

Introduction to Parallel Programming for shared memory machines using OpenMP



Ali Kerrache



compute | calcul
canada | canada

E-mail: ali.kerrache@umanitoba.ca

- ❑ Introduction to parallel programming (OpenMP)
- ❑ Definition of OpenMP **API**
 - Constitution of an OpenMP program
 - OpenMP programming Model
 - **OpenMP syntax** [C/C++, Fortran]: compiler directives
 - Run or submit an OpenMP job [SLURM, PBS]
- ❑ Learn OpenMP by Examples
 - Hello World program
 - ❖ **Work sharing in OpenMP**
 - ✓ Sections
 - ✓ Loops
 - Compute $\pi = 3.14$
 - ❖ Serial and Parallel versions
 - ❖ **Race condition**
 - ❖ SPMD model
 - ❖ **Synchronization**

❑ Use ssh client: PuTTY, MobaXterm, Terminal (Mac or Linux) to connect to **cedar** and/or **graham**:

➤ `ssh -Y username@cedar.computecanada.ca`

➤ `ssh -Y username@graham.computecanada.ca`

❑ Download the files using **wget**:

wget <https://ali-kerrache.000webhostapp.com/uofm/openmp.tar.gz>

wget <https://ali-kerrache.000webhostapp.com/uofm/openmp-slides.pdf>

Or from the website

<https://westgrid.github.io/manitobaSummerSchool2018/>

❑ Unpack the archive and change the directory:

tar -xvf openmp.tar.gz

cd UofM-Summer-School-OpenMP

Concurrency:

- ❑ Condition of a system in which multiple tasks are logically **active at the same time** ... but they may **not necessarily run in parallel**.



Parallelism:

- **subset of concurrency**
- ❑ Condition of a system in which multiple tasks are **active at the same time** and run **in parallel**.



What do we mean by parallel machines?

Serial Programming:

- Develop a serial program.
- **Performance & Optimization?**

But in real world:

- Run multiple programs.
- Large & complex problems.
- Time consuming.

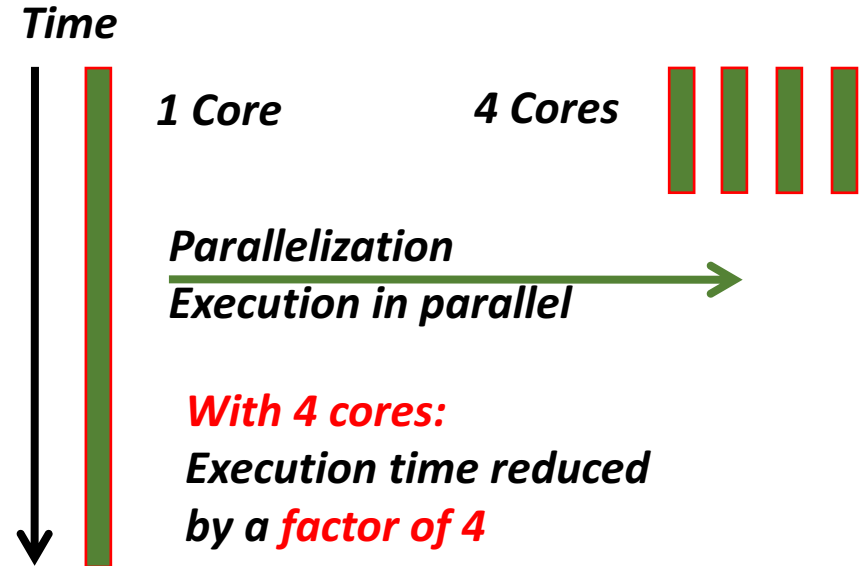
Solution:

- Use Parallel Machines.
- Use Multi-Core Machines.

Why Parallel?

- Reduce the execution time.
- Run multiple programs.

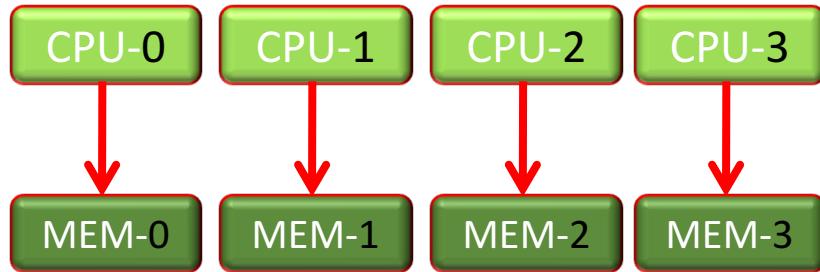
Example:



What is Parallel Programming?

Obtain the **same amount** of **computation** with multiple cores at low frequency (**fast**).

Distributed Memory Machines



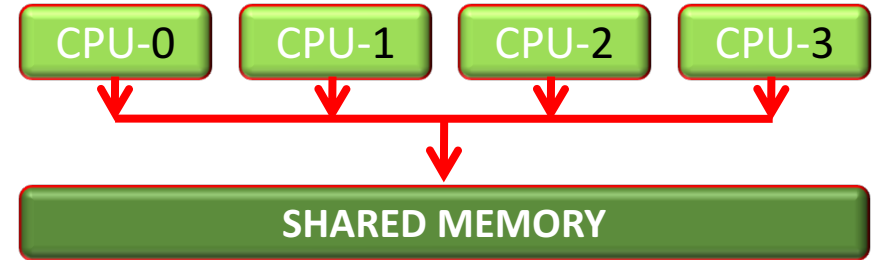
- Each processor has its **own memory**.
- The variables are **independent**.
- Communication by **passing messages** (network).

Multi-Processing

- **Difficult** to program.
- **Scalable**.

MPI based programming

Shared Memory Machines



- All processors **share the same memory**.
- The variables can be **shared** or **private**.
- Communication via **shared memory**.

Multi-Threading

- Portable, **easy** to program and use.
- **Not very scalable**.

