# Chapter 6

# Shared-memory parallel programming with OpenMP

In the multicore world, one-socket single-core systems have all but vanished except for the embedded market. The price vs. performance "sweet spot" lies mostly in the two-socket regime, with multiple cores (and possibly multiple chips) on a socket. Parallel programming in a basic form should thus start at the shared-memory level, although it is entirely possible to run multiple processes without a concept of shared memory. See Chapter 9 for details on distributed-memory parallel programming.

However, shared-memory programming is not an invention of the multicore era. Systems with multiple (single-core) processors have been around for decades, and appropriate portable programming interfaces, most notably POSIX threads [P9], have been developed in the 1990s. The basic principles, limitations and bottlenecks of shared-memory parallel programming are certainly the same as with any other parallel model (see Chapter 5), although there are some peculiarities which will be covered in Chapter 7. The purpose of the current chapter is to give a nonexhaustive overview of OpenMP, which is the dominant shared-memory programming standard today. OpenMP bindings are defined for the C, C++, and Fortran languages as of the current version of the standard (3.0). Some OpenMP constructs that are mainly used for optimization will be introduced in Chapter 7.

We should add that there are specific solutions for the C++ language like, e.g., Intel Threading Building Blocks (TBB) [P10], which may provide better functionality than OpenMP in some respects. We also deliberately ignore compiler-based automatic shared-memory parallelization because it has up to now not lived up to expectations except for trivial cases.

## 6.1 Short introduction to OpenMP

Shared memory opens the possibility to have immediate access to all data from all processors without explicit communication. Unfortunately, POSIX threads are not a comfortable parallel programming model for most scientific software, which is typically loop-centric. For this reason, a joint effort was made by compiler vendors to establish a standard in this field, called OpenMP [P11]. OpenMP is a set of *compiler directives* that a non-OpenMP-capable compiler would just regard as comments and ignore. Hence, a well-written parallel OpenMP program is also a valid serial program

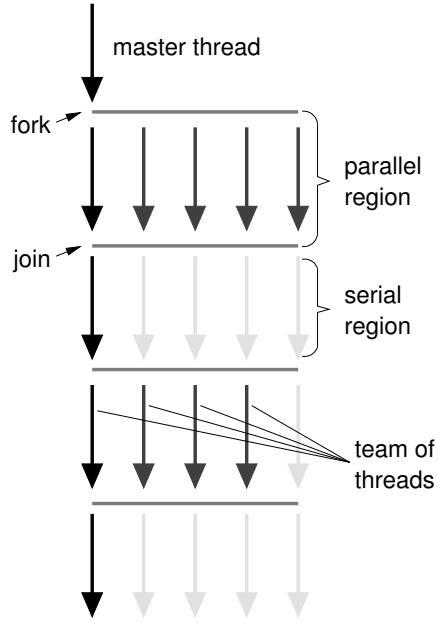


**Figure 6.1:** Model for OpenMP thread operations: The master thread "forks" team of threads, which work on shared memory in a parallel region. After the parallel region, the threads are "joined," i.e., terminated or put to sleep, until the next parallel region starts. The number of running threads may vary among parallel regions.



(this is certainly not a requirement, but it simplifies development and debugging considerably). The central entity in an OpenMP program is not a process but a *thread*. Threads are also called "lightweight processes" because several of them can share a common address space and mutually access data. Spawning a thread is much less costly than forking a new process, because threads share everything but instruction pointer (the address of the next instruction to be executed), stack pointer and register state. Each thread can, by means of its local stack pointer, also have "private" variables, but since all data is accessible via the common address space, it is only a matter of taking the address of an item to make it accessible to all other threads as well. However, the OpenMP standard actually *forbids* making a private object available to other threads via its address. It will become clear later that this is actually a good idea.

We will concentrate on the Fortran interface for OpenMP here, and point out important differences to the C/C++ bindings as appropriate.

### 6.1.1 Parallel execution

In any OpenMP program, a single thread, the *master thread*, runs immediately after startup. Truly parallel execution happens inside *parallel regions*, of which an arbitrary number can exist in a program. Between two parallel regions, no thread except the master thread executes any code. This is also called the "fork-join model" (see Figure 6.1). Inside a parallel region, a *team of threads* executes instruction streams concurrently. The number of threads in a team may vary among parallel regions.

OpenMP is a layer that adapts the raw OS thread interface to make it more us-

able with the typical structures that numerical software tends to employ. In practice, parallel regions in Fortran are initiated by `!$OMP PARALLEL` and ended by `!$OMP END PARALLEL` directives, respectively. The `!$OMP` string is a so-called *sentinel*, which starts an OpenMP directive (in C/C++, `#pragma omp` is used instead). Inside a parallel region, each thread carries a unique identifier, its *thread ID*, which runs from zero to the number of threads minus one, and can be obtained by the `omp_get_thread_num()` API function:

```
1   use omp_lib      ! module with API declarations
2
3   print *,'I am the master, and I am alone'
4  !$OMP PARALLEL
5   call do_work_package(omp_get_thread_num(),omp_get_num_threads())
6  !$OMP END PARALLEL
```

The `omp_get_num_threads()` function returns the number of active threads in the current parallel region. The `omp_lib` module contains the API definitions (in Fortran 77 and C/C++ there are include files `mpif.h` and `omp.h`, respectively). Code between `OMP PARALLEL` and `OMP END PARALLEL`, including subroutine calls, is executed by every thread. In the simplest case, the thread ID can be used to distinguish the tasks to be performed on different threads; this is done by calling the `do_work_package()` subroutine in above example with the thread ID and the overall number of threads as parameters. Using OpenMP in this way is mostly equivalent to the POSIX threads programming model.

An important difference between the Fortran and C/C++ OpenMP bindings must be stressed here. In C/C++, there is no `end parallel` directive, because all directives apply to the following statement or structured block. The example above would thus look like this in C++:

```
1   #include <omp.h>
2
3   std::cout << "I am the master, and I am alone";
4  #pragma omp parallel
5  {
6   do_work_package(omp_get_thread_num(),omp_get_num_threads());
7  }
```

The curly braces could actually be omitted in this particular case, but the fact that a structured block is subject to parallel execution has consequences for data scoping (see below).

