Shared memory parallel programming models

Pthread and OpenMP by example

Houssam-Eddine Zahaf
houssameddine.zahaf@univ-nantes.fr

shared-mem

Plan

1. Introduction to shared-memory parallel programming model

2. Parallel programming using OpenMP

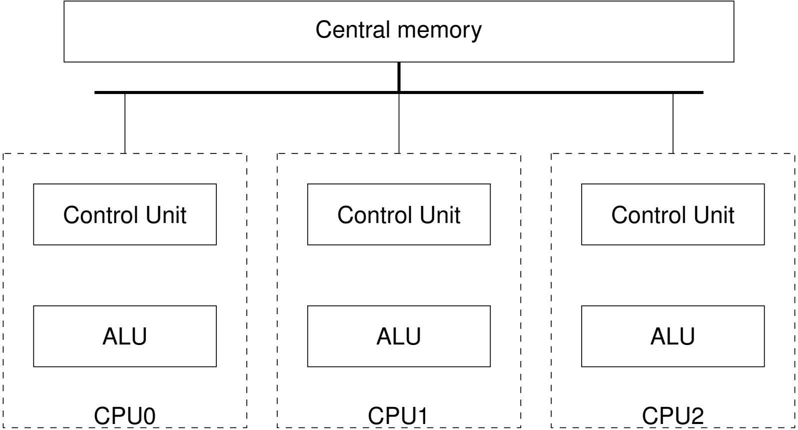
3. Pthread as shared memory parallel programming model

4. Exercices

Plan

1. **Introduction to shared-memory parallel programming model**
2. Parallel programming using OpenMP
3. Pthread as shared memory parallel programming model
4. Exercices

Shared memory model : Multiple CPU - Single DRAM



- Global memory which can be accessed by all processors of a parallel computer.
- Data in the global memory can be read/write by any of the processors.

Shared memory : programming

- Programmed thanks to a collection of threads
    - A thread is the smallest schedulable unit within an operating system
    - Can be created dynamically

Shared memory : programming

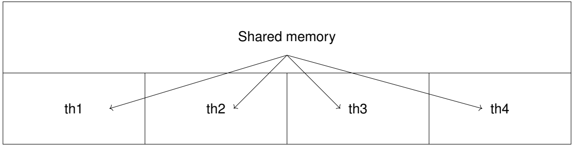
- Programmed thanks to a collection of threads
  - A thread is the smallest schedulable unit within an operating system
  - Can be created dynamically
- Each thread has its set of **private** variables (local stack variables)
- Might share variables

Shared memory : programming

- Programmed thanks to a collection of threads
  - A thread is the smallest schedulable unit within an operating system
  - Can be created dynamically
- Each thread has its set of **private** variables (local stack variables)
- Might share variables

Threads communicate

- **implicitly** by reading and writing shared variables
- coordinate using synchronization mechanisms on shared variables



Programming platforms for shared memory hardware

- Several Thread Libraries/systems
  - **PTHREADS** is the POSIX Standard (Portable Operating System Interface)

Programming platforms for shared memory hardware

- Several Thread Libraries/systems
  - **PTHREADS** is the POSIX Standard (Portable Operating System Interface)
  - **OpenMP standard** for application level programming OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran on many platforms, instruction-set architectures and operating systems.

Programming platforms for shared memory hardware

- Several Thread Libraries/systems
  - **PTHREADS** is the POSIX Standard (Portable Operating System Interface)
  - **OpenMP standard** for application level programming OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran on many platforms, instruction-set architectures and operating systems.

  - **TBB** Thread Building Blocks Intel is a C++ template library developed by Intel for parallel programming on multi-core processors. Using TBB, a computation is broken down into tasks that can run in parallel. The library manages and schedules threads to execute these tasks.
  - **CILK: Cilk, Cilk++, Cilk Plus and OpenCilk** are general-purpose programming languages designed for multithreaded parallel computing. They are based on the C and C++ programming languages.
  - **Java threads** Built on top of POSIX threads : threads in the java virtual machine

Plan

1. Introduction to shared-memory parallel programming model
2. **Parallel programming using OpenMP**
3. Pthread as shared memory parallel programming model
4. Exercices

OpenMP

- Stands for Open MultiProcessing

- Three languages supported: C, C++, Fortran

- Portable :
    - Supported on multiple operating systems: UNIX, Linux, Windows, etc.
    - Supported by multiple compilers : gcc, Intel C/C++, etc

OpenMP

- Stands for Open MultiProcessing
- Three languages supported: C, C++, Fortran
- Portable :
  - Supported on multiple operating systems: UNIX, Linux, Windows, etc.
  - Supported by multiple compilers : gcc, Intel C/C++, etc

Compiler directives

Functions library

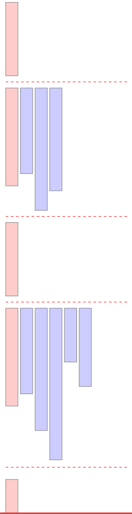
Environment variables

OpenMP execution model

- OpenMP program is a sequence fork-joins (sequential and parallel regions)

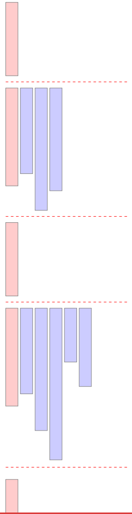
OpenMP execution model

- OpenMP program is a sequence fork-joins (sequential and parallel regions)
- Sequential code is executed by the main process/thread



