Programming Multi-core and Shared Memory Multiprocessors Using OpenMP

3

Chapter Summary
Of many different parallel and distributed systems, multi-core and shared memory multiprocessors are most likely the easiest to program if only the right approach is taken. In this chapter, programming such systems is introduced using OpenMP, a widely used and ever-expanding application programming interface well suited for the implementation of multithreaded programs. It is shown how the combination of properly designed compiler directives and library functions can provide a programming environment where the programmer can focus mostly on the program and algorithms and less on the details of the underlying computer system.

3.1 Shared Memory Programming Model

From the programmer's point of view, a model of a **shared memory multiprocessor** contains a number of independent processors all sharing a single main memory as shown in Fig. 3.1. Each processor can directly access any data location in the main memory and at any time different processors can execute different instructions on different data since each processor is driven by its own control unit. Using *Flynn's taxonomy* of parallel systems, this model is referred to as *MIMD*, i.e., multiple instruction multiple data.

Although of paramount importance, when the parallel system is to be fully utilized to achieve the maximum speedup possible, many details like cache or the internal structure of the main memory are left out of this model to keep it simple and general. Using a simple and general model also simplifies the design and implementation of portable programs that can be optimized for a particular system once the parallel system specifications are known.

Most modern CPUs are **multi-core processors** and, therefore, consist of a number of independent processing units called **cores**. Moreover, these CPUs support (simultaneous) **multithreading** (SMT) so that each core can (almost) simultaneously

R. Trobec et al., *Introduction to Parallel Computing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-98833-7_3

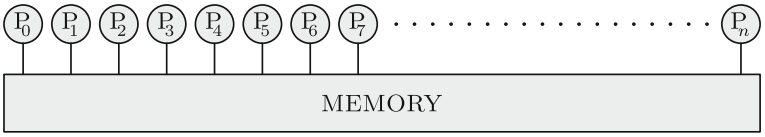

Fig. 3.1 A model of a shared memory multiprocessor

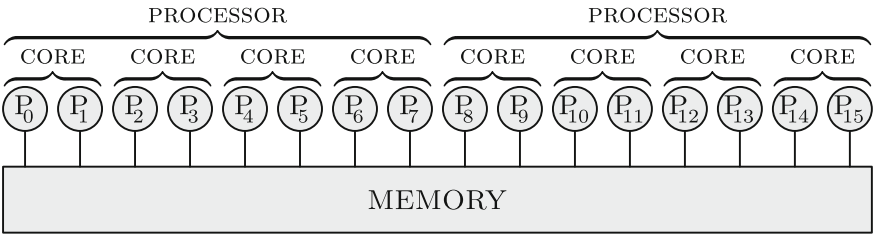

Fig. 3.2 A parallel system with two quad-core CPUs supporting simultaneous multithreading contains 16 logical cores all connected to the same memory

execute multiple independent streams of instructions called **threads**. To a programmer, each core within each processor acts as several **logical cores** each able to run its own program or a thread within a program independently.

Today, mobile, desktop and server CPUs typically contain 2–24 cores and with multithreading support, they can run 4–48 threads simultaneously. For instance, a dual-core mobile Intel i7 processor with hyper-threading (Intel's SMT) consists of 2 (physical) cores and thus provides 4 logical cores. Likewise, a quad-core Intel Xeon processor with hyper-threading provides 8 logical cores and a system with two such CPUs provides 16 logical cores as shown in Fig. 3.2. If the common use of certain resources like bus or cache is set aside, each logical core can execute its own thread independently. Regardless of the physical implementation, a programmer can assume that such system contains 16 logical cores each acting as an individual processor as shown in Fig. 3.1 where $n = 16$.

Apart from multi-core CPUs, *manycore processors* comprising tens or hundreds of physical cores are also available. Intel Xeon Phi, for instance, provides 60–72 physical cores able to run 240–288 threads simultaneously.

The ability of modern systems to execute multiple threads simultaneously using different processors or (logical) cores comes with a price. As individual threads can access any memory location in the main memory and to execute instruction streams independently may result in a **race condition**, i.e., a situation where the result depends on precise timing of read and write accesses to the same location in the main memory. Assume, for instance, that two threads must increase the value stored at the same memory location, one by 1 and the other by 2 so that in the end it is increased by 3. Each thread reads the value, increases it and writes it back. If these three instructions are first performed by one thread, and then by the other, the correct result is produced. But because threads execute instructions independently, the sequences of these three instructions executed by each thread may overlap in time

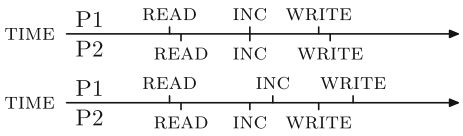

Fig. 3.3 Two examples of a race condition when two threads attempt to increase the value at the same location in the main memory

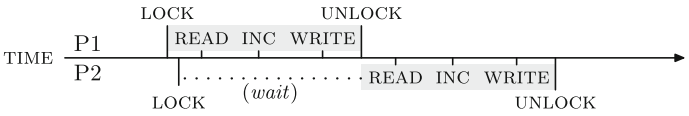

Fig. 3.4 Preventing race conditions as illustrated in Fig. 3.3 using locking

as illustrated in Fig. 3.3. In such situations, the result is both incorrect and undefined: in either case, the value in the memory will be increased by either 1 or 2 but not by 1 and 2.

To avoid the race condition, exclusive access to the shared address in the main memory must be ensured using some mechanism like **locking** using semaphores or **atomic access** using read-modify-write instructions. If locking is used, each thread must lock the access to the shared memory location before modifying it and unlock it afterwards as illustrated in Fig. 3.4. If a thread attempts to lock something that the other thread has already locked, it must wait until the other thread unlocks it. This approach forces one thread to wait but guarantees the correct result.

The peripheral devices are not shown in Figs. 3.1 and 3.2. It is usually assumed that all threads can access all peripheral devices but it is again up to software to resolve which thread can access each device at any given time.

3.2 Using OpenMP to Write Multithreaded Programs

A parallel program running on a shared memory multiprocessor usually consists of multiple threads. The number of threads may vary during program execution but at any time each thread is being executed on one logical core. If there are less threads than logical cores, some logical cores are kept idle and the system is not fully utilized. If there are more threads than logical cores, the operating system applies multitasking among threads running on the same logical cores. During program execution, the operating system may perform *load balancing*, i.e., it may migrate threads from one logical core to another in an attempt to keep all logical cores equally utilized.

A multithreaded program can be written in different programming languages using many different libraries and frameworks. On UNIX, for instance, one can use `pthreads` in almost any decent programming language, but the resulting program is littered with low-level details that the compiler could have taken care of, and is not portable. Hence, it is better to use something dedicated to writing parallel programs.

One such thing is OpenMP, a parallel programming environment best suitable for writing parallel programs that are to be run on shared memory systems. It is not yet another programming language but an add-on to an existing language, usually Fortran or C/C++. In this book, OpenMP atop of C will be used.

The application programming interface (API) of OpenMP is a collection of

- compiler directives,
- supporting functions, and
- shell variables.

OpenMP compiler directives tell the compiler about the parallelism in the source code and provide instructions for generating the parallel code, i.e., the multi-threaded translation of the source code. In C/C++, directives are always expressed as `#pragma`s. Supporting functions enable programmers to exploit and control the parallelism during the execution of a program. Shell variables permit tunning of compiled programs to a particular parallel system.

3.2.1 Compiling and Running an OpenMP Program

To illustrate different kinds of OpenMP API elements, we will start with a simple program in Listing 3.1.

```
#include <stdio.h>
#include <omp.h>

int main() {
  printf ("Hello, world:");
  #pragma omp parallel
    printf (" %d", omp_get_thread_num ());
  printf ("\n");
  return 0;
}
```

Listing 3.1 "Hello world" program, OpenMP style.

This program starts as a single thread that first prints out the salutation. Once the execution reaches the `omp parallel` directive, several additional threads are created alongside the existing one. All threads, the initial thread and the newly created threads, together form a *team of threads*. Each thread in the newly established team of threads executes the statement immediately following the directive: in this example it just prints out its unique thread number obtained by calling OpenMP function `omp_get_thread_num`. When all threads have done that threads created by the `omp parallel` directive are terminated and the program continues as a single thread that prints out a single new line character and terminates the program by executing `return 0`.

To compile and run the program shown in Listing 3.1 using GNU GCC C/C++ compiler, use the command-line option `-fopenmp` as follows:

OpenMP: parallel regions

A parallel region within a program is specified as

`#pragma omp parallel` *[clause [[,] clause] …]*
structured-block

A team of threads is formed and the thread that encountered the `omp parallel` directive becomes the **master thread** within this team. The *structured-block* is executed by every thread in the team. It is either a single statement, possibly compound, with a single entry at the top and a single exit at the bottom, or another OpenMP construct. At the end, there is an implicit **barrier**, i.e., only after all threads have finished, the threads created by this directive are terminated and only the master resumes execution.

A parallel region might be refined by a list of *clause*s, for instance

- `num_threads(`*integer*`)` specifies the number of threads that should execute *structured-block* in parallel.

Some other *clause*s applicable to `omp parallel` will be introduced later.

```
$ gcc -fopenmp -o hello-world hello-world.c
$ env OMP_NUM_THREADS=8 ./hello-world
```

(See Appendix A for instructions on how to make OpenMP operational on Linux, macOS, and MS Windows.)

In this program, the number of threads is not specified explicitly. Hence, the number of threads matches the value of the shell variable `OMP_NUM_THREADS`. Setting the value of `OMP_NUM_THREADS` to 8, the program might print out

```
Hello, world: 2 5 1 7 6 0 3 4
```

Without `OMP_NUM_THREADS` being set, the program would set the number of threads to match the number of logical cores threads can run on. For instance, on a CPU with 2 cores and hyper-threading, 4 threads would be used and a permutation of numbers from 0 to 3 would be printed out.

Once the threads are started, it is up to a particular OpenMP implementation and especially the underlying operating system to carry out scheduling and to resolve competition for the single standard output the permutation is printed on. Hence, if the program is run several times, a different permutation of thread numbers will most likely be printed out each time. Try it.

OpenMP: controlling the number of threads

Once a program is compiled, the number of threads can be controlled using the following shell variables:

- `OMP_NUM_THREADS` *comma-separated-list-of-positive-integers*
- `OMP_THREAD_LIMIT` *positive-integer*

The first one sets the number of threads the program should use (or how many threads should be used at every nested level of parallel execution). The second one limits the number of threads a program can use (and takes the precedence over `OMP_NUM_THREADS`).

Within a program, the following functions can be used to control the number of threads:

- `void omp_set_num_threads()` sets the number of threads used in the subsequent parallel regions without explicit specification of the number of threads;
- `int omp_get_num_threads()` returns the number of threads in the current team relating to the innermost enclosing parallel region;
- `int omp_get_max_threads()` returns the maximal number of threads available to the subsequent parallel regions;
- `int omp_get_thread_num()` returns the thread number of the calling thread within the current team of threads.

