# Hands-on introduction to parallelization with OpenMP

R. Pastor-Satorras

Departament de Física
UPC

Eines Informàtiques Avançades / EIA

# Parallel computing

Parallel computing consists, in general, the use of more than one computer resource to run a program

1. The program must be broken in a set of parts that can be executed concurrently (at the same time)
2. Each part is set up in a set of instructions
3. Each set of instructions is executed simultaneously in a different computer resource

The benefit of parallel computing is that the program, running in more that one resource, can be executed more efficiently, saving time and possibly money

# Elements of parallel computing

A *CPU* is an electronic circuit in a computer that handles the instructions that receives from the hardware and the software. A computer can have one or more CPUs

A *core* a basic processing unit of a CPU that receives instructions and executes them, one instruction at a time. Cores are located inside the CPU. A CPU can have more than one core

*Threads* are virtual components that divide a physical core into virtual multiple cores. When one opens an application the operating system (OS) creates a thread for performing the tasks associated to the application.

# Elements of parallel computing

A *hardware thread* is a physical core, so a e.g. 6-core CPU can genuinely support 6 hardware threads at the same time, running independently.

Hyper-threading (common in Intel and AMD architectures) is a process by which a CPU doubles its physical cores into *virtual* cores that are treated by the operating systems as if they were actual hardware cores. Thus a 6-core CPU with hyper-threading can support 12 threads (12 *logical cores*) at the same time, running independently

A CPU can generate as many threads as it needs, according to the applications that are running. This is done by *time-slicing*, in which each thread gets some milliseconds to be executed in a core before the OS schedules another thread to be executed in the same core

In a normal operation of a computer, many threads are needed to run the different application open (browser, editor, viewer)

In performing a parallel calculation, it is interesting to use at most the number of logical cores available to extract the best performance

# The OpenMP framework for parallel computing

OpenMP (standing for Open multi-processing) is a programming interface (API) based in multi-threading and memory sharing, that supports parallel computing for different computer languages (C, C++, Fortran)

OpenMP is managed by the non-profit consortium *OpenMP Architecture Review Board*, including leading hardware and software vendors, such as AMD, IBM, Intel, Cray or Nvidia

OpenMP uses a scalable model that offers programmers a simple interface for developing parallel applications in a wide range of platforms

OpenMP allows to run parallel programs on a personal computer with several cores, or in a node of a cluster

# The OpenMP framework for parallel computing

OpenMP is one of the simplest parallelization frameworks, usually involving only some small changes in a working serial code

One can parallelize only some parts of the program by opening a *parallel block*, that will be executed in parallel, the rest being executed serially

One should thus invest effort in parallelizing only the time-consuming sections of the code (bottle-necks)

You can get some (or a lot of) speed-up with a limited investment of time and effort

Moreover, serial and OpenMP versions can easily coexist, even in the same source file!

# The basics of OpenMP

The API of OpenMP consists of

1. Compiler directives that are preprocessed by the compiler
   - Directives signaled by a *sentinel*: `!$omp` in Fortran, `#pragma omp` in C/C++
2. A set of subroutines that can affect the behavior of the computations, loaded in a small number of libraries
3. A set of environment variables, used to pass information to the compiler from outside the program

With these elements, we can control when a parallel block opens and closes, and how the memory (variables) are managed inside the block

# The basics of OpenMP

To compile an OpenMP program, we must instruct explicitly the compiler to link the appropriate libraries:

```
bash-3.2$ gfortran -fopenmp program_file.f08
```

The examples shown here were compiled with a `gfortran` from the GCC Compiler Collection

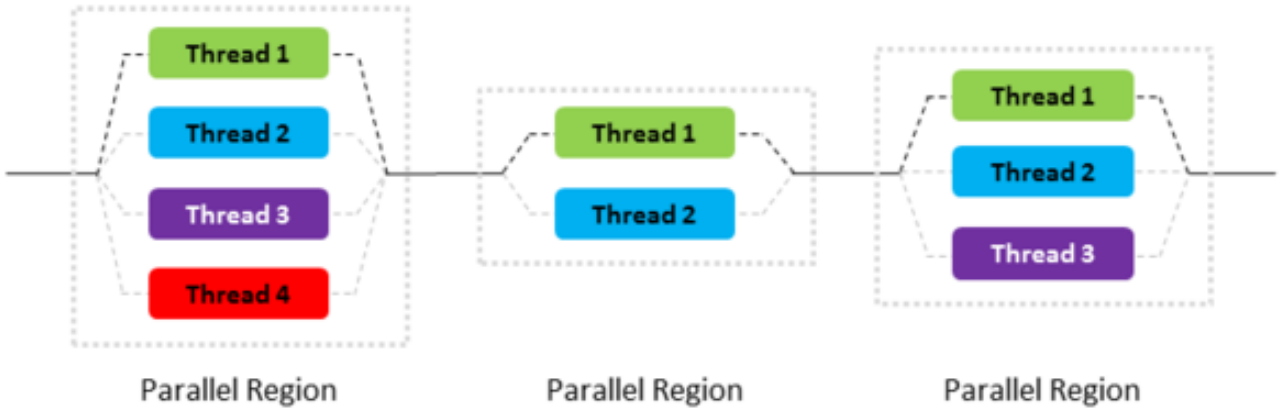
```
bash-3.2$ gfortran --version
GNU Fortran (Homebrew GCC 12.2.0) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR P
```

Note that other versions of fortran might use slightly different compilation flags

# How OpenMP works

OpenMP works following the *fork-join* model

1. The program starts with a single *master* thread
2. When needed, in a parallel block of the program, the master creates a team of parallel *worker* threads (*fork*)
3. The statements in a parallel block are executed in parallel by the spawned threads
4. At the end of the parallel block, worker threads synchronize and join the master thread (*join*)



# How OpenMP works

The OpenMP framework considers a *shared memory* approach: All threads share the same memory space and can have access to all the variables defined in the program

This mostly implies that OpenMP is valid for single machine parallelization.