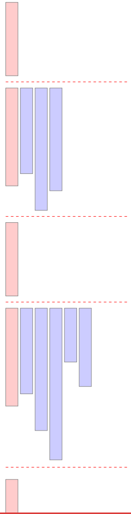
OpenMP execution model

- OpenMP program is a sequence fork-joins (sequential and parallel regions)
- Sequential code is executed by the main process/thread
- Parallel code can be executed by one or multiple workers



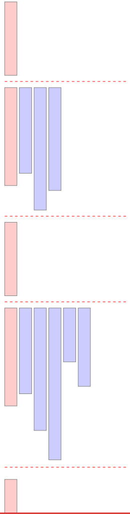
OpenMP execution model

- OpenMP program is a sequence fork-joins (sequential and parallel regions)
- Sequential code is executed by the main process/thread
- Parallel code can be executed by one or multiple workers
- Synchronization primitives can be used to synchronize each fork-joint phase



OpenMP execution model

- OpenMP program is a sequence fork-joins (sequential and parallel regions)
- Sequential code is executed by the main process/thread
- Parallel code can be executed by one or multiple workers
- Synchronization primitives can be used to synchronize each fork-joint phase
- It allows to parallelize:
  - loops
  - Region
  - Functions
  - etc.



OpenMP Structure

- OpenMP parallelization instruction are provided through `directives` and `clauses`
- They define how to (i) share work between several workers, synchronize them and (ii) the policy to the shared data management.

OpenMP Structure

- OpenMP parallelization instruction are provided through `directives` and `clauses`
- They define how to (i) share work between several workers, synchronize them and (ii) the policy to the shared data management.
- pragmas start by “#” and they are ignored by default, except if the correct compiler options are specified → helps to write sequential and parallel program in the same “syntax”

OpenMP Structure

- OpenMP parallelization instruction are provided through `directives` and `clauses`
- They define how to (i) share work between several workers, synchronize them and (ii) the policy to the shared data management.
- pragmas start by “#” and they are ignored by default, except if the correct compiler options are specified → helps to write sequential and parallel program in the same “syntax”

OpenMP routine

- Functions and subroutines are part of a OpenMP library loaded at link time

OpenMP Structure

- OpenMP parallelization instruction are provided through `directives` and `clauses`
- They define how to (i) share work between several workers, synchronize them and (ii) the policy to the shared data management.
- pragmas start by "#" and they are ignored by default, except if the correct compiler options are specified → helps to write sequential and parallel program in the same "syntax"

OpenMP routine

- Functions and subroutines are part of a OpenMP library loaded at link time

Behavior

- pragma(s) might trigger fork-join section
- the master task spawns ("forks") children tasks and ensures that at their completion the master thread "gathers" to continue onward the execution.

Main pragma(s)

We can define a parallel region using

```
#pragma omp parallel
{
  /* Parallel region code */
}
```

We can define a parallel loop using

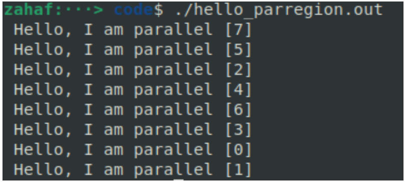
```
#pragma omp for
  /* the for loop */
```

OpenMP : sharing clauses

- `shared(...)`: list of shared variables by all OpenMP tasks
- `private(...)`: list of variables that are visible only by their task
  - the variable is not initialized within the parallel part of the code
  - the variable get its initial value when leaving the parallel region
- `firstprivate(...)`: Similar to the previous but data are initialized with their value before the parallel part of the code.
- `default(none,private,shared)`: denotes the default behavior for given variables set to shared
- by default if the sharing is not specified, it is set to shared

Hello parallel region

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

int main(int argc, char ** argv){
  # pragma omp parallel
  {
    int tid = omp_get_thread_num();
    printf (" Hello, I am parallel [%d] \n",tid);
  }
  return EXIT_SUCCESS ;
}
```



```
zahaf:···> code$ ./hello_parregion.out
 Hello, I am parallel [7]
 Hello, I am parallel [5]
 Hello, I am parallel [2]
 Hello, I am parallel [4]
 Hello, I am parallel [6]
 Hello, I am parallel [3]
 Hello, I am parallel [0]
 Hello, I am parallel [1]
```

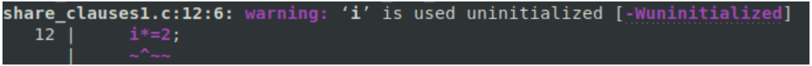
- `gcc hello_parregion.c -o hello_parregion.out -Wall -fopenmp`

Hello share clauses (1)

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


int main(int argc, char ** argv){
  int i = 0;
# pragma omp parallel private(i)
  {
    int tid = omp_get_thread_num();
    i*=2;
    printf (" ID [%d] : i : %d \n",tid,i);
  }
  return EXIT_SUCCESS ;
}
```

Hello share clauses (1)

