Shared-memory programming : OpenMP (I)

ING2-GSI-MI Architecture et Programmation Parallèle

Juan Angel Lorenzo del Castillo

juan-angel.lorenzo-del-castillo@cyu.fr

CY Cergy Paris Université

2023-2024

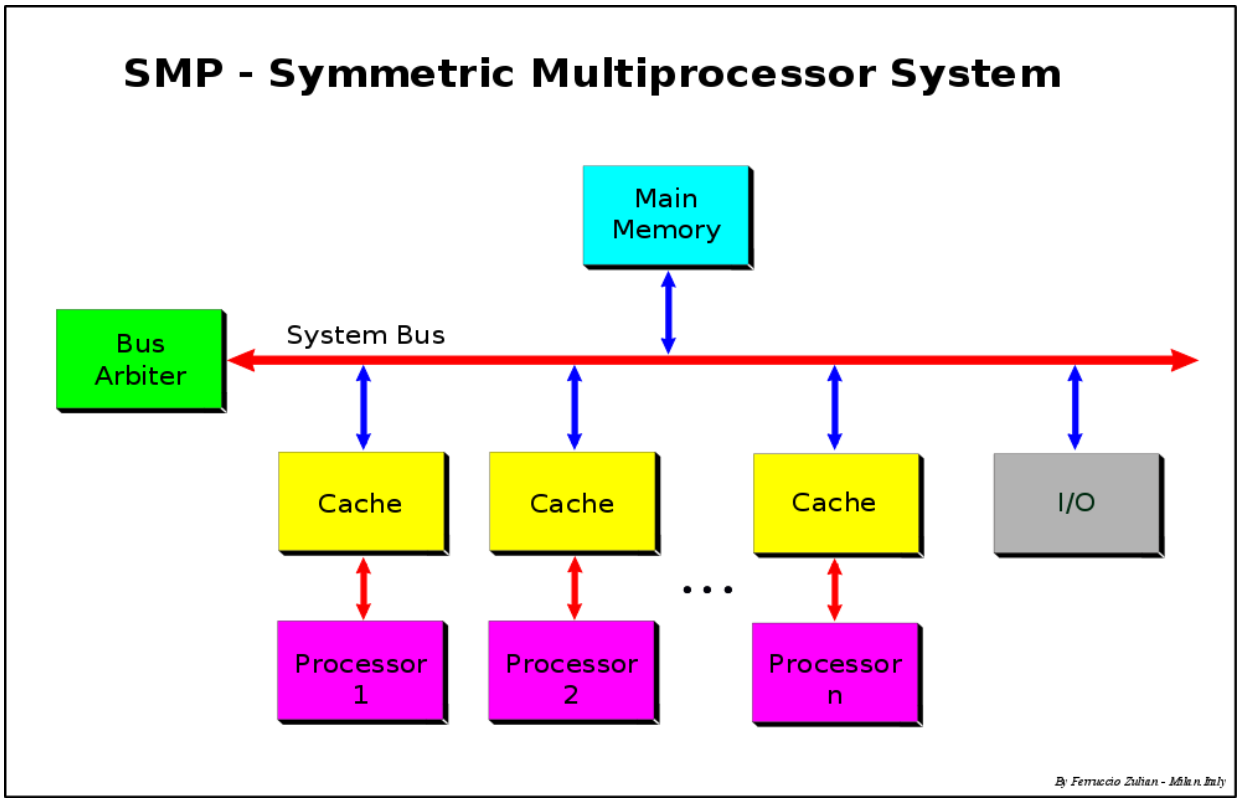
Table of Contents

Table of Contents

Shared-memory programming

Parallel programming

A program is split in processes or threads that cooperate and coordinate together.



Source : Wikipedia

Shared-memory programming

Parallel programming

A program is split in processes or threads that cooperate and coordinate together.

Shared-memory programming

- Based on the notion of *threads*
- Coordination and cooperation are done by writing and reading shared variables and by using synchronisation variables
  - **Low level:** barriers, locks, critical sections, semaphores...
  - **High level:** compilation **directives** (*OpenMP*).

