Thread-level Parallelization with OpenMP (22 – 29, October, 2019)

The following set of exercises show the most representative primitives of OpenMP to develop parallel applications based on threads. They are not intended to provide a deep and complete knowledge of all OpenMP directives and functions, but they provide a wide overview of the main components of this framework. Refer to the official specification or other sources available at the CV for a more complete description. In general, different files with the source code will be available at the corresponding directory. They have to be compiled using the *gcc* compiler; use the following command

```
gcc -fopenmp -o example example.c
```

where “example” has to be replaced by the corresponding file name. First, the right environment variables must be set with:

```
module load gcc/8.2.0
```

The complete OpenMP specification can be found at

https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

On line information about directives, clauses and API functions can be accessed at

https://msdn.microsoft.com/es-es/library/tt15eb9t.aspx

Check first these sources of information when answering the questions of the following examples. Obviously, ask your instructor if everything else fails.

1. Basic parallelization

(hello.c)

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

- How many times will you see the "Hello world!" message? Why?
- Without changing the program, how to make it to print 5 times the "Hello World!" message? (Hint: use the OMP_NUM_THREADS environment variable).

- Now, do the same by changing the source code (hint: use the omp_set_num_threads () function or use a `num_treads()` clause at the `#pragma omp parallel` directive).

2. Thread identification

(threads.c)

```
int main ()
{int tid, nthreads;
    #pragma omp parallel
    {
       nthreads = omp_get_num_threads();
       tid = omp_get_thread_num();
       printf("Hello world! from tid: %d out of %d \n", tid,
       nthreads);
    }
    return 0;
}
```

- How many messages are printed? Are they printed in any specific order (execute it several times to check it)?

3. Data sharing

(data_sharing.c)

This program illustrates the use of some data sharing clauses that can be used in several directives.

```
int main ()
{
    omp_set_num_threads(8);

    int x=0;
    #pragma omp parallel shared(x)
    {
     x++;
     printf("Within first parallel (shared) x is: %d\n",x);
    }
    printf("After first parallel (shared) x is: %d\n",x);

    x=5;
    #pragma omp parallel private(x)
```

```
    {
      x++;
      printf("Within second parallel (private) x is: %d\n",x);
    }
    printf("After second parallel (private) x is: %d\n",x);

    x=71;
    #pragma omp parallel firstprivate(x)
    {
      x++;
      printf("Within third parallel (first private) x is:
%d\n",x);
    }
    printf("After third parallel (first private) x is: %d\n",x);

    return 0;
}
```

- Which is the value of x after the execution of each parallel region with different data-sharing (shared, private and firstprivate)?
- Add the necessary changes in the first parallel block to ensure that the value after it is always 8.
- Is there any potential data race in the program?

4. Parallel sections and data sharing

(sections.c)

This program specifies two parallel sections and illustrates four different ways to specify which variables are shared between different threads and how variables are initialized or copied back at the beginning and the end, respectively, of parallel threads.

```
int main( ) {
   int tid, section_count = 5;
   omp_set_num_threads(2);
   #pragma omp parallel
   #pragma omp sections
//   #pragma omp sections private( section_count )
//   #pragma omp sections firstprivate( section_count )
//   #pragma omp sections lastprivate( section_count )
   {
      #pragma omp section
      {
//      sleep(1);
```

```
        section_count++;
        section_count++;
        printf( "section_count %d from thread; %d \n",
        section_count, omp_get_thread_num() );
      }
      #pragma omp section
      {
//          sleep (1);
            section_count++;
           printf( "section_count %d from thread: %d \n",
           section_count, omp_get_thread_num() );
      }
   }
printf( "FINAL value section_count %d from thread: %d \n",
section_count, omp_get_thread_num() );
return 0;
}
```

- In the first case, variables are shared by all threads. What are the potential race conditions that affect the `section_count` variable? Try out different timing scenarios (using calls to sleep()) and check what values are printed in this program?
- When `section_count` is declared as a private variable, what are the differences when firstprivate, private or lastprivate clauses are used? (modify the program accordingly to try out each case). Do values of `section_count` depend on the order of execution of threads now? Why?