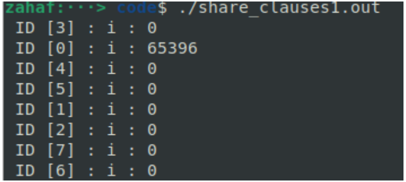
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>


int main(int argc, char ** argv){
  int i = 0;
# pragma omp parallel private(i)
  {
    int tid = omp_get_thread_num();
    i*=2;
    printf (" ID [%d] : i : %d \n",tid,i);
  }
  return EXIT_SUCCESS ;
}
```



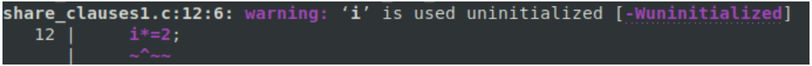
```
share_clauses1.c:12:6: warning: ‘i’ is used uninitialized [-Wuninitialized]
   12 |       i*=2;
      |       ~^~~
```

Hello share clauses (1)

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


int main(int argc, char ** argv){
  int i = 0;
# pragma omp parallel private(i)
  {
    int tid = omp_get_thread_num();
    i*=2;
    printf (" ID [%d] : i : %d \n",tid,i);
  }
  return EXIT_SUCCESS ;
}
```



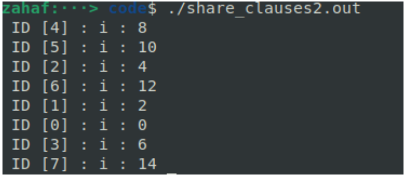
```
zahaf:···> code$ ./share_clauses1.out
 ID [3] : i : 0
 ID [0] : i : 65396
 ID [4] : i : 0
 ID [5] : i : 0
 ID [1] : i : 0
 ID [2] : i : 0
 ID [7] : i : 0
 ID [6] : i : 0
```



```
share_clauses1.c:12:6: warning: ‘i’ is used uninitialized [-Wuninitialized]
   12 |     i*=2;
      |     ~^~~
```

Hello share clauses (2) - Correction

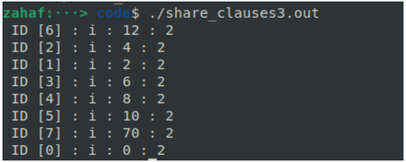
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>


int main(int argc, char ** argv){
  int i = 2;
# pragma omp parallel firstprivate(i)
  {
    int tid = omp_get_thread_num();
    i*=tid;
    printf (" ID [%d] : i : %d \n",tid,i);
  }
  return EXIT_SUCCESS ;
}
```



```
zahaf:···> code$ ./share_clauses2.out
 ID [4] : i : 8
 ID [5] : i : 10
 ID [2] : i : 4
 ID [6] : i : 12
 ID [1] : i : 2
 ID [0] : i : 0
 ID [3] : i : 6
 ID [7] : i : 14
```

Hello share clauses

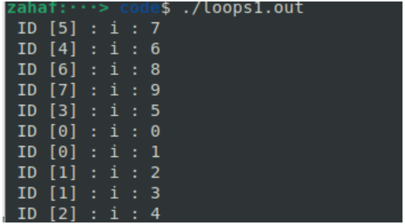
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>


int main(int argc, char ** argv){
  int i = 2;
# pragma omp parallel default(shared)
  {
    int tid = omp_get_thread_num();
    i*=tid;
    int z = 1;
    z++;
    printf (" ID [%d] : i : %d : %d\n",tid,i, z);
  }
  return EXIT_SUCCESS ;
}
```



```
zahaf:···> code$ ./share_clauses3.out
 ID [6] : i : 12 : 2
 ID [2] : i : 4 : 2
 ID [1] : i : 2 : 2
 ID [3] : i : 6 : 2
 ID [4] : i : 8 : 2
 ID [5] : i : 10 : 2
 ID [7] : i : 70 : 2
 ID [0] : i : 0 : 2
```

Loops by example

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


int main(int argc, char ** argv){
# pragma omp parallel
  {
#pragma omp for
    for (int i=0;i<10;i++){

      int tid = omp_get_thread_num();
      printf (" ID [%d] : i : %d \n",tid,i);

    }
  }
  return EXIT_SUCCESS ;
}
```



```
zahaf:···> code$ ./loops1.out
 ID [5] : i : 7
 ID [4] : i : 6
 ID [6] : i : 8
 ID [7] : i : 9
 ID [3] : i : 5
 ID [0] : i : 0
 ID [0] : i : 1
 ID [1] : i : 2
 ID [1] : i : 3
 ID [2] : i : 4
```

Loops : a particular attention

- The iterator of a omp for loop must use additions/substractions to get to the next iteration (no other types of post conditions are supported)

Loops : a particular attention

- The iterator of a omp for loop must use additions/substractions to get to the next iteration (no other types of post conditions are supported)
- The iterator of the parallelized loop is always (not the ones if nested loops exist)

Loops : a particular attention

- The iterator of a omp for loop must use additions/substractions to get to the next iteration (no other types of post conditions are supported)
- The iterator of the parallelized loop is always (not the ones if nested loops exist)
- There is an implicit barrier at the end of the loop.
    - remove it by adding the clause nowait on the same line: `#pragma omp for nowait`

Loops : a particular attention

- The iterator of a omp for loop must use additions/substractions to get to the next iteration (no other types of post conditions are supported)
- The iterator of the parallelized loop is always (not the ones if nested loops exist)
- There is an implicit barrier at the end of the loop.
  - remove it by adding the clause nowait on the same line: `#pragma omp for nowait`

Specify iteration scheduling : `schedule(ScheduleType, chunksize)`

Loops : a particular attention

- The iterator of a omp for loop must use additions/substractions to get to the next iteration (no other types of post conditions are supported)
- The iterator of the parallelized loop is always (not the ones if nested loops exist)
- There is an implicit barrier at the end of the loop.
  - remove it by adding the clause nowait on the same line: `#pragma omp for nowait`

Specify iteration scheduling : `schedule(ScheduleType, chunksize)`

- **[static]** distributes the iteration chunks across threads in a round-robin
  - default chunksize is computer to load balance different threads
  - useful when iterations are regular

Loops : a particular attention

- The iterator of a omp for loop must use additions/substractions to get to the next iteration (no other types of post conditions are supported)
- The iterator of the parallelized loop is always (not the ones if nested loops exist)
- There is an implicit barrier at the end of the loop.
  - remove it by adding the clause nowait on the same line: `#pragma omp for nowait`

Specify iteration scheduling : `schedule(ScheduleType, chunksize)`

- **[static]** distributes the iteration chunks across threads in a round-robin
  - default chunksize is computer to load balance different threads
  - useful when iterations are regular
- **[dynamic]** (work stealing) divides the iteration space to multiple chunks. When complete a chunk, it dequeues the next one.
  - By default, chunksize is 1.
  - Very useful if the time to process individual iterations varies.

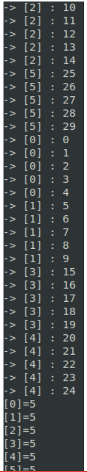
Loops scheduling (Link configuration, result)

