Introduction to OpenMP

Fabio Pitari, Cineca

January/2022

Univerza v Ljubljani

CINECA

IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Co-funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Introduction
- OpenMP vs MPI
- OpenMP execution model
- OpenMP programming model
- OpenMP memory model

Worksharing constructs
- Worksharing constructs rules
- for/do loop
- sections
- single
- workshare

How to avoid data races
- Critical construct
- Reduction clause
- Barrier construct
- Atomic construct

SIMD
- Basic concepts
- Autovectorization
- OpenMP simd directive
- Vectorization of functions

Tasks
- Basic concepts
- Data scopes and reduction
- Tasks synchronization
- Taskloops

Conclusions

Introduction

Disadvantages of MPI

Fabio Pitari, Cineca

⇒ The data to be shared must be exchanged with explicit inter-process communications

! It is in charge to the programmer to design and implement the data exchange between process (taking care of work balance)

- You can not adopt a strategy of incremental parallelization
  - ⇒ The communication structure of the entire program has to be implemented
- It is difficult to maintain a single version of the code for the serial and MPI program
  - ⇒ Additional variables are needed
  - ⇒ Need to add a lot of boilerplate code

What is OpenMP?

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- De-facto standard Application Program Interface (API) to write shared memory parallel applications in C, C++ and Fortran
- Consists of **compilers directives**, **run-time routines** and **environment variables**
- "Open specifications for Multi Processing" maintained by the OpenMP Architecture Review Board (http://www.openmp.org)

Founding concepts

The "workers" who do the work in parallel (thread) "cooperate" through shared memory

- Memory accesses instead of explicit messages
- "local" model parallelization of the serial code
- It allows an incremental parallelization

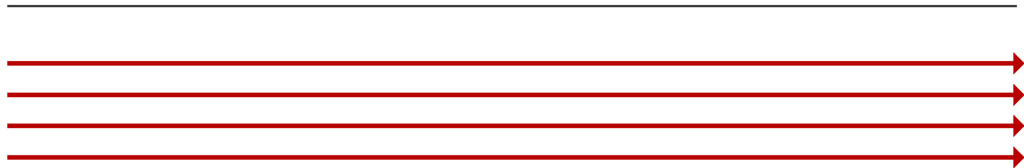
Fabio Pitari, Cineca

- Born to satisfy the need of unification of proprietary solutions
- **The past**
  - October 1997 - Fortran version 1
  - October 1998 - C/C++ version 1
  - November 1999 - Fortran version 1.1 (interpretations)
  - November 2000 - Fortran version 2
  - March 2002 - C/C++ version 2
  - May 2005 - combined C/C++ and Fortran version 2
  - May 2008 - version 3.0
  - July 2011 - version 3.1

- **The present**
  - July 2013 - version 4.0
  - November 2015 - version 4.5
  - November 2018 - version 5.0
  - November 2020 - version 5.1
  - November 2021 - version 5.2

MPI Execution model

Fabio Pitari, Cineca

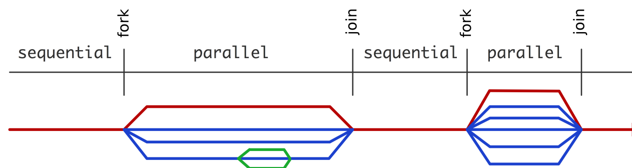


MPI model

- Everything lies in a huge parallel region where the same code is executed by all the ranks
- Comunication among ranks has to be explicitely built when needed

OpenMP Execution model

Fabio Pitari, Cineca



Fork-Join model

Fork At the beginning of a parallel region the master thread creates a team of threads composed by itself and by a set of other threads

Join At the end of the parallel region the thread team ends the execution and only the master thread continues the execution of the (serial) program

OpenMP: Hello world

Fabio Pitari, Cineca

A set of instructions can be executed on the whole set of threads using the *parallel* directive

C/C++

```
#include <stdio.h>
int main () {
  /* Serial part */
  {
    printf("Hello world\n");
  }
  /* Serial part */
  return 0;
}
```

OpenMP: Hello world

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

A set of instructions can be executed on the whole set of threads using the *parallel* directive

C/C++

```
#include <stdio.h>
int main () {
  /* Serial part */
#pragma omp parallel
  {
    printf("Hello world\n");
  }
  /* Serial part */
  return 0;
}
```

A set of instructions can be executed on the whole set of threads using the *parallel* directive

Fortran

```
PROGRAM HELLO
! Serial code
  Print *, "Hello World!!!"
! Serial code
END PROGRAM HELLO
```

A set of instructions can be executed on the whole set of threads using the *parallel* directive

Fortran

```
PROGRAM HELLO
! Serial code
!$OMP PARALLEL
  Print *, "Hello World!!!"
!$OMP END PARALLEL
! Serial code
END PROGRAM HELLO
```

- **Compiler directives**

C/C++

```
#pragma omp <directive> [clause [clause] ...]
```

Fortran

```
!$omp <directive> [clause [clause]...]
```

`directive` mark the block of code that should be made parallel

`clause` add information to the directives

⇒ Variables handling and scoping (shared, private, default)

⇒ Execution control (how many threads, work distribution...)

Note

GNU (gcc, g++, gfortran) `-fopenmp`

Intel (icc, icpc, ifort) `-qopenmp`

- **Compiler directives**
- **Environment variables**

`OMP_NUM_THREADS`	`size`	set the number of threads
`OMP_DYNAMIC`	`true|false`	set the number of threads automatically
`OMP_PLACES`	`cores|threads|sockets`	set the place where to allocate threads
`OMP_PROC_BIND`	`true|false|close|spread`	bound threads to such place
`OMP_NESTED`	`true|false`	allows nested parallelism

Other commonly used variables will be clearer later (listed here for reference):

`OMP_STACKSIZE`	`size [B|K|M|G]`	size of the stack for threads
`OMP_SCHEDULE`	`schedule[,chunk]`	iteration scheduling scheme

Note

```
$ export VARIABLE_NAME=value
```

Note

Intel compilers add some additional variables, which takes precedence over the OpenMP standard variables if set both. They enable specific run-time libraries included in the Intel compilers. For instance:

`KMP_DYNAMIC`	`load_balance|thread_limit`	replace OMP_DYNAMIC
`KMP_AFFINITY`	`compact|scatter|balanced|verbose`	replace OMP_PROC_BIND
`KMP_STACKSIZE`	`size [B|K|M|G]`	replace OMP_STACKSIZE

and many others you can find in the Intel documentation. In particular you might want to refine affinity starting from KMP_AFFINITY variable (see this link).

Affinity is relevant

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- The way in which cores are mapped by threads can be relevant with respect to performances
- Switching different values of the variables allows to easily test different bindings

Note

Affinity might be relevant also for initializations (*first touch issue*)

- Memory addresses refers to the whole physical memory, disregarding the socket on which the corresponding physical memory is attached

⇒ This means that a serial initialization tries to allocate everything on the physical memory of the socket of the master thread

- Mapping is effectively executed when data are written to such addresses

⇒ This means that in a following parallel region the threads on the other socket will need to access to a memory not attached to that socket (bottleneck)

```
// serial initialization
for (i=0; i<N; i++)
  x[i] = 0;

// parallel access
#pragma omp parallel
for (i=0; i<N; i++)
   // something with x[i]
```

Programming model

Fabio Pitari, Cineca

- **Compiler directives**
- **Environment variables**
- **Run-time library**

`omp_get_thread_num()`	get thread ID
`omp_get_num_threads()`	get number of threads in the team for threads
`omp_set_num_threads(int n)`	set the number of threads
`omp_get_wtime()`	returns elapsed wallclock time

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Note

```
#include <omp.h>  // C++

USE omp_lib       ! Fortran
```

C/C++

```
#ifdef _OPENMP
  printf("OpenMP support:%d",_OPENMP);
#else
  printf("Serial execution.");
#endif
```

