OpenMP parallel programming paradigm

with C programming language implementation point of view

HPC 4GMM 2021/2022

F. Couderc[1] and L. Giraud[2]

[1]CNRS, Institut de Mathématiques de Toulouse.
[2]INRIA Bordeaux Sud-Ouest.

Table of contents

1- Introduction

1- What is OpenMP ?

Main Idea

A parallel paradigm is primarily driven by the memory view, with OpenMP, all the **threads** share the same address space.

OpenMP (abbrevation for Open specifications for Multi-Processing) is:

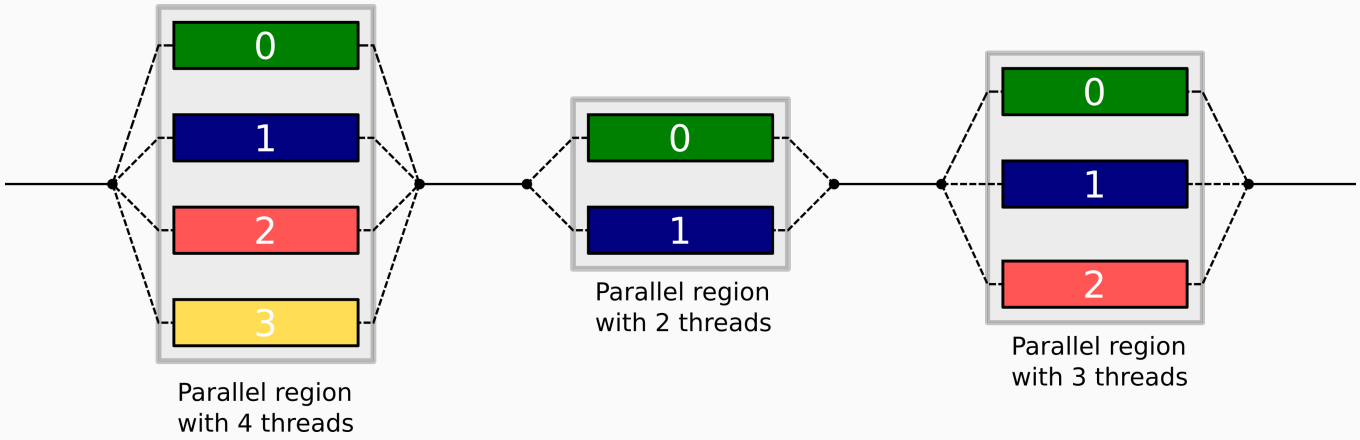
- An open standard API to explicitly perform direct **multi-threaded** and **shared memory** parallelism, defined and developed by a consortium of hardware and sotfware vendors: Intel, Nvidia, AMD, etc ... and some universities (https://www.openmp.org/).
- Comprised of three primary components:
  1. **Compiler directives**: inserted in a C,C++ or Fortran source and interpreted by the compiler.
  2. **Runtime Library Routines**: usefull routines that can be directly called from the source.
  3. **Environment Variables**: controlling the program at execution.

And with Python ?

No direct implementation, but number of packages can be used to do so with same paradigm (Python Wiki). The **numpy** package is certainly the easiest way because routines (like **`numpy.dot`**) use in back C/Fortran with OpenMP (depends on the installation).

1- Fork and Join Model

OpenMP uses the fork-join model of parallel execution:



1. Program execution is started by a single thread called master thread.
2. Master thread can then creates a team of threads in a parallel region.
3. The set of instructions enclosed in a parallel region (task) is executed.
4. All the threads synchronize and terminate at the end of the parallel region and only the master thread is living.
5. and so on ...

2- The parallel construsct

2- The basis

- You usually need to include the **<omp.h>** header file.
- A preprocessing command can also be inserted to ensure code portability for compiler that do not support OpenMP.
- Compiler directives appear as **comments** in the source code and are ignored by compilers unless the appropriate compiler flag (**-fopenmp** which **gcc** or **gfortran**) is specified.

C

```c
#ifdef _OPENMP
#include <omp.h>
#endif

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

Fortran

```fortran
USE OMP_LIB

!$OMP parallel
...
!$OMP end parallel
```

2- The parallel construct

The parellel construct is the main OpenMP construct identifying a block of source code that will be executed by a team of threads.

C/C++

```
#pragma omp parallel [clause ...]
                     shared (list)
                     private (list)
                     firstprivate (list)
                     default (private|firstprivate|shared|none)
                     reduction (operator: list)
                     if (scalar_expression)
                     num_threads (integer-expression)
{
...
}
```

1. The master thread is member of the team and has thread number 0.
2. Starting from the beginning of this parallel region, the code is duplicated and all threads will execute the code.
3. There is an **implied barrier** at the end of a parallel region.

2- Data scoping

An important consideration for OpenMP programming is the understanding and use of data scoping (or data sharing). The OpenMP data scope attribute clauses are used to explicitly define how variables should be scoped. They include:

- shared (list): declares a list of variables to be shared among all team threads (by default as OpenMP is based upon the shared memory programming model). A shared variable exists in only one memory location and all threads can read or write to that address. It is the programmer's responsibility to ensure that multiple threads properly access shared variables.
- private (list): declares a list of variables of the same type of their original objects, created and private for each thread, but remain uninitialized.
- firstprivate (list): combines the the private clause behavior with automatic initialization according to the value of their original objects prior to entry into the parallel region.
- reduction (operator: list): performs a reduction on the variables that appear in its list (see later in synchronization section for a better explanation).

2- First ’hello.c’ program

hello.c