```
# include <stdio.h>
# include <stdlib.h>
# include <omp.h>
int main(int argc, char ** argv){
  int nthreads=0;
  ;
#pragma omp parallel
  {
    nthreads= omp_get_num_threads();
  }
  int nb[nthreads];
  for (int i=0;i<nthreads;i++) nb[i]=0;
#pragma omp parallel for schedule(static, 3)
  for (int i=0;i<30;i++){
    int tid = omp_get_thread_num();
    nb[tid]++;
    printf("-> [%d] : %d \n", tid, i);
  }
  for (int i=0;i<nthreads;i++)
    printf("[%d]=%d\n", i, nb[i]);
  return EXIT_SUCCESS ;
}
```

Loops scheduling (Link configuration, result)

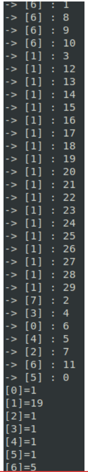
```c
# include <stdio.h>
# include <stdlib.h>
# include <omp.h>
int main(int argc, char ** argv){
  int nthreads=0;
  ;
#pragma omp parallel
  {
    nthreads= omp_get_num_threads();
  }
  int nb[nthreads];
  for (int i=0;i<nthreads;i++) nb[i]=0;
#pragma omp parallel for schedule(static, 3)
  for (int i=0;i<30;i++){
    int tid = omp_get_thread_num();
    nb[tid]++;
    printf("-> [%d] : %d \n", tid, i);
  }
  for (int i=0;i<nthreads;i++)
    printf("[%d]=%d\n", i, nb[i]);
  return EXIT_SUCCESS ;
}
```

dynamic, 5



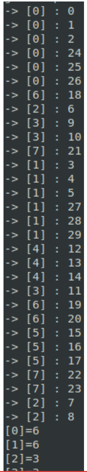
```
-> [2] : 10
-> [2] : 11
-> [2] : 12
-> [2] : 13
-> [2] : 14
-> [5] : 25
-> [5] : 26
-> [5] : 27
-> [5] : 28
-> [5] : 29
-> [0] : 0
-> [0] : 1
-> [0] : 2
-> [0] : 3
-> [0] : 4
-> [1] : 5
-> [1] : 6
-> [1] : 7
-> [1] : 8
-> [1] : 9
-> [3] : 15
-> [3] : 16
-> [3] : 17
-> [3] : 18
-> [3] : 19
-> [4] : 20
-> [4] : 21
-> [4] : 22
-> [4] : 23
-> [4] : 24
[0]=5
[1]=5
[2]=5
[3]=5
[4]=5
[5]=5
```

dynamic, 3



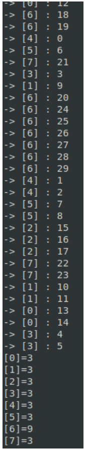
```
-> [6] : 1
-> [6] : 8
-> [6] : 9
-> [6] : 10
-> [1] : 3
-> [1] : 12
-> [1] : 13
-> [1] : 14
-> [1] : 15
-> [1] : 16
-> [1] : 17
-> [1] : 18
-> [1] : 19
-> [1] : 20
-> [1] : 21
-> [1] : 22
-> [1] : 23
-> [1] : 24
-> [1] : 25
-> [1] : 26
-> [1] : 27
-> [1] : 28
-> [1] : 29
-> [7] : 2
-> [3] : 4
-> [0] : 6
-> [4] : 5
-> [2] : 7
-> [6] : 11
-> [5] : 0
[0]=1
[1]=19
[2]=1
[3]=1
[4]=1
[5]=1
[6]=5
```

dynamic, 1



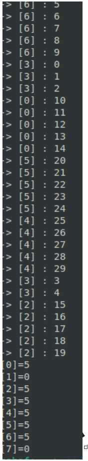
```
-> [0] : 0
-> [0] : 1
-> [0] : 2
-> [0] : 24
-> [0] : 25
-> [0] : 26
-> [6] : 18
-> [2] : 6
-> [3] : 9
-> [3] : 10
-> [7] : 21
-> [1] : 3
-> [1] : 4
-> [1] : 5
-> [1] : 27
-> [1] : 28
-> [1] : 29
-> [4] : 12
-> [4] : 13
-> [4] : 14
-> [3] : 11
-> [6] : 19
-> [6] : 20
-> [5] : 15
-> [5] : 16
-> [5] : 17
-> [7] : 22
-> [7] : 23
-> [2] : 7
-> [2] : 8
[0]=6
[1]=6
[2]=3
```

static, 5