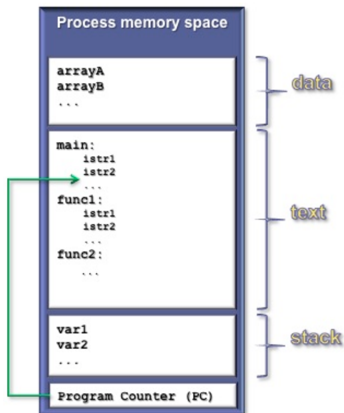
Fortran

```
#ifdef _OPENMP
  print *,"OpenMP support:", _OPENMP
#else
  print *,"Serial execution."
#endif
```

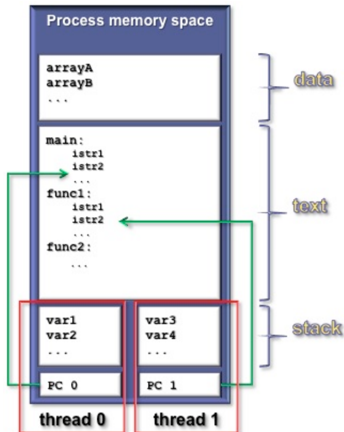
Note The macro `_OPENMP` has the value `yyyymm`, which contains the release date of the OpenMP version currently being used

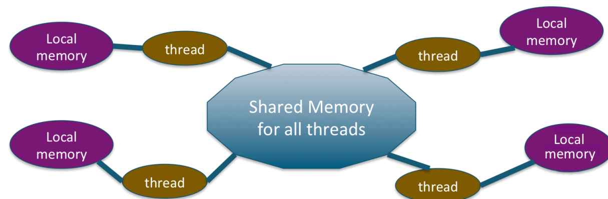
Fabio Pitari, Cineca

- A process is an instance of a computer program
- Some information included in a process are:
  - Text
    - ⇒ Machine code
  - Data
    - ⇒ Global variables
  - Stack
    - ⇒ Local variables
  - Program counter (PC)
    - ⇒ A pointer to the instruction to be executed



- The process contains several concurrent execution flows (threads)
  - Each thread has its own program counter (PC)
  - Each thread has its own private stack (variables local to the thread)
  - The instructions executed by a thread can access:
    - the process global memory (data)
    - the thread local stack



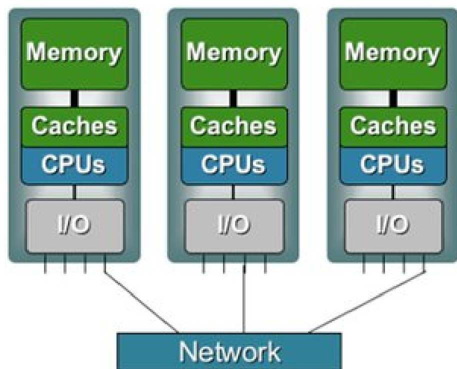


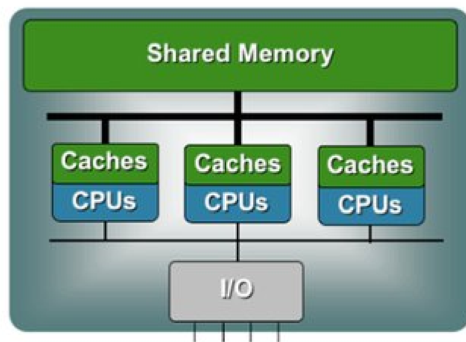
Shared-Local model

- each thread is allowed to have a temporary view of the shared memory
- each thread has access to a thread-private memory
- two kinds of data-sharing attributes: private and shared

Distributed and shared memory

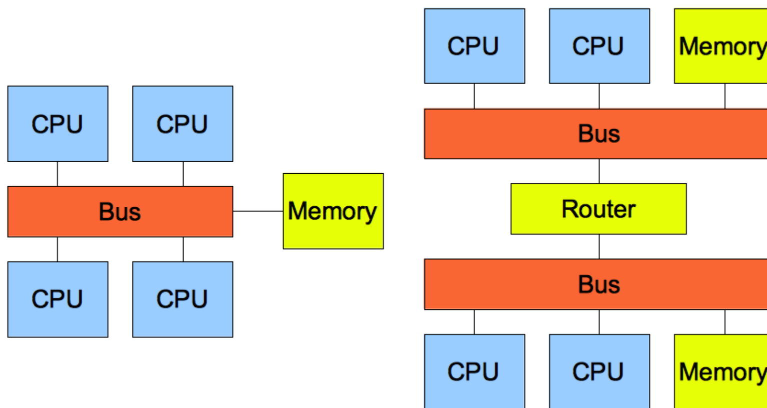
Fabio Pitari, Cineca





UMA and NUMA systems

Fabio Pitari, Cineca



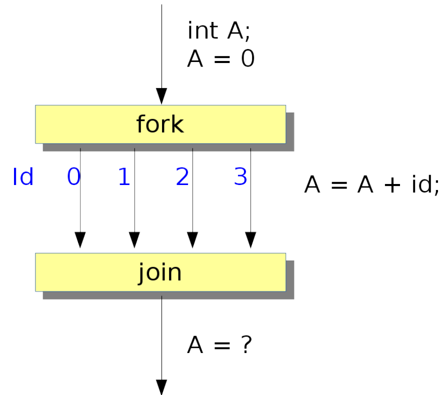
Defined or undefined?

Fabio Pitari, Cineca

What's the result?

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

void main(){
    int a;
    a = 0;
    #pragma omp parallel
    {
        // omp_get_thread_num
        // returns the id of the thread
        a = a + omp_get_thread_num();
    }
    printf("%d\n", a);
}
```



Defined or undefined?

Fabio Pitari, Cineca

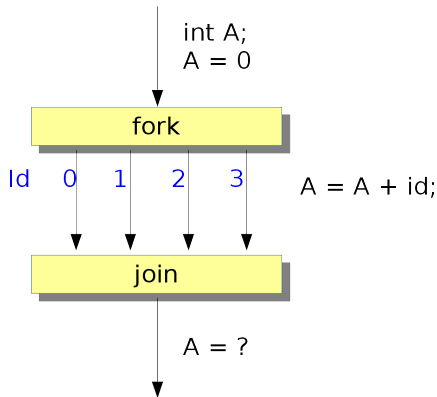
What's the result?

```
program race_condition
use omp_lib

integer :: a
a = 0

!$omp parallel
a = a + omp_get_thread_num()
!$omp end parallel

write(*,*) a
end program race_condition
```



- A **race condition** (or data race) is when two or more threads access the same memory location:
  - asyncronously and,
  - without holding any common exclusive locks and,
  - at least one of the accesses is a **write/store**
- In this case the resulting values are **undefined**

Critical construct

Fabio Pitari, Cineca

The critical construct is a possible solution to **data races**:

- The block of code inside a critical construct is executed by only one thread at a time
- It locks the associated region

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

void main(){
    int a;
    a = 0;
    #pragma omp parallel
    {
        // omp_get_thread_num
        // returns the id of the thread
        #pragma omp critical
        a = a + omp_get_thread_num();
    }
    printf("%d\n", a);
}
```

Critical construct

Fabio Pitari, Cineca

The critical construct is a possible solution to **data races**:

- The block of code inside a critical construct is executed by only one thread at a time
- It locks the associated region

```
program race_condition
use omp_lib

integer :: a
a = 0

!$omp parallel
!$omp critical
a = a + omp_get_thread_num()
!$omp end critical
!$omp end parallel

write(*,*) a
end program race_condition
```

Exercise 1

```
#include <stdio.h>
#ifdef _OPENMP
#include<omp.h>
#endif

int main(int argc, char* argv[])
{
#ifdef _OPENMP
int iam;
#pragma omp parallel
    { /* the parallel block starts here */
        iam=omp_get_thread_num();
#pragma omp critical
        printf("Hello from %d\n",iam);
    } /* the parallel block ends here */

#else
        printf("Hello, this is a serial program.\n");
#endif

    return 0;
}
```

- Compile
- Run
- Experiment with the `OMP_NUM_THREADS` variable.