## MPI

It is possible to implement parallelization using different machines that do not share the same memory. To do so, the different parallel processes must exchange messages. One such implementation is the *Message Passing Interface* (MPI). More on this in the following lectures of this course

The fact that all threads can share all data can be convenient, but it can also lead to problems: *Data races* when the same memory position is changed by different threads, and whose value depends on which thread accessed it last

# How OpenMP works

In an OpenMP program, there are two kinds of data

1. Shared data
   - There is only one instance of the data
   - All threads can read and write the data simultaneously
   - Changes are immediately visible to all threads
2. Private
   - Each thread has an independent copy of the data
   - No other thread can access the data of a given thread
   - Changes to private data can only be seen by the thread that owns it

By default, all data is shared. We must declare data as private, if needed, at the beginning of a parallel block, specially to avoid data races

# The first OpenMP program: Parallel "Hello world"

Simplest serial version of the Hello World program

```
! hello_serial.f08
program hello_serial
  implicit none

  print *, "Hello world"

end program hello_serial
```

# The first OpenMP program: Parallel "Hello world"

## Simplest serial version of the Hello World program

```
! hello_serial.f08
program hello_serial
  implicit none

  print *, "Hello world"

end program hello_serial
```

```
bash-3.2$ gfortran hello_serial.f08
bash-3.2$ ./a.out
Hello world
bash-3.2$
```

# Defining a parallel region

OpenMP programs must load the module `omp_lib`

To define a parallel region, we use the compiler directives, starting with the Fotran sentinels, *!omp parallel* and *!omp end parallel*

```
! hello_parallel.f08
program hello_parallel
  use, intrinsic :: omp_lib
  implicit none

  !$omp parallel

  print *, "Hello world"

  !$omp end parallel

end program hello_parallel
```

# Defining a parallel region

OpenMP programs must load the module `omp_lib`

To define a parallel region, we use the compiler directives, starting with the Fotran sentinels, *!omp parallel* and *!omp end parallel*

```
! hello_parallel.f08
program hello_parallel
  use, intrinsic :: omp_lib
  implicit none

  !$omp parallel

  print *, "Hello world"

  !$omp end parallel

end program hello_parallel
```

```
bash-3.2$ gfortran -fopenmp hello_parallel.f08
bash-3.2$ ./a.out
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
bash-3.2$
```

# Defining a parallel region

OpenMP programs must load the module `omp_lib`

To define a parallel region, we use the compiler directives, starting with the Fotran sentinels, *!omp parallel* and *!omp end parallel*

```
! hello_parallel.f08
program hello_parallel
  use, intrinsic :: omp_lib
  implicit none

  !$omp parallel

  print *, "Hello world"

  !$omp end parallel

end program hello_parallel
```

```
bash-3.2$ gfortran -fopenmp hello_parallel.f08
bash-3.2$ ./a.out
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
 Hello world
bash-3.2$
```

The parallel region spawns a number of threads equal to the number of logical cores (6 hardware cores in my machine). Each thread executes independently the body of instructions of the parallel block. The behavior of the parallel block can be modified using one or several *clauses*

# Identifying the threads

Each thread is identified by a number. The number 0 is reserved for the master thread. We can identify a thread using the function `omp_get_thread_num()`

# Identifying the threads

Each thread is identified by a number. The number 0 is reserved for the master thread. We can identify a thread using the function `omp_get_thread_num()`

```
! hello_parallel_2.f08
program hello_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer id

  !$omp parallel

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_2
```

# Identifying the threads

Each thread is identified by a number. The number 0 is reserved for the master thread. We can identify a thread using the function `omp_get_thread_num()`

```
! hello_parallel_2.f08
program hello_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer id

  !$omp parallel

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_2
```

```
bash-5.1$ gfortran -fopenmp hello_parallel_2.f08
bash-5.1$ a.out
 Hello world from thread           5
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
bash-5.1$ a.out
 Hello world from thread           1
 Hello world from thread           0
 Hello world from thread           5
 Hello world from thread           2
 Hello world from thread           4
 Hello world from thread           6
 Hello world from thread           6
 Hello world from thread           6
```

# Identifying the threads

Each thread is identified by a number. The number 0 is reserved for the master thread. We can identify a thread using the function `omp_get_thread_num()`

```
! hello_parallel_2.f08
program hello_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer id

  !$omp parallel

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_2
```

```
bash-5.1$ gfortran -fopenmp hello_parallel_2.f08
bash-5.1$ a.out
 Hello world from thread           5
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
bash-5.1$ a.out
 Hello world from thread           1
 Hello world from thread           0
 Hello world from thread           5
 Hello world from thread           2
 Hello world from thread           4
 Hello world from thread           6
 Hello world from thread           6
 Hello world from thread           6
```

Something strange is going on here:

# Identifying the threads

Each thread is identified by a number. The number 0 is reserved for the master thread. We can identify a thread using the function `omp_get_thread_num()`

```
! hello_parallel_2.f08
program hello_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer id

  !$omp parallel

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_2
```

```
bash-5.1$ gfortran -fopenmp hello_parallel_2.f08
bash-5.1$ a.out
 Hello world from thread           5
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
 Hello world from thread           4
bash-5.1$ a.out
 Hello world from thread           1
 Hello world from thread           0
 Hello world from thread           5
 Hello world from thread           2
 Hello world from thread           4
 Hello world from thread           6
 Hello world from thread           6
 Hello world from thread           6
```

Something strange is going on here: A data race on the variable `id`, that is being accessed and written by all threads simultaneously

# Identifying the threads

The variable `id` must be proper of each thread. If we do not want it to be overwritten, and messed, by other threads, it must be private. We declare it private with the clause `private(id)`

```
! hello_parallel_2.f08
program hello_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer id

  !$omp parallel private(id)

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_2
```

# Identifying the threads

The variable `id` must be proper of each thread. If we do not want it to be overwritten, and messed, by other threads, it must be private. We declare it private with the clause `private(id)`

```
! hello_parallel_2.f08
program hello_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer id

  !$omp parallel private(id)

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_2
```

```
bash-5.1$ gfortran -fopenmp hello_parallel_2.1.f08
bash-5.1$ ./a.out
 Hello world from thread           0
 Hello world from thread           3
 Hello world from thread           7
 Hello world from thread           6
 Hello world from thread           5
 Hello world from thread           2
 Hello world from thread           4
 Hello world from thread           1
bash-5.1$ ./a.out
 Hello world from thread           3
 Hello world from thread           4
 Hello world from thread           6
 Hello world from thread           2
 Hello world from thread           1
 Hello world from thread           7
 Hello world from thread           5
 Hello world from thread           0
```

