Processors consist of a number of cores (or calculation units) which increases with time: initially, only one, then two, now often 4, or even 6 or 8 up to 20 for computing servers.
Processors consist of a number of cores (or calculation units) which increases with time: initially, only one, then two, now often 4, or even 6 or 8 up to 20 for computing servers.

Compared to a parallelism where calculations are done on different machines, we can use the fact that on a single machine, the memory used by all computing units is the same and can be accessible theoretically simply by all.
Processors consist of a number of cores (or calculation units) which increases with time: initially, only one, then two, now often 4, or even 6 or 8 up to 20 for computing servers.

Compared to a parallelism where calculations are done on different machines, we can use the fact that on a single machine, the memory used by all computing units is the same and can be accessible theoretically simply by all.

We can calculate on the different units at the same time, provided that you do not want to walk on your feet (that is, not to write at the same time at the same memory locations).
Processors consist of a number of cores (or calculation units) which increases with time: initially, only one, then two, now often 4, or even 6 or 8 up to 20 for computing servers.

Compared to a parallelism where calculations are done on different machines, we can use the fact that on a single machine, the memory used by all computing units is the same and can be accessible theoretically simply by all.

We can calculate on the different units at the same time, provided that you do not want to walk on your feet (that is, not to write at the same time at the same memory locations).

The OPENMP library contained in both Gnu and Intel compilers allows these operations to be performed.
Outline

- OPENMP: definition
OPENMP: definition

Definition of the world of calculation
Outline

- OPENMP: definition
- Definition of the world of calculation
- Compiling and running an OPENMP program, environment variables
Outline

- OPENMP: definition
- Definition of the world of calculation
- Compiling and running an OPENMP program, environment variables
- Parallel loop
Outline

- OPENMP: definition
- Definition of the world of calculation
- Compiling and running an OPENMP program, environment variables
- Parallel loop
- Internal functions
Outline

- OPENMP: definition
- Definition of the world of calculation
- Compiling and running an OPENMP program, environment variables
- Parallel loop
- Internal functions
- Reduction
OPENMP: definition
Definition of the world of calculation
Compiling and running an OPENMP program, environment variables
Parallel loop
Internal functions
Reduction
Conclusion and references.
OPENMP: The current version is 4.5 and dated November 2015. The version 5.0 is forthcoming!
OPENMP: The current version is 4.5 and dated November 2015. The version 5.0 is forthcoming!

Website: http://www.openmp.org/.

OPENMP is a library for Fortran, C, C++, and Python languages. Programming with OPENMP can be simple. It consists of inserting directives into an existing sequential program. You can destroy the original code! This means that the program can still operate sequentially or by using calculation units associated with a memory single center. You can destroy the original code!

Since computing sites are made up of one or more computers and networked, the ideal is to combine OPENMP and MPI, but it's more complicated for development.
OPENMP: definition and several versions

- OPENMP: The current version is 4.5 and dated November 2015. The version 5.0 is forthcoming!
- Website: http://www.openmp.org/.
- The compilers contain the OPENMP library, especially the Gnu compiler and the Intel compiler
OPENMP: definition and several versions

- OPENMP: The current version is 4.5 and dated November 2015. The version 5.0 is forthcoming!
- Website: http://www.openmp.org/.
- The compilers contain the OPENMP library, especially the Gnu compiler and the Intel compiler
- OPENMP is a library for Fortran, C,C++ and Python languages.
OPENMP: definition and several versions

- OPENMP: The current version is 4.5 and dated November 2015. The version 5.0 is forthcoming!

- Website: http://www.openmp.org/. 

- The compilers contain the OPENMP library, especially the Gnu compiler and the Intel compiler

- OPENMP is a library for Fortran, C, C++, and Python languages.

- Programming with OPENMP can be simple. It consists of inserting directives into an existing sequential program. You can destroy the original code! This means that the program can still operate sequentially or by using calculation units associated with a memory single center. You can destroy the original code!
OPENMP: definition and several versions

- OPENMP: The current version is 4.5 and dated November 2015. The version 5.0 is forthcoming!