```
-> [0] : 12
-> [6] : 18
-> [6] : 19
-> [4] : 0
-> [5] : 6
-> [7] : 21
-> [3] : 3
-> [1] : 9
-> [6] : 20
-> [6] : 24
-> [6] : 25
-> [6] : 26
-> [6] : 27
-> [6] : 28
-> [6] : 29
-> [4] : 1
-> [4] : 2
-> [5] : 7
-> [5] : 8
-> [2] : 15
-> [2] : 16
-> [2] : 17
-> [7] : 22
-> [7] : 23
-> [1] : 10
-> [1] : 11
-> [0] : 13
-> [0] : 14
-> [3] : 4
-> [3] : 5
[0]=3
[1]=3
[2]=3
[3]=3
[4]=3
[5]=3
[6]=9
[7]=3
```

static, 2



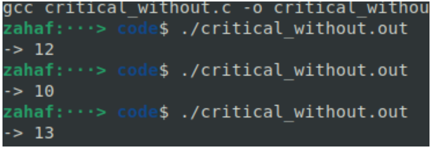
```
> [6] : 5
> [6] : 6
> [6] : 7
> [6] : 8
> [6] : 9
> [3] : 0
> [3] : 1
> [3] : 2
> [0] : 10
> [0] : 11
> [0] : 12
> [0] : 13
> [0] : 14
> [5] : 20
> [5] : 21
> [5] : 22
> [5] : 23
> [5] : 24
> [4] : 25
> [4] : 26
> [4] : 27
> [4] : 28
> [4] : 29
> [3] : 3
> [3] : 4
> [2] : 15
> [2] : 16
> [2] : 17
> [2] : 18
> [2] : 19
[0]=5
[1]=0
[2]=5
[3]=5
[4]=5
[5]=5
[6]=5
[7]=0
```

What does it do?

```
# include <stdio.h>
# include <stdlib.h>
# include <omp.h>
int main(int argc, char ** argv){
  int count=0;
#pragma omp parallel for schedule(dynamic) shared(count)
  for (int i=0;i<30;i++){
      count++;
  }

  printf("-> %d \n", count);
  return EXIT_SUCCESS ;
}
```

What does it do?

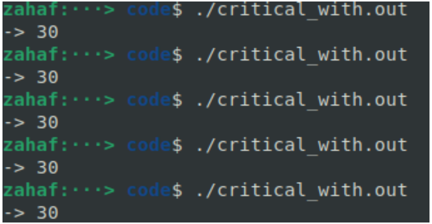
```c
# include <stdio.h>
# include <stdlib.h>
# include <omp.h>
int main(int argc, char ** argv){
  int count=0;
#pragma omp parallel for schedule(dynamic) shared(count)
  for (int i=0;i<30;i++){
      count++;
  }

  printf("-> %d \n", count);
  return EXIT_SUCCESS ;
}
```



```
gcc critical_without.c -o critical_withou
zahaf:···> code$ ./critical_without.out
-> 12
zahaf:···> code$ ./critical_without.out
-> 10
zahaf:···> code$ ./critical_without.out
-> 13
```

Critical section

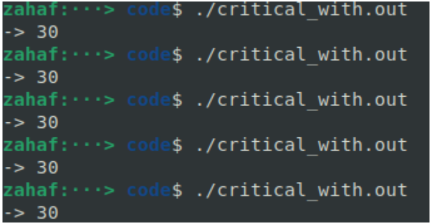
```c
# include <stdio.h>
# include <stdlib.h>
# include <omp.h>
int main(int argc, char ** argv){
  int count=0;
#pragma omp parallel for schedule(dynamic) shared(count)
for (int i=0;i<30;i++){
  # pragma omp critical
    {
      count++;
    }
  }
  printf("-> %d \n", count);
  return EXIT_SUCCESS ;
}
```



```
zahaf:···> code$ ./critical_with.out
-> 30
zahaf:···> code$ ./critical_with.out
-> 30
zahaf:···> code$ ./critical_with.out
-> 30
zahaf:···> code$ ./critical_with.out
-> 30
zahaf:···> code$ ./critical_with.out
-> 30
```

Critical section

```c
# include <stdio.h>
# include <stdlib.h>
# include <omp.h>
int main(int argc, char ** argv){
  int count=0;
#pragma omp parallel for schedule(dynamic) shared(count)
for (int i=0;i<30;i++){
  # pragma omp critical
    {
      count++;
    }
  }
  printf("-> %d \n", count);
  return EXIT_SUCCESS ;
}
```