Did you obtain the behaviour you expected?

Exercise 1

```
Program Hello_from_Threads
#ifdef _OPENMP
   use omp_lib
#endif
   implicit none

   integer :: iam

#ifdef _OPENMP
!$omp parallel
   iam=omp_get_thread_num()

!$omp critical
   write(*,*) 'Hello from', iam
!$omp end critical

!$omp end parallel
#else
   write(*,*) 'Hello, this is a serial program'
#endif
end program Hello_from_Threads
```

- Compile
- Run
- Experiment with the `OMP_NUM_THREADS` variable.

Did you obtain the behaviour you expected?

Shared and private variables

Fabio Pitari, Cineca

Inside a parallel region the scope of a variable can be `shared` or `private`.

Shared

There is only one instance of the variable.

- Directive: `shared(a,b,c,…)`
  - The variable is accessible by all threads in the team
  - Threads can read and write the variable simultaneously

Shared and private variables

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Inside a parallel region the scope of a variable can be `shared` or `private`.

Private

Each thread has a copy of the variable. The variable is accessible only by the owner thread.

- Directive: `private(a,b,c,…)`
  - Values are undefined on entry and exit
- Directive: `firstprivate(a,b,c,…)`
  - variables are initialized with the value that the original object had before entering the parallel construct
- Directive: `lastprivate(a,b,c,…)`
  - the thread that executes the sequentially last iteration or section updates the value of the variable

Fabio Pitari, Cineca

The default behaviours might be confusing:

Shared by default:

- variables allocated outside the parallel region
- assumed size arrays
- variables with save attributes (Fortran)

Private by default:

- variables allocated inside the parallel region
- variables with automatic storage duration
- inner loop indexes in `loop` directive when using Fortran

Best practice: Nullify the default with the clause `default(none)`

```
int a=0;
float b=1.5;
int c=3;
#pragma omp parallel default(none) shared(a) private(b) firstprivate(c)
{
    // each thread can access to "a"
    // each thread has its own copy of "b", with an undefined value
    // each thread has its own copy of "c", with c=3
}
```

Solution of exercise 1

```
#include <stdio.h>
#ifdef _OPENMP
#include<omp.h>
#endif

int main(int argc, char* argv[])
{
#ifdef _OPENMP
int iam;
#pragma omp parallel private(iam)
    { /* the parallel block starts here */
        iam=omp_get_thread_num();
#pragma omp critical
        printf("Hello from %d\n",iam);
    } /* the parallel block ends here */

#else
        printf("Hello, this is a serial program.\n");
#endif

    return 0;
}
```

Solution of exercise 1

```
Program Hello_from_Threads
#ifdef _OPENMP
   use omp_lib
#endif
   implicit none

   integer :: iam

#ifdef _OPENMP
!$omp parallel private(iam)
   iam=omp_get_thread_num()

!$omp critical
   write(*,*) 'Hello from', iam
!$omp end critical

!$omp end parallel
#else
   write(*,*) 'Hello, this is a serial program'
#endif
end program Hello_from_Threads
```

- The *parallel* directive has an implicit barrier at its end, i.e. all the threads have to wait that every other thread complete its code block;
- *critical* doesn't have an implicit barrier at the end; it just executes the threads one by one;
- In order to build an OpenMP parallelization it is enough to use:
  - parallel directive;
  - critical directive;
  - omp_get_thread_num() function;
  - omp_get_num_threads() function;

However, this might imply to readapt the code (just like in MPI), while the aim of the OpenMP approach is to keep the parallelization as simple as possible by sharing the work among the threads in an automatic way.
In order to automatize such distribution you can use *worksharing directives*.

Worksharing constructs

Worksharing constructs: rules

Fabio Pitari, Cineca

Distribute the execution of the associated region

Rules

1. A worksharing region has **no barrier** on entry
2. An **implied barrier** exists at the end, unless `nowait` is specified
3. Each region **must** be encountered by all threads or none
   ⇒ Every thread must encounter the **same sequence** of worksharing regions and barrier regions

Worksharing constructs: rules

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Constructs

- **for/do loop**
- **sections**
- **single**
- **workshare**

Loop construct: syntax

Fabio Pitari, Cineca

C/C++

```
#pragma omp for [clause[[,] clause] ... ]
  for-loops
```

Fortran

```
!$omp do [clause[[,] clause] ... ]
  do-loops
!$omp end do [nowait]
```

Fabio Pitari, Cineca

Useful Clauses

`schedule` can be used to specify how iterations are divided into chunks

`collapse` can be used to specify how many loops are parallelized

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Rules

1. The iterations of the loop are **distributed** over the threads that already exist in the team (`scheduling` clause can manage their distribution)
2. The **iteration variable** in the `for` loop
   - if shared, is **implicitly** made private
   - must **not be modified** during the execution of the loop
   - has an **unspecified value** after the loop
3. Only loops with **canonical forms** are allowed, i.e.:
   - the **iteration count** must be known before executing the loops
   - the **incremental expression** must be addition or subtraction expression.
4. By default (i.e. without the `collapse` clause) only the external loop is parallelized, whereas the internal ones are executed sequentially for each thread. The indexes of the internal loops, if allocated outside the parallel region:
   - in C/C++ are **shared** by default (as usual)
   - in Fortran are made **private** by default

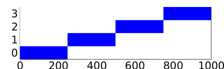
C/C++

```
#pragma omp for schedule(kind[, chunk_size])
  for-loops
```

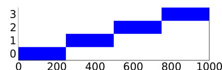
Fortran

```
!$omp do schedule(kind[, chunk_size])
  do-loops
[!$omp end do [nowait] ]
```

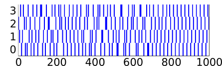
1. **static**
   - if no `chunk_size` is specified the iterations space is divided in chunks of equal size and one chunk per thread
   - if `chunk_size` is specified, chunks are assigned to the threads in a round-robin fashion



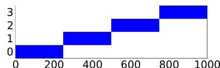
1. **static**
    - if no `chunk_size` is specified the iterations space is divided in chunks of equal size and one chunk per thread
    - if `chunk_size` is specified, chunks are assigned to the threads in a round-robin fashion



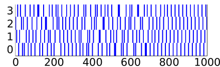
2. **dynamic**
    - iterations are divided into chunks of size `chunk_size` with default value 1
    - the chunks are assigned to the threads as they request them



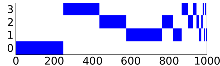
1. **static**
   - if no `chunk_size` is specified the iterations space is divided in chunks of equal size and one chunk per thread
   - if `chunk_size` is specified, chunks are assigned to the threads in a round-robin fashion



2. **dynamic**
   - iterations are divided into chunks of size `chunk_size` with default value 1
   - the chunks are assigned to the threads as they request them



3. **guided**
   - iterations are divided into chunks of decreasing size, where `chunk_size` controls the minimum size
   - the chunks are assigned to the threads as they request them



Example

```
#pragma omp parallel
{
#pragma omp for collapse(2)
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < m; j++) {
      A[i][j] = i*m + j;
    }
  }
}
```

- The collapse clause indicates how many loops should be collapsed (including the outer loop)
- Allows parallelization of perfectly **nested rectangular loops**
- Compiler forms a **single loop** (e.g. of length nxm) and then parallelizes it
- Useful if n < number of threads, so parallelizing the outer loop makes balancing the load difficult.