OpenMP based programming

- ❖ **Library** used to **divide** computational **work** in a program and add **parallelism** to a serial program (**create threads**).
- ❖ **Supported** by compilers: **Intel** (ifort, icc), **GNU** (gcc, gfortran, ...).
- ❖ Programming languages: C/C++, Fortran.
- ❖ **Compilers:** <http://www.openmp.org/resources/openmp-compilers/>

OpenMP

Compiler Directives

Directives to add to a serial program.
Interpreted at compile time.

Runtime Library

Directives executed at run time.

Environment Variables

Directives introduced after compile time to control & execute OpenMP program.

Application / Serial program / End user

OpenMP

Compiler Directives

Runtime Library

**Environment
Variables**

Compilation / Runtime Library / Operating System

Thread creation & Parallel Execution

Thread 0

Thread 1

Thread 2

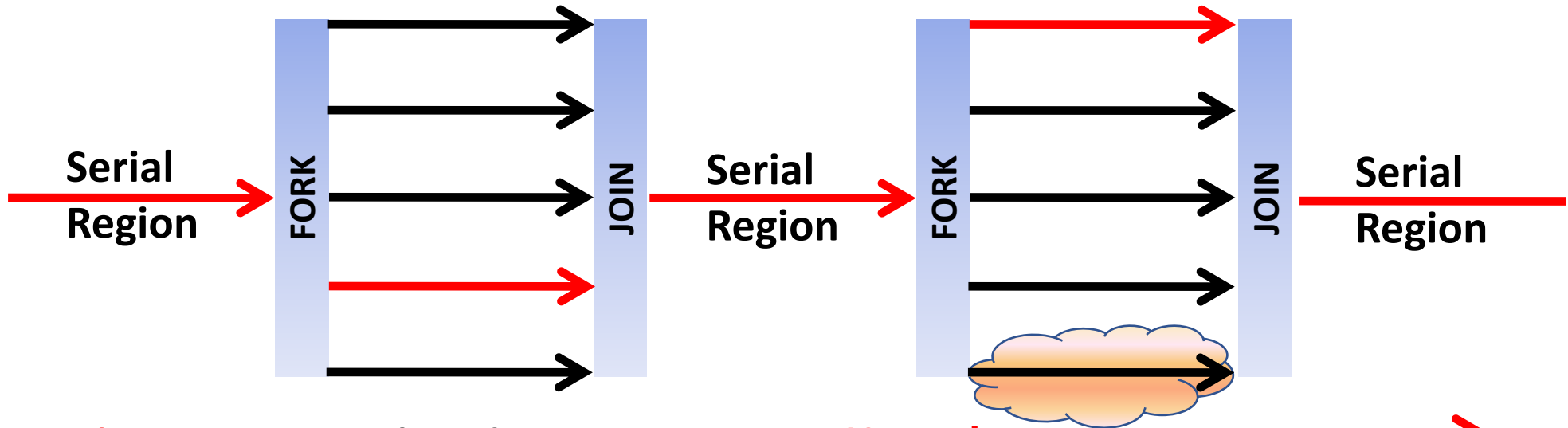
Thread 3

Thread 4

...

N-1

What is the OpenMP programming model?



Serial region: master thread
Parallel region: all threads

Nested Region

- Master thread spawns a team of threads as needed.
- The Parallelism is added incrementally: that is, the sequential program evolves into a parallel program.

Serial Program

Define the regions to parallelize, then add OpenMP directives

❖ **Example_00: Threads creation.**

- ✓ How to go from a serial code to a parallel code?
- ✓ How to **create threads**?
- ✓ Introduce some **constructs** of OpenMP.
- ✓ Compile and run an OpenMP program
- ✓ submit an OpenMP job

❖ **Example_01: Work sharing** using:

- ✓ **Loops**
- ✓ **Sections**

❖ **Example_02: Common problem in OpenMP programming.**

- ✓ False sharing and race conditions.

❖ **Example_03: Single Program Multiple Data** model:

- ✓ as solution to **avoid race conditions**.

❖ **Example_04:**

- ✓ **More OpenMP constructs.**
- ✓ **Synchronization.**

Most of the constructs in **OpenMP** are compiler directives or **pragma**:

❖ For C/C++, the **pragma** take the form:

#pragma omp *construct* [clause [clause]...]

❖ For Fortran, the directives take one of the forms:

!\$OMP *construct* [clause [clause]...]

C\$OMP *construct* [clause [clause]...]

***\$OMP *construct* [clause [clause]...]**

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

```
#pragma omp parallel
```

```
{
```

```
Block of a C/C++ code;
```

```
}
```

```
use omp_lib
```

```
!$omp parallel
```

```
Block of Fortran code
```

```
!$omp end parallel
```

- ✓ For C/C++ include the **Header** file: **#include <omp.h>**
- ✓ For **Fortran 90** use the **module**: **use omp_lib**
- ✓ For **F77** include the Header file: **include 'omp_lib.h'**

Most of **OpenMP** constructs apply to **structured blocks**

- ❑ **Structured block:** a block with one point of entry at the top and one point of exit at the bottom.
- ❑ The only “**branches**” allowed are **STOP** statements in Fortran and **exit()** in C/C++

Structured block

```
#pragma omp parallel
{
int id = omp_get_thread_num();
more: res[id] = do_big_job (id);
if (conv (res[id]) goto more;
}
printf (“All done\n”);
```

Non structured block

```
if (go_now()) goto more;
#pragma omp parallel
{
int id = omp_get_thread_num();
more: res[id] = do_big_job(id);
if (conv (res[id]) goto done;
goto more;
}
done: if (!Really_done()) goto more;
```

❑ Compile and enable OpenMP library:

- **GNU:** add **-fopenmp** to C/C++ & Fortran compilers.
- **Intel compilers:** add **-openmp**, **-qopenmp** (accepts also **-fopenmp**)
- ✓ **PGI Linux compilers:** add **-mp**
- ✓ **Windows:** add **/Qopenmp**