The actual number of running threads does not have to be known at compile time. It can be set by the environment variable prior to running the executable:

```
1 $ export OMP_NUM_THREADS=4
2 $ ./a.out
```

Although there are also means to set or alter the number of running threads under program control, an OpenMP program should always be written so that it does not assume a specific number of threads.

**Listing 6.1:** "Manual" loop parallelization and variable privatization. Note that this is *not* the intended mode for OpenMP.

```
1   integer :: bstart, bend, blen, numth, tid, i
2   integer :: N
3   double precision, dimension(N) :: a,b,c
4   ...
5  !$OMP PARALLEL PRIVATE(bstart,bend,blen,numth,tid,i)
6   numth = omp_get_num_threads()
7   tid = omp_get_thread_num()
8   blen = N/numth
9   if(tid.lt.mod(N,numth)) then
10    blen = blen + 1
11    bstart = blen * tid + 1
12  else
13    bstart = blen * tid + mod(N,numth) + 1
14  endif
15  bend = bstart + blen - 1
16  do i = bstart,bend
17    a(i) = b(i) + c(i)
18  enddo
19 !$OMP END PARALLEL
```

### 6.1.2 Data scoping

Any variables that existed before a parallel region still exist inside, and are by default shared between all threads. True work sharing, however, makes sense only if each thread can have its own, *private* variables. OpenMP supports this concept by defining a separate stack for every thread. There are three ways to make private variables:

1. A variable that exists before entry to a parallel construct can be privatized, i.e., made available as a private instance for every thread, by a `PRIVATE` clause to the `OMP PARALLEL` directive. The private variable's scope extends until the end of the parallel construct.

2. The index variable of a worksharing loop (see next section) is automatically made private.

3. Local variables in a subroutine called from a parallel region are private to each calling thread. This pertains also to copies of actual arguments generated by the call-by-value semantics, and to variables declared inside structured blocks in C/C++. However, local variables carrying the `SAVE` attribute in Fortran (or the `static` storage class in C/C++) will be shared.

Shared variables that are not modified in the parallel region do not have to be made private.

A simple loop that adds two arrays could thus be parallelized as shown in Listing 6.1. The actual loop is contained in lines 16–18, and everything before that is

just for calculating the loop bounds for each thread. In line 5 the `PRIVATE` clause to the `PARALLEL` directive privatizes all specified variables, i.e., each thread gets its own instance of each variable on its local stack, with an undefined initial value (C++ objects will be instantiated using the default constructor). Using `FIRSTPRIVATE` instead of `PRIVATE` would initialize the privatize instances with the contents of the shared instance (in C++, the copy constructor is employed). After the parallel region, the original values of the privatized variables are retained if they are not modified on purpose. Note that there are separate clauses (`THREADPRIVATE` and `COPYIN`, respectively [P11]) for privatization of global or static data (`SAVE` variables, common block elements, `static` variables).

In C/C++, there is actually less need for using the `private` clause in many cases, because the `parallel` directive applies to a structured block. Instead of privatizing shared instances, one can simply declare local variables:

```
1 #pragma omp parallel
2 {
3   int bstart, bend, blen, numth, tid, i;
4   ...              // calculate loop boundaries
5   for(i=bstart; i<=bend; ++i)
6     a[i] = b[i] + c[i];
7 }
```

Manual loop parallelization as shown here is certainly not the intended mode of operation in OpenMP. The standard defines much more advanced means of distributing work among threads (see below).

### 6.1.3 OpenMP worksharing for loops

Being the omnipresent programming structure in scientific codes, loops are natural candidates for parallelization if individual iterations are independent. This corresponds to the medium-grained data parallelism described in Section 5.2.1. As an example, consider a parallel version of a simple program for the calculation of $\pi$ by integration:

$$\pi = \int_0^1 \mathrm{d}x \, \frac{4}{1+x^2} \tag{6.1}$$

Listing 6.2 shows a possible implementation. In contrast to the previous examples, this is also valid serial code. The initial value of `sum` is copied to the private instances via the `FIRSTPRIVATE` clause on the `PARALLEL` directive. Then, a `DO` directive in front of a `do` loop starts a *worksharing* construct: The iterations of the loop are distributed among the threads (which are running because we are in a parallel region). Each thread gets its own iteration space, i.e., is assigned to a different set of `i` values. How threads are mapped to iterations is implementation-dependent by default, but can be influenced by the programmer (see Section 6.1.6 below). Although shared in the enclosing parallel region, the loop counter `i` is privatized automatically. The final `END DO` directive after the loop is not strictly necessary here, but may be required in cases where the `NOWAIT` clause is specified; see Section 7.2.1 on page 170 for

**Listing 6.2:** A simple program for numerical integration of a function in OpenMP.

```
1   double precision :: pi,w,sum,x
2   integer :: i,N=1000000
3
4   pi = 0.d0
5   w = 1.d0/N
6   sum = 0.d0
7  !$OMP PARALLEL PRIVATE(x) FIRSTPRIVATE(sum)
8  !$OMP DO
9     do i=1,n
10      x = w*(i-0.5d0)
11      sum = sum + 4.d0/(1.d0+x*x)
12    enddo
13 !$OMP END DO
14 !$OMP CRITICAL
15    pi= pi + w*sum
16 !$OMP END CRITICAL
17 !$OMP END PARALLEL
```



details. A `DO` directive must be followed by a `do` loop, and applies to this loop only. In C/C++, the `for` directive serves the same purpose. Loop counters are restricted to integers (signed or unsigned), pointers, or random access iterators.

In a parallel loop, each thread executes "its" share of the loop's iteration space, accumulating into its private `sum` variable (line 11). After the loop, and still inside the parallel region, the partial sums must be added to get the final result (line 15), because the private instances of `sum` will be gone once the region is left. There is a problem, however: Without any countermeasures, threads would write to the result variable `pi` concurrently. The result would depend on the exact order the threads access `pi`, and it would most probably be wrong. This is called a *race condition*, and the next section will explain what one can do to prevent it.

Loop worksharing works even if the parallel loop itself resides in a subroutine called from the enclosing parallel region. The `DO` directive is then called *orphaned*, because it is outside the lexical extent of a parallel region. If such a directive is encountered while no parallel region is active, the loop will not be workshared.