3.2.2 Monitoring an OpenMP Program

During the design, development, and debugging of parallel programs reasoning about parallel algorithms and how to encode them better rarely suffices. To understand how an OpenMP program actually runs on a multi-core system, it is best to monitor and measure the performance of the program. Even more, this is the simplest and the most reliable way to know how many cores your program actually runs on.

Let us use the program in Listing 3.2 as an illustration. The program starts several threads, each of them printing out one Fibonacci number computed using the naive and time-consuming recursive algorithm.

On most operating systems, it is usually easy to measure the running time of a program execution. For instance, compiling the above program and running it using `time` utility on Linux as

```
$ gcc -fopenmp -O2 -o fibonacci fibonacci.c
$ env OMP_NUM_THREADS=8 time ./fibonacci
```

yields some Fibonacci numbers, and then as the last line of output, the information about the program's running time:

```
#include <stdio.h>
#include <omp.h>

long fib (int n) { return (n < 2 ? 1 : fib (n - 1) + fib (n - 2)); }

int main () {
  int n = 45;
  #pragma omp parallel
    {
      int t = omp_get_thread_num ();
      printf ("%d: %ld\n", t, fib (n + t));
    }
  return 0;
}
```

Listing 3.2 Computing some Fibonacci numbers.

```
106.46 real       298.45 user         0.29 sys
```

(See Appendix A for instructions on how to measure time and monitor the execution of a program on Linux, macOS and MS Windows.)

The user and system time amount to the total time that all logical cores together spent executing the program. In the example above, the sum of the user and system time is bigger than the real time, i.e., the elapsed or wall-clock time. Hence, various parts of the program must have run on several logical cores simultaneously.

Most operating systems provide system monitors that among other metrics show the amount of computation performed by individual cores. This might be very informative during OpenMP program development, but be careful as most system monitor reports the overall load on an individual logical core, i.e., load of all programs running on a logical core.

Using a system monitor while the program shown in Listing 3.2 is run on an otherwise idle system, one can observe the load on individual logical cores during program execution. As threads finish one after another, one can observe how the load on individual logical cores drops as the execution proceeds. Toward the end of execution, with only one thread remaining, it can be seen how the operating system occasionally migrates the last thread from one logical core to another.

3.3 Parallelization of Loops

Most CPU-intensive programs for solving scientific or technical problems spend most of their time running loops so it is best to start with some examples illustrating what OpenMP provides for the efficient and portable implementation of **parallel loops**.

3.3.1 Parallelizing Loops with Independent Iterations

To avoid obscuring the explanation of parallel loops in OpenMP with unnecessary details, we start with a trivial example of a loop parallelization: consider printing out all integers from 1 to some user-specified value, say *max*, in no particular order. The parallel program is shown in Listing 3.3.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (int argc, char *argv[]) {
5    int max; sscanf (argv[1], "%d", &max);
6    #pragma omp parallel for
7      for (int i = 1; i <= max; i++)
8        printf ("%d: %d\n", omp_get_thread_num (), i);
9    return 0;
10 }
```

Listing 3.3 Printing out all integers from 1 to *max* in no particular order.

The program in Listing 3.3 starts as a single initial thread. The value *max* is read and stored in variable `max`. The execution then reaches the most important part of the program, namely, the `for` loop which actually prints out the numbers (each preceded by the number of a thread that prints it out). But the `omp parallel for` directive in line 6 specifies that the `for` loop must be executed in parallel, i.e., its iterations must be divided among and executed by multiple threads running on all available processing units. Hence, a number of **slave threads** is created, one per each available processing unit or as specified explicitly (minus one that the initial thread runs on). The initial thread becomes the **master thread** and together with the newly created slave threads the *team of threads* is formed. Then,

- iterations of the parallel `for` loop are divided among threads where each iteration is executed by the thread it has been assigned to, and
- once all iterations have been executed, all threads in the team are synchronized at the implicit barrier at the end of the parallel `for` loop and all slave threads are terminated.

Finally, the execution proceeds sequentially and the master thread terminates the program by executing `return 0`. The execution of the program in Listing 3.3 is illustrated in Fig. 3.5.

Several observations must be made regarding the program in Listing 3.3 (and execution of parallel `for` loops in general). First, the program in Listing 3.3 does not specify how the iterations should be divided among threads (as explicit scheduling of iterations will be described later). In such cases, most OpenMP implementations divide the entire iteration space into chunks where each chunk containing a subinterval of all iterations is executed by one thread. Note, however, that this must not be the case as if left unspecified, it is up to a particular OpenMP implementation to do as it likes.

OpenMP: parallel loops

A parallel `for` loops are declared as

`#pragma omp for` *[clause [[,] clause] …]*
for-loops

This directive, which must be used within a parallel region, specifies that iterations of one or more nested `for` loops will be executed by the team of threads within the parallel region (`omp parallel for` is a shorthand for writing a `for` loop that itself encompasses the entire parallel region). Each `for` loop among *for-loops* associated with the `omp for` directive must be in the **canonical form**. In C, that means that

- the loop variable is made private to each thread in the team and must be either (unsigned) integer or a pointer,
- the loop variable should not be modified during the execution of any iteration;
- the condition in the `for` loop must be a simple relational expression,
- the increment in the `for` loop must specify a change by constant additive expression;
- the number of iterations of all associated loops must be known before the start of the outermost `for` loop.

A *clause* is a specification that further describes a parallel loop, for instance,

- `collapse(`*integer*`)` specifies how many outermost `for` loops of *for-loops* are associated with the directive, and thus parallelized together;
- `nowait` eliminates the implicit barrier and thus synchronization at the end of *for-loops*.

Some other clauses applicable to `omp parallel for` will be introduced later.

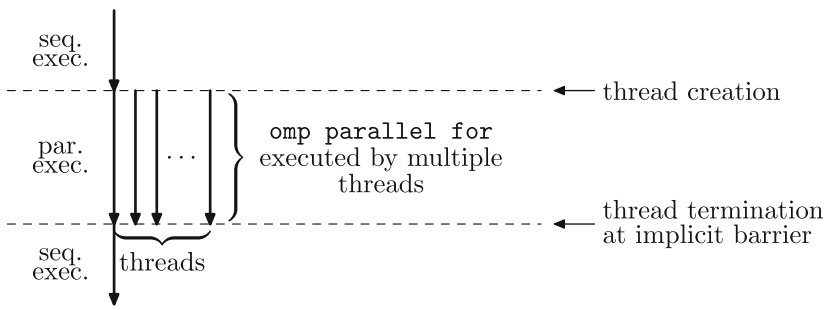

Fig. 3.5 Execution of the program for printing out integers as implemented in Listing 3.3

OpenMP: data sharing

Various data sharing clauses might be used in `omp parallel` directive to specify whether and how data are shared among threads:

- `shared(`*list*`)` specifies that each variable in the *list* is shared by all threads in a team, i.e., all threads share the same copy of the variable;
- `private(`*list*`)` specifies that each variable in the *list* is private to each thread in a team, i.e., each thread has its own local copy of the variable;
- `firstprivate(`*list*`)` is like `private` but each variable listed is initialized with the value it contained when the parallel region was encountered;
- `lastprivate(`*list*`)` is like `private` but when the parallel region ends each variable listed is updated with its final value within the parallel region.

No variable listed in these clauses can be a part of another variable.

If not specified otherwise,

- automatic variables declared outside a parallel construct are shared,
- automatic variables declared within a parallel construct are private,
- static and dynamically allocated variables are shared.

Race conditions, e.g., resulting from different lifetimes of `lastprivate` variables or updating shared variables, must be avoided explicitly by using OpenMP constructs described later on.

Second, once the iteration space is divided into chunks, all iterations of an individual chunk are executed sequentially, one iteration after another. And third, the parallel `for` loop variable `i` is made private in each thread executing a chunk of iterations as each thread must have its own copy of `i`. On the other hand, variable `max` can be shared by all threads as it is set before and is only read within the parallel region.

However, the most important detail that must be paid attention to is that the overall task of printing out all integers from 1 to *max* in no particular order can be divided into *N totally independent* subtasks of *almost the same size*. In such cases, the parallelization is trivial.

As the access to the standard output is serialized, printing out integers does not happen as parallel as it might seem. Therefore, an example of truly parallel computation follows.

Example 3.1 Vector addition

Consider vector addition. The function implementing it is shown in Listing 3.4. Write a program for testing it. As vector addition is not a complex computation at all, use long vectors and perform a large number of vector additions to measure and monitor it.

The structure of function `vectAdd` is very similar to the program for printing out integers shown in Listing 3.3: a simple parallel `for` loop where the result of one iteration is completely independent of the results produced by other loops. Even more, different iterations access different array elements, i.e., they read from and write to completely different memory locations. Hence, no race conditions can occur. □

```
1 double* vectAdd (double *c, double *a, double *b, int n) {
2   #pragma omp parallel for
3     for (int i = 0; i < n; i++)
4       c[i] = a[i] + b[i];
5   return c;
6 }
```

Listing 3.4 Parallel vector addition.

Consider now printing out all pairs of integers from 1 to *max* in no particular order, something that calls for two nested `for` loops. As all iterations of both nested loops are independent, either loop can be parallelized while the other is not. This is achieved by placing the `omp parallel for` directive in front of the loop targeted for parallelization. For instance, the program with the outer `for` loop parallelized is shown in Listing 3.5.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (int argc, char *argv[]) {
5    int max; sscanf (argv[1], "%d", &max);
6    #pragma omp parallel for
7      for (int i = 1; i <= max; i++)
8        for (int j = 1; j <= max; j++)
9          printf ("%d: (%d,%d)\n", omp_get_thread_num (), i, j);
10   return 0;
11 }
```

Listing 3.5 Printing out all pairs of integers from 1 to *max* in no particular order by parallelizing the outermost `for` loop only.

Assume all pairs of integers from 1 to *max* are arranged in a square table. If 4 threads are used and $max = 6$, each iteration of the parallelized outer `for` loop prints out a few lines of the table as illustrated in Fig. 3.6a. Note that the first two threads are assigned twice as much work than the other two threads which, if run on 4 logical cores, will have to wait idle until the first two complete as well.

However, there are two other ways of parallelizing nested loops. First, the two nested `for` loops can be collapsed in order to be parallelized together using clause `collapse(2)` as shown in Listing 3.6.