5. Data races

(datarace.c)

This program illustrates the appearance of data races when data is shared between threads. Execute several times to see the behavior of the program (data races might be hidden by some correct results that are obtained randomly). Then answer the questions.

```
#include <stdio.h>
#include <omp.h>
#define N 1 << 10
#define NUM_THREADS 8
```

```
int main()
{
    int i, x=0;

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(i)
     {
     int id=omp_get_thread_num();
        for (i=id; i < N; i+=NUM_THREADS) {
           x++;
        }
    }
    printf ("N = %d \n", N);
    if   (x==N)   printf("Congratulations!,   program   executed
correctly (x = %d)\n", x);
    else printf("Sorry, something went wrong, value of x = %d\n",
x);

    return 0;
}
```

- Is the program always executing correctly?
- Add two alternative directives to make it correct. Explain why they make the execution correct.

6. Barriers

(barrier.c)

- Can you predict the sequence of printf in this program? Do threads exit from the barrier in any specific order?

```
int main ()
{
    int myid;
    #pragma omp parallel private(myid) num_threads(4)
    {
     myid=omp_get_thread_num();
     printf("(%d) going to sleep for %d seconds..\n",
     myid,2+myid*3);
     sleep(2+myid*3);
     printf("(%d) wakes up and enters barrier ...\n",myid);
     #pragma omp barrier
     printf("(%d) We are all awake!\n",myid);
    }
    return 0;
```

```
}
```

7. Basic loops and worksharing

(for.c)

This program illustrates the use of the *omp for* directive, which constitutes the main worksharing construct.

```
#define N 32
int main()
{
    int i;

    omp_set_num_threads(8);
    #pragma omp parallel
    {
     printf("Going to distribute iterations in first loop
...\n");
     #pragma omp for
        for (i=0; i < N; i++) {
           int id=omp_get_thread_num();
           printf("(%d) gets iteration %d\n",id,i);
        }
    }
    return 0;
}
```

- How many and which iterations from the loop are executed by each thread? Which kind of *schedule* is applied by default?
- Add a directive so that the first "printf" is executed only once by the first thread that finds it.

8. Worksharing

(hello_for.c)

Check the output of this program and explain the differences you see in the execution of each for loop.

```
int main ()
{int i, tid, nthreads;

    #pragma omp parallel num_threads(4) private (i)
    {
      nthreads = omp_get_num_threads();
       for (i=0; i<nthreads; i++){
           tid = omp_get_thread_num();
           printf("Hello world! from tid: %d out of %d \n", tid,
           nthreads);
       }
     }
   #pragma omp parallel for
     for (i=0; i<nthreads; i++){
           tid = omp_get_thread_num();
           printf("Second Hello world! from tid: %d out of %d
           \n", tid, nthreads);
       }
  return 0;
}
```

- Modify the program so that the second loop has 6 iterations and each iteration is executed by a different thread (your solution has to be general for any number of iterations/threads)

- Modify the original program in such a way that only one parallel region is used but you get the same output (regard that you have to guarantee that all output messages from the first loop are printed before any output from the second loop). What are the main changes that are required to guarantee the order of messages?

9. Worksharing schedule

(schedule.c)

This example illustrates the use of the schedule clause within parallel for loops.

- Which iterations of the loops are executed by each thread for each schedule kind?

```
#define N 64

int main()
{
    int i;

    omp_set_num_threads(3);
    #pragma omp parallel
    {
     #pragma omp for schedule(static)
        for (i=0; i < N; i++) {
          int id=omp_get_thread_num();
          printf("Loop 1: (%d) gets iteration %d\n",id,i);
        }
     #pragma omp for schedule(static, 2)
        for (i=0; i < N; i++) {
          int id=omp_get_thread_num();
          printf("Loop 2: (%d) gets iteration %d\n",id,i);
        }

     #pragma omp for schedule(dynamic,2)
        for (i=0; i < N; i++) {
          int id=omp_get_thread_num();
          printf("Loop 3: (%d) gets iteration %d\n",id,i);
        }

     #pragma omp for schedule(guided,2)
        for (i=0; i < N; i++) {
          int id=omp_get_thread_num();
          printf("Loop 4: (%d) gets iteration %d\n",id,i);
        }
    }
    return 0;
}
```