Finally, if a separation of the parallel region from the workshared loop is not required, the two directives can be combined:

```
1  !$OMP PARALLEL DO
2    do i=1,N
3      a(i) = b(i) + c(i) * d(i)
4    enddo
5  !$OMP END PARALLEL DO
```

The set of clauses allowed for this *combined parallel worksharing directive* is the union of all clauses allowed on each directive separately.

### 6.1.4 Synchronization

#### Critical regions

Concurrent write access to a shared variable or, in more general terms, a shared resource, must be avoided by all means to circumvent race conditions. *Critical regions* solve this problem by making sure that at most one thread at a time executes some piece of code. If a thread is executing code inside a critical region, and another thread wants to enter, the latter must wait (block) until the former has left the region. In the integration example (Listing 6.2), the `CRITICAL` and `END CRITICAL` directives (lines 14 and 16) bracket the update to `pi` so that the result is always correct. Note that the order in which threads enter the critical region is undefined, and can change from run to run. Consequently, the definition of a "correct result" must encompass the possibility that the partial sums are accumulated in a random order, and the usual reservations regarding floating-point accuracy do apply [135]. (If strong sequential equivalence, i.e., bitwise identical results compared to a serial code is required, OpenMP provides a possible solution with the `ORDERED` construct, which we do not cover here.)

Critical regions hold the danger of *deadlocks* when used inappropriately. A deadlock arises when one or more "agents" (threads in this case) wait for resources that will never become available, a situation that is easily generated with badly arranged `CRITICAL` directives. When a thread encounters a `CRITICAL` directive inside a critical region, it will block forever. Since this could happen in a deeply nested subroutine, deadlocks are sometimes hard to pin down.

OpenMP has a simple solution for this problem: A critical region may be given a *name* that distinguishes it from others. The name is specified in parentheses after the `CRITICAL` directive:

```
1  !$OMP PARALLEL DO PRIVATE(x)
2    do i=1,N
3      x = SIN(2*PI*x/N)
4  !$OMP CRITICAL (psum)
5      sum = sum + func(x)
6  !$OMP END CRITICAL (psum)
7    enddo
8  !$OMP END PARALLEL DO
9    ...
10   double precision func(v)
11   double precision :: v
12 !$OMP CRITICAL (prand)
13   func = v + random_func()
14 !$OMP END CRITICAL (prand)
15   END SUBROUTINE func
```

The update to `sum` in line 5 is protected by a critical region. In subroutine `func()` there is another critical region because it is not allowed to call `random_func()` (line 13) by more than one thread at a time; it probably contains a random seed with a `SAVE` attribute. Such a function is not *thread safe*, i.e., its concurrent execution would incur a race condition.

Without the names on the two different critical regions, this code would deadlock because a thread that has just called `func()`, already in a critical region, would immediately encounter the second critical region and wait for itself indefinitely to free the resource. With the names, the second critical region is understood to protect a different resource than the first.

A disadvantage of named critical regions is that the names are unique identifiers. It is not possible to have them indexed by an integer variable, for instance. There are OpenMP API functions that support the use of *locks* for protecting shared resources. The advantage of locks is that they are ordinary variables that can be arranged as arrays or in structures. That way it is possible to protect each single element of an array of resources individually, even if their number is not known at compile time. See Section 7.2.3 for an example.



**Barriers**

If, at a certain point in the parallel execution, it is necessary to synchronize *all* threads, a `BARRIER` can be used:

```
1 !$OMP BARRIER
```

The barrier is a *synchronization point*, which guarantees that all threads have reached it before any thread goes on executing the code below it. Certainly it must be ensured that every thread hits the barrier, or a deadlock may occur.

Barriers should be used with caution in OpenMP programs, partly because of their potential to cause deadlocks, but also due to their performance impact (synchronization is overhead). Note also that every parallel region executes an implicit barrier at its end, which cannot be removed. There is also a default implicit barrier at the end of worksharing loops and some other constructs to prevent race conditions. It can be eliminated by specifying the `NOWAIT` clause. See Section 7.2.1 for details.

### 6.1.5 Reductions

The example in Listing 6.3 shows a loop code that adds some random noise to the elements of an array `a()` and calculates its vector norm. The `RANDOM_NUMBER()` subroutine may be assumed to be thread safe, according to the OpenMP standard.

Similar to the integration code in Listing 6.2, the loop implements a *reduction* operation: Many contributions (the updated elements of `a()`) are accumulated into a single variable. We have previously solved this problem with a critical region, but OpenMP provides a more elegant alternative by supporting reductions directly via the `REDUCTION` clause (end of line 5). It automatically privatizes the specified variable(s) (`s` in this case) and initializes the private instances with a sensible starting value. At the end of the construct, all partial results are accumulated into the shared instance of `s`, using the specified operator (+ here) to get the final result.

There is a set of supported operators for OpenMP reductions (slightly different for Fortran and C/C++), which cannot be extended. C++ overloaded operators are not allowed. However, the most common cases (addition, subtraction, multiplication,

**Listing 6.3:** Example with reduction clause for adding noise to the elements of an array and calculating its vector norm.

```
1   double precision :: r,s
2   double precision, dimension(N) :: a
3
4   call RANDOM_SEED()
5  !$OMP PARALLEL DO PRIVATE(r) REDUCTION(+:s)
6   do i=1,N
7     call RANDOM_NUMBER(r)  ! thread safe
8     a(i) = a(i) + func(r)  ! func() is thread safe
9     s = s + a(i) * a(i)
10  enddo
11 !$OMP END PARALLEL DO
12
13  print *,'Sum = ',s
```

logical, etc.) are covered. If a required operator is not available, one must revert to the "manual" method as shown in the Listing 6.2.

Note that the automatic initialization for reduction variables, though convenient, bears the danger of producing invalid serial, i.e., non-OpenMP code. Compiling the example above without OpenMP support will leave `s` uninitialized.

