3 Tasking

Tasking was introduced in 2008 and provided a major addition to OpenMP 3.0 [1]. Thanks to this feature, algorithms with an irregular and runtime dependent execution flow may now be parallelized. Before tasking, this was not always possible, or was at least tedious and sometimes downright ugly.

A relatively simple example is a while-loop with independent chunks of work in the loop body. Although these chunks could be executed simultaneously, OpenMP did not provide the means to do so without changing the source code. In the best case, the loop could be manually transformed into a for-loop and then parallelized, but few users would like to do so.

Tasking provides an elegant solution to this problem. A queueing system dynamically handles the assignment of threads to the chunks of work that need to be performed. The threads pick up work from the queue until the queue is empty.

By design, the initial version of tasking was rather rudimentary. The idea was to decide on additional features through feedback from the users. That is indeed what has happened in subsequent releases of the specifications, and we are quite sure the current feature set is not final.

In this chapter, the tasking concepts, as well as all the features, are presented and discussed in detail. We start with a series of examples. Collectively they touch upon many of the features supported with tasking. In many cases, the features are only described briefly, but the follow-on sections after the examples go into much more detail.

3.1 Hello Task

Simply stated, an OpenMP *task* is a block of code contained in a parallel region that can be executed simultaneously with other tasks in the same region.[1]

Before we show the first example, more should be said about tasking in OpenMP. As with almost every feature in OpenMP, a task is part of a parallel region. Depending on the specific situation, this could pose a challenge. Without special precautions, the same task(s) may be executed by different threads. This is very different from the worksharing constructs, where it is well-defined how the work is distributed over the threads. As we shall see shortly, tasks are much more dynamic, and the same approach cannot be applied.

[1] This is a simplification. Subsequent sections contain the more formal coverage and definitions.

It may be needed that each thread executes all tasks, but it is more likely that different threads must execute different tasks. The problem is that, unlike a work-sharing construct, the work to be performed is not known upfront. This is why something special needs to be done.

The solution chosen by the OpenMP language committee makes sense, but at first glance may not seem intuitive. To guarantee each task is generated only once, the tasks could be embedded in a `single`, or `master` construct. There is a potential problem with the latter though, which is discussed in Section 3.1.3.

This is already a novel use of these constructs, but the thing that tends to confuse users new to tasking in OpenMP most, is that the tasks are not necessarily executed where they are defined in the source code. Tasks are *guaranteed* to be executed only at special and well-defined points in the code. Such points are implied with some constructs, but they are also under explicit control of the user.

The idea behind this approach is that one thread generates the tasks, while the other threads execute them as they become available. Eventually the generating thread may participate in the execution part, after it has completed generating the tasks. How this works is best illustrated with the example that follows next.

3.1.1 Parallelizing a Palindrome

Consider the following problem. We want to write a program that either prints `"race car "`, or `"car race "`, with no preference for either phrase.[2] At runtime, there is an equal chance of seeing either one of these phrases on the screen.

This could easily be solved using parallel sections, but let's assume these are not available. Instead, we like to use the concept of a task, briefly introduced above. The code fragment using tasks is shown in Figure 3.1.

The parallel region spans lines $1 - 10$ and has one single region only. This would not make any sense if it were not for the new task directives shown here. Each `#pragma omp task` directive defines a task. The code enclosed in the curly braces ({ and }) defines the task. The assumption is that tasks can execute independently. It is the responsibility of the user to ensure this produces correct results.

There are two tasks. One prints `"race "` and the other prints `"car "`. The order in which the tasks are executed is non-deterministic and most likely varies from run to run. For a correct parallel program, this should not affect the result. Note the trailing space in both print statements.

[2]Yes, the trailing space in the output matters here.

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
      {printf("race ");} // Task #1
    #pragma omp task
      {printf("car ");}  // Task #2
  } // End of single region
} // End of parallel region
```

Figure 3.1: **Code fragment with two tasks** – There are two tasks. One prints the word "race ". The other one prints "car ". The order in which tasks are executed is runtime dependent and may vary across multiple runs.

How about this single region then? Here is how this code is executed. Assume there are two threads. One thread encounters the `single` construct and starts executing the code block. Both tasks are generated, but not yet executed. Meanwhile, the other thread falls through and waits in the implied barrier at the end of the `single` construct.

In the case of tasks, idle threads do not really wait in the barrier though. Instead, these threads are available to execute the tasks, as they are generated.

In this simple case, that means one thread generates the two tasks consisting of a single print statement. Any available thread executes the tasks, but not until it has reached the barrier. The executing thread(s) print `"race "` and `"car "`, but not necessarily in this order.

The thread generating the tasks also ends up in the barrier, once it has completed the generation phase. Therefore, theoretically both threads could print a word each, but both tasks are so short that it is more likely the second thread prints both. The order is non-deterministic however.

This code works, but it is not a real sentence yet. Let's modify the program to print either `"A race car "`, or `"A car race "`. The relevant program fragment is listed in Figure 3.2.

By inserting a print statement before the two tasks, it is guaranteed that `"A "` is printed first. Next, the tasks are generated and executed in the barrier that is implied with the `single` construct.

```
 1 #pragma omp parallel
 2 {
 3   #pragma omp single
 4   {
 5     printf("A ");
 6     #pragma omp task
 7       {printf("race ");} // Task #1
 8     #pragma omp task
 9       {printf("car ");}  // Task #2
10   } // End of single region
11 } // End of parallel region
```

Figure 3.2: **Augmented code fragment with two tasks** – The print statement at line 5 is executed by one thread only and before the tasks are generated. This ensures the sentence starts with "A ".

3.1.2 Parallelizing a Sentence with a Palindrome

Although longer, this is still not a full sentence. We want to append `"is fun to watch."` to it and also include a new-line symbol. That seems easy, doesn't it? The code fragment in Figure 3.3 adds a print statement before the end of the single region.

Although this seems straightforward, the output is not what we would expect to see. The six possible results are listed in Figure 3.4. Which one of these is printed at runtime depends on the number of threads used, the load on the system, and the implementation details of tasking.

The sentence always starts with `"A "`, but what happens next "depends." Remember that the tasks are executed by threads waiting in the barrier.

To simplify the discussion, the reference numbers in the first column of Figure 3.4 are used. The thread executing the single region is referred to as T_S.

In one scenario, assuming there are at least two threads, the other thread(s) could arrive in the barrier after thread T_S has executed the single region. In this case, either output 1 or 2 is printed.

It may also happen that one or more thread(s) arrive in the barrier before T_S has completed the single region. Then, one of the outputs $3 - 6$ is printed. In the case of 3 or 4, one task was executed after T_S finished. If this thread finishes after the other threads, outputs 5 or 6 are printed.

```
 1 #pragma omp parallel
 2 {
 3   #pragma omp single
 4   {
 5     printf("A ");
 6     #pragma omp task
 7       {printf("race ");} // Task #1
 8     #pragma omp task
 9       {printf("car ");}  // Task #2
10     printf("is fun to watch.\n");
11   } // End of single region
12 } // End of parallel region
```

Figure 3.3: **An incorrect sentence with a palindrome and two tasks** – This program does not produce the desired result. The print statement at line 10 is executed by one thread only. Although at the end of the single region, this line may be printed before the tasks get to print their word.

Reference ID	Text printed

1	`A is fun to watch.`
	`race car`

2	`A is fun to watch.`
	`car race`

3	`A race is fun to watch.`
	`car`

4	`A car is fun to watch.`
	`race`

5	`A race car is fun to watch.`
6	`A car race is fun to watch.`

Figure 3.4: **The six possible outputs from the program in Figure 3.3** – The first column contains a reference number used in the explanation. The second column lists the possible lines printed from this program.

Any of the above scenario's are possible. It depends on the number of threads used. With two threads the behavior is different compared to using three threads. Also the load on the system has an impact. If, for example, one thread gets delayed, the output may be different.

```
 1 #pragma omp parallel
 2 {
 3   #pragma omp single
 4   {
 5     printf("A ");
 6     #pragma omp task
 7       {printf("race ");} // Task #1
 8     #pragma omp task
 9       {printf("car ");}  // Task #2
10     #pragma omp taskwait
11       printf("is fun to watch.\n");
12   } // End of single region
13 } // End of parallel region
```

Figure 3.5: **A correct sentence with a palindrome and two tasks** – At line 10, the `taskwait` construct forces the pending child tasks to complete before execution resumes. In this way it is guaranteed the words are printed in the right order.

The tasking implementation details matter also. A work queueing system is used to handle the generation and execution of tasks. There are several different ways to assign threads to tasks, and this impacts the order of execution.

In a way, none of this matters though, because this program is not guaranteed to produce the output we want to see. What we need to do is to guarantee the two tasks are completed *before* the print statement at line 10 is executed. In this simple case, we could easily solve this by moving the last print statement out of the parallel region, but what if this is not possible? For such cases, the `taskwait` construct can be used. This feature is used in Figure 3.5 to achieve the desired result.

Execution proceeds as follows. Thread T_S enters the single region and executes the statement at line 5, printing `"A "`. Next, it generates the two tasks with the print statement. At some point in time, the other threads enter the implied barrier that is part of the `single` construct. Once the tasks become available, they will be executed by those threads.

The `taskwait` construct forces thread T_S to wait, until the two tasks have completed. In this case this means it must wait for `"race car "` or `"car race "` to have been printed. After that, it prints `"is fun to watch."`.

3.1.3 Closing Comments on the Palindrome Example

In this example the `single` construct is used to ensure each task is generated only once. Theoretically, the same could be achieved using the `master` construct, but there is a problem with that. In the barrier, all explicit tasks generated by the team must be executed to completion. The `master` construct has no implied barrier, however.

In this case, no harm is done, because then the tasks are executed in the barrier that is implied with the parallel region. This is, of course, specific to the example. In general, if the `master` construct is used with tasks, care needs to be taken when they are executed.

Generally, this is not the best approach anyhow. The master thread tends to be quite busy with the overall execution already. Why add more burden to it? Instead, we recommend using the `single` construct, possibly with the `nowait` clause. In the case of the latter, the same caveat about task completion holds.

Another option is to use parallel sections, but with only one single section. As with the `single` construct, the `nowait` clause can be used to eliminate the redundant barrier.

Another thing worth mentioning is that in this particular example, there is no data environment to worry about. This is very unlikely to be the case in more realistic applications. What you need to know about the data environment in tasks is extensively covered in the sections to follow, in particular in Section 3.5 on page 136.

3.2 Using Tasks to Parallelize a Linked List

The previous example introduced tasking at a very basic level and was somewhat crafted in the sense that other OpenMP constructs are more suitable and easier to solve the problem. The next example does not have such an easy solution though.

This program uses a linked list. The assumption is that each node in the linked list represents work independent of the other nodes. The independence opens up opportunities for parallelism, but unfortunately a linked list needs to be traversed sequentially, ruining the parallelism.

That is, unless we can generate a list with all the nodes first. The data elements from this list can then be processed simultaneously. This makes it a natural candidate for tasking.

