13

Parallelization

Recent trends in computer hardware have made modern computers parallel computers; that is, the power of today's computers depends on the number of processors.[1] In order to use these powerful machines, we must split the algorithmic tasks into pieces and assign them to a large number of processors. In many cases, this requires nontrivial modifications of the algorithm. In this chapter, we discuss several basic concepts about parallelizing an algorithm and illustrate them in the context of loop/cluster identification.

13.1 Parallel architectures

The key issue in parallel computation is the distribution of computer memory. In other words, how much memory is available and at what access speed. Memories are organized in a hierarchical structure: L1 cache, L2 cache, main memory, and so on, all with different access speeds. The access speed also depends on the physical distance between the computing unit and the memory block.[2]

Discussing how to fine-tune computer programs, taking all machine details into account, is clearly beyond the scope of this book. Therefore, in the following, we focus our discussion of parallel computers on two common types of architectures (Fig. 13.1): shared memory and distributed memory. In either case, we assume that the parallel computer has N_p processors.[3]

In most parallel computers available today, each local memory block is directly accessible by only a small number of processors in the local processor block which is physically closest to it. To access a remote block of memory, a processor

[1] In 2016, it is of the order of 10^6 for the most powerful machines.

[2] If a computer operates with the clock rate of 1 GHz, a photon can travel only 30 cm in a clock cycle. Therefore, if two processors are separated by 10 meters, which is not unrealistic for big parallel machines, they have to wait at least 30 clock cycles for a message to travel from one to the other, no matter what technology is used.

[3] Here a "processor" means the smallest computing unit. In the case where a central processing unit (CPU) consists of multiple processors, that is, *cores*, our use of "processor" means a core rather than a CPU.

https://doi.org/10.1017/CBO9780511902581.014 Published online by Cambridge University Press

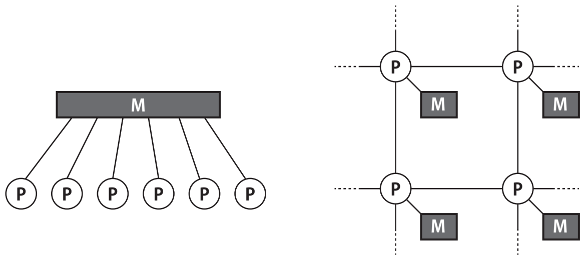

Figure 13.1 The shared-memory model (left) and the distributed-memory model (right). Circles with “P” represent processors, and rectangles with “M” the memory.

must communicate with another processor that has direct access to this block. In some sense, the shared-memory architecture is a model for a local processor block. Alternatively, it can be regarded as a model of the whole computer system in which the communication cost is negligible. In this model, we do not have to account for the process of communicating between the processors. We simply assume that they are all reading and writing to the same block of memory, and each processor immediately knows when something is written to memory by another. The distributed-memory architecture, on the other hand, is a model in which every processor monopolizes the access to the block of memory that it possesses, and for a processor to get the information written on another processor’s memory, the owner of the information must explicitly send it the content of this memory. Because today’s high-performance computers have aspects of both architectures, two different parallelization approaches might be used at the same time, a process called *hybrid* parallelization.

13.2 Single-spin update on a shared-memory computer

Let us start with a simple problem: parallelizing the single-spin update algorithm for an Ising chain (Chapter 4.2) on a shared-memory computer. As for the size of the memory, we assume that the available memory is infinitely large, which is not unrealistic because Monte Carlo simulations are often computation limited, not memory limited. We may thus assume that the information about the spin configuration of the whole lattice fits into the shared memory.

The most natural way to split the task is to divide the set of spins into N_p subsets $\Omega_1, \Omega_2, \ldots, \Omega_{N_p}$ and assign a subset to each processor (Algorithm 42). Let M be the size of Ω_i, that is, $M \equiv N/N_p$, where N is the total number of spins. The p-th processor performs Algorithm 6 with some fixed order of the spins in Ω_p instead of random picking. The probability of the new spin state relative to the state with the chosen spin s_i satisfies the detailed balance condition.

Algorithm 42 Single-spin update of the Ising model: the p-th processor's task on a shared-memory computer. (Error prone on a parallel computer.)

```
Input: Spin configuration in block $p$, $\{s_i = \pm 1\}$ ($i \in \Omega_p$), and values of
    neighboring spins.
  for $k = 1, 2, \ldots, M\,(\equiv N/N_p)$ do
      Let $i$ be the $k$-th member of $\Omega_p$ ;
      Change $s_i$ to the new state chosen probabilistically ;
  end for
  return the updated spin configuration in block $p$.
```