Shared-memory programming

Parallelisation with directives

- **Directive:** A special line of source code with meaning only to certain compilers. It is distinguished by a sentinel at the start of the line.
  - The developer adds the compilation directives to the code.
  - The compiler will convert them into library calls.
- The original code is not modified.
- Portable, fast (but perhaps not too flexible nor efficient).
- Example: `#pragma omp`

Shared-memory programming

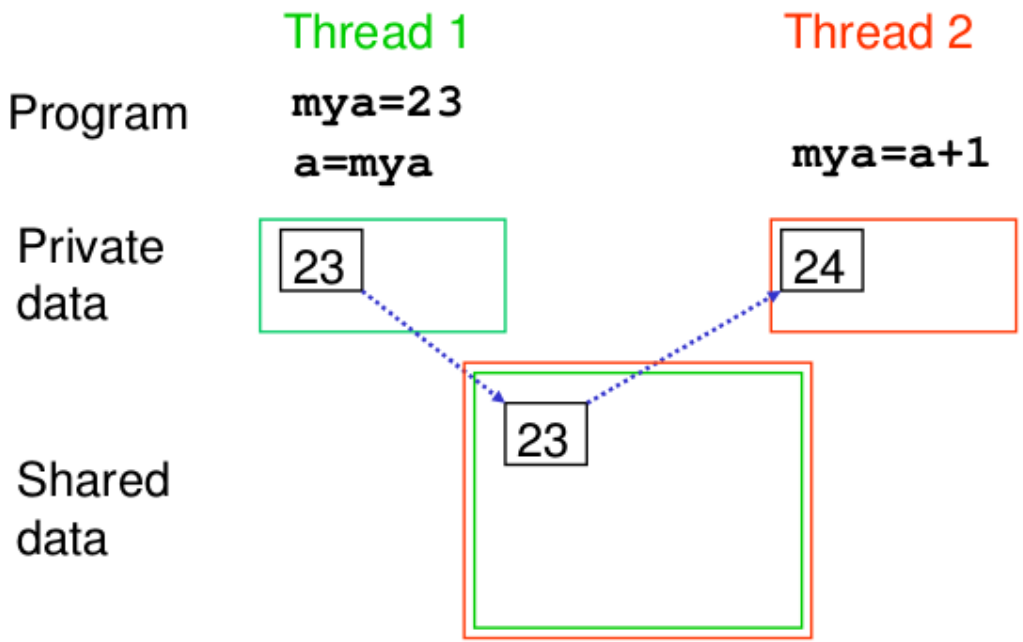
OpenMP is based on the notion of ***threads***

Reminder: Threads.

- Threads are like processes, except that threads can share memory with each other (as well as having private memory)
- Shared data can be accessed by all threads
- Private data can only be accessed by the owning thread
- Different threads can follow different flows of control through the same program
- Usually run one thread per processor (but could be more)
- Definitions :
  - *Thread team*: set of threads which cooperate on a task.
  - *Master thread*: is responsible for coordinating the team.

Shared-memory programming

Thread communication:



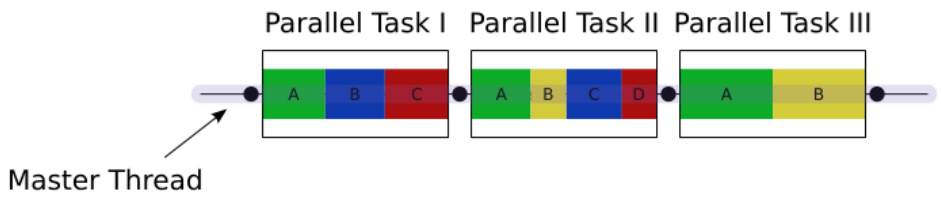
Source : EPCC

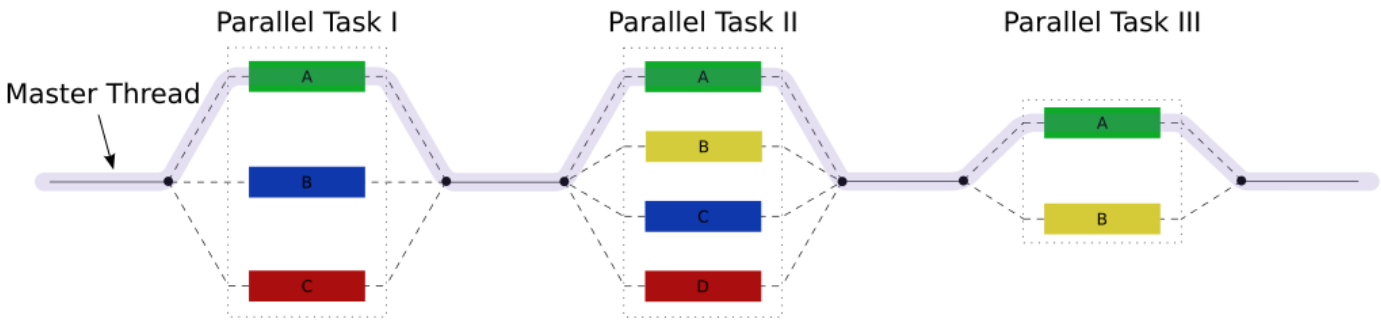
Table of Contents

Presentation

OpenMP (*Open MultiProcessing*)

- Set of directives, libraries and variables available for Fortran, C and C++.
- It is the standard on shared-memory programming.
- References:
  - www.openmp.org
  - www.compunity.org
- Based on the `fork-join` execution model.





Source : Wikipedia

OpenMP directives

Most relevant OpenMP directives

- Parallel regions construction
  - `parallel`
- Work sharing
  - `for`, `sections`, `single`.
- Synchronisation
  - `master`, `critical`, `atomic`, `barrier`.
- Task management
  - `task`, `taskwait`.

There are more...

OpenMP directives

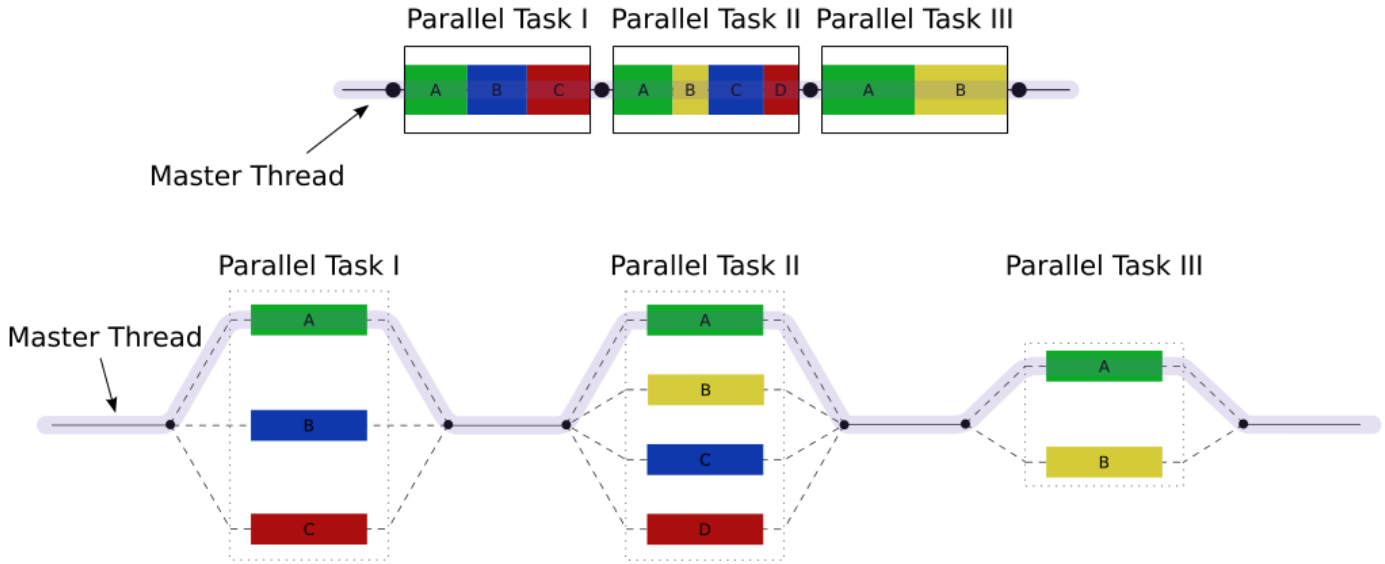
Most relevant OpenMP directives

- Parallel regions construction
  - **`parallel`**
- Work sharing
  - `for`, `sections`, `single`.
- Synchronisation
  - `master`, `critical`, `atomic`, `barrier`.
- Task management
  - `task`, `taskwait`.

