilde{C}\,\lceil \log_2 P \rceil$ term in the denominator of (10.21) increases with P, meaning that the speedup will eventually saturate and decrease after passing a threshold value of P, dependent on the actual size of n and $\tilde{C}$.

■

Example 10.5. Table 10.1 shows some speedup results measured on a small cluster of PCs, where the interconnect is 1 GB-Ethernet. The parallel program is written using the message-passing interface (MPI [14, 26]) and implements the compos-

Table 10.1 Speedup results for computing the composite trapezoidal integration rule using an MPI program

P	1	2	4	8	16
$T(P)$	0.03909	0.02331	0.01599	0.008227	0.004398
$S(P)$	N/A	1.68	2.44	4.75	8.89
$\eta(P)$	N/A	0.84	0.61	0.59	0.56

ite trapezoidal integration rule, see Sect. 10.3.2. The integral to be approximated is $\int_0^1 \sin(\pi x)dx$ using $n = 10^6$ sampling intervals.

Due to the relatively small problem size and the slow interconnect, the speedup results in Table 10.1 are not very impressive. This example shows that communication overhead is an obstacle to good parallel efficiency, which typically deteriorates with increasing P.

■

In general, considering all the factors that have an impact on speedup, we should be prepared that a fix-sized problem often has an upper limit on the number of processors, beyond which speedup will decrease instead of increase.

10.3 Parallel Programming

So far we have discussed how serial scientific computations can be transformed into their parallel counterparts. The basic steps include parallelism identification, work division, and inter-processor collaboration. For the resulting computations to run on the parallel hardware, code implementation must be done accordingly. The hope of many people for an automatic tool that is able to analyze a serial code, find the parallelism, and insert parallelization commands has proved to be too ambitious. This observation is at least true for scientific codes of reasonable complexity. Therefore, some form of manual parallel programming is needed. There are currently three main forms of parallel programming: (1) recode in a specially designed parallel programming language, (2) annotate a serial code with compiler directives to provide the compiler with hints for parallelization tasks, and (3) restructure a serial code and insert calls to library functions for explicitly enforcing inter-processor collaboration.

The present section will consider the two latter options. More specifically, we will briefly explain the use of OpenMP [1] and MPI [14, 26], which target shared-memory and distributed-memory systems, respectively. Both are well-established standards of application programming interfaces for parallelization and thus provide code portability. For the newcomer to parallel programming, the most important thing is not the syntax details, which can be easily found from textbooks or online resources; rather, it is important to see that parallel programming requires a "mental picture" of multiple execution streams, which often need to communicate with each other by data exchange and/or synchronization.

10.3.1 OpenMP Programming

The OpenMP standard is applicable to shared-memory systems. It assumes that a parallel program has one or several parallel regions, where a parallel region is a piece of code that can be parallelized, e.g., a `for`-loop that implements the composite trapezoidal integration rule (10.6). Between the parallel regions the code is executed sequentially just like a serial program. Within each parallel region, however, a number of threads are spawned to execute concurrently. The programmer's responsibility is to insert OpenMP directives together with suitable clauses, so that an OpenMP-capable compiler can use these hints to automatically parallelize the annotated parallel regions. A great advantage is that a non-OpenMP-capable compiler will ignore the directives and treat the code as purely serial.

The OpenMP directive for constructing a parallel region is `#pragma omp parallel` in the C and C++ programming languages. The two most important OpenMP directives in C/C++ for parallelization are (1) `#pragma omp for` suitable for data parallelism associated with a `for`-loop, and (2) `#pragma omp sections` suitable for task parallelism. In the Fortran language, the OpenMP directives and clauses have slightly different names. For in-depth discussions about OpenMP programming, we refer the reader to [8] and [9].