Algorithm 42 yields correct results (up to the statistical error) as long as the operations do not interfere with each other. However, since this independency condition does not hold in general, naive parallelization may yield erroneous results. When a processor, say, P_1, tries to update spin s_i in its domain, it needs the information about the spins surrounding s_i. However, some neighboring spin, say, s_j, may belong to a different processor, say, P_2. The processor P_1 obtains the information about s_j at the beginning of the update cycle described in Algorithm 42. The problem is that this information is not updated until the cycle completes on all processors and they communicate with each other. Therefore, the value of s_j used by P_1 for updating s_i may have been changed by P_2 when P_1 starts to work on s_i. As a simple example, let us consider a small system with only two ferromagnetically coupled Ising spins. Suppose we assign one spin to each of the two processors, and apply Algorithm 42 with the Metropolis update as in Algorithm 6, which always flips the spins whenever this flip lowers the energy. What will happen? Once these spins become antiparallel at some point of time, they will remain antiparallel forever![4]

To avoid this problem,[5] we must carefully order the examination of spins in Ω_p and ensure that the processors are appropriately synchronized. In the case of the one-dimensional Ising model, we replace the somewhat generic algorithm just described by Algorithm 43, illustrated in Fig. 13.2. This new procedure ensures that neighboring spins are never updated simultaneously. In the figure, the update order of the solid circles covered by the same processor might affect the efficiency, but it should not cause systematic errors. At the end of each step, we synchronize all processors to avoid mixing open and solid circles; that is, we let the proces-

[4] At the beginning of the cycle, each processor obtains the information that the neighboring spin is antiparallel, and therefore judges that flipping its own spin will lower the energy. Because both processors flip their spins, the spin configuration will stay antiparallel.

[5] This particular type of problem is caused by what is called a *race condition*, as the result of the operation is determined by a race between two or more processors.

Algorithm 43 Single-spin update of the one-dimensional Ising model: the p-th processor's task on a shared-memory computer. (Free from error even when $M > 1$.)

```
Input: Spin configuration in block $p$, $s_i = \pm 1$ ($M(p-1)+1 \le i \le Mp$), and
  value of neighboring spins.
  for $k = 1, 2, 3, \ldots, M/2$ do
    $i \leftarrow M(p-1)+k$ ;
    Update $s_i$ ;    ▷ e.g., by the Metropolis algorithm
  end for
  Synchronize ;    ▷ Wait for the other processors to finish their tasks
  for $k = M/2+1, M/2+2, \ldots, M$ do
    $i \leftarrow M(p-1)+k$ ;
    Update $s_i$ ;
  end for
  Synchronize ;
  return the updated spin configuration in block $p$.
```

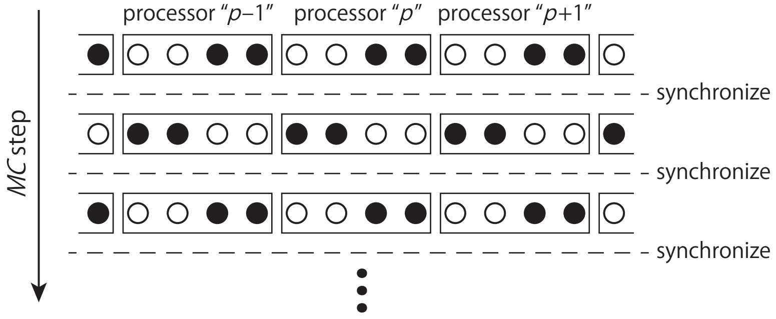

Figure 13.2 The parallelized single-spin update. Each processor takes care of four spins in this example, that is, $M = 4$. In each step, only the spins represented by open circles are updated so that each processor can carry out its task independently. At the end of each step, all the processors are synchronized, and in the case of the distributed memory machine, the value of the boundary spin is passed to the neighboring processor.

sors wait until all reach the same point in the algorithm.[6] Extensions to higher dimensions are obvious: We split each processor's domain into a few subdomains (domain decomposition) in such a way that two neighboring spins are not examined simultaneously.

[6] While synchronization resolves the issue in the present example, the use of this procedure becomes less and less efficient as the computer has more and more processors.

Algorithm 44 Single-spin update of the one-dimensional Ising model: the p-th processor's task on a distributed-memory computer.

```
Input: Spin configuration in block $p$, $s_i = \pm 1$ $(M(p-1)+1 \le i \le Mp)$.
  Send $s_{Mp}$ to the $(p+1)$-th processor and receive $s_{M(p-1)}$ from the $(p-1)$-th processor ;
  for $k = 1, 2, 3, \dots, M/2$ do
    $i \leftarrow M(p-1)+k$ ;
    Update $s_i$ ;
  end for
  Synchronize ;
  Send $s_{M(p-1)+1}$ to the $(p-1)$-th processor and receive $s_{Mp+1}$ from the $(p+1)$-th processor ;
  for $k = M/2+1, M/2+2, \dots, M$ do
    $i \leftarrow M(p-1)+k$ ;
    Update $s_i$ ;
  end for
  Synchronize ;
  return the updated spin configuration in block $p$.
```

13.3 Single-spin update on a distributed-memory computer

In the distributed-memory computing model, a processor might not have all the information necessary for executing its task. Therefore, it must ask other processors, usually its neighbors, for the missing information. In the Ising model example, a processor lacks necessary information when it tries to update a spin on the boundary of its domain. Therefore, it must issue an inquiry to its neighbor before it can continue with its task. The same is true for all other processors. As is evident from Algorithm 44, these inquiries generate frequent exchanges of information.