### 6.1.6 Loop scheduling

As mentioned earlier, the mapping of loop iterations to threads is configurable. It can be controlled by the argument of a `SCHEDULE` clause to the loop worksharing directive:

```
1 !$OMP DO SCHEDULE(STATIC)
2   do i=1,N
3     a(i) = calculate(i)
4   enddo
5 !$OMP END DO
```

The simplest possibility is `STATIC`, which divides the loop into contiguous chunks of (roughly) equal size. Each thread then executes on exactly one chunk. If for some reason the amount of work per loop iteration is not constant but, e.g., decreases with loop index, this strategy is suboptimal because different threads will get vastly different workloads, which leads to load imbalance. One solution would be to use a *chunksize* like in "`STATIC,1`," dictating that chunks of size 1 should be distributed across threads in a round-robin manner. The chunksize may not only be a constant but any valid integer-valued expression.

There are alternatives to the static schedule for other types of workload (see Figure 6.2). *Dynamic* scheduling assigns a chunk of work, whose size is defined by the chunksize, to the next thread that has finished its current chunk. This allows for a very flexible distribution which is usually not reproduced from run to run. Threads that get assigned "easier" chunks will end up completing more of them, and load

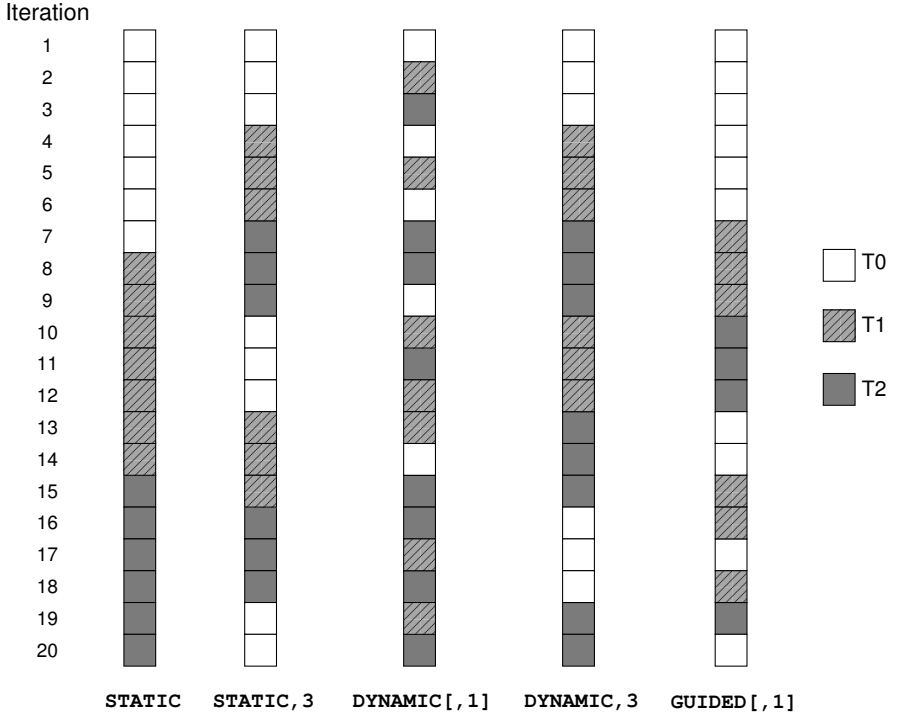


**Figure 6.2:** Loop schedules in OpenMP. The example loop has 20 iterations and is executed by three threads (T0, T1, T2). The default chunksize for `DYNAMIC` and `GUIDED` is one. If a chunksize is specified, the last chunk may be shorter. Note that only the `STATIC` schedules guarantee that the distribution of chunks among threads stays the same from run to run.



imbalance is greatly reduced. The downside is that dynamic scheduling generates significant overhead if the chunks are too small in terms of execution time (see Section 7.2.1 for an assessment of scheduling overhead). This is why it is often desirable to use a moderately large chunksize on tight loops, which in turn leads to more load imbalance. In cases where this is a problem, the *guided* schedule may help. Again, threads request new chunks dynamically, but the chunksize is always proportional to the remaining number of iterations divided by the number of threads. The smallest chunksize is specified in the schedule clause (default is 1). Despite the dynamic assignment of chunks, scheduling overhead is kept under control. However, a word of caution is in order regarding dynamic and guided schedules: Due to the indeterministic nature of the assignment of threads to chunks, applications that are limited by memory bandwidth may suffer from insufficient access locality on ccNUMA systems (see Section 4.2.3 for an introduction to ccNUMA architecture and Chapter 8 for ccNUMA-specific performance effects and optimization methods). The static schedule is thus the only choice under such circumstances, if the standard worksharing

directives are used. Of course there is also the possibility of "explicit" scheduling, using the thread ID number to assign work to threads as shown in Section 6.1.2.

For debugging and profiling purposes, OpenMP provides a facility to determine the loop scheduling at runtime. If the scheduling clause specifies "RUNTIME," the loop is scheduled according to the contents of the OMP_SCHEDULE shell variable. However, there is no way to set different schedulings for different loops that use the SCHEDULE(RUNTIME) clause.

### 6.1.7 Tasking

In early versions of the standard, parallel worksharing in OpenMP was mainly concerned with loop structures. However, not all parallel work comes in loops; a typical example is a linear list of objects (probably arranged in a std::list<> STL container), which should be processed in parallel. Since a list is not easily addressable by an integer index or a random-access iterator, a loop worksharing construct is ruled out, or could only be used with considerable programming effort.

OpenMP 3.0 provides the *task* concept to circumvent this limitation. A task is defined by the TASK directive, and contains code to be executed.[1] When a thread encounters a task construct, it may execute it right away or set up the appropriate data environment and defer its execution. The task is then ready to be executed later by any thread of the team.

As a simple example, consider a loop in which some function must be called for each loop index with some probability:

```
1    integer i,N=1000000
2    type(object), dimension(N) :: p
3    double precision :: r
4    ...
5  !$OMP PARALLEL PRIVATE(r,i)
6  !$OMP SINGLE
7    do i=1,N
8      call RANDOM_NUMBER(r)
9      if(p(i)%weight > r) then
10 !$OMP TASK
11       ! i is automatically firstprivate
12       ! p() is shared
13       call do_work_with(p(i))
14 !$OMP END TASK
15     endif
16   enddo
17 !$OMP END SINGLE
18 !$OMP END PARALLEL
```

The actual number of calls to do_work_with() is unknown, so tasking is a natural choice here. A do loop over all elements of p() is executed in a SINGLE region (lines 6–17). A SINGLE region will be entered by one thread only, namely the one that reaches the SINGLE directive first. All others skip the code until the

[1] In OpenMP terminology, "task" is actually a more general term; the definition given here is sufficient for our purpose.