- Website: http://www.openmp.org/.

- The compilers contain the OPENMP library, especially the Gnu compiler and the Intel compiler

- OPENMP is a library for Fortran, C, C++ and Python languages.

- Programming with OPENMP can be simple. It consists of inserting directives into an existing sequential program. You can destroy the original code! This means that the program can still operate sequentially or by using calculation units associated with a memory single center. You can destroy the original code!

- Since computing sites are made up of one or more computers and networked, the ideal is to combine OPENMP and MPI, but it’s more complicated for development.
Computing world: definition
```c
#include <stdio.h>

int main(void)
{
    #pragma omp parallel
    {
        printf("Hello, world.\n");
    }
    printf("bonjour, monde.\n");
    return 0;
}
```
the `#pragma omp` parallel and the following braces indicate that the code of this block must be parallelized.
Compiling and running a program OPENMP, environment variables

- the `#pragma omp parallel` and the following braces indicate that the code of this block must be parallelized.
- Thus the Hello, World print will appear as many times as calculation (virtual) units have been solicited.

To compile this program with gcc:
```
gcc hello.c -fopenmp -o hello
```

To compile this program with intel:
```
icc hello.c -openmp -o hello
```

If you run this program, it chooses the number of units available.
To fix the number of units, the environment variable in the terminal window must be set before launching the program:
```
export OMP_NUM_THREADS = 20
```
Compiling and running a program OPENMP, environment variables

- The `#pragma omp parallel` and the following braces indicate that the code of this block must be parallelized.

- Thus the Hello, World print will appear as many times as calculation (virtual) units have been solicited.

- To compile this program with gcc:
  ```
  gcc hello.c -fopenmp -o hello
  ```

- To compile this program with intel:
  ```
  icc hello.c -openmp -o hello
  ```

- To fix the number of units, the environment variable in the terminal window must be set before launching the program:
  ```
  export OMP_NUM_THREADS = 20
  ```
Compiling and running a program OPENMP, environment variables

- The `#pragma omp parallel` and the following braces indicate that the code of this block must be parallelized.

- Thus the Hello, World print will appear as many times as calculation (virtual) units have been solicited.

- To compile this program with gcc:
  gcc hello.c -fopenmp -o hello

- To compile this program with intel:
  icc hello.c -openmp -o hello
the `#pragma omp parallel` and the following braces indicate that the code of this block must be parallelized.

Thus the Hello, World print will appear as many times as calculation (virtual) units have been solicited.

To compile this program with gcc:
```bash
gcc hello.c -fopenmp -o hello
```

To compile this program with intel:
```bash
icc hello.c -openmp -o hello
```

If you run this program, it chooses the number of units available.
the `#pragma omp parallel` and the following braces indicate that the code of this block must be parallelized.

Thus the Hello, World print will appear as many times as calculation (virtual) units have been solicited.

To compile this program with gcc:
```
gcc hello.c -fopenmp -o hello
```

To compile this program with intel:
```
icc hello.c -openmp -o hello
```

If you run this program, it chooses the number of units available.

To fix the number of units, the environment variable in the terminal window must be set before launching the program:
```
export OMP_NUM_THREADS = 20
```
The number of units is virtual, and does not necessarily correspond to the physical number of cores. However, for a simulation code, the number of units number should not be less or equal to the number of cores. Indeed, in many cases, the running time can not decrease if you exceed this limit (to subtleties with multitheading).
The number of units is virtual, and does not necessarily correspond to the physical number of cores. However, for a simulation code, the number of units number should not be less or equal to the number of cores. Indeed, in many cases, the running time can not decrease if you exceed this limit (to subtleties with multitheading).