Accordingly, we now have to pay special attention to the cost of the communication. Roughly speaking, if the time needed for interprocessor communication is shorter than the time required for each processor to execute its operations, we say that the process is well balanced, because we will benefit from increasing the number of processors as long as this condition remains true.[7]

Let us consider the performance of the computation when the problem size (that is, the number of spins N in the present case) increases while the number of

[7] In such a case, we can even effectively eliminate the communication time by so-called *communication-latency hiding*. Namely, by exploiting the fact that the calculating unit and the communicating unit can work simultaneously and independently from each other, a processor can update spins while sending/receiving the information to/from other processors.

processors remains fixed, making the size of the subsystem assigned to each processor proportional to the problem size. In the Ising chain example, the communication time is always $\mathcal{O}(1)$ as each processor sends and receives only two spins. The number of operations executed by each processor, on the other hand, is proportional to $M = N/N_p$. Therefore, the ratio of the interprocessor communication time to the operation time within each processor is $R = \mathcal{O}(M^{-1})$. It is easy to generalize this estimate to higher dimensions. Now, M is the length scale of each processor's domain, which is a d-dimensional hypercube, for example. Then, each processor must send and receive the values of the spins on the boundary, whose area scales as M^{d-1}. The communication time is therefore proportional to M^{d-1}. The operation time is again proportional to the number of spins covered by a processor, which is M^d. Therefore, the communication to operation ratio is

$$R = \mathcal{O}(M^{-1})$$

regardless of the dimension. From this we can conclude that the interprocessor communication time is negligible, as long as the decomposition is sufficiently coarse.[8] If, on the contrary, the size of each domain is too small, the relative communication cost can be large and we cannot benefit from increasing the number of processors. Simply put, *large computers are efficient only for large problems*. This statement also applies to other parallelized algorithms based on domain decomposition.

13.4 Loop/cluster update and union-find algorithm

We now consider the loop/cluster algorithm. We recall that it consists of two steps: graph assignment and cluster flips. Our aim is to split space-time into N_p regions and assign a processor to each region. Each processor therefore has the complete information about the region assigned to it, such as the space-time positions of the vertices, the local state of a given segment delimited by kinks, and the state at the boundary of the region.

We first consider the parallelization of the graph assignment step. If the space-time decomposition is only temporal and no boundary separates two regions spatially, interprocessor communication is absent because both ends of a graph element, which is local in time, always belong to the same region. On the other hand, when the decomposition is not purely temporal, we may have to place a graph element across the boundary between two regions, say, A and B. In this case, we must transfer information about the local state near the boundary of region B to the processor in charge of A. The A processor then computes the temporal positions

[8] Whether we can make the space decomposition sufficiently coarse in this sense, of course, depends on whether the original problem size is large enough for the number of processors available.

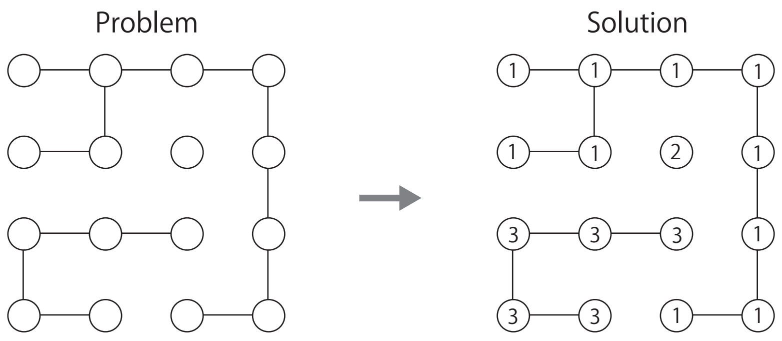

Figure 13.3 The cluster identification problem.

of the new vertices and sends this information back to the B processor. Therefore, the amount of interprocessor communication is proportional to the number of the vertices between A and B, which in turn is proportional to the length of the spatial boundary separating them.

Now we consider the parallelization of the loop/cluster spin flips, which is much less straightforward because of the nonlocal nature of the cluster identification. Because we will use graph manipulation algorithms for this identification, we adopt a graph theoretical terminology. Since the essence of the difficulty does not depend on whether it is a classical model or a quantum model, let us consider the classical model for the sake of clarity of the description. In this condition, a vertex is a point and an edge is a line connecting two points.

To update the spins we need to identify clusters. To be more specific, our task is to obtain an array, say, p, that defines a mapping from the vertex number to the cluster number. For example, $p(3) = 5$ means that the vertex 3 belongs to the cluster 5. The cluster number is such that $p(v_a) = p(v_b)$ if and only if the vertices v_a and v_b are connected by a sequence of edges (Fig. 13.3). In the cluster algorithm for the Ising model, once we obtain such an array, we update the spin variables simply by generating one-bit random numbers, $b(1), b(2), \ldots, b(n_c)$, that represent the spin values of the n_c clusters and assign $b(p(v))$ to each vertex v as its new spin value.