# Identifying the threads

The variable `id` must be proper of each thread. If we do not want it to be overwritten, and messed, by other threads, it must be private. We declare it private with the clause `private(id)`

```
! hello_parallel_2.f08
program hello_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer id

  !$omp parallel private(id)

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_2
```

```
bash-5.1$ gfortran -fopenmp hello_parallel_2.1.f08
bash-5.1$ ./a.out
 Hello world from thread           0
 Hello world from thread           3
 Hello world from thread           7
 Hello world from thread           6
 Hello world from thread           5
 Hello world from thread           2
 Hello world from thread           4
 Hello world from thread           1
bash-5.1$ ./a.out
 Hello world from thread           3
 Hello world from thread           4
 Hello world from thread           6
 Hello world from thread           2
 Hello world from thread           1
 Hello world from thread           7
 Hello world from thread           5
 Hello world from thread           0
```

Threads finish executing their task in random times

# Controlling the number of threads

We can impose the number of threads using the procedure `omp_set_num_threads(nthreads)`

```
! hello_parallel_3.f08
program hello_parallel_3
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: nthreads = 7

  integer id

  call omp_set_num_threads(nthreads)

  !$omp parallel private(id)

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_3
```

# Controlling the number of threads

We can impose the number of threads using the procedure `omp_set_num_threads(nthreads)`

```
! hello_parallel_3.f08
program hello_parallel_3
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: nthreads = 7

  integer id

  call omp_set_num_threads(nthreads)

  !$omp parallel private(id)

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_3
```

```
bash-3.2$ gfortran -fopenmp hello_parallel_3.f08
bash-3.2$ ./a.out
 Hello world from thread           4
 Hello world from thread           3
 Hello world from thread           2
 Hello world from thread           5
 Hello world from thread           1
 Hello world from thread           0
 Hello world from thread           6
bash-3.2$
```

# Controlling the number of threads

We can impose the number of threads using the procedure `omp_set_num_threads(nthreads)`

```
! hello_parallel_3.f08
program hello_parallel_3
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: nthreads = 7

  integer id

  call omp_set_num_threads(nthreads)

  !$omp parallel private(id)

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_3
```

```
bash-3.2$ gfortran -fopenmp hello_parallel_3.f08
bash-3.2$ ./a.out
 Hello world from thread           4
 Hello world from thread           3
 Hello world from thread           2
 Hello world from thread           5
 Hello world from thread           1
 Hello world from thread           0
 Hello world from thread           6
bash-3.2$
```

We can use as many threads as we wish, but it is usually most efficient to use at most as many threads as logical cores. Sometimes, even less, just the number of hardware cores

# Controlling the number of threads

We can also use the environment variable `OMP_NUM_THREADS`

```
! hello_parallel_2.f08
program hello_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer id

  !$omp parallel

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_2
```

# Controlling the number of threads

## We can also use the environment variable OMP_NUM_THREADS

```
! hello_parallel_2.f08
program hello_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer id

  !$omp parallel

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_2
```

```
bash-3.2$ gfortran -fopenmp hello_parallel_2.f08
bash-3.2$ export OMP_NUM_THREADS=3
bash-3.2$ ./a.out
 Hello world from thread           0
 Hello world from thread           1
 Hello world from thread           2
bash-3.2$ export OMP_NUM_THREADS=7
bash-3.2$ ./a.out
 Hello world from thread           5
 Hello world from thread           4
 Hello world from thread           0
 Hello world from thread           2
 Hello world from thread           3
 Hello world from thread           1
 Hello world from thread           6
bash-3.2$
```

# Controlling the number of threads

We can also use the environment variable `OMP_NUM_THREADS`

```
! hello_parallel_2.f08
program hello_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer id

  !$omp parallel

  id = omp_get_thread_num()

  print *, "Hello world from thread", id

  !$omp end parallel

end program hello_parallel_2
```

```
bash-3.2$ gfortran -fopenmp hello_parallel_2.f08
bash-3.2$ export OMP_NUM_THREADS=3
bash-3.2$ ./a.out
 Hello world from thread           0
 Hello world from thread           1
 Hello world from thread           2
bash-3.2$ export OMP_NUM_THREADS=7
bash-3.2$ ./a.out
 Hello world from thread           5
 Hello world from thread           4
 Hello world from thread           0
 Hello world from thread           2
 Hello world from thread           3
 Hello world from thread           1
 Hello world from thread           6
bash-3.2$
```

Useful to experiment the speed-up for different number of threads

The function `omp_set_num_threads()` overrides the environment variable `OMP_NUM_THREADS`

Notice that we have used here the `bash` shell of UNIX to set the environmental variable `OMP_NUM_THREADS`

# Practical use of OpenMP

As we have seen, simple parallel blocks just perform the set instructions inside the block independently for each one of the spawned threads

But we usually do not want this boring behavior in a parallel program

Simple parallel blocks can be used to perform practical tasks, but this requires that we hand-code the different tasks to be performed by each thread using the `id` of the threads

Luckily, OpenMP implements parallelization instructions for the most common time consuming operation in programming, which is the loop

# Parallelization of loops

The loop is one of the most used expression in scientific computing, and it can be quite time-consuming, specially is loops are nested

Serial version of a loop

# Parallelization of loops

The loop is one of the most used expression in scientific computing, and it can be quite time-consuming, specially is loops are nested

Serial version of a loop

```
  ! loop_series.f08
program loop_series
  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: n = 100000000

  real(kind=dp), dimension(:), allocatable :: a, b, c
  integer :: i

  allocate(a(n), b(n), c(n))

  ! Initialise the PRNG and fill matrices A and B
  ! with random numbers.
  call random_seed()
  call random_number(a)
  call random_number(b)

  do i = 1, n
     c(i) = exp(sqrt(a(i)*b(i)))
  end do

end program loop_series
```

# Parallelization of loops

The loop is one of the most used expression in scientific computing, and it can be quite time-consuming, specially is loops are nested