❑ Set the environment variable: **OMP_NUM_THREADS**

- ✓ OpenMP will spawn **one thread per hardware thread**.
- **\$ export OMP_NUM_THREADS=value** (*bash shell*)
- **\$ setenv OMP_NUM_THREADS value** (*tcsh shell*)
value: number of threads [For example 4]

❑ Execute or run the program:

- **\$./exec_program {options, parameters} or ./a.out**

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem-per-cpu=2500M
#SBATCH --time=0-00:30
```

```
# Load compiler module and/or your
# application module.
```

```
cd $SLURM_SUBMIT_DIR
```

```
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
```

```
echo "Starting run at: `date`"
```

```
./your_openmp_program_exec {options and/or parameters}
```

```
echo "Program finished with exit code $? at: `date`"
```

Resources:

- nodes=1
- ntasks=1
- cpus-per-task=1 to number of cores per node

- **Cedar:** nodes with 32 or 48 cores
- **Graham:** nodes with 32 cores
- **Niagara:** nodes with 40 cores

```
#!/bin/bash
#PBS -S /bin/bash
#PBS -l nodes=1:ppn=4
#PBS -l pmem=2000mb
#PBS -l walltime=24:00:00
#PBS -M <your-valid-email>
#PBS -m abe
```

```
# Load compiler module
# and/or your application
# module.
```

```
cd $PBS_O_WORKDIR
echo "Current working directory is `pwd`"
export OMP_NUM_THREADS=$PBS_NUM_PPN
./your_openmp_exec <input_file> output_file
echo "Program finished at: `date`"
```

Resources:

- ✓ nodes=1
- ✓ ppn=1 to maximum of N CPU (hardware)
- ✓ nodes=1:ppn=4 (for example).

On systems where \$PBS_NUM_PPN is not available, one could use:

```
CORES=`/bin/awk 'END {print NR}'  
$PBS_NODEFILE`
```

```
export OMP_NUM_THREADS=$CORES
```

shared

- only a **single instance** of variables in shared memory.
- all threads have **read** and **write** access to these variables.

private

- Each **thread** allocates its **own private copy** of the data.
- These local copies **only exist** in **parallel** region.
- **Undefined** when **entering** or **exiting** the parallel region.

firstprivate

- variables are also declared to be **private**.
- additionally, get **initialized** with value of original variable.

lastprivate

- declares variables as **private**.
- variables get **value** from the **last iteration** of the loop.

C/C++: **default (shared | none)**

Fortran: **default (private | firstprivate | shared | none)**

It is highly recommended to use: default (none)

- ❖ **Objective:** simple serial program in C/C++ and Fortran
- ❖ **Directory:** Example_00 {hello_c_seq.c; hello_f90_seq.f90}

C/C++ program

```
#include <stdio.h>
int main() {
    printf("Hello World\n");
}
```

Fortran 90 program

```
program Hello
    implicit none
    write(*,*) "Hello World"
end program Hello
```

- ❖ **To do:** compile and run the serial program (C/C++ or Fortran).

❑ C/C++:

- **icc** [CFLAGS] hello_c_seq.c -o exec_prog.x
- **gcc** [CFLAGS] hello_c_seq.c -o exec_prog.x

❑ Fortran:

- **ifort** [FFLAGS] hello_f90_seq.f90 -o exec_prog.x
- **gfortran** [FFLAGS] hello_f90_seq.f90 -o exec_prog.x

- ❑ **Run the program:** **./a.out** or **./exec_prog.x**

- ❖ **Objective:** create a parallel region and spawn threads.
- ❖ **Directory:** Example_00
- ❖ **Templates:** `hello_c_omp-template.c`; `hello_f90_omp-template.f90`

For C/C++ program

```
#include <omp.h>  
#pragma omp parallel  
{  
Structured bloc or blocs;  
}
```

For Fortran 90 program

```
use omp_lib  
!$omp parallel  
Structured bloc  
Structured bloc  
!$omp end parallel
```

- ❖ **To do:**
 - Edit the program template and add OpenMP directives:
 - ✓ compiler directives.
 - Compile and run the program of your choice (C/C++, Fortran).
 - ✓ Set the number of threads to 4 and run the program.
 - ✓ Run the same program using 2 and 3 threads.

C/C++

```
#include <omp.h>
#include <stdio.h>
int main() {
    #pragma omp parallel
    {
        printf("Hello World\n");
    }
}
```

Header

Compiler directives

Fortran 90

```
program Hello
    use omp_lib
    implicit none
    !$omp parallel
        write(*,*) "Hello World"
    !$omp end parallel
end program Hello
```

module

Compiler directives

- ❖ C and C++ use **exactly the same constructs**.
- ❖ **Slight differences** between C/C++ and Fortran.

*Next example: helloworld_*_template.**

Runtime Library

- Thread rank: ➤ **omp_get_thread_num();**
- Number of threads: ➤ **omp_get_num_threads();**
- Set number of threads: ➤ **omp_set_num_threads();**
- Compute time: ➤ **omp_get_wtime();**

Overview of the program Hello World!

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

```
#define NUM_THREADS 4
```

```
int main() {
```

```
int ID, nthr, nthreads; double start_time, elapsed_time;
```

```
omp_set_num_threads(NUM_THREADS);
```

```
nthr = omp_get_num_threads();
```

```
start_time = omp_get_wtime();
```

```
#pragma omp parallel default(none) private(ID) shared(nthreads) {
```

```
    ID = omp_get_thread_num(); nthreads = omp_get_num_threads();
```

```
    printf("Hello World!; My ID is equal to [ %d ] – The total of threads is: [ %d ]\n",
```

```
    ID, nthreads); }
```

```
elapsed_time = omp_get_wtime() - start_time;
```

```
printf("\nThe time spend in the parallel region is: %f\n\n", elapsed_time);
```