We first discuss the serial version of this union-find algorithm (Hoshen and Kopelman, 1976). To construct the array $p(v)$ for a given list of edges, we grow a forest of trees[9] such that each tree corresponds to a cluster to be identified. Every component of the tree (the root, a branching point, or a leaf) corresponds to a vertex. During the grafting operations we discuss below, $p(v)$ is always a pointer to the "parent" vertex, that is, the vertex at the next step along the path toward the root. We suppose the vertices are sequentially numbered, and their number is fixed.

[9] If a graph has N_v vertices, it is a tree if and only if it has $N_v - 1$ edges. For a tree the path between any two vertices is unique. In other words, a tree does not contain any loops.

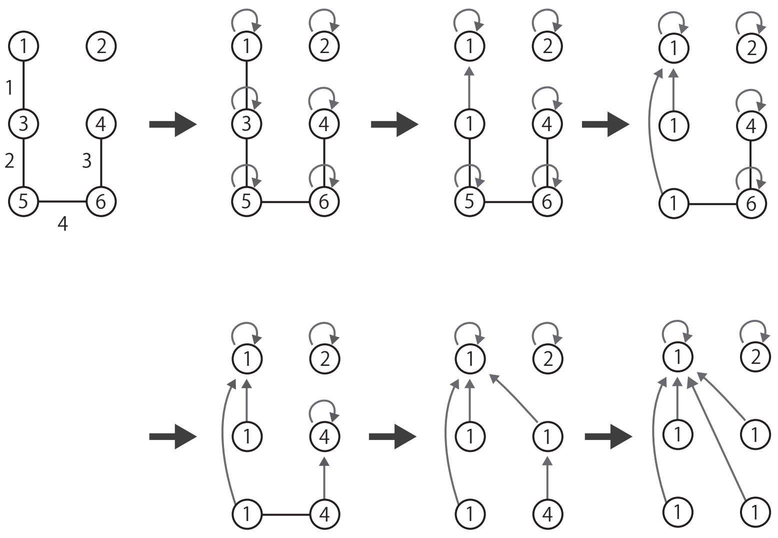

Figure 13.4 The “grafting” algorithm for the union-find problem (Algorithm 45). The numbers in the circles are $p(v)$ in the algorithm. The edges are numbered in the order in which they are examined.

In the initial state, every vertex is the root of a tree (second diagram in Fig. 13.4). In other words, we have N trees of height zero. We grow the trees by feeding them information about edges, one by one.

Suppose an edge e, the first in the list of unchecked edges, connects two vertices v_a and v_b, that is, $e = (v_a, v_b)$. Then we start from v_a and trace the branches until we reach the root of the tree, say, r_a. We do the same starting from v_b, reaching the root r_b. If r_a and r_b differ, v_a and v_b are on different trees. In this case, one tree must be grafted onto the other. If the vertex number of r_a is smaller than that of r_b, we graft tree b onto tree a as a branch connecting b directly to the root of a. Otherwise, we graft tree a onto tree b in the same manner. As is obvious from this procedure, after examining all edges, there must be a one-to-one correspondence between the trees and the clusters. In other words, any two vertices that are connected by a sequence of edges belong to the same tree. Once all trees are completed, we start from each vertex and trace the branches to the root and then assign the root’s vertex number to the starting vertex as its cluster number. This procedure is summarized in Algorithm 45.

The theoretical upper bound for the number of operations required to execute the procedures of Algorithm 45 is $\mathcal{O}(N_E^2)$, where N_E is the number of edges. The worst case is an edge list of the type $(100, 99)$, $(99, 98)$, $(98, 97)$, $\ldots$, $(2, 1)$. Here the order is important: If the edges are examined in this order, the content of the array $p(v)$ at the end of the grafting phase is $p(1) = 1, p(2) = 1, p(3) = 2, p(4) = 3, \ldots,$ $p(100) = 99$. Then, in the height reduction phase, we need to trace this sequence

Algorithm 45 Hoshen-Kopelman union-find algorithm (serial).

```
Input: List of edges $\{(v_a, v_b)\}$
  for every vertex $v$ do
    $p(v) \leftarrow v$ ;    ▷ Initially, every vertex points to itself.
  end for
  for every edge $e = (v_a, v_b)$ do
    $r_a \leftarrow \mathrm{Root}(v_a)$ ;
    $r_b \leftarrow \mathrm{Root}(v_b)$ ;
    if $r_a < r_b$ then
      $p(r_b) \leftarrow r_a$ ;
    else
      $p(r_a) \leftarrow r_b$ ;
    end if
  end for
  for every $v$ do
    $r \leftarrow \mathrm{Root}(v)$ ;
    $p(v) \leftarrow r$ ;
  end for
  return $p$.

  function Root($v$)
  $u \leftarrow v$ ;
  while $p(u) \neq u$ do
    $u \leftarrow p(u)$ ;
  end while
  return $u$.
```

to its root. Depending on the order in which the vertices are examined, the total number of operations varies. The worst case occurs when the vertices are examined in decending order, that is, 100, 99, ..., 2, 1. When we start from vertex 100, we need 99 replacement operations to find that the root of this vertex is 1. Then, $p(100)$ is set to 1. Next we start from vertex 99, and after 98 replacements we finish the task. The computational complexity for these operations will be $\mathcal{O}(N_V N_E)$ with N_V being the number of vertices. Because $O(N_V) = O(N_E)$ in most applications in condensed matter physics, we obtain a computational complexity of $\mathcal{O}(N_E^2)$.[10] This cost may completely invalidate the advantage of the loop/cluster procedure.

