Reading questions

The first two questions are questions from last time, but worth revisiting. These are up rather late, but do what you can, and come with questions for class!

1. The class cluster consists of eight nodes and fifteen Xeon Phi accelerator boards. Based on an online search for information on these systems, what do you think is the theoretical peak flop rate (double-precision floating point operations per second)? Show how you computed this, and give URLs for where you got the parameters in your calculation. (We will return to this question again after we cover some computer architecture.)

From the course website, we know the machine has "fifteen Xeon Phi 5110P boards hosted in eight 12-core compute nodes consisting of Intel Xeon E5-2620 v3 processors".

Xeon Phi 5110P boards

from:

http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-theoretical-maximums.html

we know that flops of Theoretical Peak single precision FLOPS of a Xeon Phi 5110P is 32 FLOPS/cycle x 60 cores x 1.053 GHz = 2021.76 GF/s.

So For 15 Xeon Phi 5110P, the Theoretical Peak (single precision) FLOPS is 15*1021.76GF/s=15326.4 GF/s

(2 flops/FMA ×8 FMA/vector FMA ×2 vector FMAs/cycle = 32 FLOPS/cycle single precision)

For Intel® Xeon® Processor E5-2620 v3
http://ark.intel.com/products/83352/Intel-Xeon-Processor-E5-2620-v3-15M-Cache-2_40-GHz

8machine *32 FLOPS/cycle x 12 cores x 3.2 GHz = 9830.4 GF/s

so together 15326.4+ 9830.4=25156.8GF/s

2. What is the approximate theoretical peak flop rate for your own machine?

From:

http://ark.intel.com/products/72056/Intel-Core-i5-3230M-Processor-3M-Cache-up-to-

3_20-GHz-BGA

16FLOPS/clock x 2 cores x 2.6 GHz = 83.2 FLOPS

Not sure whether 16 FLOPS/clock is a correct, I choose it because according to the slides, the length of vectors of AVX is half the length on Xeon Phi

3. Suppose there are t tasks that can be executed in a pipeline with p stages. What is the speedup over serial execution of the same tasks?

t*p/(t+p-1)

4. Consider the following list of tasks (assume they can't be pipelined):

compile GCC (1 hr)
compile OpenMPI (0.5 hr) - depends on GCC
compile OpenBLAS (0.25 hr) - depends on GCC
compile LAPACK (0.5 hr) - depends on GCC and OpenBLAS
compile application (0.5 hr) - depends on GCC, OpenMPI, OpenBLAS, LAPACK

What is the minimum serial time between starting to compile and having a compiled application? What is the minimum parallel time given an arbitrary number of processors?

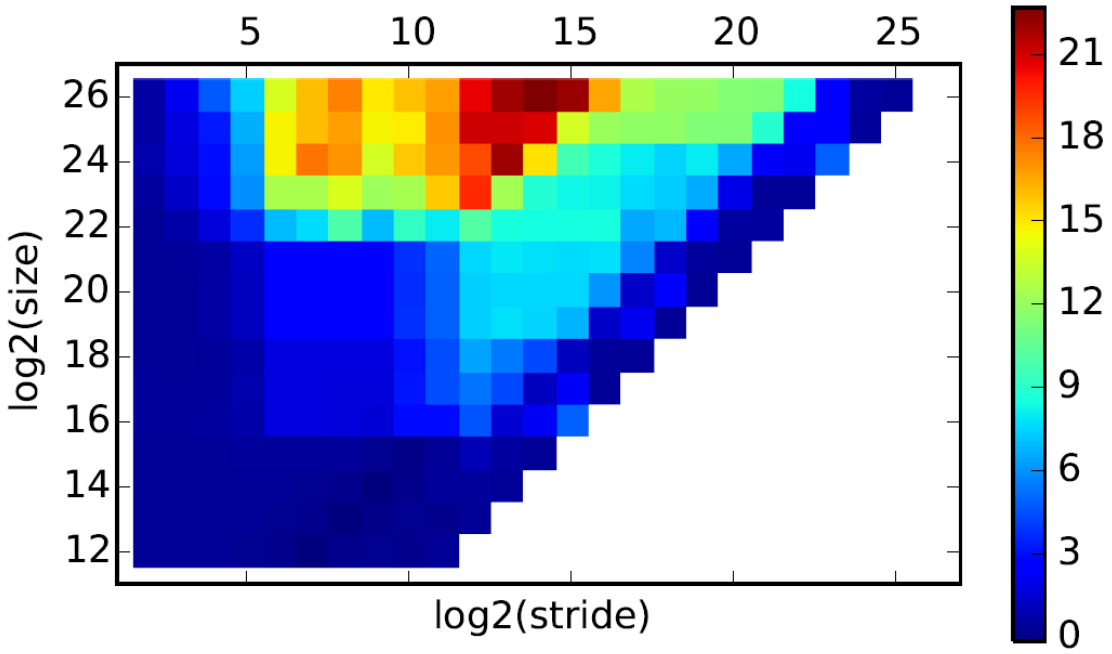
the minimum serial time is 1+0.5+0.25+0.5+0.5=2.75hour
As GCC-> BLAS->LAPACK->application must be done in serial,
the minimum parallel time is 1+0.25+0.5+0.5=2.25