Loop construct: tips and tricks

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<n; i++){
    // do something
  }
}
```

is equivalent to:

```
#pragma omp parallel for
  for (i=0; i<n; i++){
    // do something
  }
```

C/C++

```
#pragma omp sections [clause[[,] clause]...]
{
  #pragma omp section
    structured-block
  #pragma omp section
    structured-block
  ...
}
```

1. `sections` is a non-iterative worksharing construct
   - it contains a set of `structured-blocks`
   - each `section` is executed **once** by **only one** of the threads
2. Scheduling of the sections is **implementation defined**
3. There is an implied barrier at the end of the construct

Sections construct

Fabio Pitari, Cineca

Fortran

```
!$omp sections [clause[[,] clause]...]
  !$omp section
    structured-block
  !$omp section
    structured-block
  ...
!$omp end sections [nowait]
```

1. `sections` is a non-iterative worksharing construct
   - it contains a set of `structured-blocks`
   - each `section` is executed **once** by **only one** of the threads
2. Scheduling of the sections is **implementation defined**
3. There is an implied barrier at the end of the construct

Single construct

Fabio Pitari, Cineca

C/C++

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

Fortran

```
!$omp single [clause[[,] clause] ... ]
    structured-block
!$omp end single [nowait]
```

1. The associated structured block is executed **by only one thread**
2. The other threads wait at an **implicit barrier** (unless a `nowait` clause is specified)
3. The method of choosing a thread is **implementation defined**

Fabio Pitari, Cineca

Note: The `master` construct is similar but specifies a structured block:

- that is **executed by the master** thread
- with **no implied barrier** on entry or exit (it isn't a worksharing construct)

C/C++

```
#pragma omp master
  structured-block
```

Fortran

```
!$omp master
  structured-block
!$omp end master
```

The advantage w.r.t. a `single` construct with `nowait` clause is that:

- not being a worksharing construct can be nested inside one of them;
- allows some operations in some specific context where only the master is allowed to perform some operations (e.g. MPI communications when `MPI_THREAD_FUNNELED` is set)

Note: only available in Fortran

Fortran

```
!$omp workshare
  structured-block
!$omp end workshare [nowait]
```

Divides the lines of the code block into **units of work**, and each of them is then executed by a different thread. The allowed instructions are (mainly):

1. scalar assignments
2. array assignments (one element for thread)
3. `FORALL` statements
4. `WHERE` statements

When the code lines are not included in the cases above, they're treated as a single unit of work (and thus executed sequentially by one thread).

Exercise 2

Fabio Pitari, Cineca

The code performs a serial matrix multiplication

- Try to parallelize it with OpenMP, acting only on the most important loop
- Try also to add the timing functions before and after the loop and print the elapsed time.

Fabio Pitari, Cineca

C/C++

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

int main(int argc, char **argv)
{
   int n;
   int i, j, k;

   if(argc != 2) {
      fprintf(stderr,"Usage: %s  matrix size\n", argv[0]);
      exit(EXIT_FAILURE);
    }
```

Exercise 2

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

C/C++

```
n = atoi(argv[1]);
if ( n > 0) {
  printf("Matrix size is %d\n",n);
}
else {
  fprintf(stderr,"Error, matrix size is %d \n", n);
  exit(EXIT_FAILURE);
}

double (*a)[n] =  malloc(sizeof(double[n][n]));
double (*b)[n] =  malloc(sizeof(double[n][n]));
double (*c)[n] =  malloc(sizeof(double[n][n]));

if ( a == NULL || b == NULL || c == NULL) {
    fprintf(stderr, "Not enough memory!\n");
    exit(EXIT_FAILURE);
```

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

C/C++

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++) {
      a[i][j] = ((double)rand())/((double)RAND_MAX);
      b[i][j] = ((double)rand())/((double)RAND_MAX);
      c[i][j] = 0.0;
   }

for (i=0; i<n; i++)
  for (k=0; k<n; k++)
     for (j=0; j<n; j++)
       c[i][j] += a[i][k]*b[k][j];
```

Exercise 2

Fabio Pitari, Cineca

C/C++

```c
    //check a random element
    i = rand()%n;
    j = rand()%n;
    double d = 0.0;
    for (k=0; k<n; k++)
       d += a[i][k]*b[k][j];

    printf("Check on a random element: %18.9lE\n", fabs(d-c[i][j]));

    return 0;

}
```

Exercise 2

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Fortran

```
Program matrix_matrix_prod
   implicit none
   integer :: n
   real(kind(1.d0)), dimension(:,:), allocatable   :: a, b, c
   real(kind(1.d0)) :: d
   integer            :: i, j, k, ierr
   character(len=128) :: command
   character(len=80) :: arg
```

Fortran

```
call get_command_argument(0,command)
if (command_argument_count() /= 1) then
  write(0,*) 'Usage:', trim(command), '   matrix size'
  stop
else
  call get_command_argument(1,arg)
  read(arg,*) n
endif
if (n > 0 ) then
  write(*,*) 'Matrix size is ', n
else
  write(0,*) "Error, matrix size is ", n
endif
```

Exercise 2

Fabio Pitari, Cineca

Fortran

```fortran
allocate(a(n,n),b(n,n),c(n,n),stat=ierr)

if(ierr/=0) STOP 'a,b,c matrix allocation failed'

call random_number(a)
call random_number(b)
c = 0.d0
```

Exercise 2

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Fortran

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

Fortran

```
    call random_number(d)
    i = int( d*n+1)
    call random_number(d)
    j = int( d*n+1)
    d = 0.d0
    do k=1, n
          d = d + a(i,k)*b(k,j)
    end do

    write(*,*) "Check on a random element:" , abs(d-c(i,j))

end program matrix_matrix_prod
```

How to avoid data races

C/C++

```
#pragma omp critical [clause [clause] ...]
```

Fortran

```
!$omp critical [clause [clause]...]
```

- As previously shown, the block of code inside a critical construct is executed by only one thread at time
- This is clearly unefficient since it serializes the execution of the process
- It has to be considered an extreme solution, but less invasive possibilities has to be considered in the first instance, like the following ones

A reduction variable is used to accumulate a value from the different threads

```
double x[n];
double sum=0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<n; i++){
  sum+=x[i]
}
```

The reduction clause:

Reduction clause: some facts

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

The reduction clause:

- is valid both on `parallel` and `work-sharing` constructs

The reduction clause:

- is valid both on `parallel` and `work-sharing` constructs
- specifies an operator and one or more list items

C/C++ `+, *, -, &, |, &&, ||, max, min`

Fortran `+, *, -, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor`

The reduction clause:

- is valid both on `parallel` and `work-sharing` constructs
- specifies an operator and one or more list items

  C/C++ `+, *, -, &, |, &&, ||, max, min`

  Fortran `+, *, -, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor`
- The items that appears in a `reduction` clause must be shared
  - a **local copy** is created and initialized based on the reduction operation
  - **updates** occur on the local copy.
  - local copies are **reduced** into a single value and combined with the original global value.

Exercise 3

Fabio Pitari, Cineca

The code determines the value of π, by calculating an integral between 0 and 1. The integral is approximated as a sum of n intervals.

- Parallelize it with OpenMP
- Try also to solve the exercise without using the reduction clause

C/C++

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

#define PI25DT 3.141592653589793238462643
#define INTERVALS 100000000

int main(int argc, char **argv)
{
    long int i, intervals = INTERVALS;
    double x, dx, f, sum, pi;
    double time2;
    time_t time1 = clock();
    printf("Number of intervals: %ld\n", intervals);

    sum = 0.0;
    dx = 1.0 / (double) intervals;
```

C/C++

```
for (i = 1; i <= intervals; i++) {
    x = dx * ((double) (i - 0.5));
    f = 4.0 / (1.0 + x*x);
    sum = sum + f;
}
```

C/C++

```
    pi = dx*sum;

    time2 = (clock() - time1) / (double) CLOCKS_PER_SEC;

    printf("Computed PI %.24f\n", pi);
    printf("The true PI %.24f\n\n", PI25DT);
    printf("Elapsed time (s) = %.2lf\n", time2);

    return 0;
}
```