```
nthr = omp_get_num_threads();
```

```
printf("Number of threads is: %d\n\n",nthr);
```

```
}
```

Development: set number of threads.

Production: use OMP_NUM_THREADS

Set OMP_NUM_THREADS

Get number of threads (Nth = 1)

Compute elapsed time.

Get OMP_NUM_THREADS

Print number of threads (Nth = 1)

Compile

```
$ icc -openmp helloworld_c_omp.c  
$ gcc -fopenmp helloworld_c_omp.c
```

Compile

```
$ ifort -openmp helloworld_f90_omp.f90  
$ gfortran -fopenmp helloworld_f90_omp.f90
```

Run the program for **OMP_NUM_THREADS** between 1 to 4

Execute the program

```
$ export OMP_NUM_THREADS=4
```

```
$ ./a.out
```

```
Hello World!; My ID is equal to [ 0 ] - The total of threads is: [ 4 ]
```

```
Hello World!; My ID is equal to [ 3 ] - The total of threads is: [ 4 ]
```

```
Hello World!; My ID is equal to [ 1 ] - The total of threads is: [ 4 ]
```

```
Hello World!; My ID is equal to [ 2 ] - The total of threads is: [ 4 ]
```

```
$ ./a.out
```

```
Hello World!; My ID is equal to [ 3 ] - The total of threads is: [ 4 ]
```

```
Hello World!; My ID is equal to [ 0 ] - The total of threads is: [ 4 ]
```

```
Hello World!; My ID is equal to [ 2 ] - The total of threads is: [ 4 ]
```

```
Hello World!; My ID is equal to [ 1 ] - The total of threads is: [ 4 ]
```

```
$ export OMP_NUM_THREADS=1  
$ ./a.out  
$ export OMP_NUM_THREADS=2  
$ ./a.out  
$ export OMP_NUM_THREADS=3  
$ ./a.out  
$ export OMP_NUM_THREADS=4  
$ ./a.out
```

OpenMP directives for loops:

□ C/C++

- `#pragma omp parallel for { ... }`
- `#pragma omp for { ... }`

□ Fortran

`!$OMP PARALLEL DO`

...

`!$OMP END PARALLEL DO`

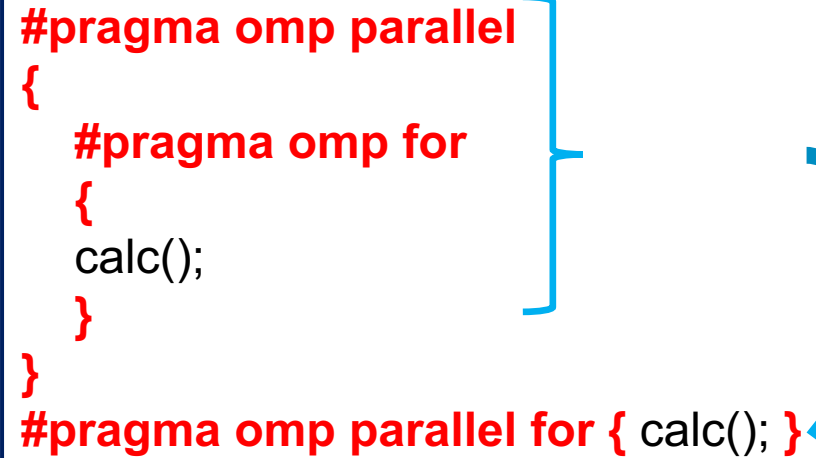
`!$OMP DO`

...

`!OMP END DO`

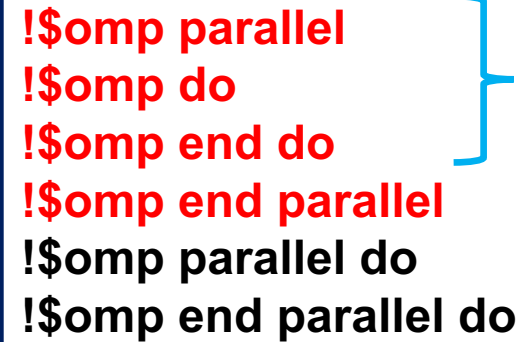
C/C++

```
#pragma omp parallel
{
    #pragma omp for
    {
        calc();
    }
}
#pragma omp parallel for { calc(); }
```



Fortran

```
!$omp parallel
!$omp do
!$omp end do
!$omp end parallel
!$omp parallel do
!$omp end parallel do
```



C/C++

```
#pragma omp parallel
{
  #pragma omp for

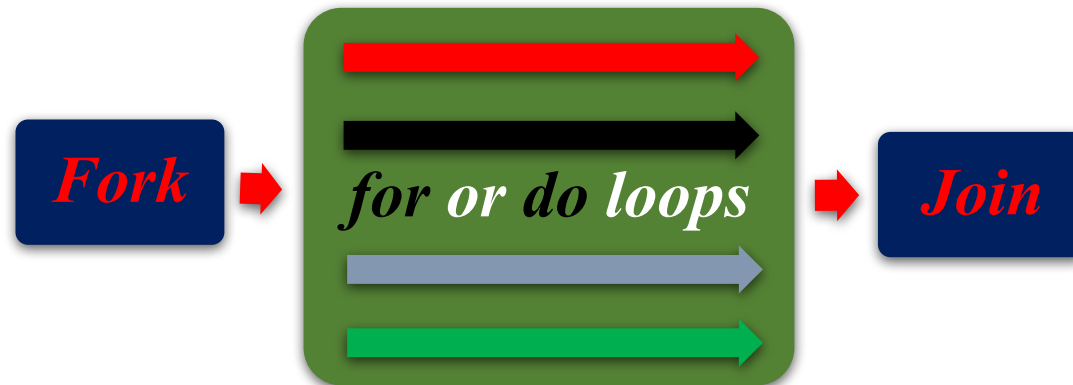
  for (i = 0; i < nloops; i++)
    do_some_computation();
}
```

#pragma omp parallel for { }

Fortran