[10] When we examine the vertices in ascending order, we can always reach the root within two steps, which reduces the computational complexity to $\mathcal{O}(N_E)$. However, in a parallel implementation it may not be possible to control the order of the vertices.

Algorithm 46 Improved union-find algorithm (serial).

```
Input: Edge list $\{(v_a, v_b)\}$.
  for every vertex $v$ do
    $p(v) \leftarrow v$ ;
    $w(v) \leftarrow 1$ ;
  end for
  for every edge $e = (v_a, v_b)$ do
    $r_a \leftarrow \text{Root}(v_a)$ ;
    $r_b \leftarrow \text{Root}(v_b)$ ;
    if $w(r_a) \geq w(r_b)$ then
      $r_0 \leftarrow r_a$ ;
      $r_1 \leftarrow r_b$ ;
    else
      $r_0 \leftarrow r_b$ ;
      $r_1 \leftarrow r_a$ ;
    end if
    $w(r_0) \leftarrow w(r_0) + w(r_1)$ ;    ▷ Updating the weights
    $p(r_1) \leftarrow r_0$ ;    ▷ The small tree grafted on the large
    $\text{Compress}(v_a, r_0)$ ;    ▷ Compressing one tree
    $\text{Compress}(v_b, r_0)$ ;    ▷ Compressing the other
  end for
  for every $v$ do
    $r \leftarrow \text{Root}(v)$ ;
    $\text{Compress}(v, r)$ ;
  end for
  return $p$.

  function Root($v$)
  $u \leftarrow v$ ;
  while $p(u) \neq u$ do
    $u \leftarrow p(u)$ ;
  end while
  return $u$.

  function Compress($v$,$r$)
  $u \leftarrow v$ ;
  while $p(u) \neq u$ do
    $s \leftarrow p(u)$ ;
    $p(u) \leftarrow r$ ;
    $u \leftarrow s$ ;
  end while
  return
```

To reduce the computational cost, we make two improvements (Aho et al., 1983). One is to introduce a weight $w(v)$ for the tree defined as the number of vertices included in the tree, and when grafting always grafts the smaller weighted tree on the larger one. The other is to reduce the height of the trees frequently. The improved algorithm is Algorithm 46. Its complexity is $\mathcal{O}((N_E+N)\lg N)$ (Aho et al., 1983) where $\lg X$ is the inverse Ackerman function, which increases very slowly. Roughly speaking, it is the number of times we have to take $\log_2$ of X to obtain 1, for example, $\lg 1 = 0$, $\lg 2 = 1$, $\lg 2^2 = 2$, $\lg 2^{2^2} = 3$, $\lg 2^{2^{2^2}} = 4$, etc. For practical purposes, we can regard it as a constant.

13.5 Union-find algorithm for shared-memory computers

We now discuss the shared-memory parallelization of the union-find algorithm using the prescription proposed in Todo et al. (2012). It breaks the set of edges into smaller subsets and assigns a subset to each processor. First, let us imagine that each processor simply performs its task with Algorithm 45 or Algorithm 46. Obviously, we do not have control of the order in which the edges are examined. At first glance, this does not seem to matter, since the final outcome (clustering) should not depend on the order as long as all edges are taken into account.[11] It is, however, not trivial to ensure that all edges are properly accounted for.

To see this, let us consider a graph consisting of six vertices. Suppose the roots of the vertices 4, 5, and 6 are 1, 2, and 3 (Fig. 13.5) and that processors A and B are about to handle edges $(4, 6)$ and $(5, 6)$, respectively. We might expect the following sequence of events:

(A1) Processor A finds the root of 6 to be 3.
(A2) Processor A finds the root of 4 to be 1.
(A3) Processor A changes $p(3)$ to 1.
(B1) Processor B finds the root of 6 to be 1.
(B2) Processor B finds the root of 5 to be 2.
(B3) Processor B changes $p(2)$ to 1.

The result is a single cluster, as illustrated in the left-hand side of Fig. 13.5. However, the order of the events need not necessarily be this. The following is equally likely to happen:

[11] The order of edges may affect the shape of the tree in the final state, but any two vertices must belong to the same tree as long as all the edges are properly examined, as in Fig. 13.5.