Fortran

```
program pi
    implicit none

    integer(selected_int_kind(18)) :: i
    integer(selected_int_kind(18)), parameter :: intervals=1e8

    real(kind(1.d0)) :: dx,sum,x
    real(kind(1.d0)) :: f,pi

    real(kind(1.d0)), parameter :: PI25DT = acos(-1.d0)
    real :: time1, time2

    call cpu_time(time1)

    print *, 'Number of intervals: ', intervals
    sum=0.d0
    dx=1.d0/intervals
```

Fabio Pitari, Cineca

Fortran

```fortran
do i=1,intervals
    x=dx*(i-0.5d0)
    f=4.d0/(1.d0+x*x)
    sum=sum+f
end do
```

Fortran

```
    pi=dx*sum

    call cpu_time(time2)

    PRINT '(a13,2x,f30.25)',' Computed PI =', pi
    PRINT '(a13,2x,f30.25)',' The True PI =', PI25DT
    PRINT *, ' '
    PRINT *, 'Elapsed time ', time2-time1 ,' s'

end program
```

Barrier construct

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

The `barrier` construct specifies an **explicit barrier** at the point at which the construct appears

Reminder: implicit barriers are assumed at the end of a worksharing construct, and can be removed via the `nowait` clause

Note: when not necessary, a barrier can cause slowdowns

C/C++

```
#pragma omp barrier
```

Fortran

```
!$omp barrier
```

Example: waiting for the master

```
int counter = 0;
#pragma omp parallel
{
#pragma omp master
  counter = 1;
#pragma omp barrier
  printf("%d\n", counter);
}
```

C/C++

```
#pragma omp atomic [read | write | update | capture]
  expression-stmt
```

Fortran

```
!$omp atomic [read | write | update | capture]
  expression-stmt
!$omp end atomic
```

Atomic construct

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

The `atomic` construct:

The `atomic` construct:

- Ensures a specific storage location to be **updated atomically**, i.e. does not expose it to multiple, simultaneous writing threads

The `atomic` construct:

- Ensures a specific storage location to be **updated atomically**, i.e. does not expose it to multiple, simultaneous writing threads
- Possible clauses are:

`read` forces an atomic read regardless of the machine word size
`write` forces an atomic write regardless of the machine word size
`update` forces an atomic update (default)
`capture` same as an update, but captures original or final value (e.g. allows a = b++ with both a and b atomically updated)

The `atomic` construct:

- Ensures a specific storage location to be **updated atomically**, i.e. does not expose it to multiple, simultaneous writing threads
- Possible clauses are:

`read`	forces an atomic read regardless of the machine word size
`write`	forces an atomic write regardless of the machine word size
`update`	forces an atomic update (default)
`capture`	same as an update, but captures original or final value (e.g. allows a = b++ with both a and b atomically updated)

- Accesses to the same location must have **compatible** types

Exercise 4

Fabio Pitari, Cineca

The code solves a 2-D Laplace equation by using a relaxation scheme.

- Parallelize the code by using OpenMP directives. Work on the most computationally intensive loop
- Try to include also the while loop in the parallel region

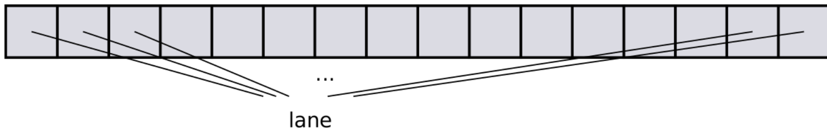
- Try to parallelize your heatflow code with OpenMP worksharing directives; take a look at `src/cxx/grid.cpp`
- Work on the most computationally intensive loops

SIMD

Fabio Pitari, Cineca

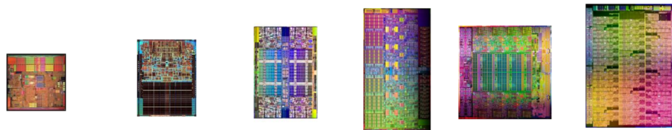
`SIMD` : Single Instruction stream, Multiple Data stream

- a vector register (or SIMD register) can hold many values of a single type;
- each value in a SIMD register is called SIMD lane or simply lane
- SIMD instructions are hardware instructions that modify the vector registers
- SIMD instruction can operate on several lanes (tipically on all the lanes) of a SIMD register at the same time



Hardware evolution (e.g. Intel)

Fabio Pitari, Cineca



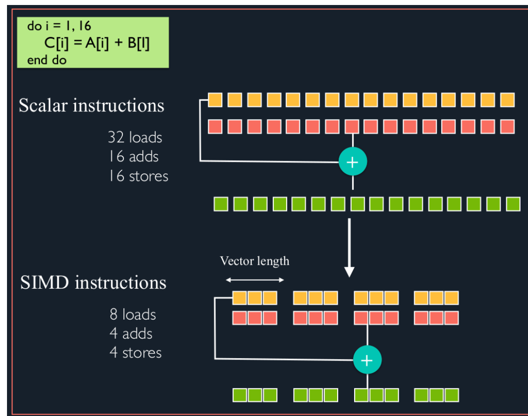
Images not intended to reflect actual die sizes

	64-bit Intel® Xeon® processor	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor E5-2600 series	Intel® Xeon Phi™ Co-processor 5110P
Frequency	3.6GHz	3.0GHz	3.2GHz	3.3GHz	2.7GHz	1053MHz
Core(s)	1	2	4	6	8	60
Thread(s)	2	2	8	12	16	240
SIMD width	128 (2 clock)	128 (1 clock)	128 (1 clock)	128 (1 clock)	256 (1 clock)	512 (1 clock)

How vectorization works

Fabio Pitari, Cineca

- Vectorization operates on entire blocks of data (vectors)
- At CPU level, a single instruction operates upon multiple data elements concurrently
- This increase the FLOP/s rate of the processor
- SIMD instructions use special SIMD registers containing multiple data elements
- Vectors help to make good use of the memory hierarchy, and to write code which has good access patterns to maximise memory bandwidth



- Compiler can detect loops or blocks of codes that can be vectorized
- Auto-vectorization relies on static analysis
- Increased complexity of instructions makes it hard for the compiler to select proper instructions
- Code pattern needs to be recognized by the compiler
- Precision requirements often inhibit SIMD code gen

Note

A common compilers' feature is to print a brief report related to vectorization

GNU (gcc, g++, gfortran) `-ftree-vectorize -ftree-vectorizer-verbose` (automatically enabled with `-O3`)

Intel (icc, icpc, ifort) `-qopt-report=2 -qopt-report-phase=vec`

Autovectorization

Fabio Pitari, Cineca

- Modern compilers are very good at automatically vectorizing the loops
- Compilers need to be sure it's safe to vectorize
- When vectorization is not considered safe, autovectorization is skipped

Some reasons for failing vectorization

- Data dependency
- Alignement (see next slide)
- Function calls in the loop
- Conditional branches
- Non-constant bounds of the loops
- Mixed data types
- Non-unit stride between two elements
- Loop body too complex (register pressure)
- Vectorization seems inefficient
- …

Note

You can try to understand why autovectorization fails increasing verbosity of its output

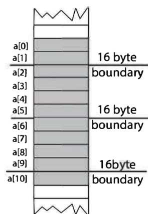
GNU (gcc, g++, gfortran) `-ftree-vectorizer-verbose=N -fopt-info-all=filename`

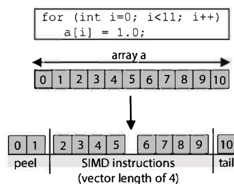
Intel (icc, icpc, ifort) `-qopt-report=5`

Autovectorization

Fabio Pitari, Cineca

- Sometimes the compiler needs help in con rming loops are vectorizeable
- To get the full bene t from SIMD, the starting address of the vectors may need to be aligned on the correct boundary
- The address in memory must be a multiple of the vector length in bytes





OpenMP provides the simd directive, in order to manually tune the vectorization of the loops by the compiler