## Serial version of a loop

```
  ! loop_series.f08
program loop_series
  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: n = 100000000

  real(kind=dp), dimension(:), allocatable :: a, b, c
  integer :: i

  allocate(a(n), b(n), c(n))

  ! Initialise the PRNG and fill matrices A and B
  ! with random numbers.
  call random_seed()
  call random_number(a)
  call random_number(b)

  do i = 1, n
     c(i) = exp(sqrt(a(i)*b(i)))
  end do

end program loop_series
```

```
bash-3.2$ gfortran loop_series.f08
bash-3.2$ time ./a.out

real        0m2.608s
user        0m2.111s
sys         0m0.438s
bash-3.2$
```

# Parallelization of loops

A loop can be parallelized in OpenMP surrounding the body of the loop with the directives *!omp do* and *!omp end do*, inside a parallel block

```
  ! loop_parallel.f08
program loop_parallel
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: n = 100000000
  real(kind=dp), dimension(:), allocatable :: a, b, c
  integer :: i

  allocate(a(n), b(n), c(n))

  ! Initialise the PRNG and fill matrices A and B
  ! with random numbers.
  call random_seed()
  call random_number(a)
  call random_number(b)

  !$omp parallel
  !$omp do

   do i = 1, n
     c(i) = exp(sqrt(a(i)*b(i)))
  end do

  !$omp end do
  !$omp end parallel
end program loop_parallel
```

# Parallelization of loops

A loop can be parallelized in OpenMP surrounding the body of the loop with the directives *!omp do* and *!omp end do*, inside a parallel block

```
  ! loop_parallel.f08
program loop_parallel
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: n = 100000000
  real(kind=dp), dimension(:), allocatable :: a, b, c
  integer :: i

  allocate(a(n), b(n), c(n))

  ! Initialise the PRNG and fill matrices A and B
  ! with random numbers.
  call random_seed()
  call random_number(a)
  call random_number(b)

  !$omp parallel
  !$omp do

   do i = 1, n
     c(i) = exp(sqrt(a(i)*b(i)))
  end do

  !$omp end do
  !$omp end parallel
end program loop_parallel
```

```
bash-3.2$ gfortran -fopenmp loop_parallel.f08
bash-3.2$ time ./a.out

real        0m1.537s
user        0m2.935s
sys         0m0.576s
```

# Scheduling of iterations

The effect of the *!omp do* directive is to split the iterations of the loop between the threads in the parallel block, in such a way that each thread performs a different number of interations

The standard does not specify how the iterations are partitioned between threads, but most compilers split the loop in equal sized chunks by default

# Scheduling of iterations

The effect of the *!omp do* directive is to split the iterations of the loop between the threads in the parallel block, in such a way that each thread performs a different number of interations

The standard does not specify how the iterations are partitioned between threads, but most compilers split the loop in equal sized chunks by default

```
do i=1, 1000
```
(in a four core processor)

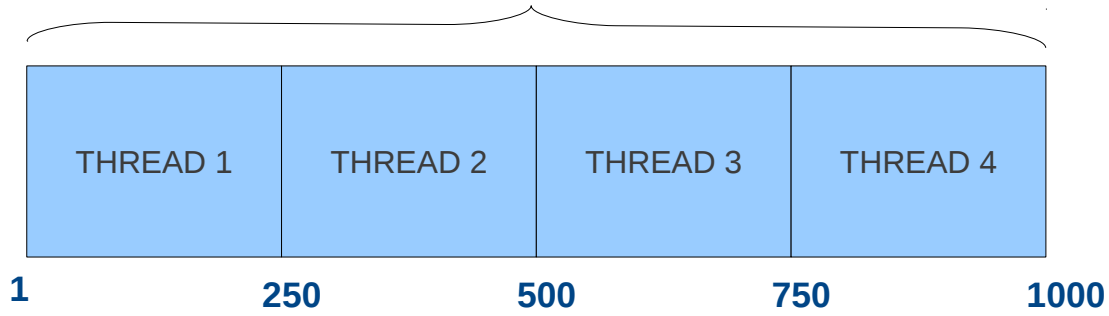
# Scheduling of iterations

The effect of the *!omp do* directive is to split the iterations of the loop between the threads in the parallel block, in such a way that each thread performs a different number of interations

The standard does not specify how the iterations are partitioned between threads, but most compilers split the loop in equal sized chunks by default

```
do i=1, 1000
```

(in a four core processor)




Each one of the 4 threads executes a consecutive chunk of 250 iterations

This is called a *static* schedule

# Scheduling of iterations

The way iterations are assigned to threads can be specified using the clause `schedule`

For example, the static schedule can be imposed with the directive

```
!omp do schedule(static)
```

Each threads executes the same number of iterations, but this can be a problem if some iterations take more time than others. Some threads can finish their tasks, and will have to wait idly until others finish

# Scheduling of iterations

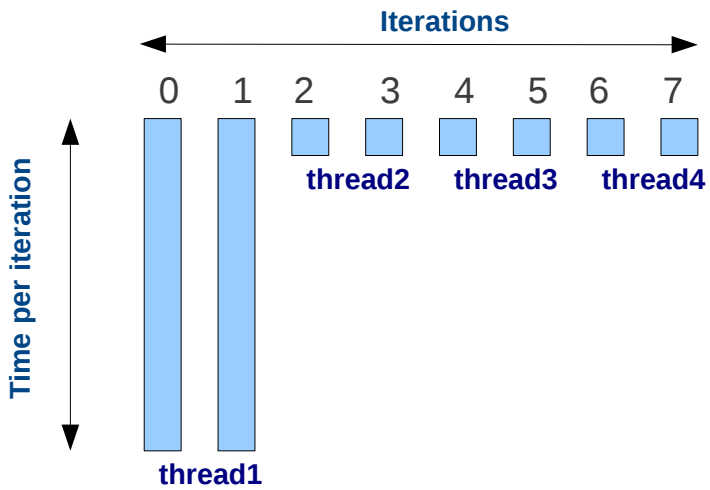
The way iterations are assigned to threads can be specified using the clause `schedule`

For example, the static schedule can be imposed with the directive

```
!omp do schedule(static)
```

Each threads executes the same number of iterations, but this can be a problem if some iterations take more time than others. Some threads can finish their tasks, and will have to wait idly until others finish

