OpenMP

Cédric Bastoul
cedric.bastoul@unistra.fr

University of Strasbourg – French-Azerbaijani University

Using the Course Material

This course material is built as a portal

- Links to the outside or the inside of the document
  - Follow them for more details
  - To go back in the navigation history: click on ◁
- Codes provided through a tag [code]
  - Tags are located at the top right of the code or of the page
  - Do not copy codes from the PDF document
- Details of each concept provided through a tag [doc]
  - Link to the correct page of the OpenMP standard document
  - Consult it in case of a doubt or if the material is not enough

References

This course mainly refers at the following materials:

- OpenMP 5.0 Standard

  www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
- R. van der Pas's OpenMP course at Oracle

  www.openmp.org/wp-content/uploads/ntu-vanderpas.pdf
- J. Chergui and P.-F. Lavallée's OpenMP course at IDRIS

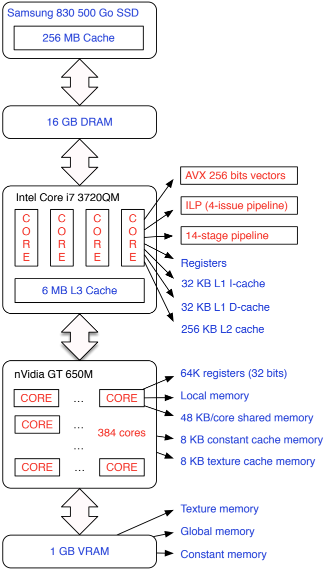
  www.idris.fr/media/eng/formations/openmp/idris_openmp_cours-eng-v2.9.pdf
- B. Barney's OpenMP tutorial

  computing.llnl.gov/tutorials/openMP
- B. Chapman, G. Jost and R van der Pas *Using OpenMP* MIT Press, 2007
- R. van der Pas, E. Stotzer and C. Terboven *Using OpenMP - The Next Step* MIT Press, 2017

Review of a Modern Architecture

This computer:

- 2.7 billion transistors
- **17 types of memory**
- 5 types of parallelism
- At least 4 programming models + API



"Open Multi-Processing" standard API (Application Programming Interface) for programming *parallel* applications on *shared memory architectures*

- Thread-based programming
- Directives to target vector units (OpenMP 4+)
- Directives to target accelerators (OpenMP 4+)

- Mature and widespread industrial standard
- Good performance if we do it well
- Minimal programming effort
- Portable

History

SMP Architectures ("Symmetric Multiprocessing" or "Shared Memory multiProcessor") since IBM System/360 model 67 (1966), programed using proprietary directives at first

- 1991 The Parallel Computing Forum industrial group defines a set of directives for parallel loops in Fortran (never normalized)
- 1997 OpenMP 1.0 for Fortran, C/C++ in 1998
- 2000 OpenMP 2.0 Fortran 95 constructs
- 2008 OpenMP 3.0 tasks
- 2013 OpenMP 4.0 SIMD, accelerators, etc.
- 2015 OpenMP 4.5 data mapping, doacross, etc.
- 2018 OpenMP 5.0 accelerators, performance analysis

Principle

Add directives to the code to tell the compiler:

- Which are the instructions to run in parallel
- How to distribute instructions and data amongst threads

- Directives are often optional (the program is semantically equivalent with or without them)
- Parallelism detection or extraction is left to the programmer
- Impact on the original sequential code is often limited
- But to get speedups you have to work a little bit!

Positioning

Main parallel programming models:

- Shared memory architectures
  - *Intrinsics* assembly vector instructions (Intel SSE2, ARM NEON), very low level
  - *Posix Threads* standardized library, low level
  - ***OpenMP*** De facto standard API
  - *CUDA* proprietary framework for accelerators
  - *OpenCL* API and language for SMP + accelerators
- Distributed memory architectures
  - *Sockets* standardized library, low level
  - ***MPI*** Message Passing Interface, de facto standard library for distributed memory architectures (works also on SMP), major code modifications

Example: Sequential Dot Product

[code]

```c
#include <stdio.h>
#define SIZE 256

int main() {
  double sum, a[SIZE], b[SIZE];

  // Initialization
  sum = 0.;
  for (size_t i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  for (size_t i = 0; i < SIZE; i++)
    sum = sum + a[i]*b[i];

  printf("sum␣=␣%g\n", sum);
  return 0;
}
```

Example: MPI Dot Product

[code]

```
#include <stdio.h>
#include "mpi.h"
#define SIZE 256

int main(int argc, char* argv[]) {
  int numprocs, my_rank, my_first, my_last;
  double sum, sum_local, a[SIZE], b[SIZE];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  my_first = my_rank * SIZE/numprocs;
  my_last = (my_rank + 1) * SIZE/numprocs;

  // Initialization
  sum_local = 0.;
  for (size_t i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  for (size_t i = my_first; i < my_last; i++)
    sum_local = sum_local + a[i]*b[i];
  MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  if (my_rank == 0)
    printf("sum␣=␣%g\n", sum);
  MPI_Finalize();
  return 0;
}
```