Example 10.6. Let us show below an OpenMP parallelization of the composite trapezoidal rule (10.6), implemented in C/C++. Since the computational work in a serial implementation is typically contained in a `for`-loop, the OpenMP parallelization should let each thread carry out the work for one segment of the `for`-loop. This will result in a local s_p value on each thread, as described in (10.7). All the s_p values will then need to be added up by a reduction operation as mentioned in Section 10.2.2. Luckily, a programmer does not have to explicitly specify how the `for`-loop is to be divided, which is handled automatically by OpenMP behind the scene. The needed reduction operation is also incorporated in OpenMP's `#pragma omp for` directive:

```
  h = (b-a)/n;
  sum = 0.;

#pragma omp parallel for reduction(+:sum)
  for (i=1; i<=n-1; i++)
    sum += f(a+i*h);

  sum += 0.5*(f(a)+f(b));
  sum *= h;
```

In comparison with a serial implementation, the only difference is the line starting with `#pragma omp parallel for`, which is the combined OpenMP directive for parallelizing a single `for`-loop that constitutes an entire parallel region. The `reduction(+:sum)` clause on the same line enforces, at the end of the loop, a reduction operation of adding all the `sum` variables over all threads. Additional clauses can also be added. For example, `schedule(static,chunksize)` will result in loop iterations that are divided statically into chunks of size `chunksize`, and the chunks are assigned to the different threads cyclically. An appropriate choice of work division and scheduling, with a suitable value of `chunksize`, is important for the parallel performance.

■

Example 10.7. Next, let us look at the most important section of an OpenMP implementation in C/C++ of the 1D diffusion problem (7.82)–(7.85), for which the explicit numerical method was given as (10.12)–(10.14).

```
    u_prev = (double*)malloc((n+1)*sizeof(double));
    u = (double*)malloc((n+1)*sizeof(double));
    t = 0.;

  #pragma omp parallel private(k)
    {
  #pragma omp for
      for (i=0; i<=n; i++)      /* enforce initial condition */
        u_prev[i] = I(i*dx);

      for (k=1; k<=m; k++) {    /* time integration loop */
  #pragma omp for schedule(static,100)
        for (i=1; i<n; i++)    /* computing inner points */
      u[i]=u_prev[i]+alpha*(u_prev[i-1]-2*u_prev[i]+u_prev[i+1])
               +dt*f(i*dx,t);

  #pragma omp single
        {
      u[n] = u_prev[n]
            +2*alpha*(u_prev[n-1]-u_prev[n]+N1(t)*dx)
            +dt*f(1,t); /* right physical boundary condition */
      t += dt;
      u[0] = D0(t);     /* left physical boundary condition */
        }

  #pragma omp for schedule(static,100)
        for (i=0; i<=n; i++)
      u_prev[i] = u[i];  /* data copy before next time step */
      }
    }
```

The above code contains considerably more instances of `#pragma` than in the previous example. Apart from parallelizing the `for`-loop for enforcing the initial condition, two `for`-loops, one for computing $u_i^{\ell+1}$ on the inner points and the other for copying array $\mathbf{u}^{\ell+1}$ to array $\mathbf{u}^{\ell}$, are also parallelized inside each time step. It can be seen that almost the entire code section is wrapped inside a large parallel region, indicated by `#pragma omp parallel`. This means that P threads are spawned at the entrance of the parallel region and stay alive throughout the region. This is why the OpenMP directive `#pragma omp single` is necessary to mark that only one thread does the work of incrementing the shared `t` variable and enforcing the two physical boundary conditions. Otherwise, letting all the threads repeat the same work will produce erroneous results.

Another possibility is to not use the large parallel region, but use `#pragma omp parallel for` in the three locations where `#pragma omp for` now stand. Such an approach will avoid using `#pragma omp single`, but will have to repeatedly spawn new threads and then terminate them, giving rise to more overhead in thread creation.

