Programming for performance (2013)
Lab Session 5

(Note: You can use the gcc or the icc compiler for the problems in this assignment. The results may be different depending on the compiler used, so please report the compiler and compiler flags that were used). Enlist all the (meaningful!) optimizations you have attempted.)

1. Membench

i. Get membench.tgz You can do this by typing
ii. Type "make" to create the membench executable. (You may need to change the entries in the Makefile for the make to work correctly).
iii. To run the program, type "make run_sge". This will automagically create a script that runs membench on the compute nodes (creates qsub batch job !!).
iv. You should *not* run membench directly on the front-end node. . You can see whether your job is waiting, running, or complete by typing the command "qstat -u username" at the shell.
v. Use the "make run_serial" command to run membench on the interactive nodes.
vi. When the program is done running, type "make membench.pdf" in order to make nice-looking plots of the raw output file (membench.out).
vii. Download membench.pdf and membench.out to your local machine. You might be interested in looking at membench.pdf. The cache line length is 64 bytes; if you look at the behavior at stride 64, can you see the four basic memory access times for a hit in L1, a hit in L2, a hit in L3, and an access to main memory?
viii. You can now try to identify how the features of the memory hierarchy show up in the plots. And you are expected to give solid reasons for the observations that you claim. For ex, when you say that a TLB misses are the reason for the low performance for a array of size X and with stride Y, then you have to submit the relevant profiling data to substantiating that. You can run any profiling tool (perf, perfsuite on NCSA, perfexpert on TACC systems Or even PAPI) to analyze the performance and obtain the relevant performance counter data. So, for this you can have a separate membench1.c with just the core kernel of membench, where you can specify at the command line the stride and array size parameters.

To be submitted:

i. membench.pdf ii. observations.pdf (Please use the same names as given here).

What's in the "observations.pdf": Name of the students (at the top of the first page), and the observations from the plot as descriptive as it is in the document, http://www.cs.cornell.edu/~bindel/class/cs5220-f11/notes/lec02.pdf (pages 9-11) along with profiling data.

2. Matrix Matrix Multiplication

Take a glance at matrixMultiply.c. You'll notice that the file contains multiple implementations of matrix multiply with 3 nested loops. Compile and run this code with the following command: "make ex1"

Note that *it is important here that we use the '-O3' flag to turn on compiler optimizations*. The makefile will run matrixMultiply twice. Copy the results somewhere so that you do not have to run this program again and use them to help you answer the following questions:

a. ***Why does performance drop for large values of n?***
b. The second run of matrixMultiply runs all 6 different loop orderings. Which ordering(s) perform best for 1000-by-1000 matrices? Which ordering(s) perform the worst? **How does the way we stride through the matrices with respect to the innermost loop affect performance? What is the miss ratio for all the 6 orderings?**

3. Matrix Transposition

Compile and run the naive matrix transposition implemented in transpose.c (*make sure to use the '-O3' flag: gcc -o transpose -O3 transpose.c*).

a. Note the time required to perform the naive transposition for a matrix of size 2000-by-2000.
b. Modify the function called "transpose" in transpose.c to implement a single level of cache blocking. That is, loop over all matrix blocks and transpose each into the destination matrix. *Make sure to handle special cases of the transposition:* What if we tried to transpose the 5-by-5 matrix above with a blocksize of 2?
c. Try block sizes of 2-by-2, 100-by-100, 200-by-200, 400-by-400 and 1000-by-1000. **Which performs best on the 2000-by-2000 matrix? Which performs worst?**
d. Is your algorithm works if you run the code with a block size of 30-by-30 ?