Example: Pthreads Dot Product

[code]

```c
#include <stdio.h>
#include <pthread.h>
#define SIZE 256
#define NUM_THREADS 4
#define CHUNK SIZE/NUM_THREADS

int id[NUM_THREADS];
double sum, a[SIZE], b[SIZE];
pthread_t tid[NUM_THREADS];
pthread_mutex_t mutex_sum;

void* dot(void* id) {
  size_t i;
  int my_first = *(int*)id * CHUNK;
  int my_last = (*(int*)id + 1) * CHUNK;
  double sum_local = 0.;

  // Computation
  for (i = my_first; i < my_last; i++)
    sum_local = sum_local + a[i]*b[i];

  pthread_mutex_lock(&mutex_sum);
  sum = sum + sum_local;
  pthread_mutex_unlock(&mutex_sum);
  return NULL;
}
```

```c
int main() {
  size_t i;

  // Initialization
  sum = 0.;
  for (i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  pthread_mutex_init(&mutex_sum, NULL);

  for (i = 0; i < NUM_THREADS; i++) {
    id[i] = i;
    pthread_create(&tid[i], NULL, dot,
                   (void*)&id[i]);
  }

  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

  pthread_mutex_destroy(&mutex_sum);

  printf("sum = %g\n", sum);
  return 0;
}
```

Reminder: Sequential Dot Product

[code]

```
#include <stdio.h>
#define SIZE 256

int main() {
  double sum, a[SIZE], b[SIZE];

  // Initialization
  sum = 0.;
  for (size_t i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  for (size_t i = 0; i < SIZE; i++)
    sum = sum + a[i]*b[i];

  printf("sum␣=␣%g\n", sum);
  return 0;
}
```

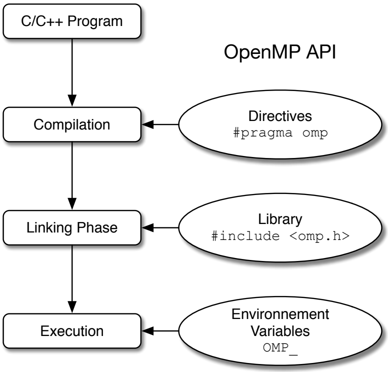
Example: OpenMP Dot Product

[code]

```c
#include <stdio.h>
#define SIZE 256

int main() {
  double sum, a[SIZE], b[SIZE];

  // Initialization
  sum = 0.;
  for (size_t i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  #pragma omp parallel for reduction(+:sum)
  for (size_t i = 0; i < SIZE; i++) {
    sum = sum + a[i]*b[i];
  }

  printf("sum␣=␣%g\n", sum);
  return 0;
}
```

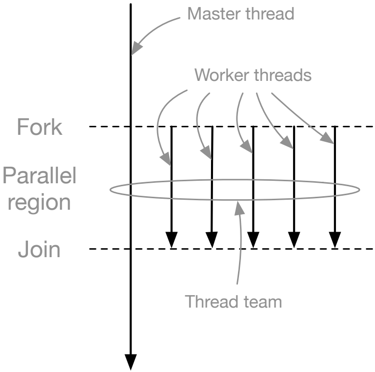
OpenMP API

1. *Directives* to explicit parallelism, synchronizations and scope of the data (private, shared ...)
2. *Library* for specific features (dynamic information, runtime management...)
3. *Environment Variables* to control the program execution (number of threads, scheduling strategies...)



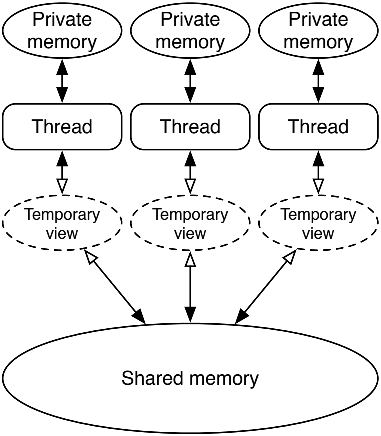
OpenMP Execution Model

1. The user inserts directives to specify *parallel regions*
2. At runtime, parallel regions execute according to the *fork-join* model:
   - The master thread creates worker threads, and teams up with them
   - The team threads synchronize at the end of the parallel region, then the worker threads terminate
   - The master thread continues its execution



OpenMP Memory Model

- All threads can access to the same *shared memory*
- Each thread has its own *private memory*
- Shared data may be accessed by all threads simultaneously
- Private data may be accessed only by the owner thread
- Data transfer is transparent to the programmer



OpenMP Directives

Main OpenMP Directives

- Parallel region creation
  - `parallel`: create a "fork-join" parallel region
- Work-sharing
  - `for`: share iterations of a parallel loop
  - `sections`: specify blocks to be executed in parallel
  - `single`: specify a block to be executed by a single thread
- Synchronization
  - `master`: block to be executed by the master thread
  - `critical`: block to be executed one thread at a time
  - `atomic`: statement with atomic access to storage location
  - `barrier`: wait until all threads of the team reach that point
- Task management
  - `task`: create a task to be added to the team pool
  - `taskwait`: wait for the tasks in the team pool to complete