■

OpenMP programming is quite simple, because the standard consists only of a small number of directives and clauses. Much of the parallelization work, such as work division and task scheduling, is hidden from the user. On the one side, this programming philosophy provides great user friendliness. On the other side, however, the user has very little control over which thread accesses which part of the shared memory. This limit is normally bad with respect to the performance on a modern processor architecture, which relies heavily on the use of caches. To fully utilize a cache, it is important that the data items that are read from memory to cache should be reused as much as possible. Data locality – either when data items that are located close by in memory participate in one computing operation, or when one data item is repeatedly used in consecutive operations – gives rise to good cache usage, but is hard to enforce in an OpenMP program.

10.3.2 MPI Programming

MPI is a standard specification for message-passing programming on distributed-memory computers. On shared-memory systems it is also common to have MPI installations that are implemented using efficient shared-memory intrinsics for passing messages. Thus MPI is the most portable approach to parallel programming. There are two parts of MPI: The first part contains more than 120 functions and constitutes the core of MPI [26], whereas the second part contains advanced extensions [14]. Here, we only intend to give a brief introduction to MPI programming. For more in-depth learning, we refer the reader to the standard MPI textbooks [15, 23].

A message in the context of MPI is simply an array of data elements of a predefined data type. The logical execution units in an MPI program are called *processes*, which are initiated by the `MPI_Init` call at the start of a program and terminated by the `MPI_Finalize` call at the end. The number of started MPI processes is usually the same as the number of available processors, although one processor can in principle be assigned with several MPI processes. All the started MPI processes constitute a so-called global MPI *communicator* named `MPI_Comm_World`, which can be subdivided into smaller communicators if needed. Almost all the MPI functions require an input argument of type MPI communicator, which together with a process rank (between 0 and $P - 1$) is used to specify a particular MPI process.

Compared with OpenMP, MPI programming is clearly more difficult, not only because of the large number of available functions, but also because one MPI call normally requires many arguments. For example, the simplest function in C for sending a message is

```
MPI_Send(void *buf, int count, MPI_Datatype datatype,
         int dest, int tag, MPI_Comm comm)
```

The first three arguments constitute the outgoing message, by specifying the initial memory address of the array, the number of data elements, and the data type. The `dest` argument gives the rank of the receiving process relative to the `comm` communicator, whereas `tag` is an integer argument used to label the particular message.

Correspondingly, the simplest MPI function for receiving a message is

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype,
         int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Compared with the `MPI_Send` function, the extra argument in the `MPI_Recv` function is a pointer to a so-called `MPI_Status` object, which can be used to check some details of a received message.

Below we will give two examples of MPI programming in the C language. Compared with Examples 10.6 and 10.7, we will see that a programmer is responsible for more details of the parallelization. Communications have to be enforced explicitly in MPI. Despite the extra programming effort, parallel MPI programs are usually good with respect to data locality, because the local data structure owned by each MPI process is small relative to the global data structure. In addition, the user has full control over work division, which can be of great value for performance enhancement.

For a beginner, it is important to realize that each MPI process executes the same MPI program. The distinction between the processes, which are spawned by some parallel runtime system, is through the unique process rank. The rank typically determines the work assignment for each process. Moreover, an `if`-test with respect to a particular process rank can allow the chosen process to perform different operations than the other processes.

Example 10.8. The following is the most important part of an MPI implementation of the composite trapezoidal integration rule (10.6):

```
#include <mpi.h>

/*
  code omitted for defining function f
*/

int main (int nargs, char** args)
{
  int P, my_id, n, n_p, i_start_p, i, remainder;
  double a, b, h, x, s_p, sum;

  MPI_Init (&nargs, &args);
  MPI_Comm_size(MPI_COMM_WORLD,&P);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_id);

  n = 1000000; a = 0.; b = 1.;

  remainder = (n-1)%P;
  n_p = (my_id<remainder) ? (n-1)/P+1 : (n-1)/P;
  i_start_p = 1+my_id*((n-1)/P)+min(my_id,remainder);

  h = (b-a)/n;
  s_p = 0.; x = a+i_start_p*h;
  for (i=0; i<n_p; i++) {
    s_p += f(x); x += h;
  }

  MPI_Allreduce (&s_p,&sum,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);

  sum += 0.5*(f(a)+f(b));
  sum *= h;

  MPI_Finalize();
  return 0;
}
```