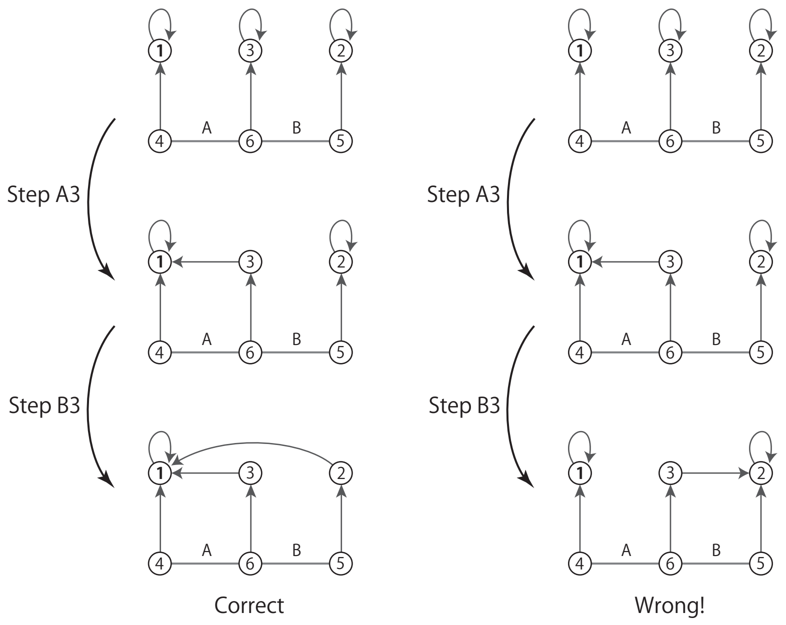

Figure 13.5 Nonthread-safe operations in the union-find algorithm.

(A1) Processor A finds the root of 6 to be 3.
(B1) Processor B finds the root of 6 to be 3.
(A2) Processor A finds the root of 4 to be 1.
(B2) Processor B finds the root of 5 to be 2.
(A3) Processor A changes $p(3)$ to 1.
(B3) Processor B changes $p(3)$ to 2.

Obviously, the information about the edge $(4, 6)$ is lost, as shown in the right-hand side of Fig. 13.5. It happens since, at (B3) in the latter process, processor B uses obsolete information about the tree with vertex 6 when it grafts the tree. If no other edge reconnects the two clusters later, the result will be wrong.

To prevent such an information loss, whenever a processor tries to graft a root r_a to another root r_b, it must first lock them so that no other processor can change $p(r_a)$ or $p(r_b)$ before the first processor finishes grafting. One way of modifying the algorithm is illustrated in Algorithm 47, in which a locked vertex v is encoded by a negative value of $p(v)$. The function "Lock" must be implemented so that the operations in the if-clause beginning with "**if** $p(v) = v$ **then**" are executed as a single step. If not, an interruption from other processors might cause $p(v)$ in the first operation (comparison, $p(v) = v$) and $p(v)$ in the second operation (substitution, $p(v) \leftarrow -v$) to differ from each other, thus leading to an error. While the strategy to avoid such an interruption may depend on the machine, we can do it with a

Algorithm 47 The task of a processor in union-find on a shared-memory computer. The omitted parts are the same as in Algorithm 46.

```
Input: Sublist of the edge list $E_p \equiv \{(v_a, v_b)\}$.
  ··· the initialization part omitted ···
  for every edge $e \in E_p$ do
    $r_a \leftarrow$ Root($v_a$) ;
    $r_b \leftarrow$ Root($v_b$) ;
    loop
      $q_a \leftarrow$ Lock($r_a$) ;
      $q_b \leftarrow$ Lock($r_b$) ;
      if $q_a$ = "success" and $q_b$ = "success" then
        Break ;
      else
        if $q_a$ = "success" then
          Unlock($r_a$) ;
        end if
        if $q_b$ = "success" then
          Unlock($r_b$) ;
        end if
        $r_a \leftarrow$ Root($r_a$) ;
        $r_b \leftarrow$ Root($r_b$) ;
      end if
    end loop
    ··· the grafting part omitted ···
    Unlock($r_a$) ;
    Unlock($r_b$) ;
  end for
  ··· the tree-compression part omitted ···
  return $p$

  function Lock($v$)
    ▷ The following if-clause must be done as a single operation.
  if $p(v) = v$ then
    $p(v) \leftarrow -v$ ;
    return "success" ;    ▷ Lock succeeded
  end if
  return "failure" ;    ▷ Lock failed

  function Unlock($v$)
  $p(v) \leftarrow -p(v)$ ;
  return
```

compare-and-swap atomic (noninterruptable) instruction that compares and swaps in a single instruction (Todo et al., 2012).

13.6 Union-find algorithm for distributed-memory computers

We now discuss the parallelization of the union-find algorithm for a distributed-memory computer. Whereas most graph manipulation algorithms assume no particular spatial structure, the finiteness of the space dimension is crucial to the algorithm discussed below. In this sense, the algorithm is not for all union-find applications. We slightly generalize the ideas of Todo et al. (2012), and also describe the identification of clusters, instead of loops.