There are more...

Today's class

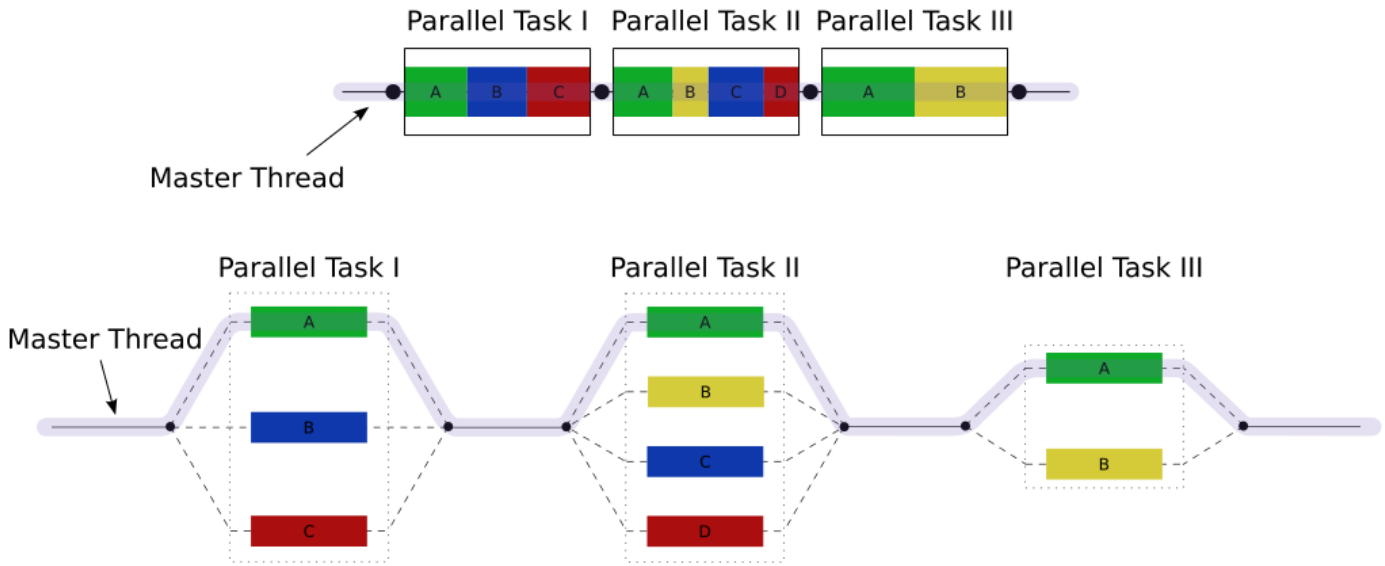
Parallel regions



Source : Wikipedia

- A program begins execution on a single thread (the *master* thread).
- When the first parallel region is encountered, the master thread creates a team of threads (fork/join model).
- Every thread executes the statements which are inside the parallel region.
- At the end of the parallel region, the master thread waits for the other threads to finish, and continues executing the next statements.

Parallel regions



Source : Wikipedia

- Inside a parallel region, variables can be either shared (all threads see the same copy) or private (each thread has its own copy).
- All threads see the same copy of shared variables.
- All threads can read or write shared variables.
- Each thread can have its own private variables, invisible to the other threads.
- A private variable can only be read or written by its owner thread.
- **Implicit barrier** at the end of the parallel region.