- simd directive in OpenMP cut loops into chunks in order to fit them in vector registers
- no thread parallelization of the loop body

C/C++

```
#pragma omp simd [clause, ...]
  // structured block
```

Fortran

```
!$omp simd [clause, ...]
  ! structured block
!$omp end simd
```

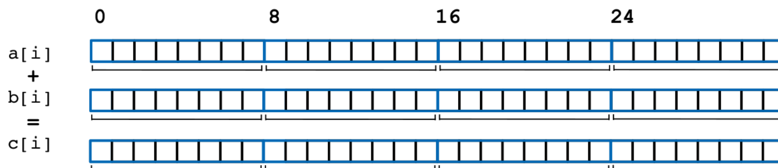
Fabio Pitari, Cineca

C/C++

```
#pragma omp simd
for ( int i=0 ; i<n ; i++ )
        c[i] = a[i] + b[i];
```

Fortran

```
!$omp simd
do i = 1,n
    c[i] = a[i] + b[i]
!$omp end simd
```



simd clauses: safelen

Fabio Pitari, Cineca

safelen allows to indicate the number of iterations that will run concurrently without breaking a dependence; i.e. the distance between to iterations in which is to safe to vectorize

C/C++

```
#pragma omp simd safelen(4)
for ( int i=1; i<SIZE-4 ; i++ ) {
        A[i] = A[i] + A[i+4];
}
```

Fortran

```
!$omp simd safelen(4)
do i=1,N-4
   A(i) = A(i) + A(i+4)
end do
!$omp end simd
```

- it can be combined with any reduction or data-sharing clauses already seen
- hardcoding explicit vector lengths may bring to code obsolescence, as vector lengths continuously change

The linear clause allows to workaround some loop dependencies among integer variables that otherwise would break vectorization.

Note

linear provides a superset of private clause functionalities

Example

- Arrays a and c are accessed through the loop variable i
- Array b is indexed through another variable j
- j has a linear relationship with the loop iteration variable i, which is incremented by 2 in each iteration, while j is incremented by one

```
#pragma omp simd linear(j:1)
for (int i=offset; i<N; i+=2)
        c[i] = a[j++] + b[i];
```

OpenMP SIMD worksharing construct

Fabio Pitari, Cineca

There is also an additional construct for simd which combines thread chunks with simd vectorization

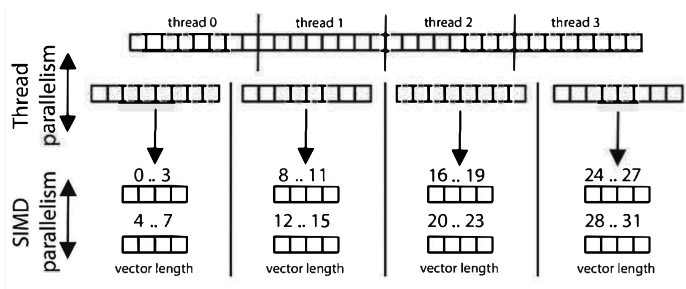
1. loops are divided into chunks (just like in the loop worksharing construct)
2. each thread is then vectorized (just like in the simd construct)

C/C++

```
#pragma omp for simd
for ( int i=0 ; i<n ; i++ )
        c[i] = a[i] + b[i];
```

Fortran

```
!$omp do simd
do i = 1,n
    c[i] = a[i] + b[i]
!$omp end do simd
```



The declare simd construct allows to point out to the compiler that some function body can be vectorized, so that the function call inside a loop does not inhibit the loop vectorization

- the function body must be a structured block
- whatever alters the execution of concurrent iterations on the SIMD unit (e.g. branching in and out from the function) breaks the compatibility with the construct
- declare simd only combines with simd directive (outer from the function call)

Example

```
#pragma omp declare simd
int min (int a, int b) {
     return a < b ? a : b;
}

#pragma omp simd
for (i=0; i<N; i++)
    c[i] = min(a[i], b[i]);
```

Optional exercise

Fabio Pitari, Cineca

- Try to repeat the previous exercises using simd parallelization
- Compare the results of vectorization with the ones of worksharing approach

Tasks

The task construct

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

C/C++

```
#pragma omp task [clause]
  // structured block
```

Fortran

```
!$omp task [ clauses ]
   ! structured block
!$omp end task
```

The task construct timeline:

2008 Introduced in OpenMP 3.0

2013 Improved in OpenMP 4.0

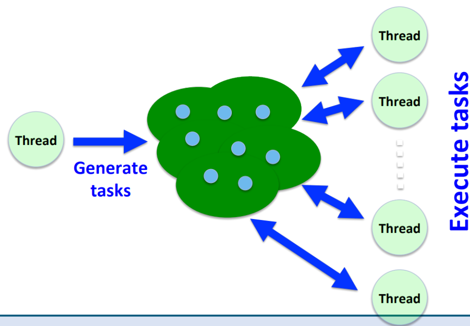
2018 Further improvements in OpenMP 5.0 (notably reduction on tasks)

Useful for dealing with:

- large and complex applications
- load unbalancing
- irregular and dynamic structures
- unbounded loops
- recursive functions

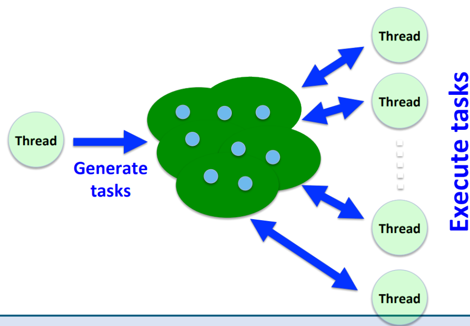
What is a task?

Is a block of instructions and data that is scheduled to be executed by a thread, concurrently with other tasks



What is a task?

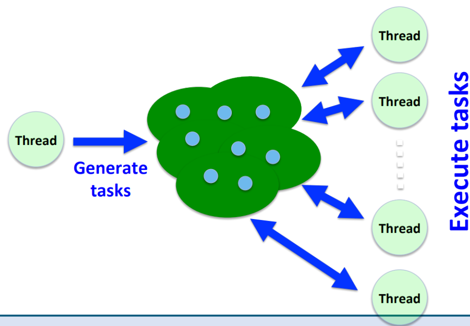
Is a block of instructions and data that is scheduled to be executed by a thread, concurrently with other tasks



1. A parallel region creates a team of threads

What is a task?

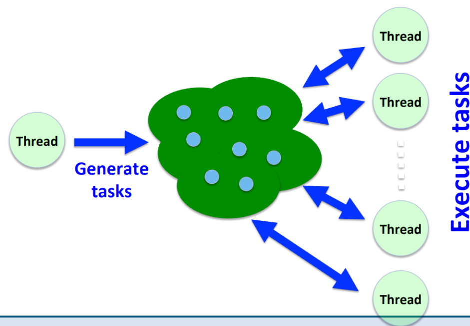
Is a block of instructions and data that is scheduled to be executed by a thread, concurrently with other tasks



1. A parallel region creates a team of threads
2. A single thread then creates the tasks, adding them to a queue that belongs to the team

What is a task?

Is a block of instructions and data that is scheduled to be executed by a thread, concurrently with other tasks



1. A parallel region creates a team of threads
2. A single thread then creates the tasks, adding them to a queue that belongs to the team
3. The *task scheduler* assigns the tasks in the queue to the threads in the team

Tasks parallelization let the system to decide at runtime when to run a task with respect to avaliable resources, which add more flexibility and asynchronicity to the execution

Note

When a parallel region is created, an implicit task is created behind the scenes with a set of instructions, which in the early stage are distributed among the avaliable threads. In contrast, using the task directive allows to queue an explicit task which will be assigned at some point to an idle thread.