In this algorithm, space-time is split into domains and a processor is assigned to each space-time domain. A processor's task is to obtain the cluster number, $p(v)$, for all v in its domain, say, Ω_0. It starts the cluster identification using only the information of edges in Ω_0 by one of the algorithms discussed above. When this is done, the cluster identification of the domain Ω_0 is complete within Ω_0. (We say "the cluster identification of the domain A is complete within B" when the cluster numbers assigned to all the vertices in A would be correct if there were no other edges than those within region B.) The processor then takes into account edges within a neighboring region, say, Ω_1, and also edges on the boundary between Ω_0 and Ω_1, and updates the cluster number table. The result is the cluster number table of Ω_0 that is complete within $\Omega_0 \cup \Omega_1$. In this way, the processor keeps doubling the domain for edges until the cluster number table of Ω_0 is complete within Ω_{whole}, the whole space.

Each processor possesses information about the vertices in its initial domain, the vertices on the boundary of its current domain, and the root of trees that include one or more of the boundary vertices. To be more specific, the local memory associated with a processor stores $p(v)$ and $w(v)$ for the vertices in $\Omega_0 \cup \partial\Omega \cup \mathcal{R}(\partial\Omega)$, where Ω_0 is the processor's original domain, Ω and $\partial\Omega$ are the current domain and its boundary, and $\mathcal{R}(\partial\Omega)$ is the set of the roots of the trees that have nonvanishing overlap with $\partial\Omega$. The information on the edges that connect the current domain to others is also stored in memory, as well as information on the edges within the domain. The latter, however, is necessary only for the first iteration. Each processor uses these pieces of information in Algorithms 45, 46, or 47.

Initially, each processor uses the information of edges and vertices within Ω_0 to identify clusters by applying Algorithm 46.[12] However, this cluster identification is clearly incomplete, since two vertices that are disjoint within the domain might con-

[12] If each processor has many cores and is actually a shared-memory parallel machine itself, we execute hybrid parallelization by letting each processor use Algorithm 47 instead of the algorithm for the serial computer.

nect via a path that runs outside of the domain. To account for such outside paths, the processor, referred to as P_0 hereafter, must obtain information from others. To do this, it picks one of its nearest-neighbor processors, say, P_1, and asks for the information. It does not, however, need all of the information that P_1 has. P_0's task is to find the cluster numbers of vertices only in Ω_0, and any outside path must run through one of the boundary vertices. Therefore, for P_0, it suffices to obtain the pointers $p(v)$ and the weights $w(v)$ of vertices on the domain boundary between P_0's domain and P_1's and those of their roots. Once P_0 has obtained this information, it examines the edges that connect the two domains and grafts trees by the prescription of Algorithm 46. When finished, the cluster identification of Ω_0 is complete within $\Omega_0 \cup \Omega_1$ where Ω_1 is P_1's original domain. We repeat this procedure in a hierarchical fashion until the domain covered by each processor becomes the whole system.

There are several ways of organizing this hierarchy. Here, we consider one represented by an "escargot" pattern (Fig. 13.6). A processor, P_0, first completes the cluster identification within Ω_0 (the darkest square in the figure). Likewise, all the other processors do their job. Then, P_0 asks its nearest neighbor, P_1, in the x-direction for the cluster numbers of the boundary vertices. Using this information, P_0 completes its cluster identification within the domain whose area is now twice as large (the union of the darkest and the second darkest square). Again, all the others do the same. Next, P_0 asks for the information from its nearest neighbor in the y-direction, P_2. This time, the neighbor has the boundary information of the doubled domain (the horizontal rectangle above the darkest square). As a result, P_0

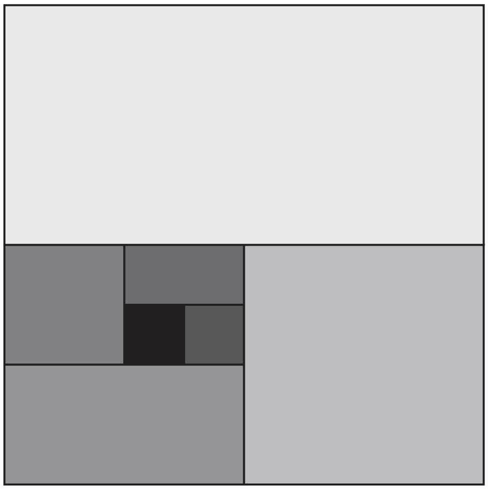

Figure 13.6 The domain covered by a processor. The darkest and smallest square represents the domain that it covers at the beginning. As the operation proceeds by one step, the domain is doubled, and the cluster identification is completed within this domain.

completes the cluster identification for a square twice as large as the initial square in both directions. In the third step, P_0 gets information from its neighbor, P_3, located at position $(-2, 0)$ relative to the original square in units of the original square size. After this, it repeats this procedure with the other processors located at $(0, -2)$, $(4, 0)$, $(0, 4)$, $(-8, 0)$, $(0, -8)$, and so on. In general, in d dimensions, this sequence would be $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_d, -2\boldsymbol{e}_1, -2\boldsymbol{e}_2, \ldots, -2\boldsymbol{e}_d, 4\boldsymbol{e}_1, 4\boldsymbol{e}_2, \ldots, 4\boldsymbol{e}_d, \ldots$ with $\boldsymbol{e}_\mu$ being the unit lattice vector in the μ-th direction.