Parallel regions

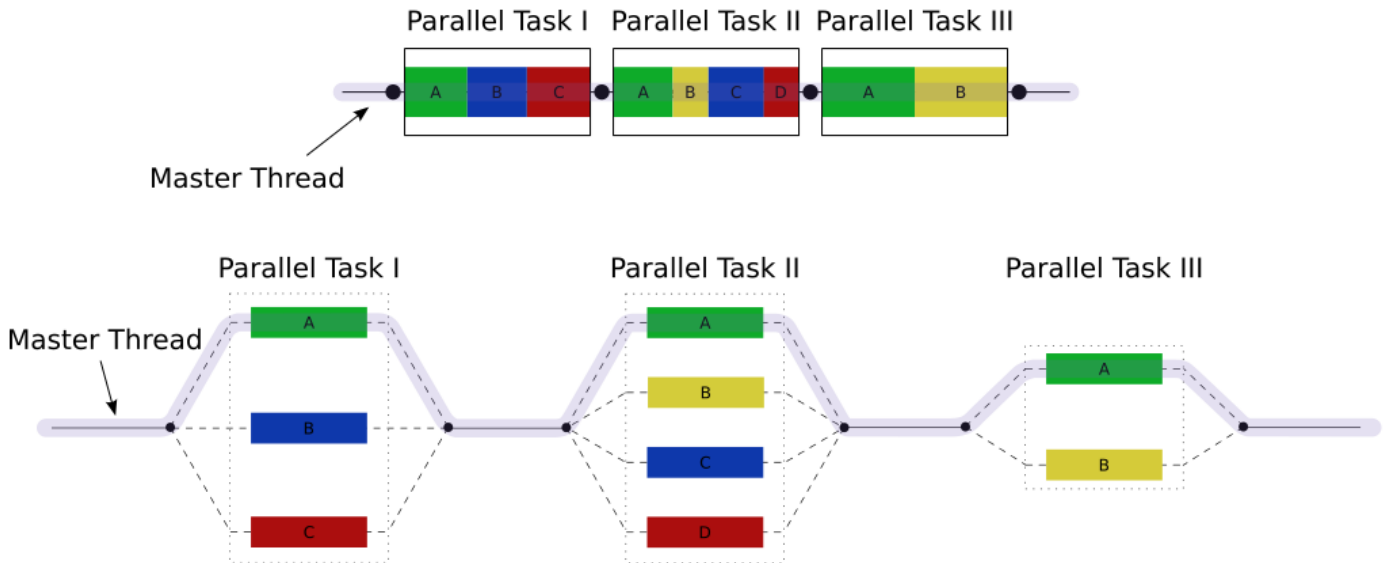
OpenMP example:

```
1  # include <omp.h>
2  # include <stdio.h>
3
4  int main () {
5
6     int sharedvar = 1;
7     printf(" Hola! I am the master thread :-) \n");
8
9     # pragma omp parallel
10     {
11         int id = omp_get_thread_num();
12         printf("I am thread %d. sharedvar = %d\n",id,sharedvar);
13     }
14
15     printf(" The master thread ends... \n");
16     return 0;
17 }
```

Compilation:

```
gcc -fopenmp -Wall -Wextra -o example01 example01.c
```

Parallel regions



Source : Wikipedia

- Note that the variables defined in the sequential section are **shared** inside the parallel region.
- Variables defined inside the parallel region are **private** to each thread.
- If the parallel region contains function calls, the local variables in the function will be **private**.

Setting/getting the number of threads

Setting the number of threads:

- As an environment variable (`bash/ksh`):
  - `export OMP_NUM_THREADS=8`

Setting/getting the number of threads

Setting the number of threads:

- As an environment variable (`bash/ksh`):
  - `export OMP_NUM_THREADS=8`

```
1 #!/bin/bash
2
3 export OMP_NUM_THREADS=4
4 ./example01
```

Setting/getting the number of threads

Setting the number of threads:

- As an environment variable (`bash/ksh`):
  - `export OMP_NUM_THREADS=8`
- In the program code:
  - `void omp_set_num_threads(num_threads);`

Setting/getting the number of threads

Setting the number of threads:

- As an environment variable (`bash/ksh`):
  - `export OMP_NUM_THREADS=8`
- In the program code:
  - `void omp_set_num_threads(num_threads);`

```
 1 # include <omp.h>
 2 # include <stdio.h>
 3
 4 int main () {
 5    omp_set_num_threads(8);
 6
 7    printf(" Hola! I am the master thread :-) \n");
 8
 9    # pragma omp parallel
10     {
11         int id = omp_get_thread_num();
12         printf("I am thread %d \n",id);
13     }
14
15     printf(" The master thread ends... \n");
16     return 0;
17 }
```

Setting/getting the number of threads

Setting the number of threads:

- As an environment variable (`bash/ksh`):
  - `export OMP_NUM_THREADS=8`
- In the program code:
  - `void omp_set_num_threads(num_threads);`
- Inside a parallel region:
  - `#pragma omp parallel num_threads(num_threads)`