1. Load imbalance



# Scheduling of iterations

The *dynamic* schedule assigns tasks in chunks of given size to the threads. When a thread finishes its tasks, it take care of another chunk

```
!omp do schedule(dynamic, chunksize)
```

One must be careful, however, because dynamic scheduling imposes an overhead in managing the tasks assigned

# Differences in scheduling

Static schedule for a clearly imbalanced computation

# Differences in scheduling

## Static schedule for a clearly imbalanced computation

```
  ! imbalance_static.f08
program imbalance_static
  use, intrinsic :: omp_lib
  implicit none
  integer ::  i, value, num_loops = 300000

  !$omp parallel
  !$omp do schedule(static)
  do i=1, num_loops
     call computation(i, value)
  enddo
  !$omp end do
  !$omp end parallel

end program imbalance_static

subroutine computation(n, result)
  implicit none
  integer, intent(in) :: n
  integer, intent(out) :: result
  integer ::  i, tot

  tot = 0
  do i=1,n
     tot = tot + 1
  end do
  result = tot
end subroutine computation
```

# Differences in scheduling

## Static schedule for a clearly imbalanced computation

```
  ! imbalance_static.f08
program imbalance_static
  use, intrinsic :: omp_lib
  implicit none
  integer ::  i, value, num_loops = 300000

  !$omp parallel
  !$omp do schedule(static)
  do i=1, num_loops
     call computation(i, value)
  enddo
  !$omp end do
  !$omp end parallel

end program imbalance_static

subroutine computation(n, result)
  implicit none
  integer, intent(in) :: n
  integer, intent(out) :: result
  integer ::  i, tot

  tot = 0
  do i=1,n
     tot = tot + 1
  end do
  result = tot
end subroutine computation
```

```
bash-3.2$ gfortran -fopenmp imbalance_static.f08
bash-3.2$ export OMP_NUM_THREADS=10
bash-3.2$ time ./a.out

real        0m14.552s
user        1m19.914s
sys         0m0.228s
bash-3.2$
```

# Differences in scheduling

Implemented instead with a dynamic scheduling

# Differences in scheduling

## Implemented instead with a dynamic scheduling

```
  ! imbalance_dynamic.f08
program imbalance_dynamic
  use, intrinsic :: omp_lib
  implicit none
  integer ::  i, value, num_loops = 300000

  !$omp parallel
  !$omp do schedule(dynamic, 1000)
  do i=1, num_loops
     call computation(i, value)
  enddo
  !$omp end do
  !$omp end parallel

end program imbalance_dynamic

subroutine computation(n, result)
  implicit none
  integer, intent(in) :: n
  integer, intent(out) :: result
  integer ::  i, tot

  tot = 0
  do i=1,n
     tot = tot + 1
  end do
  result = tot
end subroutine computation
```

# Differences in scheduling

## Implemented instead with a dynamic scheduling

```
  ! imbalance_dynamic.f08
program imbalance_dynamic
  use, intrinsic :: omp_lib
  implicit none
  integer ::  i, value, num_loops = 300000

  !$omp parallel
  !$omp do schedule(dynamic, 1000)
  do i=1, num_loops
     call computation(i, value)
  enddo
  !$omp end do
  !$omp end parallel

end program imbalance_dynamic

subroutine computation(n, result)
  implicit none
  integer, intent(in) :: n
  integer, intent(out) :: result
  integer ::  i, tot

  tot = 0
  do i=1,n
     tot = tot + 1
  end do
  result = tot
end subroutine computation
```

```
bash-3.2$ gfortran -fopenmp imbalance_dynamic.f08
bash-3.2$ export OMP_NUM_THREADS=10
bash-3.2$ time ./a.out

real        0m9.842s
user        1m34.754s
sys         0m0.209s
bash-3.2$
```

# Sums inside loops

Imagine that we want, as is very usual, to use a loop to compute a value.

```
  ! loop_reduce.f08
program loop
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: num_loops = 100000000

  integer i
  real(kind=dp) total

  total = 0.0

  !$omp parallel
  !$omp do

  do i = 1, num_loops
     total = total + exp(-real(i, dp))*cos(real(i, dp))
  enddo

  !$omp end do
  !$omp end parallel

  print *, "Total =", total

  ! correct value total = 8.59725722621757737E-002
end program loop
```

# Sums inside loops

Imagine that we want, as is very usual, to use a loop to compute a value.

```
  ! loop_reduce.f08
program loop
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: num_loops = 100000000

  integer i
  real(kind=dp) total

  total = 0.0

  !$omp parallel
  !$omp do

  do i = 1, num_loops
     total = total + exp(-real(i, dp))*cos(real(i, dp))
  enddo

  !$omp end do
  !$omp end parallel

  print *, "Total =", total

  ! correct value total = 8.5972572262175737E-002
end program loop
```

```
bash-3.2$ gfortran -fopenmp loop_reduce.f08
bash-3.2$ ./a.out
 Total =  -1.0060599750366619E-002
bash-3.2$ ./a.out
 Total =   2.0058833612066657E-006
bash-3.2$ ./a.out
 Total =  -5.6314091257512161E-002
bash-3.2$
```

# Sums inside loops

Imagine that we want, as is very usual, to use a loop to compute a value.

```
  ! loop_reduce.f08
program loop
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: num_loops = 100000000

  integer i
  real(kind=dp) total

  total = 0.0

  !$omp parallel
  !$omp do

  do i = 1, num_loops
     total = total + exp(-real(i, dp))*cos(real(i, dp))
  enddo

  !$omp end do
  !$omp end parallel

  print *, "Total =", total

  ! correct value total = 8.5972572262175737E-002
end program loop
```

```
bash-3.2$ gfortran -fopenmp loop_reduce.f08
bash-3.2$ ./a.out
 Total =  -1.0060599750366619E-002
bash-3.2$ ./a.out
 Total =   2.0058833612066657E-006
bash-3.2$ ./a.out
 Total =  -5.6314091257512161E-002
bash-3.2$
```

What's going on here??