In this “escargot” arrangement, if the clusters to be identified are small, a processor completes its task before its domain covers the whole system, because as soon as the processor covers all the clusters overlapping with Ω_0, the cluster numbers of vertices in Ω_0 are assigned correctly, and no further change occurs even if the processor keeps searching. This advantage can be significant when the typical cluster size is much smaller than the whole system, which is often the case in the disordered state. Specifically, if the maximum cluster size is ξ, all processors stop as soon as the size of the domain reaches 4ξ.

13.7 Back to the future

In Chapter 1 we recounted that in what is generally regarded as the first publication to use the phrase “Monte Carlo method,” Metropolis and Ulam (1949) described the philosophy of this new method and noted its ability to do what we today say is “to break the curse of dimensionality” that often limits the effective numerical solution of partial differential equations for complex problems. Generally unnoticed in this landmark publication is a remark that the Monte Carlo method is ideal for parallel computing, an observation made decades before other scientists started to think about parallel computers and only two years after the now standard von Neumann architecture was proposed. The remark likely concerns the simplest, but unarguably the most efficient parallelization, the natural parallelization whereby we give each processor its own input (at least its own random number seed) and the same executable. Each processor runs the executable independently, and one collects the individual outputs when all processors are finished. The executable may be any of the Monte Carlo algorithms discussed in this book: single-spin update, cluster update, worm update, determinant method, variational Monte Carlo, and the rest. We omitted the discussion of the natural parallelization in this chapter, mainly because it is so natural. However, we emphasize here that natural parallelization is one of the big advantages of the Monte Carlo technique.[13]

[13] Instead of being called “naturally parallel,” Monte Carlo methods are often called “embarrassingly parallel” because they are an embarrassment to methods such as molecular dynamics or finite element methods that are not naturally parallel.

Ulam and Metropolis were speaking about the Monte Carlo method that Ulam and von Neumann proposed just a few years prior for simulating the diffusion-collision transport of radiation through fissile material. As we also remarked in Chapter 1, Ulam and von Neumann used a Markov chain to predict the distribution of radiation that required sampling from a stationary distribution that is unknown beforehand, and in 1953, Metropolis et al. proposed the Metropolis algorithm, their now famous other form of Monte Carlo that samples from a stationary distribution that is known beforehand. In his recollections about the development of the Metropolis algorithm, Marshall Rosenbluth (2003) relates that they were studying the melting of a solid, and the most natural way to simulate it was to follow the dynamics of the interacting particles under the action of Newton's laws. However, the memory and speed of the computer available to them, Metropolis's MANIAC, were too limited to follow an adequate number of particles for a large number of small time steps, so they had to think of another way. As they were interested only in equilibrium properties, they decided to take advantage of statistical mechanics and ensemble averages instead of following the detailed kinematics. If Metropolis's computer, one of the earliest built with a von Neumann architecture, had been more advanced, they would have invented molecular dynamics instead of the Metropolis algorithm!

Over time, the basic characteristic of Monte Carlo algorithms has not changed as much as the computer hardware on which we use them. Indeed, in the future the Monte Carlo method will still break the curse of dimensionality, will still be naturally parallelizable, and will still require relatively modest core memory. We also know that it usually samples phase space by taking relatively local steps that in many cases promotes processor asynchronization. It needs to pass few messages, and these passings are often local.

If the main obstacle is the statistical noise in the Monte Carlo simulation, and this noise problem is not exponentially hard, the natural parallelization removes it and is the method one should use. However, in some cases, for example, if the computational autocorrelation time is too long to wait for the system to relax to its equilibrium distribution or if the system is too large to store a whole configuration in the memory of a single processor, then we need something that goes beyond natural parallelization.[14] In this chapter, we gave an example of such a nontrivial parallelization of a very important task for a popular class of classical and quantum Monte Carlo algorithms. In particular, the final approach to parallelization of the loop/cluster algorithm discussed in this chapter is a step up from natural parallelization. The parallelization of other types of algorithms is currently an active

[14] Another important example is the Fourier transformation.

research topic (see, e.g., Masaki-Kato et al. (2014) for the parallelization of the worm algorithm).

As we remarked in the text, big computers are efficient only for big problems. The visions for the next generation of big computers are ones with orders of magnitude more processors each being not much faster than today’s processors. Writing parallel codes for these computers is a challenge for computational physics, but the inherent advantages of the Monte Carlo approach, as well as the development of nontrivial parallelization algorithms, poises Monte Carlo for extensive use in the dawn of exascale computing. How we use the Monte Carlo method will evolve. How we parallelize it will change. What will always remain important is identifying suitable problems that can be tackled successfully with improved algorithms or more powerful computing platforms.