OPENMP LAB

Due Date: Friday June 8 2018, 11:59pm

Application

When you go to the hospital and get an MRI [magnetic resonance imaging] scan, it takes a stack of images as cross-sectional slices of the scanned area. Individually, these images are of very poor quality. They are blurry, noisy, and polluted by artifacts creating from the scan process. Because of this, image processing programs use the wealth of information present in all these images collectively to improve the quality of any individual imaged region. This is done by leveraging the fact that these images are all pictures of the same thing, just taken of different parts of this same object. Thus to improve the quality of any given pixel, all nearby pixels can be examined for clues of our targeted pixel's true value.

One of these image processing operations is deblurring, which takes a fuzzy image and produces a clearer image from it. This is the application that has been given to you to parallelize.

The distributed project consists of 5 source files and 2 data files.

The 2 data files contain 3D images of random static that can be used as test platforms. They are not actual scan images, but are useful to pick up algorithmic problems that may otherwise only surface probabilistically., as a result of either race conditions between threads or strange arithmetic corner-cases.

The 5 source files are main.c, containing a driver program that measures your solution's performance, seqDeblur[.h/.c] and ompDeblur[.h/.c]. seqDeblur.c implements a deblurring image pass sequentially. ompDeblur.c currently also implements a deblurring image pass sequentially, and is identical to the seqDeblur case, but you are aiming to correct that. Function prototypes are found in both .h files. The driver will measure the execution time of your OMP_Deblur, and compare it against the sequential SEQ_Deblur.

To initialize your function, OMP_Initialize will be called. To clean up your function, OMP_Finish will be called. Neither Initialize or Finish will count toward your execution time. The only function that will be timed will be OMP_Deblur.

Download and working

Edit and submit only ompDeblur.c. Grading will be based on the speed up achieved over the stock version. The expected minimum performance improvement, using whatever optimization method that you would like, will be 10-times. This means that your optimized version is expected to run in at most 1/10th the time as the unmodified version.

Steps to download and work

1. You can copy **OMPlab2018.tgz** from this path
   /w/class.1/cs/cs33/csbin/OpenMP_Spring2018_handout/
2. Untar the file to any seas machine in your directory
   tar -zxf OMPlab2018.tgz
3. This will create the OMPlab2018 directory
4. For compiling

a.	make all	This will create the deblurTest executable.
b.	make all GPROF=1	Will enable profiling.
c.	make clean	Will delete all object files.

5. Once compiled, you can run the executable
   ./deblurTest
6. Each time your modify ompDeblur.c , you need to recompile the program using steps in 4
7. Remember to add your name, ucla id and email details in the comment section in ompDeblur.c
8. You shouldn't edit seqDeblur.c as we are going to measure your ompDeblur against seqDeblur.

While this is appropriate for debugging purposes, this will not be graded. The performance of your optimized code also depends on the load (what else is running) on the system. You can use programs like top to see the load of the system. After top starts press 1 to see the load on each core of the machine.

The servers to test your code on: lnxsrv08
Remember that grading will be done on this server.

These are shared by all students in the engineering departments, including your fellow CS 33. When multiple students are running their programs the performance of your program may be drastically lower. To avoid problems testing problems, START EARLY!

Submission

Submit only **ompDeblur.c** in the CCLE. Remember to add you name, ucla id and email id in the comment section in ompDeblur.c. The last date for submission is Friday, June 8, 2018.

Profiling

When optimizing a large program, it is useful to profile it to determine which portions
take up the largest portion of the execution time. There is no point in optimizing code that

only takes up a small fraction of the overall computations. gprof is a simple profiling tool which works with gcc to give approximate statistics on how often each function is called. gprof works by interrupting your program at fixed time intervals and noting what function is currently executing.As shown earlier, to compile with gprof support, simply add GPROF=1 to the make command. Then when you run the executable, it will produce the file gmon.out containing the raw statistics. To view the statistics in a readable format, run gprof with the name of the executable.
Ex) gprof ./deblurTest

Grading

Grading will be done on lnxsrv08.seas.ucla.edu to prevent any other processes from interfering with the timing results. You will receive points proportional to the speed up:

10X speed up	– 100/100
9X speed up	– 90/100
8X speed up	– 80/100
7X speed up	– 70/100
6X speed up	– 60/100
5X speed up	– 50/100
4X speed up	– 40/100
3X speed up	– 30/100
2X speed up	– 20/100
< 2X speed up	– 0/100

"Remember, there is an opportunity to earn as much as 20 bonus points for additional speedup above 10x ! "

NOTE: If your solution does not pass "CompareResults", you will receive a 0 regardless of your speed up

Lnxsrv08 server details

Name	Lnxsrv08.seas.ucla.edu
Processors	2
Cores per processor	8
Threads / core	2
CPU	Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz

ISSUES

If you are finding any issues or trouble with the compilation or with lnxsrv08, please reach out to Akshay Shetty [akshashe@ucla.edu]