```
 1 struct linked_list {
 2   int64_t value;
 3   struct linked_list *next;
 4 };
 5
 6 typedef struct linked_list my_data;
 7
 8 static int function_call_count = 0;
 9
10 int main(int argc, char *argv[])
11 {
12  my_data *previous, *ptr, *current, *head_of_list;
13
14  int64_t ntasks;
15  int64_t no_of_tasks_failed = 0;
16
17  (void) get_parameters(argc, argv, &ntasks);
18
19  }
```

Figure 3.6: **The first part of the sequential implementation of the linked list program** – The key data structures are defined and the input parameters are read.

3.2.1 The Sequential Version of the Linked List Program

This program is relatively lengthy, but for reasons that become clear soon, most of the code is presented and discussed. The initial part of the code is listed in Figure 3.6.

This first part is fairly straightforward. At lines $1-4$, the key list data structure, called `linked_list`, is defined. The list has two elements only: an integer value and a pointer to the next node in the list. To make things easier, an alias named `my_data` to reference the linked list is defined. Line 8 defines and initializes a static (global) variable called `function_call_count` to be used later on. Lines $12-15$ define other program variables needed. This includes four pointers to `my_data`. At line 17, a single input parameter is returned by the function call. This variable is called `ntasks` and defines the length of the list. As we shall see shortly, the number of tasks corresponds to the number of entries in the list, hence the name. Although

```
20  for (int64_t i=0; i<ntasks; i++) {
21      if ( (ptr = (my_data *) malloc(sizeof(my_data))) == NULL ) {
22         perror("ptr"); exit(-1);
23      }
24
25      if (i == 0 ) {
26         head_of_list = ptr;
27      } else {
28         previous->next = ptr;
29      }
30
31      ptr->value = i+1;
32      ptr->next  = NULL;
33      previous   = ptr;
34  }
```

Figure 3.7: **The second part of the sequential implementation of the linked list program** – By design, the linked list has `ntasks` entries. A for-loop with this length is used to construct the list.

seemingly a detail, variables `ntasks` and `no_of_tasks_failed`, are declared to be a 64-bit integer. This is because the number of tasks may easily be very large.

In the next block of code, the linked list is initialized. The code fragment is listed in Figure 3.7. By design, the list has `ntasks` entries and a for-loop is used to initialize these entries. This loop spans lines $20 - 34$. Memory for the next node in the list is allocated at lines $21 - 23$. Variable `ptr` contains the pointer to the new memory block. At line 26, the starting point of the linked list is saved in variable `head_of_list`. Upon subsequent loop iterations, the previous value of the pointer is saved (line 28). Lines $31 - 32$ assign a value to the data part of the list and the `NULL` pointer to the next node. The latter is because the end of the linked list needs to be terminated by the `NULL` pointer. At line 33, the list is updated with the new pointer.

After execution of this for-loop, a `NULL`-terminated linked list, consisting of `ntasks` nodes has been created. Variable `head_of_list` contains the start of the list.

The most interesting and relevant part of this example is shown in Figure 3.8. This is where the linked list is processed. Pointer variable `current` is used to keep track of the most recent node. At line 35, it is initialized to the start of the linked list.

```
35  current = head_of_list;
36  while (current != NULL) {
37    int64_t return_value;
38    return_value = do_work(current->value);
39    if ( return_value < 0 ) {
40       no_of_tasks_failed++;
41    }
42    current = current->next;
43  } // End of while loop
```

Figure 3.8: **The third part of the sequential implementation of the linked list program** – This is where the linked list is processed. At line 35, the most recent node location is set to the start of the list. The while-loop spans lines 36 – 43 and executes a function called `do_work()` for each data element in the list. The parameter to the function call is the value of the data element. At line 42, the pointer advances to the next node in the list.

The key part of this program is the while-loop spanning lines 36 – 43. This loop terminates in case the pointer to the next node is `NULL`. At line 38, a function called `do_work()` is called. It has one input parameter, `current->value`, that contains the value of the data element in the current node.

In case there was a problem executing `do_work()`, a negative value is returned. This is checked for at line 39 and if a problem has occurred, status variable `no_of_tasks_failed` is incremented by one (line 40). Note that it was initialized to zero at line 15 in Figure 3.6. After completion of the while-loop, the main program can use the value of variable `no_of_tasks_failed` to check for failures.

The details of function `do_work()` need to be discussed, because it requires some attention in the next section. The source code is listed in Figure 3.9. This function emulates a runtime error and uses a global variable, called `function_call_count`, to control the value returned to the caller.

At line 3, this variable is updated each time the function is called. By design, the first `MAX_ERRORS` calls to this function return a negative value, emulating a runtime error. After this, the function returns zero, indicating there were no errors. This is what the code at lines 5 – 9 accomplishes. This block of code could be replaced by a single statement, but as shown in the next section, these lines need special care. This would be equally true for the one-liner. Note that the input parameter is not used, but has been included here to make the code look more realistic. In a

```
int64_t do_work(int64_t input_value)
{
  function_call_count++;

  if ( function_call_count <= MAX_ERRORS ) {
     return(-1);
  } else {
     return(0);
  }
}
```

Figure 3.9: **The source code of function do_work()** – This function is a template to emulate an error that may have occurred during execution. For demonstration purposes, a global variable, called `function_call_count`, is updated.

real-world application, this value is most likely needed.

This concludes the description of the sequential version of this program. From now on, the focus is on the core part of the program, as shown in Figure 3.8. The challenge is going to be to parallelize it.

3.2.2 The Parallel Version of the Linked List Program

The `NULL` terminated while-loop is not straightforward to parallelize, because, unlike a for-loop, it is not known at runtime how often the loop body is going to be executed.

The main thing to realize here is that there is actually no need to know this. What if there was a dynamic queueing system to handle the assignment of threads to the chunks of work that need to be performed? If such a system is available, the threads can pick up work from the queue until the queue is empty. This is somewhat akin to the `dynamic` and `guided` loop iteration scheduling algorithms, available for parallel loops. The big difference is that the tasking concept is much more flexible and dynamic.

This is exactly the idea behind tasking. With tasking, it is up to the implementation to set up an internal (queueing) system that ensures the tasks are generated and executed. This system also decides upon the assignment of threads to the tasks in the queue. Therefore, we need to find a way to define the execution of the

```
 1 current = head_of_list;
 2 #pragma omp parallel firstprivate(current) \
 3                      shared(no_of_tasks_failed)
 4 {
 5   #pragma omp single nowait
 6   {
 7     printf("Thread %d executes the single region\n",
 8            omp_get_thread_num());
 9     while (current != NULL) {
10       #pragma omp task firstprivate(current) \
11                        shared(no_of_tasks_failed)
12       {
13         int64_t return_value;
14         return_value = do_work(current->value);
15         if ( return_value < 0 ) {
16              #pragma omp atomic update
17                no_of_tasks_failed++;
18         }
19       } // End of task
20       current = current->next;
21     } // End of while loop
22   } // End of single region
23 } // End of parallel region
```

Figure 3.10: **The parallel implementation of the linked list program** – The entire while-loop is embedded inside a parallel region, spanning lines 2 – 23. This region consists of one single region (lines 5 – 22). The thread executing this region creates the tasks defined at lines 10 – 19. Tasks are executed in the implicit barrier at the end of the parallel region (line 23). For more details, refer to the main text.

while-loop in terms of tasks. Due to the sequential nature of scanning the nodes of the list, a `single` construct is ideally suited to generate the tasks.

The relevant code fragment is shown in Figure 3.10. As before, the current node in the linked list is stored in variable `current` and initialized to `head_of_list` (line 1). The entire while-loop is embedded inside a parallel region, spanning lines 2 – 23.

The scoping is fairly straightforward. All threads get their private copy of pointer variable `current`. Thanks to the `firstprivate` clause, every thread has a copy, initialized to point to the first node in the list. Variable `no_of_tasks_failed` needs to be updated by all threads and is therefore a shared variable. The same scoping is used for the tasks (lines $10-11$).

The parallel region consists of one relatively gigantic single region starting at line 5 and ending at line 22. Similar to what was seen in the previous example in Section 3.1.1, this approach is used to ensure one thread generates the tasks. For diagnostic purposes, the thread ID of the thread executing this single region is printed (lines $7-8$).

The two adjacent implied barriers at lines $22-23$ are not needed. We can easily eliminate the one at line 22 by using the `nowait` clause in line 5. As a result, all other threads wait in the implied barrier at the end of the parallel region at line 23.

The tasks are defined at lines $10-19$. Ignoring the details of the tasks for a moment, we know that this one thread executes the while-loop and scans through the linked list. This is exactly the same as we saw earlier with the sequential version. When this thread encounters the task region (lines $10-19$), it generates the appropriate code, plus data environment, to execute the specific task. At line 20, the pointer advances to the next node and at line 10 the next task can be generated. This is still a sequential operation. Meanwhile, the other threads wait in the barrier at line 23, but they are not really waiting. Instead, they start executing the tasks, generated by the thread executing the single region. Once this thread has finished, and if there is still work left to do, it joins the other threads and starts executing tasks from the queue.

Now that it is covered in what way these tasks are generated and executed, it is time to zoom in on the code that defines a task. At line 13, variable `return_value` is declared. It is private to the task and used to store the value returned by function `do_work()`.[3]

At this point, things get interesting, because we want to detect a possible error that has occurred when executing this function. For good reasons it is not allowed to jump out of a parallel region, because then other threads wait in the next barrier for one or more threads that never arrive there.

In Section 2.4.2 it is shown how the thread cancellation feature can be used for these situations, but in this example the more conventional approach is chosen. After the parallel region has completed, the value of variable `no_of_tasks_failed`

[3]This variable can be eliminated, but is used here to show the (simple) use of such a private variable.

