Practical Work: Parallelization

Master CSMI
Compilation & Performance
Bérenger Bramas

November 18, 2020

1 Summary

In this current work, you will parallelize code using OpenMP.

2 Ressources

- OpenMP 4.0 API C/C++ Syntax Quick Reference Card: `https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf`

Section 7 provides a nice example of OpenMP code.

3 Practical work organization (always the same)

In the practical work, you will obtain the code from my repository and push it to your repository. Therefore, you will have to clone one branch per session and push it to your own repository. It is mandatory that you **commit and push** frequently (after each question and at the end of the session, at least) such that I can easily look at what you have coded at the end of the session, how you did progress (and potentially compare it with the latest version you will have).

It is required that you filled the report.md file to let me know what you did.

In the rest of the document, we consider you have a repository named *cnp-tp-2020* on *git.unistra.fr* that is private but that I can access in read.

3.1 Get the practical work

Consider you are in your project directory do the following:

```
# Clone my repo
git clone https://git.unistra.fr/bbramas/csmi-tp-2020.git --branch=TP5 csmi-tp5
# If you use SSH, use:
# git clone git@git.unistra.fr:bbramas/csmi-tp-2020.git --branch=TP5 csmi-tp5
# Go in the newly created directory
cd csmi-tp5
```

3.2 Add your repository as remote

You will push on your own repository:

```
# Rename my remote
git remote rename origin old-origin
# Add your own remote
git remote add origin https://git.unistra.fr/[YOU LOGIN HERE]/cnp-tp-2020.git
# If you use an SSH key:
# git remote add origin git@git.unistra.fr:[YOU LOGIN HERE]/cnp-tp-2020.git
# Push the current branch and active the tracking
git push -u origin TP5
```

3.3 During the session and while you work on the project

After each question or important modification push the current changes:

```
# No matter where you are in the project directory
git commit -a -m "I did something"
git push
```

3.4 When you are done

You have fully finished your work (at most D+14 H-2):

```
git commit -a -m "I did something"
git push
```

3.5 Important!

Do not forget that you have to fill the report.md file and commit it. Remember to commit regularly to keep track of your work and let me see a history of it if I need it. Do not share any code with someone else, as I am here to answer all questions and support all of you. Remember that you have questions to answer in the moodle before the end of the session and that you must push your branch at the end of the session too. Additional credits can be obtained if you make some modifications after the session to improve your solution. Changes can be made until two weeks after the session minus two hours, ie you must push before the beginning of the n+2 practical work. Do not remove code from the test functions as I use them to evaluate your code. Therefore, if you need you can add extra functions for your own testing/debugging. If some of them are showing interesting things about your code, simply leave a comment in the code and the report.md.

3.6 Compilation

To compile, we use CMake:

```
cd TP5
mkdir build
cd build
cmake ..
make # Will make all
make something # Will build only something
VERBOSE=1 make # Will show the commands used to compile (including the flags)
```

4 Reminder

Modern CPU have multiple cores, where each core can execute a thread/process. Both processes and threads are independent sequences of execution. The main difference between them is that threads that belong to the same process run in a shared memory space, while processes run in separate memory spaces. So threads can use the same variables.

To create threads, Linux proposes the "clone" function, which is really low level and requires to really understand what should be shared between the different threads/processes. At a little higher level, it is common to use the POSIX thread API. But this API requires to write lots of code and to add many callback functions. An alternative, as we have seen in class, is to use OpenMP. This is what you will use here.

