Shared-memory programming : OpenMP (III)

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

OpenMP directives

Most relevant OpenMP directives

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

There are more...

Today's class

Synchronisation

Why synchronising?

- Need to synchronise actions on shared variables.
- Need to ensure correct ordering of reads and writes.
- Need to protect updates to shared variables (not atomic by default).
- Need to ensure that all members of a thread team have finished a task before starting the next one.

Synchronisation

#pragma omp master

Syntax:

```
#pragma omp master
    structured block
```

- Only the `master` thread executes the structured block of code.
- The other threads will skip the directive.
- There are **no implicit barriers** before or after the directive.

Synchronisation

#pragma omp critical

Syntax:

```
#pragma omp critical [(name)]
    structured block
```

- A critical section is a block of code which can be executed by only one thread at a time.
- All threads will execute the block, but only one thread has access to the code at a time.
- Can be used to protect updates to shared variables.
- If one thread is in a critical section with a given name, no other thread may be in a critical section with the same name (though they can be in critical sections with other names).
- Example: pushing and popping a task stack.

Synchronisation

#pragma omp atomic

Syntax:

```
#pragma omp atomic
    code statement
```

- Used to protect a single update to a shared variable.
- Applies only to a single statement (the one next to the pragma).

Statements allowed:

- $x{+}{+},\ {+}{+}x,\ x{-}{-},\ {-}{-}x,\ x\ \ binop = expr$
  - binop : +, *, -, /, &, |, ^, «, »
- Note that the evaluation of `expr` is not atomic.
- May be more efficient than using `critical` directives.

Synchronisation

#pragma omp barrier

Syntax:

```
#pragma omp barrier
```

- No thread can proceed past a barrier until all the other threads have arrived.

Synchronisation

#pragma omp ordered

Syntax:

```
#pragma omp ordered
    structured block
```

- Can specify code **within a `for` loop** which must be done in the order it would have been done if executed sequentially.
- That way we can execute sequentially a block of code inside a parallel region.

Synchronisation

#pragma omp ordered

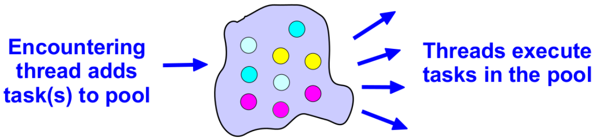
```c
# include <omp.h>
# include <stdio.h>

const int SIZE = 12;
void showArray(double *M);

int main () {
 omp_set_num_threads(4);
 int i,id;
 double A[SIZE];

 #pragma omp parallel private(id)
 {
  id=omp_get_thread_num();

  #pragma omp for
  for (i=0; i<SIZE; i++)
    A[i] = id + 1.0;

  #pragma omp single
  {
     printf("Before... \n");
     showArray(A);
  }

  #pragma omp for ordered
  for (i=1; i<SIZE; i++)
  {
    /* More code here... */

    #pragma omp ordered
    {
      A[i] /= A[i-1];
    }
  }
 }

 printf("After... \n");
 showArray(A);
 return 0;
}

void showArray(double *M)
{
 int i;
 for (i=0;i < SIZE; i++)
   printf("| %2.2lf",M[i]);

 printf("|\n");

}
```

Tasking

- Tasking was introduced in OpenMP 3.0
- Until then it was impossible to efficiently and easily implement certain types of parallelism (e.g. linked lists and recursive algorithms).
- Tasks are **work units** which execution ***may*** be deferred
  - they can also be executed immediately!
- Tasks are composed of:
  1. code to execute
  2. data environment

Tasking

Tasking example:



Source : Oracle

- When a thread encounters a task construct, a new task is generated
- The new task will be executed either by the same thread or by any other thread from the thread team (*delayed* execution).
- The moment of execution of the task is up to the runtime system.
- Completion of a task can be enforced through **task synchronisation**.
- Switch from the parallel-data paradigm to a parallel-task one (useful for recursive algorithms).
- Producer/Consumer model.

Tasking

Syntax:

```
#pragma omp task[clauses]
   structured block
```

Allowed clauses:

- `if`.
- `untied`.
- `private`.
- `firstprivate`.
- `shared`.
- `default (private | firstprivate | shared | none)`.

Tasking

#pragma omp task if(expression)

- If `expression` is `False`, the encountering task is suspended and the new task is executed immediately.

#pragma omp task untied

- By default, tasks are executed always by the same thread (`tied`). We can lift this restriction with `untied`.

#pragma omp task firstprivate(liste de variables)
#pragma omp task private(liste de variables)
#pragma omp task shared(liste de variables)
#pragma omp task default(private | firstprivate | shared | none)

- Already seen.

Tasking

Tasking Example:

Write a program that prints either “An engineering student” or “An student engineering” and maximise the parallelism.

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

int main(int argc, char *argv[])
{

        printf("An ");
        printf("engineering ");
        printf("student ");

    printf("\n");
    return(0);
}
```

```
$ gcc -o 02-taskingExample 02-taskingExample.c

$ ./02-taskingExample
An engineering student
```

Tasking

Tasking Example (II):

Write a program that prints either “An engineering student” or “An student engineering” and maximise the parallelism.

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

int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
      printf("An ");
      printf("engineering ");
      printf("student ");
  }

   printf("\n");
   return(0);
}
```

What will the output be with 2 threads?

```
$ gcc -fopenmp -o 03-taskingExample 03-taskingExample.c
$ export OMP_NUM_THREADS=2

$ ./03-taskingExample
An engineering student An engineering student
```

But this program could have also printed:

```
An An engineering engineering student student
An engineering student student engineering
An student engineering An student engineering
...
```

Tasking

Tasking Example (III):

Write a program that prints either “An engineering student” or “An student engineering” and maximise the parallelism.