```
 1 int64_t do_work(int64_t input_value)
 2 {
 3   int64_t TID, return_value;
 4
 5   TID = omp_get_thread_num();
 6
 7   #pragma omp critical
 8   {
 9     function_call_count++;
10
11     printf("TID = %-6ld Input: %-6ld function_call_count: %d\n",
12            TID, input_value, function_call_count);
13
14     if ( function_call_count <= MAX_ERRORS ) {
15        return_value = -(TID+1);
16     } else {
17        return_value = 0;
18     }
19   } // End of critical region
20
21   return(return_value);
22 }
```

Figure 3.11: **The thread-safe version of function do_work()** – This is the thread-safe version of the sequential code shown in Figure 3.9. The update at line 9 needs to be guarded against a data race and the read of this variable at line 14 has to be protected as well. This is achieved by using a critical section spanning lines 7 – 19.

may be checked. If it is non-zero, an error has occurred and the program can take appropriate action.

Simply incrementing variable `no_of_tasks_failed` in case the function returns a negative value, is correct in the sequential version. For the parallel version, it is necessary to prevent that multiple threads simultaneously update this variable. The solution is straightforward. This is a perfect case to use an atomic update (line 16 – 17) to avoid the data race. See also Section 2.1.6 for more details on the `atomic` construct.

There is one more thing left to do. Function `do_work()` needs to be made thread-safe. The modified, parallel, source code of this function is listed in Figure 3.11. It is the thread-safe version of the sequential code listed earlier in Figure 3.9.

At line 5, the thread ID is assigned to variable `TID`. This is used to ensure that each invocation of this function returns a unique negative number in case an error has occurred. As before, a negative value is returned the first `MAX_ERRORS` times this function is executed. After that, a value of zero is returned. In this way, the number of erroneous function calls is easily controlled and simulated.

In a sequential program, it is easy to keep track of the number of times the function is called. Global variable `function_call_count` is simply updated each time the function is called. In a multi-threaded environment, the update of a global variable requires care. Without any precaution, a data race occurs at line 9, because multiple threads are allowed to update the same variable at the same time. Typically, an atomic update is used to avoid this, but in this case there is also a potential data race between the write at line 9, and the read at line 14. This is why the entire code block is a critical region. It is assumed that in a real-world application quite some work is performed here. Otherwise, the cost of the critical region offsets, or even outweighs, any performance gain.

Last, but not least, the return value of this function needs to be handled. In the sequential version in Figure 3.9, the `return()` function was called at lines 6 and 8, using the appropriate argument. Since control cannot be transferred out of a critical region, calling the `return()` function is not possible. Instead, the desired value is assigned to variable `return_value` and this is used as the argument in the `return()` call at line 21.

Note that when setting the return value at line 15, a value of one is added to variable `TID`. Otherwise the return value for the master thread is zero, and not distinguishable from the return value set upon correct execution.

For demonstration purposes a print statement has been included in lines $11-12$. The thread ID, the value of the input parameter, and the number of times the function has been called, are printed. This program has been executed using 10 tasks and 20 threads. The results in Figure 3.12 show that the generating thread, 17, also executes two tasks. Not all threads execute the same number of tasks. The other eight tasks are distributed over six threads. Although there are sufficient threads to execute one task each, this is not what happens at runtime. Apparently, only seven threads were put to work. This demonstrates the dynamic behavior that comes with tasking.

```
Thread 17 executes the single region
TID = 13     Input: 1      function_call_count: 1
TID = 11     Input: 4      function_call_count: 2
TID = 17     Input: 10     function_call_count: 3
TID = 12     Input: 5      function_call_count: 4
TID = 11     Input: 8      function_call_count: 5
TID = 13     Input: 6      function_call_count: 6
TID = 14     Input: 2      function_call_count: 7
TID = 15     Input: 7      function_call_count: 8
TID = 17     Input: 9      function_call_count: 9
TID = 8      Input: 3      function_call_count: 10
```

Figure 3.12: **Example output from the parallel linked list program** – The program was executed for `ntasks=10` and using 20 threads. This output demonstrates that the generating thread, 17, also executes two tasks.

3.2.3 Closing Comments on the Linked List Example

This type of algorithm was notoriously hard to parallelize and one of the reasons to introduce the tasking concept in OpenMP.

What confuses users new to tasking, is that the parsing of the linked list is still a sequential process. Due to the pointer-chasing nature of scanning such a list, this cannot be avoided, but there is also no need to do so.

With tasks, as soon as an entry from the list has been created, the processing can start in a natural way.

3.3 Sorting Things Out with Tasks

Through nested parallelism, OpenMP has supported the parallelization of recursive algorithms, but this is a rather heavy and rigid mechanism. The typical cost of a parallel region is non-negligible. To make it even worse, each parallel region has an implied barrier and there is no way to omit it. Both drawbacks make it very difficult to efficiently parallelize, and load balance, fine-grained portions of work in recursive algorithms.

This example shows how a recursive divide-and-conquer algorithm can be parallelized using tasks. As with the other examples, this is a template for how to tackle similar problems in this category.

The popular quicksort algorithm has been selected [22]. It is conceptually simple, but has a structure that allows us to demonstrate various tasking features needed to implement the parallelism.

3.3.1 The Sequential Quicksort Algorithm

The quicksort algorithm uses a divide-and-conquer algorithm. Such an algorithm splits the work in two independent sections. In a recursive manner, each section is then split again. The sequential quicksort algorithm to sort an array is defined as follows:

1. Select an element, called a *pivot*, from the array.
2. In the *partitioning* phase, the array is re-ordered, such that all elements with values less than the pivot are placed before the pivot. All elements with values greater than the pivot are moved after it.[4] After this partitioning step, the pivot is guaranteed to be in its final position.
3. Recursively apply the above two steps to the elements to the left and right of the pivot.
4. The recursion ends when there is only one, or no, element left.

In the example discussed here, a specific implementation of the algorithm outlined above is presented. This is certainly not the only version. Among other choices, the selection of the pivot and partitioning phase are parameters affecting the performance. The partitioning algorithm used in this section is listed in Figure 3.13.

Function `choosePivot` is defined at lines $1-3$. It returns $\mathit{floor}(lo+hi)/2$.[5] If $lo+hi$ is an even number, it returns the array value in the center. Otherwise, it returns the integer value rounded down from $(lo+hi)/2$. For example, for $lo=3$ and $hi=6$, a value of 4 is returned.

The actual partitioning algorithm is defined at lines $10-24$. After the pivot value and corresponding index are defined (lines $10-11$), the pivot value is moved to `a[hi]`, the end of the array section. The value that was originally there has been moved by the `swap` function to where the pivot was.

[4]Equal values can go either way

[5]The Glossary contains the definition of this function.