Because of the clause `collapse(2)` in line 6, the compiler merges the two nested `for` loops into one and parallelizes the resulting single loop. The outer `for` loop running from 1 to *max* and *max* inner `for` loops running from 1 to *max* as well, are replaced by a single loop running from 1 to max^2. All max^2 iterations are divided among available threads together. As only one loop is parallelized, i.e., the

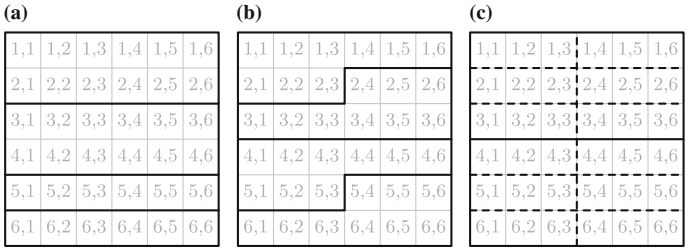

Fig. 3.6 Partition of the problem domain when all pairs of integers from 1 to 6 must be printed using 4 threads: **a** if only the outer `for` loop is parallelized, **b** if both `for` loops are parallelized together, and **c** if both `for` loops are parallelized separately

```
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  #pragma omp parallel for collapse(2)
    for (int i = 1; i <= max; i++)
      for (int j = 1; j <= max; j++)
        printf ("%d: (%d,%d)\n", omp_get_thread_num (), i, j);
  return 0;
}
```

Listing 3.6 Printing out all pairs of integers from 1 to *max* in no particular order by parallelizing both `for` loops together.

one that comprises iterations of both nested `for` loops, the execution of the program in Listing 3.6 still follows the pattern illustrated in Fig. 3.5. For instance, if $max = 6$, all 36 iterations of the collapsed single loop are divided among 4 thread as shown in Fig. 3.6b. Compared with the program in Listing 3.5, the work is more evenly distributed among threads.

The other method of parallelizing nested loops is by parallelizing each `for` loop separately as shown in Listing 3.7.

```
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  #pragma omp parallel for
    for (int i = 1; i <= max; i++) {
      #pragma omp parallel for
        for (int j = 1; j <= max; j++) {
          printf ("%d: (%d,%d)\n", omp_get_thread_num (), i, j);
        }
    }
  return 0;
}
```

Listing 3.7 Printing out all pairs of integers from 1 to max in no particular order by parallelizing each nested `for` loop separately.

OpenMP: nested parallelism

Nested parallelism is enabled or disabled by setting the shell variable

- `OMP_NESTED` *nested*

where *nested* is either `true` or `false`. Within a program, this can be achieved usign the following two functions:

- `void omp_set_nested(int nested)` enables or disables nested parallelism;
- `int omp_get_nested()` tells whether nested parallelism is enabled or disabled.

The number of threads at each nested level can be set by calling function `omp_set_num_threads` or by setting `OMP_NUM_THREADS` . In the latter case, if a list of integers is given each integer specifies the number of threads at a successive nesting level.

To have one parallel region within the other as shown in Listing 3.7 active at the same time, nesting of parallel regions must be enabled first. This is achieved by calling `omp_set_nested(1)` before `mtxMul` is called or by setting `OMP_NESTED` to `true`. Once nesting is activated, iterations of both loops are executed in parallel separately as illustrated in Fig. 3.7. Compare Figs. 3.5 and 3.7 and note how many more threads are created and terminated in the latter, i.e., if nested loops are parallelized separately.

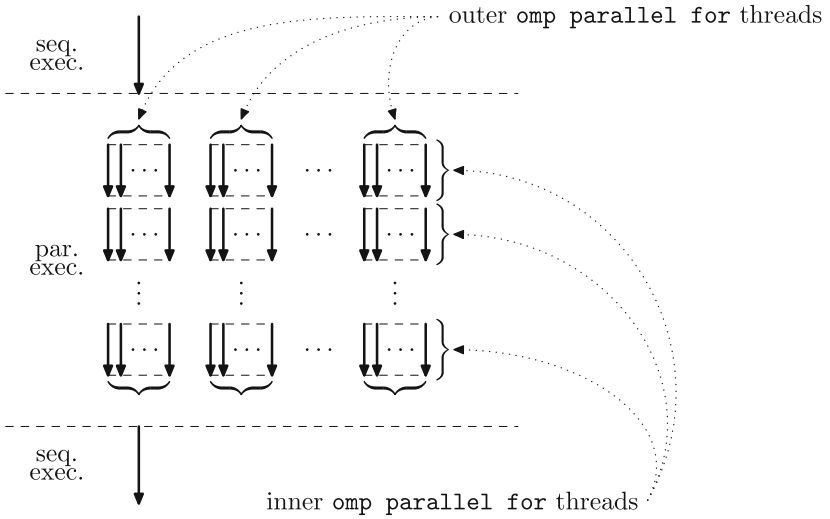

Fig. 3.7 The execution of the program for printing out all pairs of integers using separately parallelized nested loops as implemented in Listing 3.7

By setting `OMP_NUM_THREADS=2,2` and running the program in Listing 3.7, the team of two (outer) threads is established to execute three iterations of the outer loop each as they would even if nesting was disabled. Each iteration of the outer loop must compute one line of the table and thus establishes a team of two (inner) threads to execute 3 iterations of the inner loop each. The table of pairs to be printed out is divided among threads as shown in Figure 1.6 (c). However, be careful while interpreting the output: threads of every iteration of the inner loop are counted from 0 onward because function `omp_get_thread_num` always returns the thread number relative to its team.

Example 3.2 Matrix multiplication

Another example from linear algebra is matrix by matrix multiplication. The classical algorithm, based on the definition, encompasses two nested `for` loops used to compute n^2 independent dot products (and each dot product is computed using yet another, innermost, `for` loop).

Hence, the structure of the matrix multiplication code resembles the code shown in Listings 3.5, 3.6 and 3.7 except that the simple code for printing out the pair of integers is replaced by yet another `for` loop for computing the dot product.

The function implementing multiplication of two square matrices where the two outermost `for` loops are collapsed and parallelized together, is shown in Listing 3.8. As before, write a program for testing it.

```
double **mtxMul (double **c, double **a, double **b, int n) {
  #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) {
        c[i][j] = 0.0;
        for (int k = 0; k < n; k++)
          c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
  return c;
}
```

Listing 3.8 Matrix multiplication where the two outermost loops are parallelized together.

The other way of parallelizing matrix multiplication is shown in Listing 3.9.

```
double **mtxMul (double **c, double **a, double **b, int n) {
  #pragma omp parallel for
    for (int i = 0; i < n; i++)
      #pragma omp parallel for
        for (int j = 0; j < n; j++) {
          c[i][j] = 0.0;
          for (int k = 0; k < n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
  return c;
}
```

Listing 3.9 Matrix multiplication where the two outermost loops are parallelized separately.

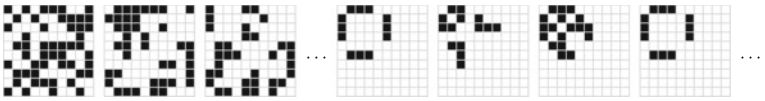

Fig. 3.8 Conway's Game of Life: a particular initial population turns into an oscillating one

Writing functions for matrix multiplication where only one of the two outermost `for`loops is parallelized, either outer of inner, is left as an exercise. □

Example 3.3 Convay's Game of Life

Both examples so far have been taken from linear algebra. Let us now consider something different: Conway's Game of life. It is a zero-player game played on a (finite) plane of square cells, each of which is either "dead" or "alive". At each step in time, a new generation of cells arises where

- each live cell with fewer than two neighbors dies of underpopulation,
- each live cell with two or three neighbors lives on,
- each live cell with more than three neighbors dies of overpopulation, and
- each dead cell with three neighbors becomes a live cell.

It is assumed that each cell has eight neighbors, four along its sides and four on its corners.

Once the initial generation is set, all the subsequent generations can be computed. Sometimes the population of live cells die out, sometimes it turns into a static colony, other times it oscillates forever. Even more sophisticated patterns can appear including traveling colonies and colony generators. Figure 3.8 shows an evolution of an oscillating colony on the 10×10 plane.

The program for computing Convay's Game of life is too long to be included entirely, but its core is shown in Listing 3.10. To understand it, observe the following:

- variable `gens` contains the number of generations to be computed;
- the current generation is stored in a two-dimensional array `plane` containing `size` × `size` cells;
- the next generation is computed into a two dimensional array `aux_plane` containing `size` × `size` cells;
- both two dimensional arrays, i.e., `plane` and `aux_plane`, are allocated as a one-dimensional array of pointers to rows of two-dimensional plane;
- function `neighbors` returns the number of neighbors of the cell in the plane specified by the first argument of size specified by the second argument at position specified by the third and fourth arguments.

Except for the `omp parallel for` directive, the code in Listing 3.10 is the same as if it was written for the sequential execution: the (outermost) `while` loop

runs over all generations to be computed while the inner two loops are used to compute the next generation and store it in `aux_plane` given the current generation in `plane`. More precisely, the rules of the game are implemented in the `switch` statement in lines 6–11: the case for `plane[i][j]==0` implements the rule for dead cells and the case for `plane[i][j]==1` implements the rules for live cells. Once the new generation has been computed, the arrays are swapped so that the generation just computed becomes the current one.

The `omp parallel for` directive in line 2 is used to specify that the iterations of the two `for` loops in lines 3–12 can be performed simultaneously. By inspecting the code, it becomes clear that just like in matrix multiplication every iteration of the outer `for` loop computes one row of the plane representing the next generation and that every iteration of the inner loop computes a single cell of the next generation. As array `plane` is read only and the (i, j)-th iteration of the collapsed loop is the only one writing to the (i, j)-th cell of array `aux_plane`, there can be no race conditions and there are no dependencies among iterations.

The implicit synchronization at the end of the parallelized loop nest is crucial. Without synchronization, if the master thread performed the swap in line 13 before other threads finished the computation within both `for` loops, it would cause all other threads to mess up the computation profoundly.

Finally, instead of parallelizing the two `for` loops together it is also possible to parallelize them separately just like in matrix multiplication. But the outermost loop, i.e., `while` loop, cannot be parallelized as every iteration (except the first one) depends on the result of the previous one. □

3.3.2 Combining the Results of Parallel Iterations

In most cases, however, individual loop iterations aren't entirely independent as they are used to solve a single problem together and thus each iteration contributes its part to the combined solution. Most often then not partial results of different iterations must be combined together.

```
1  while (gens-- > 0) {
2    #pragma omp parallel for collapse(2)
3      for (int i = 0; i < size; i++)
4        for (int j = 0; j < size; j++) {
5          int neighs = neighbors (plane, size, i, j);
6          switch (plane[i][j]) {
7            case 0: aux_plane[i][j] = (neighs == 3);
8                    break;
9            case 1: aux_plane[i][j] = (neighs == 2) || (neighs == 3);
10                   break;
11         }
12       }
13   char **tmp_plane = aux_plane; aux_plane = plane; plane = tmp_plane;
14 }
```

Listing 3.10 Computing generations of Convay's Game of Life.