```
!$omp parallel
!$omp do
  do i = 1, nloops
    do_some_computation
  end do
!$omp end do
!$omp end parallel
```

!\$omp parallel do
!\$omp end parallel do



C/C++

```
#include <omp.h>
#define nloops 8
int main()
{
  int ID, nthreads;
#pragma omp parallel default(none) private(ID) shared(nthreads) {
  ID = omp_get_thread_num();
  if ( ID == 0 ) { nthreads = omp_get_num_threads(); }
  int i;
#pragma omp for
  for ( i = 0; i < nloops; i++ ) {
    printf("Hello World!;
    My ID is equal to [ %d of %d ] –
    I get the value [ %d ]\n",ID,nthreads,i);  }
}
}
```

File: Example_01/

helloworld_loop_c_omp.cpp

```
#pragma omp single
  nthreads = omp_get_num_threads();
```


C/C++

```
#pragma omp parallel list-of-some-directives \  
    list-of-other-directives \  
    list-of some-other-directives  
{  
structured block of C/C++ code;  
}
```

The list of directives
continues on the next lines

Fortran

```
!$omp parallel list-of-some-directives &  
!$omp list-of-other-directives &  
!$omp list-of some-other-directives  
structured block of Fortran code  
!$omp end parallel
```

The list of directives
continues on the next lines

Fortran

```
use omp_lib
implicit none
integer :: ID, nthreads, i
integer, parameter :: nloops = 8
```

```
!$omp parallel default(none) shared (nthreads) private(ID)
```

```
  ID = omp_get_thread_num()
```

```
  if ( ID ==0 ) nthreads = omp_get_num_threads()
```

```
!$omp do
```

```
  do i = 0, nloops - 1
```

```
    write(*,fmt="(a,l2,a,l2,a,l2,a)") "Hello World!, My ID is equal to &  
      & [ ", ID, " of ",nthreads, " ] - I get the value [ ",i, "]"
```

```
  end do
```

```
!$omp end do
```

```
!$omp end paralle
```

File: Example_01/

helloworld_loop_f90_omp.f90

```
!$omp single
```

```
  nthreads = omp_get_num_threads()
```

```
!$omp end single
```

C/C++ and Fortran (last versions of OpenMP: 4.0)

Preprocessor macro **_OPENMP** for C/C++ and Fortran

#ifdef _OPENMP

```
MyID = omp_get_thread_num();
```

#endif

- Taken into account when compiled with OpenMP.
- Ignored if compiled in serial mode.

Special comment for Fortran preprocessor

```
!$ MyID = OMP_GET_THREAD_NUM()
```

Helpful check of serial and parallel version of the code

Compile and run the program

```
$ export OMP_NUM_THREADS=2  
$ ./a.out
```

```
Hello World!; My ID is equal to [ 0 of 2 ] - I get the value [ 0 ]  
Hello World!; My ID is equal to [ 1 of 2 ] - I get the value [ 4 ]  
Hello World!; My ID is equal to [ 0 of 2 ] - I get the value [ 1 ]  
Hello World!; My ID is equal to [ 1 of 2 ] - I get the value [ 5 ]  
Hello World!; My ID is equal to [ 0 of 2 ] - I get the value [ 2 ]  
Hello World!; My ID is equal to [ 1 of 2 ] - I get the value [ 6 ]  
Hello World!; My ID is equal to [ 0 of 2 ] - I get the value [ 3 ]  
Hello World!; My ID is equal to [ 1 of 2 ] - I get the value [ 7 ]
```

```
$ export OMP_NUM_THREADS=1  
$ ./a.out  
$ export OMP_NUM_THREADS=2  
$ ./a.out  
$ export OMP_NUM_THREADS=3  
$ ./a.out  
$ export OMP_NUM_THREADS=4  
$ ./a.out
```

- ❑ Thread 0 gets the values: 0, 1, 2, 3
- ❑ Thread 1 gets the values: 4, 5, 6, 7

- Thread 0 gets the values: 0, 1, 2
- Thread 1 gets the values: 3, 4, 5
- Thread 2 gets the values: 6, 7

Example of output using:
8 loops and 2 threads

Example of output using:
8 loops and 3 threads

- ❖ **Create threads:**
 - ❑ C/C++: `#pragma omp parallel { }`
 - ❑ Fortran: `!$omp parallel !$omp end parallel`
- ❖ Include the header: **<omp.h>** in C/C++; and **use omp_lib** in Fortran
- ❖ **Number of threads:** `omp_get_num_threads()`
- ❖ **Thread number or rank:** `omp_get_thread_num()`
- ❖ **Set number of threads:** `omp_set_num_threads()`
- ❖ **Evaluate the time:** `omp_get_wtime()`
- ❖ **single construct:** `omp_single()`
- ❖ **Variables:**
 - `default(none)`, `shared()`, `private()`
- ❖ **Work sharing: loops, sections [section]:**
 - C/C++: `#pragma omp for` or `#pragma omp parallel for`
 - ✓ Fortran:
 - ❑ `!$omp do ... !$omp end do`
 - ❑ `!$omp parallel do ... !$omp end parallel do`

Mathematically:

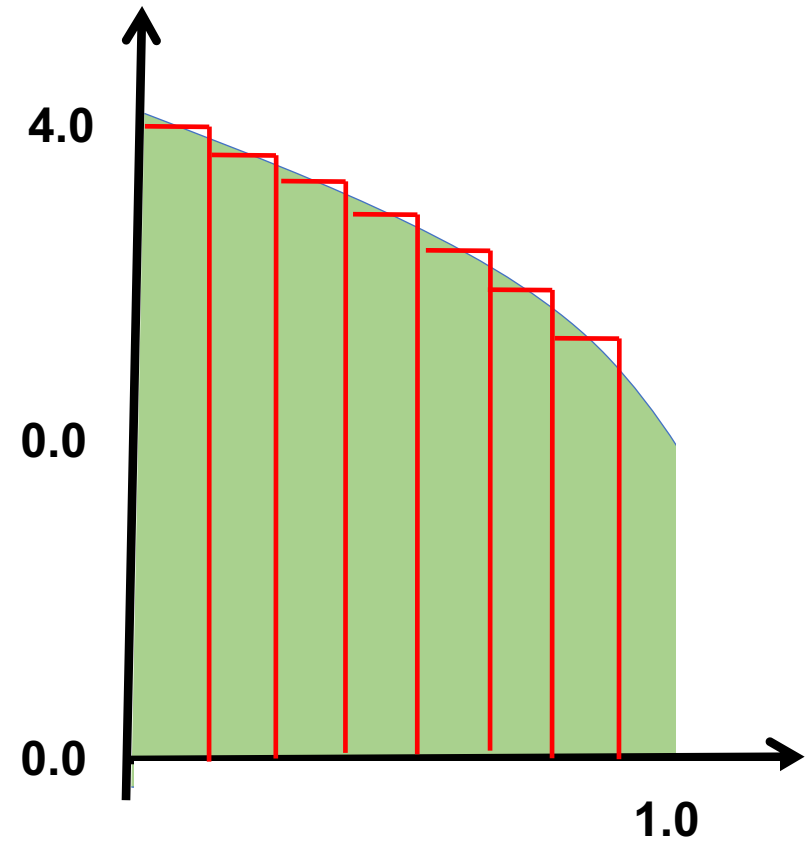
$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

This function can be approximated by a sum of rectangles:

$$\sum_{i=1}^n F(X_i) \Delta X \approx \pi$$

Where each rectangle has a width ΔX and height $F(X_i)$ at the middle of the interval $[i, i+1]$

Numerical integration:



- Directory: **Example_02**
- Files: **compute_pi_c_seq.c; compute_pi_f90_seq.f90**

C/C++