```c
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#endif
void main()
{
   int tid, max_threads, num_threads;
   #ifdef _OPENMP
      max_threads = omp_get_max_threads();
      printf("\n Parallel execution with a maximum of %d threads callable\n\n",max_threads);
   #else
      printf("\n Sequential execution\n\n");
   #endif
   #pragma omp parallel private(tid) shared(max_threads) num_threads(2)
   {
      #ifdef _OPENMP
         tid         = omp_get_thread_num();
         num_threads = omp_get_num_threads();
         printf("Hello World from thread %2d / %2d / %2d\n", tid, num_threads, max_threads);
      #else
         printf("Hello World\n");
      #endif
   }
}
```

2- First 'hello.c' program

1. Compile the source code typing the command:

```
gcc -o hello_seq hello.c
```

   in a terminal. Excute it and observe the behavior.

2. Compile the source code calling OpenMP typing the command:

```
gcc -o hello_par hello.c -fopenmp
```

   in a terminal. Excute it and observe the behavior.

3. Try to change the value of the environment variable **OMP_NUM_THREADS** (to `x=2,3,4` for instance) typing the command:

```
export OMP_NUM_THREADS=x
```

   in a terminal. Reexecute for each `x` the last parallel code and observe the behavior.

3- The worksharing constructs

3- The for construct: basis

- The for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team.
- This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.
- This can be concatenated with the parallel directive (`#pragma omp parallel for`).
- There is an **implied barrier** at the end of the construct until the nowait clause is prescribed.

C/C++

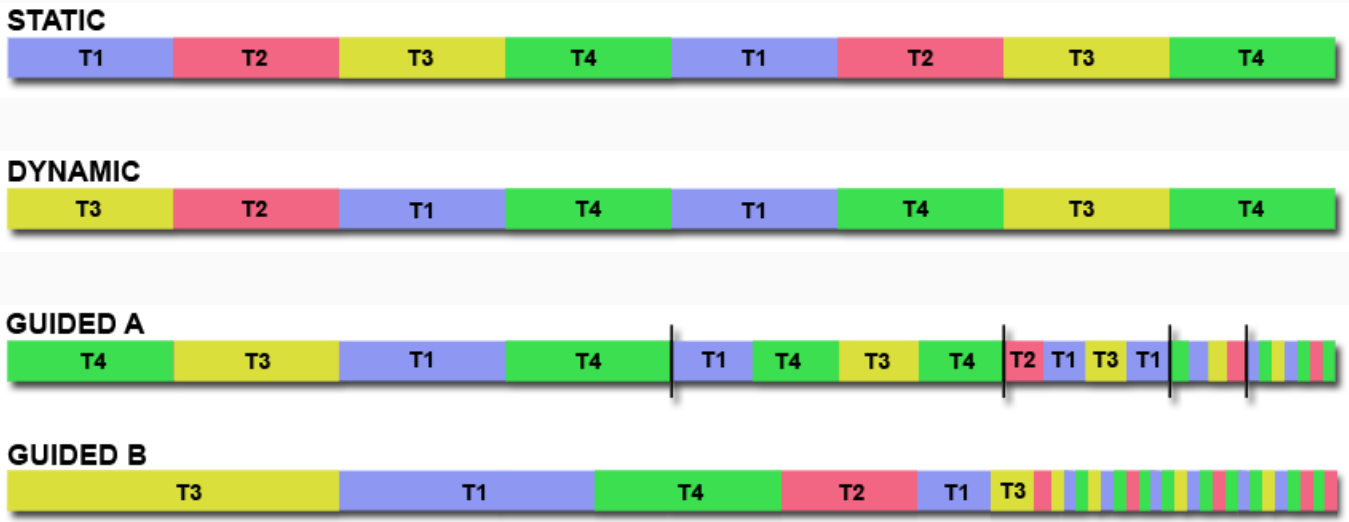
```
#pragma omp for [clause ...]
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

  for_loop
```

3- The for construct: scheduling

The schedule describes how iterations of the loop are divided among the threads in the team.

- static: loop iterations are evenly divided (if possible) among threads, or divided into pieces of size chunk if prescribed, and contiguously assigned to threads.
- dynamic: loop iterations are divided into pieces of size chunk (by default 1), and dynamically assigned to threads at runtime.
- guided: for a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For greater chunk size, the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations.
- runtime: deferred until runtime by the environment variable `OMP_SCHEDULE`.



3- The for construct: sample program

do.c