```
zahaf:···> code$ ./critical_with.out
-> 30
zahaf:···> code$ ./critical_with.out
-> 30
zahaf:···> code$ ./critical_with.out
-> 30
zahaf:···> code$ ./critical_with.out
-> 30
zahaf:···> code$ ./critical_with.out
-> 30
```

- `pragma omp atomic` can be used for an atomic arithmetic instruction
  - if supported by hardware, low level atomic instruction execution can be generated

Other synchronization mechanisms

barrier Directive : `#pragma omp barrier`

- All threads will wait at this point
- All parallel regions have an implicit barrier.

Other synchronization mechanisms

barrier Directive : `#pragma omp barrier`

- All threads will wait at this point
- All parallel regions have an implicit barrier.

`single` Directive

- a single thread will execute the sequence of instructions located in the single region
- There is an implicit barrier at the end of the region

Other synchronization mechanisms

barrier Directive : `#pragma omp barrier`

- All threads will wait at this point
- All parallel regions have an implicit barrier.

`single` Directive

- a single thread will execute the sequence of instructions located in the single region
- There is an implicit barrier at the end of the region

`master` Directive

- only the master will execute the sequence of instructions located in the single region,
- the region will be executed only once without implicit barrier at the end of the region.

Other synchronization mechanisms

barrier Directive : `#pragma omp barrier`

- All threads will wait at this point
- All parallel regions have an implicit barrier.

`single` Directive

- a single thread will execute the sequence of instructions located in the single region
- There is an implicit barrier at the end of the region

`master` Directive

- only the master will execute the sequence of instructions located in the single region,
- the region will be executed only once without implicit barrier at the end of the region.

`nowait` Clause

- can be used on `omp for`, `single`, and `critical directives` to remove the implicit barrier they feature

Environment variables

- Under Linux: `export` VAR=VALUE

Variable Name	Default	Description
`OMP_NUM_THREADS`	Number of procs (OS)	Sets the maximum number of threads to use by parallel regions
`OMP_SCHEDULE`	STATIC	Sets the run-time schedule type and an optional chunk size
`OMP_DYNAMIC`	FALSE	Enables (.TRUE.) or disables (.FALSE.) the dynamic adjustment of the number of threads.
`OMP_NESTED`	FALSE	Enables (.TRUE.) or disables (.FALSE.)nested parallelism.
`OMP_STACKSIZE`	depend on arch.	Sets the number of bytes to allocate for each OpenMP thread to use as the private stack for the thread.
`OMP_MAX_ACTIVE_LEVELS`	No enforced limit	Limits the number of simultaneously executing threads . in an OpenMP program

Environment variables

- Under Linux: `export` VAR=VALUE

Variable Name	Default	Description
`OMP_NUM_THREADS`	Number of procs (OS)	Sets the maximum number of threads to use by parallel regions
`OMP_SCHEDULE`	STATIC	Sets the run-time schedule type and an optional chunk size
`OMP_DYNAMIC`	FALSE	Enables (.TRUE.) or disables (.FALSE.) the dynamic adjustment of the number of threads.
`OMP_NESTED`	FALSE	Enables (.TRUE.) or disables (.FALSE.)nested parallelism.
`OMP_STACKSIZE`	depend on arch.	Sets the number of bytes to allocate for each OpenMP thread to use as the private stack for the thread.
`OMP_MAX_ACTIVE_LEVELS`	No enforced limit	Limits the number of simultaneously executing threads . in an OpenMP program

Other dependant variables (Intel)

- `KMP_BLOCKTIME`, (default 200 milliseconds) : Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping.
- `KMP_LIBRARY`, (default throughput) : Selects the OpenMP run-time library execution mode. The options for the variable value are throughput, turnaround, and serial.
- ...

Plan

1. Introduction to shared-memory parallel programming model

2. Parallel programming using OpenMP

3. **Pthread as shared memory parallel programming model**

4. Exercices

Pthread as shared memory parallel programming model

- Stands for “Posix Threads”
- Posix is an IEEE standard for a Portable Operating System

Pthread as shared memory parallel programming model

- Stands for “Posix Threads”
- Posix is an IEEE standard for a Portable Operating System

- functions are spawn as separate threads
- The thread terminates when:
  - the function completes,
  - `pthread_exit`() is called

Pthread as shared memory parallel programming model

- Stands for “Posix Threads”
- Posix is an IEEE standard for a Portable Operating System

- functions are spawn as separate threads
- The thread terminates when:
  - the function completes,
  - `pthread_exit`() is called

- All threads share a single executable, a single set of global variables,
- Each thread has its own stack (function arguments, private variables)

Pthread as shared memory parallel programming model

- Stands for “Posix Threads”
- Posix is an IEEE standard for a Portable Operating System

- functions are spawn as separate threads
- The thread terminates when:
  - the function completes,
  - `pthread_exit`() is called

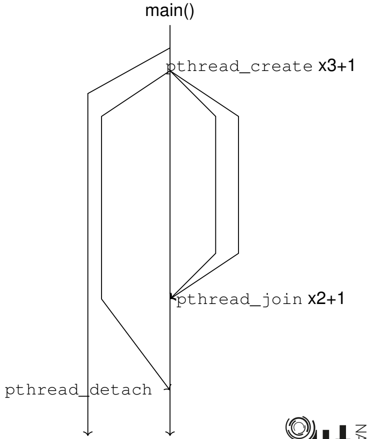
- All threads share a single executable, a single set of global variables,
- Each thread has its own stack (function arguments, private variables)

- Compared to OpenMP, it is a low-level API, indeed OpenMP is implemented on the top of Pthreads

Pthread execution model

main routines

- `int pthread_create(`
  ```
  pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *),
  void *arg);
  ```
- `int pthread_join(pthread_t thread, void **retval);`
- `void pthread_exit(void *retval);`
- `int pthread_detach(pthread_t thread);`



Minimal Example using pthread

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *foo(void *arg){
  sleep(1);
  printf("Hello, I am a thread \n");
  return NULL;
}

int main(){
  pthread_t thread_id;
  printf("Before invoking the thread function\n");
  pthread_create(&thread_id, NULL, foo, NULL);
  pthread_join(thread_id, NULL);
  printf("After Thread\n");
}
```

