UCLA-CS33-Spring22-OpenMP

This is the OpenMP lab for UCLA CS33 Spring 2022. The leading TA of this lab is Zhengrong Wang (seanzw@ucla.edu).

Introduction

In this lab, you will use OpenMP to parallelize an important kernel which is widely used in DNA sequencing algorithms. Before starting this lab, please first make sure you followed some tutorials on OpenMP and have the basic understanding.

First log into your seas lab server (`lnxsrv07.seas.ucla.edu`) and clone this repo:

```
git clone https://github.com/seanzw/UCLA-CS33-Spring22-OpenMP.git
cd UCLA-CS33-Spring22-OpenMP
```

The folder contains the following files:

```
main.c        // Driving code, with the main function.
kernel_seq.c  // The sequential implementation of the kernel.
kernel_omp.c  // Parallelized implementation of the kernel.
```

The baseline implementation (sequential, not parallelized) is in `kernel_seq.c` . You can run the sequential version by:

```
make seq
./seq 8192 1
```

The main function takes two arguments: - The size of the input matrix. For simplicity, here we assume it must be multiple of 128. - The number of threads to use. For sequential baseline, it ignores this argument and always uses 1 thread.

So the above command will run the sequential baseline with a matrix of size `8192x8192` and 1 thread. It will report the time of running the kernel:

```
Num of threads: 1. Matrix size 8193x8193
Total time: 0.195 seconds
```

The actual running time on your machine may be different. Also, it reports the matrix size as `8193x8193` as the algorithm requires padding the matrix by 1.

Now let's run the parallelized version and check if the result is correct.

```
make check
./check 8192 1
```

Since your haven't implemented the parallelized version, `kernel_omp()` is now just an empty function, and the result of course will not match with `kernel_seq()` . So you should expect the output like this:

```
Num of threads: 1. Matrix size 8193x8193
Total time: 0.000 seconds
Wrong result at 1x1 -2 != 0.
```

So the overall goal of this lab is to implement `kernel_omp()` and achieve high speedup while still producing the correct result!

You should only edit `kernel_omp.c` . To build parallelized version and run it with 4 threads.

```
make omp
./omp 8192 4
```

Grading

You will submit two files: `kernel_omp.c` and `omp_lab.txt`. The first one is your parallelized implementation. The second one is your experiment report (how you parallelized it, how you choose any parameters and the impact on the performance).

We will test with input size `2048x2048` and 4 threads. Your score will be proportional to the speedup you achieved, starting from `1.0x` (0 point, no speedup) to `2.8x` (100 point). The testing will be conducted on seas lab servers. For every extra `0.1x` speedup over `2.8x`, you will get extra 1 point (at most 5 extra points). Also, your implementation should generate correct result to receive any points.

NOTE: For this lab, the evaluation will be conducted on the server `lnxsrv07.seas.ucla.edu`, NOT `cs33.seas.ucla.edu`! Just change the server host name before logging in.

Hint

NOTE: Please try to solve the lab before reading this hint! It will be much more fun and you will learn more if you figure it out by yourself!

The kernel in `kernel_seq.c` has inter-iteration dependence: the iteration `(i, j)` depends on iteration `(i-1, j)`, `(i, j-1)` and `(i-1, j-1)`. This makes it impossible to parallize along rows or columns, as you breaks the dependence and will get wrong results.

However, if you process the array in an anti-diagonal fashion, i.e. computing in the following order:

```
// Each (row, col) represents one element in the array.
(1, 1)
(2, 1), (1, 2)
(3, 1), (2, 2), (1, 3)
(4, 1), (3, 2), (2, 3), (1, 4)
...
```

You will have no dependence between elements within the same line, and they can be easily parallelized. For example, these four elements can be processed in parallel:

```
(4, 1), (3, 2), (2, 3), (1, 4)
```

However, this parallization scheme makes the memory access pattern very unfriendly to the hardware and cache, since the core now accesses in anti-diagonal fashion. To deal with this, a common techinque is "tiling" or "blocking". Instead of processing each individual elements in anti-diagonal order, you can break the matrix into small blocks, and process each block in the original row-column order, while processing blocks in anti-diagonal order.