The MPI functions `MPI_Comm_size` and `MPI_Comm_rank` give, respectively, the number of processes P and the unique id (between 0 and $P-1$) of the current process. The work division, i.e., computation of n_p and $i_{\text{start},p}$, follows the formulas (10.8) and (10.9). The `for`-loop implements the formula (10.7) for computing s_p. The `MPI_Allreduce` function is a collective reduction operation involving all the processes. Its effect is that all the local s_p values are added up and the final result is stored in the `sum` variable on all processes. An alternative is to use the `MPI_Reduce` function, which runs faster than `MPI_Allreduce`, but the final result will only be available on a chosen process.

■

Example 10.9. Let us show below the most important part of an MPI implementation in C of Algorithm 10.1.

```
#include <mpi.h>
#include <malloc.h>

/*
  code omitted for defining functions I,f,D0,N1
*/

int main (int nargs, char** args)
{
  int P, my_id, n, n_p, i_start_p, i, k, m, remainder;
  int left_neighbor_id, right_neighbor_id;
  double dt, dx, alpha, t, *u_prev, *u;

  MPI_Init (&nargs, &args);
  MPI_Comm_size(MPI_COMM_WORLD,&P);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_id);

  /*
    code omitted for choosing n and dt, computing dx,alpha etc.
  */

  remainder = (n-1)%P;
  n_p = (my_id<remainder) ? (n-1)/P+1 : (n-1)/P;
  i_start_p = my_id*((n-1)/P)+min(my_id,remainder);
  u_prev = (double*)malloc((n_p+2)*sizeof(double));
  u = (double*)malloc((n_p+2)*sizeof(double));

  for (i=0; i<=n_p+1; i++)  /* enforce initial condition */
    u_prev[i] = I((i_start_p+i)*dx);

  left_neighbor_id = (my_id==0) ? MPI_PROC_NULL : my_id-1;
  right_neighbor_id = (my_id==P-1) ? MPI_PROC_NULL : my_id+1;

  t = 0.;
  for (k=1; k<=m; k++) {     /* time integration loop */
    for (i=1; i<=n_p; i++)  /* computing local inner points */
      u[i] = u_prev[i]+alpha*(u_prev[i-1]-2*u_prev[i]+u_prev[i+1])
            +dt*f((i_start_p+i)*dx,t);

    if (my_id==0)          /* left physical boundary condition */
      u[0] = D0(t+dt);

    if (my_id==P-1)        /* right physical boundary condition */
      u[n_p+1] = u_prev[n_p+1]
                +2*alpha*(u_prev[n_p]-u_prev[n_p+1]+N1(t)*dx)
                +dt*f(1,t);

    MPI_Sendrecv(&(u[1]),1,MPI_DOUBLE,left_neighbor_id,100,
         &(u[n_p+1]),1,MPI_DOUBLE,right_neighbor_id,100,
         MPI_COMM_WORLD,&status);
    MPI_Sendrecv(&(u[n_p]),1,MPI_DOUBLE,right_neighbor_id,200,
         &(u[0]),1,MPI_DOUBLE,left_neighbor_id,200,
         MPI_COMM_WORLD,&status);

    for (i=0; i<=n_p+1; i++)/* date copy before next time step */
      u_prev[i] = u[i];
    t += dt;
  }

  MPI_Finalize();
  return 0;
}
```

We can see that the above code uses a slightly modified Algorithm 10.1, in that two calls to the `MPI_Sendrecv` function replace two pairs of `MPI_Send` and `MPI_Recv`. The modified code is thus more compact. A more important motivation for the modification is to avoid potential communication *deadlocks*, which will arise if a pair of neighboring processes both start with `MPI_Recv` before calling the matching `MPI_Send` command. These two MPI processes will be blocked, i.e., none will be able to return from its `MPI_Recv` call, which relies on its neighbor to have issued the `MPI_Send` call. A bullet-proof sequence of `MPI_Send` and `MPI_Recv` calls is that process A calls `MPI_Send` and `MPI_Recv`, whereas process B calls `MPI_Recv` and `MPI_Send`.