```
double x, pi, sum;
int i;
sum = 0.0;
for (i = 0; i < nb_steps; i++) {
    x = (i + 0.5) * step;
    sum += 1.0/(1.0 + x * x);
}
pi = 4.0 * sum * step;
```

Compile & run the code

```
$ gcc compute_pi_c_seq.c
$ ./a.out
pi = 3.14159
```

Fortran

```
real(8) :: pi, sum, x
integer :: i
sum = 0.0d0
do i = 0, nb_steps
    x = (i + 0.5) * step
    sum = sum + 1.0/(1.0 + x * x)
end do
pi = 4.0 * sum * step
```

Compile & run the code

```
$ gfortran compute_pi_f90_seq.f90
$ ./a.out
pi = 3.14159
```

File: Example_02

compute_pi_c_omp-template.c

File: Example_02

compute_pi_f90_omp-templtae.f90

To Do:

- ❖ Add the compiler directives to create the OpenMP version:
 - C/C++: **#pragma omp parallel** { }
 - Fortran: **!\$omp parallel** **!\$omp end parallel**
- ❖ Include the header: **<omp.h>** in C/C++; and **use omp_lib** in Fortran
- ❖ Variables:
 - **default(none)**, **shared()**, **private()**
- Optionally: **omp_get_wtime()**

Change the program and compile

```
$ gcc -fopenmp compute_pi_c_omp-template.c
```

```
$ gfortran -fopenmp compute_pi_f90_omp-template.f90
```


File: Example_02

`compute_pi_c_omp_race.c`

C/C++

```
#pragma omp parallel default(none)
private(i) shared(x,sum) {
  int i; double x;
  for (i = 0; i < nb_steps; i++) {
    x = (i + 0.5) * step;
    sum += 1.0/(1.0 + x * x);
  }
}
pi = 4.0*sum*step;
```

File: Example_02

`compute_pi_f90_omp_race.f90`

Fortran

```
!$omp parallel default(none)
private(i) shared(x,sum)

do i = 0, nb_steps
  x = (i + 0.5) * step
  sum = sum + 1.0/(1.0 + x * x)
end do
!$omp end parallel
pi = 4.0*sum*step
```

Compile and run the code

```
$ gcc -fopenmp compute_pi_c_omp_race.c
$ gfortran -fopenmp compute_pi_f90_omp_race.f90
```

Compile & run the program

`compute_pi_c_omp_race.c`

Compile & run the program

`compute_pi_f90_omp_race.f90`

Run the program

\$./a.out

The value of pi is [**9.09984**]; Computed using [**20000000**] steps in [**9.280**] s.

\$./a.out

The value of pi is [**11.22387**]; Computed using [**20000000**] steps in [**11.020**] s.

\$./a.out

The value of pi is [**5.90962**]; Computed using [**20000000**] steps in [**5.640**] s.

\$./a.out

The value of pi is [**8.89411**]; Computed using [**20000000**] steps in [**8.940**] s.

\$./a.out

The value of pi is [**10.94186**]; Computed using [**20000000**] steps in [**10.870**] s.

\$./a.out

The value of pi is [**10.89870**]; Computed using [**20000000**] steps in [**11.030**] s.

Wrong answer & slower than serial program

How to solve this problem?

SPMD:

- ❑ a technique to achieve parallelism.
- ❑ each thread receive and execute a copy of a same program.
- ❑ each thread will execute a copy as a function of its ID.

➤ **Cyclic Distribution**

Thread 0:	0, 3, 6, 9
Thread 1:	1, 4, 7, 10, ...
Thread 2:	2, 5, 8, 11, ...

C/C++

```
#pragma omp parallel  
{  
    for (i=0; i < n; i++) { computation[i]; }  
}
```

SPMD