OpenMP Directives

C/C++ directive format [doc]

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

Composed of four parts:

1. Sentinel: `#pragma omp`
2. Valid directive name
3. Optional list of *clauses* (providing additional information)
4. Carriage return

General rules:

- Case sensitive (caution!)
- Directives apply to the directly following code block
- Long directives can be continued to a new line by putting a backslash "`\`" character at the end of the previous line

OpenMP Directives

Parallel Region Construction

`parallel` Directive (1/2)

C/C++ `parallel` directive format [doc]

```
#pragma omp parallel [clause [clause] ...]
{
  // Parallel region
}
```

How it works:

- When a thread reaches a `parallel` directive, it creates a thread team and becomes the master thread of that team with number 0; all threads in the team execute the block
- The number of threads in the team depends, in that order, of the evaluation of the `if` clause, the `num_threads` clause, the `omp_set_num_threads()` function, the `OMP_NUM_THREADS` environment variable, the default value
- There is an implicit barrier at the end of the parallel region

`parallel` Directive (2/2)

- By default, the scope of the variables declared before the parallel region is **shared** in that region
- The scope of the variables declared within the parallel region is **private** in that region
- Branching out or into the parallel region (`goto`) is forbidden
- Accepted clauses: `if`, `num_threads`, `private`, `shared`, `default`, `firstprivate`, `reduction`, `copyin`

if Clause

C/C++ if clause format [doc]

```
if (/* Scalar expression */)
```

- When this clause is present, a thread team is created only if the scalar expression is non-zero, the region is executed sequentially by the master thread otherwise

Usecase example of the if clause

```
#include <stdio.h>
#define PARALLEL 1 // 0 for sequential, != 0 for parallel

int main() {
  #pragma omp parallel if (PARALLEL)
  printf("Hello, openMP\n");
  return 0;
}
```

num_threads Clause

C/C++ num_threads clause format [doc]

```
num_threads(/* Integer expression */)
```

- Specify the number of threads of the team that will execute the next parallel region
- The integer expression must evaluate to a positive integer value

Exercise

A first example [code]

```
#include <stdio.h>

int main() {
  #pragma omp parallel
   printf("Hello,\n");
   printf("world\n");
  return 0;
}
```

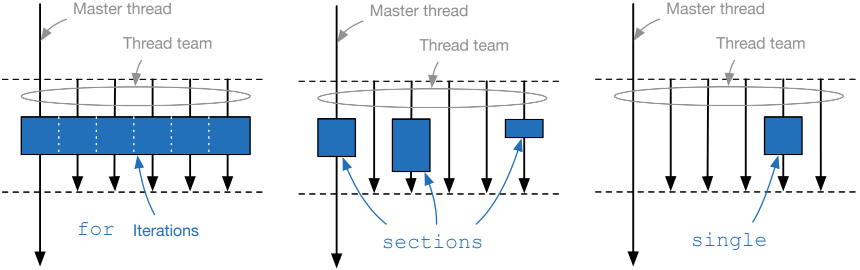
- Compile this program without and with the `-fopenmp` option
- Run this program in both cases
- What is the default number of threads? Is it reasonable?
- Modify the number of threads used to run your program

OpenMP Directives

Work Sharing

Work Sharing Directives

- Included in parallel regions
- Must be reached by all threads or none
- Divide work among different threads
- Imply a barrier at the end of the construct (except if the `nowait` clause is specified)



`for` : divide the iterations of a parallel loop

`sections` : divide according to user-defined code blocks

`single` : only one thread is in charge of the code block

`for` Directive

C/C++ `for` directive format [doc]

```
#pragma omp for [clause [clause] ...]
for (...)
  ...
```

- Specify that the iterations of the loop that follows the directive must be executed in parallel by the thread team
- The iteration variable is private by default
- Target loops must have a simple iterative form [doc]
  - Loop bounds must be the same for all threads
  - Infinite or `while` loops are not supported
- Reminder: the programmer is responsible for the semantics
- Accepted clauses: `schedule`, `ordered`, `private`, `firstprivate`, `lastprivate`, `reduction`, `collapse`, `nowait`

Exercise

- Write a C program that performs the sum of each element of an array with a scalar in a second array
- Parallelize that program with OpenMP

`schedule` Clause (1/2)

C/C++ `schedule` clause format [doc]

```
schedule(type[, chunk])
```

- Specify the iteration distribution policy
- 5 possible types [doc] :

static Iterations are divided into consecutive blocks of `chunk` iterations; blocks are assigned to threads in a round-robin fashion; if `chunk` is not specified, blocks of similar sizes are created, one per thread

dynamic Each thread requests a block of `chunk` consecutive iterations as soon as it finishes its workload (the last block may be smaller); if `chunk` is not specified, it is set to 1

`schedule` Clause (2/2)