# Reduction of loops

The problem here are issues with the fact that `total` is shared by all the threads, and is written and read by all of them (data race)

To fix this, OpenMP provides the clause `reduction(op:var)`, where `op` is any binary operator and `var` is a scalar variable

The `reduction` clause works for any structure of the form

```
total = 0
do i=1, n
  total = total op f(i)
end do
```

where `f(i)` is some function or an array, that does not reference the variable `total`

`op` is a binary operator of the class +, -, *, `MAX`, `MIN`, etc (the last two ones in Fortran)

# Reduction of loops

## With the reduction clause

```
  ! loop_reduce_good.f08
program loop
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: num_loops = 100000000

  integer i
  real(kind=dp) total

  total = 0.0

  !$omp parallel

  !$omp do reduction(+:total)

  do i = 1, num_loops
     total = total + exp(-real(i, dp))*cos(real(i, dp))
  enddo

  !$omp end do

  !$omp end parallel

  print *, "Total =", total

  ! correct value total = 8.5972572262175737E-002
end program loop
```

# Reduction of loops

## With the `reduction` clause

```
  ! loop_reduce_good.f08
program loop
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: num_loops = 100000000

  integer i
  real(kind=dp) total

  total = 0.0

  !$omp parallel

  !$omp do reduction(+:total)

  do i = 1, num_loops
     total = total + exp(-real(i, dp))*cos(real(i, dp))
  enddo

  !$omp end do

  !$omp end parallel

  print *, "Total =", total

  ! correct value total = 8.5972572262175737E-002
end program loop
```

```
bash-3.2$ gfortran loop_reduce_good.f08
bash-3.2$ ./a.out
 Total =   8.5972572262175737E-002
bash-3.2$
```

# Nested loops

## Many practical applications involve nested loops: Matrix operations

```
  ! matrix_serial

program matrix_serial
  implicit none
  integer, parameter :: dp       = kind(0.0d0)
  integer, parameter :: n        = 10000
  integer :: i, j
  real(kind=dp), dimension(:,:), allocatable :: a, b
  real(kind=dp) :: total
  allocate(a(n, n), b(n, n))

  total = 0.0
  do j = 1, n
     do i = 1, n
        a(i,j) = sin(real(i, dp))*sin(real(j, dp))
        b(i, j) = cos(real(i, dp))*cos(real(j, dp))
     end do
  end do
  do j = 1, n
     do i = 1, n
        total = total + a(i,j) * b(i, j)
     end do
  end do
  print *, total

end program matrix_serial
```

# Nested loops

## Many practical applications involve nested loops: Matrix operations

```
  ! matrix_serial

program matrix_serial
  implicit none
  integer, parameter :: dp       = kind(0.0d0)
  integer, parameter :: n        = 10000
  integer :: i, j
  real(kind=dp), dimension(:,:), allocatable :: a, b
  real(kind=dp) :: total
  allocate(a(n, n), b(n, n))

  total = 0.0
  do j = 1, n
     do i = 1, n
        a(i,j) = sin(real(i, dp))*sin(real(j, dp))
        b(i, j) = cos(real(i, dp))*cos(real(j, dp))
     end do
  end do
  do j = 1, n
     do i = 1, n
        total = total + a(i,j) * b(i, j)
     end do
  end do
  print *, total

end program matrix_serial
```

```
bash-5.1$ gfortran matrix_serial.f08
bash-5.1$ time ./a.out
   3.0793920155282414E-002

real        0m4.159s
user        0m3.866s
sys         0m0.285s
```

# Parallelizing nested loops

```
  ! matrix_parallel_1
program matrix_parallel_1
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp       = kind(0.0d0)
  integer, parameter :: n        = 10000
  integer :: i, j
  real(kind=dp), dimension(:,:), allocatable :: a, b
  real(kind=dp) :: total
  allocate(a(n, n), b(n, n))

  total = 0.0
  !$omp parallel
  !$omp do schedule(static)
  do j = 1, n
     do i = 1, n
        a(i,j) = sin(real(i, dp))*sin(real(j, dp))
        b(i, j) = cos(real(i, dp))*cos(real(j, dp))
     end do
  end do
  !$omp end do
  !$omp do schedule(static) reduction(+:total)
  do j = 1, n
     do i = 1, n
        total = total + a(i,j) * b(i, j)
     end do
  end do
  !$omp end do
  !$omp end parallel
  print *, total


end program matrix_parallel_1
```

# Parallelizing nested loops

```
  ! matrix_parallel_1
program matrix_parallel_1
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp       = kind(0.0d0)
  integer, parameter :: n        = 10000
  integer :: i, j
  real(kind=dp), dimension(:,:), allocatable :: a, b
  real(kind=dp) :: total
  allocate(a(n, n), b(n, n))

  total = 0.0
  !$omp parallel
  !$omp do schedule(static)
  do j = 1, n
     do i = 1, n
        a(i,j) = sin(real(i, dp))*sin(real(j, dp))
        b(i, j) = cos(real(i, dp))*cos(real(j, dp))
     end do
  end do
  !$omp end do
  !$omp do schedule(static) reduction(+:total)
  do j = 1, n
     do i = 1, n
        total = total + a(i,j) * b(i, j)
     end do
  end do
  !$omp end do
  !$omp end parallel
  print *, total


end program matrix_parallel_1
```

```
bash-5.1$ gfortran -fopenmp matrix_parallel_1.f
bash-5.1$ time ./a.out
   3.0793920155270257E-002

real        0m0.817s
user        0m7.832s
sys         0m0.523s
```

# Parallelizing nested loops

```
  ! matrix_parallel_1
program matrix_parallel_1
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp       = kind(0.0d0)
  integer, parameter :: n        = 10000
  integer :: i, j
  real(kind=dp), dimension(:,:), allocatable :: a, b
  real(kind=dp) :: total
  allocate(a(n, n), b(n, n))

  total = 0.0
  !$omp parallel
  !$omp do schedule(static)
  do j = 1, n
     do i = 1, n
        a(i,j) = sin(real(i, dp))*sin(real(j, dp))
        b(i, j) = cos(real(i, dp))*cos(real(j, dp))
     end do
  end do
  !$omp end do
  !$omp do schedule(static) reduction(+:total)
  do j = 1, n
     do i = 1, n
        total = total + a(i,j) * b(i, j)
     end do
  end do
  !$omp end do
  !$omp end parallel
  print *, total


end program matrix_parallel_1
```

```
bash-5.1$ gfortran -fopenmp matrix_parallel_1.f
bash-5.1$ time ./a.out
   3.0793920155270257E-002

real        0m0.817s
user        0m7.832s
sys         0m0.523s
```