```
#pragma omp parallel  
{  
    int numthreads = omp_get_num_threads();  
    int ID = omp_get_thread_num();  
    for (i=0+ID; i < n; i+=numthreads) {  
        computation[i][ID]; }  
}
```

File: Example_03/

`compute_pi_c_spmd-template.c`

File: Example_03/

`compute_pi_f90_spmd-template.f90`

- ❖ Add the compile directives to create the OpenMP version:
 - C/C++: `#pragma omp parallel { }`
 - Fortran: `!$omp parallel !$omp end parallel`
- ❖ Include the header: `<omp.h>` in C/C++; and `use omp_lib` in Fortran
- ❖ Promote the variable `sum` to an array: each thread will compute a `sum` as a function of its `ID`; then compute a global `sum`.
- ❖ Compile and run the program.

File: Example_03/

`compute_pi_c_spmd_simple.c`

C/C++

```
#pragma omp parallel
{
  Int nthreads = omp_get_num_threads();
  Int ID = omp_get_thread_num();
  sum[id] = 0.0;
  for (i = 0+ID; i < nb_steps; i+=nthreads) {
    x = (i + 0.5) * step;
    sum[ID] = sum[ID] + 1.0/(1.0 + x*x); }
}
compute_tot_sum(); [ i = 1 to nthreads]
pi = 4.0 * tot_sum * step;
```

File: Example_03/

`compute_pi_f90_spmd_simple.f90`

Fortran

```
!$omp parallel
nthreads = omp_get_num_threads()
ID = omp_get_thread_num();
sum(id) = 0.0
do i = 1+ID, nb_steps, nthreads
  x = (i + 0.5) * step;
  sum(ID) = sum(ID) + 1.0/(1.0 + x*x);
end do
!$omp end parallel
compute_tot_sum [ i = 1 to nthreads]
pi = 4.0 * tot_sum * step
```

Compile and run the code: **the answer is correct but very slow than serial**

Execute the program

\$ a.out

The value of pi is [**3.14159**; Computed using [**20000000**] steps in [**0.4230**] seconds
The value of pi is [**3.14166**; Computed using [**20000000**] steps in [**1.2590**] seconds
The value of pi is [**3.14088**; Computed using [**20000000**] steps in [**1.2110**] seconds
The value of pi is [**3.14206**; Computed using [**20000000**] steps in [**1.9470**] seconds

The answer is correct

Slower than serial program

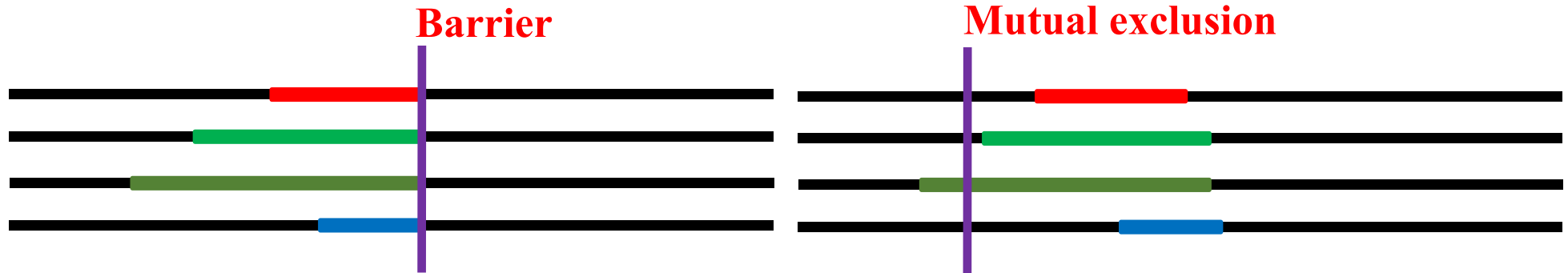
❖ How to speed up the execution of pi program?

➤ **Synchronization**

➤ **Control how the variables are shared to avoid race condition**

Synchronization: Bringing one or more threads to a well defined point in their execution.

- **Barrier:** each thread wait at the barrier until all threads arrive.
- **Mutual exclusion:** one thread at a time can execute.



High level constructs:

- critical
- atomic
- barrier
- ordered

Low level constructs:

- flush
- locks:
 - simple
 - nested

Synchronization:

- can reduce the performance.
- cause overhead and cost a lot.
- more barriers will serialize the program.
- Use it when needed.

C/C++

```
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    A[ID] = Big_A_Computation(ID);

    #pragma omp barrier
    A[ID] = Big_B_Computation(A, ID);
}
```

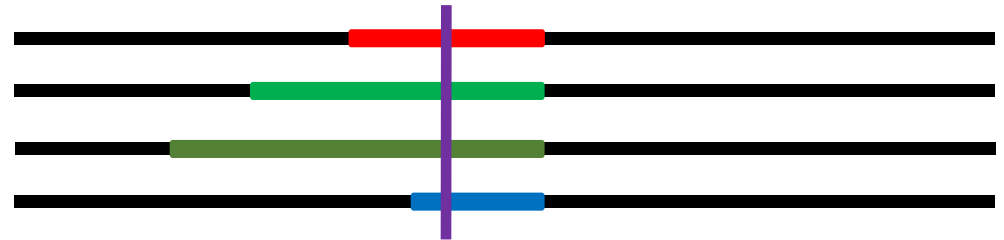
Fortran

```
!$omp parallel
    int ID = omp_get_thread_num()
    A[ID] = Big_A_Computation(ID)

!$omp barrier
    A[ID] = Big_B_Computation(A, ID)
!$omp end barrier

!$omp end parallel
```

- **Barrier:**
each thread wait at the barrier
until **all** threads arrive.



C/C++

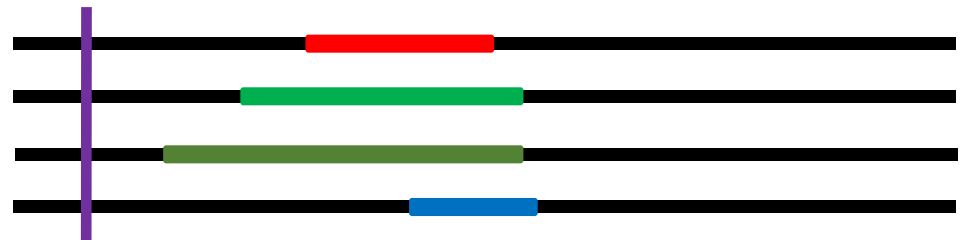
```
#pragma omp parallel
{
  float B; int i, id, nthrds;
  id = omp_get_thread_num();
  nthrds = omp_get_num_threads();
  for (i=id; i < niters; i+=nthrds) {
    B = big_calc_job(i);
    #pragma omp critical
    res += consume (B);
  }
}
```

Fortran