5. Clone the membench repository from GitHub:

git clone git@github.com:cornell-cs5220-f15/membench.git

On your own machine, build `membench` and generate the associated plots; for many of you, this should be as simple as typing `make` at the terminal (though I assume you have Python with pandas and Matplotlib installed; see also the note about Clang and OpenMP in the leading comments of the Makefile). Look at the output file timings-heat.pdf; what can you tell about the cache architecture on your machine from the plot?

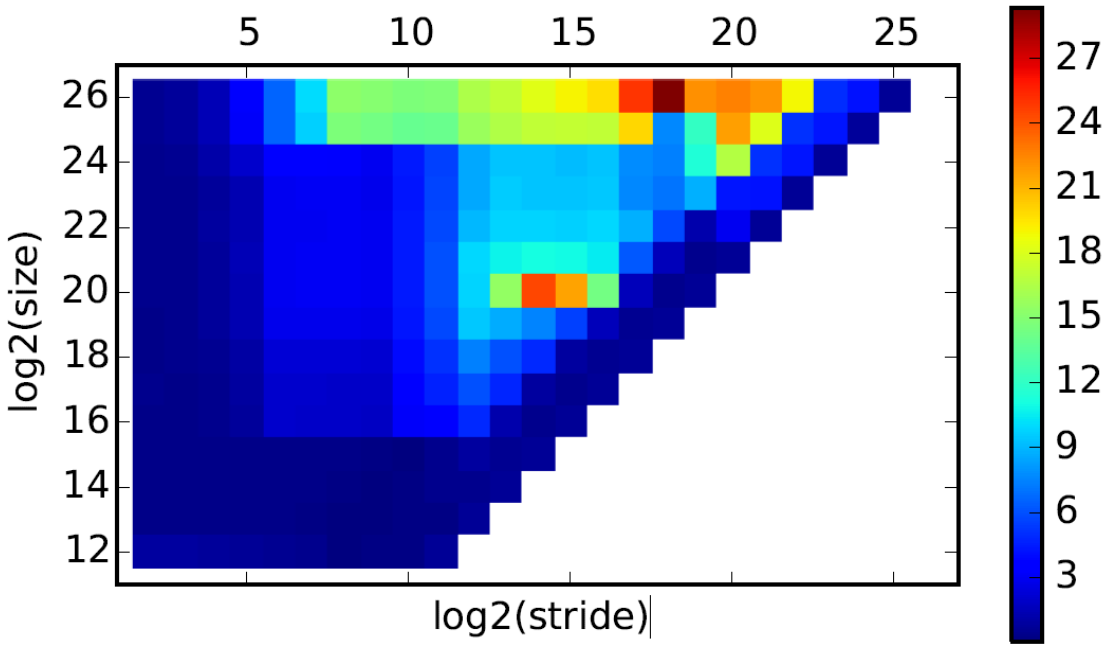
each page size should be about 2^14 long.