■

10.3.3 Concluding Remarks

Parallel programming is an evolving science, with new tools, languages, and paradigms steadily coming along. We should therefore bear this in mind and be prepared for future developments. There is no guarantee that MPI and OpenMP will always be the dominant standards for parallel programming. For example, the arrival of multicore chips has given rise to a serious debate on the applicability of these two approaches. The important principles of parallel computing will survive, no matter the standing of programming techniques.

Let us emphasize again that syntax details are not the most important stuff. A programmer needs to have a correct notion about the multiple execution streams that go side by side in a parallel program. A proper work division should give each execution stream a distinct set of operations to carry out, balanced among all the streams. The execution streams normally need to communicate with each other for the purpose of exchanging data and/or synchronizing pace. Although parallel programming libraries can provide a lot of ready-made communication functions, challenges will remain, such as the treatment of possible conflicts between the streams and performance-harming pitfalls. Interested readers are therefore referred to the rich literature on parallel programming and computing.

10.4 Exercises

Exercise 10.1. The peak performance of the world's most powerful parallel computer has undergone an exponential growth. Table 10.2 lists the history between 1993 and 2009, according to the Top500 List [2]. Suppose the growth can be described mathematically by the following formula:

$$G(t) = G_0 \cdot 2^{\frac{t-t_0}{t^\star}},$$

where $G(t)$ denotes the development of the theoretical peak performance as a function of time, in terms of GFLOPS, and G_0 denotes the GFLOPS value at some initial time t_0. The purpose of this exercise is to estimate the value of $t^\star$, i.e., the average length of time over which the peak performance doubles. The method of least squares should be used, where the value of t_0 can be chosen as year 1990. If the same growth continues, what will the peak performance of the most powerful parallel computer be in year 2020? ◇

Exercise 10.2. We want to partition a set of n non-equal computational tasks among P homogeneous processors. Let us first suppose that half of the tasks cost twice the computing time of the other half's. How should a fair work division be? Next, suppose task 1 requires one unit of computing time, task 2 requires two time units, task 3 requires three time units, and so on. How should fair work division take place for this case? ◇

Table 10.2 The development of the world's most powerful parallel computer according to the Top500 List [2]

Time	System model	CPU type	CPUs	GFLOPS
June 1993	TMC CM5	SuperSPARC I 32MHz	1024	131
June 1994	Intel Paragon	Intel 80860 50MHz	3680	184
June 1995	Fujitsu VPP	Fujitsu 105MHz	140	235.79
June 1996	Hitachi SR2201	HARP-1E 150MHz	1024	307.2
June 1997	Intel Paragon	Pentium Pro 200MHz	7264	1453
June 1998	Intel Paragon	Pentium Pro 200MHz	9152	1830.4
June 1999	Intel Paragon	Pentium Pro 333MHz	9472	3154
June 2000	Intel Paragon	Pentium Pro 333MHz	9632	3207
June 2001	IBM SP	POWER3 375MHz	8192	12288
June 2002	NEC SX6	NEC 1000MHz	5120	40960
June 2005	IBM BlueGene/L	PowerPC 440 700MHz	65536	183500
June 2006	IBM BlueGene/L	PowerPC 440 700MHz	131072	367000
June 2008	IBM BladeCenter Cluster	PowerXCell 8i 3200MHz	122400	1375776
June 2009	IBM BladeCenter Cluster	PowerXCell 8i 3200MHz	129600	1456704