```
 1 int64_t choosePivot(int64_t *a, int64_t lo, int64_t hi)
 2 {
 3  return( (lo+hi)/2 );
 4 }
 5
 6 int64_t partitionArray(int64_t *a, int64_t lo, int64_t hi)
 7 {
 8   int64_t pivotIndex, pivotValue, storeIndex;
 9
10   pivotIndex = choosePivot(a, lo, hi);
11   pivotValue = a[pivotIndex];
12
13   (void) swap(&a[hi], &a[pivotIndex]);
14
15   storeIndex = lo;
16
17   for (int64_t i=lo; i <= hi-1; i++)
18   {
19    if ( a[i] < pivotValue ) {
20       (void) swap(&a[i], &a[storeIndex]);
21       storeIndex++;
22    }
23   }
24   (void) swap(&a[storeIndex], &a[hi]);
25
26   return(storeIndex);
27 }
```

Figure 3.13: **Implementation of the partitioning phase** – This code implements the partitioning phase in the quicksort algorithm. In this case, the pivot element is defined to be $\mathit{floor}(lo + hi)/2$, the value in the center, or one element less, of the array. The value returned by the partitioning function is an index. On return, array elements with an index less than the return value are smaller than the pivot value. The array value at this index is the pivot value.

Variable `storeIndex` is initialized at line 15. It is used to point to the next available index in the left part of the array. This is where the values less than the pivot value are stored.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	`storeIndex`
Initial values	8	6	3	4	4	5	3	9	2	2	
Before initial swap	8	6	3	4	**4**	5	3	9	2	**2**	
After initial swap	8	6	3	4	2	5	3	9	2	4	`0`
Source line 19 (i = 2)	**8**	6	**3**	4	2	5	3	9	2	4	`0`
Swap and update (i = 2)	3	6	8	4	2	5	3	9	2	4	`1`
Source line 19 (i = 4)	3	**6**	8	4	**2**	5	3	9	2	4	`1`
Swap and update (i = 4)	3	2	8	4	6	5	3	9	2	4	`2`
Source line 19 (i = 6)	3	2	**8**	4	6	5	**3**	9	2	4	`2`
Swap and update (i = 6)	3	2	3	4	6	5	8	9	2	4	`3`
Source line 19 (i = 8)	3	2	3	**4**	6	5	8	9	**2**	4	`3`
Swap and update (i = 8)	3	2	3	2	6	5	8	9	4	4	`4`
Before final swap	3	2	3	2	**6**	5	8	9	4	**4**	`4`
After final swap	$\underline{3}$	$\underline{2}$	$\underline{3}$	$\underline{2}$	$\underline{4}$	5	8	9	4	6	`4`

Figure 3.14: **An example of the partitioning phase** – This shows how the array elements are moved around during the partitioning phase. The top row has the array index values. In this example, `lo = 0` and `hi = 9`. The pivot element is `4` and the pivot index is `4` as well. Array elements marked in bold are to be swapped next. In the last column the value of variable `storeIndex` is shown. The last row contains the resulting array. The elements that are moved to the left of the pivot are underlined.

The for-loop spanning lines $17 - 23$ moves all values less than the pivot value, if there are any, to the left. They are stored in a linear fashion, starting at `a[lo]`. After this loop has completed, $a[i] < pivotValue$ for $i = lo, \ldots, storeIndex - 1$. These values are not necessarily sorted yet. At line 24, the pivot value is put back and placed at the end of this sequence. Variable `storeIndex` separates the array values: $a[i] \leq pivotValue$ for $i = lo, \ldots, storeIndex$. Array elements for an index value exceeding `storeIndex` are equal or larger.

Figure 3.14 shows an example for an array consisting of 10 elements. As the partitioning algorithm proceeds, elements are swapped until those strictly less than the pivot value are to the left of the pivot.

Step by step, array elements are moved to their new position. Upon completion of this function, the elements are partially sorted. The index value returned partitions

```
 1 void seqQuicksort(int64_t *a, int64_t lo, int64_t hi)
 2 {
 3   if ( lo < hi ) {
 4     int64_t p = partitionArray(a, lo, hi);
 5
 6     (void) seqQuicksort(a, lo, p - 1); // Left branch
 7
 8     (void) seqQuicksort(a, p + 1, hi); // Right branch
 9   }
10 }
```

Figure 3.15: **The sequential quicksort implementation** – This is the core part of the algorithm. After the partitioning phase, array element `a[p]` is in the correct place. Elements to the left and right of it are each partitioned in a recursive manner.

the array. The corresponding array element is in its final position. All elements to the left are strictly less than the pivot value. The array elements with a higher index value are equal, or larger.

The next step is to repeat this process for the elements to the left and right of the index value returned. These two operations are independent and can be carried out simultaneously. This process is then applied repeatedly until all elements have been sorted. The key part of the sequential quicksort source code implementing this divide and conquer algorithm is listed in Figure 3.15.

The code is executed as follows. If `lo` is equal to, or exceeds `hi`, the function immediately returns, terminating the current execution path. Otherwise, at line 4, array `a` is partitioned within the range defined by parameters `lo` and `hi`. After this call, array element `a[p]` is in its final position and the array sections to the left (line 6) and to the right (line 8) are partitioned.

This pattern repeats recursively. Because the range decreases, at some point, the if-statement at line 3 evaluates to `false` and the execution terminates. Step by step, the function calls are skipped and when the program terminates, the array values have been sorted.

3.3.2 The OpenMP Quicksort Algorithm

As a first step towards a parallel implementation using tasking, the framework is set up. The still-to-be-written parallel function is called from here. To underline

```
1 #pragma omp parallel default(none) shared(a,nelements)
2 {
3   #pragma omp single nowait
4   {
5     (void) ompQuicksort(a, 0, nelements-1);
6   } // End of single section
7 } // End of parallel region
```

Figure 3.16: **The driver part for the OpenMP quicksort function** – As seen earlier, the function call is embedded in a parallel region. Since function `ompQuicksort` uses tasks, the `single` directive ensures only one thread generates the tasks. To avoid a redundant barrier, the `nowait` clause is used.

this is the OpenMP version, the function is called `ompQuicksort`. The relevant part of the calling program is listed in Figure 3.16. This driver part is straightforward. To ensure only one thread generates the tasks, a `single` directive is used to enforce this. The `nowait` clause at line 3 removes a redundant barrier.

Next, tasks need to be used to parallelize the sequential code shown in Figure 3.15. The main decision to be made is to identify which part(s) of the code should be defined as tasks. In this case, these are the calls at lines 6 and 8 of the sequential code. The modified source code using tasks is shown in Figure 3.17.

When comparing the two listings, it is easy to see that, other than changing the function name to distinguish the sequential and parallel versions, and the addition of two directives, the code has not changed.

As always, one must think about the scoping of the variables and it is good practice to specify this explicitly. Although it is the default, the variables that control the section of the array to be sorted are defined to be `firstprivate`.

The same could be done for the pointer to array `a`, but it is more natural to make it shared. This reflects that all tasks are working on the same block of memory. It is our responsibility to avoid data races. In this case, this is not an issue, because there is never a duplicate index value.

The first time function `ompQuicksort` is called, two tasks are created. They are working on two disjoint parts of the array to the left and right of the pivot element, which is in its final position.

The very first time the function is called, the entire array is scanned and partitioned. This is a sequential phase in the parallel algorithm, but after this initial

```
void ompQuicksort(int64_t *a, int64_t lo, int64_t hi)
{
  if ( lo < hi ) {
    int64_t p = partitionArray(a, lo, hi);

    #pragma omp task default(none) shared(a) firstprivate(lo,p)
      { (void) ompQuicksort(a, lo, p - 1); } // Left branch

    #pragma omp task default(none) shared(a) firstprivate(hi,p)
      { (void) ompQuicksort(a, p + 1, hi); } // Right branch
  }
}
```

Figure 3.17: **The OpenMP quicksort implementation** – The two (recursive) calls to `ompQuicksort` are defined to be tasks. The variables defining the array sections to be sorted are declared `firstprivate`. Although not the only choice, making array `a` shared is most straightforward.

split, there are two disjoint branches executing in parallel. Each branch subsequently splits itself into two more parallel function calls, generating two tasks. This splitting continues until there is only one element left in a particular branch. No more tasks are generated in this branch, ending the sorting process for this part of the array.

Unlike the example in Section 3.1.1, there is no `taskwait` construct needed, because each task executes independently. There is no need to wait for other tasks.

3.3.3 Fine-Tuning the OpenMP Quicksort Algorithm

The implementation shown in the previous section works fine. The code executes in parallel and the array is correctly sorted. There is an issue though. For a sufficiently large array, many tasks are generated. This in itself may already impact performance due to resource constraints, but in case of a divide-and-conquer method, the amount of work performed per task decreases, as the algorithm progresses. The cost of task management does not decrease, however and eventually this overhead dominates.

These performance issues are not restricted to this example. In many algorithms that use tasks, these need to be considered, especially if the amount of work

```
 1 void ompQuicksort(int64_t *a, int64_t lo, int64_t hi)
 2 {
 3   if ( lo < hi ) {
 4
 5    if ( (hi - lo + 1) < cutoff_seq ) {
 6       seqQuicksort(a, lo, hi); // Call sequential version
 7    } else {
 8       int64_t p = partitionArray(a, lo, hi);
 9
10       #pragma omp task default(none) shared(a) firstprivate(lo,p)
11         { (void) ompQuicksort(a, lo, p - 1); } // Left branch
12
13       #pragma omp task default(none) shared(a) firstprivate(hi,p)
14         { (void) ompQuicksort(a, p + 1, hi); } // Right branch
15    }
16   }
17 }
```

Figure 3.18: **The OpenMP quicksort implementation with escape hatch** – The code at lines 5 – 6 ensures that a different function is executed in case the array length is below a certain threshold. The obvious choice is to call the sequential version of the algorithm then.

performed decreases over time. Let's first look at limiting the number of tasks generated.

In Figure 3.18, a solution is shown, where an escape hatch is introduced. In case the array length drops below a user-defined threshold `cutoff_seq`, the sequential version of the algorithm is executed.

This approach is simple and effective, but it has two drawbacks. One must write a sequential version of the algorithm. Admittedly, this is very similar to the OpenMP version, but there is also the issue of having to maintain two (nearly identical) sources.

Another problem to deal with is that one must choose the appropriate value for the threshold variable `cutoff_seq`. This is not trivial, because the optimal value depends on certain system and OpenMP implementation-dependent characteristics. The problem size may also influence the value for the cutoff. It is therefore highly recommended to conduct some experiments to find the optimal value. As a guideline, the cutoff value can be set, such that the OpenMP version using a single thread performs the same as, or similar to, the sequential version.[6]

The above approach was the most common solution chosen at the time tasking was introduced in OpenMP 3.0, but in OpenMP 3.1 two more clauses were added to make it easier to fine-tune the performance of tasking.

The `final(`*`expr`*`)` clause can be used to avoid the runtime system continuing to generate tasks. It targets recursive algorithms and nested tasks, where the number of tasks tends to increase exponentially. This cannot only clog the runtime system, but especially if the amount of work decreases over time, the tasking overhead may offset any performance gains. The `final` clause takes an expression `expr`. If it evaluates to `true`, no more tasks are generated and the code is executed immediately. This is propagated to all of the child tasks.

This clause can be combined with the `mergeable` clause to avoid a separate data environment being created. This topic is covered in more detail in Section 3.7.

Effectively, the combination of these clauses, together with a suitable cutoff value, eliminates the need to have a separate sequential version. This is demonstrated in Figure 3.19.

The if-statement and call to the sequential version of the algorithm have been removed. Instead, the two clauses have been added on lines 7 and 11. They achieve the same functionality, but eliminate the need to have two separate source trees to maintain. As before, some experimentation is needed to determine the optimal value for the threshold variable `cutoff_tasks`.

There is one more thing that may be done to make this algorithm perform more efficiently. There is actually no need to have two tasks. The source for this approach is shown in Figure 3.20. The code is obviously nearly identical to the previous version. Only the tasking pragma and curly braces around the second call to execute the right branch have been removed. In this way, the tasking overhead is reduced, while there is still parallelism to be exploited.

As trivial as this change may seem, the order matters. With the approach shown, each invocation of the function generates a task. All these tasks are put on the runtime queue and executed by the threads. Meanwhile the sequential calls to the right branch are executed as well.

With tasking pragma on the right branch, there is a sequential path first. Only when this has finished, are the tasks generated and executed.

[6]Using bisection, this value can often be found relatively quickly.