With a simple *!$omp do* only the first loop is parallelized, the second look is executed sequentially by the thread in charge of the corresponding chunk of the first loop

# Parallelizing nested loops

We can parallelize more than one level of a set of nested loops with the clause `collapse(num_loops)`

It however can only be applied to fully nested loops, of the form

```
do j = 1, n
    do i = 1, n

    ! stuff here
```

with no instructions between the declaration of the first and the second loop

# Parallelizing nested loops

```
  ! matrix_parallel_2
program matrix_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp       = kind(0.0d0)
  integer, parameter :: n        = 10000
  integer :: i, j
  real(kind=dp), dimension(:,:), allocatable :: a, b
  real(kind=dp) :: total
  allocate(a(n, n), b(n, n))

  total = 0.0
  !$omp parallel
  !$omp do schedule(static) collapse(2)
  do j = 1, n
     do i = 1, n
        a(i,j) = sin(real(i, dp))*sin(real(j, dp))
        b(i, j) = cos(real(i, dp))*cos(real(j, dp))
     end do
  end do
  !$omp end do
  !$omp do schedule(static) reduction(+:total) collapse(2)
  do j = 1, n
     do i = 1, n
        total = total + a(i,j) * b(i, j)
     end do
  end do
  !$omp end do
  !$omp end parallel
  print *, total

end program matrix_parallel_2
```

# Parallelizing nested loops

```
  ! matrix_parallel_2
program matrix_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp       = kind(0.0d0)
  integer, parameter :: n        = 10000
  integer :: i, j
  real(kind=dp), dimension(:,:), allocatable :: a, b
  real(kind=dp) :: total
  allocate(a(n, n), b(n, n))

  total = 0.0
  !$omp parallel
  !$omp do schedule(static) collapse(2)
  do j = 1, n
     do i = 1, n
        a(i,j) = sin(real(i, dp))*sin(real(j, dp))
        b(i, j) = cos(real(i, dp))*cos(real(j, dp))
     end do
  end do
  !$omp end do
  !$omp do schedule(static) reduction(+:total) collapse(2)
  do j = 1, n
     do i = 1, n
        total = total + a(i,j) * b(i, j)
     end do
  end do
  !$omp end do
  !$omp end parallel
  print *, total

end program matrix_parallel_2
```

```
bash-5.1$ gfortran -fopenmp matrix_parallel_2.f
bash-5.1$ time ./a.out
   3.0793920155192382E-002

real        0m0.801s
user        0m7.855s
sys         0m0.521s
```

# Parallelizing nested loops

```
  ! matrix_parallel_2
program matrix_parallel_2
  use, intrinsic :: omp_lib
  implicit none
  integer, parameter :: dp       = kind(0.0d0)
  integer, parameter :: n        = 10000
  integer :: i, j
  real(kind=dp), dimension(:,:), allocatable :: a, b
  real(kind=dp) :: total
  allocate(a(n, n), b(n, n))

  total = 0.0
  !$omp parallel
  !$omp do schedule(static) collapse(2)
  do j = 1, n
     do i = 1, n
        a(i,j) = sin(real(i, dp))*sin(real(j, dp))
        b(i, j) = cos(real(i, dp))*cos(real(j, dp))
     end do
  end do
  !$omp end do
  !$omp do schedule(static) reduction(+:total) collapse(2)
  do j = 1, n
     do i = 1, n
        total = total + a(i,j) * b(i, j)
     end do
  end do
  !$omp end do
  !$omp end parallel
  print *, total

end program matrix_parallel_2
```

```
bash-5.1$ gfortran -fopenmp matrix_parallel_2.f
bash-5.1$ time ./a.out
   3.0793920155192382E-002

real        0m0.801s
user        0m7.855s
sys         0m0.521s
```

The two loops are unrolled, and considered as a single loop on `(i, j)`, that is fully parallelized

# Troubles with the loops

When we parallelize loops, we must be careful that the loops can be parallelized. This essentially means that the order in which the iterations are performed does not affect the final result, since we do not know in principle the order that OpenMP will follow

Example: sum of arrays

```
do i=1, n
  a(i) = b(i) + c(i)
end do
```

it has no problems. It is easy to see that each iteration of the loop is independent of the others

Example: iterative calculation

```
a(1) = b(1)
do i=2, n
  a(i) = a(i-1) + b(i)
end do
```

will not work, since now the order matters. Not everything can be parallelized straightforwardly

# Troubles with loops

The rule of thumb is that loops in which iterations are independent can be easily parallelized

- Loops that give the same result if the order of iteration is inverted

More complex loops can be still parallelized, but in a more complex way, since they have to be rewritten with some parts that must be computed sequentially in order to synchronize threads

# Tips and tricks for OpenMP

- Be careful with the sentinel `!$omp`. A typo like `!omp` will go unnoticed (it is a comment!)
- Write code that compiles also in serial. Use the conditional compilation sentinel `!$`, without `omp`
- The typical overhead of executing a parallel region is of some tens of microseconds. Parallelize only regions that take much more to execute sequentially
- Always check if a loop can be parallelized: Execute it in reverse order
- The schedule `static` is not always enforced as the default in all implementations. Get used to impose it
- When using `dynamic` schedule, the optimal chunk size can depend on the number of threads. Check it
- `private` variables are not initialized at the beginning of a parallel section

# Tips and tricks for OpenMP

- When the body of a loop is very large, refactor it in terms of a procedure and implement parallelization in the procedure
- Large `private` data structures can run out of the stack memory space (fast allocation memory) assigned to each thread. You can increase it with the environment variable `OMP_STACKSIZE`
- In a `reduction` ((total op expr), `total` must be a scalar variable, and `expr` a scalar expression that does not reference `total`
- You can measure execution time with the function `omp_get_wtime()`
- The default behavior of variables for parallel regions is usually `shared`. Be careful. You can impose in a parallel region a default behavior of private with the clause `default(none)`, and then declare the state of each shared variable as needed.

