MATH 5490 — HPC-xTC

Home Work #2 **Group 5** Jorge Flores, Xiukun Hu, Grigorii Sarnitskii

1 Step 2

1.1 Results

The code is capable with square blocks (i.e. with `blk_rows` and `blk_cols` to be the same) and multiple blocks in A and B. The multiplication result will be written to disk in file(s) `C.*.*`, and can be printed on screen.

The result is examed by Matlab for 100×100, 1000×1000 and 5000×5000 block multiplication. The absolute difference between two results is less than $1e-11$ for all tests.

Total time spent for 5000×5000 matrices multiplication for this code is $1.37e3$ seconds and computing time is $1.36e3$ seconds, while for Matlab the computing time is only 4.28 seconds.

1.2 Requirement

Unzip the folder and every file needed is inside. Use `make run` command to run it.

1.3 Compute Bound

For MacBook Pro with 2.7GHz Intel Core i5, 8G DDR3 memory, it gets compute bound when the block is over 50×50.

For Lenovo with 2.4GHz Intel Core i7, running in a 64bit Red Hat virtual machine, it gets compute bound when the block is over 50×50.

2 Step 3

2.1 Algorithm

The pseudocode is as follow:

```
OMP PARALLEL
  OMP single
    ablock[tog] ← READ(A_0_0)                          ▷ tog initialized as 0
    bblock[tog] ← READ(B_0_0)
  ENDOMP single                                        ▷ Implicit barrier here
  while i < rows of blocks in C do
    OMP single nowait
      ablock[1 − tog] ←READ(next A block)
      bblock[1 − tog] ←READ(next B block)
    ENDOMP single nowait
    OMP for nowait dynamic
      for every continuous WIDTH (macro) elements in ablock do
        for each column of B do
          temp+ = WIDTH elements of ablock[tog]∗WIDTH rows of bblock[tog]
          OMP atomic update
            cblock[ctog] corresponding element + = temp
          ENDOMP atomic update
        end for
      end for
    ENDOMP for nowait dynamic
    if blk_cols cannot be divided by WIDTH then
      OMP for nowait dynamic
        for each row of the remainder columns in ablock[tog] do
          update cblock[ctog]
        end for
      ENDOMP for nowait dynamic
    end if
   OMP BARRIER
    if k == columns of blocks in A then
      OMP single nowait
        WRITE cblock[ctog]
        fill cblock[ctog] with zeros
      ENDOMP single nowait
      ctog = 1 − ctog
    end if
    tog = 1 − tog
    update i, j, k to point to next block in A and B;
  end while
ENDOMP PARALLEL
```

2.2 Results

The code works for square blocks (i.e. with `blk_rows` and `blk_cols` being equal) and multiple blocks in A and B. The multiplication result will be written to disk in file(s) of the form `C.*.*`.

The result was verified to be correct using MATLAB for 1000×1000 block size and 10×10 blocks matrix multiplication.

We performed a one dimensional analysis on the performance of our code using different block sizes and block matrix sizes. These tests where performed on an Intel(R) Xeon(R) CPU E5-4650 @2.70GHz with 8 cores and 2 threads per core (for a total of 16 threads). The results are summarized in the table below, were the total time represents the time it took our code to multiply the matrices and write the resulting blocks of C to disk.

Block/Matrix Size Analysis		
Block Size	Block Matrix Size	Total Time (seconds)
100x100	1x1	1.130199×10^{-2}
100x100	2x2	5.166101×10^{-2}
100x100	5x5	6.825109×10^{-1}
1000x1000	1x1	2.701371×10^{-1}
1000x1000	2x2	1.635549
1000x1000	5x5	22.65355

It is obvious that the total time is $O(n^3)$.

We also performed a one dimensional analysis, this time focusing on the performance of our code using different thread numbers. These tests were performed on an Intel(R) Xeon(R) CPU E5-4650 @2.70GHz with 8 cores and 2 threads per core (for a total of 16 threads). The results are summarized in the table below.

Thread Number Analysis			
Number of Threads	Block Block Size	Block Matrix Size	Total Time (seconds)
4	1000x1000	5x5	43.33175
8	1000x1000	5x5	22.61952
12	1000x1000	5x5	20.78736
16	1000x1000	5x5	23.37497

We note that increasing the thread number from 4 threads to 8 threads, the total time nearly halved. However, changing from 8 to 12 and 16 threads, we see only a small change in the total time. This might be due to the fact that the system we used only has 8 physical cores.

2.3 Requirement

Unzip the folder called hw2.zip, every file needed to run the code is inside. Use `make matrix` to first generate and write to disk the matrices (block entries for each matrix) that will be used. (Notice that the block should be square.) Then use `make run` to multiply the matrices in parallel and write the resulting matrix (in blocks) to the disk. Alternatively, use `make runserial` to multiply the matrices in serial.

2.4 Compute Bound

For a MacBook Pro with 2.7GHz Intel Core i5, 8G DDR3 memory, compute bound is achieved when the block size is over 40×40.

For a Lenovo with 2.4GHz Intel Core i7, running in a 64bit Red Hat virtual machine, compute bound is achieved when the block size is over 40×40.

For an Intel(R) Core(TM) i3-6100U CPU @ 2.30GHz running Ubuntu, compute bound was achieved when the block size is over 50×50.

The information is summarized in the following table:

Compute Bound Analysis	
System	Block Size
Intel(R) Core(TM) i7-4700MQ CPU @ 2.40Ghz GNU bash, version 4.2.46(1)-release ($x86-64$-redhat-linux-gnu), 4 threads	above 40 x 40
Intel(R) Core(TM) i3-6100U CPU @ 2.30GHz gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.2)	above 50 x 50
2.7 GHz Intel Core i5 gcc-6 (Homebrew gcc 6.2.0) 6.2.0, OS X Sierra, 4 threads	above 40 x 40