```
int64_t omp_quicksort(int64_t *a, int64_t lo, int64_t hi)
{
 if ( lo < hi ) {

   int64_t p = partition(a, lo, hi);

   #pragma omp task final( (p - lo) < cutoff_tasks) mergeable \
                    default(none) shared(a) firstprivate(lo,p)
   { (void) omp_quicksort(a, lo, p - 1); } // Left branch

   #pragma omp task final( (hi - p) < cutoff_tasks) mergeable \
                    default(none) shared(a) firstprivate(hi,p)
   { (void) omp_quicksort(a, p + 1, hi); } // Right branch

   return(p);
 }
}
```

Figure 3.19: **The OpenMP quicksort implementation using the final and mergeable clauses** – The test and call to the sequential version have been removed. Instead the `final` and `mergeable` clauses are used to achieve the same result. Just as with the previous version, finding the correct value for the threshold variable `cutoff_tasks` requires some experimentation.

3.3.4 Closing Comments on the OpenMP Quicksort Algorithm

Although this algorithm has been covered in quite some detail, there are some additional comments to be made.

On the algorithmic side, the most straightforward implementation has been selected, because it is already sufficiently complex to demonstrate the use of tasking to parallelize a divide-and-conquer type of algorithm. Without a doubt, there are more efficient sequential versions out there.

For one thing, the choice of the pivot is quite crucial. In our code, the middle element is selected, because that is good for load balancing. It is however not necessarily the optimal choice to get the best sequential performance.[7]

[7]This could be something to consider for the sequential version in Figure 3.18, if available.

```
int64_t omp_quicksort(int64_t *a, int64_t lo, int64_t hi)
{
 if ( lo < hi ) {

   int64_t p = partition(a, lo, hi);

   #pragma omp task final( (p - lo) < cutoff_tasks) mergeable \
                    default(none) shared(a) firstprivate(lo,p)
   { (void) omp_quicksort(a, lo, p - 1); } // Left branch

   (void) omp_quicksort(a, p + 1, hi);     // Right branch

   return(p);
 }
}
```

Figure 3.20: **The OpenMP quicksort implementation with one task per call** – There is only one task per call now. Each time this function is called, a task is generated and put in the queue for the threads to work on. Note that issuing the task first is more efficient than the other way round.

Last, but not least, when trying this, be prepared to conduct several experiments. Not only should this be done to determine the threshold(s), but also to decide on the best thread affinity strategy, which is covered in Chapter 4.

3.4 Overlapping I/O and Computations Using Tasks

In this section, the code shown in Figure 1.11 on page 21 is revisited. The example shows the use of parallel sections to overlap I/O and computations. Under the assumption that the I/O can be processed in chunks, a pipeline is set up, such that the computational part waits for a read operation to be completed. Once completed, the data is processed and passed on to a post-processing function. This pipeline approach is demonstrated in Figure 3.21.

The pipeline works as follows. At the start, the first chunk of data is read. All subsequent activities need to wait for this operation to be completed. Once finished, the processing can start. Meanwhile, the next chunk of data can be read, resulting in potentially two activities executing concurrently. After the computation on the

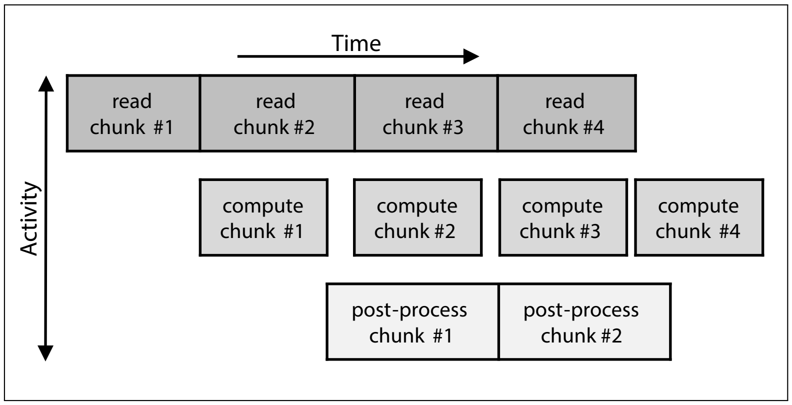

Figure 3.21: **A pipeline to overlap I/O and computations** – There are three activities: read a chunk of data, perform computations on this data and post-process the results. After a start-up phase, these activities can be executed simultaneously, but on different chunks of data.

first chunk of data has finished, the post-processing phase can start, while at the same time, the third chunk of data is read and processing of the second chunk of data starts soon after. At this point, there are three overlapping activities. This continues until the last chunk of data is read and the pipeline starts to wind down.

The code in Figure 1.11 demonstrates how these activities are parallelized using three parallel sections. Through various shared status arrays, the three phases are synchronized.

This approach works, but there is no good way to handle a load-imbalance. Another issue is that the communication of these phases through the status arrays is cumbersome. A polling mechanism is needed to check if the status of a phase has changed, and the OpenMP flush construct is required to ensure changes are made visible to the threads executing the other phases. Although less of a concern, the algorithm also requires either one, or otherwise at least three threads to work correctly.

3.4.1 Using Tasks and Task Dependences

As shown next, tasks provide a much more natural vehicle to implement such a pipeline. The code is listed in Figure 3.22.

```
#pragma omp parallel default(none) \
                  shared(fp_read, n_io_chunks, n_work_chunks) \
                  shared(a, b, c) \
                  shared(status_read, status_processing) \
                  shared(status_postprocessing)
{
  #pragma omp single nowait
  {
    for (int64_t i=0; i<n_io_chunks; i++) {

      #pragma omp task depend(out: status_read[i]) \
                     priority(20)
      {
         (void) read_input(fp_read, i, a, b, &status_read[i]);
      } // End of task reading in a chunk of data

      #pragma omp task depend(in: status_read[i]) \
                       depend(out: status_processing[i]) \
                       priority(10)
      {
         (void) compute_results(i, n_work_chunks, a, b, c,
                                &status_processing[i]);
      } // End of task performing the computations

      #pragma omp task depend(in: status_processing[i]) \
                       priority(5)
      {
         (void) postprocess_results(i, n_work_chunks, c,
                                    &status_postprocessing[i]);
      } // End of task postprocessing the results