10. Nowait

(nowait.c)

This example illustrates the use of the *nowait* directive.

```
int main()
{
    int i;

    omp_set_num_threads(8);
    #pragma omp parallel
    {
     #pragma omp for schedule(static,2) nowait
        for (i=0; i < N; i++) {
          int id=omp_get_thread_num();
          printf("Loop 1: (%d) gets iteration %d\n",id,i);

        }

     #pragma omp for schedule(static, 2) nowait
        for (i=0; i < N; i++) {
          int id=omp_get_thread_num();
          printf("Loop 2: (%d) gets iteration %d\n",id,i);

        }
    }
    return 0;
}
```

- How does the sequence of printf change if the nowait clause is removed from the first for directive?
- If the nowait clause is removed in the second for pragma, will you observe any difference?

11. Collapse

(collapse.c)

This example illustrates the use of the *collapse* directive.

```
#define N 6

int main()
{
    int i,j;

    omp_set_num_threads(8);
    #pragma omp parallel for collapse(2)
    for (i=0; i < N; i++) {
       for (j=0; j < N; j++) {
       int id=omp_get_thread_num();
       printf("(%d) Iter (%d %d)\n",id,i,j);
     }
    }
    return 0;
}
```

- Which iterations of the loops are executed by each thread when the collapse clause is used?
- Is the execution correct if we remove the collapse clause? Add the appropriate clause to make it correct.

12. Ordered

(ordered.c)

This example illustrates the use of the *ordered* directive.

```
#define N 16
int main()
{
    int i;

    omp_set_num_threads(8);
    #pragma omp parallel
    {
        #pragma omp for schedule(dynamic) ordered
        for (i=0; i < N; i++) {
        int id=omp_get_thread_num();
        printf("Before ordered - (%d) gets iteration
        %d\n",id,i);
        #pragma omp ordered
        printf("Inside ordered - (%d) gets iteration
        %d\n",id,i);
        }
    }
    return 0;
}
```

- Can you explain the order in which printf appear?
- How can you ensure that a thread always executes two consecutive iterations in order during the execution of the ordered part of the loop body?

13. Doacross

(doacross.c)

This example illustrates the use of the *depend* clause to describe dependences in a doacross loop. The *ordered (n)* clause with an integer argument *n* is used to define the number of loops within the doacross nest.

```
#define N 16
#define M 5
```

```
int main()
{
  float a[N], b[N], c[N];
  float a1[M][M], b1[M][M], c1[M][M];

  omp_set_num_threads(8);
  #pragma omp parallel for ordered(1) schedule(dynamic)
  for (int i=1; i<N; i++) {
        a[i] = 1.3;
       printf("Outside from %d executing %d\n",
       omp_get_thread_num(), i);
        #pragma omp ordered depend(sink: i-2)
      {
      printf("Inside from %d executing %d\n",
      omp_get_thread_num(), i);
      b[i] = a[i] * b[i-2];
      }
    #pragma omp ordered depend(source)
    c[i] = b[i] * 0.1234;
 }

 #pragma omp parallel for ordered(2) schedule(dynamic)
   for (int i=1; i<M; i++) {
     for (int j=1; j<M; j++) {
      a1[i][j] = 3.45;
       #pragma omp ordered depend(sink: i-1,j)
       depend(sink: i,j-1)
        {
        printf("Computing iteration %d %d\n", i, j);
         b1[i][j] = a1[i][j] * (b1[i-1][j] + b1[i][j-1]);
        sleep(1);
         }
        #pragma omp ordered depend(source)
         c1[i][j] = b1[i][j] / 2.19;
      }
    }
    return 0;
}
```

- In which order are the "Outside" and "Inside" messages printed? What would happen at the contents of arrays b and c if the depend clause is removed?
- In which order are the iterations in the second loop nest executed?

- What would happen if you remove the invocation of sleep(1). Execute several times to answer in the general case. Can you explain the order in which printf appear?