Tasks are useful to do things that are hard or impossible with the `loop` and `section` constructs

Linked list example

```
#pragma omp parallel            // create a team of threads
{
#pragma omp single              // where a single thread
{
  p = head_of_list();           // starts from the head of the list
  while (!end_of_list(p)){      // and, until the end of the list,
    #pragma omp task            // submits a task
      process( p );             // that will process the element
    p = next_element(p);        // and goes to the next element
  }
}
}
```

Example: Sudoku solution

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(The example is taken from OpenMP tutorial by C. Terboven and M. Klemm)

Worksharing approach

Brute force is a recursive problem

- External loop on each blank box
- Internal loop from 1 to 9 for each blank box (open a parallel region)
- You need to check if each number is valid, so you need to fill the next blank box (open a nested parallel region) and check every combination
- …and so on for every blank box (which in turn open a further nested parallel region)

This (inefficient) procedure quickly overcrowd the available resources

Example: Sudoku solution

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(The example is taken from OpenMP tutorial by C. Terboven and M. Klemm)

Tasking approach

Each candidate combination of numbers is independent and thus can be evaluated in parallel

- External loop on each blank box

Example: Sudoku solution

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(The example is taken from OpenMP tutorial by C. Terboven and M. Klemm)

Tasking approach

Each candidate combination of numbers is independent and thus can be evaluated in parallel

- External loop on each blank box
- A single thread is selected in the parallel region in order to start the algorithm

```
#pragma omp parallel
#pragma omp single
```

Why tasks?

Fabio Pitari, Cineca

Example: Sudoku solution

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(The example is taken from OpenMP tutorial by C. Terboven and M. Klemm)

Tasking approach

Each candidate combination of numbers is independent and thus can be evaluated in parallel

- External loop on each blank box
- A single thread is selected in the parallel region in order to start the algorithm

```
#pragma omp parallel
#pragma omp single
```

- Internal loop from 1 to 9 for each blank box schedule a new task

```
#pragma omp task
```

At the end of the recursion a copy of the board with a different combination of numbers is assigned to a different task

Example: Sudoku solution

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(The example is taken from OpenMP tutorial by C. Terboven and M. Klemm)

Tasking approach

Each candidate combination of numbers is independent and thus can be evaluated in parallel

- External loop on each blank box
- A single thread is selected in the parallel region in order to start the algorithm

```
#pragma omp parallel
#pragma omp single
```

- Internal loop from 1 to 9 for each blank box schedule a new task

```
#pragma omp task
```

At the end of the recursion a copy of the board with a different combination of numbers is assigned to a different task

- master thread assign tasks as soon a thread is idle

Example: Sudoku solution

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(The example is taken from OpenMP tutorial by C. Terboven and M. Klemm)

Tasking approach

Each candidate combination of numbers is independent and thus can be evaluated in parallel

- External loop on each blank box
- A single thread is selected in the parallel region in order to start the algorithm

```
#pragma omp parallel
#pragma omp single
```

- Internal loop from 1 to 9 for each blank box schedule a new task

```
#pragma omp task
```

  At the end of the recursion a copy of the board with a different combination of numbers is assigned to a different task

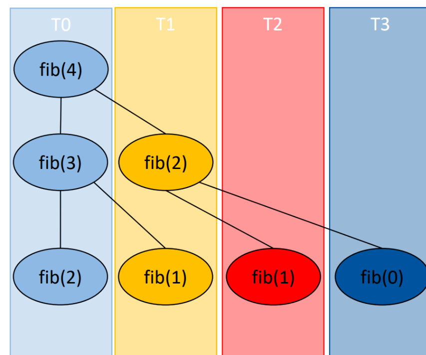
- master thread assign tasks as soon a thread is idle
- wait for completion; tasks synchronization can be achieved via the taskwait directive

```
#pragma omp taskwait
```

Fabio Pitari, Cineca

Fibonacci example

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  #pragma omp task shared(x)
  {
    x = fib(n - 1);
  }
  #pragma omp task shared(y)
  {
    y = fib(n - 2);
  }
  #pragma omp taskwait
  return x+y;
}
```



In this case the approach is recursive again, but the recursion levels are not independent one to each other.

Tasks data scoping

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
int a=1;
void foo(){
  int b=2, c=3;
  #pragma omp parallel shared(b) private(c)
  {
    int d=4;
    #pragma omp task
    {
    int e=5;
    // a is shared
    // b is shared
    // c is firstprivate
    // d is firstprivate
    // e is private
    }
  }
}
```

Rules (if `default` is not specified):

1. A variable that is determined to be `shared` in all enclosing constructs is `shared`
2. It is `firstprivate` otherwise (this avoid undefined values if switching on a different thread)

Reduction with tasks

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Starting from OpenMP 5.0, it is possible to use reduction among tasks, as long as the keyword task is specified together with the reduction clause

C/C++

```
double x[n];
double sum=0;
#pragma omp parallel for reduction (task, +:sum)
for (i=0; i<n; i++){
    #pragma omp task in_reduction(+:sum)
    sum+=x[i]
}
```

Tasks synchronization: untied clause

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- Under some condition some tasks can be suspended, for instance an outer task in which is invoked an inner one, or explicitly with the taskyeld directive

C/C++

```
#pragma omp taskyeld
```

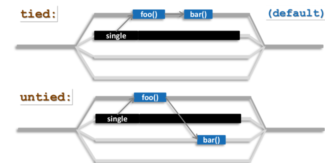
Fortran

```
!$omp taskyeld
```

- By default,a suspended task is bonded to the thread on which it was initialited, which means that such thread will stay idle until the task is restarted (deadlock risk)
- The untied clause removes this default behaviour and let the thread free while suspended; it will restart on any idle thread (this might bring to some inconsistencies with thread-related clauses not treated here)

Example

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task untied
    {
        foo();
        #pragma omp taskyeld
        bar();
    }
}
```



Tasks synchronization: taskwait

Fabio Pitari, Cineca

- taskwait is the equivalent of `barrier` for tasks
- it only waits for tasks with the same parent thread, but not for their nested tasks; use taskgroups to handle more complex schemes

Note

barrier directive waits for all the tasks in all the threads

Example

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task
        // waited
        #pragma omp task
        // waited
        {
            #pragma omp task
            // not waited
        }
        #pragma omp taskwait
    }
}
```

Tasks synchronization: taskgroups

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- taskgroup group a set of tasks and add an implicit barrier at the end
- taskgroup also allows reduction among tasks (starting from OpenMP 5.0)
- notably, this allows reduction among while loops

C/C++

```
#pragma omp taskgroup task_reduction(+:sum)
{
// some code, e.g. while loop
    #pragma omp task in_reduction(+:sum)
   sum += //...
}
```

Example

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskgroup
    {
        #pragma omp task
        // waited
        #pragma omp task
        // waited
        {
            #pragma omp task
            // waited
        }
    } // implicit taskwait
}
```

Tasks synchronization: depend

Fabio Pitari, Cineca

```
void foo(){
  int a, b=2, c=3;
  #pragma omp parallel
  {
  #pragma omp single
  {
    #pragma omp task
    b=b+7;
    #pragma omp task
    c=c+4;
    //-- some kind of barrier
    #pragma omp task
    a=b+c;
  }
  }
}
```

1. **`taskwait`**: waits for tasks spawned by current task and itself

   ```
   #pragma omp taskwait
   ```

2. **`depend`**: explicit declaration of tasks dependencies. Some more words to say ...

Tasks synchronization: depend

Fabio Pitari, Cineca


```
void foo(){
  int a, b=2, c=3;
  #pragma omp parallel
  {
  #pragma omp single
  {
    #pragma omp task depend(out:b)
    b=b+7;
    #pragma omp task depend(out:c)
    c=c+4;

    #pragma omp task depend(in:b,c)
    a=b+c;
  }
  }
}
```

- `in` it will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause
- `out` it will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in` or `inout` clause
- `inout` it will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out` or `inout` clause

The `depend` clause: an example

Fabio Pitari, Cineca