Scheduling pthreads : the Linux case

- Specified through `pthread_attr_t`

Scheduling pthreads : the Linux case

- Specified through `pthread_attr_t`

```
#include <pthread.h>
...
int pthread_attr_setschedpolicy(pthread_attr_t *att, int policy);
```

Scheduling pthreads : the Linux case

- Specified through `pthread_attr_t`

```
#include <pthread.h>
...
int pthread_attr_setschedpolicy(pthread_attr_t *att, int policy);
```

- `att` : Scheduling attributes (the priority)

Scheduling pthreads : the Linux case

- Specified through `pthread_attr_t`

```
#include <pthread.h>
...
int pthread_attr_setschedpolicy(pthread_attr_t *att, int policy);
```

- `att` : Scheduling attributes (the priority)
- `policy` : Scheduling policy : `SCHED_FIFO`, `SCHED_RR`, `SCHED_DEAD`

Scheduling pthreads : the Linux case

- Specified through `pthread_attr_t`

```
#include <pthread.h>
...
int pthread_attr_setschedpolicy(pthread_attr_t *att, int policy);
```

- `att` : Scheduling attributes (the priority)
- `policy` : Scheduling policy : `SCHED_FIFO`, `SCHED_RR`, `SCHED_DEAD`
- It is required to be **super-user** to apply the real-time scheduling policies
  - → Attention : A thread with the highest RT priority will never be preempted.

Steps before creating pthreads

- Highest priority thread is scheduled first
- Priority is specified at thread creation time
  → `pthread_attr_t` et `struct sched_param`

E1. create thread id structures and scheduling parameters :
  - `pthread_t th;` `pthread_attr_t my_attr;` `struct sched_param param;`

E2. `pthread_attr_init(&my_attr)` → initialization of the attribute

E3. `pthread_attr_setschedpolicy(&my_attr, SCHED_FIFO);` → select the scheduler

E4. `param.sched_priority = 1;` → Set the priority

E5. `pthread_attr_setschedparam(&my_attr, &param);` → Link scheduling parameters and attributes

E6. `pthread_create(&th1, &my_attr, foo, 0);` → launch the threads.

E7. `pthread_attr_destroy(&my_attr);` → destroying the parameters

Sched-other (Round-Robin fair scheduler)

- Not based on the priority

Sched-other (Round-Robin fair scheduler)

- Not based on the priority
- Each task is served for a quantum

Sched-other (Round-Robin fair scheduler)

- Not based on the priority

- Each task is served for a quantum

- Quantum is defined based on the thread execution, and the load of the system:
    - Very small quantum: system overloading due to the important number of context switches
    - Very long quantum: system reactivity is compromised
    - Implemented through the management of two run-queues : served, and not-yet-served

Sched-other (Round-Robin fair scheduler)

- Not based on the priority
- Each task is served for a quantum
- Quantum is defined based on the thread execution, and the load of the system:
    - Very small quantum: system overloading due to the important number of context switches
    - Very long quantum: system reactivity is compromised
    - Implemented through the management of two run-queues : served, and not-yet-served
- Calling routine `nice()`, and `setpriority()` increases the quantum

Race condition and mutual exclusion

Race condition and mutual exclusion

- The more threads are independent, the better.
- BUT : in parallel programming, they share resources and communicate

Race condition and mutual exclusion

- The more threads are independent, the better.
- BUT : in parallel programming, they share resources and communicate

Race condition and mutual exclusion

- The more threads are independent, the better.
- BUT : in parallel programming, they share resources and communicate

Objective

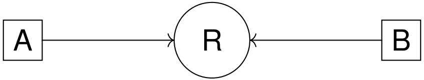
- Guarantee that concurrency and communications will not impact the system correctness (similarly as in OpenMP)

Race condition and mutual exclusion

- The more threads are independent, the better.
- BUT : in parallel programming, they share resources and communicate

Objective

- Guarantee that concurrency and communications will not impact the system correctness (similarly as in OpenMP)

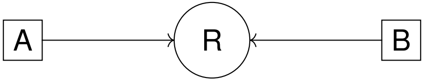


Race condition and mutual exclusion

- The more threads are independent, the better.
- BUT : in parallel programming, they share resources and communicate

Objective

- Guarantee that concurrency and communications will not impact the system correctness (similarly as in OpenMP)



Constraints

- Guarantee that a task will have access to shared resources
- Ensure that the access to the shared resource is controlled

Solution : Mutex

The problem to solve

Consider thread A and B sharing a buffer. A produces the buffer and B consumes it. We would like to guarantee that if A is producing data, therefore B can not consume it, and vice-versa.

```
mutex m;
```

```
while(1) {
  //  proccessing 3
  p(mutex);
  for (int i=0;i<buffer_size;i++){
    buffer[i] = function(i);
  } // ressource partagée
  v(mutex);
}
```

Tâche A

```
while(1) {
   // proccessing 1
   p(mutex);
   process(buffer)
   v(mutex);
   // processing 2

 }
```

Tâche B

Revised solution

Issues

- The solution depends on the duration of function `function()` and function `process`

