Parallel Programming in Fortran 95 using OpenMP

Miguel Hermanns
School of Aeronautical Engineering
Departamento de Motopropulsión y Termofluidodinámica
Universidad Politécnica de Madrid
Spain
email: hermanns@tupi.dmt.upm.es

19^{th} of April 2002

Contents

Chapter 1

OpenMP Fortran Application Program Interface

1.1 Introduction

In the necessity of more and more computational power, the developers of computing systems started to think on using several of their existing computing machines in a joined manner. This is the origin of parallel machines and the start of a new field for programmers and researches.

Nowadays parallel computers are very common in research facilities as well as companies all over the world and are extensively used for complex computations, like simulations of atomic explosions, folding of proteins or turbulent flows.

A challenge in parallel machines is the development of codes able of using the capabilities of the available hardware in order to solve larger problems in less time. But parallel programming is not an easy task, since a large variety of architectures exist. Mainly two families of parallel machines can be identified:

Shared-memory architecture : these parallel machines are build up on a set of processors which have access to a common memory. Usually the name of **SMP machines** is used for computers based on this architecture, where SMP stands for **Symmetric Multi Processing**.

Distributed-memory architecture : in these parallel machines each processor has its own private memory and information is interchanged between the processors through messages. The name of **clusters** is commonly used for this type of computing devices.

Each of the two families has its advantages and disadvantages and the actual parallel programming standards try to exploit these advantages by focusing only on one of these architectures.

In the last years a new industry standard has been created with the aim to serve as a good basis for the development of parallel programs on shared-memory machines: **OpenMP**.

1.1.1 Historical remarks

Shared-memory machines exist nowadays for a long time. In the past, each vendor was developing its own *"standard"* of compiler directives and libraries, which allowed a program to make use of the capabilities of their specific parallel machine.

An earlier standardization effort, ANSI X3H5 was never formally adopted, since on one hand no strong support of the vendors was existing and on the other hand distributed memory machines, with their own *more standard* message passing libraries PVM and MPI, appeared as a good alternative to shared-memory machines.

But in 1996-1997, a new interest in a standard shared-memory programming interface appeared, mainly due to:

1. A renewed interest from the vendors side in shared-memory architectures.

2. The opinion by a part of the vendors, that the parallelization of programs using message passing interfaces is cumbersome and long and that a more abstract programming interface would be desirable.

OpenMP[1] is the result of a large agreement between hardware vendors and compiler developers and is considered to be an *"industry standard"*: it specifies a set of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran and C/C++ programs.

OpenMP consolidates all this into a single syntax and semantics and finally delivers the long-awaited promise of single source portability for shared-memory parallelism. But OpenMP is even more: it also addresses the inability of previous shared-memory directive sets to deal with **coarse-grain parallelism**[2]. In the past, limited support for coarse grain work has led to developers to think that shared-memory parallel programming was inherently limited to **fine-grain parallelism**[3].

1.1.2 Who is participating

The OpenMP specification is owned, written and maintained by the **OpenMP Architecture Review Board**, which is a join of the companies actively taking part in the development of the standard shared-memory programming interface. In the year 2000, the permanent members of the OpenMP ARB were:

- US Department of Energy, through its ASCI program

- Compaq Computer Corp.

[1] **MP** stands for **Multi Processing** and **Open** means that the standard is defined through a specification accessible to anyone.

[2] **Coarse-grain parallelism** means that the parallelism in the program is achieved through a decomposition of the target domain into a set of subdomains that is distributed over the different processors of the machine.

[3] **Fine-grain parallelism** means that the parallelism in the program is achieved by distributing the work of the do-loops over the different processors, so that each processor computes part of the iterations.

- Fujitsu
- Hewlett-Packard Company
- Intel Corp.
- International Business Machines
- Kuck & Associates, Inc.
- Silicon Graphics Incorporate
- Sun Microsystems

Additionally to the OpenMP ARB, a large number of companies contribute to the development of OpenMP by using it in their programs and compilers and reporting problems, comments and suggestions to the OpenMP ARB.

1.1.3 About this document

This document has been created to serve as a good starting point for Fortran 95 programmers interested in learning OpenMP. Special importance has been given to graphical interpretations and performance aspects of the different OpenMP directives and clauses, since these are lacking in the OpenMP specifications released by the OpenMP ARB[4]. It is advisable to complement the present document with these OpenMP specifications, since some aspects and possibilities have not been addressed here for simplicity.

Only the Fortran 95 programming language is considered in the present document, although most of the concepts and ideas are also applicable to the Fortran 77 programming language. Since the author believes in the superiority of Fortran 95 over Fortran77 and in the importance of a good programming methodology, the present document only presents those features of OpenMP which are in agreement with such a programming philosophy. This is the reason why it is advisable to have also a look at the OpenMP specifications, since the selection of the concepts presented here are a personal choice of the author.

Since the existing documentation about OpenMP is not very extensive, the present document has been released for free distribution over the Internet, while the copyright of it is kept by the author. Any comments regarding the content of this document are welcome and the author encourages people to send constructive comments and suggestions in order to improve it.

At the time of writing this document (winter 2001-spring 2002) two different OpenMP specifications are used in compilers: version 1.1 and version 2.0. Since the latter enhances the capabilities of the former, it is necessary to differentiate what is valid for each version. This is accomplished by using a different color for the text that only applies to the OpenMP Fortran Application Program Interface, version 2.0.

[4]It has no sense that performance issues are addressed in a specification, since they are implementation dependent and in general different for each machine.

1.2 The basics

OpenMP represents a collection of compiler directives, library routines and environment variables meant for parallel programming in shared-memory machines. A chapter is going to be devoted to each of these elements, but before starting with the review of the available compiler directives, it is necessary to have a look at some basic aspects of OpenMP.

Although named as *"basic aspects"*, the information presented in this section is the fundamental part of OpenMP which allows the inclusion of OpenMP commands in programs and the creation as well as destruction of parallel running regions of code.

1.2.1 The sentinels for OpenMP directives and conditional compilation

One of the aims of the OpenMP standard is to offer the possibility of using the same source code lines with an OpenMP-compliant compiler and with a normal compiler. This can only be achieved by hiding the OpenMP directives and commands in such a way, that a normal compiler is unable to see them. For that purpose the following two **directive sentinels** are introduced:

```
!$OMP
```

```
!$
```

Since the first character is an exclamation mark "`!`", a normal compiler will interpret the lines as comments and will neglect their content. But an OpenMP-compliant compiler will identify the complete sequences and will proceed as follows:

`!$OMP` : the OpenMP-compliant compiler knows that the following information in the line is an OpenMP directive. It is possible to extend an OpenMP directive over several lines by placing the same sentinel in front of the following lines and using the standard Fortran 95 method of braking source code lines:

```
!$OMP PARALLEL DEFAULT(NONE) SHARED(A, B) PRIVATE(C, D) &
!$OMP REDUCTION(+:A)
```

It is mandatory to include a white space between the directive sentinel `!$OMP` and the following OpenMP directive, otherwise the directive sentinel is not correctly identified and the line is treated as a comment.

`!$` : the corresponding line is said to be affected by a **conditional compilation**. This means that its content will only be available to the compiler in case of being OpenMP-compliant. In such a case, the two characters of the sentinel are substituted by two white spaces so that the compiler is taking into account the line. As in the previous case, it is possible to extend a source code line over several lines as follows:

```
!$ interval = L * OMP_get_thread_num() / &
!$               (OMP_get_num_threads() - 1)
```

Again, it is mandatory to include a white space between the conditional compilation directive `!$` and the following source code, otherwise the conditional compilation directive is not correctly identified and the line is treated as a comment.

Both sentinels can appear in any column as long as they are preceded only by white spaces; otherwise, they are interpreted as normal comments.

1.2.2 The parallel region constructor

The most important directive in OpenMP is the one in charge of defining the so called **parallel regions**. Such a region is a block of code that is going to be executed by multiple threads running in parallel.

Since a parallel region needs to be created/opened and destroyed/closed, two directives are necessary, forming a so called **directive-pair**: `!$OMP PARALLEL`/`!$OMP END PARALLEL`. An example of their use would be:

```
!$OMP PARALLEL
  write(*,*) "Hello"
!$OMP END PARALLEL
```

Since the code enclosed between the two directives is executed by each thread, the message `Hello` appears in the screen as many times as threads are being used in the parallel region.

Before and after the parallel region, the code is executed by only one thread, which is the normal behavior in serial programs. Therefore it is said, that in the program there are also so called **serial regions**.

When a thread executing a serial region encounters a parallel region, it creates a team of threads, and it becomes the **master thread** of the team. The master thread is a member of the team as well and takes part in the computations. Each thread inside the parallel region gets a unique **thread number** which ranges from zero, for the master thread, up to $N_p - 1$, where N_p is the total number of threads within the team. In figure 1.1 the previous example is represented in a graphical way to clarify the ideas behind the parallel region constructor.

At the beginning of the parallel region it is possible to impose **clauses** which fix certain aspects of the way in which the parallel region is going to work: for example the scope of variables, the number of threads, special treatments of some variables, etc. The syntaxis to use is the following one:

```
!$OMP PARALLEL clause1 clause2 ...

...

!$OMP END PARALLEL
```

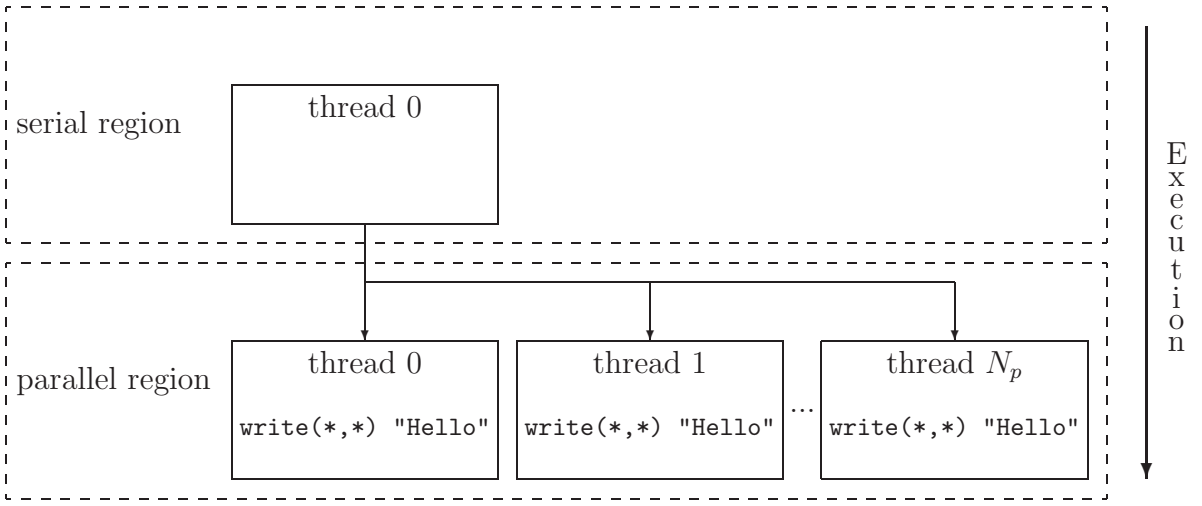

Figure 1.1: Graphical representation of the example explaining the working principle of the `!$OMP PARALLEL`/`!$OMP END PARALLEL` directive-pair.

Not all available clauses presented and explained in chapter 3 are allowed within the opening-directive `!$OMP PARALLEL`, only the following ones:

- `PRIVATE(`*list*`)`: see page 37.
- `SHARED(`*list*`)`: see page 38.
- `DEFAULT( PRIVATE | SHARED | NONE )`: see page 39.
- `FIRSTPRIVATE(`*list*`)`: see page 40.
- `COPYIN(`*list*`)`: see page 42.
- `REDUCTION(`*operator:list*`)`: see page 43.
- `IF(`*scalar_logical_expression*`)`: see page 46.
- `NUM_THREADS(`*scalar_integer_expression*`)`: see page 47.

The `!$OMP END PARALLEL` directive denotes the end of the parallel region. Reached that point, all the variables declared as local to each thread (`PRIVATE`) are erased and all the threads are killed, except the master thread, which continues execution past the end of the parallel region. It is necessary that the master thread waits for all the other threads to finish their work before closing the parallel region; otherwise information would get lost and/or work would not be done. This waiting is in fact nothing else than a **synchronization** between the parallel running threads. Therefore, it is said that the `!$OMP END PARALLEL` directive has an **implied synchronization**.

When including a parallel region into a code, it is necessary to satisfy two conditions to ensure that the resulting program is compliant with the OpenMP specification:

1. The `!$OMP PARALLEL`/`!$OMP END PARALLEL` directive-pair must appear in the same routine of the program.

2. The code enclosed in a parallel region must be a structured block of code. This means that it is not allowed to jump in or out of the parallel region, for example using a `GOTO` command.

Despite these two rules there are not further restrictions to take into account when creating parallel regions. Even though, it is necessary to be careful when using parallel regions, since it is easy to achieve non-correct working programs, even when considering the previous restrictions.

The block of code directly placed between the two directives `!$OMP PARALLEL` and `!$OMP END PARALLEL` is said to be in the **lexical extent** of the directive-pair. The code included in the lexical extent plus all the code called from inside the lexical extent is said to be in the **dynamic extent** of the directive-pair. For example:

```
!$OMP PARALLEL
  write(*,*) "Hello"
  call be_friendly()
!$OMP END PARALLEL
```

In this case the code contained inside the subroutine `be_friendly` is part of the dynamic extent of the directive-pair, but is not part of the lexical extent. These two concepts are important, since some of the clauses mentioned before apply only to the lexical extent, while others apply to the dynamic extent.

It is possible to nest parallel regions into parallel regions. For example, if a thread in a parallel team encounters a new parallel region, then it creates a new team and it becomes the master thread of the new team. This second parallel region is called a **nested parallel region**. An example of a nested parallel region would be:

```
!$OMP PARALLEL
  write(*,*) "Hello"
  !$OMP PARALLEL
    write(*,*) "Hi"
  !$OMP END PARALLEL
!$OMP END PARALLEL
```

If in both parallel regions the same number of threads N_p is used, then a total number of $N_p^2 + N_p$ messages will be printed on the screen[5]. The resulting tree structure is represented graphically in figure 1.2 for the case of using two threads at each level of nesting, $N_p = 2$. Also shown in the same figure is the screen output associated to each of the threads.

[5]There will be N_p messages saying `Hello` and N_p^2 messages saying `Hi`.

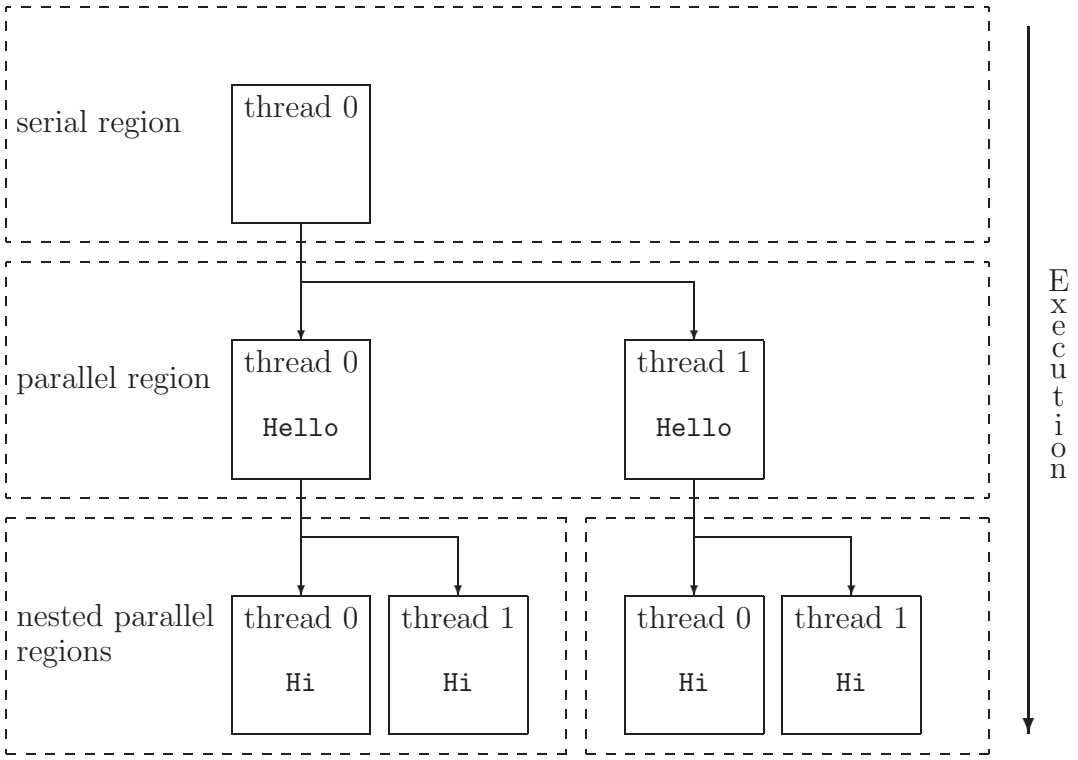

Figure 1.2: Graphical representation of the example explaining the concept of nested parallel regions.

Chapter 2

OpenMP constructs

If only the previous parallel region constructor would exist, then the only possible thing to do would be that all the threads perform exactly the same task, but this is not the aim of parallelism. Therefore, OpenMP defines additional constructs which allow to distribute a given task over the different threads and achieve in this way a real parallel working program.

Four different groups of OpenMP directives or constructs exist. Each group has a different aim and the selection of one directive or another inside the same group depends on the nature of the problem to be solved. Therefore, it is good to understand the principles of each of these directives in order to perform the correct choices.

2.1 Work-sharing constructs

The first group of OpenMP directives looks forward to divide a given work into pieces and to give one or more of these pieces to each parallel running thread. In this way the work, which would be done by a single thread in a serial program, is distributed over a team of threads achieving a faster running program[1].

All work-sharing constructs must be placed inside dynamic extends of parallel regions in order to be effective. If this is not the case, the work-sharing construct will still work, but a team with only one thread will be used. The reason is that a work-sharing construct is not able of creating new threads; a task reserved to the `!$OMP PARALLEL`/`!$OMP END PARALLEL` directive-pair.

The following restrictions need to be taken into account when using a work-sharing construct:

- Work-sharing constructs must be encountered by all threads in a team or by none at all.

[1]Obviously, this is only true, if the team of threads is executed on more than one processor, which is the case of SMP machines. Otherwise, when using a single processor, the computational overhead due to the OpenMP directives as well as due to the need of executing several threads in a sequential way lead to slower parallel programs than the corresponding serial versions!

