Multicore machines and shared memory

Multicore CPUs have more than one 'core' processor that can execute instructions at the same time. The cores share main memory. In the next few activities, we will learn how to use a library called OpenMP to enable us to write programs that can use multicore processors and shared memory to write programs that can complete a task faster by taking advantage of using many cores. These programs are said to work "in parallel". We will start with our own single machines, and then eventually use a machine with a large number of cores provided by Intel Corporation, called the Manycore Testing Lab (MTL).

Parallel programs use multiple 'threads' executing instructions simultaneously to accomplish a task in a shorter amount of time than a single-threaded version. A *process* is an execution of a program. A *thread* is an independent execution of (some of) a process's code that shares (some) memory and/or other resources with that process. When designing a parallel program, you need to determine what portions could benefit from having many threads executing instructions at once. In this lab, we will see how we can use "task parallelism" to execute the same task on different parts of a desired computation in threads and gather the results when each task is finished.

Getting started with OpenMP

We will use a standard system for parallel programming called OpenMP, which enables a C or C++ programmer to take advantage of multi-core parallelism primarily through preprocessor *pragmas*. These are directives that enable the compiler to add and change code (in this case to add code for executing sections of it in parallel).

More resources about OpenMP can be found here: http://openmp.org/wp/resources/.

Extra: Comments in this format indicate possible avenues of exploration for people seeking more challenge in this lab.

We will begin with a short C++ program, parallelize it using OpenMP, and improve the parallelized version. This initial development work can be carried out on a linux machine. Working this time with C++ will not be too difficult, as we will not be using the object-oriented features of the language, but will be taking advantage of easier printing of output.

The following program computes a Calculus value, the "trapezoidal approximation of

$$\int_0^\pi \sin(x)\, dx$$

using 2^{20} equal subdivisions." The exact answer from this computation should be 2.0.

```
#include <iostream>
```

```
#include <cmath>
#include <cstdlib>
using namespace std;

/* Demo program for OpenMP: computes trapezoidal approximation to an
integral*/

const double pi = 3.141592653589793238462643383079;

int main(int argc, char** argv) {
  /* Variables */
  double a = 0.0, b = pi;  /* limits of integration */;
  int n = 1048576; /* number of subdivisions = 2^20 */
  double h = (b - a) / n; /* width of subdivision */
  double integral; /* accumulates answer */
  int threadct = 1;  /* number of threads to use */

  /* parse command-line arg for number of threads */
  if (argc > 1)
    threadct = atoi(argv[1]);

  double f(double x);

#ifdef _OPENMP
  cout << "OMP defined, threadct = " << threadct << endl;
#else
  cout << "OMP not defined" << endl;
#endif

  integral = (f(a) + f(b))/2.0;
  int i;

  for(i = 1; i < n; i++) {
    integral += f(a+i*h);
  }

  integral = integral * h;
  cout << "With n = " << n << " trapezoids, our estimate of the
integral" <<
    " from " << a << " to " << b << " is " << integral << endl;
}

double f(double x) {
  return sin(x);
}
```

Comments on this code:

- If a command line argument is given, the code segment below converts that argument to an integer and assigns that value to the variable threadct, overriding the default value of 1. This uses the two arguments of the function main(), namely argc and argv. This demo program makes no attempt to check whether a first command line argument argv[1] is actually an integer, so make sure it is (or omit it).

```
    if (argc > 1)
      threadct = atoi(argv[1]);
```

- The variable threadct will be used later to control the number of threads to be used. Recall that a *process* is an execution of a program. A *thread* is an independent execution of (some of) a process's code that shares (some) memory and/or other resources with that process. We will modify this program to use multiple threads, which can be executed in parallel on a multi-core computer.
- The preprocessor macro _OPENMP is defined for C++ compilations that include support for OpenMP. Thus, the code segment below provides a way to check whether OpenMP is in use.

```
  #ifdef _OPENMP
    cout << "_OPENMP defined, threadct = " << threadct << endl;
  #else
    cout << "_OPENMP not defined" << endl;
  #endif
```

- This code also shows the convenient way to print to stdout in C++.
- The following lines contain the actual computation of the trapezoidal approximation:

```
    integral = (f(a) + f(b))/2.0;
    int i;

    for(i = 1; i < n; i++) {
      integral += f(a+i*h);
    }

    integral = integral * h;
```

Since n == 2^{20}, the for loop adds over 1 million values. Later in this lab, we will parallelize that loop, using multiple cores which will each perform part of this summation, and look for speedup in the program's performance.

On a linux machine, create a file containing the program above, perhaps named trap-omp.C. To compile your file, you can enter the command

```
% g++ -o trap-omp trap-omp.C -lm -fopenmp
```

(Here, % represents your shell prompt, which is usually a machine name followed by either % or $.) First, try running the program without a command-line argument

```
 %  ./trap-omp
```

This should print a line "_OPENMP defined, threadct = 1", followed by a line indicating the computation with an answer of 2.0. Next, try

```
 %  ./trap-omp 2
```

This should indicate a different thread count, but otherwise produce the same output. Finally, try recompiling your program *omitting the* -fopenmp *flag*. This should report _OPENMP not defined, but give the same answer 2.0.

Note that *the program above is actually uses only a single core*, whether or not a command-line argument is given. It is an ordinary C++ program in every respect, and OpenMP does not magically change ordinary C++ programs; in particular, the variable threadct is just an ordinary local variable with no special computational meaning.

To request a parallel computation, add the following *pragma* preprocessor directive, just before the for loop.

```
   #pragma omp parallel for num_threads(threadct) \
     shared (a, n, h, integral) private(i)
```

The resulting code will have this format:

```
      int i;

   #pragma omp parallel for num_threads(threadct) \
     shared (a, n, h, integral) private(i)
      for(i = 1; i < n; i++) {
        integral += f(a+i*h);
      }
```

Here are some comments on this pragma:

- Make sure no characters follow the backslash character before the end of the first line. This causes the two lines to be treated as a single pragma (useful to avoid long lines).
- The strings omp parallel for indicate that this is an OpenMP pragma for parallelizing the for loop that follows immediately. The OpenMP system will divide the 2^{20} iterations of that loop up into threadct segments, each of which can be executed in parallel on multiple cores.
- The OpenMP *clause* num_threads(threadct) specifies the number of threads to use in the parallelization.

Extra: OpenMP also provides other ways to set the number of threads to use, namely the omp_set_num_threads() library function and the OMP_NUM_THREADS environment variable.

- The clauses in the second line indicate whether the variables that appear in the for loop should be shared with the other threads, or should be local private variables used only by a single thread. Here, four of those variables are globally shared by all the threads, and only the loop control variable i is local to each particular thread.

Extra: OpenMP provides several other clauses for managing variable locality, initialization, etc. Examples: default, firstprivate, lastprivate, copyprivate .

Enter this change, compile, and test the resulting executable.

After inserting the parallel for pragma, observe that the program runs and produces the correct result 2.0 for threadct == 1 (e.g., no command-line argument), but that

```
    % ./trap-omp 2
```

which sets threadct == 2, produces an incorrect answer (perhaps about 1.5). What happens with repeated runs with that and other (positive) thread counts? Can you explain why?

Note: Try reasoning out why the computed answer is correct for one thread but incorrect for two or more threads. *Hint:* All of the values being added in this particular loop are *positive* values, and the erroneous answer is too low.

If you figure out the cause, think about how to fix the problem. You may use the OpenMP website or other resources to research your solution, if desired.