The pragma instruction leads to execute $N$ times the same group of instructions. It is therefore necessary to code properly for a calculation to be distributed and not executed $N$ times.
The number of units is virtual, and does not necessarily correspond to the physical number of cores. However, for a simulation code, the number of units number should not be less or equal to the number of cores. Indeed, in many cases, the running time can not decrease if you exceed this limit (to subtleties with multitheading).

The pragma instruction leads to execute $N$ times the same group of instructions. It is therefore necessary to code properly for a calculation to be distributed and not executed N times.

Physical Limitations of an OMP Program So that part of the code is executed by doing N units of computation, the system has to create threads, and finally destroy them. The time required is not always negligible.
The number of units is virtual, and does not necessarily correspond to the physical number of cores. However, for a simulation code, the number of units should not be less than or equal to the number of cores. Indeed, in many cases, the running time can not decrease if you exceed this limit (to subtleties with multitreading).

The pragma instruction leads to execute $N$ times the same group of instructions. It is therefore necessary to code properly for a calculation to be distributed and not executed $N$ times.

**Physical Limitations of an OMP Program** So that part of the code is executed by doing $N$ units of computation, the system has to create threads, and finally destroy them. The time required is not always negligible.

The time associated with this process is in the order of $0.1 \mu s$. Again very much faster than clock time of the processor. du processeur.
In order to have a parallel loop, one adds `#pragma omp parallel for`
In order to have a parallel loop, one adds `#pragma omp parallel for`

The compiler will split the loop into different parts executed on virtual units. Once executed, the program destroys the threads.
In order to have a parallel loop, one adds `#pragma omp parallel for`

The compiler will split the loop into different parts executed on virtual units. Once executed, the program destroys the threads.

One can look at what happens with the unix instruction `top` in another terminal window that the percentage is greater than 100 %. This illustrates the fact that the program mobilizes multiple computing units at runtime.
In order to have a parallel loop, one adds `#pragma omp parallel for`

The compiler will split the loop into different parts executed on virtual units. Once executed, the program destroys the threads.

One can look at what happens with the unix instruction `top` in another terminal window that the percentage is greater than 100 %. This illustrates the fact that the program mobilizes multiple computing units at runtime.

Additional instructions can be added to the previous statement that specify which variables are common to all threads and those that are internal. By default, the compiler is supposed to guess, but nothing prevents you from guiding it to make the right choices.
In order to have a parallel loop, one adds `#pragma omp parallel for`.

The compiler will split the loop into different parts executed on virtual units. Once executed, the program destroys the threads.

One can look at what happens with the unix instruction `top` in another terminal window that the percentage is greater than 100 %. This illustrates the fact that the program mobilizes multiple computing units at runtime.

Additional instructions can be added to the previous statement that specify which variables are common to all threads and those that are internal. By default, the compiler is supposed to guess, but nothing prevents you from guiding it to make the right choices.

For a loop with a loop index called `i`, we can write `# pragma omp parallel for default (shared), private (i)`.
In order to have a parallel loop, one adds \#pragma omp parallel for.

The compiler will split the loop into different parts executed on virtual units. Once executed, the program destroys the threads.

One can look at what happens with the unix instruction top in another terminal window that the percentage is greater than 100 %. This illustrates the fact that the program mobilizes multiple computing units at runtime.

Additional instructions can be added to the previous statement that specify which variables are common to all threads and those that are internal. By default, the compiler is supposed to guess, but nothing prevents you from guiding it to make the right choices.

For a loop with a loop index called i, we can write
\# pragma omp parallel for default (shared), private (i)

By changing the OMP _NUM_THREADS statement, you can change the default number of threads used.
```c
#include <stdio.h>
#include <math.h>
#include<stdio.h>
inline double essai(double x)
{ return(5.0+10.0*x+x*x*exp(x)*log(x+0.1)+sqrt(abs(x)));
}

int main(){
    int NITER=200000000;
    double *a;
    a=(double *) malloc(sizeof(double)*NITER);
    #pragma omp parallel for default(shared)
    for (int j=0;j<NITER;j++){
        a[j] = essai(j*0.01);
    }
    exit(0);
}
```
There are several internal functions that allow to have information about parallel processes and / or modifies inside of the program how the parallelization is carried out.
Internal functions

There are several internal functions that allow to have information about parallel processes and/or modifies inside of the program how the parallelization is carried out.