```
...
  #pragma omp parallel shared(a,b,c,chunk,which) private(i,tid)
   {
      tid = omp_get_thread_num();
      #pragma omp for schedule(static,chunk)
         for (i=0; i<N; i++)
         {
            c[i] = a[i] + b[i];
            which[i] = tid;
         }
   }
...
```

- Compile and execute the program with the **`-fopenmp`** compiler flag.
- Change the scheduling from **`static`** to **`dynamic`** and study the difference in loop indexes assigned to threads.
- Same changing the preprocessor variable **`CHUNKSIZE`**.
- Same changing the scheduling to **`guided`**.

3- the section construct: basis

- The sections directive is a non-iterative work-sharing construct. It species that the enclosed section(s) of code are to be divided among the threads in the team.
- There is an **implied barrier** at the end of the construct.

C/C++

```
#pragma omp sections [clause ...]
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
{
   #pragma omp section
   {...}
   #pragma omp section
   {...}
}
```

3- the section construct: sample program

sections1.c

```
...
#pragma omp parallel
{
   printf ("outer     id = %d, \n", omp_get_thread_num());
   #pragma omp sections
   {
      #pragma omp section
      {
          printf ("section 1 id = %d, \n", omp_get_thread_num());
      }
      #pragma omp section
      {
          printf ("section 2 id = %d, \n", omp_get_thread_num());
      }
   }
}
...
```

- Compile and execute the program.
- Change the value of **OMP_NUM_THREADS** and study the behavior.
- Same with `sections2.c` and observe the difference.

3- the single construct: basis

- The single directive specifies that the enclosed code is to be executed by only one thread in the team (may be useful when dealing with sections of code that are not ***thread safe***).
- There is an **implied barrier** at the end of the construct

C/C++

```
#pragma omp single [clause ...]
                   private (list)
                   firstprivate (list)
                   nowait
{
...
}
```

4- Synchronization

4- the critical, master and barrier directives

- The critical directive specifies a region of code that must be **executed by only one threa**d at a time. If a thread is currently executing inside the region and another thread reaches that region and attempts to execute it, it will block until the first thread exits.
- There is an **implied barrier** at the end of the construct.

C/C++

```
#pragma omp critical
{...}
```

- The master directive specifies a region that is to be **executed only by the master thread** of the team. All other threads on the team skip this section of code.
- There is **no implied barrier** at the end of the construct.

C/C++

```
#pragma omp master
{...}
```

- The barrier directive **synchronizes all threads in the team**. When a barrier directive is rached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

C/C++

```
#pragma omp barrier
```

4- the atomic directive

- The atomic construct ensures that a specific storage location is accessed *atomically*, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values (no competition, or so called **data race**, between threads).
- It is only applied to 'simple' operations that can be contolled by the clauses: read, write, update and capture. In a 'basic' practice, only the update clause is really used for operations like **`x += y`**.
- Less restrictive than the critical or master directives.

C/C++

```
#pragma omp atomic [read|write|update|capture]
{
...
}
```

4- the atomic directive: sample program

atomic.c

```
...

#pragma omp parallel for default(shared) private(i)
   for (i=0; i<N1; i++)
   {
      y[i] += work2(i);
      #pragma omp atomic update
         x[index[i]] += work1(i);
   }
...
```

- Compile and execute the program with and without the atomic directive.
- Observe the *'good'* and the *'bad'* behavior.

4- the reduction clause

- The reduction clause performs a reduction operation on the variables that appear in its list.
- A private copy for each list variable is created and initialized for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.
- Can be ‘emulated ‘ with atomic directive for example, but the reduction clause is optimized at runtime in comparison.

reduction.c

```
#pragma omp parallel for default(shared) private(i)\
        reduction(+:norm1, norm2) reduction(max:norminf)
   for (i=0; i<n; i++)
   {
      norm1 = norm1 + abs(x[i]);
      norm2 = norm2 + x[i]*x[i];
      normInf = fmax(normInf, abs(x[i]));
   }
```

5- Further readings

5- Further readings

- Other courses and docs:
  - complete course that greatly inspired the present condensed course comparatively: https://hpc.llnl.gov/tuts/openMP/
  - OpenMP 5.1 API syntax reference guide (Nov 2020): https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5.1-web.pdf (check the numerous other possibilities than ones presented here)
  - OpenMP 5.1 complete specification (Nov 2020): https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf
  - OpenMP 5.1 examples (August 2021): https://www.openmp.org/wp-content/uploads/openmp-examples-5.1.pdf
- GPU:
  - **OpenACC** '*equivalent*' to OpenMP but to target GPU: https://www.openacc.org/, different from **Cuda** in sense that the same code can be executed both on CPU and GPU rather than only target GPU, but with less control over optimization ...
  - **Sycl**, an High-level C++ abstraction layer for OpenCL: https://www.khronos.org/sycl/.