    } // End of for-loop
  } // End of single region
} // End of parallel region
```

Figure 3.22: **Overlapping I/O and computations using tasks** – Each phase has been turned into a task. By using task dependences, the correct order of execution is guaranteed. In addition to this, priorities are used to give hints to the runtime system.

As we've seen several times before, the entire parallel region consists of a gigantic single region, but that does not mean this code executes sequentially. On the contrary, this is how parallelism is created. To understand why that is, let's go through the core part of this code fragment line by line.

The for-loop spans lines 9 – 32. The assumption is that `n_io_chunks` of data can be read independently. Since the loop is within a single region, one thread executes all the loop iterations. This thread encounters the tasks, generates the corresponding code, and puts them in the queue.

The queued tasks are made available for execution in, what is called a *task synchronization construct*, which in this case is the barrier implied at the end of the parallel region at line 34. In a task synchronization construct, any thread that is available, may be put to work. More on this can be found in Section 3.7, starting on page 141.

The first task spans lines 11 through 15. This is the task reading the chunk of data corresponding to iteration `i` of the loop. The data is read from the file associated with file pointer `fp_read` and stored in array elements `a[i]` and `b[i]`. Upon successful completion, a status value is written into array element `status_-read[i]`. This value is meant to be checked later on to ensure correct execution.

The new element here is the use of the `depend` clause at line 11. It is used to define *task dependences*.[8] There is also another new clause, the `priority` clause at line 12. We'll get to that soon, but first look at task dependences in more detail.

The `depend` clause takes a dependence type, which can be `in`, `out`, or `inout`, with a variable. The variable is used to set up a dependence between two tasks, while the type defines the direction of the dependence.

In this case, the second task, `compute_results`, needs to be made dependent upon the first task, `read_input`. This is controlled through variable `status_read[i]`. By using the `in` type for the second task, and type `out` on the first task, it is guaranteed that the second task does not start before the first task. In a similar manner, a dependence between the second and third task is set up. Variable `status_processing[i]` is used to define the dependence. By using type `out` on the second task, and type `in` on the third task, the third task starts only upon completion of the second task. The loop variable `i` is included in the dependence variables to allow the runtime system to schedule multiple similar tasks at the same time.

[8]This is another use of the `depend` clause. In Section 2.4.4, it is shown how this clause is used to parallelize a `doacross` loop.

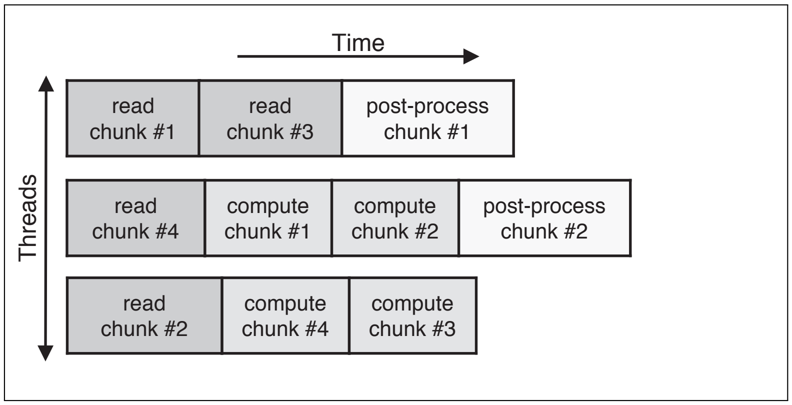

Figure 3.23: **The dynamic behavior of the pipeline** – It is assumed that three threads are used. As long as the dependences are obeyed, the loop iterations can be executed in an arbitrary order, as demonstrated here.

It is however worth noting that these three status arrays are not strictly necessary. Here they are used to verify correct execution afterwards, but as far as the dependences go, simply using the loop variable `i` instead is sufficient.[9]

The a-synchronous execution of the pipeline is illustrated in Figure 3.23. In the diagram, three threads are used, but this is only for the sake of the example. This code runs fine using any number of threads. The key thing is that for any given iteration, the three different phases must be executed in the correct order. This is achieved using the dependences. Aside from this, there are no restrictions on the order of execution and due to the more flexible scheduling, load balancing is less of an issue.

As mentioned above, in this code, *task priorities* are used as well. These are specified through the `priority` clause. This clause takes a non-negative scalar integer expression, specifying the priority. For example, at line 12, the priority of this task is set to 20. The second task has a priority of 10, and the third task has a priority of 5.

These priorities are hints to the runtime system to assist the scheduler to determine which tasks to choose to execute. The priority values are relative numbers only. In this example we want to emphasize that reading the data is most impor-

[9]Error handling can also be handled through cancellation, covered in Section 2.4.2.

tant (because everything else depends on it) and the post-processing part is least important. The priorities are suggestions only and correct execution of the program may not depend on them. In case there is a certain execution order, dependences must be used to enforce this.

Prior to program start up, environment variable `OMP_MAX_TASK_PRIORITY` must be set to the maximum value used. By default this variable is set to zero, that is, there are no higher priority tasks. Runtime function `omp_get_max_task_-priority()` can be used to return the maximum value set through this environment variable.[10]

The solution shown in Figure 3.22 works fine. A major advantage over the original approach using parallel sections is that the explicit flush construct is no longer needed. Load balancing is also less of a concern and this code executes correctly using any number of threads.

The one possible drawback is that there are three tasks for each chunk of data read, and task scheduling overhead could become an issue. Especially if each task does not perform a sufficient amount of work. Features like the `final` and `if` clauses may be used to control this, but there is an alternative solution worth considering and discussed in the next section.

3.4.2 Using the Taskloop Construct

To reduce the tasking overhead, a solution may be to place all three function calls within a single task and apply similar constructs as before. There is, however, a more elegant and powerful construct, available since OpenMP 4.5. It is called the *taskloop construct* and is meant to simplify using tasks in a loop context. It is definitely much more than syntactic sugar and relieves the user from relatively low level coding details when using tasks.

The `taskloop` construct applies to a (nested) loop. The loop iterations are partitioned into tasks and executed as tasks by the runtime system. Many clauses from the tasking construct are inherited, but there are two new clauses worth mentioning.

Figure 3.24 shows the relevant code fragment using the taskloop construct. With one exception, the first few lines are the same as in the previous example. The status arrays, used to express the dependences, are gone. As mentioned earlier,

[10]There is no equivalent to set the priorities at runtime. This was considered to be too complex to implement.

```
 1 #pragma omp parallel default(none) \
 2                    shared(fp_read, n_io_chunks, n_work_chunks) \
 3                    shared(a, b, c)
 4 {
 5   #pragma omp single nowait
 6   {
 7     #pragma omp taskloop num_tasks(n_io_chunks/10) grainsize(50)
 8     for (int64_t i=0; i<n_io_chunks; i++) {
 9           (void) read_input(fp_read, i, a, b);
10           (void) compute_results(i, n_work_chunks, a, b, c);
11           (void) postprocess_results(i, n_work_chunks, c);
12     } // End of for-loop
13   } // End of single region
14 } // End of parallel region
```

Figure 3.24: **Overlapping I/O and computations using the taskloop construct** – Each triplet, with the functions called in sequence, is a task. This construct supports many clauses. To reduce the overhead and improve efficiency, the `num_tasks` and `grainsize` clauses can be used to fine-tune the performance. Note the absence of the dependence arrays.

this may be handled in a more simple way, but in any case, a variable to specify the dependences is no longer needed.

The new element is the use of the `taskloop` construct at line 7. Without the additional clauses, this creates a task for each triplet of the function calls inside the loop body. As illustrated in Figure 3.25, this ensures each sequence with the three calls occurs in the right order.

With the new approach, larger units of work are scheduled, and the overhead is reduced. As is typical with tasking, there is the risk that too many tasks are generated (`n_io_chunks` in this case) and the overhead of managing these tasks could slow down the execution. This is why the `num_tasks` and `grainsize` clauses are so convenient.

The `num_tasks` clause takes an integer expression. The runtime system generates as many tasks as this expression evaluates to, or fewer if there are less loop iterations. The `grainsize` clause takes an integer expression, the *grain-size*. The number of iterations assigned to a task is the minimum of *grain-size* and the number of loop iterations, but does not exceed twice the value of *grain-size*. These two

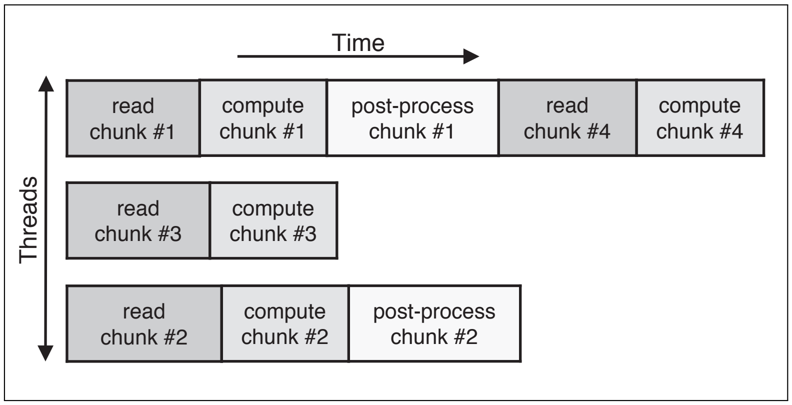

Figure 3.25: **The dynamic behavior of the pipeline using the taskloop construct** – Three threads are used. The loop iterations can be executed in any order, but for each iteration value, the triplet of functions is executed in order.

clauses provide powerful ways to fine-tune the efficiency when using the `taskloop` construct. They are discussed in more detail in Section 3.8, starting on page 146.

Their usage is shown at line 7. The number of tasks is restricted to be 1/10 of the maximum number of `n_io_chunks`. This number of 10% is not based upon actual measurements, or a recommendation. We merely want to demonstrate that this value may be an expression that depends on the runtime value(s) of other variables. The grain-size is set to 50, another arbitrary choice. Possibly, one would like to make this value dependent upon the value of certain program variables and that is supported too.

The effect of the `grainsize` clause is to increase the amount of work performed per task. In general this is a good thing to do, but if the workload is not equally balanced across the loop iterations, a too high value for the `grainsize`, may result in a load-balancing issue. That is ironic, because one of the strengths of tasking is that load-balancing tends to be less of an issue.

In any case, it is strongly recommend to conduct performance experiments to find the optimal values for these parameters. Don't be surprised if they turn out to be dependent upon several factors, and some differentiation depending upon the details of the computer system and OpenMP runtime implementation, may be needed.

3.4.3 Closing Comments on the Pipeline Example

Both versions that implement a pipeline to overlap I/O and computations, use tasks, but the behavior is different.

The first version has the maximum flexibility to schedule the tasks and is most robust in handling load-balancing issues. A downside is that it is easy to generate too many tasks and fine-tuning through the `final` and `if` clauses may be required. The second version uses the powerful `taskloop` construct. This relieves the user of several details to worry about, but load-balancing may be more of an issue. Through the use of the `num_tasks` and `grainsize` clauses, the overhead of tasking can be kept low.

Given the difference in the pros and cons it is difficult to give a recommendation. If the user has more insight into the application characteristics, this may help to select the optimal version.

3.5 The Data Environment with Tasks

So far, we have glossed over the data environment, or *scoping*, with tasking. It is important enough to elaborate upon, though.

Although some of the default scoping rules are intuitively clear, in general it is *strongly* recommended to use the `default(none)` clause and explicitly specify the data sharing attributes of the variable(s). Having said that, let's dig a little deeper into the default rules.

Some rules that apply to the parallel region carry over. For example, global and static variables are automatically shared. Local variables are private by default, by their very nature, as defined in C/C++ within the scope of curly braces (and unlike Fortran). If shared scoping is not inherited, orphaned task variables are `firstprivate` by default, while non-orphaned variables inherit the shared attribute. Variables are `firstprivate`, unless they are shared in the enclosing context.

This is illustrated in the example shown in Figure 3.26, where the scoping and values of the variables are as follows:

- Variable `a` is a global variable and shared. Because the initial value is not changed, it has a value of 1 in all tasks.
- Variable `b` is more challenging. It starts as a shared variable, but is made `private` at line 10. This makes the value undefined within the inner parallel