```
!$omp parallel
  real(8) :: B; integer :: i, id, nthrds
  id = omp_get_thread_num()
  nthrds = omp_get_num_threads()
  do i = id, niters, nthrds
    B = big_calc_job(i);
    !$omp critical
    res = res + consume (B);
  !$omp end critical
  end do
!$omp end parallel
```

Mutual exclusion:

- **Critical:** only one thread at a time can enter a critical region (**calls consume()**)



Synchronization: atomic (basic form),

- Atomic provides mutual exclusion but only applies to the update of a statement of a memory location: **update of X variable in the following example.**

C/C++

```
#pragma omp parallel  
{  
    double tmp, B;  
    B = DOIT();  
    tmp = big_calculation(B);  
    #pragma omp atomic  
        X += tmp;  
}
```

Fortran

```
!$omp parallel  
real(8) :: tmp, B  
B = DOIT()  
tmp = big_calculation(B)  
!$omp atomic  
    X = X + tmp  
!$omp end parallel
```

- ❖ Aggregating values from different threads is a common operation that OpenMP has a special *reduction variable*
 - Similar to private and shared
 - Reduction variables support several types of operations: + - *

- ❖ Syntax of the reduction clause: **reduction (op : list)**

- ☐ Inside a parallel or a work-sharing construct:
 - A **local copy** of each list of variables is **made and initialized** depending on the “**op**” (e.g. 0 for “+”, 0 for -, 1 for *).
 - Updates occur on the **local copy**.
 - Local copies are reduced into a **single value** and combined with the **original global value**.
 - The variables in “**list**” must be shared in the enclosing parallel region.

C/C++

```
Int MAX = 10000;
double ave=0.0;
A[MAX]; int i;

#pragma omp parallel for
reduction (+:ave)
  for (i=0; i < MAX; i++) {
    ave + = A[i];
  }
ave = ave / MAX
```

Fortran

```
real(8) :: ave = 0.0;
integer :: MAX = 10000
real :: A(MAX); integer :: I

!$omp parallel do reduction(+:ave)
  do i = 1, MAX
    ave = ave + A(i)
  end do
!$omp end parallel do
ave = ave / MAX
```

- ❖ The variable **ave** is initialized outside the parallel region.
- ❖ **Inside the parallel region:**
 - Each thread will have **its own copy, initialize it, update it.**
 - At the end, all the **local copies will be reduced** to a final result.

Files: Example_04/

C/C++: `compute_pi_c_omp_critical-template.c`
`compute_pi_c_omp_reduction-template.c`

F90: `compute_pi_f90_omp_critical-template.f90`
`compute_pi_f90_omp_reduction-template.f90`

- ❖ Start from the sequential version of pi program, then add the compile directives to create the OpenMP version:
 - **C/C++:** `#pragma omp parallel { }`
 - **Fortran:** `!$omp parallel !$omp end parallel`
 - Include the header: `<omp.h>` in C/C++; and `use omp_lib` in Fortran
- ❖ Use the **SPMD** pattern with critical construct in one version and reduction in the second one.
- ❖ Compile and run the programs.

Example of output

\$ a.out

The Number of Threads = 1

The value of pi is [**3.14159**]; Computed using [**2000000**] steps in [**0.40600**] seconds

The Number of Threads = 2

The value of pi is [**3.14159**]; Computed using [**2000000**] steps in [**0.20320**] seconds

The Number of Threads = 3

The value of pi is [**3.14159**]; Computed using [**2000000**] steps in [**0.13837**] seconds

The Number of Threads = 4

The value of pi is [**3.14159**]; Computed using [**2000000**] steps in [**0.10391**] seconds

□ Results:

- **Correct results.**
- **The program runs faster (4 times faster using 4 cores).**

OpenMP:

- ❑ **create threads:**
 - C/C++ **#pragma omp parallel { ... }**
 - Fortran: **!\$omp parallel ... !\$omp end parallel**
- ❑ **Work sharing: (loops and sections).**
- ❑ **Variables: default(none), private(), shared()**
- **Environment variables and runtime library.**

Few construct of OpenMP:

- **single** construct
- **barrier** construct
- **atomic** construct
- **critical** construct
- **reduction** clause

**omp_set_num_threads()
omp_get_num_threads()
omp_get_thread_num()
omp_get_wtime()**

**For more advanced runtime library clauses
and constructs, visit:**

<http://www.openmp.org/specifications/>

OpenMP - API:

- **Simple parallel programming** for shared memory machines.
- **Speed up the execution** (but not very scalable).
- **compiler directives, runtime library, environment variables.**

Take a serial code, add the compiler directives and test:

- **Define concurrent regions that can run in parallel.**
- **Add compiler directives and runtime library.**
- **Control how the variables are shared.**
- **Avoid the false sharing and race condition by adding synchronization clauses (choose the right ones).**
- **Test the program and compare to the serial version.**
- **Test the scalability of the program as a function of threads.**

- **OpenMP:** <http://www.openmp.org/>
- **Compute Canada Wiki:** <https://docs.computecanada.ca/wiki/OpenMP>
- **Reference cards:** <http://www.openmp.org/specifications/>
- **OpenMP Wiki:** <https://en.wikipedia.org/wiki/OpenMP>
- **Examples:**
<http://www.openmp.org/updates/openmp-examples-4-5-published/>
- **Contact:** support@westgid.ca
- **WestGrid events:** <https://www.westgrid.ca/events>



Thank you