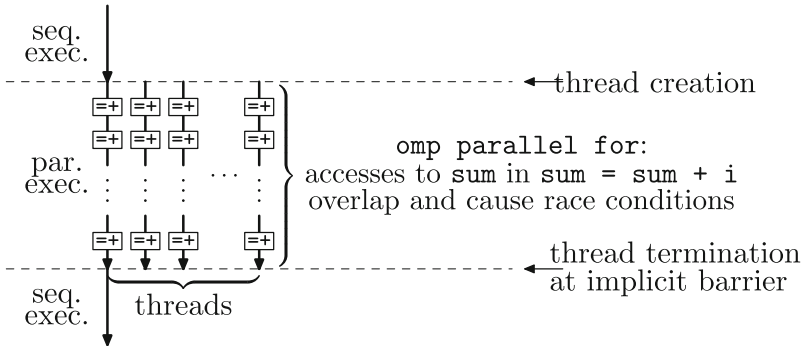

Fig. 3.9 Execution of the summation of integers as implemented in Listing 3.11

If integers from the given interval are to be added instead of printed out, all subtasks must somehow cooperate to produce the correct sum. The first parallel solution that comes to mind is shown in Listing 3.11. It uses a single variable `sum` where the result is to be accumulated.

```
1  #include <stdio.h>
2
3  int main (int argc, char *argv[]) {
4    int max; sscanf (argv[1], "%d", &max);
5    int sum = 0;
6    #pragma omp parallel for
7      for (int i = 1; i <= max; i++)
8        sum = sum + i;
9    printf ("%d\n", sum);
10   return 0;
11 }
```

Listing 3.11 Summation of integers from a given interval using a single shared variable — wrong.

Again, iterations of the parallel `for` loop are divided among multiple threads. In all iterations, threads use the same shared variable `sum` on both sides of assignment in line 8, i.e., they read from and write to the same memory location. As illustrated in Fig. 3.9 where every box containing =+ denotes the assignment `sum = sum + i`, the accesses to variable `sum` overlap and the program is very likely to encounter race conditions illustrated in Fig. 3.3.

Indeed, if this program is run multiple times using several threads, it is very likely that it will not always produce the same result. In other words, from time to time it will produce the wrong result. Try it.

To avoid race conditions, the assignment `sum = sum + i` can be put inside a **critical section** — a part of a program that is performed by at most one thread at a time. This is achieved by the `omp critical` directive which is applied to the statement or a block immediately following it. The program using critical sections is shown in Listing 3.12.

The program works correctly because the `omp critical` directive performs locking around the code it contains, i.e., the code that accesses variable `sum`, as

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  int sum = 0;
  #pragma omp parallel for
    for (int i = 1; i <= max; i++)
      #pragma omp critical
        sum = sum + i;
  printf ("%d\n", sum);
  return 0;
}
```

Listing 3.12 Summation of integers from a given interval using a critical section — slow.

illustrated in Fig. 3.4 and thus prevents race conditions. However, the use of critical sections in this program makes the program slow because at every moment at most one thread performs the addition and assignment while all other threads are kept waiting as illustrated in Fig. 3.10.

It is worth comparing the running times of the programs shown in Listings 3.11 and 3.12. On a fast multi-core processor, a large value for *max* possibly causing an overflow is needed so that the difference can be observed.

Another way to avoid race conditions is to use **atomic access** to variables as shown in Listing 3.13.

Although `sum` is a single variable shared by all threads in the team, the program computes the correct result as the `omp atomic` directive instructs the compiler

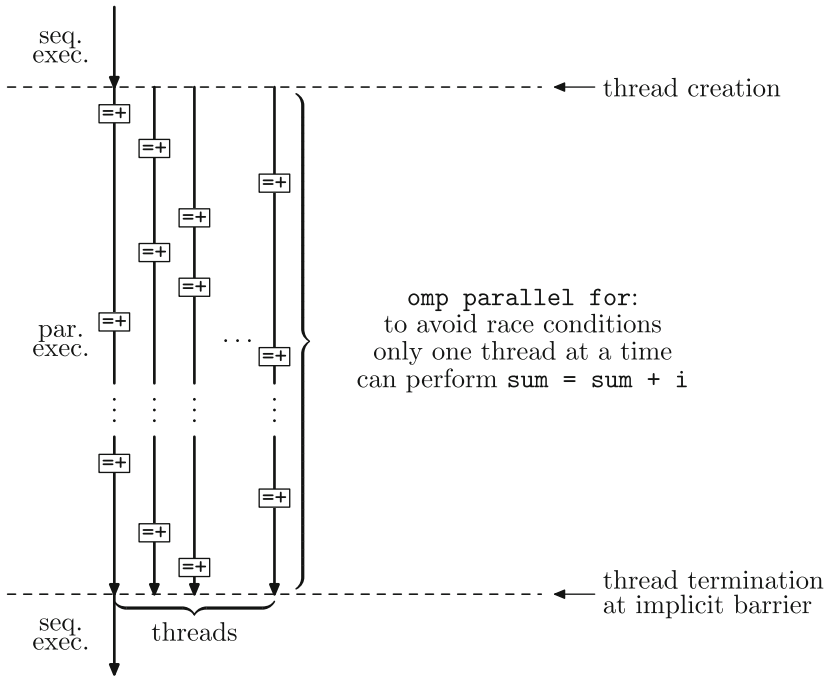

Fig. 3.10 Execution of the summation of integers as implemented in Listing 3.12

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  int sum = 0;
  #pragma omp parallel for
    for (int i = 1; i <= max; i++)
      #pragma omp atomic
        sum = sum + i;
  printf ("%d\n", sum);
  return 0;
}
```

Listing 3.13 Summation of integers from a given interval using a atomic variable access — faster.

to generate code where `sum = sum + i` update is performed as a single atomic operation (possibly using a hardware supported read-modify-write instructions).

The concepts of a critical section and atomic accesses to a variable are very similar, except that an atomic access is much simpler, and thus usually faster than a critical section that can contain much more elaborate computation. Hence, the program in Listing 3.13 is essentially executed as illustrated in Fig. 3.10.

OpenMP: critical sections

A critical section is declared as

`#pragma omp critical` *[*`(`*name*`)` *[*`hint(`*hint*`)`*]]*
structured-block

The *structured-block* is guaranteed to be executed by a single thread at a time. A critical section can be given *name*, an identifier with external linkage so that different tasks can implement their own implementation of the same critical section. A named critical section can be given a constant integer expression *hint* to establish a detailed control underlying locking.

To prevent race conditions and to avoid locking or explicit atomic access to variables at the same time, OpenMP provides a special operation called **reduction**. Using it, the program in Listing 3.11 is rewritten to the program shown in Listing 3.14.

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  int sum = 0;
  #pragma omp parallel for reduction(+:sum)
    for (int n = 1; n <= max; n++)
      sum = sum + n;
  printf ("%d\n", sum);
  return 0;
}
```

Listing 3.14 Summation of integers from a given interval using reduction — fast.

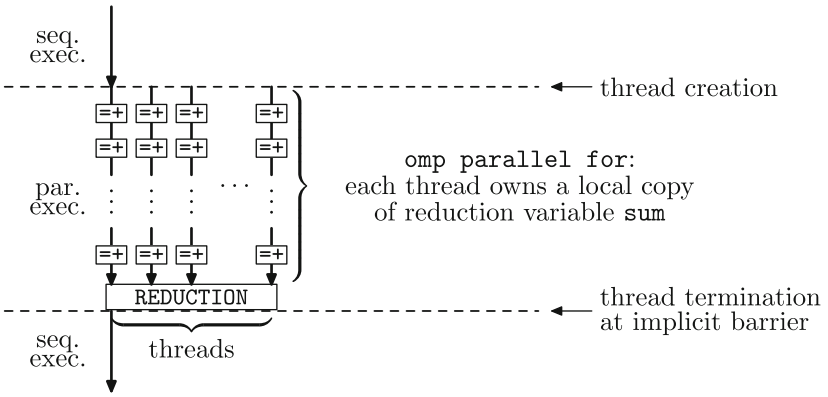

Fig. 3.11 Execution of the summation of integers as implemented in Listing 3.14

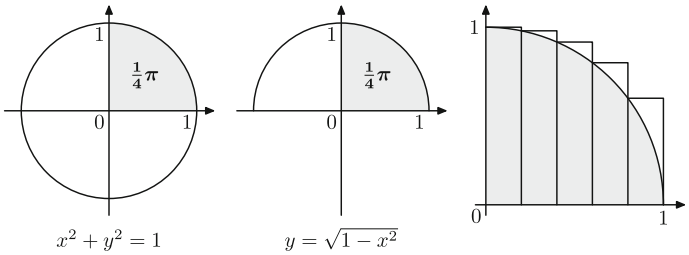

Fig. 3.12 Integrating $y = \sqrt{1 - x^2}$ numerically from 0 to 1

The additional clause `reduction(+:sum)` states that T private variables `sum` are created, one variable per thread. The computation within each thread is performed using the private variable `sum` and only when the parallel `for` loop has finished are the private variables `sum` added to variable `sum` declared in line 5 and printed out in line 9. The compiler and the OpemMP runtime system perform the final summation of local variables `sum` in a way suitable for the actual target architecture.

The program in Listing 3.14 is executed as shown in Fig. 3.11. At first glance, it is similar to the execution of the incorrect program in Listing 3.11, but there are no race conditions because in line 8 each thread uses its own private variable `sum`. At the end of the parallel `for` loop, however, these private variables are added to the global variable `sum`.

Example 3.4 Computing π by numerical integration

There are many problems that require combining results of loop iterations. Let us start with a numerical integration in one dimension. Suppose we want to compute number π by computing the area of the unit circle defined by equation $x^2 + y^2 = 1$. Converting it to its explicit form $y = \sqrt{1 - x^2}$, we want to compute number π using the equation

$$\pi = 4 \int_0^1 \sqrt{1 - x^2} dx$$

OpenMP: atomic access

Atomic access to a variable within *expression-stmt* is declared as

`#pragma omp atomic` [`seq_cst` [,]] *atomic-clause* [[,] `seq_cst`]
expression-stmt

or

`#pragma omp atomic` [`seq_cst`]
expression-stmt

when `update` is assumed. The `omp atomic` directive enforces an exclusive access to a storage location among all threads in the binding thread set without regard to the teams to which the threads belong.

The three most important *atomic-clause*s are the following:

- `read` causes an atomic read of `x` in statements of the form *expr* = `x`;
- `write` causes an atomic write to `x` in statements of the form `x` = *expr*;
- `update` causes an atomic update of `x` in statements of the form `++x`, `x++`, `--x`, `x--`, `x` = `x` *binop expr*, `x` = *expr binop* `x`, `x` *binop*= *expr*.

If `seq_cst` is used, an implicit flush operation of atomically accessed variable is performed after *statement-expr*.

OpenMP: reduction

Technically, reduction is yet another data sharing attribute specified by

`reduction(`*reduction-identifier* `:` *list*`)`

clause.
For each variable in the *list*, a private copy is created in each thread of a parallel region, and initialized to a value specified by the *reduction-identifier*. At the end of a parallel region, the original variable is updated with values of all private copies using the operation specified by the *reduction-identifier*.
The *reduction-identifier* may be `+`, `-`, `&`, `|`, `^`, `&&`, `||`, `min`, and `max`. For `*` and `&&` the initial value is `1`; for `min` and `max` the initial value is the largest and the smallest value of the variable's type, respectively; for all other operations, the initial value is `0`.

as illustrated in Fig. 3.12.

The integral on the right hand side of the equation above is computed numerically. Therefore, the interval [0, 1] is cut into N intervals $[\frac{1}{N}i, \frac{1}{N}(i+1)]$ where $0 \leq i \leq (N-1)$ for some chosen number of intervals N. To keep the program simple, the left Riemann sum is chosen: the area of each rectangle is computed as the width of a

rectangle, i.e., $1/N$, multiplied by the function value computed in the left-end point of the interval, i.e., $\sqrt{1-(i/N)^2}$. Thus,

$$\int_0^1 \sqrt{1-x^2}dx \approx \sum_{i=0}^{N-1} \left(\frac{1}{N} \sqrt{1-\left(\frac{1}{N}i\right)^2} \right)$$

for large enough N.

The program for computing π using the sum on the right-hand side of the approximation is shown in Listing 3.15.

