Assignment 1: 3-D Monte Carlo Integration

Due: Monday 1/28/11 (before class); Mult Fac = 1.0

In this assignment, you will use OpenMP to parallelize 3-D Monte Carlo integration. Your program should take the following optional configuration flags from the command line:

```
   -N <# of iterations> -S <seed base> -p <# threads>
```

Any other runtime flags will cause a usage message to printed, and the program will exit with an error. If the number of threads is not supplied or is given as 0, you should instead use the number of available processors as the number of threads. The defaults values should be $N = 16,000,000, S = 0xFCB321$. For those of you who are unfamiliar with getting command-line arguments in C, here is a code fragment that would do so if our only flag was N:

```
int main(int nargs, char **args)
....

   for (i=1; i < nargs; i++)
   {
      if (args[i][0] != '-')
         PrintUsage(args[0], i);
      switch(args[i][1])
      {
      case 'N':
         if (++i == nargs)
            PrintUsage(args[0], i);
         *N = atoi(args[i]);
         break;
      default:
         PrintUsage(args[0], i);
      }
   }
```

I have produced several files for your use, which may be found in:

```
   ~whaley/classes/spring11/cs4823/mcint/
```

You will want to view all the files, but only the `Makefile` (which is used to build your executable) must be copied to your own directory, where you will also supply the required `mcint.c`.

`GetCycleCount.S` is assembly language that returns a 64-bit integer indicating the processor cycle, and `mytime.[c,h]` provide a C API timer that calls `GetCycleCount` and returns the elapsed time in seconds. Notice that you must change the CPP macro defined in the `Makefile`'s `DEFS` macro in order to assemble `GetCycleCount` for the x86 or SPARC. `mytime.c` ensures there is no rounding when going from a 64-bit integer to a 64-bit float by keeping a start time around. Scope this code and understand it. In order to find the Mhz of your processor, `cat /proc/cpuinfo` will help under Linux, `/usr/bin/psrinfo -v` will help under Solaris.

`func.h` prototypes two functions that will allow you to do your integration. `int IsInFunc(double x, double y, double z)` returns 0 if the coordinates are outside the function, and 1 if they lie on or within the function. `void GetBoundaries` returns the coordinates of the corners of a rectangular prism that is guaranteed to contain the function being integrated. Finally, this file defines the CPP macro `KNOWN_VOL` which is defined to the actual volume of the given function. Note that the present functions provide a sphere, but you should not assume this (no need to in Monte Carlo integration); I am free to switch to any shape for grading.

Note that getting something to run is not sufficient: we want to be sure we are speeding up the application! Therefore, you should put timing calls around the main loop (including thread startup/shutdown overheads). Since this problem is embarassingly parallel, we are looking for speedup of roughly p when using p processors (more threads than physical processors should not result in additional speedups), assuming we have enough iterations to do.

Here is the relavant section of the serial Monte Carlo integration (be sure to use print statements like these, so that your output matches mine exactly):

```
   srand48(seed);
   for (i=0; i < N; i++)
   {
         x = drand48()*xdist + xmin;
         y = drand48()*ydist + ymin;
         z = drand48()*zdist + zmin;
         if (IsInFunc(x, y, z))
            hits++;
   }
/*
 * Need to add timing calls to compute ms
 */
   fprintf(stdout, "   rectVol = %f * %f * %f = %f\n",
           xdist, ydist, zdist, rectVol);
   fprintf(stdout, "   N=%u, nhits=%u, %% hits = %f\n", N, hits,
           (hits*100.0)/(1.0*N));
   fprintf(stdout, "   Time for %d iterations: %f (ms)\n", N, ms);
   fprintf(stdout, "funcVol = %lf, givenVol=%lf error=%e\n",
           compVol, KNOWN_VOL, fabs(compVol-KNOWN_VOL));
```

One of the challenges of threading is finding efficient thread-safe routines. `drand48()` is not thread safe and/or parallelizable. Therefore, for your parallel code, you will need to use a thread-safe random number generator. If you examine `func.h`, you will see I have provided macros for using the standard `rand` or `rand_r` functions. **Hint**: think carefully about the seed: should it be shared, private, how should it be initialized, etc?

Our code should work correctly in a serial environment. If the compiler does not support OpenMP, it will not define the standard CPP macro _OPENMP. After you get things working, use `#ifdef _OPENMP` to ensure your program can run serially. Note that there is no need to surround the `#pragma`'s with this guard, as ANSI/ISO C defines that compilers will ignore unknown `pragmas`. To make sure this works, you can use the target `make serial`.

This whole assignment should be written in a single file, called `mcint.c`. At the top of the program, put a comment that identifies you, the assignment, and the class. You should e-mail the source code for `mcint.c` to `whaley@cs.utsa.edu` before class on the due date.

Your program should work under both Solaris and Linux. You should do your original development on the Solaris machine, which is a normal UTSA machine, and so uses your usual name and password. The solaris machine is an 8-processor UltraSPARC II, and is called `intrepid.cs.utsa.edu`. For Linux, I have given you access to my experimental 8-processor 3rd generation Opteron machine, `etl-k10h8.cs.utsa.edu`. This machine goes up and down randomly, so `scp` your files often! Please login immediately and reset your `etl-k10h8` password.

Here are some things to examine once you are up and running:

1. Put in a debugging print of each thread's number and number of local iterations in your parallel region to make sure these values add up to the requested numbers (don't leave this debug statement in for handin).
2. Remember to check that $p = 0$ results in $p = nprocs$.
3. Make sure that you get roughly p speedup, $\forall\ p \leq nprocs$ for large N.
4. Ensure that N iterations run on p threads gives an answer that is roughly as accurate as N iterations run serially. Be certain it is not merely as accurate as N/p iterations run serially!
5. Check that as N rises, total error generally is reduced.
6. Remember to check that if `_OPENMP` is not defined, your code compiles and runs normally as a serial program. You would not want to call `rand_r` in a serial program. One option is to make sure you can compile and run with gcc without the `OpenMP` enabling flag, as when you do `make serial`.

Collaboration: As I have given you access to the answer for testing, and have discussed the general implementation strategy, students should not collaborate in any way on this assignment. If you need debugging help, ask a fellow student not enrolled in this class or schedule an appointment with me. Do not consult with anyone on detailed implementation issues.