`END SINGLE` directive and wait there in an implicit barrier. With a probability determined by the current object's content, a `TASK` construct is entered. One task consists in the call to `do_work_with()` (line 13) together with the appropriate data environment, which comprises the array of types `p()` and the index `i`. Of course, the index is unique for each task, so it should actually be subject to a `FIRSTPRIVATE` clause. OpenMP specifies that variables that are private in the enclosing context are automatically made `FIRSTPRIVATE` inside the task, while shared data stays shared (except if an additional data scoping clause is present). This is exactly what we want here, so no additional clause is required.

All the tasks generated by the thread in the `SINGLE` region are subject to dynamic execution by the thread team. Actually, the generating thread may also be forced to suspend execution of the loop at the `TASK` construct (which is one example of a *task scheduling point*) in order to participate in running queued tasks. This can happen when the (implementation-dependent) internal limit of queued tasks is reached. After some tasks have been run, the generating thread will return to the loop. Note that there are complexities involved in task scheduling that our simple example cannot fathom; multiple threads can generate tasks concurrently, and tasks can be declared *untied* so that a different thread may take up execution at a task scheduling point. The OpenMP standard provides excessive examples.

Task parallelism with its indeterministic execution poses the same problems for ccNUMA access locality as dynamic or guided loop scheduling. Programming techniques to ameliorate these difficulties do exist [O58], but their applicability is limited.

### 6.1.8 Miscellaneous

#### Conditional compilation

In some cases it may be useful to write different code depending on OpenMP being enabled or not. The directives themselves are no problem here because they will be ignored gracefully. Beyond this default behavior one may want to mask out, e.g., calls to API functions or any code that makes no sense without OpenMP enabled. This is supported in C/C++ by the preprocessor symbol `_OPENMP`, which is defined only if OpenMP is available. In Fortran the special sentinel "`!$`" acts as a comment only if OpenMP is not enabled (see Listing 6.4).

#### Memory consistency

In the code shown in Listing 6.4, the second API call (line 8) is located in a `SINGLE` region. This is done because `numthreads` is global and should be written to only by one thread. In the critical region each thread just prints a message, but a necessary requirement for the `numthreads` variable to have the updated value is that no thread leaves the `SINGLE` region before the update has been "promoted" to memory. The `END SINGLE` directive acts as an implicit barrier, i.e., no thread can continue executing code before all threads have reached the same point. The OpenMP memory model ensures that barriers enforce memory consistency: Variables that have been held in registers are written out so that cache coherence can make sure that

**Listing 6.4:** Fortran sentinels and conditional compilation with OpenMP combined.

```
1  !$ use omp_lib
2    myid=0
3    numthreads=1
4  #ifdef _OPENMP
5  !$OMP PARALLEL PRIVATE(myid)
6    myid = omp_get_thread_num()
7  !$OMP SINGLE
8    numthreads = omp_get_num_threads()
9  !$OMP END SINGLE
10 !$OMP CRITICAL
11   write(*,*) 'Parallel program - this is thread ',myid,&
12                                        ' of ',numthreads
13 !$OMP END CRITICAL
14 !$OMP END PARALLEL
15 #else
16   write(*,*) 'Serial program'
17 #endif
```



all caches get updated values. This can also be initiated under program control via the `FLUSH` directive, but most OpenMP worksharing and synchronization constructs perform implicit barriers, and hence flushes, at the end.

Note that compiler optimizations can prevent modified variable contents to be seen by other threads immediately. If in doubt, use the `FLUSH` directive or declare the variable as `volatile` (only available in C/C++ and Fortran 2003).

### Thread safety

The `write` statement in line 11 is serialized (i.e., protected by a critical region) so that its output does not get clobbered when multiple threads write to the console. As a general rule, I/O operations and general OS functionality, but also common library functions should be serialized because they may not be thread safe. A prominent example is the `rand()` function from the C library, as it uses a static variable to store its hidden state (the seed).

### Affinity

One should note that the OpenMP standard gives no hints as to how threads are to be bound to the cores in a system, and there are no provisions for implementing locality constraints. One cannot rely at all on the OS to make a good choice regarding placement of threads, so it makes sense (especially on multicore architectures and ccNUMA systems) to use OS-level tools, compiler support or library functions to explicitly pin threads to cores. See Appendix A for technical details.

**Environment variables**

Some aspects of OpenMP program execution can be influenced by environment variables. `OMP_NUM_THREADS` and `OMP_SCHEDULE` have already been described above.

Concerning thread-local variables, one must keep in mind that usually the OS shell restricts the maximum size of all stack variables of its processes, and there may also be a system limit on each thread's stack size. This limit can be adjusted via the `OMP_STACKSIZE` environment variable. Setting it to, e.g., "`100M`" will set a stack size of 100 MB per thread (excluding the initial program thread, whose stack size is still set by the shell). Stack overflows are a frequent source of problems with OpenMP programs.

The OpenMP standard allows for the number of active threads to dynamically change between parallel regions in order to adapt to available system resources (dynamic thread number adjustment). This feature can be switched on or off by setting the `OMP_DYNAMIC` environment variable to `true` or `false`, respectively. It is unspecified what the OpenMP runtime implements as the default.



## 6.2 Case study: OpenMP-parallel Jacobi algorithm

The Jacobi algorithm studied in Section 3.3 can be parallelized in a straightforward way. We add a slight modification, however: A sensible convergence criterion shall ensure that the code actually produces a converged result. To do this we introduce a new variable `maxdelta`, which stores the maximum absolute difference over all lattice sites between the values before and after each sweep (see Listing 6.5). If `maxdelta` drops below some threshold `eps`, convergence is reached.