→ General rule : reduce at maximum the duration of critical section

```
while(1) {
  //  proccessing 3
  for (int i=0;i<buffer_size;i++){
    buffer_1[i] = function(i);
  } // ressource partagée
  p(mutex_1);
  swap(buffer_1,buffer);
  v(mutex_1);
}
```

```
while(1) {
  // proccessing 1
  p(mutex_1);
  swap(buffer,buffer_2);
  v(mutex_1);

  process(buffer_2)

 }
```

Mutex with pthread

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Semaphores and other forms of synchronization

Semaphores and other forms of synchronization

Semaphores

```
nt sem_init(sem_t *semaphore, int pshared, unsigned int valeur);
nt sem_wait(sem_t *semaphore);
nt sem_timedwait(sem_t *semaphore, const struct timespec *abs_timeout);
nt sem_post(sem_t *semaphore);
```

Semaphores and other forms of synchronization

Semaphores

```
nt sem_init(sem_t *semaphore, int pshared, unsigned int valeur);
nt sem_wait(sem_t *semaphore);
nt sem_timedwait(sem_t *semaphore, const struct timespec *abs_timeout);
nt sem_post(sem_t *semaphore);
```

Barrier

```
 int pthread_barrier_destroy(pthread_barrier_t *barrier);
 int pthread_barrier_init(pthread_barrier_t *restrict barrier,
 const pthread_barrierattr_t *restrict attr, unsigned count);
```

Semaphores and other forms of synchronization

Semaphores

```
nt sem_init(sem_t *semaphore, int pshared, unsigned int valeur);
nt sem_wait(sem_t *semaphore);
nt sem_timedwait(sem_t *semaphore, const struct timespec *abs_timeout);
nt sem_post(sem_t *semaphore);
```

Barrier

```
 int pthread_barrier_destroy(pthread_barrier_t *barrier);
 int pthread_barrier_init(pthread_barrier_t *restrict barrier,
 const pthread_barrierattr_t *restrict attr, unsigned count);
```

Spin-locks

- Spin-locks : active wait : thread is kept in active state

Other useful pthreads functions

- pthread_detach(pthread_t thread); Detach a thread
- pthread_exit(address_t value);Terminate this thread, returning value to any thread that is waiting for it
- pthread_cancel(pthread_t thread);Cancel a thread
- pthread_kill(pthread_t thread, int sig);Send a signal to a thread (e.g., SIGINT, SIGKILL)
- pthread_self(( ) Returns the thread id of this thread
- pthread_equal(pthread_t id1, pthread_t id2)Tells you if two thread ids refer to the same thread. It returns 0 (false) or !0 (true).
- pthread_once_t inits and pthread_once_init(&inits);

Pthread affinity: by example

- `int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset);`
  - first parameter is the pid, 0 = calling thread
  - second parameter is the size of your cpuset
  - third param is the cpuset in which your thread will be placed. Each bit represents a CPU.

Pthread affinity: by example

- `int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset);`
  - first parameter is the pid, 0 = calling thread
  - second parameter is the size of your cpuset
  - third param is the cpuset in which your thread will be placed. Each bit represents a CPU.

```
cpu_set_t cpuset;
int cpu = 2;

CPU_ZERO(&cpuset);
CPU_SET( cpu , &cpuset);
sched_setaffinity(0, sizeof(cpuset), &cpuset);
```

Thread input data passing by example

- Manipulated data are passed using the last parameter of function `pthread_create`

  ```
  int pthread_create( ..., void *restrict arg);
  ```

  - If multiple parameters are to be passed → must be regrouped in a **struct** or defined as **global**

Thread input data passing by example

- Manipulated data are passed using the last parameter of function `pthread_create`

  `int pthread_create( ..., void *restrict arg);`

  - If multiple parameters are to be passed → must be regrouped in a **struct** or defined as **global**

```
#include ...

int a;
int b;

v8DEoid *foo(void *arg){
  sleep(1);
  printf("Hello, I am a thread %d %d \n", a, b);
  return NULL;
}
int main(){
  a=1;
  b=2;
  pthread_t thread_id;
  printf("Before invoking the thread function\n");
  pthread_create(&thread_id, NULL, foo, NULL);
  pthread_join(thread_id, NULL);
  printf("After Thread\n");
}
```

```
#include ...
struct couple {
  int a;
  int b;
};
void *foo(void *arg){
  struct couple * c = (struct couple *) (arg);
  sleep(1);
  printf("Hello, I am a thread %d %d \n", c->a, c->b);
  return NULL;
}
int main(){
  struct couple c={.a = 1, .b=2};
  pthread_t thread_id;
  printf("Before invoking the thread function\n");
  pthread_create(&thread_id, NULL, foo, &c);
  pthread_join(thread_id, NULL);
  printf("After Thread\n");
}
```

Example of parallel computations : data parallelism

- Array addition using Pthreads

Plan

1. Introduction to shared-memory parallel programming model

2. Parallel programming using OpenMP

3. Pthread as shared memory parallel programming model

4. **Exercices**

Exercise Work-stealing with OpenMP

define a work stealing mechanism for Pthreads to compute array addition.

```c
long num_steps = 10000;
double step;
int main ()
{ int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i=0;i<num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
  printf("%lf \n", pi);
}
```

- Create a parallel version of the pi program using a parallel construct, without any use of synchronization mechanisms (without parallelizing loops with the help of open mp).
- Improve the previous solution using synchronization mechanisms
- use parallel loops with openMP, improve the solving time using the OpenMP schedules