- **guided** Same as `dynamic` but the chunk size decreases exponentially; if `chunk` is more than 1, it is the minimum number of iterations in a chunk (except for the last one)
- **runtime** The choice of the policy is postponed until the execution time where it is set for example using the environment variable `OMP_SCHEDULE`
- **auto** The choice of the policy is left to the compiler and/or the runtime

- The choice of the policy is critical for performance

Exercice (1/2)

Read, study, compile and run the following code [code]

```c
#include <stdio.h>
#include <omp.h>
#define SIZE  100
#define CHUNK  10

int main() {
  int tid;
  double a[SIZE], b[SIZE], c[SIZE];

  for (size_t i = 0; i < SIZE; i++)
    a[i] = b[i] = i;

  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    if (tid == 0)
      printf("Nb threads = %d\n", omp_get_num_threads());
    printf("Thread %d: starting...\n", tid);

    #pragma omp for schedule(dynamic, CHUNK)
    for (size_t i = 0; i < SIZE; i++) {
      c[i] = a[i] + b[i];
      printf("Thread %d: c[%2zu] = %g\n", tid, i, c[i]);
    }
  }
  return 0;
}
```

Exercice (2/2)

- Analyze the program: which are the instructions executed by all threads? By a single thread?
- Run the program several times. What do you think about the execution order of the instructions?
- Redirect the output of the executable to the `sort` utility. Run and observe the iteration distribution.
- Repeat several times. Is the distribution stable?
- Change the scheduling policy to `static`. Repeat several times. Is the distribution stable?
- Discuss the impact of the scheduling policy on performance.

collapse Clause

C/C++ collapse clause format [doc]

```
collapse(/* Strictly positive integer expression */)
```

- Specify the number of loops associated to the directive `for` (default value: 1)
- If the integer expression is greater than 1, the iterations of associated loops are grouped together to form a single iteration space that will be distributed among the threads
- The order of the iterations of the grouped loop is the same as the order of the iterations of the original loops

```
#pragma omp for collapse(2)
for (i = 0; i < 10; i++)
  for (j = 0; j < 10; j++)
    f(i, j);
```

$\equiv$

```
#pragma omp for
for (i = 0; i < 100; i++)
  f(i/10, i%10);
```

Clause `nowait`

C/C++ `nowait` clause format

```
nowait
```

- Removes the implied barrier at the end of the work sharing construct
- Threads finishing their workload can continue after the work sharing construct without waiting for other threads
- The programmer must ensure that the program semantics is preserved

Exercice (1/2)

Read, study, compile and run the following code [code]

```
#include <stdio.h>
#define SIZE 100

int main() {
  double a[SIZE], b[SIZE], c[SIZE], d[SIZE];

  for (size_t i = 0; i < SIZE; i++)
    a[i] = b[i] = i;

  #pragma omp parallel
  {
    #pragma omp for schedule(static) nowait
    for (size_t i = 0; i < SIZE; i++)
      c[i] = a[i] + b[i];

    #pragma omp for schedule(static)
    for (size_t i = 0; i < SIZE; i++)
      d[i] = a[i] + c[i];
  }

  for (size_t i = 0; i < SIZE; i++)
    printf("%g␣", d[i]);
  printf("\n");
  return 0;
}
```

Exercice (2/2)

- Run the program several times. Do the results seem inconsistent?
- Analyze the program: which iterations will be executed by which threads (the documentation of the `static` policy in the OpenMP standard will help you [doc]) ?
- After analysis, does the use of the `nowait` clause seem reasonable to you?
- Change the second loop scheduling policy to `guided`.
- Run the program several times. Do the results seem inconsistent? If you have not seen the problem, look better ;-)!

sections Directive

C/C++ sections directive format [doc]

```
#pragma omp sections [clause [clause] ...]
{
  #pragma omp section
  // Block 1
  ...
  #pragma omp section
  // Block N
}
```

- Specify that the instructions in the various sections must be executed in parallel by the thread team
- Each section is executed only once
- Sections must be defined in the static scope
- Accepted clauses: `private`, `firstprivate`, `lastprivate`, `reduction`, `nowait`

`single` Directive

C/C++ `single` directive format [doc]

```
#pragma omp single [clause [clause] ...]
{
  // Block
}
```

- Specify that the code block following the directive will be executed by a single thread
- No way to predict which thread will execute the block
- Useful for non-thread-safe code parts (e.g., I/O)
- Accepted clauses: `private`, `firstprivate`, `copyprivate`, `nowait`

`parallel for/sections` Shortcuts

C/C++ `parallel for` directive format

```
#pragma omp parallel for [clause [clause] ...]
for (...)
  ...
```

C/C++ `parallel sections` directive format

```
#pragma omp parallel sections [clause [clause] ...]
{
  #pragma omp section
  // Block 1
  ...
  #pragma omp section
  // Block N
}
```

- Create a parallel region with a single construct
- Accepted clauses: union of clauses except `nowait`

Orphaned Directives

The parallel region is bound to the directly following code block (*static extent*) and the functions called inside (*dynamic extent*)

- Directives outside the static extent said to be "orphaned"
- Bound to the parallel region that directly executes them
- Ignored at runtime if not bound to a parallel region