Fortunately the OpenMP Fortran interface permits using the `MAX()` intrinsic function in `REDUCTION` clauses, which simplifies the convergence check (lines 7 and 15 in Listing 6.5). Figure 6.3 shows performance data for one, two, and four threads on an Intel dual-socket Xeon 5160 3.0 GHz node. In this node, the two cores in a socket share a common 4 MB L2 cache and a frontside bus (FSB) to the chipset. The results exemplify several key aspects of parallel programming in multicore environments:

- With increasing $N$ there is the expected performance breakdown when the working set ($2 \times N^2 \times 8$ bytes) does not fit into cache any more. This breakdown occurs at the same $N$ for single-thread and dual-thread runs if the two threads run in the same L2 group (filled symbols). If the threads run on different sockets (open symbols), this limit is a factor of $\sqrt{2}$ larger because the aggregate cache size is doubled (dashed lines in Figure 6.3). The second breakdown at very large $N$, i.e., when two successive lattice rows exceed the L2 cache size, cannot be seen here as we use a square lattice (see Section 3.3).

- A single thread can saturate a socket's FSB for a memory-bound situation, i.e.,

**Listing 6.5:** OpenMP implementation of the 2D Jacobi algorithm on an $N \times N$ lattice, with a convergence criterion added.

```
1   double precision, dimension(0:N+1,0:N+1,0:1) :: phi
2   double precision :: maxdelta,eps
3   integer :: t0,t1
4   eps = 1.d-14        ! convergence threshold
5   t0 = 0 ; t1 = 1
6   maxdelta = 2.d0*eps
7   do while(maxdelta.gt.eps)
8     maxdelta = 0.d0
9  !$OMP PARALLEL DO REDUCTION(max:maxdelta)
10    do k = 1,N
11      do i = 1,N
12        ! four flops, one store, four loads
13        phi(i,k,t1) = (  phi(i+1,k,t0) + phi(i-1,k,t0)
14                       + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
15        maxdelta = max(maxdelta,abs(phi(i,k,t1)-phi(i,k,t0)))
16      enddo
17    enddo
18 !$OMP END PARALLEL DO
19    ! swap arrays
20    i = t0 ; t0=t1 ; t1=i
21  enddo
```



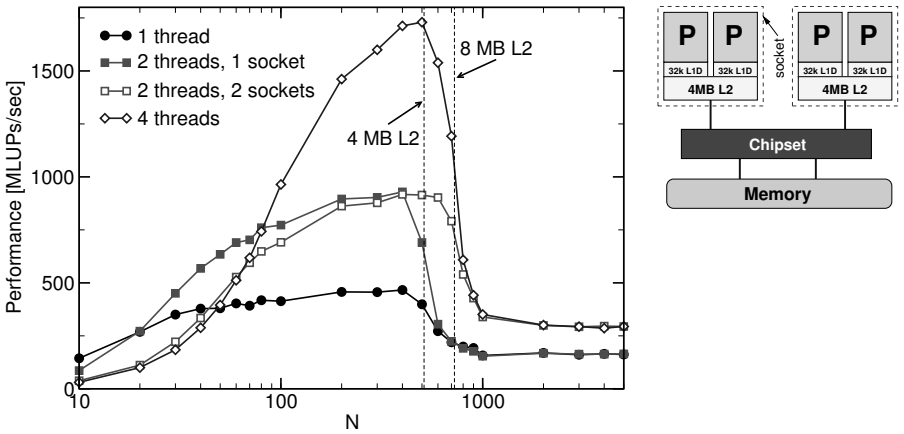


**Figure 6.3:** Performance versus problem size of a 2D Jacobi solver on an $N \times N$ lattice with OpenMP parallelization at one, two, and four threads on an Intel dual-core dual-socket Xeon 5160 node at 3.0 GHz (right). For two threads, there is a choice to place them on one socket (filled squares) or on different sockets (open squares).

at large $N$. Running two threads on the same socket has no benefit whatsoever in this limit because contention will occur on the frontside bus. Adding a second socket gives an 80% boost, as two FSBs become available. Scalability is not perfect because of deficiencies in the chipset and FSB architecture. Note that bandwidth scalability behavior on all memory hierarchy levels is strongly architecture-dependent; there are multicore chips on which it takes two or more threads to saturate the memory interface.

- With two threads, the maximum in-cache performance is the same, no matter whether they run on the same or on different sockets (filled vs. open squares). This indicates that the shared L2 cache can saturate the bandwidth demands of both cores in its group. Note, however, that three of the four loads in the Jacobi kernel are satisfied from L1 cache (see Section 3.3 for an analysis of bandwidth requirements). Performance prediction can be delicate under such conditions [M41, M44].

- At $N < 50$, the location of threads is more important for performance than their number, although the problem fits into the aggregate L1 caches. Using two sockets is roughly a factor of two slower in this case. The reason is that OpenMP overhead like the barrier synchronization at the end of the OpenMP worksharing loop dominates execution time for small $N$. See Section 7.2 for more information on this problem and how to ameliorate its consequences.

Explaining the performance characteristics of this bandwidth-limited algorithm requires a good understanding of the underlying parallel hardware, including issues specific to multicore chips. Future multicore designs will probably be more "anisotropic" (see, e.g., Figure 1.17) and show a richer, multilevel cache group structure, making it harder to understand performance features of parallel codes [M41].



## 6.3 Advanced OpenMP: Wavefront parallelization

Up to now we have only encountered problems where OpenMP parallelization was more or less straightforward because the important loops comprised independent iterations. However, in the presence of loop-carried dependencies, which also inhibit pipelining in some cases (see Section 1.2.3), writing a simple worksharing directive in front of a loop leads to unpredictable results. A typical example is the *Gauss–Seidel* algorithm, which can be used for solving systems of linear equations or boundary value problems, and which is also widely employed as a smoother component in multigrid methods. Listing 6.6 shows a possible serial implementation in three spatial dimensions. Like the Jacobi algorithm introduced in Section 3.3, this code solves for the steady state, but there are no separate arrays for the current and the next time step; a stencil update at $(i, j, k)$ directly re-uses the three neighboring sites with smaller coordinates. As those have been updated in the very same sweep

**Listing 6.6:** A straightforward implementation of the Gauss–Seidel algorithm in three dimensions. The highlighted references cause loop-carried dependencies.

```
1  double precision, parameter :: osth=1/6.d0
2  do it=1,itmax   ! number of iterations (sweeps)
3    ! not parallelizable right away
4    do k=1,kmax
5      do j=1,jmax
6        do i=1,imax
7          phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
8                       + phi(i,j-1,k) + phi(i,j+1,k)
9                       + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
10       enddo
11     enddo
12   enddo
13 enddo
```



before, the Gauss–Seidel algorithm has fundamentally different convergence properties as compared to Jacobi (Stein-Rosenberg Theorem).