```
#include <stdio.h>
#include <math.h>

int main (int argc, char *argv[]) {
  int intervals; sscanf (argv[1], "%d", &intervals);
  double integral = 0.0;
  double dx = 1.0 / intervals;
  #pragma omp parallel for reduction(+:integral)
    for (int i = 0; i < intervals; i++) {
      double x = i * dx;
      double fx = sqrt (1.0 - x * x);
      integral = integral + fx * dx;
    }
  double pi = 4 * integral;
  printf ("%20.18lf\n", pi);
  return 0;
}
```

Listing 3.15 Computing π by integrating $\sqrt{1-x^2}$ from 0 to 1.

Despite the elements of numerical integration, the program in Listing 3.15 is remarkably similar to the program in Listing 3.14—after all, this numerical integration is nothing but a summation of rectangular areas. Nevertheless, one important detail should not be missed: unlike `intervals`, `integral`, and `dx` variables `x` and `fx` must be thread private.

At this point, it is perhaps worth showing once again that not every loop can be parallelized. By rewriting lines 7–13 of Listing 3.15 to code shown in Listing 3.16 a multiplication inside the loop was replaced by addition.

```
  double x = 0.0;
  #pragma omp parallel for reduction(+:integral)
    for (int i = 0; i < intervals; i++) {
      double fx = sqrt (1.0 - x * x);
      integral = integral + fx * dx;
      x = x + dx;
    }
```

Listing 3.16 Computing π by integrating $\sqrt{1-x^2}$ from 0 to 1 using a non-paralellizable loop.

This works well if the program is run by only one thread (set `OMP_NUM_THREADS` to 1), but produces the wrong result if multiple threads are used. The reason is that the iterations are no longer independent: the value of `x` is propagated from one iteration

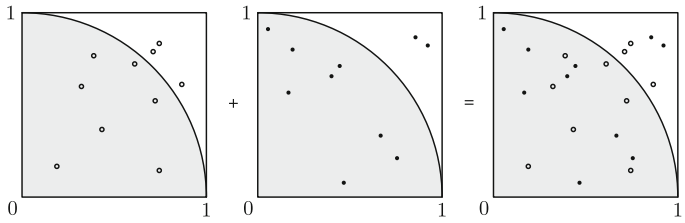

Fig. 3.13 Computing π by random shooting: different threads shoot independently, but the final result is a combination of all shots

to another so the next iteration cannot be performed until the previous has been finished. However, the `omp parallel for` directive in line 2 of Listing 3.16 states that the loop can and should be parallelized. The programmer unwisely requested the parallelization and took the responsibility for ensuring the loop can indeed be parallelized too lightly. □

Example 3.5 Computing π using random shooting

Another way of computing π is to shoot randomly into a square $[0, 1] \times [0, 1]$ and count how many shots hit inside the unit circle and how many do not. The ratio of hits vs. all shots is an approximation of the area of the unit circle within $[0, 1] \times [0, 1]$. As each shot is independent of another, shots can be distributed among different threads. Figure 3.13 illustrates this idea if two threads are used. The implementation, for any number of threads, is shown in Listing 3.17.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  double rnd (unsigned int *seed) {
6    *seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
7    return ((double)(*seed)) / (1 << 24);
8  }
9
10 int main (int argc, char *argv[]) {
11   int num_shots; sscanf (argv[1], "%d", &num_shots);
12   unsigned int seeds[omp_get_max_threads()];
13   for (int thread = 0; thread < omp_get_max_threads(); thread++)
14     seeds[thread] = thread;
15   int num_hits = 0;
16   #pragma omp parallel for reduction(+:num_hits)
17     for (int shot = 0; shot < num_shots; shot++) {
18       int thread = omp_get_thread_num();
19       double x = rnd (&seeds[thread]);
20       double y = rnd (&seeds[thread]);
21       if (x * x + y * y <= 1) num_hits = num_hits + 1;
22     }
23   double pi = 4.0 * (double)num_hits / (double)num_shots;
24   printf ("%20.18lf\n", pi);
25   return 0;
26 }
```

Listing 3.17 Computing π by random shooting using a parallel loop.

From the parallel programming view, the program in Listing 3.17 is basically simple: `num_shots` are shot within the parallel `for` loop in lines 17–23 and the number of hits is accumulated in variable `num_shots` . Furthermore, it also resembles the program in Listing 3.14: the results of independent iterations combined together to yield the final result (Fig. 3.14).

The most intricate part of the program is generating random shots. The usual random generators, i.e., `rand` or `random`, are not *reentrant* or *thread-safe*: they should not be called in multiple threads because they use a single hidden state that is modified on each call regardless of the thread the call is made in. To avoid this problem, function `rnd` has been written: it takes a seed, modifies it, and returns a random value in the interval [0, 1). Hence, a distinct seed for each thread is created in lines 12–14 where OpenMP function `omp_get_max_threads` is used to obtain the number of future threads that will be used in the parallel `for` loop later on. Using these seeds, the program contains one distinct random generator for each thread.

The rate of convergence toward π is much lower than if random shooting is used instead of numerical integration. However, this example shows how simple it is to implement a wide class of Monte Carlo methods if random generator is applied correctly: one must only run all random based individual experiments, e.g., shots into $[0, 1] \times [0, 1]$ in lines 18–20, and aggregate the results, e.g., count the number of hits within the unit circle.

As long as the number of individual experiments is known in advance and the complexity of individual experiments is approximately the same, the approach is presented in Listing 3.17 suffices. Otherwise, a more sophisticated approach is needed, but more about that later.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  double rnd (unsigned int *seed) {
6    *seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
7    return ((double)(*seed)) / (1 << 24);
8  }
9
10 int main (int argc, char *argv[]) {
11   int num_shots; sscanf (argv[1], "%d", &num_shots);
12   int num_hits = 0;
13   #pragma omp parallel
14   {
15     unsigned int seed = omp_get_thread_num();
16     #pragma omp for reduction(+:num_hits)
17       for (int shot = 0; shot < num_shots; shot++) {
18         double x = rnd (&seed);
19         double y = rnd (&seed);
20         if (x * x + y * y <= 1) num_hits = num_hits + 1;
21       }
22   }
23   double pi = 4.0 * (double)num_hits / (double)num_shots;
24   printf ("%20.18lf\n", pi);
25   return 0;
26 }
```

Listing 3.18 Computing π by random shooting using a parallel loop.

Before proceeding, we can rewrite the program in Listing 3.17 to a simpler one. By splitting the `omp parallel` and `omp for` we can define the thread-local seed inside the parallel region as shown in Listing 3.19.

Let us demonstrate that computing π by random shooting into $[0, 1] \times [0, 1]$ and counting the shots inside the unit circle can also be encoded differently as shown in Listing 3.19, but at its core it stays the same.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  double rnd (unsigned int *seed) {
6    *seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
7    return ((double)(*seed)) / (1 << 24);
8  }
9
10 int main (int argc, char *argv[]) {
11   int num_shots; sscanf (argv[1], "%d", &num_shots);
12   int num_hits = 0;
13   #pragma omp parallel reduction(+:num_hits)
14     {
15       unsigned int seed = omp_get_thread_num ();
16       int loc_shots = (num_shots / omp_get_num_threads ()) +
17         ((num_shots % omp_get_num_threads () > ↩
     omp_get_thread_num ())
18           ? 1 : 0);
19       while (loc_shots-- > 0) {
20         double x = rnd (seed);
21         double y = rnd (seed);
22         if (x * x + y * y <= 1) num_hits = num_hits + 1;
23       }
24     }
25   double pi = 4.0 * (double)num_hits / (double)num_shots;
26   printf ("%lf\n", pi);
27   return 0;
28 }
```

Listing 3.19 Computing π by random shooting using parallel sections.

Namely, the parallel regions, one per each available thread, specified by the `omp parallel` directive in line 13 are used instead of the parallel `for` loop (see also Listings 3.1 and 3.2). Within each parallel region, the seed for the thread-local random generator is generated in lines 15. Then, the number of shots that must be carried out by the thread is computed in lines 16–18 and finally all shots are performed in a thread-local sequential `while` loop in lines 19–23. Unlike the iterations of the parallel `par` loop in Listing 3.18, the iterations of the `while` loop do not contain a call of function `omp_get_thread_num`. However, the aggregation of the results obtained by the parallel regions, i.e., the number of hits, is done using the reduction in the same way as in Listing 3.18. □

3.3.3 Distributing Iterations Among Threads

So far no attention has been paid on how iterations of a parallel loop, or of a several collapsed parallel loops, are distributed among different threads in a single team of threads. However, OpenMP allows the programmer to specify several different iteration scheduling strategies.