Among these functions, let’s mention `omp_get_num_threads()` which gives the number of threads, `omp_get_thread_num()` which gives the thread label and `omp_get_wtime()` which gives the time in decimal value.
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <omp.h>

inline double essai(double x)
{ return(5.0+10.0*x+x*x*exp(x)*log(x+0.1)+sqrt(abs(x)));}

int main()
{  
    int NITER=20000000;
    double *a;
    a=(double *) malloc(sizeof(double)*NITER);
#pragma omp parallel
    int nthreads=omp_get_num_threads();
    double debut=omp_get_wtime();
#pragma omp parallel for default(shared)
    for (int j=0;j<NITER;j++)
    {  
a[j] = essai(j*0.01);
    }
    double fin=omp_get_wtime();
    printf("nombre de threads %d temps %e \n",nthreads,fin-debut);
    exit(0);
}
Data can be collected on each computing unit and operations performed elementary addition and multiplication.
Data can be collected on each computing unit and operations performed elementary addition and multiplication.

the statement is added as argument of a pragma like reduction (operator: list)
Reduction

- Data can be collected on each computing unit and operations performed elementary addition and multiplication.
- The statement is added as argument of a pragma like reduction (operator: list)
- In addition to the usual operations, you can also search for the largest or smallest item in a list
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include<omp.h>

double f( double a ) { return (4.0 / (1.0 + a*a)); }

int main( int argc, char *argv[])
{
    int n= 1000000000, myid, numprocs;
    double PI25DT = 3.141592653589793238462643;
    double pi, h, sum=0.0;
    double startwtime = 0.0, endwtime;
    h = 1.0 / (double) n;

    #pragma omp parallel
        if (omp_get_thread_num() == 0) {startwtime = omp_get_wtime();}
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i <= n; i++)
    {
        double x = h * ((double)i - 0.5);
        sum += f(x);
    }
    pi = h * sum;
    if (omp_get_thread_num() == 0)
    {
        printf("pi is approximately %20.15e, Error is %e\n", pi, fabs(pi - PI25DT));
    endwtime = omp_get_wtime();
    printf("wall clock time = %f\n",endwtime-startwtime);
}
exit(0);
}
OpenMP is a shared memory library, so most code variables are visible, by default, by all threads.
OpenMP is a shared memory library, so most code variables are visible, by default, by all threads.

Sometimes, private variables are needed to avoid conflicts in memory and it is necessary to pass values between the sequential part and in the parallel region. Data management is done through data attribute sharing clauses by adding them to the OpenMP instructions.
The different types of clauses are

- **shared**: data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the sharing region are shared except for the loop counter.
  
- **private**: the data within a parallel region is specific to each thread, which means that each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not retained for use outside the parallel region.
  
- **default**: Allows the programmer to specify that the default value for data in a region.

The different options are either `shared`, `private`, `firstprivate` or `none`.

- **none**: The none option imposes to the programmer to declare each variable in the parallel region using the data attribute sharing clauses.
The different types of clauses are

- **shared**: data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the sharing region are shared except for the loop counter.

- **private**: the data within a parallel region is specific to each thread, which means that each thread will have a local copy and use it as a temporary variable.
The different types of clauses are

- **shared**: data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the sharing region are shared except for the loop counter.

- **private**: the data within a parallel region is specific to each thread, which means that each thread will have a local copy and use it as a temporary variable.

  - a private variable is not initialized and the value is not retained for use outside the parallel region.
The different types of clauses are

- **shared**: data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the sharing region are shared except for the loop counter.

- **private**: the data within a parallel region is specific to each thread, which means that each thread will have a local copy and use it as a temporary variable.