Parallelization of the Jacobi algorithm is straightforward (see the previous section) because all updates of a sweep go to a different array, but this is not the case here. Indeed, just writing a `PARALLEL DO` directive in front of the `k` loop would lead to race conditions and yield (wrong) results that most probably vary from run to run.

Still it is possible to parallelize the code with OpenMP. The key idea is to find a way of traversing the lattice that fulfills the dependency constraints imposed by the stencil update. Figures 6.4 and 6.5 show how this can be achieved: Instead of simply cutting the $k$ dimension into chunks to be processed by OpenMP threads, a *wavefront* travels through the lattice in $k$ direction. The dimension along which to parallelize is $j$, and each of the $t$ threads $\mathrm{T}_0 \ldots \mathrm{T}_{t-1}$ gets assigned a consecutive chunk of size $j_{\max}/t$ along $j$. This divides the lattice into blocks of size $i_{\max} \times j_{\max}/t \times 1$. The very first block with the lowest $k$ coordinate can only be updated by a single thread ($\mathrm{T}_0$), which forms a "wavefront" by itself ($\mathrm{W}_1$ in Figure 6.4). All other threads have to wait in a barrier until this block is finished. After that, the second wavefront ($\mathrm{W}_2$) can commence, this time with two threads ($\mathrm{T}_0$ and $\mathrm{T}_1$), working on two blocks in parallel. After another barrier, $\mathrm{W}_3$ starts with three threads, and so forth. $\mathrm{W}_t$ is the first wavefront to actually utilize all threads, ending the so-called *wind-up phase*. Some time ($t$ wavefronts) before the sweep is complete, the *wind-down phase* begins and the number of working threads is decreased with each successive wavefront. The block with the largest $k$ and $j$ coordinates is finally updated by a single-thread ($\mathrm{T}_{t-1}$) wavefront $\mathrm{W}_n$ again. In the end, $n = k_{\max} + t - 1$ wavefronts have traversed the lattice in a "pipeline parallel" pattern. Of those, $2(t-1)$ have utilized less than $t$ threads. The whole scheme can thus only be load balanced if $k_{\max} \gg t$.

Listing 6.7 shows a possible implementation of this algorithm. We assume here for simplicity that $j_{\max}$ is a multiple of the number of threads. Variable `l` counts the

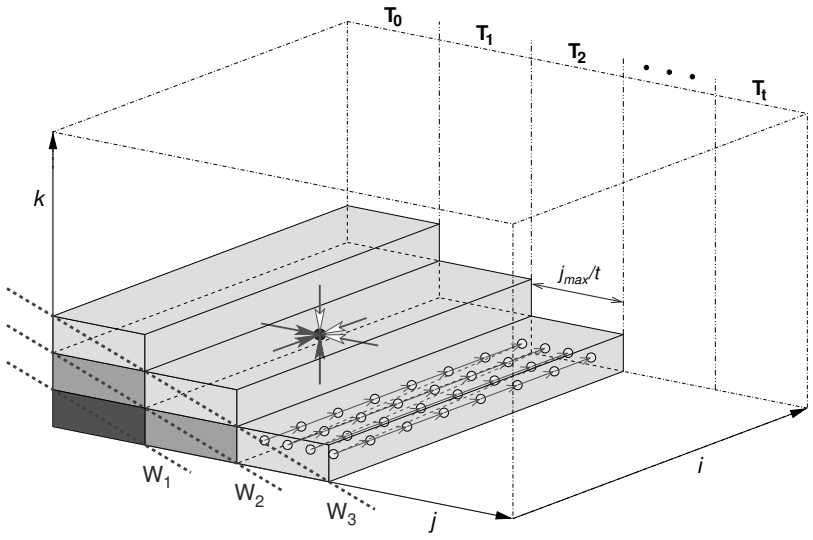


**Figure 6.4:** Pipeline parallel processing (PPP), a.k.a. wavefront parallelization, for the Gauss–Seidel algorithm in 3D (wind-up phase). In order to fulfill the dependency constraints of each stencil update, successive wavefronts ($W_1$,$W_2$,...,$W_n$) must be performed consecutively, but multiple threads can work in parallel on each individual wavefront. Up until the end of the wind-up phase, only a subset of all $t$ threads can participate.

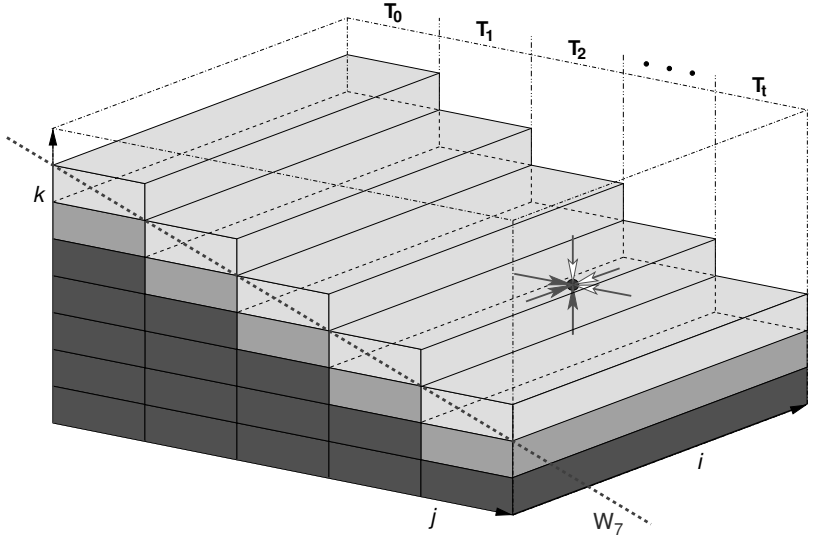


**Figure 6.5:** Wavefront parallelization for the Gauss–Seidel algorithm in 3D (full pipeline phase). All $t$ threads participate. Wavefront $W_7$ is shown as an example.