```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){




  v[0]=sum(v[0],v[1]);

  v[4]=sum(v[4],1);

  v[2]=sum(v[3],v[0]);

  v[5]=sum(v[2],v[4]);

  v[1]=sum(v[0],v[4]);

  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }


}
```

The `depend` clause: an example

Fabio Pitari, Cineca


```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){




  v[0]=sum(v[0],v[1]);

  v[4]=sum(v[4],1);

  v[2]=sum(v[3],v[0]);

  v[5]=sum(v[2],v[4]);

  v[1]=sum(v[0],v[4]);

  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }


}
```

```
Execution time:
                                  12s
Output:
                             tot = 345
```

The `depend` clause: an example

Fabio Pitari, Cineca


```c
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```c
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{

  v[0]=sum(v[0],v[1]);

  v[4]=sum(v[4],1);

  v[2]=sum(v[3],v[0]);

  v[5]=sum(v[2],v[4]);

  v[1]=sum(v[0],v[4]);

  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }
}
}
}
```

```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{

  v[0]=sum(v[0],v[1]);

  v[4]=sum(v[4],1);

  v[2]=sum(v[3],v[0]);

  v[5]=sum(v[2],v[4]);

  v[1]=sum(v[0],v[4]);

  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }
}
}
}
```

```
Execution time:
                                  12s
Output:
                            tot = 345
```

The `depend` clause: an example

Fabio Pitari, Cineca


```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{
  #pragma omp task depend(inout:v[0]) depend(in:v[1])
  v[0]=sum(v[0],v[1]);

  v[4]=sum(v[4],1);

  v[2]=sum(v[3],v[0]);

  v[5]=sum(v[2],v[4]);

  v[1]=sum(v[0],v[4]);

  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }
}
}
}
```

The `depend` clause: an example

Fabio Pitari, Cineca


```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{
  #pragma omp task depend(inout:v[0]) depend(in:v[1])
  v[0]=sum(v[0],v[1]);
  #pragma omp task depend(inout:v[4])
  v[4]=sum(v[4],1);
  #pragma omp task depend(in:v[0],v[3]) depend(out:v[2])
  v[2]=sum(v[3],v[0]);
  #pragma omp task depend(in:v[2],v[4]) depend(out:v[5])
  v[5]=sum(v[2],v[4]);
  #pragma omp task depend(in:v[0],v[4]) depend(out:v[1])
  v[1]=sum(v[0],v[4]);
  #pragma omp task depend(inout:v[3])
  v[3]=sum(v[3],-3);
  #pragma omp taskwait
  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }
}
}
}
```

The `depend` clause: an example

Fabio Pitari, Cineca


```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
Execution time:
                                    9s
Output:
                             tot = 345
```

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{
  #pragma omp task depend(inout:v[0]) depend(in:v[1])
  v[0]=sum(v[0],v[1]);
  #pragma omp task depend(inout:v[4])
  v[4]=sum(v[4],1);
  #pragma omp task depend(in:v[0],v[3]) depend(out:v[2])
  v[2]=sum(v[3],v[0]);
  #pragma omp task depend(in:v[2],v[4]) depend(out:v[5])
  v[5]=sum(v[2],v[4]);
  #pragma omp task depend(in:v[0],v[4]) depend(out:v[1])
  v[1]=sum(v[0],v[4]);
  #pragma omp task depend(inout:v[3])
  v[3]=sum(v[3],-3);
  #pragma omp taskwait
  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }
}
}
}
```

The `depend` clause: an example

Fabio Pitari, Cineca


```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{
  #pragma omp task depend(inout:v[0]) depend(in:v[1])
  v[0]=sum(v[0],v[1]);
  #pragma omp task depend(inout:v[4])
  v[4]=sum(v[4],1);
  #pragma omp task depend(in:v[0],v[3]) depend(out:v[2])
  v[2]=sum(v[3],v[0]);
  #pragma omp task depend(in:v[2],v[4]) depend(out:v[5])
  v[5]=sum(v[2],v[4]);
  #pragma omp task depend(in:v[0],v[4]) depend(out:v[1])
  v[1]=sum(v[0],v[4]);
  #pragma omp task depend(inout:v[3])
  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)
    #pragma omp task depend(in:v[i])
    {
      #pragma omp atomic update
      tot+=square(v[i]);
    }
}
}
}
```

The `depend` clause: an example

Fabio Pitari, Cineca


```c
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
Execution time:
                                    4s
Output:
                              tot = 345
```

```c
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{
  #pragma omp task depend(inout:v[0]) depend(in:v[1])
  v[0]=sum(v[0],v[1]);
  #pragma omp task depend(inout:v[4])
  v[4]=sum(v[4],1);
  #pragma omp task depend(in:v[0],v[3]) depend(out:v[2])
  v[2]=sum(v[3],v[0]);
  #pragma omp task depend(in:v[2],v[4]) depend(out:v[5])
  v[5]=sum(v[2],v[4]);
  #pragma omp task depend(in:v[0],v[4]) depend(out:v[1])
  v[1]=sum(v[0],v[4]);
  #pragma omp task depend(inout:v[3])
  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)
    #pragma omp task depend(in:v[i])
    {
      #pragma omp atomic update
      tot+=square(v[i]);
    }
}
}
}
```

Taskloop directive

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- Loops can be parallelized with tasks not only with an explicit task in the middle, but also with an external taskloop directive
- Loop chunks are scheduled on tasks
- Implcitely create a taskgroup

C/C++

```
#pragma omp taskloop
// for loop
```

Fortran

```
!$omp taskloop
! do loop
!$omp end taskloop
```

! Don't forget to enclose it in a `single` (or `master`) directive!

Taskloop directive: clauses

Fabio Pitari, Cineca

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

grainsize(N) : each task contains at least N iterations (but no more than 2N)

num_tasks(M) : create M tasks (with at least one iteration)

(If none of the above clauses is specified, the number of iteration per task is implementation defined)

nogroup : remove the implicit barrier at the end

collapse(P) : P nested loop levels are parallelized (and not just the outer loop)

reduction

Reduction among the tasks on the specified variable and operation

```
#pragma omp taskloop reduction(+:sum)
```

in_reduction

Reduction from an outer taskgroup

```
#pragma omp taskgroup task_reduction(+:sum)
{
    #pragma omp taskloop in_reduction(+:sum)
    for (i=0, i<N, i++)
        sum += a[i]
}
```

taskloop simd directive

Fabio Pitari, Cineca

- `taskloop` and `simd` can be combined in the composite construct taskloop simd
- each task of the taskloop will be vectorized (or tried to)
- every clause of both simd and taskloop can be applied

C/C++

```
#pragma omp taskloop simd
// for loop
```

Fortran

```
!$omp taskloop simd
! do loop
!$omp end taskloop
```

Optional exercise

Fabio Pitari, Cineca

- Try to repeat the previous exercises using tasks parallelization
- Compare the results of tasks parallelization with the ones of worksharing approach

Conclusions

- Always check the OpenMP version installed
- If interested, `www.openmp.org` is your bible!
- Some material, in particular for the Tasks section, is derived from the Eurofusion Webinars by Christian Terboven and Michael Klemm, whom I thank. If you want to go deeper with OpenMP topics you can find their very good lectures in the "Webinars on GPUs EUROfusion" youtube channel

A special thank to Paola Arcuri, Gianfranco Abrusci, Alessandro Colombo and to all the colleagues who contributed more or less synchronously and more or less consciously to these slides so far:

Mirko Cestari, Nitin Shukla, Fabio Affinito, Cristiano Padrin, Neva Besker, Pietro Bonfá, Gian Franco Marras, Marco Comparato, Massimiliano Culpo, Giorgio Amati, Federico Massaioli, Marco Rorro, Vittorio Ruggiero, Francesco Salvadore, Claudia Truini, etc

Thank you for your attention!

https://sctrain.eu/