CSCI 2500: Computer Organization

Lab 11 – Exercise   Introduction to Parallel Programming

Note: Comments in red like this one point to possible areas of exploration for students seeking to research more aspects of parallel programming (not required for completion of the lab).

In this lab you will create a program that uses multi-core parallelism and explore the issues in parallelism and concurrency that can arise.

Prerequisite: Make sure your Cygwin installation has the OpenMP libraries installed. This can be downloaded using the Cygwin package manager. Those students who are using an Ubuntu Linux (or other flavor) should already have the OpenMP libraries installed for gcc. Students who are using Mac OS X can get an OpenMP compatible gcc compiler from Homebrew (http://brew.sh).

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on most processor architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. You will start with a short C program, parallelize it using OpenMP, and improve the parallelized version.

The following program computes the a Calculus value, the "trapezoidal approximation of using 2^{22} equal subdivisions." The exact answer from this computation should be 2.0.

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

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* Introduction to OpenMP: Computes trapezoidal */
/*     approximation to an integral             */

const double pi = 3.141592653589793238462643383079;

double f(double); /* function definition */

int main(int argc, char** argv)
{
   double a = 0.0, b = pi;  /* limits of integration */;
   int n = 4194304; /* number of subdivisions = 2^22 */
   double h = (b - a) / n; /* width of subdivision */
   double integral; /* accumulates answer */
   long threadct = 1;  /* number of threads to use */

   int i; /* loop variable */

   /* parse command-line arg for number of threads */
   if (argc > 1)
      threadct = strtol(argv[1],NULL,10);

#ifdef _OPENMP
   printf("OMP defined, threadct = %lu\n", threadct);
#else
   printf("OMP not defined\n");
#endif

   integral = (f(a) + f(b))/2.0;  /* initialize variable integral */

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

   integral = integral * h;
   printf("With n = %d trapezoids, our estimate ", n);
   printf("of the integral from %g to %g is %g\n", a, b, integral);
}

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

Notes on the code above:

- 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`.
- 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. You 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.

- The following lines contain the actual computation of the trapezoidal approximation:

  ```
  integral = (f(a) + f(b))/2.0;
  for(i = 1; i < n; i++)
  {
     integral += f(a+i*h);
  }
  ```

  Since n = 2^{22}, the loop above will add over 4 million values. Later in this lab, you will parallelize that loop, using multiple cores which will each perform part of this summation, and look for a speedup in the program's performance.

Copy the code above into a file called `lab11.c`. To compile the file use the following command:

```
gcc -o lab11 lab11.c -lm -fopenmp
```

First, try running the program without a command-line argument:

```
./lab11
```

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

```
./lab11 2
```

This should indicate a different thread count, but otherwise produce exactly the same output, since nothing else has changed. (In particular, the computation still uses only one thread.) Finally, try recompiling your program omitting the `-fopenmp` flag. This should report `_OPENMP not defined`, but give the same answer of 2.0.

The program above actually uses only a single thread on 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. 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)
```

Some comments on the pragma above:

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

After you have added the pragma, compile, and then test the resulting executable. You will observe that the program runs and produces the correct result 2.0 for threadct == 1 (e.g., no command-line argument), but the command:

```
./lab11 2
```

produces an incorrect answer (approximately 1.24). What happens with repeated runs with that and other (positive) thread counts? Can you explain why?

A race condition exists within a program if the correct behavior of a program depends on the timing of its execution. With 2 or more threads, the program in this lab has a race condition concerning the shared variable integral, which is the accumulator for the summation performed by that program's for loop.

When threadct == 1, the single thread of execution updates the shared variable integral on every iteration, by reading the prior value of the memory location integral, computing and adding the value f(a+i*h), then storing the result into that memory location integral. (Recall that a variable is a named location in main memory.)

But when threadct > 1, there are at least two independent threads, executed on separate physical cores, that are reading then writing the memory location integral. The incorrect answer results when the reads and writes of that memory location get out of order. Here is one example of how unfortunate ordering can happen with two threads:

```
Thread 1                        | Thread 2
                                |
code: integral += f(a+i*h);     | code: integral += f(a+i*h);
                                |
exec: 1. read value of integral | exec:
                2. add f(a+i*h) | 1. read value of integral
                                | 2. add f(a+i*h)
       3. write sum to integral |
                                | 3. write sum to integral
```

In this example, during one poorly timed iteration for each thread, Thread 2 reads the value of the memory location integral before Thread 1 can write its sum back to integral. The consequence is that Thread 2 replaces (overwrites) Thread 1's value of integral, so the amount added by Thread 1 is omitted from the final value of the accumulator integral.

Can you think of other situations where unfortunate ordering of thread operations leads to an incorrect value of integral? Write down at least one other bad timing scenario.

Note: Thousands of occurrences of bad timing can lead to the computed answer for integral being off by often 25% or more. One approach to avoiding this program's race condition is to use a separate local variable integral for each thread instead of a global variable that is shared by all the threads.

But declaring integral to be private instead of shared in the pragma will only generate `threadct` partial sums in those local variables named integral – the partial sums in those temporary local variables will not be added to the program's variable integral. In fact, the value in those temporary local variables will be discarded when each thread finishes its work for the parallel for if we simply make integral private instead of shared.

Fortunately, OpenMP provides a convenient and effective solution to this problem. The OpenMP clause:

```
 reduction(+: integral)
```

will do the following:

- cause the variable integral to be private (local) during the execution of each thread
- add the results of all those private variables
- store that sum of private variables in the global variable named integral

Add this clause to your OpenMP pragma, and remove the variable integral from the shared clause, then recompile and test your program. You should now see the correct answer 2.0 when computing with multiple threads -- a correct multi-core program!

A code segment is said to be thread-safe if it remains correct when executed by multiple independent threads. The body of this loop is not thread-safe.

Some libraries are identified as thread-safe, meaning that each function in that library is thread-safe. Of course, calling a thread-safe function doesn't insure that the code with that function call is thread-safe. For example, the function `f()` in the program above, is thread-safe, but the body of that loop is not thread-safe.

We can obtain the running time for a program using the `time` Linux program. For example, the command:

```
time -p ./lab11
```

might display the following output:

```
OMP defined, threadct = 1
With n = 4194304 trapezoids, our estimate of the integral from 0 to
3.14159 is 2
real 0.04
user 0.04
sys 0.00
```

Note: The actual format of the time information depends on your computer system. The -p flag produces output in a format comparable to what you will see in many large commercial systems (e.g. IBM, HP, and SGI). The `real` time measures actual time elapsed during the running of your command `lab11`. user measures the amount of time executing user code, and sys measures the time executing in kernel code.

You should find that real time decreases somewhat when changing from 1 thread to 2 threads; user time increases somewhat. Can you think of reasons that might produce these results?

Also, real time and user time may increase when increasing from 2 to 3 or more threads. What might explain that?

For more information on parallel programming libraries please see the following sites:

OpenMP: http://openmp.org/wp

Message Passing Interface (MPI, networked multi-CPU clustering): https://www.mpich.org/ or http://www.open-mpi.org

Hadoop and MapReduce: http://hadoop.apache.org/