Orphaned omp for [code]

```c
#include <stdio.h>
#define SIZE 1024

void init(int* vec) {
  size_t i;
  #pragma omp for
  for (i = 0; i < SIZE; i++)
    vec[i] = 0;
}

int main() {
  int vec[SIZE];
  #pragma omp parallel
  init(vec);
  return 0;
}
```

Nested Directives

It is possible to nest parallel regions

- Implementation may ignore internal regions
- Supposedly arbitrary nesting level
- Be careful about performance

Nested directives [code]

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

int main() {
  omp_set_nested(1);
  #pragma omp parallel num_threads(2)
  {
    #pragma omp parallel num_threads(2)
    printf("Hello,␣world\n");
  }
  return 0;
}
```

OpenMP Directives

Data Scope Attribute Clauses

Data Scope Attribute Clauses

- OpenMP targets shared memory architectures; most variables are shared by default
- We can control data scoping
  - Which data from sequential regions are transferred to parallel regions and how
  - Which data are shared by all threads or private to a thread
- Main clauses:
  - `private`: specifies a list of private variables
  - `firstprivate`: `private` with automatic initialization
  - `lastprivate`: `private` with automatic update
  - `shared`: specifies a list of shared variables
  - `default`: change the default data scoping
  - `reduction`: specifies a list of reduction variables

`private` Clause

C/C++ `private` clause format [doc]

```
private(/* List of variables */)
```

- Specifies a list of variables to declare in private memory
- There is no link with the original variables
- All references in the parallel region are to the private variables
- A thread can not access private variables of another thread
- Changes to a private variable are visible only to the owner thread
- Initial and final values are undefined

Exercise

Usecase example of the `private` clause [code]

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

int main() {
  int val;
  #pragma omp parallel private(val)
  {
    val = rand();
    sleep(1);
    printf("My val : %d\n", val);
  }
  return 0;
}
```

- Compile and run this code with and without `private(val)`
- What do you observe and why?
- Is it risky even with the clause `private` ?

`firstprivate` Clause

C/C++ `firstprivate` clause format [doc]

```
firstprivate(/* List of variables */)
```

- Combine the behavior of the `private` clause with automatic initialization
- Variables in the list are initialized with the value of their corresponding original variable at the time of entry into the parallel region

`lastprivate` Clause

C/C++ `lastprivate` clause format [doc]

```
lastprivate(/* List of variables */)
```

- Combine the behavior of the `private` clause with an automatic update of the original variables at the end of the parallel region
- Variables in the list are updated with the value of the corresponding private variable at the end of the thread that executes either the last iteration of a loop or the last section according to sequential execution

`shared` Clause

C/C++ `shared` clause format [doc]

```
shared(/* List of variables */)
```

- Define a list of variables to be placed in shared memory
- There is only one instance of each shared variable
- All threads in the team can access shared variables simultaneously (unless an OpenMP directive forbids it, such as `atomic` or `critical`)
- Changes to a shared variable are visible to all threads in the team (but not always immediately, unless an OpenMP directive specifies it, such as `flush`)

`default` Clause

C/C++ `default` clause format [doc]

```
default(shared | none)
```

- Allows the user to change the default scoping of the variables of the parallel region (except local and automatic variables of the called functions)
- Choosing `none` requires the programmer to specify the scoping of each variable
  - Nice to avoid variables shared by mistake

`reduction` Clause

C/C++ `reduction` clause format [doc]

```
reduction(operator: /* list of variables */)
```

- Achieve a reduction on the variables of the list
- A private copy of each variable in the list is created for each thread; at the end of the construction, the reduction operator is applied to the private variables and the result is stored in the corresponding shared variable
- `operator` may be `+`, `-`, `*`, `&`, `|`, `^`, `&&` or `||`
- Variables in the list must be shared
- Variables int he list can only be used in instructions with restricted form (see OpenMP standard, page 167)
- Be careful about numerical stability

Exercise

- Write a program C computing the sum of the elements of an array.
- Parallelize this program using OpenMP.
- Compare sequential execution time and parallel execution time.

threadprivate Directive

C/C++ threadprivate directive format [doc]

```
// Declaration of global and/or static variables
#pragma omp threadprivate(/* List of global/static variables */)
```

- Specify that variables in the list are private *and persistent* to each thread through multiple parallel regions
- The value of the variables is not specified in the first parallel region unless the clause `copyin` is used
- Then, the variables are preserved
- The directive must follow the declaration of the target global/static variables
- The number of threads must remain constant (`omp_set_dynamic(0)`)
- Accepted clauses: none

Exercise

Study, run this code and discuss the results [code]

