CSCE 435 Parallel Computing, Fall 2017
Project 1: parallel matrix multiplication

Tim Davis

Assigned Monday, Sept 25. Due at 2pm on Tuesday, Oct 3

In this project you will write three matrix multiplication methods and compare their performance:

- The first one is the simple method in Problem 2.4 in the book. This method will also be used to multiply each block in the second and third methods.
- The second method is a sequential one, with no OpenMP parallelism, but it computes its result `C=A*B` by blocks, as in the solution to Problem 2.5 from homework 1.
- The third method is just like the second method, except that it uses 20 threads (the number of cores on each node of ADA) to compute the blocks of `C` in parallel.

Each of the methods will be test on a single set of square matrices of size 2048, but with different block sizes for methods 2 and 3.

I have written some of this project for you to get you started. See the `proj1.zip` file. The `mult.c` file contains the main test program and skeletons of the three functions you need to write. It also provides a helper macro called `SUBMATRIX` that gives you a simple method of passing submatrices to a function:

```
#define SUBMATRIX(C,I,J,b) ((double (*)[NMAX]) & (C [I*b][J*b]))
```

Suppose `A` is a 2D array of size `NMAX`-by-`NMAX`, defined as:

```
double A [NMAX][NMAX] ;
```

A submatrix can be passed to a function `func` using, for exampe:

```
func (SUBMATRIX (A, 2, 5, 8), 8) ;
```

where `func` is defined as

```
void func (double X [ ][NMAX], int n)
```

This submatrix `X` is the (2,5) block of the matrix `A`, of size 8-by-8. It consists of the region of `A` in rows 16 to 23 (starting at row 2*8) and columns 40 to 47). Inside the function `func`, this matrix `X` can be accessed as if it were an `n`-by-`n` matrix with rows and columns range from 0 to 7 (or 0 to `n-1`). That is, the function `func` does not need to be aware that `X` is a submatrix of a larger matrix. All it needs to know is that each row of `X` is held in a memory space of size `NMAX`. The C compiler will handle all the index calculations for you. For example, the entry `X[0][0]` is the same as `A[16][40]`, `X[1][2]` is the same as `A[17][42]`, and so on.

Files in `proj1.zip`:

- `Makefile`: for both Linux and the Mac OSX. If you test it on a Mac, you will need to install XCode, including the command line utilities. Testing on a Mac is handy, if you have one, but for best timing results, however, please report your final results on ADA.
- `mult.c`: the primary program. See the `TODO` comments. I have written the test driver for you. All you have to do is write the three matrix multiplication methods.
- `tictoc.c`: timing routines.
- `tictoc.h`: include file for using `tictoc`.
- `mymult.lsf`: job submission file to run the tests on ADA. Use the command `bsub < mymult.lsf`. You may need to adjust the run time limit. To check the status of your jobs use `bjobs`.

Write up a short project report discussing your solution and the performance obtained by each method. Which method is fastest overall? What is the peak speedup over the sequential method (note this is not printed by the `mult.c` program)?

Honesty statement: As Aggies, we follow the Aggie Honor Code. Period. This is a solo assignment. Do not share code; I can find it no matter how you obscure it. Trust me on that; I have been writing code, now used by millions, since before you were born. Copying code results in a zero on the project and a record in the honor system; don't risk it.