Exercise 10.3. Suppose now that the P processors are not equal in computing speed, where half of them have double the speed of the others. How should n equally expensive computational tasks be divided? ◇

Exercise 10.4. We said in Sect. 10.2.2 that parallelizing the composite trapezoidal rule can be done by two approaches. In the first approach the $n-1$ inner points are divided among P homogeneous processors using (10.8) and (10.9). In the second approach the integral domain $[a,b]$ is divided into P segments, so that each processor applies the numerical integration rule on its assigned subdomain $[X_p, X_{p+1}]$. Derive the formula for calculating X_p, so that the n intervals are fairly divided among the processors. Also find out the exact difference (summed over all P processors) in the numbers of function evaluations and floating-point arithmetic operations between the two approaches. ◇

Exercise 10.5. Derive the parallel algorithm of the composite Simpson's rule (6.4) using two approaches that are similar to the case of the composite trapezoidal rule. ◇

Exercise 10.6. If $S^\star$ is a desired speedup value, how can we choose the number of processors P according to Amdahl's law (10.17)? Prove also that, if $\alpha > 0$, the parallel efficiency is a monotonically decreasing function of P as the result of Amdahl's law. If $\eta^\star$ is the acceptable threshold of parallel efficiency, what can be the maximum value of P? ◇

Exercise 10.7. We recall that Gustafson–Barsis's law (10.18) uses α_P to denote the fraction of serial computing time in $T(P)$. What is the corresponding fraction α of non-parallelizable computing time in $T(1)$? If we know that the current speedup is $S(P) = S_1$ associated with $T(P)$ and the present value of α_P. By how much should the parallelizable computations be increased such that we can achieve an improved speedup S_2? ◇

Exercise 10.8. The so-called "Ping-Pong" test is a well-used technique for measuring the values of latency τ and bandwidth $1/\beta$, which are used in (10.19). More specifically, a pair of processors keeps exchanging a message of length L_1 several times, using the `MPI_Send` and `MPI_Recv` commands. The measured time is then divided by twice the number of repetitions that the message was bounced between the two processors, giving rise to the actual cost of $t_C(L_1)$. Thereafter, the Ping-Pong test is repeated for different values of L, resulting in measurements of $t_C(L_2)$, $t_C(L_3)$, and so on. Write an MPI program that implements the above Ping-Pong test, and then uses the method of least squares to estimate the values of τ and $1/\beta$. ◇

Exercise 10.9. Implement a serial code that performs a matrix-vector multiplication, where the matrix is dense and of dimension $n \times n$, and the vector is of length n. Parallelize the code using both OpenMP and MPI and compare the obtained speedup results. ◇

Exercise 10.10. Carry out the performance analysis of Algorithm 10.1 in the same fashion as in Example 10.4. The one-to-one message transfer cost model (10.19) should be used in the analysis, while the actual values of τ and β are obtained from Exercise 10.8. Then, use the MPI program as described in Example 10.9 to verify the actual speedup results. Based on the time measurements, can you quantify how expensive it is to update one inner point $u_i^{\ell+1}$, relative to the value of τ? ◇

10.5 Project: Parallel Solution of a 2D Diffusion Equation

Let us consider the following scaled diffusion equation in two space dimensions:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + f(x,y,t), \quad (x,y) \in (0,1) \times (0,1), \ \ t > 0. \tag{10.22}$$

For the initial condition we have $u(x,y,0) = I(x,y)$, and the boundary conditions are as follows:

$$\begin{aligned} u(1,y,t) &= D_1(y,t), \quad t > 0,\\ u(x,1,t) &= D_2(x,t), \quad t > 0, \end{aligned}$$

$$\begin{aligned} \frac{\partial}{\partial x}u(0,y,t) &= N_3(y,t), \quad t > 0,\\ \frac{\partial}{\partial y}u(x,0,t) &= N_4(x,t), \quad t > 0. \end{aligned}$$