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

int tid, tprivate, rprivate;
#pragma omp threadprivate(tprivate)

int main() {
  // Explicitly forbid to dynamically change the number of threads
  omp_set_dynamic(0);

  printf("Parallel␣region␣#1\n");
  #pragma omp parallel private(tid, rprivate)
  {
    tid = omp_get_thread_num();
    tprivate = tid;
    rprivate = tid;
    printf("Thread␣%d:␣tprivate=%d␣rprivate=%d\n", tid, tprivate, rprivate);
  }

  printf("Parallel␣region␣#2\n");
  #pragma omp parallel private(tid, rprivate)
  {
    tid = omp_get_thread_num();
    printf("Thread␣%d:␣tprivate=%d␣rprivate=%d\n", tid, tprivate, rprivate);
  }
  return 0;
}
```

`copyin` Clause

C/C++ `copyin` clause format [doc]

```
copyin(/* List of threadprivate variables */)
```

- Specify that the values of the master thread `threadprivate` variables in the list should be copied to the corresponding private variables of the worker threads at the beginning of the parallel region

`copyprivate` Clause

C/C++ `copyprivate` clause format [doc]

```
copyprivate(/* List of private variables */)
```

- Request to copy the values of the private variables of a thread to the corresponding private variables of the other threads of the team
- Usable only with the directive `single`
- Can not be used in conjunction with the `nowait` clause
- The copy takes place after the execution of the block associated with the `single` directive and before releasing the barrier at the end of the block

OpenMP Directives

Synchronization

Typical Mistake: *Data Race*

Potential data race [code]

```c
#include <stdio.h>
#define MAX 10000

int main() {
  size_t i;
  int n = 0;

  #pragma omp parallel for
  for (i = 0; i < MAX; i++)
    n++;

  printf("n = %d\n", n);
  return 0;
}
```

At runtime, this program may display a value less than `MAX`:

- Concurent accesses to `n`
- Non atomic increment
- "Lost" increments

Thread A	Thread B
`load  R1, @n`	`load  R1, @n`
`add   R1, 1`	`add   R1, 1`
`store @n, R1`	`store @n, R1`

Thread B interrupts thread A
Thread A overwrites the value written by B

Typical Mistake: Coherence Issue

Coherence issue [code]

```c
#include <stdio.h>

int main() {
  int done = 0;

  #pragma omp parallel sections
  {
    #pragma omp section
    {
      while (!done)
        printf("Not␣done\n");
    }
    #pragma omp section
    {
      done = 1;
      printf("Done\n");
    }
  }
  return 0;
}
```

At runtime, this program may display "`Not done`" before "`Done`":

- Interruption of the first section between the `done` evaluation and the `printf` (data race)
- Usage of an obsolete shared memory temporary view by the thread running the first section (coherence issue)

Typical Mistake: Synchronization Issue

Synchronization issue [code]

```c
#include <stdio.h>
#include <omp.h>
int main() {
  double total, part1, part2;
  #pragma omp parallel \
               num_threads(2)
  {
    int tid;
    tid = omp_get_thread_num();
    if (tid == 0)
      part1 = 25;
    if (tid == 1)
      part2 = 17;
    if (tid == 0) {
      total = part1 + part2;
      printf("%g\n", total);
    }
  }
  return 0;
}
```

At runtime, this program may display a value different that 42

- Thread 0 does not wait for thread 1 before computing and displaying

Synchronization Mechanisms

- *Barrier* to wait until all threads have reached a given point
  - Implicit at the end of OpenMP constructs (except `nowait`)
  - `barrier` directive
- *Ordering* to guarantee a global execution order
  - `ordered` clause
- *Mutual exclusion* to ensure that only one task at a time executes a given code block
  - `critical` directive
  - `atomic` directive
- *Assignment* to assign a code block to a given thread
  - `master` directive
- *Lock* to schedule the execution of at least two threads
  - OpenMP library functions

`barrier` Directive

C/C++ `barrier` directive format [doc]

```
#pragma omp barrier
```

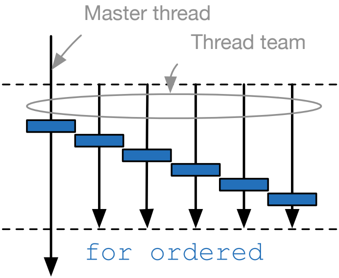
- Synchronization between all threads of a team
- When a thread reaches a `barrier` directive, it waits for all the other threads to reach it; when that happens, all threads resume their execution
- Must be reached by all threads or none: be careful about deadlocks
- Accepted clauses: none

ordered Directive

C/C++ ordered directive format [doc]

```
#pragma omp ordered
{
  // Bloc
}
```

- Only within the body of a for with ordered clause
- Executions of the block following the directive respect the sequential ordering
- Threads may wait for each other to respect that ordering
- Parts of the loop body outside this directive scope can run in parallel



critical Directive

C/C++ critical directive format [doc]

```
#pragma omp critical [name]
{
  // Block
}
```

- A code block within the scope of this directive must be executed one thread at a time
- If a thread executes a block within the scope of a `critical` directive and a second thread reaches that block, then the second one will have to wait for the first one to finish before starting the block execution
- Blocks within the scope of a `critical` directive with the same name are executed in mutual exclusion
- Accepted clauses: none

`atomic` Directive

C/C++ `atomic` directive format [doc]