Consider computing the sum of integers from a given interval again using the program shown in Listing 3.14. This time, however, the program will be modified as shown in Listing 3.20. First, the `schedule(runtime)` clause is added to the `omp for` directive in line 8. It allows the iteration schedule strategy to be defined once the program is started using the shell variable `OMP_SCHEDULE` . Second, in line 10, each iteration prints out the the number of thread that executes it. And third, different iterations take different time to execute as specified by the argument of function `sleep` in line 11: iterations 1, 2, and 3 require 2, 3, and 4 s, respectively, while all other iterations require just 1 second.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <omp.h>
4
5  int main (int argc, char *argv[]) {
6    int max; sscanf (argv[1], "%d", &max);
7    long int sum = 0;
8    #pragma omp parallel for reduction(+:sum) schedule(runtime)
9      for (int i = 1; i <= max; i++) {
10       printf ("%2d @ %d\n", i, omp_get_thread_num());
11       sleep (i < 4 ? i + 1 : 1);
12       sum = sum + i;
13     }
14   printf ("%ld\n", sum);
15   return 0;
16 }
```

Listing 3.20 Summation of integers from a given interval where iteration scheduling strategy is determined in runtime.

Suppose 4 threads are being used and $max = 14$:

- If `OMP_SCHEDULE=static`, the iterations are divided into chunks each containing approximately the same number of iterations and each thread is given at most one chunk. One possible `static` distribution of 14 iterations among 4 threads (but not the only one, see [18]) is

$$T_0\colon \underbrace{\{1,2,3,4\}}_{10\text{ secs}} \quad T_1\colon \underbrace{\{5,6,7,8\}}_{4\text{ secs}} \quad T_2\colon \underbrace{\{9,10,11\}}_{3\text{ secs}} \quad T_3\colon \underbrace{\{12,13,14\}}_{3\text{ secs}}$$

  Thread T_0 is assigned to all the most time-consuming iterations: although iteration when `i` equals 4 takes 1 second, iterations when `i` is either 1, 2, or 3 require 2, 3, and 4 s, respectively. Each iteration assigned to any other thread takes only

OpenMP: scheduling parallel loop iterations

Distributing iterations of parallel loops among team threads is controlled by the `schedule` clause. The most important options are as follows:

- `schedule(static)`: The iterations are divided into chunks each containing approximately the same number of iterations and each thread is given at most one chunk.
- `schedule(static,`*chunk_size*`)`: The iterations are divided into chunks where each chunk contains *chunk_size* iterations. Chunks are then assigned to threads in a round-robin fashion.
- `schedule(dynamic,`*chunk_size*`)`: The iterations are divided into chunks where each chunk contains *chunk_size* iterations. Chunks are assigned to threads dynamically: each thread takes one chunk at a time out of the common pool of chunks, executes it and requests a new chunk until the pool is empty.
- `schedule(auto)`: The selection of the scheduling strategy is left to the compiler and the runtime system.
- `schedule(runtime)`: The scheduling strategy is specified at run time using the shell variable `OMP_SCHEDULE`.

If no `schedule` clause is present in the `omp for` directive, the compiler and runtime system are allowed to choose the scheduling strategy on their own.

1 s. Hence, thread T_0 finishes much later than all other threads as can be seen in Fig. 3.15.

- If `OMP_SCHEDULE=static,1` or `OMP_SCHEDULE=static,2`, the iterations are divided into chunks containing 1 or 2 iterations, respectively. Chunks are then assigned to threads in a round-robin fashion as

$$T_0: \underbrace{\{1;5;9;13\}}_{5\text{ secs}} \quad T_1: \underbrace{\{2;6;10;14\}}_{6\text{ secs}} \quad T_2: \underbrace{\{3;7;11\}}_{6\text{ secs}} \quad T_3: \underbrace{\{4;8;12\}}_{3\text{ secs}}$$

or

$$T_0: \underbrace{\{1,2;9,10\}}_{7\text{ secs}} \quad T_1: \underbrace{\{3,4;11,12\}}_{7\text{ secs}} \quad T_2: \underbrace{\{5,6;13,14\}}_{4\text{ secs}} \quad T_3: \underbrace{\{7,8\}}_{2\text{ secs}}$$

where semicolon separates different chunks assigned to the same thread.
As shown in Fig. 3.14, the running times of different threads differ less than if simple static scheduling is used. Furthermore, the overall running time is reduced from 10 to 6 or 7 s, depending on the chunk size.

- If `OMP_SCHEDULE=dynamic,1` or `OMP_SCHEDULE=dynamic,2`, the iterations are divided into chunks containing 1 or 2 iterations, respectively. Chunks are assigned to threads dynamically: each thread takes one chunk at a time out of the common pool of chunks, executes it and requests a new chunk until the pool

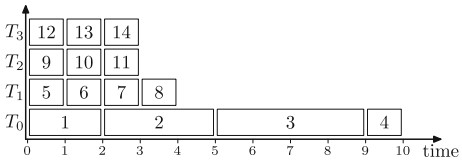

Fig. 3.14 A distribution of 14 iterations among 4 threads where iterations 1, 2 and 3 require more time than other iterations, using `static` iteration scheduling strategy

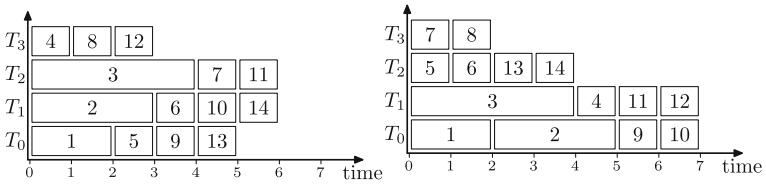

Fig. 3.15 A distribution of 14 iterations among 4 threads where iterations 1, 2 and 3 require more time than other iterations, using `static,1` (left) and `static,2` (right) iteration scheduling strategies

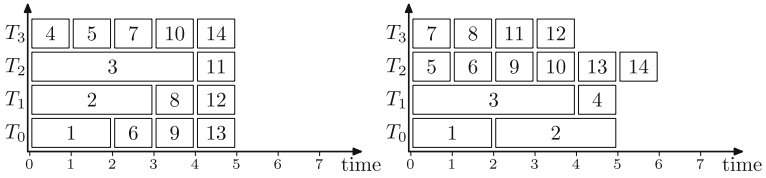

Fig. 3.16 A distribution of 14 iterations among 4 threads where iterations 1, 2 and 3 require more time than other iterations, using `dynamic,1` (left) and `dynamic,2` (right) iteration scheduling strategies

is empty. Hence, two possible dynamic assignments, for chunks consisting of 1 and 2 iterations, respectivelly, are

$$T_0: \underbrace{\{1; 6; 9; 13\}}_{5\text{ secs}} \quad T_1: \underbrace{\{2; 8; 12\}}_{5\text{ secs}} \quad T_2: \underbrace{\{3; 11\}}_{5\text{ secs}} \quad T_3: \underbrace{\{4; 5; 7; 10; 14\}}_{5\text{ secs}}$$

or

$$T_0: \underbrace{\{1, 2\}}_{5\text{ secs}} \quad T_1: \underbrace{\{3, 4\}}_{5\text{ secs}} \quad T_2: \underbrace{\{5, 6; 9, 10; 13, 14\}}_{6\text{ secs}} \quad T_3: \underbrace{\{7, 8; 11, 12\}}_{4\text{ secs}}$$

The scheduling of iterations is illustrated in Fig. 3.16: the overall running is further reduced to 5 or 6 s, again depending on the chunk size. The overall running time of 5 s is the minimal possible as each thread performs the same amount of work.

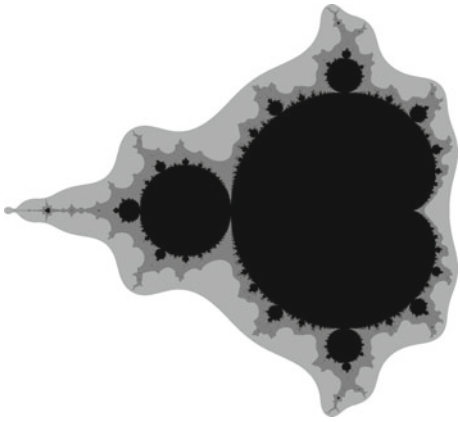

Fig. 3.17 The Mandelbrot set in the complex plane $[-2.6, +1.0] \times [-1.2i, +1.2i]$ (black) and its complement (white, light and dark gray). The darkness of each point indicates the time needed to establish whether a point belongs to the Mandelbrot set or not

Example 3.6 Mandelbrot set

The appropriate choice of iteration scheduling strategy always depends on the problem that is being solved. To see why the iteration scheduling strategy matters, consider computing the Mandelbrot set. It is defined as

$$M = \{c \; ; \; \limsup_{n \to \infty} |z_n| \leq 2 \text{ where } z_0 = 0 \; \wedge \; z_{n+1} = z_n^2 + c\}$$

and shown in Fig. 3.17.

The Mandelbrot set can be computed using the program shown in Listing 3.21 which generates a picture consisting of `i_size` ×`j_size` pixels. For each pixel, the sequence $z_{n+1} = z_n^2 + c$ where c represents the pixel coordinates in the complex plane, is iterated until it either diverges ($|z_{n+1}| > 2$) or the maximum number of iterations (`max_iters`, defined in advance) is reached.

```
#pragma omp parallel for collapse(2) schedule(runtime)
  for (int i = 0; i < i_size; i++) {
    for (int j = 0; j < j_size; j++) {
      // printf ("# (%d,%d) t=%d\n", i, j, omp_get_thread_num());
      double c_re = min_re + i * d_re;
      double c_im = min_im + j * d_im;

      double z_re = 0.0;
      double z_im = 0.0;
      int iters = 0;
      while ((z_re * z_re + z_im * z_im <= 4.0) &&
             (iters < max_iters)) {
        double new_z_re = z_re * z_re - z_im * z_im + c_re;
        double new_z_im = 2 * z_re * z_im + c_im;
        z_re = new_z_re; z_im = new_z_im;
        iters = iters + 1;
      }
      picture[i][j] = iters;
    }
  }
```

Listing 3.21 Computing the Mandelbrot set.

To produce Fig. 3.17, `max_iters` has been set to 100. Each point of the black region, i.e., within the Mandelbrot set, takes 100 iterations to compute. However, each point within the dark gray region requires more than 10 yet less than 100 iterations. Likewise, each point within the light gray region requires more than 5 and less than 10 iterations and all the rest, i.e., points colored white, require at most 5 iterations each. As different points and thus different iterations of the collapsed `for` loops in lines 2 and 3 require significantly different amount of computation, it matters what iteration scheduling strategy is used. Namely, if `static,100` is used instead of simply `static`, the running time is reduced by approximately 30 percent; the choice of `dynamic,100` reduces the running time even more. Run the program and measure its running time under different iteration scheduling strategies. □

3.3.4 The Details of Parallel Loops and Reductions

The parallel `for` loop and reduction operation are so important in OpenMP programming that they should be studied and understood in detail.

Let us return to the program for computing the sum of integers from 1 to *max* as shown in Listing 3.14. If it assumed that T, the number of threads, divides *max* and the `static` iteratin scheduling startegy is used, the program can be rewritten into the one shown in Listing 3.22. (See exercises for the case when T does not divide *max*.)

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (int argc, char *argv[]) {
5    int max; sscanf (argv[1], "%d", &max);
6    int ts = omp_get_max_threads ();
7    if (max % ts != 0) return 1;
8    int sums[ts];
9    #pragma omp parallel
10     {
11       int t = omp_get_thread_num ();
12       int lo = (max / ts) * (t + 0) + 1;
13       int hi = (max / ts) * (t + 1) + 0;
14       sums[t] = 0;
15       for (int i = lo; i <= hi; i++)
16         sums[t] = sums[t] + i;
17     }
18   int sum = 0;
19   for (int t = 0; t < ts; t++) sum = sum + sums[t];
20   printf ("%d\n", sum);
21   return 0;
22 }
```

Listing 3.22 Implementing efficient summation of integers by hand using simple reduction.

