OpenMP Exercises

CPD

Jorge Barbosa

1. Parallel loops

Using the first (line x column) or the second (line x line) implementations of the matrix product, analyze the next two solutions, and verify their performance.

```
#pragma omp parallel for
for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      for (int k=0; k<n; k++)
      {        }
```

```
#pragma omp parallel
for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      #pragma omp for
      for (int k=0; k<n; k++)
      {        }
```

Note:
Command to compile a OpenMP program: g++ file.cpp -fopenmp -o <file>

2. Loop scheduling

Download loop_scheduling.cpp and analyze the execution with different pragmas for loop scheduling, as indicated in the code.

3. PIPELINE 1

Download the pipeline1.cpp and measure the performance of this code in terms of time. The program is a simulation to study some OpenMP pragmas. It consists in reading 4 (N) files and then each file is split in 10 chunks (ProcessingNum), which are processed by 10 threads in parallel. After computation each result is written to a file.
Note that, instead of reading, processing and writing, the simulation represents each operation by a specific elapsed time in seconds.

```
for (i=0; i<N; i++) {
        ReadFromFile(i,...);

        for (j=0; j<ProcessingNum; j++)
                ProcessData(); /* here is the work */

        WriteResultsToFile(i);
}
```

4. PIPELINE 2

Download the pipeline2.cpp and compare the performance to the pipeline1.cpp in terms of time.

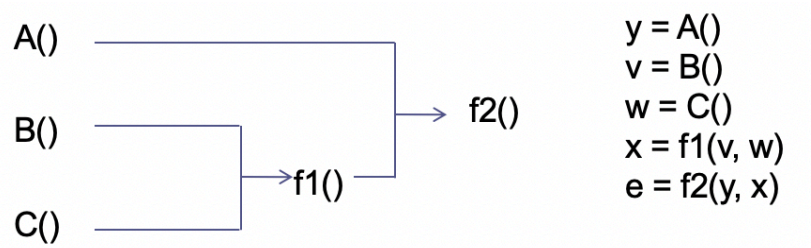
```
#pragma omp parallel private(i)
{
    /* preload data to be used in first iteration of the i-loop */
(1) #pragma omp single
            {ReadFromFile(0,...);}

    for (i=0; i<N; i++) {
            /* preload data for next iteration of the i-loop */
            #pragma omp single nowait
(2)         {ReadFromFile(i+1...);}

(3)         #pragma omp for schedule(dynamic)
            for (j=0; j<ProcessingNum; j++)
                ProcessChunkOfData(); /* here is the work */
            /* there is a barrier at the end of this loop */

(4)         #pragma omp single nowait
                {WriteResultsToFile(i);}

    } /* threads immediately move on to next iteration of i-loop */

} /* one parallel region encloses all the work */
/* Fig 5.28 from "Using OpenMP" */
```

Questions:

- Why we need **private(i)** ?
- Why in (1) we do not put NOWAIT like in (2) and (4)?
- Why we use **dynamic** in (3)? If we put **static**, or just remove it, what really happens?

5. Functional parallelism

a) Consider the following task dependencies:



Implement a solution as shown below. Suppose that functions take the following processing time:
A(): 7s
B(): 6s
C(): 5s
f1(): 4s
f2(): 5s

```
#pragma omp parallel
{
   #pragma omp single nowait
   {
        #pragma omp task shared(a)
            a = A();
        #pragma omp task shared(b)
            b = B();
        #pragma omp task shared(c)
            c = C();

        #pragma omp taskwait

        x = f1(b,c)

        y = f2(a,x)
   }
}
```

- Implement a program to run this experiment.
- Draw a time line showing the processing of the functions.

Questions:
- What is the execution time of this solution?
- What can be improved to reduce the execution time?

b) Compare the former solution to this one in terms of time.

```
#pragma omp parallel
{
   #pragma omp single nowait
   {
        #pragma omp task shared(y)
           y = A();
        #pragma omp taskgroup
        {
                #pragma omp task shared(v)
                    v = B();
                #pragma omp task shared(w)
                    w = C();
        }
        x = f1(v,w)
        #pragma omp taskwait
        e = f2(y,x)
   }
}
```

Questions:
- What is the execution time of this solution?
- Is x computed correctly?
- Do we need the *taskwait* pragma before computing *e* ?

c) And to this one:

```
#pragma omp parallel
   {
   #pragma omp sections
      {
         b = B();
      #pragma omp section
         c = C();
      }
   #pragma omp sections
      {
         x = f1(b, c);
      #pragma omp section
         a = A();
      }
   }
   printf ("%6.2f\n", f2(a,x));
```

Question:
- What is the execution time of this solution?

6. Implement Pipeline2.cpp with tasks.

7. Analyze the Quick Sort parallel code