  1. a private variable is not initialized and the value is not retained for use outside the parallel region.
  2. by default, loop iteration counters in OpenMP loop constructs are private.
The different types of clauses are

- **shared**: data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the sharing region are shared except for the loop counter.

- **private**: the data within a parallel region is specific to each thread, which means that each thread will have a local copy and use it as a temporary variable.

  1. a private variable is not initialized and the value is not retained for use outside the parallel region.
  2. by default, loop iteration counters in OpenMP loop constructs are private.
  3. **default** Allows the programmer to specify that the default value for data in a region

The different options are either *shared, private, firstprivate* or *none*.
The different types of clauses are

- **shared**: data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the sharing region are shared except for the loop counter.

- **private**: the data within a parallel region is specific to each thread, which means that each thread will have a local copy and use it as a temporary variable.

  1. a private variable is not initialized and the value is not retained for use outside the parallel region.
  2. by default, loop iteration counters in OpenMP loop constructs are private.
  3. **default** Allows the programmer to specify that the default value for data in a region

The different options are either **shared**, **private**, **firstprivate** or **none**.

- **none** The none option imposes to the programmer to declare each variable in the parallel region using the data attribute sharing clauses.
Synchronisation clauses

The fine surgery of OPENMP programming is done with synchronization guidelines

- **critical**: the code block will be executed by one thread at a time, and not executed simultaneously by several threads. It is often used to protect shared data to avoid conflict writings in memory.
Synchronisation clauses

The fine surgery of OPENMP programming is done with synchronization guidelines

- **critical**: the code block will be executed by one thread at a time, and not executed simultaneously by several threads. It is often used to protect shared data to avoid conflict writings in memory.

- **atomic**: update in memory (write or read-modify-write) in the next instruction will be executed atomically.
Synchronisation clauses

The fine surgery of OPENMP programming is done with synchronization guidelines

- **critical**: the code block will be executed by one thread at a time, and not executed simultaneously by several threads. It is often used to protect shared data to avoid conflict writings in memory.

- **atomic**: update in memory (write or read-modify-write) in the next instruction will be executed atomically.

- **ordered**: the structured block is executed in the order in which the iterations will be executed in a sequential loop
Synchronisation clauses

The fine surgery of OPENMP programming is done with synchronization guidelines

- **critical**: the code block will be executed by one thread at a time, and not executed simultaneously by several threads. It is often used to protect shared data to avoid conflict writings in memory.

- **atomic**: update in memory (write or read-modify-write) in the next instruction will be executed atomically.

- **ordered**: the structured block is executed in the order in which the iterations will be executed in a sequential loop

- **barrier**: Each thread waits until all other threads on the team reach that point. (Very useful!)
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <omp.h>
#define NITER 20000000

double essai(double x)
{
    double c=10.0;
    return(5.0+c*x+x*x*exp(x)*log(x+0.1)+sqrt(abs(x)));
}

int main(){
    double *a;
    a=(double *) malloc(sizeof(double)* NITER);
    int b=0;
    
    #pragma omp parallel for default(shared)
    for (int j=0;j<NITER;j++){
        a[j] = essai(j/10.0);
        if((j%1000) ==0){
            #pragma omp atomic
            b++;
        }
    }

    printf(" valeur de b %d
",b);
    exit(0);
}
The OPENMP library has been developed for several Fortran, C, C++ languages is required today to use computing units of a processor efficiently.
The OPENMP library has been developed for several Fortran, C, C++ languages is required today to use computing units of a processor efficiently.

Developing an OPENMP program is easier than the one with the MPI library, but the number of cores (sharing the same memory) is generally less than ten, but significantly lower than by using many computers with the MPI library.
The OPENMP library has been developed for several Fortran, C, C++ languages is required today to use computing units of a processor efficiently.

Developing an OPENMP program is easier than the one with the MPI library, but the number of cores (sharing the same memory) is generally less than ten, but significantly lower than by using many computers with the MPI library.