```
 1 int a = 1;
 2
 3 int main(int argc, char **argv)
 4 {
 5   int b = 2;
 6   int c = 3;
 7
 8   #pragma omp parallel shared(b)
 9   {
10     #pragma omp parallel private(b)
11     {
12       int d = 4;
13       #pragma omp task
14       {
15          int e = 5;
16
17          // Scope of a: shared,         value of a: 1
18          // Scope of b: firstprivate,  value of b: undefined
19          // Scope of c: shared,         value of c: 3
20          // Scope of d: firstprivate,  value of d: 4
21          // Scope of e: private,        value of e: 5
22
23       } // End of task
24     } // End of inner parallel region
25   } // End of outer parallel region
26 }
```

Figure 3.26: **An example of the default scoping rules for tasking** – This code demonstrates the automatic scoping for several variables. Although the rules are fairly straightforward, it is still recommended to explicitly scope variables.

region spanning lines 10 – 24. Although by default it is made `firstprivate` within the task, the damage has been done and the value is undefined.

To make the value of `b` defined within the task, the solution is to scope it as `firstprivate(b)` at line 10.

- By virtue of the default scoping rules for parallel regions, variable `c` has the `shared` attribute within the outer and inner parallel regions. This is inherited within the task and the value is 3.

- Variable `d` is a `private` variable inside the inner parallel region and therefore scoped as `firstprivate` within the task. Since the value is set to 4 at line 12, this is also the value inside the task.

- Variable `e` is declared within the task and therefore a local variable. This makes it a `private` variable with a value of 5.

The above example assumes a fairly deep understanding of the default scoping rules. A possible pitfall is that misinterpretation of a rule can lead to unexpected behavior and in general one should not count on a compiler to detect this.

Variable `b` is a good example how risky this is. By omitting the scoping on the task definition, an undefined value is used. In this simple case, a compiler might warn against this, but in a more complex code structure, this can easily be much more difficult, or even impossible, to detect.[11]

In summary, one can rely on the default rules, but using the `default(none)` clause can save a significant debugging effort at the expense of some more work in the development phase.

3.6 What is a Task?

Many aspects of tasking were discussed already and several examples of how to use tasks have been shown already. Hopefully these are sufficient to get started, but because several related topics were either skipped, or covered very briefly, a somewhat more formal description of the main constructs and features is given in this section.

To start with, what is a task exactly and how is it defined in an application? In Figure 3.27, the syntax of the task construct is shown. A task consists of the source code in the associated structured block, the data environment as defined through the implicit and explicit data scoping rules, plus the relevant ICVs. When a thread encounters a task, it packages the code and the data environment, such that the task is ready for execution. How and when the task is then executed, is discussed in more detail in Section 3.7.

[11]We know of at least one compiler that indeed detects this bug.

```
#pragma omp task [clause[[,] clause]...]
      structured block
--------------------------------------
!$omp task [clause[[,] clause]...]
      structured block
!$omp end task
```

Figure 3.27: **Syntax of the task construct in C/C++ and Fortran** – This defines the structured block of code to be a task. A task consists of the code to execute, the data environment, plus the relevant ICVs.