- Work-sharing constructs must be encountered in the same order by all threads in a team.

All work-sharing constructs have an implied synchronization in their closing-directives. This is in general necessary to ensure that all the information, required by the code following the work-sharing construct, is uptodate. But such a thread synchronization is not always necessary and therefore a mechanism of suppressing it exists, since it is a resource wasting affair[2]. For that purpose a special clause exists which is linked to the closing-directive: `NOWAIT`. Further information regarding this clause is given in chapter 3, page 47.

2.1.1 `!$OMP DO/!$OMP END DO`

This directive-pair makes the immediately following do-loop to be executed in parallel. For example

```
!$OMP DO
  do i = 1, 1000
    ...
  enddo
!$OMP END DO
```

distributes the do-loop over the different threads: each thread computes part of the iterations. For example, if 10 threads are in use, then in general each thread computes 100 iterations of the do-loop: thread 0 computes from 1 to 100, thread 1 from 101 to 200 and so on. This is shown graphically in figure 2.1.

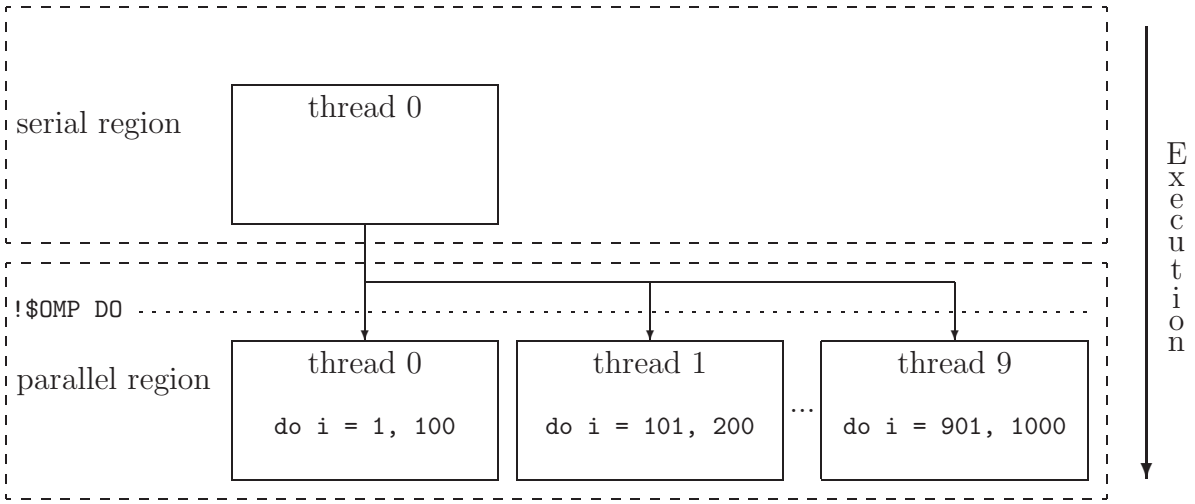

Figure 2.1: Graphical representation of the example explaining the general working principle of the `!$OMP DO/!$OMP END DO` directive-pair.

[2]Threads that reach the implied synchronization are idle until all the other threads reach the same point.

The way in which the work is distributed, and in general how the work-sharing construct has to behave, can be controlled with the aid of clauses linked to the opening-directive `!$OMP DO`. The syntax is as follows:

```
!$OMP DO clause1 clause2 ...

...

!$OMP END DO end_clause
```

Only the following clauses of those presented and explained in chapter 3 are allowed in the opening-directive `!$OMP DO`:

- `PRIVATE(`*`list`*`)`: see page 37.
- `FIRSTPRIVATE(`*`list`*`)`: see page 40.
- `LASTPRIVATE(`*`list`*`)`: see page 41.
- `REDUCTION(`*`operator:list`*`)`: see page 43.
- `SCHEDULE(`*`type, chunk`*`)`: see page 48.
- `ORDERED`: see page 52.

Additionally to the opening-directive clauses, it is possible to add to the closing-directive the `NOWAIT` clause in order to avoid the implied synchronization. Also implied to the closing-directive is an updating of the shared variables affected by the do-loop. When the `NOWAIT` clause is added, this implied updating is also suppressed. Therefore, care has to be taken, if after the do-loop the modified variables have to be used. In such a case it is necessary to add an implied or an explicit updating of the shared variables, for example using the `!$OMP FLUSH` directive. This side effect also happens in other directives, although it will not be mentioned explicitly again. Therefore, it is convenient to read the information regarding the `!$OMP FLUSH` directive in page 27 and the `NOWAIT` clause in page 47 for further information. Also the concepts explained in chapter 3 are of use to understand the impact of suppressing implied updates of the variables.

Since the work of a parallelized do-loop is distributed over a team of threads running in parallel, it has no sense that one or more of these threads can branch into or out of the block of code enclosed inside the directive-pair `!$OMP DO`/`!$OMP END DO`, for example using a `GOTO` command. Therefore, this possibility is directly forbidden by the OpenMP specification.

Since each thread is executing part of the iterations of the do-loop and the updates of the modifications made to the variables are not ensured until the end of the work-sharing construct, the following example will not work correctly using the `!$OMP DO`/`!$OMP END DO` directive-pair for its parallelization:

```
real(8) :: A(1000), B(1000)

do i = 1, 1000
  B(i) = 10 * i
  A(i) = A(i) + B(i)
enddo
```

because the correct value of the matrix `B` is not ensured until the end of the work-sharing construct `!$OMP END DO`.

The fact that the iterations of the do-loop are distributed over the different threads in an unpredictable way[3], discards certain do-loops from being parallelized. For example:

```
real(8) :: A(1000)

do i = 1, 999
  A(i) = A(i+1)
enddo
```

The reason for the incorrect working is that the value at the index `i+1` is needed in its unmodified state when performing the iteration `i`. This will lead to no trouble when executed in a serial way, but when running in parallel the unmodified state of `i+1` is not granted at the time of computing iteration `i`. Therefore, an unpredictable result will be obtained when running in parallel. This situation is known as **racing condition**: the result of the code depends on the thread scheduling and on the speed of each processor. By modifying the previous do-loop it is possible to achieve a parallelizable version leading to the following parallelized code[4]:

```
real(8) :: A(1000), dummy(2:1000:2)

!Saves the even indices
!$OMP DO
  do i = 2, 1000, 2
    dummy(i) = A(i)
  enddo
!$OMP END DO

!Updates even indices from odds
!$OMP DO
  do i = 0, 998, 2
    A(i) = A(i+1)
  enddo
!$OMP END DO

!Updates odd indices with evens
!$OMP DO
```

[3] Although the expression "*unpredictable*" usually means something negative, in this case it states something that is inherent in the philosophy of OpenMP, namely the separation between the parallel programming and the underlying hardware and work distribution.

[4] The presented modification is not the only possible one, but at least it allows the parallelization of the do-loop at expense of additional memory requirements.

```
  do i = 1, 999, 2
    A(i) = dummy(i+1)
  enddo
!$OMP END DO
```

Although not shown explicitly in the previous source code, the complete set of do-loops needs to be placed inside a parallel region; otherwise no parallelization of the do-loops takes place. This discards the definition of the matrices `A` and `B`, since these cannot have the private attribute (see chapter 3 for further information regarding private and shared variables).

The technique of splitting the do-loop into several separated do-loops allows to solve sometimes the problem, but it is necessary to evaluate the cost associated to the modifications in terms of time and memory requirements in order to see, if it is worth or not. Another example of problematic do-loop is:

```
real(8) :: A(0:1000)

do i = 1, 1000
  A(i) = A(i-1)
enddo
```

Now each iteration in the do-loop depends on the previous iteration, leading again to a dependency with the order of execution of the iterations. But this time the previously presented trick of splitting the do-loop does not help. Instead, it is necessary to impose an ordering in the execution of the statements enclosed in the do-loop using the `ORDERED` clause together with the `!$OMP ORDERED`/`!$OMP END ORDERED` directive-pair:

```
real(8) :: A(0:1000)

!$OMP DO ORDERED
  do i = 1, 1000
    !$OMP ORDERED
      A(i) = A(i-1)
    !$OMP END ORDERED
  enddo
!$OMP END DO
```

Doing so, no gain is to expect from the parallelization of the do-loop in the example, since the complete content of the do-loop has to be executed in an ordered way. Detailed information regarding the `!$OMP ORDERED`/`!$OMP END ORDERED` directive-pair can be found on page 30. When several nested do-loops are present, it is always convenient to parallelize the outer most one, since then the amount of work distributed over the different threads is maximal. Also the number of times in which the `!$OMP DO`/`!$OMP END DO` directive-pair effectively acts is minimal, which implies a minimal overhead due to the OpenMP directive. In the following example

```
do i = 1, 10
  do j = 1, 10
    !$OMP DO
```

```
      do k = 1, 10
        A(i,j,k) = i * j * k
      enddo
    !$OMP END DO
  enddo
enddo
```

the work to be computed in parallel is distributed $i \cdot j = 100$ times and each thread gets less than 10 iterations to compute, since only the innest do-loop is parallelized. By changing the place of the OpenMP directive as follows:

```
!$OMP DO
  do i = 1, 10
    do j = 1, 10
      do k = 1, 10
        A(i,j,k) = i * j * k
      enddo
    enddo
  enddo
!$OMP END DO
```

the work to be computed in parallel is distributed only once and the work given to each thread has at least $j \cdot k = 100$ iterations. Therefore, in this second case a better performance of the parallelization is to expect.

It is possible to increase even more the efficiency of the resulting code, if the ordering of the do-loops is modified as follows:

```
!$OMP DO
  do k = 1, 10
    do j = 1, 10
      do i = 1, 10
        A(i,j,k) = i * j * k
      enddo
    enddo
  enddo
!$OMP END DO
```

This new version handles better the cache memory of each processor, since the fastest changing do-loop acts on consecutive memory locations, leading to a faster code[5]. Of course it is not always possible to modify the ordering of the do-loops: in such a case it is necessary to look for a compromise between do-loop efficiency and parallelization efficiency.

[5]Although the Fortran 90 standard does not specify the way in which arrays have to be stored in memory, it is to expect that the column major form is used, since this was the standard in Fortran 77. Even though, it is advisable to have a look at the documentation of the available compiler in order to know the way in which arrays are stored in memory, since a significant speed up is achieved by cycling in the correct way over arrays.

2.1.2 !$OMP SECTIONS/!$OMP END SECTIONS

This directive-pair allows to assign to each thread a completely different task leading to an **MPMD model** of execution[6]. Each section of code is executed once and only once by a thread in the team. The syntax of the work-sharing construct is the following one:

```
!$OMP SECTIONS clause1 clause2 ...
!$OMP SECTION
  ...
!$OMP SECTION
  ...
...
!$OMP END SECTIONS end_clause
```

Each block of code, to be executed by one of the threads, starts with an `!$OMP SECTION` directive and extends until the same directive is found again or until the closing-directive `!$OMP END SECTIONS` is found. Any number of sections can be defined inside the present directive-pair, but only the existing number of threads is used to distribute the different blocks of code. This means, that if the number of sections is larger than the number of available threads, then some threads will execute more than one section of code in a serial fashion. This may lead to unefficient use of the available resources, if the number of sections is not a multiply of the number of threads. As an example, if five sections are defined in a parallel region with four threads, then three of the threads will be idle waiting for the fourth thread to execute the remaining fifth section.

The opening-directive `!$OMP SECTIONS` accepts the following clauses:

- `PRIVATE(`*list*`)`: see page 37.
- `FIRSTPRIVATE(`*list*`)`: see page 40.
- `LASTPRIVATE(`*list*`)`: see page 41.
- `REDUCTION(`*operator:list*`)`: see page 43.

while the closing-directive `!$OMP END SECTIONS` only accepts the `NOWAIT` clause. The following restrictions apply to the `!$OMP SECTIONS`/`!$OMP END SECTIONS` directive-pair:

- The code enclosed in each section must be a structured block of code: no branching into or out of the block is allowed.
- All the `!$OMP SECTION` directives must be located in the lexical extend of the directive-pair `!$OMP SECTIONS`/`!$OMP END SECTIONS`: they must be in the same routine.

A simple example of the use of the present directive-pair would be:

[6] **MPMD** stands for **Multiple Programs Multiple Data** and refers to the case of having completely different programs/tasks which share or interchange information and which are running simultaneously on different processors.

```
!$OMP SECTIONS
!$OMP SECTION
  write(*,*) "Hello"
!$OMP SECTION
  write(*,*) "Hi"
!$OMP SECTION
  write(*,*) "Bye"
!$OMP END SECTIONS
```

Now each of the messages `Hello`, `Hi` and `Bye` is printed only once on the screen. This example is shown graphically in figure 2.2. The OpenMP specification does not specify the way in which the different tasks are distributed over the team of threads, leaving this point open to the compiler developers.

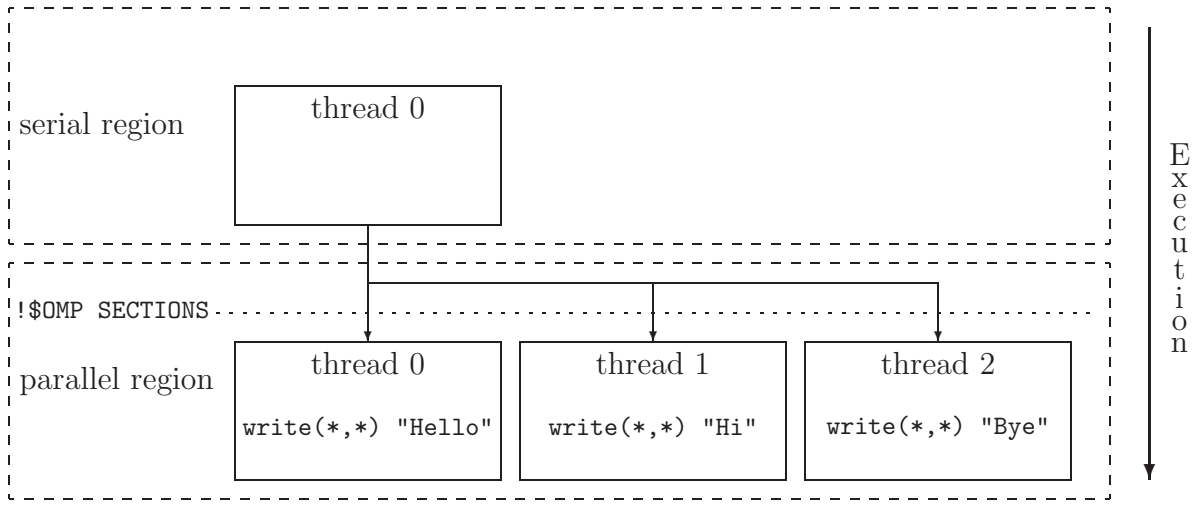

Figure 2.2: Graphical representation of the example explaining the working principle of the `!$OMP SECTIONS`/`!$OMP END SECTIONS` directive-pair.

2.1.3 !$OMP SINGLE/!$OMP END SINGLE

The code enclosed in this directive-pair is only executed by one of the threads in the team, namely the one who first arrives to the opening-directive `!$OMP SINGLE`. All the remaining threads wait at the implied synchronization in the closing-directive `!$OMP END SINGLE`, if the `NOWAIT` clause is not specified. The format of the directive-pair is as follows:

```
!$OMP SINGLE clause1 clause2 ...

...

!$OMP END SINGLE end_clause
```

where the `end_clause` can be the clause `NOWAIT` or the clause `COPYPRIVATE`, but not both at the same time. The functionality of the latter clause is explained on page 43 of chapter 3. Only the following two clauses can be used in the opening-directive:

- PRIVATE(*list*): see page 37.
- FIRSTPRIVATE(*list*): see page 40.

It is necessary that the code placed inside the directive-pair has no branch into or out of it, since this is noncompliant with the OpenMP specification. An example of use of the present directive-pair would be:

```
!$OMP SINGLE
  write(*,*) "Hello"
!$OMP END SINGLE
```

Now the message `Hello` appears only once on the screen. It is necessary to realize that all the other threads not executing the block of code enclosed in the `!$OMP SINGLE`/`!$OMP END SINGLE` directive-pair are idle and waiting at the implied barrier in the closing-directive.

Again, the use of the `NOWAIT` clause solves this problem, but then it is necessary to ensure that the work done by the lonely thread is not required by all the other threads for being able to continue correctly with their work.

The previous example is shown graphically in figure 2.3. To correctly interpret the representation it is necessary to take into account the following rules:

- Solid lines represent work done simultaneously outside of the `SINGLE` region.
- Dotted lines represent the work and idle time associated to the threads not running the block of code enclosed in the `!$OMP SINGLE`/`!$OMP END SINGLE` directive-pair while this block of code is being executed by the lonely thread.
- A non-straight dotted line crossing a region has no work or latency associated to it.
- The traffic-lights represent the existence or not of an implied or explicit synchronization which is acting or not on the corresponding thread.

In the presented example the previous rules lead to the following interpretation: thread 1 reaches the first the `!$OMP SINGLE` opening-directive. While the enclosed code is being executed by thread 1, all the other threads finish their work located before the `!$OMP SINGLE` and jump directly to the closing-directive `!$OMP END SINGLE`, where they wait for thread 1 to finish. Thereafter, all the threads continue together.