Following Sect. 7.4, we will use an explicit method based on finite differences for solving this two-dimensional diffusion equation. First, we introduce a uniform spatial mesh ($x_i = i\Delta x, y_j = j\Delta y$), with $0 \le i \le n_x$, $0 \le j \le n_y$ and $\Delta x = 1/n_x$, $\Delta y = 1/n_y$. Then, the approximate solution of u will be sought on the mesh points and at discrete time levels $t_\ell = \ell\Delta t$, $\ell > 0$.

(a) Show that an explicit numerical scheme for solving (10.22) on the inner mesh points at $t = t_{\ell+1}$ is of the following form:

$$u_{i,j}^{\ell+1} = \alpha u_{i,j}^{\ell} + \beta\left(u_{i-1,j}^{\ell} + u_{i+1,j}^{\ell}\right) + \gamma\left(u_{i,j-1}^{\ell} + u_{i,j+1}^{\ell}\right) + \Delta t f_{i,j}^{\ell}\,, \tag{10.23}$$

where $1 \le i \le n_x - 1$, $1 \le j \le n_y - 1$ and

$$\alpha = 1 - 2\left(\frac{\Delta t}{\Delta x^2} + \frac{\Delta t}{\Delta y^2}\right), \quad \beta = \frac{\Delta t}{\Delta x^2}, \quad \gamma = \frac{\Delta t}{\Delta y^2}.$$

(b) Derive the formulas for computing the numerical solutions on the four boundaries, following the discussions in Sects. 7.4.2 and 7.4.3.

(c) Implement the above explicit numerical scheme as a serial program in the C programming language, where the main computation given by (10.23) is implemented as a double-layered `for`-loop. For each of the four boundaries, a single-layered `for`-loop needs to be implemented according to (b).

(d) Parallelize the serial program using the OpenMP directive `#pragma omp for`. What are the obtained speedup results?

(e) Propose a two-dimensional domain decomposition for dividing the inner mesh points into $P_x \times P_y$ subdomains. More specifically, you should find a mapping $(i, j) \to (p_x, p_y)$ that can assign each inner mesh point (i, j) to a unique subdomain with id $(p_x, p_y) \in [0, P_x - 1] \times [0, P_y - 1]$.

(f) On each subdomain, the assigned mesh points should be expanded with one layer of additional points, which either lie on the actual physical boundaries or work as ghost points toward the neighboring subdomains. For a subdomain with id (p_x, p_y), which values of the local $u^{\ell+1}$ solution should be sent to which neighboring subdomains?

(g) Implement a new parallel program using MPI based on the above domain decomposition. The `MPI_Sendrecv` command can be used to enable inter-subdomain communication.

(h) An approach to hiding the communication overhead is to use so-called *non-blocking* communication commands in MPI. On a parallel system that is capable of carrying out communication tasks at the same time as computations, non-blocking communication calls can be used to initiate the communication (without waiting for its conclusion), followed immediately by computations. The simplest non-blocking MPI commands are `MPI_Isend` and `MPI_Irecv`, which upon return do not imply that the action of sending or receiving a message is completed. The standard `MPI_Send` and `MPI_Recv` commands are thus called blocking communication calls, meaning that the message is safely sent away or received upon the return of the calls.

To enable computation and communication overlap in the present example, we need to divide the local inner points into two sets. More specifically, the outermost layer of the local inner points on each subdomain should be computed first. Then, communication for transferring these new values between the subdomains can be initiated by the non-blocking `MPI_Isend` and `MPI_Irecv` commands. Thereafter, computation on the remaining local inner points can be carried out at the same time as communication is being handled. Finally, the blocking `MPI_Wait` command can be used to ensure that all the needed incoming messages have been received.

Construct a new MPI implementation based the above strategy of overlapping computation and communication.