The initial thread first obtains T, the number of threads available (using OpenMP function `omp_get_max_threads`), and creates an array `sums` of variables used for summation within each thread. Although the array `sums` is going to be shared by all threads, each thread will access only one of its T elements.

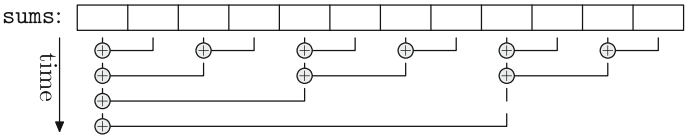

Fig. 3.18 Computing the reduction in time $O(\log_2 T)$ using $T/2$ threads when $T = 12$

Reaching `omp parallel` region the master thread creates $(T - 1)$ slave threads to run alongside the master thread. Each thread, master or slave, first computes its subinterval (lines 11–12), initializes its local summation variable to 0 (line 13), and then executes its thread-local sequential `for` loop (line 14–15). Once all threads have finished computing local sums, only the master thread is left alive. It adds the local summation variables and prints the result. The overall execution is performed as shown in Fig. 3.9. However, no race conditions appear because each thread uses its own summation variable, i.e., the `t`-th thread uses the `t`-th element `sums[t]` of array `sums`.

From the implementation point of view, the program in Listing 3.22 uses array `sums` instead of thread-local summation variables and performs the reduction by the master thread only. Array `sums` is created by the master thread before creating slave threads so that the explicit reduction, which is performed in line 18 after the slave threads have been terminated and their local variables (`t`, `lo`, `hi`, and `n`) have been lost, can be implemented.

Furthermore, the reduction is performed by adding local summation variables, i.e., elements of `sums`, one after another to variable `sum`. This takes $O(T)$ time and works fine if the number of threads is small, e.g., $T = 4$ or $T = 8$. However, if there are a few hundred threads, a solution shown in Listing 3.23 that works in time $O(\log_2 T)$ and produces the result in `sums[0]`, is often preferred (unless the target system architecture requires even more sophisticated method).

```
1  for (int d = 1; d < ts; d = d * 2)
2    #pragma omp parallel for
3      for (int t = 0; t < ts; t = t + 2 * d)
4        if (t + d < ts) sums[t] = sums[t] + sums[t + d];
```

Listing 3.23 Implementing efficient summation of integers by hand using simple reduction.

The idea behind the code shown in Listing 3.23 is illustrated in Fig. 3.18. In Listing 3.23 variable, `d` contains the distance between elements of array `sums` being added, and as it doubles in each iteration, there are $\lceil \log_2 T \rceil$ iterations of the outer loop. Variable `t` denotes the left element of each pair being added in the inner loop. But as the inner loop is performed in parallel by at least $T/2$ threads which operate on distinct elements of array `sums`, all additions in the inner loop are performed simultaneously, i.e., in time $O(1)$.

Note that either method used for computing the reduction uses $(T - 1)$ additions. However, in the first method (line 18 of Listing 3.22) additions are performed one

after another while in the second method (Listing 3.23) certain additions can be performed simultaneously.

3.4 Parallel Tasks

Although most parallel programs spend most of their time running parallel loops, this is not always the case. Hence, it is worth exploring how a program consisting of different tasks can be parallelized.

3.4.1 Running Independent Tasks in Parallel

As above, where parallelization of loops that need not combine the results of its iterations was explained first, we start with explanation of **tasks** where cooperation is not needed.

Consider computing the sum of integers from 1 to *max* one more time. At the end of a previous section, it was shown how iterations of a single parallel `for` loop are distributed among threads. This time, however, the interval from 1 to *max* is split into a number of mutually disjoint subintervals. For each subinterval, a task that first computes the sum of all integers of a subinterval and then adds the sum of the subinterval to the global sum, is used.

The idea is implemented as the program in Listing 3.24. For the sake of simplicity, it is assumed that T, denoting the number of tasks and stored in variable `tasks`, divides *max*.

Computing the sum is performed in the `parallel` block in lines 9–25. The `for` loop in line 12 creates all T tasks where each task is defined by the code in lines 13–23. Once the tasks are created, it is more or less up to OpenMP's runtime system to schedule tasks and execute them.

The important thing, however, is that the `for` loop in line 12 is executed by only one thread as otherwise each thread would create its own set of T tasks. This is achieved by placing the `for` loop in line 12 under the OpenMP directive `single`.

The OpenMP directive `task` in line 13 specifies that the code in lines 14–23 is to be executed as a single task. The local sum is initialized to 0 and the subinterval bounds are computed from the task number, i.e., `t`. The integers of the subinterval are added up and the local sum is added to the global sum using atomic section to prevent a race condition between two different tasks.

Note that when a new task is created, the execution of the task that created the new task continues without delay; once created, the new task has a life of its own. Namely, when the master thread in Listing 3.24 executes the `for` loop, it creates one new task in each iteration, but the iterations (and thus creation of new tasks) are executed one after another without waiting for the newly created tasks to finish (in fact, it would make no sense at all to wait for them to finish). However, all tasks must

```
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  int tasks; sscanf (argv[2], "%d", &tasks);
  if (max % tasks != 0) return 1;
  int sum = 0;
  #pragma omp parallel
    {
      #pragma omp single
        for (int t = 0; t < tasks; t++) {
          #pragma omp task
            {
              int local_sum = 0;
              int lo = (max / tasks) * (t + 0) + 1;
              int hi = (max / tasks) * (t + 1) + 0;
              // printf ("%d: %d..%d\n", omp_get_thread_num(), lo, hi);
              for (int i = lo; i <= hi; i++)
                local_sum = local_sum + i;
              #pragma omp atomic
                sum = sum + local_sum;
            }
        }
    }
  printf ("%d\n", sum);
  return 0;
}
```

Listing 3.24 Implementing summation of integers by using a fixed number of tasks.

finish before the `parallel` region can end. Hence, once the global sum is printed out in line 26 of Listing 3.24, all tasks has already finished.

The difference between the approaches taken in the previous and this section can be told in yet another way. Namely, when iterations of a single parallel `for` loop are distributed among threads, tasks, one per thread, are created implicitly. But when a number of explicit tasks is used, the loop itself is split among tasks that are then distributed among threads.

Example 3.7 Fibonacci numbers

Computing the first *max* Fibonacci numbers using the time-consuming recursive formula can be pretty naive, especially if a separate call of the recursive function is used for each of them. Nevertheless, it shows how to use the advantage of tasks.

The program in Listing 3.25 shows how this can be done. Note again that a single thread within a parallel region starts all tasks, one per each number. As the program is written, smaller Fibonacci numbers, i.e., for $n = 1, 2, \ldots$, are computed first while the largest are left to be computed later.

The time needed to compute the n-th Fibonacci number using function `fib` in line 4 of Listing 3.25 is of order $O(1.6^n)$. Hence, the time complexity of individual tasks grows exponentially with n. Therefore, it is perhaps better to create (and thus carry out) tasks in reverse order, the most demanding first and the least demanding last as shown in Listing 3.26. Run both programs, check this hypothesis out and investigate which tasks get carried out by which thread. □

OpenMP: tasks

A task is declared using the directive

`#pragma omp task` *[clause [[,] clause] …]*
structured-block

The `task` directive creates a new task that executes *structured-block*. The new task can be executed immediately or can be deferred. A deferred task can be later executed by any thread in the team.

The `task` directive can be further refined by a number of clauses, the most important being the following ones:

- `final(`*scalar-logical-expression*`)` causes, if *scalar-logical-expression* evaluates to *true*, that the created task does not generate any new tasks any more, i.e., the code of would-be-generated new subtasks is included in and thus executed within this task;
- `if(`*[*`task:`*]scalar-logical-expression*`)` causes, if *scalar-logical-expression* evaluates to *false*, that an undeferred task is created, i.e., the created task suspends the creating task until the created task is finished.

For other clauses see OpenMP specification.

OpenMP: limiting execution to a single thread

Within a `parallel` section, the directive

`#pragma omp single` *[clause [[,] clause] …]*
structured-block

causes *structured-block* to be executed by exactly one thread in a team (not necessarily the master thread). If not specified otherwise, all other threads wait idle at the implicit barrier at the end of the `single` directive.

The most important clauses are the following:

- `private(`*list*`)` specifies that each variable in the *list* is private to the code executed within the `single` directive;
- `nowait` removes the implicit barrier at the end of the `single` directive and thus allows other threads in the team to proceed without waiting for the code under the `single` directive to finish.

Converting a parallel `for` loop into a set of tasks is not very interesting and in most cases does not help either. The real power of tasks, however, can be appreciated when the number and the size of individual tasks cannot be known in advance. In

```
#include <stdio.h>
#include <omp.h>

long fib (int n) { return (n < 2 ? 1 : fib (n - 1) + fib (n - 2)); }

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  #pragma omp parallel
    #pragma omp single
      for (int n = 1; n <= max; n++)
        #pragma omp task
          printf ("%d: %d %ld\n", omp_get_thread_num(), n, fib (n));
  return 0;
}
```

Listing 3.25 Computing Fibonacci numbers using OpenMP's tasks: smaller tasks, i.e., for smaller Fibonacci numbers are created first.

other words, when the problem or the algorithm demands that tasks are created dynamically.

```
      for (int n = max; n >= 1; n--)
        #pragma omp task
          printf ("%d: %d\n", omp_get_thread_num(), n);
```

Listing 3.26 Computing Fibonacci numbers using OpenMP's tasks: smaller tasks, i.e., for smaller Fibonacci numbers are created last.

Example 3.8 Quicksort

A good and simple, yet well-known example of this kind is sorting using the Quicksort algorithm [5]. The parallel version using tasks is shown in Listing 3.27.

```
void par_qsort (char **data, int lo, int hi,
                int (*compare)(const char *, const char*)) {
  if (lo > hi) return;
  int l = lo;
  int h = hi;
  char *p = data[(hi + lo) / 2];
  while (l <= h) {
    while (compare (data[l], p) < 0) l++;
    while (compare (data[h], p) > 0) h--;
    if (l <= h) {
      char *tmp = data[l]; data[l] = data[h]; data[h] = tmp;
      l++; h--;
    }
  }
  #pragma omp task final(h - lo < 1000)
    par_qsort (data, lo, h, compare);
  #pragma omp task final(hi - l < 1000)
    par_qsort (data, l, hi, compare);
}
```

Listing 3.27 The parallel implementation of the Quicksort algorithm where each recursive call is performed as a new task.

The partition part of the algorithm, implemented in lines 4–14 of Listing 3.27, is the same as in the sequential version. The recursive calls, though, are modified

because they can be performed independently, i.e., at the same time. Each of the two recursive calls is therefore executed as its own task.