2.1.4 !$OMP WORKSHARE/!$OMP END WORKSHARE

Until now parallelizable Fortran 95 commands, like array notation expressions or `forall` and `where` statements, cannot be treated with the presented OpenMP directives in order to distribute their work over a team of threads, since no explicit do-loops are visible. The present work-sharing construct targets precisely these Fortran 95 commands and allows their parallelization. Despite array notation expressions, `forall` statements and `where` statements, also the following Fortran 95 transformational array intrinsic functions can

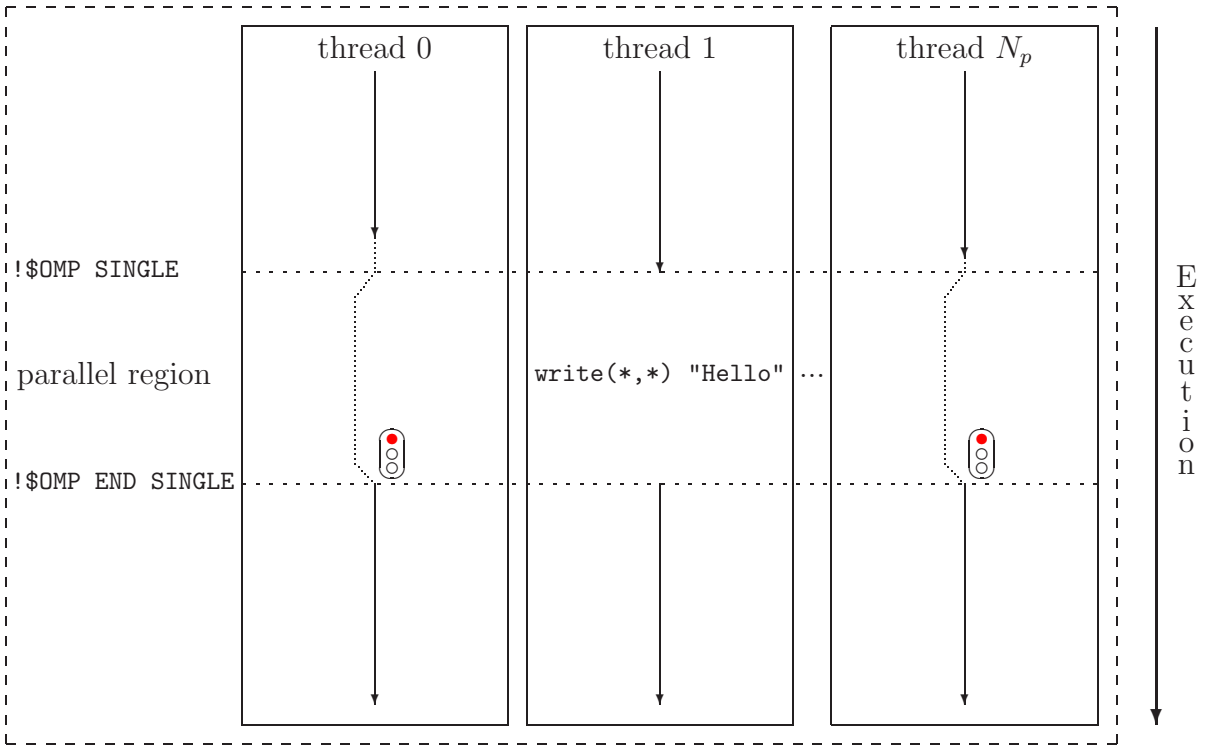

Figure 2.3: Graphical representation of the example explaining the working principle of the `!$OMP SINGLE`/`!$OMP END SINGLE` directive-pair.

be parallelized with the aid of the `!$OMP WORKSHARE`/`!$OMP END WORKSHARE` directive-pair: `matmul`, `dot_product`, `sum`, `product`, `maxval`, `minval`, `count`, `any`, `all`, `spread`, `pack`, `unpack`, `reshape`, `transpose`, `eoshift`, `cshift`, `minloc` and `maxloc`. The syntax to be used is the following one:

```
!$OMP WORKSHARE

...

!$OMP END WORKSHARE end_clause
```

where an implied synchronization exists in the closing-directive `!$OMP END WORKSHARE`, if the `NOWAIT` clause is not specified on the closing-directive.

In contrast to the previously presented work-sharing constructs, the block of code enclosed inside the present directive-pair is executed in such a way, that each statement is completed before the next statement is started. The result is that the block of code behaves exactly in the same way as if it would be executed in serial. For example, the effects of one statement within the enclosed code must appear to occur before the execution of the following statements, and the evaluation of the right hand side of an assignment must appear to have been completed prior to the effects of assigning to the left hand side. This means that the following example

```
real(8) :: A(1000), B(1000)
```

```
!$OMP DO
  do i = 1, 1000
    B(i) = 10 * i
    A(i) = A(i) + B(i)
  enddo
!$OMP END DO
```

which was not working correctly, would work using the present directive-pair, if it is rewritten, for example, as follows:

```
real(8) :: A(1000), B(1000)

!$OMP WORKSHARE
  forall(i=1:1000)
    B(i) = 10 * i
  end forall

  A = A + B
!$OMP END WORKSHARE
```

In order to achieve that the Fortran semantics is respected inside the `!$OMP WORKSHARE`/`!$OMP END WORKSHARE` directive-pair, the OpenMP implementation is allowed to insert as many synchronizations as it needs in order to fulfill the imposed requirement. The result is that an overhead due to these additional synchronizations has to be accepted. The amount of overhead depends on the ability of the OpenMP implementation to correctly interpret and translate the sentences placed inside the directive-pair.

The working principle of the `!$OMP WORKSHARE`/`!$OMP END WORKSHARE` directive-pair relies on dividing its work into separate **units of work** and distributing these units over the different threads. These units of work are created using the following rules specified in the OpenMP specification:

- Array expressions within a statement are splitted such that the evaluation of each element of the array expression is considered as one unit of work.
- Evaluation of transformational array intrinsic functions may be freely subdivided into any number of units of work.
- If the `WORKSHARE` directive is applied to an array assignment statement, the assignment of each element is a unit of work.
- If the `WORKSHARE` directive is applied to a scalar assignment statement, the assignment operation is a unit of work.
- If the `WORKSHARE` directive is applied to an elemental function with an array-type argument, then the application of this function to each of the elements of the array is considered to be a unit of work.
- If the `WORKSHARE` directive is applied to a `where` statement, the evaluation of the mask expression and the masked assignments are workshared.

- If the `WORKSHARE` directive is applied to a `forall` statement, the evaluation of the mask expression, expressions occurring in the specification of the iteration space, and the masked assignments are workshared.
- For `ATOMIC` directives and their corresponding assignments, the update of each scalar variable is a single unit of work.
- For `CRITICAL` constructs, each construct is a single unit of work.
- If none of the rules above applies to a portion of the statement in the block of code, then this portion is treated as a single unit of work.

Therefore, the application of the previous rules delivers a large amount of units of work which are distributed over the different threads; how they are distributed is OpenMP-implementation dependent.

In order to correctly use the `!$OMP WORKSHARE` directive, it is necessary to take into account the following restrictions:

- The code enclosed inside the `!$OMP WORKSHARE`/`!$OMP END WORKSHARE` directive-pair must be a structured block; no branching into or out of the block of code is allowed.
- The enclosed code must only contain array assignment statements, scalar assignment statements, `forall` and `where` statements and `!$OMP ATOMIC` and `!$OMP CRITICAL` directives.
- The enclosed code must not contain any user defined function calls unless the function is `pure`, which means it is free of side effects and has been declared with the keyword `elemental`.
- Variables, which are referenced or modified inside the scope of the `!$OMP WORKSHARE`/`!$OMP END WORKSHARE` directive-pair, must have the `SHARED` attribute; otherwise, the results are unspecified.

The scope of the `!$OMP WORKSHARE` directive is limited to the lexical extent of the directive-pair.

2.2 Combined parallel work-sharing constructs

The **combined parallel work-sharing constructs** are shortcuts for specifying a parallel region that contains only one work-sharing construct. The behavior of these directives is identical to that of explicitly specifying an `!$OMP PARALLEL`/`!$OMP END PARALLEL` directive-pair enclosing a single work-sharing construct.

The reason of existence of these shortcuts is to give the OpenMP-implementation a way of reducing the overhead cost associated to both OpenMP directive-pairs, when they appear together.

2.2.1 !$OMP PARALLEL DO/!$OMP END PARALLEL DO

This directive-pair provides a shortcut form for specifying a parallel region that contains a single !$OMP DO/!$OMP END DO directive-pair. Its syntaxis is as follows:

```
!$OMP PARALLEL DO clause1 clause2 ...
...
!$OMP END PARALLEL DO
```

where the clauses that can be specified are those accepted by either the !$OMP PARALLEL directive or the !$OMP DO directive.

2.2.2 !$OMP PARALLEL SECTIONS/!$OMP END PARALLEL SECTIONS

This directive-pair provides a shortcut form for specifying a parallel region that contains a single !$OMP SECTIONS/!$OMP END SECTIONS directive-pair. The syntaxis to be used is the following one:

```
!$OMP PARALLEL SECTIONS clause1 clause2 ...
...
!$OMP END PARALLEL SECTIONS
```

where the clauses that can be specified are those accepted by either the !$OMP PARALLEL directive or the !$OMP SECTIONS directive.

2.2.3 !$OMP PARALLEL WORKSHARE/!$OMP END PARALLEL WORKSHARE

This directive-pair provides a shortcut form for specifying a parallel region that contains a single !$OMP WORKSHARE/!$OMP END WORKSHARE directive-pair. The syntaxis looks as follows:

```
!$OMP PARALLEL WORKSHARE clause1 clause2 ...
...
!$OMP END PARALLEL WORKSHARE
```

where the clauses that can be specified are those accepted by the !$OMP PARALLEL directive, since the !$OMP WORKSHARE directive does not accept any clause.

2.3 Synchronization constructs

In certain cases it is not possible to leave each thread on its own and it is necessary to bring them back to an order. This is generally achieved through synchronizations of the threads. These synchronizations can be explicit, like the ones introduced in the present section, or implied to previously presented OpenMP directives. In both cases the functionality is the same and it is convenient to read the present section in order to understand the implications derived from the use or not use of the implied synchronizations.

2.3.1 !$OMP MASTER/!$OMP END MASTER

The code enclosed inside this directive-pair is executed only by the master thread of the team. Meanwhile, all the other threads continue with their work: no implied synchronization exists in the `!$OMP END MASTER` closing-directive. The syntaxis of the directive-pair is as follows:

```
!$OMP MASTER

...

!$OMP END MASTER
```

In essence, this directive-pair is similar to using the `!$OMP SINGLE/!$OMP END SINGLE` directive-pair presented before together with the `NOWAIT` clause, only that the thread to execute the block of code is forced to be the master one instead of the first arriving one.

As in previous OpenMP directives, it is necessary that the block of code enclosed inside the directive-pair is structured, which means that no branching into or out of the block is allowed. A simple example of use would be:

```
!$OMP MASTER
  write(*,*) "Hello"
!$OMP END MASTER
```

This example is also shown in a graphical way in figure 2.4, where threads 1 till N_p do not wait for the master thread at the closing-directive `!$OMP END MASTER`; instead, they continue with their execution while the master thread executes the lines enclosed in the directive-pair `!$OMP MASTER/!$OMP END MASTER`. After the closing-directive, the master thread is behind the other threads in its duty.

2.3.2 !$OMP CRITICAL/!$OMP END CRITICAL

This directive-pair restricts the access to the enclosed code to only one thread at a time. In this way it is ensured, that what is done in the enclosed code is done correctly. Examples of application of this directive-pair could be to read an input from the keyboard/file or to update the value of a shared variable. The syntaxis of the directive-pair is the following one:

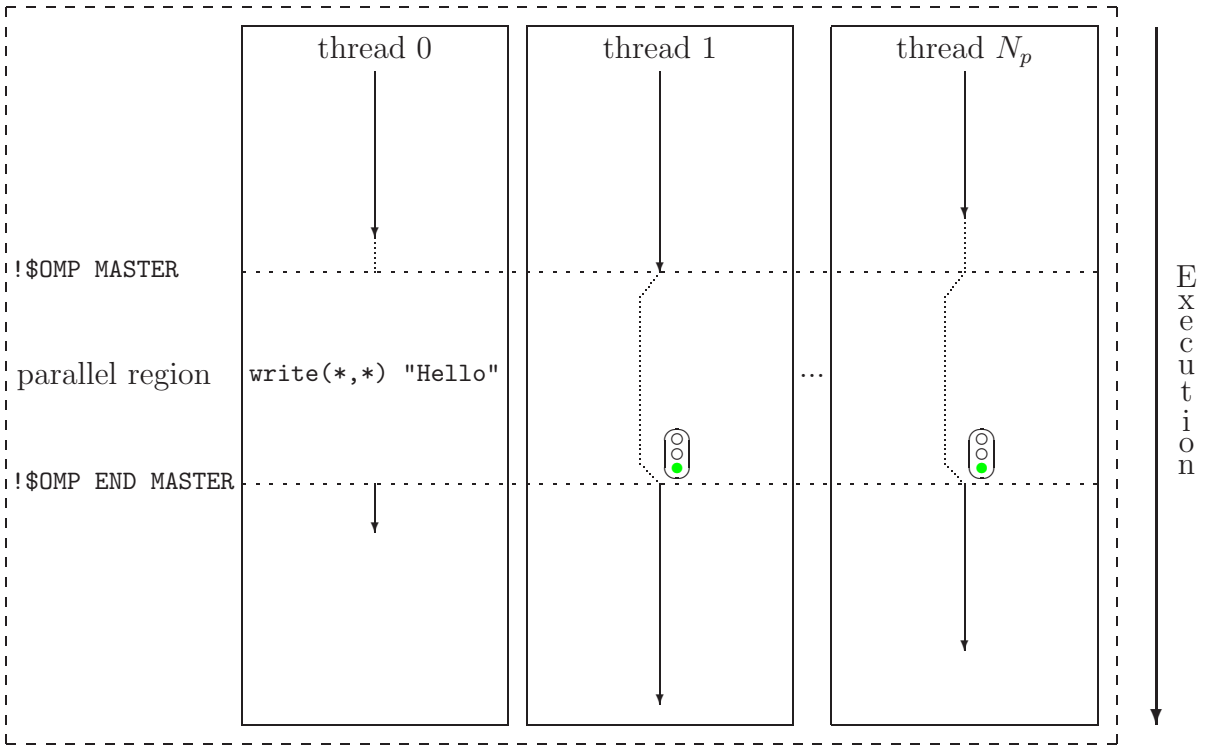

Figure 2.4: Graphical representation of the example explaining the working principle of the `!$OMP MASTER`/`!$OMP END MASTER` directive-pair.

```
!$OMP CRITICAL name

...

!$OMP END CRITICAL name
```

where the optional `name` argument identifies the critical section. Although it is not mandatory, it is strongly recommended to give a name to each critical section.

When a thread reaches the beginning of a critical section, it waits there until no other thread is executing the code in the critical section. Different critical sections using the same name are treated as one common critical section, which means that only one thread at a time is inside them. Moreover, all unnamed critical sections are considered as one common critical section. This is the reason why it is recommended to give names to the critical sections.

A simple example of use of the present directive-pair, which is also represented graphically in figure 2.5, would be:

```
!$OMP CRITICAL write_file
  write(1,*) data
!$OMP END CRITICAL write_file
```

What is shown in figure 2.5 is that thread 0 has to wait until thread N_p has left the `CRITICAL` region. Before thread N_p, thread 1 was inside the `CRITICAL` region while thread N_p was waiting.

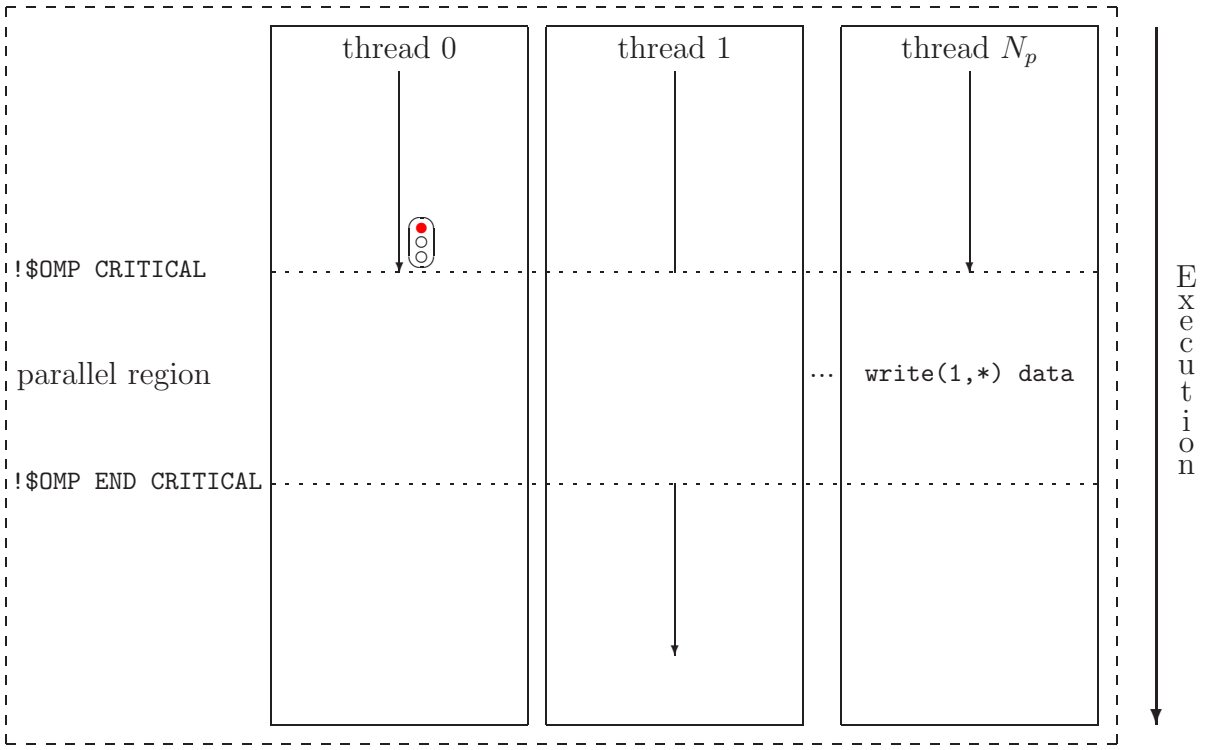

Figure 2.5: Graphical representation of the example explaining the working principle of the `!$OMP CRITICAL`/`!$OMP END CRITICAL` directive-pair.

2.3.3 !$OMP BARRIER

This directive represents an explicit synchronization between the different threads in the team. When encountered, each thread waits until all the other threads have reached this point. Its syntaxis is very simple:

```
!$OMP BARRIER
```

Two restrictions need to be taken into account when using this explicit synchronization:

- The `!$OMP BARRIER` directive must be encountered by all threads in a team or by none at all. Several situations may not fulfill this condition. For example:

  ```
  !$OMP CRITICAL
    !$OMP BARRIER
  !$OMP END CRITICAL
  ```

  Since only one thread at a time is executing the content of the `CRITICAL` region, it is impossible for the other threads to reach the explicit synchronization. The result is that the program reaches a state without exit. This situation is called a **deadlock** and it is necessary to avoid it.