Setting/getting the number of threads

Setting the number of threads:

- As an environment variable (`bash/ksh`):
  - `export OMP_NUM_THREADS=8`
- In the program code:
  - `void omp_set_num_threads(num_threads);`
- Inside a parallel region:
  - `#pragma omp parallel num_threads(num_threads)`

```
1  # include <omp.h>
2  # include <stdio.h>
3
4  int main () {
5     printf(" Hola! I am the master thread :-) \n");
6
7     # pragma omp parallel num_threads(8)
8      {
9          int id = omp_get_thread_num();
10         printf("I am thread %d \n",id);
11     }
12
13     printf(" The master thread ends... \n");
14     return 0;
15 }
```

Setting/getting the number of threads

Getting the number of threads (inside the program):

- Number of **running** threads in the team.
  - `int omp_get_num_threads();`
- Thread id
  - `int omp_get_thread_num();`
  - Between 0 and `omp_get_num_threads()` -1

Setting/getting the number of threads

Getting the number of threads (inside the program):

- Number of **running** threads in the team.
  - `int omp_get_num_threads();`
- Thread id
  - `int omp_get_thread_num();`
  - Between 0 and `omp_get_num_threads()` -1

Setting/getting the number of threads

Getting the number of threads (inside the program):

- Number of **running** threads in the team.
  - `int omp_get_num_threads();`
- Thread id
  - `int omp_get_thread_num();`
  - Between 0 and `omp_get_num_threads()` -1

```
1  # include <omp.h>
2  # include <stdio.h>
3
4  int main () {
5
6     omp_set_num_threads(8);
7     printf("How many threads (master): %d \n",omp_get_num_threads());
8
9     # pragma omp parallel num_threads(4)
10     {
11         int id = omp_get_thread_num();
12         printf("I am thread %d \n",id)  ;
13         printf("How many threads in the parallel region (%d): %d \n\n",id,
                   omp_get_num_threads());
14     }
15
16     printf(" The master thread ends... \n");
17     return 0;
18 }
```

Setting/getting the number of threads

Getting the number of threads (inside the program):

- Number of **running** threads in the team.
  - `int omp_get_num_threads();`
- Thread id
  - `int omp_get_thread_num();`
  - Between 0 and `omp_get_num_threads()` -1

Others :

- `int omp_get_max_threads();`
- `int omp_get_num_procs();`
- `int omp_in_parallel();`
- There are more...

Parallel region

Syntax:

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

Clauses allowed in a parallel region

- `if(logical expression)`
- `num_threads(integer)`
- `private(list of variables)`
- `shared(list of variables)`
- `default(shared | none)`
- `firstprivate(list of variables)`
- `reduction(operator: list of variables)`
- `copyin(list of variables)`

Parallel region

Syntax:

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

Clauses allowed in a parallel region

- `if(logical expression)`
- `num_threads(integer)`
- `private(list of variables)`
- `shared(list of variables)`
- `default(shared | none)`
- `firstprivate(list of variables)`
- `reduction(operator: list of variables)`
- `copyin(list of variables)`

Parallel region

#omp parallel if(logical expression)

- Creation of a thread team when the logical expression is different from 0. Otherwise, the execution will be sequential.

```
1  # include <omp.h>
2  # include <stdio.h>
3  # include <stdlib.h>
4
5  int main (int argc,char *argv[]) {
6
7      omp_set_num_threads(4);
8
9      if (argc != 3){
10         printf("Wrong number of parameters\n");
11         exit(0);
12     }
13
14     int a = atoi(argv[1]);
15     int b = atoi(argv[2]);
16
17     # pragma omp parallel if(a > b)
18      {
19         int id = omp_get_thread_num();
20         printf("I am thread %d\n",id);
21      }
22
23      return 0;
24 }
```

Parallel region

#omp parallel num_threads(integer)

- Creation of a thread team that will run the parallel region.
- Already seen.

Parallel region

#omp parallel private(list of variables)

- The variables in the list are declared private for every thread.
  - Every thread will have a local copy of the variable.
  - The variables **are undefined** before and after the parallel region.