# A final warning about OpenMP

1. It is not necessarily implemented the same way in different compilers
2. It is designed for machines that share the memory space (no clusters, MPI)
3. It is not guaranteed to use efficiently the shared memory
4. Does not check for data dependencies, data races, etc
5. Does not check the code
6. It does not have any automatic implementation (such as we can find in vectorization)
7. It is not designed to work exactly the same when the same code is executed sequentially (up to you)

# Exercise

1. Program one of the most time-consuming algorithms in fortran: matrix multiplication
2. Consider the multiplication of two matrices of size $n \times n$, and make $n$ large, say $n = 2000$ or 3000
3. Run the code in serial and in parallel, and check the speedup

# Solution

## Execution times in a 12-core (24 logical cores) machine:

```
bash-5.1$ gfortran  matrix_multiplication.f08 ; time a.out
real        5m51.486s

bash-5.1$ gfortran -fopenmp  matrix_multiplication.f08
bash-5.1$ export OMP_NUM_THREADS=2 ; time a.out
real        4m41.682s

bash-5.1$ export OMP_NUM_THREADS=4 ; time a.out
real        2m20.513s

bash-5.1$ export OMP_NUM_THREADS=8 ; time a.out
real        1m11.031s

bash-5.1$ export OMP_NUM_THREADS=16 ; time a.out
real        0m49.489s

bash-5.1$ export OMP_NUM_THREADS=24 ; time a.out
real        0m42.437s
```

## Speedup of up to a factor 8

# Solution

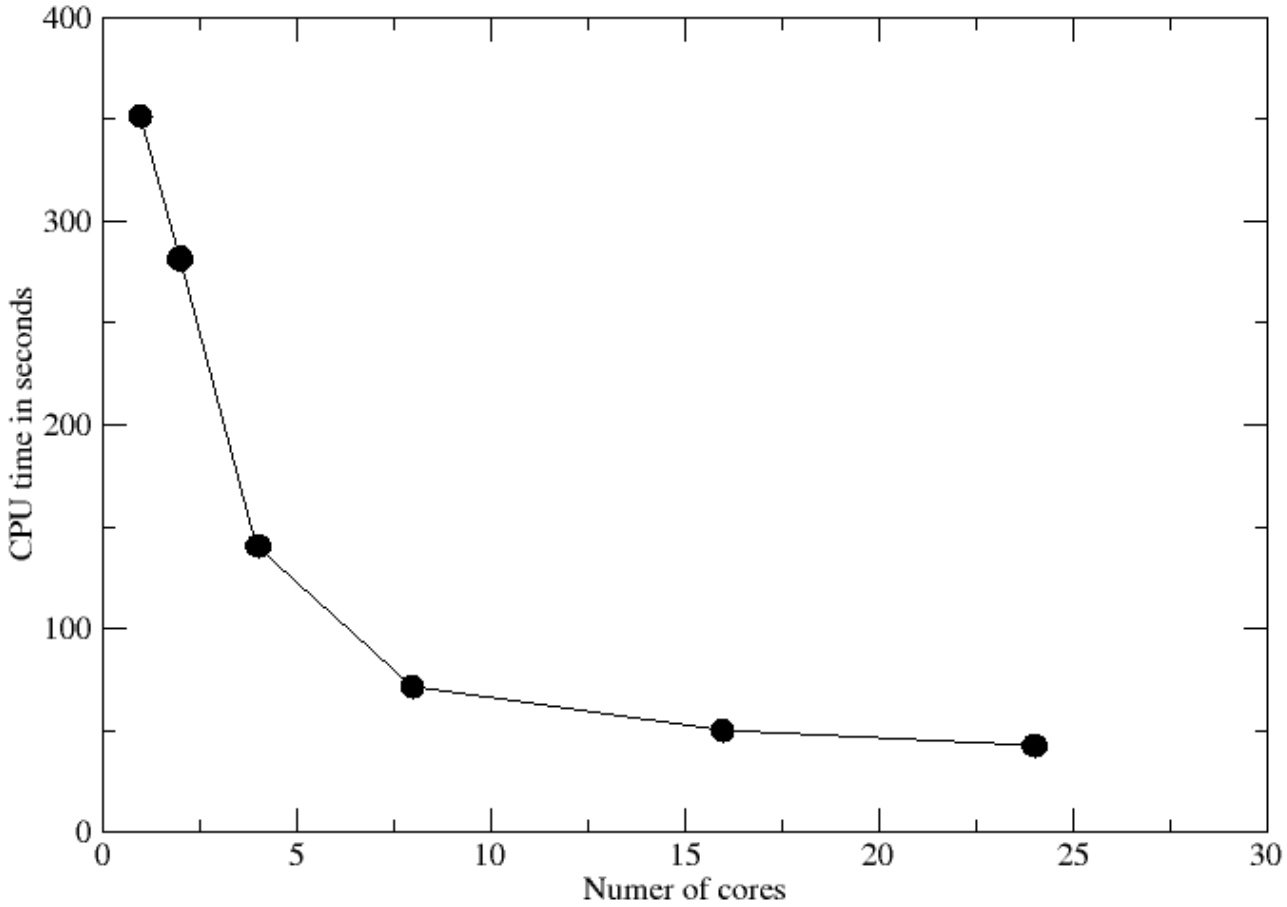
Execution times in a 12-core (24 logical cores) machine:

```
bash-5.1$ gfortran  matrix_multiplication.f08 ; time a.out
real        5m51.486s

bash-5.1$ gfortran -fopenmp  matrix_multiplication.f08
bash-5.1$ export OMP_NUM_THREADS=2 ; time a.out
real        4m41.682s

bash-5.1$ export OMP_NUM_THREADS=4 ; time a.out
real        2m20.513s

bash-5.1$ export OMP_NUM_THREADS=8 ; time a.out
real        1m11.031s

bash-5.1$ export OMP_NUM_THREADS=16 ; time a.out
real        0m49.489s

bash-5.1$ export OMP_NUM_THREADS=24 ; time a.out
real        0m42.437s
```

Speedup of up to a factor 8

With the flag -O3, execution time can be reduced even further: 0m25.721s

# Speed up


400
300
200
100
0
CPU time in seconds
0
5
10
15
20
25
30
Numer of cores