  Other examples, where also a deadlock happens, could be:

```
!$OMP SINGLE
  !$OMP BARRIER
!$OMP END SINGLE


!$OMP MASTER
  !$OMP BARRIER
!$OMP END MASTER


!$OMP SECTIONS
!$OMP SECTION
  !$OMP BARRIER
!$OMP SECTION
  ...
...
!$OMP END SECTIONS
```

These examples are quite obvious, but there are situations in which a deadlock may happen and which are not so clear, for example if the explicit synchronization is placed inside a do-loop or an if-construct. An example of the latter would be:

```
if(my_thread_ID < 5) then
  !$OMP BARRIER
endif
```

In this example, if the total number of threads is smaller than five, then nothing happens. But if the number of threads is larger than five, then there are threads which are not reaching the synchronization, leading to a deadlock. The presented logical statement of the if-construct is simple, but more complex ones, involving results obtained by each thread, are feasible and of possible practical interest.

- The `!$OMP BARRIER` directive must be encountered in the same order by all threads in a team.

The use of this directive should be restricted to cases, where it is really necessary to synchronize all the threads, since it represents a waste of resources due to the idle being of the waiting threads. Therefore, it is convenient to analyze the source code of the program in order to see, if there is any way to avoid the synchronization. This remark also applies to all the other explicit or implied synchronizations in OpenMP.

In figure 2.6 the effect of placing an `!$OMP BARRIER` directive on a team of threads is represented. To see is that thread 1 has to wait at the explicit synchronization until all the other threads reach the same point. Thread 0 will also need to wait at the synchronization, since thread N_p is behind. Finally, once all the threads have reached the explicit synchronization, execution of the following code starts.

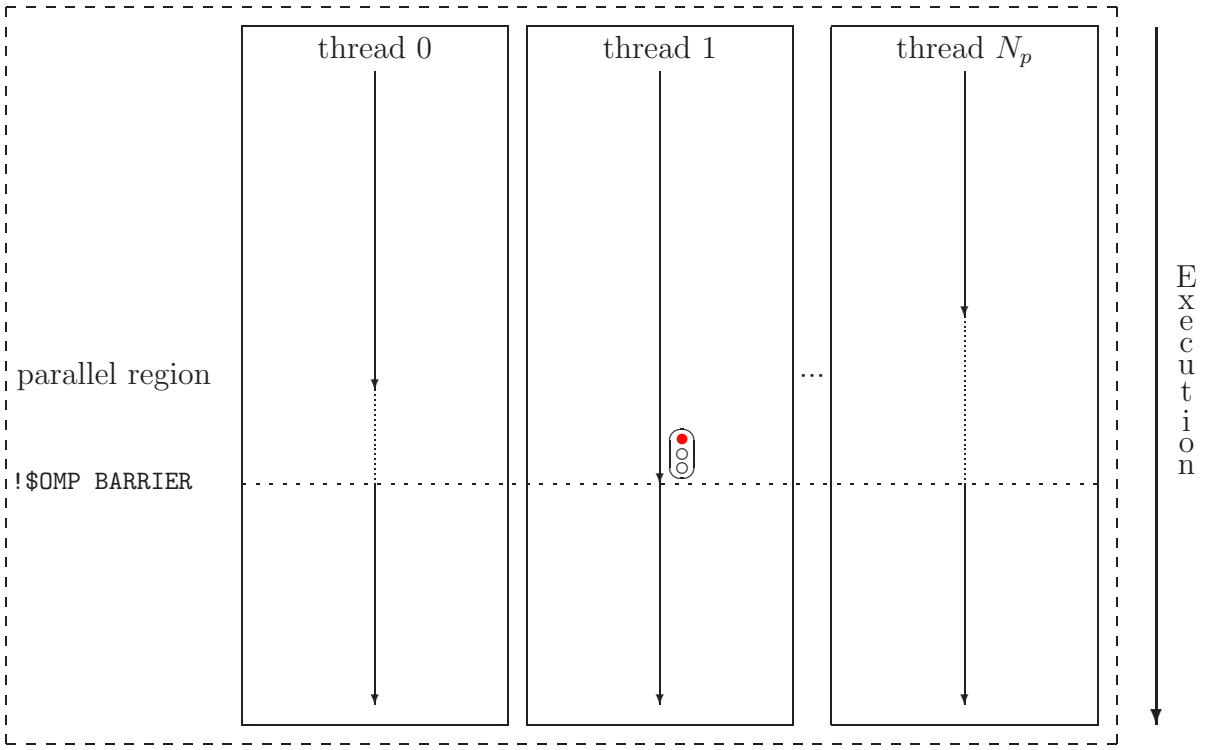

Figure 2.6: Graphical representation of the effect of the `!$OMP BARRIER` directive on a team of threads.

2.3.4 `!$OMP ATOMIC`

When a variable in use can be modified from all threads in a team, it is necessary to ensure that only one thread at a time is writing/updating the memory location of the considered variable, otherwise unpredictable results will occur. In the following example

```
!$OMP DO
  do i = 1, 1000
    a = a + i
  enddo
!$OMP END DO
```

the desired result would be that `a` contains the sum of all values of the do-loop variable `i`, but this will not be the case. The present directive targets to ensure that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneously writing threads. Its syntaxis is very simple:

```
!$OMP ATOMIC
```

This directive only affects the immediately following statement. Not all possible statements are allowed, since most cannot be treated in an atomic way. Only the following ones can be used together with the `!$OMP ATOMIC` directive:

```
x = x operator expr

x = intrinsic_procedure(x, expr_list)
```

where again not all existing `operators` and `intrinsic_procedures` are allowed, only:

```
operator            = +, *, -, /, .AND., .OR., .EQV. or .NEQV.

intrincic_procedure = MAX, MIN, IAND, IOR or IEOR
```

The variable `x`, affected by the `!$OMP ATOMIC` directive, must be of scalar nature and of intrinsic type. Obviously, the expression `expr` must also be a scalar expression.

All these limitations (and a few more appearing in the following explanations) are necessary to ensure, that the statement can be treated in an atomic way and that this is done in an efficient manner. Later on, a few additional words are going to be said about the latter aspect.

Only the load and store of the variable `x` are atomic. This means that the evaluation of `expr` can take place simultaneously in all threads. This feature, which leads to faster executing codes, limits the nature of `expr`, since it cannot make use of the value of `x`!

It is possible to emulate the effect of the `!$OMP ATOMIC` directive using the `!$OMP CRITICAL`/`!$OMP END CRITICAL` directive-pair as follows:

```
xtmp = expr
!$OMP CRITICAL x
  x = x operator xtmp
!$OMP END CRITICAL x
```

The reason for existence of the present directive is to allow optimizations beyond those possible with the `!$OMP CRITICAL`/`!$OMP END CRITICAL` directive-pair. It is left to the OpenMP-implementation to exploit this. Therefore, the gain achieved by using the `!$OMP ATOMIC` directive will be OpenMP-implementation dependent.

2.3.5 !$OMP FLUSH

This directive, whether explicit or implied, identifies a sequence point at which the implementation is required to ensure that each thread in the team has a consistent view of certain variables in memory: the same correct value has to be seen by all threads. This directive must appear at the precise point in the code at which the **data synchronization** is required.

At a first glance it seems that this directive is not necessary, since in general the writing/updating of shared variables has been done in such a way, that only one thread at a time is allowed to do that. But this is only true in theory, because what the OpenMP-implementation does to simulate the *"one thread at a time"* feature is not specified by the OpenMP specification. This is not a fault, moreover it is a door left open to the OpenMP-implementations so that they can try to optimize the resulting code. As an example of this, in the following do-loop

```
!$OMP DO
  do i = 1, 10000
    !$OMP ATOMIC
      A = A + i
  enddo
!$OMP END DO
```

the access to the variable `A` is treated atomically 10000 times, which is not efficient, since only one thread at a time is working for 10000 times. An alternative formulation of the previous do-loop could be:

```
Atmp = 0

!$OMP DO
  do i = 1, 1000
      Atmp = Atmp + i
  enddo
!$OMP END DO

!$OMP ATOMIC
A = A + Atmp
```

where `Atmp` is a temporary variable local to each thread. In this case the variable `A` is accessed atomically only N_p times, where N_p is the number of threads in the team. This new version of the do-loop is much more efficient that the first one.

The same idea, but most probably done in a different way, could be exploited by the compilers to optimize the use of the `!$OMP ATOMIC` directive. But in this optimized version the variable `A` has not the correct value until the end. This may lead to errors, if not taken into account.

The present directive is meant precisely to force the update of shared variables in order to ensure the correct working of the following code. OpenMP-implementations must ensure that their compilers introduce additional code to restore values from registers to memory, for example, or to flush write buffers in certain hardware devices.

Another case, in which the present directive is also of importance, is when different threads are working on different parts of a shared array. At a given point it may be necessary to use information from parts affected by different threads. In such a case it is necessary to ensure, that all the write/update processes have been performed before reading from the array. This consistent view is achieved with the `!$OMP FLUSH` directive.

The format of the directive, which ensures the updating of all variables, is as follows:

```
!$OMP FLUSH
```

Since ensuring the consistent view of variables can be costly due to the transfer of information between different memory locations, it may not be interesting to update all the shared variables at a given point: the `!$OMP FLUSH` directive offers the possibility of including a list with the names of the variables to be flushed:

```
!$OMP FLUSH (variable1, variable2, ...)
```

These expressions represent explicit data synchronizations which can be introduced by the user at specific points in the code. But there are also implied data synchronizations associated to most of the previously presented OpenMP directives:

- `!$OMP BARRIER`
- `!$OMP CRITICAL` and `!$OMP END CRITICAL`
- `!$OMP END DO`
- `!$OMP END SECTIONS`
- `!$OMP END SINGLE`
- `!$OMP END WORKSHARE`
- `!$OMP ORDERED` and `!$OMP END ORDERED`
- `!$OMP PARALLEL` and `!$OMP END PARALLEL`
- `!$OMP PARALLEL DO` and `!$OMP END PARALLEL DO`
- `!$OMP PARALLEL SECTIONS` and `!$OMP END PARALLEL SECTIONS`
- `!$OMP PARALLEL WORKSHARE` and `!$OMP END PARALLEL WORKSHARE`

It should be noted that the data synchronization is not implied by the following directives:

- `!$OMP DO`
- `!$OMP MASTER` and `!$OMP END MASTER`
- `!$OMP SECTIONS`
- `!$OMP SINGLE`
- `!$OMP WORKSHARE`

A very important remark is that implied data synchronizations are suppressed, when the `NOWAIT` clause has been added to the corresponding closing-directive to eliminate the implied thread synchronization!

2.3.6 !$OMP ORDERED/!$OMP END ORDERED

In certain do-loops some of the statements executed at each iteration need to be evaluated in the same order as if the do-loop would be executed sequentially. For example:

```
do i = 1, 100
  A(i) = 2 * A(i-1)
enddo
```

In this do-loop it is necessary to have computed iteration 59 before being able of computing correctly iteration 60, for instance. In such a situation a parallel treatment of the entire do-loop is not possible! The following more general case

```
do i = 1, 100
  block1

  block2

  block3
enddo
```

in which `block2` needs to be evaluated in the correct order while the other two blocks can be executed in any order, is going to be target. One possible solution to this problem could be to split the do-loop into three do-loops and to parallelize only the first and the last one. Another option is to use the present directive-pair to indicate to the compiler the part of the code that needs to be executed sequentially. The result would be:

```
!$OMP DO ORDERED
do i = 1, 100
  block1

  !$OMP ORDERED
    block2
  !$OMP END ORDERED

  block3
enddo
!$OMP END DO
```

where it is mandatory to add the `ORDERED` clause to the `!$OMP DO` opening-directive (see page 52 for further information regarding the `ORDERED` clause). The sequence in which the different blocks of code are executed, for the case of having a team with three threads, is shown graphically in figure 2.7.

The `!$OMP ORDERED`/`!$OMP END ORDERED` directive-pair only makes sense inside the dynamic extent of parallelized do-loops. On one hand the directive-pair allows only one thread at a time inside its scope and on the other hand it allows the entrance of the threads only following the order of the loop iterations: no thread can enter the `ORDERED` section until it is guaranteed that all previous iterations have been completed.

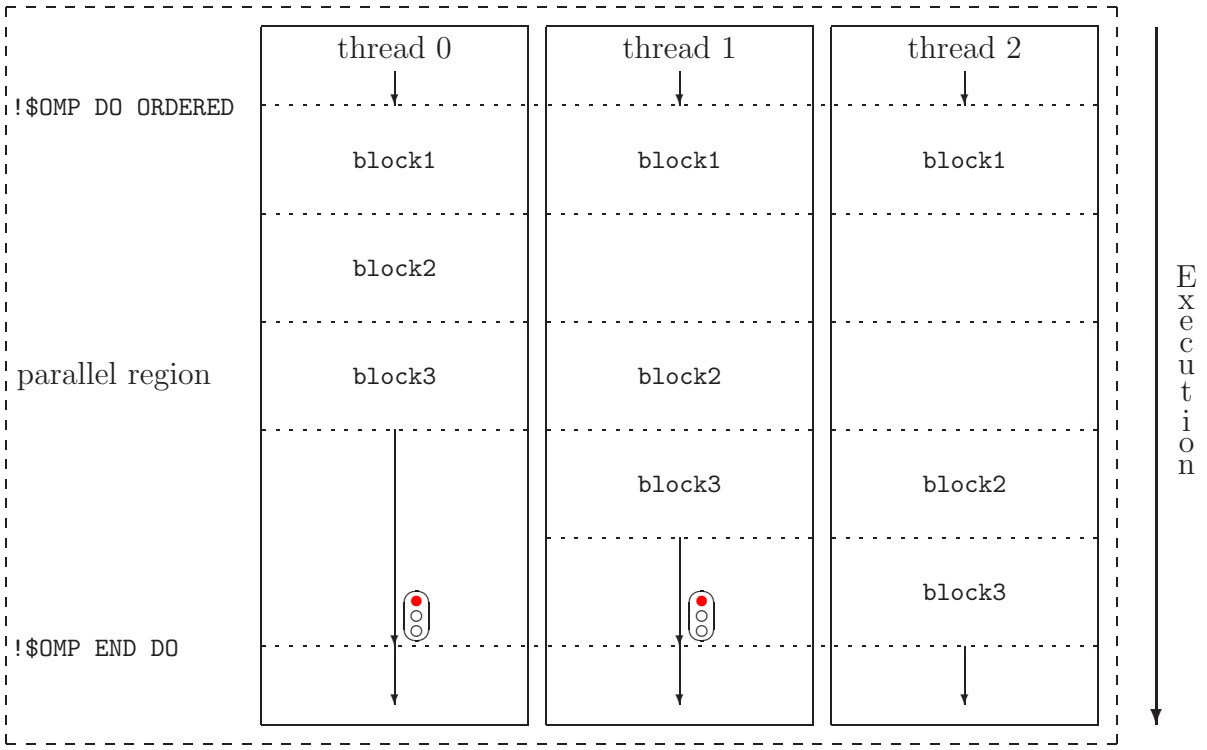

Figure 2.7: Graphical representation of the sequence of execution of the example explaining the working principle of the `!$OMP ORDERED`/`!$OMP END ORDERED` directive-pair.

In essence, this directive-pair is similar to the `!$OMP CRITICAL`/`!$OMP END CRITICAL` directive-pair, but without the implied synchronization in the closing-directive and where the order of entrance is specified by the sequence condition of the loop iterations.

Despite the typical restrictions, like for example the necessity of the enclosed code to be a structured block, it is also necessary to take into account, that only one `ORDERED` section is allowed to be executed by each iteration inside a parallelized do-loop. Therefore, the following example would not be valid:

```
!$OMP DO ORDERED
  do i = 1, 1000
    ...

    !$OMP ORDERED
    ...
    !$OMP END ORDERED

    ...

    !$OMP ORDERED
    ...
    !$OMP END ORDERED

    ...
  enddo
```

```
!$OMP END DO
```

Even though, it is possible to have a do-loop with different `ORDERED` sections, if and only if only one of these sections is seen by each iteration. For example:

```
!$OMP DO ORDERED
  do i = 1, 1000
    ...

    if(i < 10) then
      !$OMP ORDERED
        write(4,*) i
      !$OMP END ORDERED
    else
      !$OMP ORDERED
        write(3,*) i
      !$OMP END ORDERED
    endif

    ...
  enddo
!$OMP END DO
```

2.4 Data environment constructs

The last set of OpenMP directives is meant for controlling the data environment during the execution in parallel. Two different kinds of **data environment constructs** can be found:

- Those which are independent of other OpenMP constructs.
- Those which are associated to an OpenMP construct and which affect only that OpenMP construct and its lexical extend (also known as **data scope attribute clauses**).

The formers are going to be described in the present section, while the latters are explained in chapter 3 due to their extend and importance. Also the remaining clauses, which are not data environment constructs, are going to be presented in chapter 3, leading to a unified view of all OpenMP clauses.

2.4.1 `!$OMP THREADPRIVATE (list)`

Sometimes it is of interest to have global variables, but with values which are specific for each thread. An example could be a variable called `my_ID` which stores the thread identification number of each thread: this number will be different for each thread, but it would be useful that its value is accessible from everywhere inside each thread and that its value does not change from one parallel region to the next.

The present OpenMP directive is meant for defining such variables by assigning the THREADPRIVATE attribute to the variables included in the associated *list*. An example would be:

```
!$OMP THREADPRIVATE(a, b)
```

which means that a and b will be local to each thread, but global inside it. The variables to be included in *list* must be common blocks[7] or named variables. The named variables, if not declared in the scope of a Fortran 95 module, must have the save attribute set.

The !$OMP THREADPRIVATE directive needs to be placed just after the declarations of the variables and before the main part of the software unit, like in the following example:

```
real(8), save :: A(100), B
integer, save :: C

!$OMP THREADPRIVATE(A, B, C)
```

When the program enters the first parallel region, a private copy of each variable marked as THREADPRIVATE is created for each thread. Initially, these copies have an undefined value, unless a COPYIN clause has been specified at the opening-directive of the first parallel region (further information on the COPYIN clause can be found on page 42).

On entry to a subsequent parallel region, if the dynamic thread adjustment mechanism has been disabled, the status and value of all THREADPRIVATE variables will be the same as at the end of the previous parallel region, unless a COPYIN clause is added to the opening-directive.

An example showing an application of the present OpenMP directive would be:

```
integer, save :: a
!$OMP THREADPRIVATE(a)

!$OMP PARALLEL
  a = OMP_get_thread_num()
!$OMP END PARALLEL

!$OMP PARALLEL
  ...
!$OMP END PARALLEL
```

In this example the variable a gets assigned the thread identification number of each thread during the first parallel region[8]. In the second parallel region, the variable a keeps the values assigned to it in the first parallel region, since it is THREADPRIVATE. This example is shown graphically in figure 2.8, where the dotted lines represent the effect of the THREADPRIVATE attribute.