#omp parallel shared(list of variables)

- The variables in the list are shared by all threads.
  - There is a danger of concurrent access.
  - Shared variables are used when:
    - There are read-only variables that we want to share.
    - Each thread accesses a different part of the variable (e.g. arrays).
    - We want to communicate a value to all threads.

#omp parallel default(shared | none)

- All variables in the parallel region will be shared (`shared`) or undefined (`none`).

Parallel region

Examples (I) :

```
1  # include <omp.h>
2  # include <stdio.h>
3
4
5  int main () { // This code is wrong!!
6     omp_set_num_threads(4);
7
8     int a = 2;
9     int b = 3;
10
11    #pragma omp parallel private(a) shared(b)
12     {
13          int id = omp_get_thread_num();
14 #pragma omp critical  //Ignore this pragma for now
15 {
16          printf("Thread %d. Before modification a: %d, b: %d\n",id,a,b);
17
18          a++;
19          b++;
20
21          printf("Thread %d. After modification a: %d, b: %d\n",id,a,b);
22 }
23    }
24
25     printf("Master thread. a: %d. b: %d\n",a,b);
26     return 0;
27 }
```

Parallel region

Examples (II) :

```
1  # include <omp.h>
2  # include <stdio.h>
3
4  int main () {
5   omp_set_num_threads(4);
6
7   int nthreads, tid;
8
9   #pragma omp parallel private(nthreads, tid)
10  {
11     tid = omp_get_thread_num();
12     printf("Hola! I am thread = %d\n", tid);
13
14     if (tid == 0) // If I am the master
15     {
16        nthreads = omp_get_num_threads();
17        printf("Number of threads = %d\n", nthreads);
18     }
19  }
20 }
```

Parallel region

Examples (III) :

```c
#include <omp.h>
#include <stdio.h>
#define N 12

int fillArray(int, int *);

int main () {
 omp_set_num_threads(4);
 int id,i;
 int A[N];
 for (i=0;i<N;i++) A[i] = 12;

 #pragma omp parallel private(id) shared(A)
 {
   id=omp_get_thread_num();
   fillArray(id,A);
 }

 printf("| ");
 for (i=0;i<N;i++)
 {
   printf("%d |",A[i]);
 }
 printf("\n");

 return 0;
}
```

```c

int fillArray(int tid,int *array)
{
  int begin,end,nth,chunk;
  nth = omp_get_max_threads();
  chunk = N/nth;
    //integer division (on purpose)

  printf("Thread %d. Chunk: %d\n",tid,chunk);

  begin = tid*chunk;
  end = begin + chunk;

  for (int i=begin; i<end ;i++)
     array[i] = tid;

  return 0;
}
```

Parallel region

#omp parallel firstprivate(list of variables)

- In general, private variables are not initialised at the beginning of the parallel region.
- However, the variables in the list of `firstprivate` are initialised.

```
1  # include <omp.h>
2  # include <stdio.h>
3
4  int main () { //Warning! it might show race conditions
5   omp_set_num_threads(4);
6
7   int a = 2;
8   int b = 3;
9
10  # pragma omp parallel firstprivate(a) shared(b)
11  {
12    int id = omp_get_thread_num();
13
14    printf("Thread %d. a: %d. b: %d\n",id,a,b);
15    a++; b++;
16    printf("Thread %d. a++: %d. b++: %d\n",id,a,b);
17  }
18
19  printf("Master. a: %d. b: %d\n",a,b);
20
21  return 0;
22 }
```

Parallel region

#omp parallel reduction(operator : list of variables)

- Produces a single value from associative operations (+,*,max,min,and,or).

```
1  # include <omp.h>
2  # include <stdio.h>
3  # define R 2
4  # define C 3
5
6  int main () {
7   int b = 0;  //Try with b = 100 instead!
8   int a[R][C] = {{1,2,3},{4,5,6}};
9
10  omp_set_num_threads(2);
11
12  # pragma omp parallel reduction(+:b)
13  {
14    int id = omp_get_thread_num();
15    //Try here setting b = 100
16
17    for (int i=0;i<C;i++){
18         b = b + a[id][i];
19    }
20    printf("Thread %d working on line %d of a. b = %d\n",id,id,b);
21  }
22
23  printf("Back to master. b = %d\n",b);
24  return 0;
25 }
```