```
#pragma omp atomic
// Assignment statement
```

- The assignment (evaluation and store of a value) within the scope of the directive is performed **atomically**
- More efficient than the `critical` directive in this case
- Specific forms of instruction: see OpenMP standard [doc]
- Accepted clauses: none

`master` Directive

C/C++ `master` directive format [doc]

```
#pragma omp master
{
  // Block
}
```

- A code block within the scope of this directive is executed by the master thread only, the other threads pass the block
- No implicit barrier at entry or exit of the code block
- Accepted clauses: none

`flush` Directive

C/C++ `flush` directive format [doc]

```
#pragma omp flush (/* List of shared variables */)
```

- The temporary view of the thread that reaches this directive is set to be consistent with the shared memory state for all variables in the list
- Implicit after a parallel region, a work sharing construct (except if `nowait` is present), a critical section or a lock
- To be inserted after writing in one thread and before reading in another for sharing a variable in a consistent way
- Necessary even on a cache coherent system
- Accepted clauses: none

Exercise

Correct the codes provided as examples of typical mistakes:

- Data race
- Coherence issue
- Synchronisation issue

OpenMP Directives

Task Management

Concept of Task

An OpenMP task is an independent code block that has to be executed by a thread of the team

- Work sharing constructs create tasks implicitly
- OpenMP offers a way to create them explicitly
  - To parallelize `while` loops
  - To parallelize recursive codes
- Each parallel region has its own task pool
- Creating a task adds it to the pool; it may be executed by the thread which created it or not, immediatly or not
- Threads execute the pool tasks once they reach a barrier

`task` Directive

C/C++ `task` directive format [doc]

```
#pragma omp task [clause [clause] ...]
{
  // Block
}
```

- Create a new task to execute the code block in the scope of this directive
- Creating thread either executes the task immediatly (see clauses) or adds it to the pool
- Variables used inside the task are `shared` if already shared since the first parallel region, `firstprivate` otherwise
- Accepted clauses: `if` (semantics different from `parallel` directive), `final`, `untied`, `default`, `mergeable`, `private`, `firstprivate`, `shared`

Clauses for `task` directive

C/C++ `if` clause format [doc]

```
if (/* Scalar expression */)
```

- Immediate execution by the thread if true, pool if false

C/C++ `final` clause format [doc]

```
final (/* Scalar expression */)
```

- Sub-tasks merged to the task if true

C/C++ `untied` clause format [doc]

```
untied
```

- Any thread may resume the task execution

C/C++ `mergeable` clause format [doc]

```
mergeable
```

- Task may be merged if immediatly or sequentially included

taskwait Directive

C/C++ taskwait directive [doc]

```
#pragma omp taskwait
```

- The thread that reaches this directive waits for the termination of all tasks it created
- Task-specific barrier
- Accepted clauses: none

Exercise: possible output?

[code 1, 2, 3, 4]

```c
#include <stdio.h>
int main() {

  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("Hello,\n");
      printf("world!\n");
    }
  }

  return 0;
}
```

```c
#include <stdio.h>
int main() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp task
      printf("Hello,\n");
      #pragma omp task
      printf("world!\n");
    }
  }
  return 0;
}
```

```c
#include <stdio.h>
int main() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp task
      printf("Hello,\n");
      #pragma omp task
      printf("world!\n");
      printf("Bye\n");
    }
  }

  return 0;
}
```

```c
#include <stdio.h>
int main() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp task
      printf("Hello,\n");
      #pragma omp task
      printf("world!\n");
      #pragma omp taskwait
      printf("Bye\n");
    }
  }
  return 0;
}
```

Exercise

- Parallelize this code which computes Fibonacci numbers
- Compare parallel and sequential performance

Fibonacci number computation [code]

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

int fibo(int n) {
  if (n < 2)
    return n;

  return fibo(n-1) + fibo(n-2);
}

int main(int argc, char* argv[]) {
  int n = atoi(argv[1]);
  printf("fibo(%d) = %d\n", n , fibo(n));
  return 0;
}
```

OpenMP Library

OpenMP Library

- Functions to control the runtime environment
  - Change the behavior at runtime (e.g., scheduling policy)
  - Runtime monitoring (e.g., total number of threads, thread number etc.)
- General purpose utility functions, even outside parallelization (portability)
  - Time measurement
  - Lock mechanism

Preserving Independence w.r.t. OpenMP

`_OPENMP` set when the compiler supports OpenMP

- Use preprocessor directives to write code with and without OpenMP support
- Define macros for each required OpenMP function

Example of conditional compilation [code]

```c
#include <stdio.h>
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif

int main() {
  #pragma omp parallel
  printf("Hello␣from␣thread␣%d\n", omp_get_thread_num());
  return 0;
}
```

Runtime Controlling Functions

`void omp_set_num_threads(int n);`	Set the number of threads for the next parallel region to `n`
`void omp_set_dynamic(int bool);`	Enable or disable automatic adjustment of the number of threads
`void omp_set_nested(int bool);`	Enable or disable the support for nested parallel regions
`void omp_set_max_active_levels( int n);`	Set the maximum number of nestable parallel regions to `n`
`void omp_set_schedule( omp_sched_t type, int chunk);`	Set scheduling policy when `runtime` was chosen to `type` (1 for `static`, 2 for `dynamic`, 3 for `guided` ou 4 for `auto`) and specify the value of `chunk`