However, no matter how efficient creating new tasks is, it takes time. Creating a new task only makes sense if a part of the table that must be sorted using a recursive call is big enough. In Listing 3.27, the clause `final` in lines 15 and 17 is used to prevent creating new tasks for parts of table that contain less than 1000 elements. The threshold 1000 has been chosen by experience; choosing the best threshold depends on many factors (the number of elements, the time needed to compare two elements, the implementation of OpenMP's tasks, …). The experimental way of choosing it shall be, to some extent, covered in the forthcoming chapters.

There is an analogy with the sequential algorithm: recursion takes time as well and to speed up the sequential Quicksort algorithm, the insertion sort is used once the number of elements falls below a certain threshold.

There should be no confusion about the arguments for function `par_qsort` . However, function `par_qsort` must be called within a `parallel` region by exactly one thread as shown in Listing 3.28.

```
1  #pragma omp parallel
2    #pragma omp single
3      par_qsort (strings, 0, num_strings - 1, compare);
```

Listing 3.28 The call of the parallel implementation of the Quicksort algorithm.

As the Quicksort algorithm itself is rather efficient, i.e., it runs in time $O(n \log n)$, a sufficient number of elements must be used to see that the parallel version actually outperforms the sequential one. The comparison of running times is summarized in Table 3.1. By comparing the running times of the sequential version with the parallel version running within a single thread, one can estimate the time needed to create and destroy OpenMP's threads.

Using 4 or 8 threads the parallel version is definitely faster, although the speed us consider the Quicksort algorithm up is not proportional to the number of threads used. Note that the partition of the table in lines 4–14 of Listing 3.27 is performed sequentially and recall the Amdahl law. □

3.4.2 Combining the Results of Parallel Tasks

In a number of cases, parallel tasks cannot be left to execute independently of each other and leave its results in some global or shared variable. In such a situation, the programmer must take care of the lifespan of each individual task. The next example illustrates this.

Example 3.9 Quicksort revisited

Let us consider the Quicksort algorithm as an example again and modify it so that it returns the number of element pairs swapped during the partition phases.

Table 3.1 The comparison of the running time of the sequential and parallel version of the Quicksort algorithm when sorting n random strings of max length 64 using a quad-core processor with multithreading

n	SEQ	PAR		
		(1 thread)	(4 threads)	(8 threads)
10^5	0.05 s	0.07 s	0.04 s	0.04 s
10^6	0.79 s	0.99 s	0.44 s	0.32 s
10^7	11.82 s	12.47 s	4.27 s	3.57 s
10^8	201.13 s	218.14 s	71.90 s	61.81 s

Counting swaps during the partition phase in a sequential program is trivial. For instance, as shown in Listing 3.29, three new variables can be introduced, namely `count`, `locount`, and `hicount` that contain the number of swaps in the current partition phase and the total numbers of swaps in recursive calls, respectively. (In the sequential program, this could be done with a single counter, but having three counters instead is more appropriate for the developing of the parallel version.)

```
1  int par_qsort (char **data, int lo, int hi,
2                 int (*compare)(const char *, const char*)) {
3    if (lo > hi) return 0;
4    int l = lo;
5    int h = hi;
6    char *p = data[(hi + lo) / 2];
7    int count = 0;
8    while (l <= h) {
9      while (compare (data[l], p) < 0) l++;
10     while (compare (data[h], p) > 0) h--;
11     if (l <= h) {
12       count++;
13       char *tmp = data[l]; data[l] = data[h]; data[h] = tmp;
14       l++; h--;
15     }
16   }
17   int locount, hicount;
18   #pragma omp task shared(locount) final(h - lo < 1000)
19     locount = par_qsort (data, lo, h, compare);
20   #pragma omp task shared(hicount) final(hi - l < 1000)
21     hicount = par_qsort (data, l, hi, compare);
22   #pragma omp taskwait
23   return count + locount + hicount;
24 }
```

Listing 3.29 The call of the parallel implementation of the Quicksort algorithm.

In the parallel version, the modification is not much harder, but a few things must be taken care of. First, as recursive calls in lines 16 and 18 of Listing 3.27 change to assignment statements in lines 19 in 21 of Listing 3.29, the values of variables `locount` and `hicount` are set in two newly created tasks and must, therefore, be shared among the creating and the created tasks. This is achieved using `shared` clause in lines 18 and 20.

Second, remember that once the new tasks in lines 18–19 and 20–21 are created, the task that has just created them continues. To prevent it from computing the sum of all three counters and returning the result when variables `locount` and `hicount` might not have been set yet, the `taskwait` directive is used. It represents an explicit barrier: all tasks created by the task executing it must finish before that task can continue.

At the end of the parallel section is an implicit barrier before all tasks created within the parallel section must finish just like all iterations of a parallel loop must. Hence, in Listing 3.24, there is no need for an explicit barrier using `taskwait`. □

OpenMP: explicit task barrier

An explicit task barrier is created by the following directive:

```
#pragma omp taskwait
```

It specifies a point in the program the task waits until all its subtasks are finished.

3.5 Exercises and Mini Projects

Exercises

1. Modify the program in Listing 3.1 so that it uses a team of 5 threads within the parallel region by default. Investigate how shell variables `OMP_NUM_THREADS` and `OMP_THREAD_LIMIT` influence the execution of the original and modified program.
2. If run with one thread per logical core, threads started by the program in Listings 3.1 print out their thread numbers in random order while threads started by the program in Listing 3.2 always print out their results in the same order. Explain why.
3. Suppose two 100×100 matrices are to be multiplied using 8 threads. How many dot products, i.e., operations performed by the innermost `for` loop, must each thread compute if different approaches to parallelizing the two outermost `for` loops of matrix multiplication illustrated in Fig. 3.6 are used?
4. Draw a 3D graph with the size of the square matrix along one independent axis, e.g., from 1 to 100, and the number of available threads, e.g., from 1 to 16, along the other showing the ratio between the number of dot products computed by the most and the least loaded thread for different approaches to parallelizing the two outermost `for` loops of matrix multiplication illustrated in Fig. 3.6.

5. Modify the programs for matrix multiplication based on different loop parallelization methods to compute $C = A \cdot B^{\mathrm{T}}$ instead of $C = A \cdot B$. Compare the running time of the original and modified programs.
6. Suppose 4 threads are being used when the program in Listing 3.20 and $max = 20$. Determine which iteration will be performed by which thread if `static,1`, `static,2` or `static,3` is used as a iteration scheduling strategy. Try without running the program first. (Assume that iterations 1, 2 and 3 require 2, 3 and 4 units of time while all other iterations require just 1 unit of time.)
7. Suppose 4 threads are being used when the program in Listing 3.20 and $max = 20$. Determine which iteration will be performed by which thread if `dynamic,1`, `dynamic,2` or `dynamic,3` is used as a iteration scheduling strategy. Is the solution uniquely defined? (Assume that iterations 1, 2 and 3 require 2, 3 and 4 units of time while all other iterations require just 1 unit of time.)
8. Modify lines 12 and 13 in Listing 3.22 so that the program works correctly even if T, the number of threads, does not divide max. The number of iterations of the `for` loop in lines 15 and 16 should not differ by more than 1 for any two threads.
9. Modify the program in Listing 3.22 so that the modified program implements `static,c` iteration scheduling strategy instead of `static` as is the case in Listing 3.22. The chunk size `c` must be a constant declared in the program.
10. Modify the program in Listing 3.22 so that the modified program implements `dynamic,c` iteration scheduling strategy instead of `static` as is the case in Listing 3.22. The chunk size `c` must be a constant declared in the program.
    Hint: Use a shared counter of iterations that functions as a queue of not yet scheduled iterations outside the parallel section.
11. While computing the sum of all elements of `sums` in Listing 3.23, the program creates new threads within every iteration of the outer loop. Rewrite the code so that creation of new threads in every iteration of the outer loop is avoided.
12. Try rewriting the programs in Listings 3.25 and 3.26 using parallel `for` loops instead of OpenMP's tasks to mimic the behavior of the original program as close as possible. Find out which iteration scheduling strategy should be used. Compare the running time of programs using parallel `for` loops with those that use OpenMP's tasks.
13. Modify the program in Listing 3.27 so that it does not use `final` but works in the same way.
14. Check the OpenMP specification and rewrite the program in Listing 3.24 using the `taskloop` directive.

Mini Projects

P1. Write a multi-core program that uses CYK algorithm [13] to parse a string of symbols. The inputs are a context-free grammar G in Chomsky Normal Form and a string of symbols. At the end, the program should print YES if the string of symbols can be derived by the rules of the grammar and NO otherwise.

Write a sequential program (no OpenMP directives at all) as well. Compare the running time of the sequential program with the running time of the multi-core program and compute the speedup for different grammars and different string lengths.
Hint: Observe that in the classical formulation of CYK algorithm the iterations of the outermost loop must be performed one after another but that iterations of the second outermost loop are independent and offer a good opportunity for parallelization.

P2. Write a multi-core program for the "all-pairs shortest paths" problem [5]. The input is a weighted graph with no negative cycles and the expected output are lengths of the shortest paths between all pairs of vertices (where the length of a path is a sum of weights along the edges that the path consists of).
Write a sequential program (no OpenMP directives at all) as well. Compare the running time of the sequential program with the running time of the multi-core program and compute the speedup achieved

1. for different number of cores and different number of threads per core, and
2. for different number of vertices and different number of edges.

Hint 1: Take the Bellman–Ford algorithm for all-pairs shortest paths [5] and consider its matrix multiplication formulation. For a graph $G = \langle V, E \rangle$ your program should achieve at least time $O(|V|^4)$, but you can do better and achieve time $O(|V|^3 \log_2 |V|)$. In neither case should you ignore the cache performance: allocate matrices carefully.
Hint 2: Instead of using the Bellman–Ford algorithm, you can try parallelizing the Floyd–Warshall algorithm that runs in time $O(|V|^3)$ [5]. How fast is the program based on the Floyd–Warshall algorithm compared with the one that uses the $O(|V|^4)$ or $O(|V|^3 \log_2 |V|)$ Bellman–Ford algorithm?

3.6 Bibliographic Notes

The primary source of information including all details of OpenMP API is available at OpenMP web site [20] where the complete specification [18] and a collection of examples [19] are available. OpenMP version 4.5 is used in this book as version 5.0 is still being worked on by OpenMP Architecture Review Board. The summary card for C/C++ is also available at OpenMP web site.

As standards and specifications are usually hard to read, one might consider some book wholly dedicated to programming using OpenMP. Although relatively old and thus lacking the most of the modern OpenMP features, the book by Rohit Chandra et al. [4] provides a nice introduction to underlying ideas upon which OpenMP is based upon and the basic OpenMP constructs. A more recent and comprehensive description of OpenMP, version 4.5, can be found in the book by Ruud van der Pas et al. [21].