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

int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
   #pragma omp single
   {
       printf("An ");
       printf("engineering ");
       printf("student ");
   }
  }

   printf("\n");
   return(0);
}
```

What will the output be with 2 threads?

```
$ gcc -fopenmp -o 04-taskingExample 04-taskingExample.c
$ export OMP_NUM_THREADS=2

$ ./04-taskingExample
An engineering student
```

But now only one thread executes...

Tasking

Tasking Example (IV):

Write a program that prints either “An engineering student” or “An student engineering” and maximise the parallelism.

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

int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
   #pragma omp single nowait
   {
      printf("An ");
      #pragma omp task
      {printf("engineering ");}
      #pragma omp task
      {printf("student ");}
   }
  }

   printf("\n");
   return(0);
}
```

What will the output be with 2 threads?

```
$ gcc -fopenmp -o 05-taskingExample 05-taskingExample.c
$ export OMP_NUM_THREADS=2

$ ./05-taskingExample
An engineering student

$ ./05-taskingExample
An engineering student

$ ./05-taskingExample
An student engineering
```

Tasking

Task synchronisation

- Barriers (implicit or explicit): All tasks created by any thread of the current team are guaranteed to be completed at barrier exit.
- **Task barrier:**

Syntax:

```
#pragma omp taskwait
```

- The encountering task suspends until child tasks complete.

Tasking

Tasking Example (V) with synchronisation

Write a program that prints either “An engineering student” or “An student engineering”, maximise the parallelism and write “likes to party all the time” always at the end of the sentence.

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

int main(int argc, char *argv[])
{
 #pragma omp parallel num_threads(2)
 {
  #pragma omp single nowait
  {
     printf("An ");
     #pragma omp task
     {printf("engineering ");}
     #pragma omp task
     {printf("student ");}

     #pragma omp taskwait
     printf("likes to party all the time");
  }
 }
 printf("\n");
 return(0);
}
```

What will the output be with 2 threads?

```
$ gcc -fopenmp -o 06-taskingExample 06-taskingExample.c
$ export OMP_NUM_THREADS=2
$ ./06-taskingExample
An engineering student likes to party all the time

$ ./05-taskingExample
An engineering student likes to party all the time

$ ./05-taskingExample
An student engineering likes to party all the time
```

Tasks are executed first.

Tasking Example (VI): Fibonacci numbers

The Fibonacci Numbers are defined as follows:

```
F(0) = 1
F(1) = 1
F(n) = F(n-1) + F(n-2)  (n=2,3,4,.....)
```

Sequence: `1, 1, 2, 3, 5, 8, 13, 21, 34, ....`

(Sequential) Recursive algorithm:

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

long fib(int n);

int main (int argc, char **argv) {

 if (argc < 2){
    printf("No enough parameteres. Enter fib n value.\n");
    exit(0);
 }
 int n = atoi(argv[1]);

 long total = fib(n);
 printf("fib(%d): %ld\n",n,total);
}
```

```c
long fib(int n)
{
   if (n < 2){
    return n;
   }

   long x, y, z;
   x = fib(n-1);

   y = fib(n-2);

   z = x + y;
   return z;
}
```

Tasking Example (VI): Fibonacci numbers. Tasks version

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

long fib(int n);

int main (int argc, char **argv) {
 omp_set_num_threads(2);

 if (argc < 2){
    printf("No enough parameteres. Enter fib n value.\n");
    exit(0);
 }
 int n = atoi(argv[1]);

 #pragma omp parallel
 {
   #pragma omp single nowait
   {
    long total = fib(n);
    printf("fib(%d): %ld\n",n,total);
   }
 }
}
```

```c
long fib(int n)
{
  if (n < 2){
   return n;
  }

  long x, y, z;
  #pragma omp task shared(x)
  {
   x = fib(n-1);
  }

  #pragma omp task shared(y)
  {
   y = fib(n-2);
  }

  #pragma omp taskwait
  z = x + y;
  return z;
}
```

Tasking

Tasking Example (VI): Fibonacci numbers. Tasks version

Output (ordered for didactic purposes): `./07-fibonacci_debug 3`

Thread 1

```
Task 1. tid 1. Calling fib(n-1)=fib(2) x = 202

Task 1. tid 1. Calling fib(n-1)=fib(1) x = 1
n < 2 (n==1). Returning 1
Task 1. tid 1. Out of fib(n-1)=fib(1) x = 1

Task 2. tid 1. Calling fib(n-2)=fib(0) y = 4198
n < 2 (n==0). Returning 0
Task 2. tid 1. Out of fib(n-2)=fib(0) y = 0
#Goes to taskwait...

tid 1. Returning z=1. x=1 y=0. n=2
Task 1. tid 1. Out of fib(n-1)=fib(2) x = 1
#Goes to taskwait...
```

Thread 0

```
Task 2. tid 0. Calling fib(n-2)=fib(1) y = 1397
n < 2 (n==1). Returning 1
Task 2. tid 0. Out of fib(n-2)=fib(1) y = 1

#Goes to taskwait...
tid 0. Returning z=2. x=1 y=1. n=3
fib(3) = 2
```