**Listing 6.7:** The wavefront-parallel Gauss–Seidel algorithm in three dimensions. Loop-carried dependencies are still present, but threads can work in parallel.

```
1  !$OMP PARALLEL PRIVATE(k,j,i,jStart,jEnd,threadID)
2    threadID=OMP_GET_THREAD_NUM()
3  !$OMP SINGLE
4    numThreads=OMP_GET_NUM_THREADS()
5  !$OMP END SINGLE
6    jStart=jmax/numThreads*threadID
7    jEnd=jStart+jmax/numThreads ! jmax is amultiple of numThreads
8    do l=1,kmax+numThreads-1
9      k=l-threadID
10     if((k.ge.1).and.(k.le.kmax)) then
11       do j=jStart,jEnd        ! this is the actual parallel loop
12         do i=1,iMax
13           phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
14                        + phi(i,j-1,k) + phi(i,j+1,k)
15                        + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
16         enddo
17       enddo
18     endif
19 !$OMP BARRIER
20   enddo
21 !$OMP END PARALLEL
```

wavefronts, and `k` is the current $k$ coordinate for each thread. The OpenMP barrier in line 19 is the point where all threads (including possible idle threads) synchronize after a wavefront has been completed.

We have ignored possible scalar optimizations like outer loop unrolling (see the order of site updates illustrated in the $T_2$ block of Figure 6.4). Note that the stencil update is unchanged from the original version, so there are still loop-carried dependencies. These inhibit fully pipelined execution of the inner loop, but this may be of minor importance if performance is bound by memory bandwidth. See Problem 6.6 for an alternative solution that enables pipelining (and thus vectorization).

Wavefront methods are of utmost importance in High Performance Computing, for massively parallel applications [L76, L78] as well as for optimizing shared-memory codes [O52, O59]. Wavefronts are a natural extension of the pipelining scheme to medium- and coarse-grained parallelism. Unfortunately, mainstream programming languages and parallelization paradigms do not as of now contain any direct support for it. Furthermore, although dependency analysis is a major part of the optimization stage in any modern compiler, very few current compilers are able to perform automatic wavefront parallelization [O59].

Note that stencil algorithms (for which Gauss–Seidel and Jacobi are just two simple examples) are core components in a lot of simulation codes and PDE solvers. Many optimization, parallelization, and vectorization techniques have been devised over the past decades, and there is a vast amount of literature available. More information can be found in the references [O60, O61, O62, O63].

## Problems

For solutions see page 298 *ff.*

6.1 *OpenMP correctness.* What is wrong with this OpenMP-parallel Fortran 90 code?

```
1    double precision, dimension(0:360) :: a
2
3  !$OMP PARALLEL DO
4    do i=0,360
5      call f(dble(i)/360*PI, a(i))
6    enddo
7  !$OMP END PARALLEL DO
8
9    ...
10
11   subroutine f(arg, ret)
12     double precision :: arg, ret, noise=1.d-6
13     ret = SIN(arg) + noise
14     noise = -noise
15     return
16   end subroutine
```

6.2 $\pi$ *by Monte Carlo.* The quarter circle in the first quadrant with origin at (0,0) and radius 1 has an area of $\pi/4$. Look at the random number pairs in $[0,1] \times [0,1]$. The probability that such a point lies inside the quarter circle is $\pi/4$, so given enough statistics we are able to calculate $\pi$ using this so-called *Monte Carlo* method (see Figure 6.6). Write a parallel OpenMP program that performs this task. Use a suitable subroutine to get separate random number se-

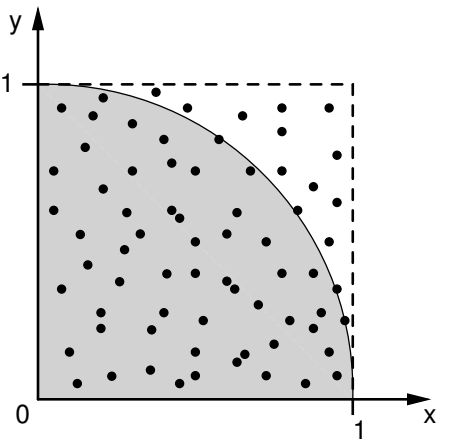


**Figure 6.6:** Calculating $\pi$ by a Monte Carlo method (see Problem 6.2). The probability that a random point in the unit square lies inside the quarter circle is $\pi/4$.

quences for all threads. Make sure that adding more threads in a weak scaling scenario actually improves statistics.

6.3 *Disentangling critical regions.* In Section 6.1.4 we demonstrated the use of named critical regions to prevent deadlocks. Which simple modification of the example code would have made named the names obsolete?

6.4 *Synchronization perils.* What is wrong with this code?

```
!$OMP PARALLEL DO SCHEDULE(STATIC) REDUCTION(+:sum)
  do i=1,N
    call do_some_big_stuff(i,x)
    sum = sum + x
    call write_result_to_file(omp_get_thread_num(),x)
!$OMP BARRIER
  enddo
!$OMP END PARALLEL DO
```

6.5 *Unparallelizable?* (This problem appeared on the official OpenMP mailing list in 2007.) Parallelize the loop in the following piece of code using OpenMP:

```
double precision, parameter :: up = 1.00001d0
double precision :: Sn
double precision, dimension(0:len) :: opt

Sn = 1.d0
do n = 0,len
  opt(n) = Sn
  Sn = Sn * up
enddo
```

Simply writing an OpenMP worksharing directive in front of the loop will not work because there is a loop-carried dependency: Each iteration depends on the result from the previous one. The parallelized code should work independently of the OpenMP schedule used. Try to avoid — as far as possible — expensive operations that might impact serial performance.

To solve this problem you may want to consider using the `FIRSTPRIVATE` and `LASTPRIVATE` OpenMP clauses. `LASTPRIVATE` can only be applied to a worksharing loop construct, and has the effect that the listed variables' values are copied from the lexically last loop iteration to the global variable when the parallel loop exits.

6.6 *Gauss–Seidel pipelined.* Devise a reformulation of the Gauss–Seidel sweep (Listing 6.6) so that the inner loop does not contain loop-carried dependencies any more. Hint: Choose some arbitrary site from the lattice and visualize all other sites that can be updated at the same time, obeying the dependency constraints. What would be the performance impact of this formulation on cache-based processors and vector processors (see Section 1.6)?