Runtime Monitoring Functions 1/2

`int omp_get_num_threads();`	Returns the number of threads in the team
`int omp_get_dynamic();`	Evaluates to true if the automatic adjustment of the number of threads is enabled, to false otherwise
`int omp_get_nested();`	Evaluates to true if the support for nested parallel regions is enabled, to false otherwise
`int omp_get_max_active_levels();`	Returns the maximum number of nestable parallel regions
`void omp_get_schedule( omp_sched_t* type, int* chunk);`	Returns the current type of scheduling policy and chunk value (see `opm_set_schedule()` for possible values)

Runtime Monitoring Functions 2/2

`int omp_get_thread_num();`	Returns the number (thread id) of the calling thread
`int omp_get_num_procs();`	Returns the number of available processors
`int omp_in_parallel();`	Evaluates to true if a parallel region is running, to false otherwise
`int omp_in_final();`	Evaluates to true if the current task is `final`, to false otherwise

And many more (see standard document)...

Time Measurement Functions

`double omp_get_wtime();`	Return the elapsed time in seconds since a reference time
`double omp_get_wtick();`	Return the elapsed time in seconds between two clock "tops" (provides the accuracy of the time measurement)

- Measurements should be compared in the same thread
- Portable functions very useful even outside parallelization

OpenMP Locks

Two types of locks coming with their associated functions:

- `omp_lock_t` for simple locks
- `omp_nest_lock_t` for nested locks, which may be locked several times by a thread and which require to be unlocked the same number of times by this thread to be released

- More flexible alternatives to `atomic` and `critical`
- Portable locks for both Unix and Windows
- Be careful to initialize them with the appropriate functions
- Be careful not to lock a simple lock several times
- Prefer `atomic` or `critical` when possible

Functions on Locks

`void omp_init_lock(omp_lock_t* l);`	Initialize a lock
`void omp_destroy_lock(omp_lock_t* l);`	Destroy a lock
`void omp_set_lock(omp_lock_t* l);`	Set a (nested) lock, the calling task is suspended until the lock is set
`void omp_unset_lock(omp_lock_t* l);`	Unset a (nested) lock
`int omp_test_lock(omp_lock_t* l);`	Try to set a lock without suspending the task; evaluates to true if the attempt succeeds, to false otherwise

- `nest_lock` instead of `lock` everywhere for neste locks

Lock Usecase Example

[code]

```
#include <stdio.h>
#include <omp.h>
#define MAX 10
int main() {
  omp_lock_t lock;
  omp_init_lock(&lock);

  #pragma omp parallel sections
  {
    #pragma omp section
    for (size_t i = 0; i < MAX; i++) {
      if (omp_test_lock(&lock)) {
        printf("Thread A: locked work\n");
        omp_unset_lock(&lock);
      } else {
        printf("Thread A: alternative work\n");
      }
    }
    #pragma omp section
    for (size_t i = 0; i < MAX; i++) {
      omp_set_lock(&lock);
      printf("Thread B: locked work\n");
      omp_unset_lock(&lock);
    }
  }

  omp_destroy_lock(&lock);
  return 0;
}
```

OpenMP Environment Variables

Background On Environment Variables

- Dynamic variables used by processes to communicate
- Considered by the OpenMP runtime with a lower priority than the library functions, themselves considered with lower priority than the directives
- Display the value of a variable `VAR_NAME`
  - Unix: `echo $VAR_NAME`
  - Windows: `set %VAR_NAME%`
- Assignment of a value `VAL` to variable `VAR_NAME`
  - Unix: `VAR_NAME=VAL`
  - Windows: `set VAR_NAME=VAL`

OpenMP Environment Variables

`OMP_NUM_THREADS`	Integer: number of threads in thread teams
`OMP_SCHEDULE`	`type[,chunk]`: specify the scheduling policy, see `schedule` clause
`OMP_DYNAMIC`	`true` or `false`: allow or forbid the runtime to dynamically adjust the number of threads
`OMP_NESTED`	`true` or `false`: enable or disable nested parallelism
`OMP_MAX_ACTIVE_LEVELS`	Integer: maximum number of active levels of nested parallelism
`OMP_THREAD_LIMIT`	Integer: maximum number of threads

And many more (see standard document)...

Take Home Best Practice

- Use the `default(none)` clause
  - To avoid using a shared variable by mistake
- Create parallel regions outside loops when possible
  - To avoid spending time creating/destroying threads
- Synchronize only if necessary
  - Consider the `nowait` clause
- Use the most suitable synchronization mechanism
  - Use `atomic` or `reduction` rather than `critical`
- Ensure load balancing amongst threads
  - Consider the various options of the `schedule` clause
- Give enough work to each thread
  - Consider the `if` clause