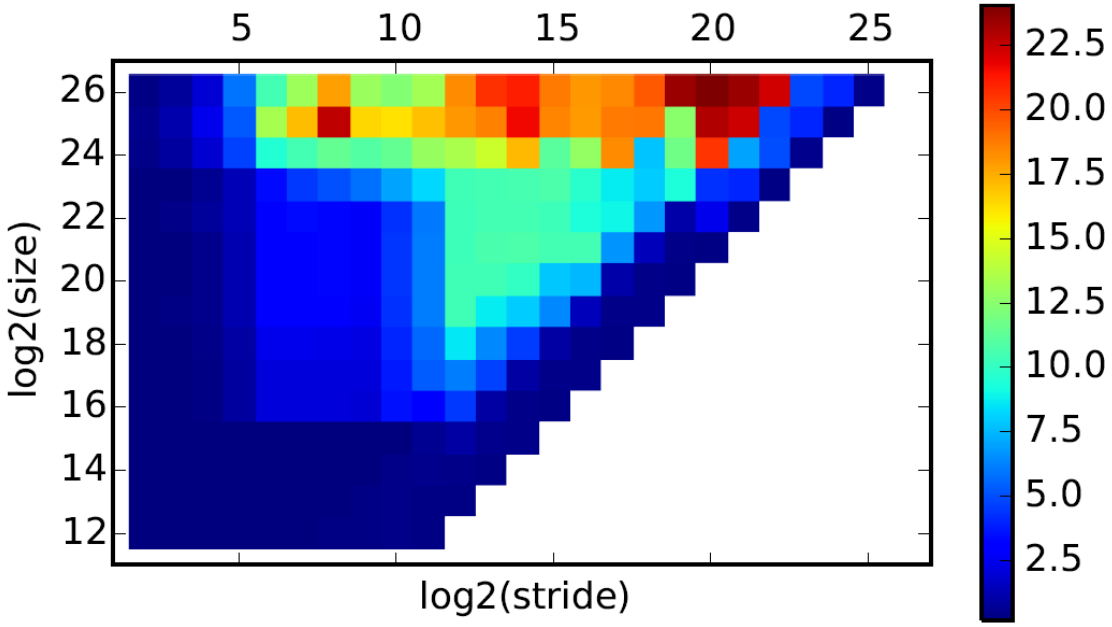
6. From the cloned repository, check out the totient branch:

    git checkout totient

    You may need to move generated files out of the way to do this. If you prefer, you can also look at the files on GitHub. Either way, repeat the exercise of problem 5. What can you tell about the cache architecture of the totient nodes?
    (on head nodes)The page size has a size of about 2^18byte



(on working nodes)The page size has a size of about 2^20 (2M) byte



7. Implement the following three methods of computing the centroid of a million two-dimensional coordinates (double precision). Time and determine which is faster:

   a. Store an array of (x,y) coordinates; loop i and simultaneously sum the xi and yi

   b. Store an array of (x,y) coordinates; loop i and sum the xi, then sum the yi in a separate loop

   c. Store the xi in one array, the yi in a second array. Sum the xi, then sum the yi.

   I recommend doing this on the class cluster using the Intel compiler. To do this, run "module load cs5220" and run (e.g.)

   icc -o centroid centroid.c

Using gcc, The result is

Version 1: 3.600000e-03

Version 2: 7.000000e-03

Version 3: 7.000000e-03

Using icc is all 0.

I think the reason that first version is faster is that it

1) It make good use of cache and register.
2) It can be better run in parallel. Since it calculate sum_x and sum_y alternatively, it has better property in running parallel. For the version 2 and version 3, we cannot run the next instruction before the result comes out.

From the result, I think the second reason is the main cause of the difference in performance.