[7] Although common blocks are still part of the Fortran 95 standard, they have been marked as **obsolescent** and therefore should no longer be used when developing new programs. An improved functionality can be achieved using Fortran 95 modules instead. This is the reason why no further reference to the use of common blocks will be given in the present explanation of the !$OMP THREADPRIVATE directive.

[8] The run-time routine OMP_get_thread_num, explained later on, returns the ID number of each thread inside the team.

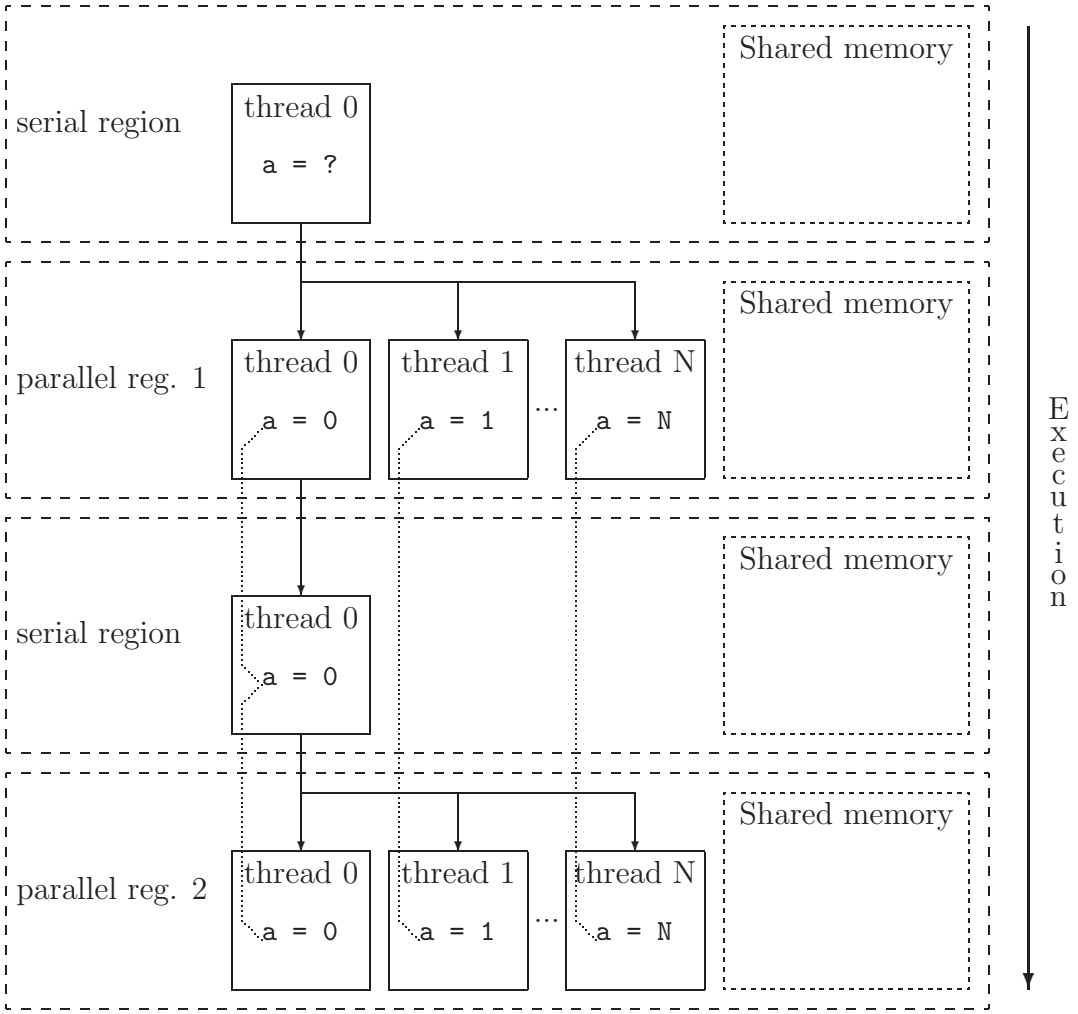

Figure 2.8: Graphical representation of the example explaining the working principle of the `!$OMP THREADPRIVATE` clause.

Any variable which has the `THREADPRIVATE` attribute set can only appear in the clauses `COPYIN` and `COPYPRIVATE`.

Chapter 3

PRIVATE, SHARED & Co.

In many of the previously presented OpenMP directives it is possible to modify their way of working by adding so called **clauses**. Two different kinds of clauses can be identified. The first kind, called **data scope attribute clauses**, specify how each variable is handled and who is allowed to see its value and to change it. The second kind includes all the clauses which do not fit into the first kind.

Not all of the following data scope attribute clauses are allowed by all directives, but the clauses that are valid on a particular directive are indicated in the previous descriptions of the directives.

Although most of the examples presented in the following descriptions make use of the directive-pair `!$OMP PARALLEL`/`!$OMP END PARALLEL`, this does not mean that their use is only restricted to that directive-pair, but most of the concepts are easier to understand with it than with other directive-pairs.

3.1 Data scope attribute clauses

3.1.1 PRIVATE(*list*)

Sometimes certain variables are going to have different values in each thread. This is only possible if and only if each thread has its own copy of the variable. This clause fixes which variables are going to be considered as **local variables** to each thread. For example

```
!$OMP PARALLEL PRIVATE(a, b)
```

says, that the variables `a` and `b` will have different values in each thread: they will be local to each thread.

When a variable is said to be private, a new object of the same type is declared once for each thread in the team and used by each thread inside the scope of the directive-pair (in the previous example inside the parallel region) instead of the original variable. This idea is shown in figure 3.1 in a graphical way.

Variables declared as private have an undefined value at the beginning of the scope of the directive-pair, since they have just been created. Also when the scope of the

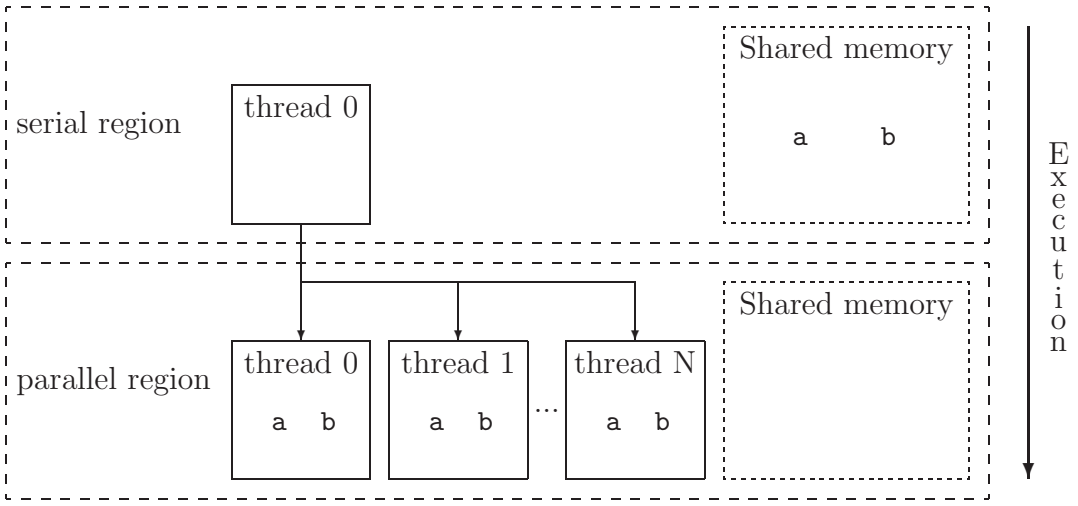

Figure 3.1: Graphical representation of the effect of the `PRIVATE` clause on the variables `a` and `b` of the presented example.

directive-pair finishes, the original variable will have an undefined value (which value from the available ones should it have!?).

The fact, that a new object is created for each thread can be a very resource consuming thing: for example, if a 5Gb array (a common size for direct numerical simulations or complex chemically reacting flow applications) is specified to be private in a parallel region with 10 threads, then the total memory requirement will be of 55Gb[1], an amount of storage not available on all SMP machines!

Variables that are used as counters for do-loops, forall commands or implicit do-loops or are specified to be `THREADPRIVATE` become automatically private to each thread, even though they are not explicitly included inside a `PRIVATE` clause at the beginning of the scope of the directive-pair.

3.1.2 SHARED(*list*)

In contrast to the previous situation, sometimes there are variables which should be available to all threads inside the scope of a directive-pair, because their values are needed by all threads or because all threads have to update their values. For example

```
!$OMP PARALLEL SHARED(c, d)
```

says that the variables `c` and `d` are seen by all threads inside the scope of the `!$OMP PARALLEL`/`!$OMP END PARALLEL` directive-pair. This example is shown in figure 3.2 in a graphical way.

When a variable is said to be shared, nothing happens to it: no new memory is reserved and its value before the starting-directive is conserved. This means that all

[1]Although it is allowed that one thread reuses the original memory location of the original variable, this is something not required by the OpenMP standard and it is left open to the developers of each OpenMP implementation to do it or not.

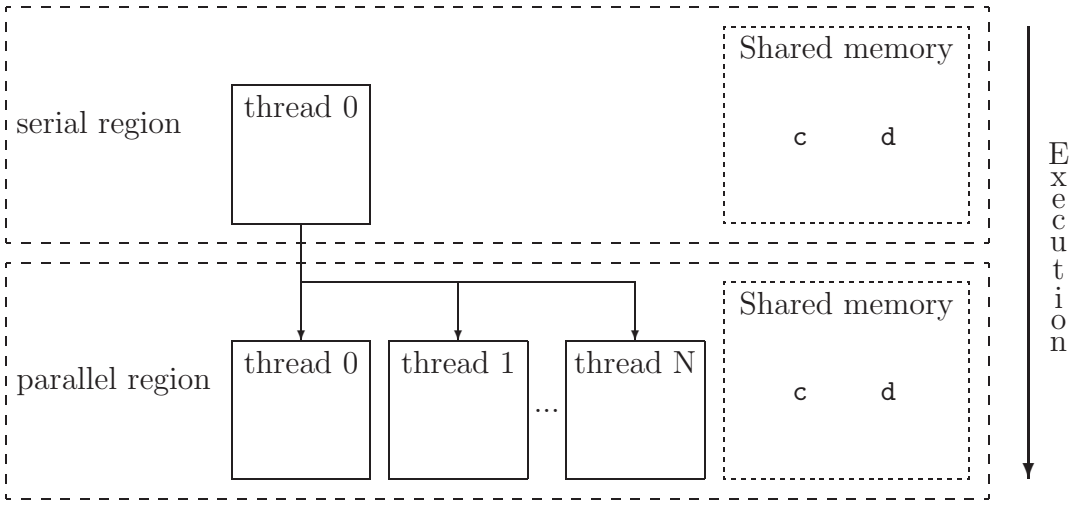

Figure 3.2: Graphical representation of the effect of the `SHARED` clause on the variables `c` and `d` of the presented example.

threads access to the same memory location for reading from/writing to the variable. Therefore, declaring a variable as `SHARED` does not consume any additional resources.

That each thread in the team access the same memory location for a shared variable does not guarantee that the threads are immediately aware of changes made to the variable by another thread; an OpenMP implementation may store the new values of shared variables in temporary variables and perform the update later on. It is possible to force the update of the shared variables by using the directive `!$OMP FLUSH`.

Further more, since more than one thread can write exactly in the same memory location at the same time, the result of the multiple write processes is clearly undefined. This so called **racing condition** has always to be avoided and it is left to the programmer to do that. One possible solution is to force the writing process to be atomically, for example using the `!$OMP ATOMIC` directive.

3.1.3 DEFAULT(PRIVATE | SHARED | NONE)

When most of the variables used inside the scope of a directive-pair are going to be private/shared, then it would be cumbersome to include all of them in one of the previous clauses. To avoid this, it is possible to specify what OpenMP has to do, when nothing is said about a specific variable: it is possible to specify a default setting. For example

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(a)
```

says that all variables excluding `a` are going to be private, while `a` is going to be shared by all threads inside the scope of the parallel region. If no `DEFAULT` clause is specified, the default behavior is the same as if `DEFAULT(SHARED)` were specified. Again, this example is shown in figure 3.3.

Additionally to the two options `PRIVATE` and `SHARED` there is a third one: `NONE`. Specifying `DEFAULT(NONE)` requires that each variable used inside the scope of the directive has

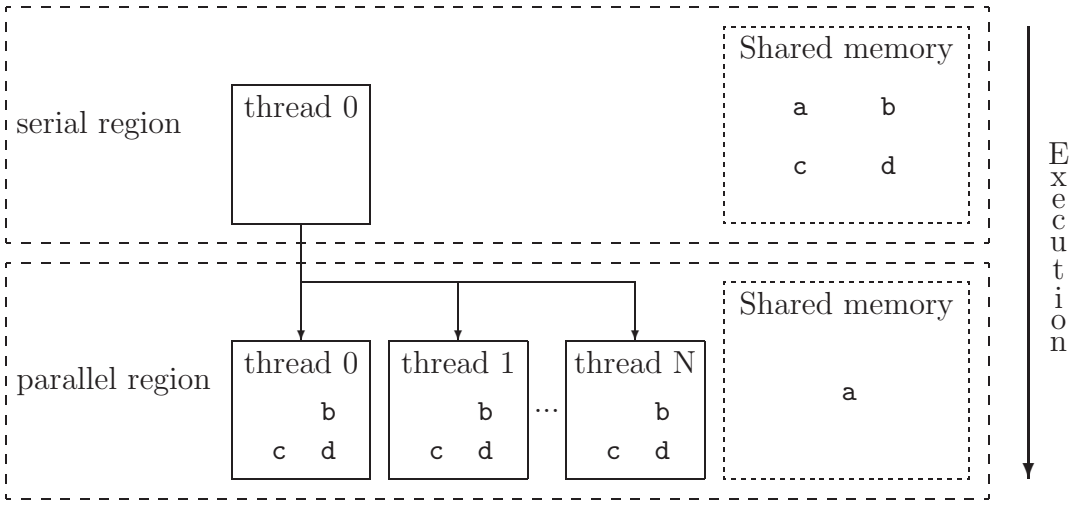

Figure 3.3: Graphical representation of the effect of the DEFAULT clause on the variables of the presented example.

to be explicitly listed in a data scope attribute clause at the beginning of the scope of the directive-pair, unless the variable is THREADPRIVATE or is the counter for a do-loop, forall command or implicit do-loop.

The effect of the DEFAULT clause is limited to the lexical extend of the directive-pair. This means that variables declared in procedures called from the lexical extend are not affected and will have the PRIVATE attribute.

3.1.4 FIRSTPRIVATE(*list*)

As already mentioned before, private variables have an undefined value at the beginning of the scope of a directive-pair. But sometimes it is of interest that these local variables have the value of the original variable before the starting-directive. This is achieved by including the variable in a FIRSTPRIVATE clause as follows:

```
a = 2
b = 1

!$OMP PARALLEL PRIVATE(a) FIRSTPRIVATE(b)
```

In this example, variable a has an undefined value at the beginning of the parallel region, while b has the value specified in the preceding serial region, namely b = 1. This example is represented in figure 3.4.

When a variable is included in a FIRSTPRIVATE clause at the beginning of the scope of a directive-pair, it automatically gets the PRIVATE status for that directive-pair and does not have to be included again in an explicit PRIVATE clause.

Since the information stored in the original variable needs to be transferred to the new private copies of it, to declare a variable as FIRSTPRIVATE can be a very costly operation from the computational time point of view. Considering again the case of the 5Gb array

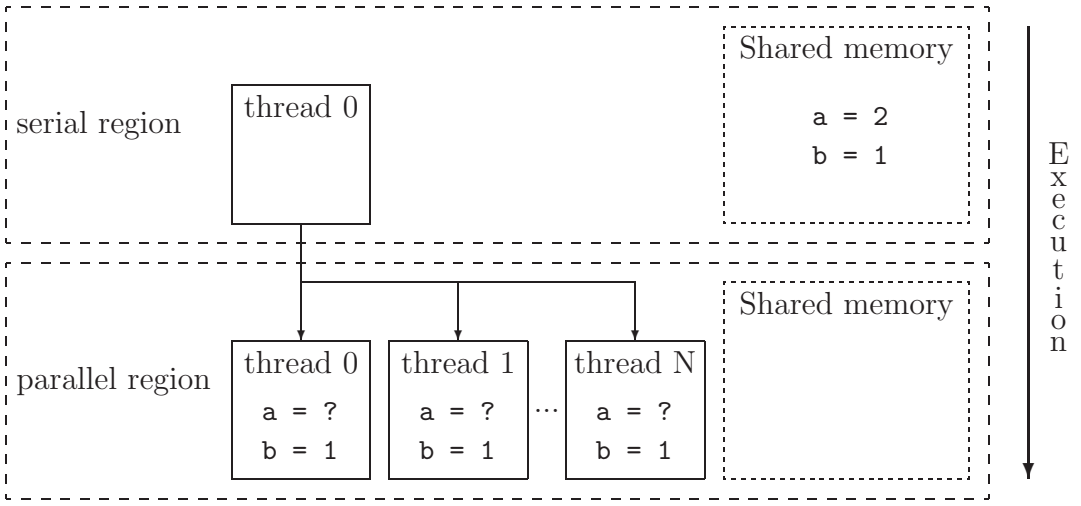

Figure 3.4: Graphical representation of the example given in the description of the `FIRSTPRIVATE` clause.

and the parallel region with 10 threads, if the array would be declared as `FIRSTPRIVATE`, then 50Gb of information would need to be moved from one memory location to another, which is a very costly operation.

3.1.5 LASTPRIVATE(*list*)

A variable declared as private at the beginning of the scope of a directive-pair will have an undefined value at the end of it. This is sometimes not convenient. By including a variable in a `LASTPRIVATE` clause, the original variable will be updated by the *"last"* value it gets inside the scope of the directive-pair, if this directive-pair would be executed in serial mode. For example:

```
!$OMP DO PRIVATE(i) LASTPRIVATE(a)
  do i = 1, 1000
    a = i
  enddo
!$OMP END DO
```

After the finishing of the scope of the `!$OMP DO`/`!$OMP END DO` directive-pair, the variable `a` will be equal to 1000, which is the value it would have, if the OpenMP directive would not exist. Another example would be using the `!$OMP SECTIONS` directive:

```
!$OMP SECTIONS LASTPRIVATE(a)
!$OMP SECTION
  a = 1
!$OMP SECTION
  a = 2
!$OMP END SECTIONS
```

In this case the thread that executes the lexically last SECTION updates the original variable a, leading to a = 2.

If the last iteration in the do-loop, or the lexically last SECTION, is not setting a value to the variable specified in the LASTPRIVATE clause, then the variable will have an undefined value after the scope of the directive-pair, like in the following example:

```
!$OMP SECTIONS LASTPRIVATE(a)
!$OMP SECTION
  a = 1
!$OMP SECTION
  a = 2
!$OMP SECTION
  b = 3
!$OMP END SECTIONS
```

The process of updating the original variable by the *"last"* value assigned inside the scope of the directive-pair takes place at the synchronization between the different threads implied in the closing-directive. Therefore, if this implied synchronization is non-existent (because of the nature of the closing-directive or because it has been forced by a NOWAIT clause), then the value of the variable will be undefined until an implied or explicit synchronization is found. In this way it is ensured that the thread executing the sequentially last iteration has computed the final value of that variable.

Since information is transferred from one memory location to another, to include a variable inside a LASTPRIVATE clause can be very time consuming, if the size of the variable is large. In contrast to the previous FIRSTPRIVATE clause, in the present case only an amount of information equal to the size of the variable is moved, therefore not depending on the number of threads existing in the parallel region.

3.1.6 COPYIN(*list*)

When a variable has been declared as THREADPRIVATE, its value in each thread can be set equal to the value in the master thread by using the present clause. The following example shows its use:

```
!$OMP THREADPRIVATE(a)

!$OMP PARALLEL
  a = OMP_get_thread_num()
!$OMP END PARALLEL

!$OMP PARALLEL
  ...
!$OMP END PARALLEL

!$OMP PARALLEL COPYIN(a)
  ...
!$OMP END PARALLEL
```

In this example the variable a gets assigned the thread identification number of each thread during the first parallel region[2]. In the second parallel region, the variable a keeps the values assigned to it in the first parallel region, since it is THREADPRIVATE. But in the third parallel region, its COPYIN clause sets the value of a to zero in all threads, which is the value of a in the master thread. In figure 3.5 this example is shown graphically.

The idea behind this clause is similar to FIRSTPRIVATE with the difference that it applies to THREADPRIVATE variables instead to PRIVATE variables.

The computational cost associated to the use of the COPYIN clause will be similar as for the use of the FIRSTPRIVATE clause, since the information stored in a variable needs to be transferred to all its private copies in the threads.

3.1.7 COPYPRIVATE(*list*)

After a single thread inside a parallel region has executed a set of instructions enclosed inside an !$OMP SINGLE/!$OMP END SINGLE directive-pair, it is possible to broadcast the value of a private variable to the other threads in the team. For example in

```
!$OMP SINGLE
  read(1,*) a
!$OMP END SINGLE COPYPRIVATE(a)
```

one thread reads the value of a from a file and, thereafter, updates the private copies of the same variable in all the other threads by using the COPYPRIVATE clause.

The effect of the COPYPRIVATE clause on the variables in its list occurs after the execution of the code enclosed within the !$OMP SINGLE/!$OMP END SINGLE directive-pair, and before any thread in the team has left the implied barrier in the closing-directive.

Since it is not allowed by the OpenMP specification to use a COPYPRIVATE and a NOWAIT clause simultaneously in the same !$OMP END SINGLE closing-directive, the update of the private variables in the list of COPYPRIVATE is always ensured after the closing-directive.

The presented COPYPRIVATE clause can only be used together with the !$OMP END SINGLE closing-directive.

3.1.8 REDUCTION(*operator:list*)

When a variable has been declared as SHARED because all threads need to modify its value, it is necessary to ensure that only one thread at a time is writing/updating the memory location of the considered variable, otherwise unpredictable results will occur. In the following example

```
!$OMP DO
  do i = 1, 1000
    a = a + i
  enddo
!$OMP END DO
```

[2]The run-time routine OMP_get_thread_num, explained later on, returns the ID number of each thread inside the team.

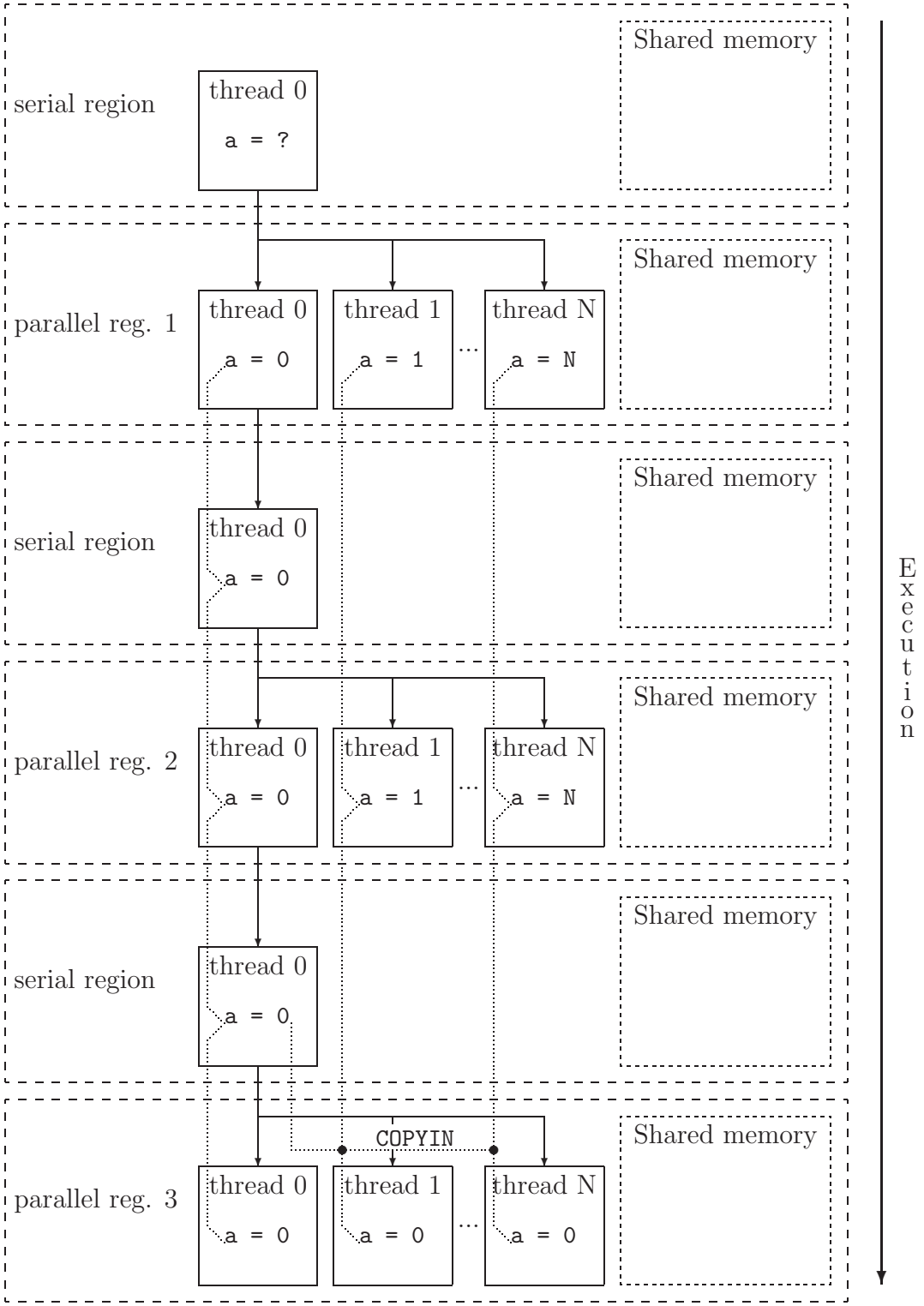

Figure 3.5: Graphical representation of the example given in the description of the `COPYIN` clause.

the desired result would be that `a` contains the sum of all values of the do-loop variable `i`, but this will not be the case. By using the clause `REDUCTION` it is possible to solve this problem, since only one thread at a time is allowed to update the value of `a`, ensuring that the final result will be the correct one. In some sense the `REDUCTION` clause has the same aim as the `!$OMP ATOMIC` and `!$OMP CRITICAL` directives, but done in a different way.

The previous example would then look as follows:

```
!$OMP DO REDUCTION(+:a)
  do i = 1, 1000
    a = a + i
  enddo
!$OMP END DO
```

The syntax of the `REDUCTION` clause associates an operator to a list of variables so that each time, inside the scope of the directive-pair, the operator appears together with one of the variables of its associated list, the previously mentioned mechanism of protecting the writing/updating process is used.

This mechanism of protecting the writing/updating process is defined as follows: a private copy of each variable in *`list`* is created for each thread as if the `PRIVATE` clause had been used; the resulting private copies are initialized following the rules shown in table 3.1; at the end of the `REDUCTION`, the shared variable is updated to reflect the result of combining the initialized value of the shared reduction variable with the final value of each of the private copies using the specified operator.

This approach of defining the protecting mechanism minimizes the number of times in which the writing/updating process of the original variable is *"really"* protected, since it will be only equal to the number of threads inside the parallel region and will not depend on the specific computations performed inside the parallel region. On the other hand, an overhead in computational time and resources will be generated due to the need of creating private copies of the variables and initialize them following the rules given in table 3.1.

Since the original shared variable is only updated at the end of the `REDUCTION` process, its value remains undefined until that point: the end of the `REDUCTION` process is linked to the synchronization step between the different threads implied normally in the closing-directive. If this implied synchronization is non-existent, then the value of the `REDUCTION` variable will be undefined until an implied or explicit synchronization is found.

If a variable is included inside a `REDUCTION` clause, it is not possible for the OpenMP implementation to guarantee a bit-identical result from one parallel run to another, since the intermediate values of the `REDUCTION` variables may be combined in random order. The `REDUCTION` clause is intended to be used only for statements involving the operators and variables in one of the following ways:

```
x = x operator expr

x = intrinsic_procedure(x, expr_list)
```

where only the following `operators` and `intrinsic_procedures` are allowed:

Operator/Intrinsic	Initialization
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	smallest representable number
MIN	largest representable number
IAND	all bits on
IOR	0
IEOR	0

Table 3.1: Initialization rules for variables included in `REDUCTION` clauses.

```
operator            = +, *, -, .AND., .OR., .EQV. or .NEQV.

intrincic_procedure = MAX, MIN, IAND, IOR or IEOR
```

The variable `x`, affected by the `REDUCTION` clause, must be of scalar nature and of intrinsic type. Even though, `x` may be the scalar entry in an array which is not of deferred shape or assumed size. If this is the case, the computational overhead due to the creation and initialization of the private copies of the variables can be very large if `x` is a large array.

Obviously, the expression `expr` must also be a scalar expression that does not make use of the value of `x`, since this magnitude is only known at the end of the `REDUCTION` process. Also, although it seems that the `intrinsic_procedures` do not fulfill this condition, this is not true, since only procedures of associative nature are allowed in the `REDUCTION` clause.

In an opening-directive it is possible to specify any number of `REDUCTION` clauses, but a variable can appear only once in these `REDUCTION` clauses.

3.2 Other clauses

Additionally to the previously presented data scope attribute clauses, a few more clauses are available, which are presented in the following subsections.

3.2.1 IF(*scalar_logical_expression*)

Maybe it is not always interesting to run certain source code lines in parallel, for example if the computational cost of creating and closing the parallel region exceeds the possible gain achieved by running the source code lines in parallel. In the following example

```
!$OMP PARALLEL IF(N > 1000)
!$OMP DO
```

```
  do i = 1, N
    ...
  enddo
!$OMP END DO
!$OMP END PARALLEL
```

the do-loop is executed in parallel only, if the number of iterations is greater than 1000, otherwise the do-loop is executed in a serial fashion.

3.2.2 NUM_THREADS(*scalar_integer_expression*)

For certain parallel running source code lines it could be desirable to use a fixed number of threads, for example, if a certain set of tasks has been defined using an `!$OMP SECTIONS` directive. The use of the clause would look as follows:

```
!$OMP PARALLEL NUM_THREADS(4)
```

where four threads are requested for the specific parallel region.

When a `NUM_THREADS` clause is used at the beginning of a parallel region, it overwrites any settings imposed by the environmental variable `OMP_NUM_THREADS` or by a call to the run-time library subroutine `OMP_set_num_threads`[3]. The specified number of threads only affects the present parallel region and not the future ones.

3.2.3 NOWAIT

Under certain circumstances it is not necessary that all threads finish a given parallel executed task at the same time, because the following operations are not depending on what has been done before. In such cases it is possible to avoid the implied synchronization present in many OpenMP directives by adding the `NOWAIT` clause to their closing-directive. In the following example

```
!$OMP PARALLEL
!$OMP DO
  do i = 1, 1000
    a = i
  enddo
!$OMP END DO NOWAIT
!$OMP DO
  do i = 1, 1000
    b = i
  enddo
!$OMP END DO
!$OMP END PARALLEL
```

[3]The run-time routine `OMP_set_num_threads`, explained later on, allows to specify the number of threads to be used in the following parallel regions.

it is not necessary to wait at the end of the first parallel do-loop for all the threads to finish, since the second do-loop has nothing in common with the first one. Therefore, a `NOWAIT` clause is added at the end of the first `!$OMP END DO` closing-directive. In this way it is possible to speed up the computations, since a costly synchronization has been eliminated.

But it is necessary to use the `NOWAIT` clause with care, since a wrong use of it can leed to unexpected results without previous notice during the development phase of the program. The programmer should also be aware of the fact that using the `NOWAIT` clause also suppresses the implied data synchronizations.

3.2.4 SCHEDULE(*type*, chunk)

When a do-loop is parallelized and its iterations distributed over the different threads, the most simple way of doing this is by giving to each thread the same number of iterations. But this is not always the best choice, since the computational cost of the iterations may not be equal for all of them. Therefore, different ways of distributing the iterations exist. The present clause is meant to allow the programmer to specify the **scheduling** for each do-loop using the following syntaxis:

```
!$OMP DO SCHEDULE(type, chunk)
```

The `SCHEDULE` clause accepts two parameters. The first one, *type*, specifies the way in which the work is distributed over the threads. The second one, *chunk*, is an optional parameter specifying the size of the work given to each thread: its precise meaning depends on the type of scheduling used.

Four different options of scheduling exist, which are:

`STATIC` :when this option is specified, the pieces of work created from the iteration space of the do-loop are distributed over the threads in the team following the order of their thread identification number. This assignment of work is done at the beginning of the do-loop and stays fixed during its execution.

By default, the number of pieces of work is equal to the number of threads in the team and all pieces are approximately equal in size. The junction of all these pieces is equal to the complete iteration space of the do-loop.

If the optional parameter *chunk* is specified, the size of the pieces is fixed to that amount. In order to correctly distribute the total work from the iteration space, one piece of work is allowed to have a different size than *chunk*. The resulting pieces of work are distributed to the threads in a round-robin fashion.

To clarify the ideas, the following example with three threads is considered:

```
!$OMP DO SCHEDULE(STATIC, chunk)
do i = 1, 600
  ...
enddo
!$OMP END DO
```

The 600 iterations are going to be distributed in different ways by changing the value of *chunk*. The results are shown graphically in figure 3.6, where each block represents a piece of work and the number inside it the thread identification number processing the piece of work.

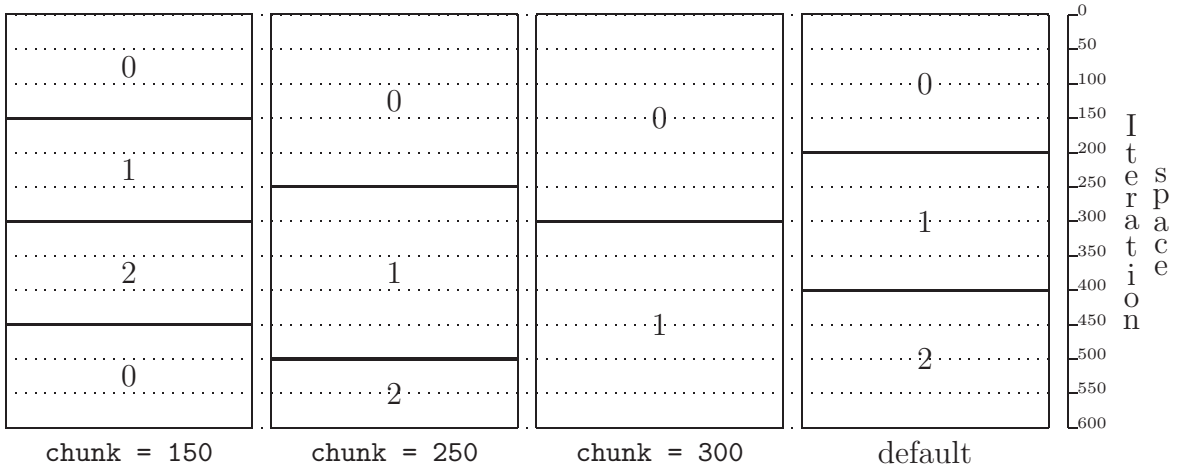

Figure 3.6: Graphical representation of the example explaining the general working principle of the `SCHEDULE(STATIC,chunk)` clause for different values of the optional parameter *chunk*.

In the first case, *chunk = 150*, four blocks of work are needed to cover the complete iteration space. Since the total number of iterations is exactly four times the value of *chunk*, all blocks are exactly equal in size. To distribute the different blocks among the existing threads, a round-robin procedure is used leading to the presented distribution. The resulting parallel running code is not very efficient, since thread 0 is going to work while all the other threads are waiting in an idle state.

In the second case, *chunk = 250*, the number of blocks equals the number of threads, which is something good. But this time the sizes of the blocks are not equal, since the total number of iterations is not an exact multiply of *chunk*. Even though, this value of the optional parameter leads to a faster code than in the previous case, because the latency times are smaller[4].

In the third case, *chunk = 300*, the number of blocks is smaller than the number of available threads. This means that thread 2 is not going to work for the present do-loop. The resulting efficiency is worse than in the previous case and equal to the first case.

In the last case, which corresponds to do not specify the optional parameter *chunk*, OpenMP creates a number of blocks equal to the number of available threads and with equal block sizes. In general this option leads to the best performance, if all iterations require the same computational time. This remark also applies to the previous cases and explanations.

[4]In the first case the fact, that thread 0 needs to compute in total 300 iterations, leads to a slower code than in the second case, where the maximum number of iterations to compute by one thread is 250.

`DYNAMIC` : in the previous scheduling method the problem of non homogeneous iterations has been pointed out. In such a situation it turns out to be difficult to forecast the optimal distribution of the iterations among the threads. A solution would be to assign the different blocks of work in a dynamic way: as one thread finishes its piece of work, it gets a new one. This idea is precisely what lies behind the present scheduling method.

When `SCHEDULE(DYNAMIC,`*`chunk`*`)` is specified, the iteration space is divided into pieces of work with a size equal to *`chunk`*. If this optional parameter is not given, then a size equal to one iteration is considered. Thereafter, each thread gets one of these pieces of work. When they have finished with their task, they get assigned a new one until no pieces of work are left.

The best way to understand the working principle of this scheduling method is by means of an example. In the top part of figure 3.7 12 different pieces of work are shown. The length of each of these blocks represents the amount of computational time required to compute them. These pieces of work are going to be distributed over a team of three threads. The resulting execution schema is shown in the bottom part of figure 3.7. Clearly to see is that each thread gets a new piece or work when it finishes the previous one.

`GUIDED` : the previous dynamic method enhances the capabilities and performance with respect to the static method, but it adds an overhead in computational cost due to the handling and distributing of the different iteration packages. This overhead is reduced as the size of the pieces of work increases, but then a larger non-equilibrium between the different threads is to expect.

Another dynamic approach to this problem could be to have pieces of work with decreasing sizes, so that their associated work is smaller and smaller as they are assigned to the different threads. The decreasing law is of exponential nature so that the following pieces of work have half the number of iterations as the previous ones. This is what the present scheduling method does.

The optional parameter *`chunk`* specifies the smallest number of iterations grouped into one piece of work. As before, the last piece of work may have a smaller number of iterations than specified in *`chunk`* in order to cover completely the iteration space. The number of pieces of work with equal size and the largest size in use are OpenMP-implementation dependent.

In figure 3.8 an example of the resulting pieces of work is shown. The associated do-loop has the following form:

```
!$OMP DO SCHEDULE(GUIDED,256)
  do i = 1, 10230
    ...
  enddo
!$OMP END DO
```

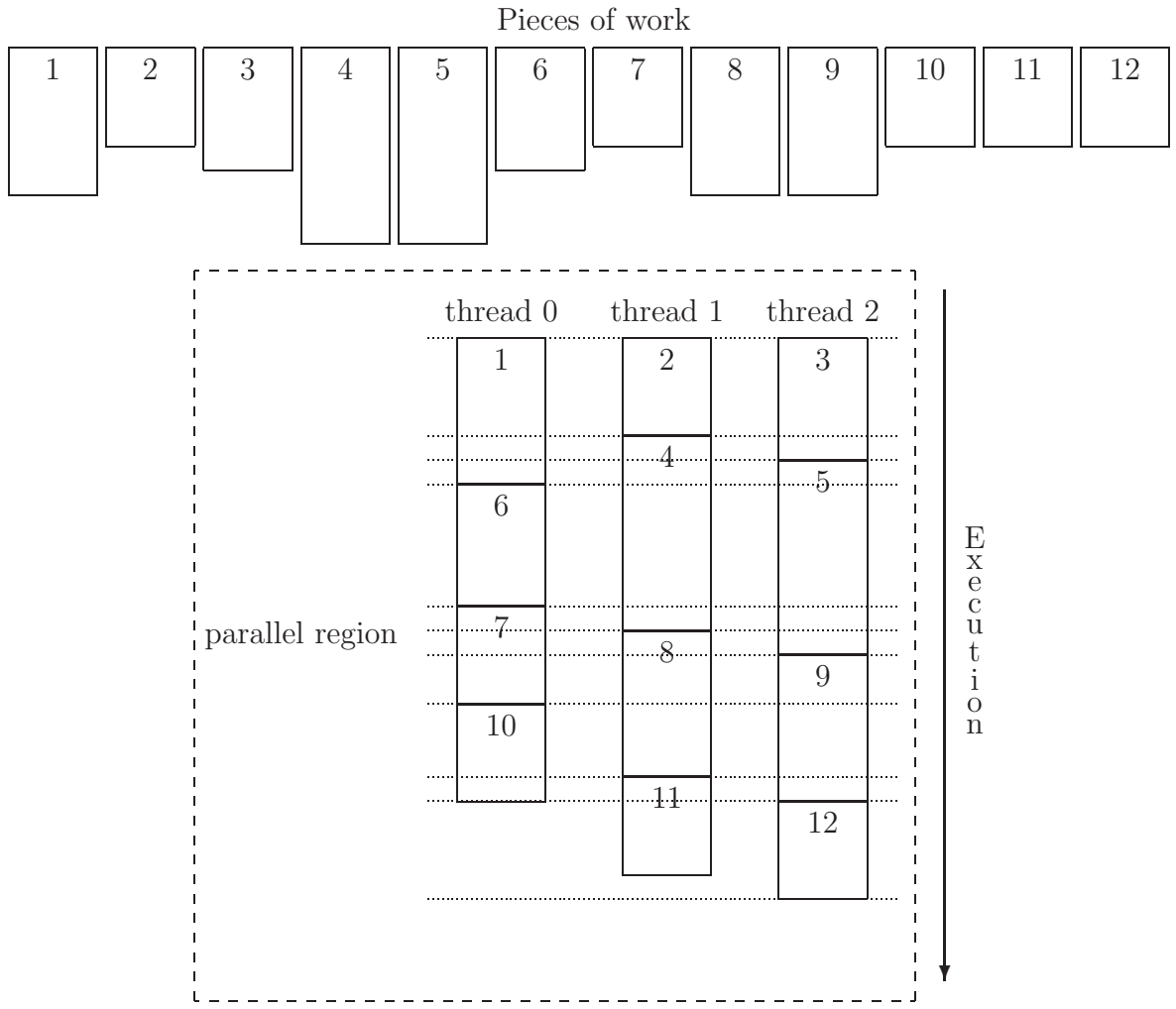

Figure 3.7: Graphical representation of the example explaining the working principle of the `SCHEDULE(DYNAMIC,`*`chunk`*`)` clause.

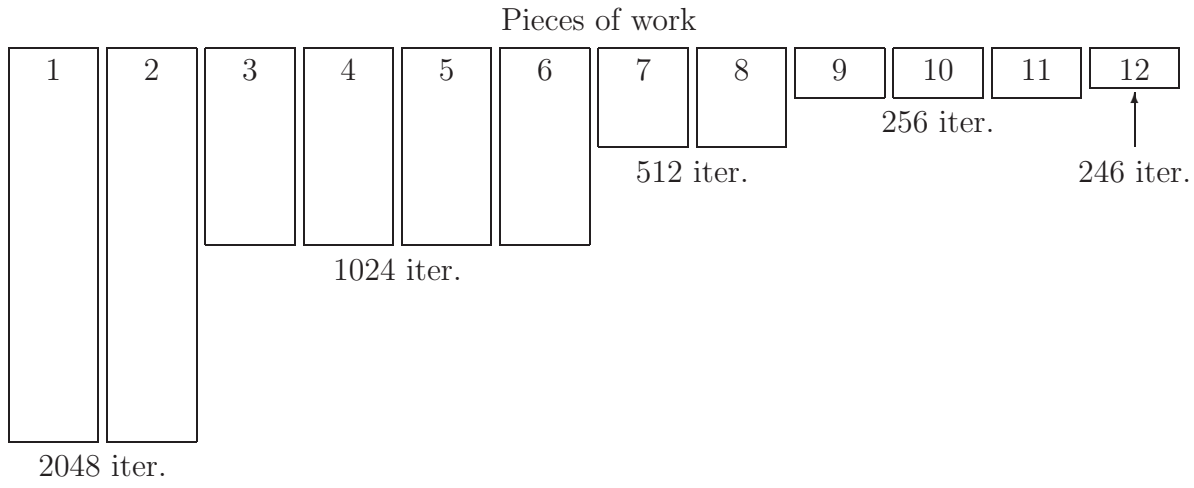

Figure 3.8: Graphical representation of the effect of the SCHEDULE(GUIDED,*chunk*) clause on a do-loop with 10230 iterations and with *chunk = 256*.

To see in figure 3.8 is that the last piece of work is smaller than the value specified in *chunk*. This is necessary in order to fulfill the total number of iterations of the do-loop, namely 10230.

Once the pieces of work have been created, the way in which they are distributed among the available threads is the same as in the previous scheduling method DYNAMIC.

RUNTIME : Any of the previous three scheduling methods needs to be fixed at the time of compiling the source code. But it may be of interest to be able to modify during runtime the way in which the work is distributed among the threads. This can be achieved by specifying the RUNTIME scheduling method:

```
!$OMP DO SCHEDULE(RUNTIME)
```

If this is done, then the content of the environment variable OMP_SCHEDULE specifies the scheduling method to be used. More information about the OMP_SCHEDULE environment variable can be found on page 70.

3.2.5 ORDERED

When a do-loop includes statements, which need to be executed sequentially, like in the following example:

```
do i = 1, 1000
  A(i) = 2 * A(i-1)
enddo
```

and the `!$OMP ORDERED`/`!$OMP END ORDERED` directive-pair is used to allow the parallelization of the do-loop, then it is mandatory to add the `ORDERED` clause to the `!$OMP DO` opening-directive to warn the compiler about the existence of the `ORDERED` section. The previous example would look as follows:

```
!$OMP DO ORDERED
  do i = 1, 1000
    !$OMP ORDERED
      A(i) = 2 * A(i-1)
    !$OMP END ORDERED
  enddo
!$OMP END DO
```

Chapter 4

The OpenMP run-time library

In some of the examples presented in previous chapters it has been mentioned the existence of a run-time library. It is the aim of the present chapter to introduce the so called **OpenMP Fortran API run-time library**, describe the subroutines and functions it includes and to explain its capabilities as well as its limitations.

The OpenMP run-time library is meant to serve as a control and query tool for the parallel execution environment, which the programmer can use from inside its program. Therefore, the run-time library is a set of external procedures with clearly defined interfaces. These interfaces are explicitly delivered by the OpenMP-implementation through a Fortran 95 module named `omp_lib`. In the following descriptions of the different routines, interfaces for them are shown: these may vary from one OpenMP-implementation to another in the precision/size of the variables, but in essence will be all equal.

4.1 Execution environment routines

The following functions and subroutines, part of the OpenMP Fortran API run-time library, allow to modify, from inside of a program, the conditions of the run-time environment that controls the parallel execution of the program:

4.1.1 `OMP_set_num_threads`

This subroutine sets the number of threads to be used by subsequent parallel regions. Therefore, it can only be called from outside of a parallel region. Its interface declaration looks as follows:

```
subroutine OMP_set_num_threads(number_of_threads)
integer(kind = OMP_integer_kind), intent(in) :: number_of_threads
end subroutine OMP_set_num_threads
```

where `number_of_threads` is the number of threads to be set. If dynamic adjustment is disabled, the value of `number_of_threads` is used as the number of threads for all subsequent parallel regions prior to the next call to this subroutine; otherwise, if dynamic adjustment is enabled, `number_of_threads` is the maximum number of threads that is allowed.

The number of threads imposed by a call to this subroutine has always precedence over the value specified in the `OMP_NUM_THREADS` environment variable.

4.1.2 OMP_get_num_threads

This function allows to know the number of threads currently in the team executing the parallel region from which it is called. Therefore, this function is meant to be used from inside a parallel region; although when called from a serial region or from inside a serialized nested parallel region, it returns the correct result, namely 1 thread. Its interface declaration is:

```
function OMP_get_num_threads()
integer(kind = OMP_integer_kind) :: OMP_get_num_threads
end function OMP_get_num_threads
```

where the returned value will be the number of threads in use.

4.1.3 OMP_get_max_threads

This function returns the maximum number of threads that can be used in the program. Its result may differ from the one returned by the function `OMP_get_num_threads`, if the dynamic adjustment of threads is enabled; otherwise, the results will be equal. Its interface declaration is equivalent to the one of the function `OMP_get_num_threads`:

```
function OMP_get_max_threads()
integer(kind = OMP_integer_kind) :: OMP_get_max_threads
end function OMP_get_max_threads
```

where the returned value is the maximum allowed number of threads. Since the returned value has nothing to do with the actual number of threads used in a given parallel region, this function can be called from serial regions as well as from parallel regions.

4.1.4 OMP_get_thread_num

This function returns the identification number of the current thread within the team. The identification numbers are between 0, the master thread, and `OMP_get_num_threads() - 1`, inclusive. The interface of this function is:

```
function OMP_get_thread_num()
integer(kind = OMP_integer_kind) :: OMP_get_thread_num
end function OMP_get_thread_num
```

This function returns 0, when called from inside a serial region or a serialized nested parallel region, which is the identification number of the lonely master thread.

4.1.5 OMP_get_num_procs

This function returns the number of processors available to the program. Its interface declaration is as follows:

```
function OMP_get_num_procs()
integer(kind = OMP_integer_kind) :: OMP_get_num_procs
end function OMP_get_num_procs
```

4.1.6 OMP_in_parallel

This function allows to know, if a given region of the program is being computed in parallel or not. For that purpose, it looks at all the parallel regions which are surrounding the block of source code enclosing the call to `OMP_in_parallel` and returns `.TRUE.`, if at least one of them is being executed in parallel; otherwise, it returns `.FALSE.`. The interface declaration of this function is the following one:

```
function OMP_in_parallel()
logical(kind = OMP_logical_kind) :: OMP_in_parallel
end function OMP_in_parallel
```

Since the returned value will be `.TRUE.` as soon as one of the enclosing parallel regions is being executed in parallel, this function returns `.TRUE.` from any point inside the dynamic extent of that parallel region, even inside nested serialized parallel regions (which are not considered as being executed in parallel!).

4.1.7 OMP_set_dynamic

The `OMP_set_dynamic` subroutine enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Its interface is:

```
subroutine OMP_set_dynamic(enable)
logical(kind = OMP_logical_kind), intent(in) :: enable
end subroutine OMP_set_dynamic
```

When `enable` is set equal to `.TRUE.`, the number of threads to be used by subsequent parallel regions can be adjusted automatically by the run-time environment in order to maximize the use of the SMP machine; otherwise, if `enable` is equal to `.FALSE.`, the dynamic adjustment is disabled.

As a consequence, the number of threads specified through the environment variable `OMP_NUM_THREADS` or by a call to the run-time library subroutine `OMP_set_num_threads` represents the maximum allowed number of threads which can be assigned to the program by the run-time environment.

The working conditions specified in a call to `OMP_set_dynamic` have precedence over the settings in the `OMP_DYNAMIC` environment variable.

The default for dynamic thread adjustment is OpenMP-implementation dependent, therefore a program needing a specific number of threads to run correctly should explicitly

disable the dynamic adjustment feature in order to be sure. Also implementations are not required to support dynamic adjustment, but they have at least to implement the present interface for portability reasons.

4.1.8 OMP_get_dynamic

This function returns the status of the dynamic thread adjustment mechanism: it will return .TRUE., if it is enabled, and .FALSE., if it is disabled. If the OpenMP-implementation does not implement dynamic thread adjustment, this function always returns .FALSE.. The interface declaration for this function is the following one:

```
function OMP_get_dynamic()
logical(kind = OMP_logical_kind) :: OMP_get_dynamic
end function OMP_get_dynamic
```

4.1.9 OMP_set_nested

This subroutine enables or disables the nested parallelism. Its interface declaration is:

```
subroutine OMP_set_nested(enable)
logical(kind = OMP_logical_kind), intent(in) :: enable
end subroutine OMP_set_nested
```

When enable is set equal to .TRUE., nested parallelism is enabled, while if it is equal to .FALSE., it is disabled. The default setting is always .FALSE., which means that, by default, nested parallel regions are serialized; that is, they are executed by a team with only one thread.

The number of threads used to execute nested parallel regions is OpenMP-implementation dependent. As a result, OpenMP-compliant implementations are allowed to serialize nested parallel regions even when nested parallelism is enabled.

A call to this subroutine overwrites the settings specified by the environment variable OMP_NESTED for the following nested parallel regions.

4.1.10 OMP_get_nested

This function returns the status of the nested parallelism mechanism: it will return .TRUE., if it is enabled, and .FALSE., if it is disabled. If the OpenMP-implementation does not support nested parallelism, this function always returns .FALSE.. The interface declaration for this function looks as follows:

```
function OMP_get_nested()
logical(kind = OMP_logical_kind) :: OMP_get_nested
end function OMP_get_nested
```

4.2 Lock routines

The second group of subroutines and functions deal with so called **locks**. These locks represent another synchronization mechanism different from the previously presented OpenMP directives like `!$OMP ATOMIC` or `!$OMP CRITICAL`.

A lock has to be seen as a flag which can be set or unset. Each thread may look at the status of the flag and handle in one way or the other depending on the status. Generally, a thread that looks at an unset flag will set it in order to get some privileges and to warn about that to the rest of the threads. The thread who sets a given lock is said to be the **ownership of the lock**. Once the thread no longer needs the acquired privileges, it unsets the lock: it **releases the ownership**.

The main difference between the previously presented synchronization directives and the locks is that in the latter case the threads are not obliged to respect the lock. This means that a given thread will not be affected by the status of the lock, if the thread is not taking care about the status of the lock. Although this seems to be a useless feature, it turns out to be extremely useful and also very flexible.

To show a possible application of the locks, consider the existence of a shared matrix `A(1:100,1:100)` and a parallel region with four threads. Each of the threads needs to modify values in the complete matrix `A`, but the order does not matter. In order to avoid race conditions, the following solution could be adopted:

```
!$OMP PARALLEL SHARED(A)

!$OMP CRITICAL
  ...                    !works on the matrix A
!$OMP END CRITICAL

!$OMP END PARALLEL
```

In this example the full matrix `A` is given to one of the threads while the others are waiting. This is clearly not a good option. Instead, the matrix could be divided into four blocks, namely `A1 = A(1:50,1:50)`, `A2 = A(51:100,1:50)`, `A3 = A(1:50,51:100)` and `A4 = A(51:100,51:100)`. Each of the threads is going to change values on all four blocks, but now it seems feasible that each thread gets one of the blocks, thereafter another of the blocks and so on. In this way racing conditions are avoided while allowing all the four threads to work on the matrix `A`.

This splitting of the synchronization can be achieved by using locks, while there is no means of doing the same using the previously presented OpenMP directives.

The natural sequence of using locks and the corresponding run-time library routines is the following one:

1. First, a lock and its associated lock variable (used as identification mechanism inside the program) need to be initialized.

2. Thereafter, a thread gets the ownership of a specified lock.

3. Meanwhile, all the threads potentially affected by the lock look at its state.

4. Once it has finished its work, it releases the ownership so that another thread can fetch it.

5. Finally, once the lock is no longer needed, it is necessary to eliminate it and its associated lock variable.

Two different types of locks exist:

simple locks : these locks cannot be locked if they are already in a locked state.

nestable locks : these locks may be locked multiple times by the same thread before being unlocked. They have associated a **nesting counter** which keeps track of the number of times a nestable lock has been locked and released.

Each run-time library routine, which works with locks, has two different versions: one which acts on simple locks and another which acts on nestable locks. It is not allowed to call one kind of routines with the type of variable of the other kind!

The run-time library routines needed to handle locks are described in the following subsections.

4.2.1 `OMP_init_lock` and `OMP_init_nest_lock`

These subroutines initialize a lock: a simple one and a nestable one, respectively. It is necessary to make a call to one of these subroutines before any other of the following routines can be used. Their interface declarations are:

```
subroutine OMP_init_lock(svar)
integer(kind = OMP_lock_kind), intent(out) :: svar
end subroutine OMP_init_lock

subroutine OMP_init_nest_lock(nvar)
integer(kind = OMP_nest_lock_kind), intent(out) :: nvar
end subroutine OMP_init_nest_lock
```

where the passed argument, called **lock variable**, is associated with a lock. Once a lock variable has been initialized, it can be used in subsequent calls to routines to refer to the specified lock. If the lock variable is nestable, its nesting counter is set to zero.

It is not allowed to call these subroutines using a lock variable that has already been associated to a lock, although it is not specified how an OpenMP-implementation has to react in such a case.

4.2.2 `OMP_set_lock` and `OMP_set_nest_lock`

When a thread wants to own a specific lock, it calls one of these subroutines. If the specified lock is not owned by another thread, then it gets the ownership of that lock. If the specified lock is already owned by another thread, then it has to wait until the other thread releases its ownership over the lock. The interface declarations of these subroutines are as follows:

```
subroutine OMP_set_lock(svar)
integer(kind = OMP_lock_kind), intent(inout) :: svar
end subroutine OMP_set_lock

subroutine OMP_set_nest_lock(nvar)
integer(kind = OMP_nest_lock_kind), intent(inout) :: nvar
end subroutine OMP_set_nest_lock
```

A simple lock will be available if it is unlocked, while a nestable lock is available if it is unlocked or if it is already owned by the thread executing the subroutine. In the latter case, the nesting counter of a nestable lock variable is incremented by one when it is set.

4.2.3 `OMP_unset_lock` and `OMP_unset_nest_lock`

These subroutines provide the means of releasing the ownership of a lock. Their interface declaration is:

```
subroutine OMP_unset_lock(svar)
integer(kind = OMP_lock_kind), intent(inout) :: svar
end subroutine OMP_unset_lock

subroutine OMP_unset_nest_lock(nvar)
integer(kind = OMP_nest_lock_kind), intent(inout) :: nvar
end subroutine OMP_unset_nest_lock
```

In the first case, a call to the subroutine releases the calling thread from the ownership of the simple lock associated to the corresponding lock variable `svar`, while in the second case the call to the subroutine decreases by one the nesting counter of `nvar` and only releases the calling thread, if the counter reaches zero.

To call these subroutines only makes sense, if the calling thread owns the corresponding lock; the behaviour is unspecified if the calling thread does not own the lock!

4.2.4 `OMP_test_lock` and `OMP_test_nest_lock`

These functions attempt to set a lock in the same way as the subroutines `OMP_set_lock` and `OMP_set_nest_lock` do, but they do not force the calling thread to wait until the specified lock is available. Their interface declarations are the following ones:

```
function OMP_test_lock(svar)
logical(kind = OMP_logical_kind) :: OMP_test_lock
integer(kind = OMP_lock_kind), intent(inout) :: svar
end function OMP_test_lock

function OMP_test_nest_lock(nvar)
integer(kind = OMP_integer_kind) :: OMP_test_nest_lock
integer(kind = OMP_nest_lock_kind), intent(inout) :: nvar
end function OMP_test_nest_lock
```

The OMP_test_lock function returns .TRUE. if the simple lock associated with svar is successfully set; otherwise it returns .FALSE..

The OMP_test_nest_lock function returns the new value of the nesting counter if the nestable lock associated with nvar is successfully set; otherwise it returns zero.

4.2.5 OMP_destroy_lock and OMP_destroy_nest_lock

Once a lock is no longer needed, it is necessary to uninitialize the corresponding lock variable. The present subroutines are meant for that. Their interface declarations are:

```
subroutine OMP_destroy_lock(svar)
integer(kind = OMP_lock_kind), intent(inout) :: svar
end subroutine OMP_destroy_lock

subroutine OMP_destroy_nest_lock(nvar)
integer(kind = OMP_nest_lock_kind), intent(inout) :: nvar
end subroutine OMP_destroy_nest_lock
```

4.2.6 Examples

To show the use of these subroutines and functions, the following example is given:

```
program Main
use omp_lib
implicit none

integer(kind = OMP_lock_kind) :: lck
integer(kind = OMP_integer_kind) :: ID

call OMP_init_lock(lck)

!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
  ID = OMP_get_thread_num()

  call OMP_set_lock(lck)
  write(*,*) "My thread is ", ID
  call OMP_unset_lock(lck)

!$OMP END PARALLEL

call OMP_destroy_lock(lck)

end program Main
```

In this example the first thing done, is the association of the lock variable lck with a simple lock, using OMP_init_lock. Once inside the parallel region, the threads ask for the ownership of the lock lck with the aid of a call to OMP_set_lock. The write to screen command following this call is only executed by the thread which owns the lock. Once the work is done, the lock is released with the call to OMP_unset_lock so that another

thread can perform the `write` process. Finally, since the lock is no longer needed after the parallel region, it is eliminated with the aid of `OMP_destroy_lock`.

The previous example could also have been implemented using the OpenMP directive `!$OMP CRITICAL`:

```
program Main
use omp_lib
implicit none

integer(kind = OMP_integer_kind) :: ID

!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
  ID = OMP_get_thread_num()

!$OMP CRITICAL
  write(*,*) "My thread is ", ID
!$OMP END CRITICAL

!$OMP END PARALLEL

end program Main
```

Looking at the previous example, the question rises about the necessity or use of the run-time lock routines, if OpenMP directives are able of doing the same. Additionally to the previously mentioned example with the matrix `A`, there are other cases where it is not possible to emulate the effect of the lock routines using OpenMP directives. For example:

```
program Main
use omp_lib
implicit none

integer(kind = OMP_lock_kind) :: lck
integer(kind = OMP_integer_kind) :: ID

call OMP_init_lock(lck)

!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
  ID = OMP_get_thread_num()

  do while(.not.OMP_test_lock(lck))
    ...    !work to do while the thread is waiting to get owner of the lock
  enddo

  ...      !work to do as the owner of the lock

!$OMP END PARALLEL

call OMP_destroy_lock(lck)

end program Main
```

In this example all the other threads, which do not own the lock, are not idle; instead, they are performing another task. This is something not possible to achieve using OpenMP directives. For certain problems, this possibility can be very useful, since the latency times due to synchronizations are filled in with work.

In the previous two examples simple locks are used. Nestable locks are useful when a subroutine needing them may be called from different locations inside a program. Then, it is not necessary to worry about who is calling. For example:

```
module data_types
use omp_lib, only: OMP_nest_lock_kind
implicit none

type number
  integer :: n
  integer(OMP_nest_lock_kind) :: lck
end type number

end module data_types
!-------------------------------------------------------!
program Main
use omp_lib
use data_types
implicit none

type(number) :: x

x%n = 0
call OMP_init_lock(x%lck)

!$OMP PARALLEL SECTIONS SHARED(x)
!$OMP SECTION
  call add(x,20)
!$OMP SECTION
  call substract(x,10)
!$OMP END PARALLEL

call OMP_destroy_lock(lck)

end program Main
!-------------------------------------------------------!
subroutine add(x,d)
use omp_lib
use data_types
implicit none
type(number) :: x
integer :: d

call OMP_set_nest_lock(x%lck)
x%n = x%n + d
call OMP_unset_nest_lock(x%lck)

end subroutine add
```

```
!-------------------------------------------------------!
subroutine substract(x,d)
use omp_lib
use data_types
implicit none
type(number) :: x
integer :: d

call OMP_set_nest_lock(x%lck)
call add(x,-d)
call OMP_unset_nest_lock(x%lck)

end subroutine substract
```

The subroutine `add` is called once by each thread. In both calls the updating of the variable `x` is correctly done, since a lock is allowing the access to only one thread at a time. The reason why `substract` is imposing the lock relays on the fact that `substract` is not obliged to know what the subroutine `add` internally does.

4.3 Timing routines

The last set of routines in the OpenMP Fortran API run-time library allows to measure absolute values of wallclock times with the idea to serve as an evaluation tool for programs.

4.3.1 `OMP_get_wtime`

This function returns the absolute wallclock time in seconds. By two calls to this function, it is possible to measure the time required to compute the commands enclosed between the two calls, like in the following example:

```
start = OMP_get_wtime()
... !work to be timed
end = OMP_get_wtime()
time = end - start
```

The interface declaration for this function is:

```
function OMP_get_wtime()
real(8) :: OMP_get_wtime
end function OMP_get_wtime
```

The times returned by this function are *"per-thread times"* by which is meant they are not required to be globally consistent across all the threads participating in a program.

4.3.2 OMP_get_wtick

This function returns the number of seconds between successive clock ticks. Its interface declaration is:

```
function OMP_get_wtick()
real(8) :: OMP_get_wtick
end function OMP_get_wtick
```

4.4 The Fortran 90 module omp_lib

The previously presented subroutines and functions have clearly defined interfaces which need to be known by the calling program in order to call them correctly. This is achieved through a Fortran 90 module called `omp_lib` which has to be supplied by the OpenMP-implementation. Even though, the following source code shows the content and aspect such a module should at least have[1]:

```
module omp_lib
implicit none

!Defines standard variable precisions of the OpenMP-implementation
integer, parameter :: OMP_integer_kind   = 4
integer, parameter :: OMP_logical_kind   = 4
integer, parameter :: OMP_lock_kind      = 8
integer, parameter :: OMP_nest_lock_kind = 8

!Defines the OpenMP version: OpenMP Fortran API v2.0
integer, parameter :: openmp_version = 200011

!Gives the explicit interface for each routine of the run-time library
interface
  subroutine OMP_set_num_threads(number_of_threads)
  integer(kind = OMP_integer_kind), intent(in) :: number_of_threads
  end subroutine OMP_set_num_threads

  function OMP_get_num_threads()
  integer(kind = OMP_integer_kind) :: OMP_get_num_threads
  end function OMP_get_num_threads

  function OMP_get_max_threads()
  integer(kind = OMP_integer_kind) :: OMP_get_max_threads
  end function OMP_get_max_threads

  function OMP_get_thread_num()
  integer(kind = OMP_integer_kind) :: OMP_get_thread_num
  end function OMP_get_thread_num
```

[1]The presented source code will not work with many Fortran 90 compilers, because they do not allow the use of `parameter` constants inside `interface` declarations. To solve this, it is necessary to substitute manually the values of these constants into the declarations.

```
function OMP_get_num_procs()
integer(kind = OMP_integer_kind) :: OMP_get_num_procs
end function OMP_get_num_procs

function OMP_in_parallel()
logical(kind = OMP_logical_kind) :: OMP_in_parallel
end function OMP_in_parallel

subroutine OMP_set_dynamic(enable)
logical(kind = OMP_logical_kind), intent(in) :: enable
end subroutine OMP_set_dynamic

function OMP_get_dynamic()
logical(kind = OMP_logical_kind) :: OMP_get_dynamic
end function OMP_get_dynamic

subroutine OMP_set_nested(enable)
logical(kind = OMP_logical_kind), intent(in) :: enable
end subroutine OMP_set_nested

function OMP_get_nested()
logical(kind = OMP_logical_kind) :: OMP_get_nested
end function OMP_get_nested

subroutine OMP_init_lock(var)
integer(kind = OMP_lock_kind), intent(out) :: var
end subroutine OMP_init_lock

subroutine OMP_init_nest_lock(var)
integer(kind = OMP_nest_lock_kind), intent(out) :: var
end subroutine OMP_init_nest_lock

subroutine OMP_destroy_lock(var)
integer(kind = OMP_lock_kind), intent(inout) :: var
end subroutine OMP_destroy_lock

subroutine OMP_destroy_nest_lock(var)
integer(kind = OMP_nest_lock_kind), intent(inout) :: var
end subroutine OMP_destroy_nest_lock

subroutine OMP_set_lock(var)
integer(kind = OMP_lock_kind), intent(inout) :: var
end subroutine OMP_set_lock

subroutine OMP_set_nest_lock(var)
integer(kind = OMP_nest_lock_kind), intent(inout) :: var
end subroutine OMP_set_nest_lock

subroutine OMP_unset_lock(var)
integer(kind = OMP_lock_kind), intent(inout) :: var
end subroutine OMP_unset_lock

subroutine OMP_unset_nest_lock(var)
```

```
  integer(kind = OMP_nest_lock_kind), intent(inout) :: var
  end subroutine OMP_unset_nest_lock

  function OMP_test_lock(var)
  logical(kind = OMP_logical_kind) :: OMP_test_lock
  integer(kind = OMP_lock_kind), intent(inout) :: var
  end function OMP_test_lock

  function OMP_test_nest_lock(var)
  integer(kind = OMP_integer_kind) :: OMP_test_nest_lock
  integer(kind = OMP_nest_lock_kind), intent(inout) :: var
  end function OMP_test_nest_lock

  function OMP_get_wtime()
  real(8) :: OMP_get_wtime
  end function OMP_get_wtime

  function OMP_get_wtick()
  real(8) :: OMP_get_wtick
  end function OMP_get_wtick
end interface

end module omp_lib
```

Chapter 5

The environment variables

The parallel environment, under which an OpenMP-parallel application runs, is controlled by **environment variables** or equivalent platform-specific mechanisms. These variables can be set from the command-line of the operating system or by calls to specific subroutines from the OpenMP Fortran API run-time library.

The way in which these environment variables are set from within the command-line of the operating system depends on the operating system itself. For example, on a Linux/Unix machine using csh as shell, the `setenv` command does the job:

```
> setenv OMP_NUM_THREADS 4
```

If the sh, ksh or bash shells are used, then the previous example looks as follows:

```
> OMP_NUM_THREADS=4
> export OMP_NUM_THREADS
```

In both cases it is possible to see the value of a given environment variable with the command `echo`:

```
> echo $OMP_NUM_THREADS
4
>
```

From within Microsoft Windows NT/2000/XP, the environment variables can be specified through the Control Panel (User Environment Variables) or by adding the corresponding lines in the `AUTOEXEC.BAT` file (AUTOEXEC.BAT Environment variables). In both cases the syntax is very simple:

```
OMP_NUM_THREADS = 4
```

The names of the environment variables must always be written in uppercase, while the characters assigned to them are case insensitive and may have leading or trailing white spaces.

In the rest of the present chapter the different environment variables are described, giving their possible values and effects on the behavior of an OpenMP-parallel program:

5.1 OMP_NUM_THREADS

This environment variable specifies the number of threads to be used during execution of the parallel regions inside an OpenMP-parallel program.

In principle, any integer number greater than 0 can be set as the number of threads to be used, although if this number is larger than the number of available physical processors, then the parallel running program will in general be *very slow*!

When dynamic adjustment of the number of threads is enabled, the value given in `OMP_NUM_THREADS` represents the maximum number of threads allowed.

Examples showing the syntaxis corresponding to Linux/Unix systems using the csh and bash shells are, respectively:

```
> setenv OMP_NUM_THREADS 16
```

```
> OMP_NUM_THREADS=16
> export OMP_NUM_THREADS
```

5.2 OMP_SCHEDULE

This environment variable affects the way in which `!$OMP DO` and `!$OMP PARALLEL DO` directives work, if they have their scheduling set to `RUNTIME`:

```
!$OMP DO SCHEDULE(RUNTIME)
```

The schedule type and chunk size for all such loops can be specified at run time by setting this environment variable to any of the recognized schedule types and to an optional chunk size, using the same notation as inside the `SCHEDULE` clause. The possible schedule types are `STATIC`, `DYNAMIC` or `GUIDED`: descriptions of them can be found in the explanation of the `!$OMP DO` directive.

The default value for the `OMP_SCHEDULE` environment variable is OpenMP-implementation dependent, so it is necessary to look at its documentation in order to know that.

If the optional chunk size is not set, a chunk size of 1 is assumed, except in the case of a `STATIC` schedule, where the default chunk size is set equal to the total number of do-loop iterations divided by the number of threads which act on the parallelized do-loop.

Examples showing the syntaxis corresponding to Linux/Unix systems using the csh and bash shells are, respectively:

```
> setenv OMP_SCHEDULE "GUIDED,4"
```

```
> OMP_SCHEDULE=dynamic
> export OMP_SCHEDULE
```

5.3 OMP_DYNAMIC

In large SMP machines it is common that several different programs share the available resources of the machine. Under such conditions it can happen that fixing the number of threads to be used inside the parallel regions of a program may lead to unefficient use of the resources.

Therefore, it is possible to enable the run-time environment of the SMP machine to adjust the number of threads dynamically, in order to maximize the use of the machine. This is done through the present environment variable which can be set equal to `TRUE`, if dynamic assignment is allowed, or equal to `FALSE`, if a fixed number of threads are to be used.

The default value for the `OMP_DYNAMIC` environment variable is OpenMP-implementation dependent, so it is necessary to look at its documentation in order to know that.

Examples showing the syntaxis corresponding to Linux/Unix systems using the csh and bash shells are, respectively:

```
> setenv OMP_DYNAMIC TRUE

> OMP_DYNAMIC=FALSE
> export OMP_DYNAMIC
```

5.4 OMP_NESTED

This environment variable specifies what the run-time environment has to do, when nested OpenMP directives are found.

If its value is set equal to `TRUE`, nested parallelism is enabled and a new set of threads forming a new team is created to work on the nested directive. The result is a tree structure as shown in figure 1.2, for the case of using two threads at each level of nesting.

If the environment variable `OMP_NESTED` is set equal to `FALSE`, then the nested directives are serialized; that is, they are executed by a team with only one thread.

Examples showing the syntaxis corresponding to Linux/Unix systems using the csh and bash shells are, respectively:

```
> setenv OMP_NESTED TRUE

> OMP_NESTED=FALSE
> export OMP_NESTED
```