5 Generate the same output

As a first exercise, you have to create the code to generate output of the following form:

```
print "Start parallel region"
each thread prints "I am thread THREAD_ID/THREAD_NUMBER
                Available cores = LIST_OF_CORES" in any order
master thread prints "I am master, all threads done CURRENT_CORE"
each thread prints "I am thread THREAD_ID" in order
```

Note that the first print of "I am thread" may have a different order and that the list cores might be different. Example of real output:

```
$ OMP_NUM_THREADS=4 ./simplyparallel
Start parallel region
I am thread 0/4 Available cores = 0  1  2  3  4  5  6  7
I am thread 1/4 Available cores = 0  1  2  3  4  5  6  7
I am thread 3/4 Available cores = 0  1  2  3  4  5  6  7
I am thread 2/4 Available cores = 0  1  2  3  4  5  6  7
I am master, all threads done 0
I am thread 0
I am thread 1
I am thread 2
I am thread 3
```

Or with pinning of the threads:

```
$ OMP_PROC_BIND=true OMP_NUM_THREADS=4 ./simplyparallel
Start parallel region
I am thread 1/4 Available cores = 4
I am thread 0/4 Available cores = 0
I am thread 3/4 Available cores = 5
I am thread 2/4 Available cores = 1
I am master, all threads done 0
I am thread 0
I am thread 1
I am thread 2
I am thread 3
```

The functions to have the list of available cores are already included. From OpenMP, you will need the following functions/pragma:

- "#pragma omp parallel": create a parallel region (using the default number of threads)
- "#pragma omp critical": create a critical section (one thread at a time will execute it)
- "#pragma omp barrier": create a barrier (all threads will pass the barrier together or will not pass)
- "#pragma omp master": create a section that only the master thread can execute
- *omp_get_thread_num*(): return the id of the thread that calls the function
- *omp_get_num_threads*(): return the number of threads in the latest parallel region

6 Parallelize the dot kernel

Update the *dotOmp* function to make it parallel. To do so use the OpenMP pragma for parallel for and tell OpenMP that the *sum* variable should be reduced by summation. Execute the kernel with different number of threads and with/without pinning.

Output on my machine:

```
$ ./dot
Check dot
TestSize = 500000
There will be 8 threads
>> Sequential exec timer : 1.17595
>> Parallel exec timer : 0.301933
```

7 Task-and-sync parallelization of Fibonnaci

Update the *FibonacciOmp* function in file fibonnaci.cpp to have a parallel execution. As you can see in the main, the parallel section already exists and only the master thread enter the scope that will call *FibonacciOmp*. But the other threads are waiting for some work to do, and we can give them work using the OpenMP task pragma. However, when doing so, we have to carefuly wait for the completion of the tasks.

In the next example, the thread that executes this code will create two tasks (that can be executed by other threads), then it calls $again_another_func$, and finally wait for the completion of the tasks. Therefore, at the end of the code the three functions have been executed and are over.

```
#pragma omp parallel // create the threads
{
    // here there might be 10 threads
    #pragma omp master // create a section for the master thread only
                       // the other threads will continue
    {
        // Create a task that can be executed by any threads
        #pragma omp task
        a_func();
        // Create a task that can be executed by any threads
        #pragma omp task
        another_func();

        // The master thread will do this
        again_another_func();

        // The master threads wait for the two task to finish
        #pragma omp taskwait
    }
} // The other threads are waiting here
```

In our case, the recursive calls can be transformed into tasks. Once you have a first version, perform some testing and benchmark.

Then, apply these three optimizations:

- Avoid creating two tasks: creating only one task is enough has the current thread could perform the second call
- Avoid createing tasks for $n < 15$: if the tasks are too tiny, their overhead will be higher than their benefit
- Use higher priority for tasks at the top (less recursion level)

Remark: Do not forget that if you meet a recursive kernel in the real life, pay attention if dynamic programming could not help you to avoid re-computing values that were alread computed.

```
$ ./fibonnaci
Check Fibonacci
TestSize = 40
>> Sequential timer : 5.82901
>> There are 8 threads
>> Omp timer : 2.01792
```

8 Task-based (tasks and dependencies)

In many applications, we do not have a single loop to parallelize, and the execution times of the paths that can run in parallel may very unbalanced. One of the more dynamic paradigm is the task-based (tasks and dependencies) approach. As you already know, we have to inform about the dependencies for each tasks to ensure that OpenMP will be able to generate a graph of tasks and execute it while respecting the dependencies.

In the taskbased.cpp file, update the *DumbExecutionOmp* function by add the appropriate dependencies keywords in the omp task pragma. A correct execution will show you the following message: *Execution succeed!*.