```
private (list)
firstprivate (list)
shared (list)
default(shared | none)                                (C/C++)
default(shared | firstprivate | private | none)       (Fortran)
if ([ task :] scalar-logical-expression)
final (scalar-logical-expression)
mergeable
depend (dependence-type : list)
priority (priority-value)
untied
```

Figure 3.28: **The clauses supported by the task construct** – With the exception of the `untied` clause, the clauses specific to tasking have been introduced in the previous examples, but additional information is given in the text.

Tasks can be dynamically nested, for example in a recursive algorithm, but they may also be lexically nested within other tasks, parallel regions, and worksharing constructs.

The `task` construct supports various clauses. They are listed in Figure 3.28. The data scoping clauses `default`, `private`, `firstprivate` and `shared` are the same as used on other constructs, but there are several clauses specific to tasks. With the exception of the `untied` clause, all of these have been used and discussed in the examples in the previous section. Below they are described in a somewhat more formal way:

- The `if` clause takes the optional `task` keyword. Support for a keyword is a general feature of this clause and may be needed in case composite, or combined, constructs are used.

The scalar expression on this clause must evaluate to `true` or `false`. In the case of the latter, the encountering task is suspended and the new task is executed immediately. The parent task resumes, once the new task has completed.

This feature can be used by an implementation to improve the performance by avoiding queueing tasks that are too small.

The difference with the `final` clause below, is that, with the `if` clause, the child tasks are not affected. This is why it was *not* used in the example shown in Figure 3.19. Due to the nature of the quicksort algorithm, once the length of the array drops below the threshold, the length for child tasks is also below this value.

- The `final` clause is a feature to fine-tune the performance of tasks. The clause takes a scalar expression evaluating to `true` or `false`. If this evaluates to `true`[12], the task is considered to be final and no additional tasks are generated.

  This is like a deeper version of the `if` clause, where nested tasks are not affected. With the `final` clause, all child tasks are final tasks too.

- The `mergeable` clause was introduced to reduce the data requirements. Since each task comes with its own data environment, the program may require a significant amount of memory. The `mergeable` clause informs the compiler the data environments can be merged.

- The `depend` clause in the context of tasking is used to specify dependences between tasks.[13] The clause takes a keyword to describe the type of dependence, and a list of variables it applies to.

  The dependence type can be `in`, `out`, or `inout`. The latter is actually a leftover from the initial support for this feature. It is identical to type `out`.

  The two types `in` and `out` are strongly related. They define the direction of the dependence. Through the `out` type, it is specified that the variable(s) in the list are output variables from that task. Any task with dependency type

[12]Note the difference between the condition `false` and `true` for the `if` and `final` clause to be triggered.

[13]The `depend` clause is used on other constructs too.

in and the same variable, depends on this task and will not be executed until the task with the corresponding out type has completed.

The list item(s) may include array sections. See also Section 6.3.4 for more information on array sections.

- To indicate which task(s) are relatively important, the priority clause can be used. This is a hint to the OpenMP runtime system to help scheduling the execution of the various tasks.

  The clause takes a scalar expression as an argument. This must evaluate to a non-negative integer value and sets the priority for that task.

  The priorities are relative values. A task with a higher priority is considered to be more important than a task with a lower priority.

  Be aware that the value for the priority may not exceed the value set through environment variable OMP_MAX_TASK_PRIORITY, which is set to zero by default. The consequence is that this variable *must* be set when using priorities.

  The function omp_get_max_task_priority() returns this upper limit. There is no "set" counterpart to change the maximum priority during the execution of the program.

- By default, tasks are tied, but the untied clause changes this to untied. More details on tied and untied tasks can be found in Section 3.7.

The if, final, mergeable, and priority clauses are provided to optimize the performance of the tasking mechanism. These clauses are used to reduce the number of tasks generated, task scheduling overhead, memory requirements, and to express runtime scheduling preferences. An implementation is however free to ignore them.

3.7 Task Creation, Synchronization, and Scheduling

By design, task creation, scheduling, and execution details are meant to be a black box. This not only simplifies the life of the user, but also gives the implementation the freedom to adapt to the underlying architecture and evolve over time, without any impact on the portability of the code.

When a thread encounters a task construct, a new task is created and queued for execution by any thread in the team. Execution of the task could be immediate, or deferred until later, depending on the task scheduling details and thread availability.

Task synchronization construct	**Description**
barrier	either an implicit, or explicit barrier.
taskwait	wait on the completion of child tasks of the current task.
taskgroup	wait on the completion of child tasks of the current task, *and* their descendants.

Figure 3.29: **The three task synchronization constructs** – Completion of a task is guaranteed at a task synchronization point, but whether descendant tasks are affected also depends on the construct.

The thing that surprises users most the first time they consider using tasking, is that, although tasks are generated when the `task` construct is encountered, they are not necessarily executed then. More often than not, execution occurs "later," or in tasking terminology, the execution is *deferred*.

To be more precise, tasks are only guaranteed to be completed at program exit and at one of the three constructs listed in Figure 3.29. They are called *task synchronization constructs*.

The `barrier` construct can either be implied, at the end of the parallel region for example, or inserted explicitly. In both cases, the tasks are guaranteed to be completed then. The immediate consequence of this is that all tasks are completed at the end of a parallel region, since it has an implied barrier that may not be omitted.

The (lack of) implied barrier is why there is a difference between using the `single` construct versus the `master` construct. The latter has no implied barrier and tasks are not guaranteed to be completed upon exiting the master region.

There are also cases where the tasks need to completed, before the next barrier is encountered. For example, in the code shown in Figure 3.5 on page 108, the two tasks must be completed before the next (print) statement. This is where the `taskwait` construct comes to the rescue. If completion of the tasks needs to be enforced, this construct can be used. It is a stand-alone directive and ensures all child tasks of the current task are completed.

Another situation where the `taskwait` construct could be needed is when the `nowait` clause has been used. Since there is no longer an implied barrier, this clause has the side effect that there is no guarantee that all tasks have completed at the end of the construct to which the clause applies. Should this still be needed,

#pragma omp taskgroup *new-line*   *structured block*
!$omp taskgroup   *structured block* **!$omp end taskgroup**

Figure 3.30: **Syntax of the taskgroup construct in C/C++ and Fortran** – This defines the structured block of code to be a taskgroup and provides deep synchronization by ensuring the completion of all child tasks, as well as their descendants.

the `taskwait` construct can be used to enforce completion of all child tasks.

The `taskwait` construct works well, unless there are descendant tasks. Those are not affected and if deeper synchronization of tasks is needed, the `taskgroup` construct should be used. This guarantees that also all descendant tasks are completed. The syntax is given in Figure 3.30.

Now that we have seen how tasks are scheduled and what mechanisms are available to enforce the completion of tasks, it is time to look at *task scheduling*.

Although not required by the specifications, all current tasking implementations use a queueing system. With this, generated tasks are put in a queue and at some point, executed.[14] For the runtime system, managing this queueing system takes time, especially since it is common to have many tasks, and threads need to be assigned to tasks. All of this adds to the parallel overhead.

This is why several features to reduce this overhead are available, such as the `if` and `final` constructs. Both can be used to prevent the queueing system from being flooded with either too small tasks, too many tasks, or both.

To formalize this and support several performance-related features, there are four tasking related concepts in the specifications:

- *A task region* - A region consisting of all code encountered during the execution of a task.

- *An undeferred task* - This is a task for which execution is not delayed, or "deferred," with respect to its generating task region. The generating task region is suspended until execution of the undeferred task has completed.

[14]The details of such queueing systems are complex and beyond the scope of this book.

In case the `if` clause on the `task` construct evaluates to `false`, an *undeferred* task is generated and the encountering thread suspends the current task region. Execution resumes only after the generated task has completed.

- *An included task* - This is an undeferred task, executed immediately by the encountering thread.

  If the `final` clause evaluates to `true`, the generated task is a final task. All tasks encountered during the execution of a final task are also final and included.

- *A tied/untied task* - Upon resuming a suspended task region, a tied task *must* be executed by the same thread again. With an untied task, there is no such restriction and any thread in the team can resume execution of the suspended task.

The above means that, while the `if` clause is "shallow" and only affects the encountering task, the `final` clause is a "deep" feature. Once a task is final, all child tasks are final too.

When using tasks, *task scheduling points* are places in the application where a thread executing a task, may temporarily suspend the current task, and switch to execute a different task. It can either begin, or resume this other task. A task scheduling point is included at the following locations:

- The point immediately following the generation of an explicit task.
- After the point of completion of a `task` region.
- In a `taskyield` region.
- In a `taskwait` region.
- At the end of a `taskgroup` region.
- In an implicit and explicit `barrier` region.
- When using the support for accelerators:
  - The point immediately following the generation of a `target` region.
  - At the beginning and end of a `target data` region.

#pragma omp taskyield *new-line*
!$omp taskyield

Figure 3.31: **Syntax of the taskyield construct in C/C++ and Fortran** – This defines an explicit task scheduling point in the application.

- In a `target update` region.
- In a `target enter data` region.
- In a `target exit data` region.
- In the `omp_target_memcpy()` runtime function.
- In the `omp_target_memcpy_rect()` runtime function.

With the exception of the `taskyield` construct, all of these scheduling points are implied. They are there, but not visible. This is actually the reason the standalone `taskyield` construct was added. With this construct, the user can add an explicit task scheduling point, allowing the thread to suspend the current task and switch to another task. The syntax is given in Figure 3.31.

One of the clauses listed in Figure 3.28 on page 139 is the `untied` clause. By default, a task is "tied," which means that the *same* thread that executed the task must resume execution after the task has been suspended in a task scheduling point.[15] This could restrict the runtime scheduler and negatively impact performance.

Through the `untied` clause, this limitation is lifted and any thread may resume the execution of the suspended task. Clearly there are performance advantages to allow this kind of scheduling freedom, but there are some important caveats too.

First of all, `threadprivate` variables may not be used. Secondly, explicitly using the thread ID, for example through function `omp_get_thread_num()`, is not allowed and third, one must be careful using locks, including critical regions.

Last, but not least, untied tasks are an optional feature. An implementation is free to ignore the clause and make all tasks tied.

[15]This thread need not be the same thread that created the task.

#pragma omp taskloop *[clause[[,] clause]. . .]*	*new-line*
for-loops	
!$omp taskloop *[clause[[,] clause]. . .]*	
do-loops	
[!$omp end taskloop]	

Figure 3.32: **Syntax of the taskloop construct in C/C++ and Fortran** – The loop iterations are executed in parallel using tasks. At runtime, the iterations are distributed over the tasks.

private *(list)*	
firstprivate *(list)*	
lastprivate *(list)*	
shared *(list)*	
default(shared | **none)**	(C/C++)
default(shared | **firstprivate** | **private** | **none)**	(Fortran)
if *([* `taskloop` *:] scalar-logical-expression)*	
grainsize *(grain-size)*	
num_tasks *(num-tasks)*	
collapse *(n)*	
final *(scalar-logical-expression)*	
priority *(priority-value)*	
untied	
mergeable	
nogroup	

Figure 3.33: **The clauses supported by the taskloop construct** – The `nogroup`, `grainsize`, and `num_tasks` clauses are specific to this construct. All other clauses are supported on at least one other construct.

3.8 The Taskloop Construct

The `taskloop` construct has been added in OpenMP 4.5 and combines the ease of use of the parallel loop with the flexibility of tasking. In the example in Section 3.4, it was used to parallelize the loop in this code.

This construct simplifies using tasks in a (perfectly nested) loop. The syntax of the `taskloop` construct is given in Figure 3.32. The compiler and runtime system create and manage the tasks, while the user does not need to worry about the tasking details.

The `taskloop` construct supports the clauses listed in Figure 3.33. The data-sharing clauses are similar as on other constructs, and as with other constructs, the `if` clause supports the optional `taskloop` keyword. The `collapse`, `final`, `priority`, `untied`, and `mergeable` clauses are the same as on the `task` construct. There are, however, also three clauses that are unique to this construct:

- The `grainsize` clause takes a positive integer expression, the *grain-size*. The number of loop iterations assigned to a task is the minimum of this *grain-size* and the number of loop iterations, but does not exceed twice the value of *grain-size*.

  This clause can be used to adjust the granularity of the work performed by the tasks. It provides an easy way to avoid that the work performed per task is too small

  In absence of this clause, an implementation-dependent default is used.

- The `num_tasks` clause takes a positive integer expression. At runtime, as many tasks as this expression evaluates to, are generated, or fewer, if there are less loop iterations. This clause can be used to limit the number of tasks generated.

  In absence of this clause, an implementation-dependent default is used.

- With the `nogroup` clause, the `taskloop` construct is not embedded in an implied `taskgroup` construct.

The `nogroup` clause is needed in the following scenario. Since by default, the `taskloop` construct is embedded in an implicit `taskgroup` construct, there is a task synchronization point at the end. In case other tasks need to execute concurrently with the tasks executing the `taskloop` construct, this creates a problem. The `taskloop` loop could be executed first, followed by the other task(s). To make matters worse, task priorities are not handled properly either.

With the `nogroup` clause, there is no implied `taskgroup` construct and all tasks can run simultaneously, plus the priorities will be honored, if the implementation supports it.

```
 1 #pragma omp parallel firstprivate(n)
 2 {
 3   #pragma omp single
 4   {
 5
 6     #pragma omp taskloop firstprivate(n) nogroup priority(10)
 7     for (int i=0; i<n; i++)
 8     {
 9        <body of loop>
10     } // End of taskloop
11
12     #pragma omp task priority(50)
13     {
14        <body of task>
15     } // End of task
16
17   } // End of single region
18 } // End of parallel region
```

Figure 3.34: **An example of the nogroup clause on the taskloop construct** – This code demonstrates the use of the `nogroup` clause. Thanks to this clause, the task below the `taskloop` construct is executed simultaneously and if task priorities are implemented, even starts first.

The use of the `nogroup` clause is illustrated in Figure 3.34. Assuming task priorities are supported, and environment variable `OMP_MAX_TASK_PRIORITY` has been set to at least 50, the task defined at lines 12 – 15 is executed first. Without the `nogroup` clause on the `taskloop` construct at line 6, this is not be the case and the isolated task at line 12 is executed last.

There is a second variant of the `taskloop` construct, called `taskloop simd`. The syntax of this construct is given in Figure 3.35. SIMD is explained in great detail in Chapter 5. On those processors supporting SIMD instructions, this construct allows the compiler to exploit them within the generated tasks. This construct supports both the clauses from the `taskloop` construct, as well as the clauses for the `simd` directives. The `collapse` clause is applied only once.

#pragma omp taskloop simd *[clause[[,] clause]. . .]* *new-line*   *for-loops*
!$omp taskloop simd *[clause[[,] clause]. . .]*   *do-loops* **[!$omp end taskloop simd]**

Figure 3.35: **Syntax of the taskloop simd construct in C/C++ and Fortran** – The loop iterations are executed in parallel using tasks. At runtime, the loop iterations are distributed over the tasks and the resulting loop uses SIMD instructions.

3.9 Concluding Remarks

This chapter has covered all aspects of tasking. Hopefully the examples, plus the additional information presented here, are sufficient to get started.

Tasks work through a queueing system, invisible to the user. A thread that encounters a tasks, generates the code to execute it, including the data environment plus ICVs, and puts it in a queue.

At task synchronization points, threads execute tasks that are waiting in the queue. These points are implied on several constructs, but through the `taskwait` and `taskgroup` constructs, the user can enforce completion of the tasks.

The runtime system is in charge of managing the queue(s), but the user has indirect, yet powerful, controls. The `if`, `final`, `mergeable`, and `priority` clauses allow the user to reduce the pressure on the queueing system, avoid tasks get too small, and help the runtime scheduler to decide which task(s) to execute. Especially if the performance is disappointing, these clauses may make quite a difference and some experimentation is highly recommended.

Task dependences provide a very powerful mechanism to create a chain of tasks, where execution happens in the right order, while leveraging the parallelism in the algorithm.

The taskloop construct makes it easier to leverage tasks in a loop and more efficiently handle a load-imbalance.

An often asked question is whether tasks replace other constructs. The answer is no. Tasks are extremely flexible and powerful, but sometimes other constructs are easier to use and get the job done as well.