Optimization Lab

Goals:

- See how compiler options help you optimize the performance of hand-coded routines.
- See how performance can be improved by calling numerical libraries.

You will be working with three different versions of a code to solve a system of linear equations via LU factorization. The tarball contains all three codes, together with a makefile to compile them and a script to submit them to the scheduler. Embedded in the script are instructions that time each code and put the timing data into an output file.

Unpack the source code with:

```
$ cd
$ tar zxf ~train100/labs/ludecomp.tgz
```

You will find a directory called ludecomp. In it are these three code versions:

- nr.c – uses code copied from a book called Numerical Recipes. It does not require any external libraries.
- gsl.c – calls the GNU Scientific Library. You need to use the module command to link and load this library in the Ranger environment.
- lapack.c – calls the standard interface to LAPACK. This call may be linked against any compatible, optimized library that performs linear algebra: either the Math Kernel Library if you are using the Intel compiler, or one of the PGI LAPACK libraries if you are using the PGI compiler. Load the appropriate module for whichever compiler you choose.

The Makefile has two targets you might want to use:

- make - This makes all three versions of the program: nr, gsl, and lapack. If make fails, it is likely because it wants libraries loaded that are not currently loaded.
- make clean - This deletes all binaries you compiled for a clean start if you are switching from Intel to PGI, or vice versa.

To get started, here are the steps to follow:

1. Add gsl to the currently-loaded modules.
2. Ensure pgi is the current compiler, again using the module command.
3. Type make in the directory `~/ludecomp`
4. Submit the job.sge script with your account in it. One way to do this is `qsub -A your_account job.sge`
5. The results of your runs will be in results.txt.

Now have a look at the codes and evaluate each based on these criteria:

1. How many lines of code did it take to implement the solution?
2. How fast does it run? Look at results.txt when it finishes.
3. For each version, how hard would it be to swap in a different algorithm by, for instance, substituting an iterative solver, or using a sparse-matrix solver?
4. Can any of these codes run multithreaded? Can they run distributed, using MPI? You may need to Google the library to figure this out.

The Makefile you were given used -g, which is the debug suite of options. Next let's experiment with the two compilers to assess how the optimization options affect running times. Everybody's results will be recorded on The Eric Chen Scoreboard, http://consultrh5.cac.cornell.edu/emc256/intro_to_ranger/ .

1. Return to the directory `~/ludecomp`
2. Use the module command to select a compiler (PGI or Intel) and a matching math library that supports LAPACK (e.g. , ACML for PGI, or MKL for Intel).
3. Edit the first few lines of the Makefile so that the COMPILER matches your module choice in the shell: pgcc for PGI, icc for Intel. Possible FFLAGS are listed below. Choose an appropriate LAPACKLIB as well.
4. Compile the three codes with make.
5. Submit the codes to the scheduler with `qsub -A your_account job.sge`
6. Examine results.txt, which will appear after the job runs; compare the times the codes took to run.
7. Try some other choices of compiler and optimizations and see what is fastest. For the codes that call libraries, how does your choice of options affect the performance? (Remember, you're not compiling the libraries!)

Here are the recommended choices.
PGI: -O3 -fast -tp barcelona-64
Intel: -O3 -xW -ipo
Don't exceed -O2 without checking whether your output is correct.

For **PGI**, here are some other common compiler options to try:
-O3 - Performs some compile time and memory intensive optimizations in addition to those executed with -O2, but may not improve performance for all programs.
-Mipa=fast,inline - Creates inter-procedural optimizations. There is a loader problem with this option.
-tp barcelona-64 - Includes specialized code for the AMD Barcelona chip.
-fast - A catchall for: -O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse -Mscalarsse -Mcache_align -Mflushz
-g, -gopt - Produces better debugging information (without disabling optimization).
-mp - Enables the parallelizer to generate multi-threaded code based on the OpenMP directives.
-Minfo=mpi,ipa - Provides information about OpenMP, and inter-procedural optimization.

For **Intel**, here are some other common compiler options to try:
-O3 - More than O2, but maybe not faster.
-ipo - Creates inter-procedural optimizations.
-vec_report[0|..|5] - Controls the amount of vectorizer diagnostic information.
-xW - Includes specialized SSE and SSE2 instructions (recommended).
-xO - Includes specialized SSE3 instructions.
-fast - Includes: -ipo, -O2, -static [DO NOT USE] static load not allowed because only dynamic loading is allowed.
-g - Produces better debugging information.
-openmp - Enable OpenMP directives
-openmp_report[0|1|2] - OpenMP parallelizer diagnostic level.

Extra credit: The ACML_MP and MKL libraries have built-in support for OpenMP multithreading. To enable it, you simply set the following environment variable:

```
export OMP_NUM_THREADS=4
```

Try this! Specify either acml_mp or mkl as the LAPACKLIB in the Makefile, run make, uncomment the above line in job.sge, and submit the job. Does the time improve or not, when compared to leaving this variable unset? What if you try a different number of threads? (Remember, Ranger nodes have 4 processors and a total of 16 cores. Note, the default value of OMP_NUM_THREADS is 1, so that when 16 MPI processes share a node they won't create chaos by forking 16xN threads.)