14. Tasking

(fibonacci.c)

The next example illustrate the use of task-based directives. The program computes a sequence of Fibonacci numbers by generating a linked list where each node in the list contains the index *n* (data) and the value for *Fibonacci(n)* (fibdata).

```
#define N 25


struct node {
   int data;
   int fibdata;
   int threadnum;
   struct node* next;
};

int fib(int n) {
   int x, y;
   if (n < 2) {
        return(1);
   } else {
      x = fib(n - 1);
      y = fib(n - 2);
      return (x + y);
   }
}

void processwork(struct node* p)
{
   int n;
   n = p->data;
   p->fibdata += fib(n);
   p->threadnum = omp_get_thread_num();
}

struct node* init_list(int nelems) {
    int i;
    struct node *head, *p1, *p2;

    p1 = malloc(sizeof(struct node));
```

```
    head = p1;
    p1->data = 0;
    p1->fibdata = 0;
    p1->threadnum = 0;
    for (i=0; i<nelems-1; i++) {
       p2  = malloc(sizeof(struct node));
       p1->next = p2;
       p2->data = i+1;
       p2->fibdata = 0;
       p2->threadnum = 0;
       p1 = p2;
    }
    p1->next = NULL;
    return head;
}

int main(int argc, char *argv[]) {
     struct node *p, *temp, *head;

        printf("Staring computation of Fibonacci for
        numbers in linked list \n");

        p = init_list(N);
     head = p;

     while (p != NULL) {
      #pragma omp task
      processwork(p);
      p = p->next;
      }

     printf("Finished computation of Fibonacci for
numbers in linked list \n");
     p = head;
     while (p != NULL) {
      printf("%d: %d computed by thread %d \n", p->data,
p->fibdata, p->threadnum);
         temp = p->next;
         free (p);
         p = temp;
     }
     free (p);
     return 0;
}
```

- Is the code printing what you expect? How many tasks are created and what computation is associated to each task? Is the code executing tasks in parallel?
- Insert the necessary pragmas to execute the code in parallel (Hint: you have to include a directive to identify which part of the code has to be executed in parallel; think also about whether variables have to be shared

or private to each thread; and which thread has to be responsible for task creation).

15. Taskloop

(taskloop.c)

The next example illustrate the use of *taskloop* and *taskwait* directives.

```
void long_running_task(int value) {
    printf("Thread %d going to sleep for %d seconds\n",
    omp_get_thread_num(), value);
    sleep(value);
     printf("Thread %d weaking up after a %d seconds
     siesta,   willing to work ...\n",
     omp_get_thread_num(), value);
}

void loop_body(int i, int j) {
     printf("Thread %d executing loop body (%d, %d)\n", o
     omp_get_thread_num(), i, j);
     sleep(1);
}

int main(int argc, char *argv[]) {
   #pragma omp parallel num_threads(4)
   #pragma omp single
   {
     printf("I am thread %d and going to create T1 and
     T2\n", omp_get_thread_num());
     #pragma omp task      // Task T1
     long_running_task(5);

     #pragma omp task      // Task T2
      {
         #pragma omp task   // Task T3
         long_running_task(10); // can execute concurrently

         #pragma omp task   // Task T4
          {
               #pragma omp taskloop grainsize(1) nogroup
               // Tasks TL
               for (long i = 0; i < 10; i++)
                 for (long j = 0; j < i; j++)
                   loop_body(i, j);
               printf("Thread %d finished the creation of
               all tasks in taskloop TL\n",
               omp_get_thread_num());
```

```
            }
            printf("Thread %d finished the execution of task
            creating T3 and T4\n", omp_get_thread_num());
         }
       printf("I am still thread %d after creating T1 and T2,
       ready to enter in the taskwait\n",
       omp_get_thread_num());
       #pragma omp taskwait
       printf("I am still thread %d, but now after exiting
       from the taskwait\n", omp_get_thread_num());
    }
    return 0;
}
```

- Execute the program several times and make sure you are able to explain when each thread in the threads team is actually contributing to the execution of work (tasks) generated in the *taskloop*.