Programming with Multiple Cores

CSInParallel Project

July 12, 2012

CONTENTS

CHAPTER

ONE

GETTING STARTED WITH MULTICORE PROGRAMMING USING OPENMP

1.1 Notes about this document

This is designed to be a lab activity that you will perform on linux machines and/or on the Intel Manycore Testing Lab (MTL).

Comments Sections labeled like this are important explanations to pay attention to.

Dig Deeper:

Comments in this format indicate possible avenues of exploration for people seeking more challenge or depth of knowledge.

1.2 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.

1.3 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).

See Also:

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

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

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

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 = 3.14159265358979323846264338307 9;

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

- 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 << "OMP defined, threadct = " << threadct << endl;
#else
  cout << "OMP not defined" << endl;
#endif
```

- The above 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.

1.4 To Do:

On a linux machine, create a file named trap-omp.C containing the program above or grab it and save it from the following link:

`download trap-omp.C` (on most browsers, right-click, save link as)

To compile your file, you can enter the command:

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

Note: 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, like this:

```
%  ./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 using 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);
  }
```

Comments

- 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 phrase **omp parallel for** indicates 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.
- 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.

Enter the above code change (add the pragma preprocessor directive), then compile and test the resulting executable with one thread, then more than one thread, like this:

```
%  g++ -o trap-omp trap-omp.C -lm -fopenmp
%  ./trap-omp
%  ./trap-omp 2
%  ./trap-omp 3
etc.
```

Dig Deeper:

- The OpenMP tutorial contains more information about advanced uses of OpenMP. Note that OpenMP is a combination of libraries and compiler directives that have been defined for both Fortran and C/C++.
- OpenMP provides other ways to set the number of threads to use, namely the omp_set_num_threads() library function (see tutorial section on library routines), and the OMP_NUM_THREADS linux/unix environment variable (see tutorial section on environment variables).
- OpenMP provides several other clauses for managing variable locality, initialization, etc. Examples: default, firstprivate, lastprivate, copyprivate. You could investigate this further in the tutorial section pertaining to clauses.

1.5 What's happening?

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

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

which sets threadct == 2, sometimes 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.

CHAPTER

TWO

CREATING A CORRECT THREADED VERSION

2.1 Race Conditions

A program has a race condition if the correct behavior of that program depends on the timing of its execution. With 2 or more threads, the program trap-omp.C 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 for `integral`, computing and adding the value f(a+i*h), then storing the result into that memory location. (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 for `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**
source code:	integral += f(a+i*h);	integral += f(a+i*h);
execution of binary code:	1. read value of integral 2. add f(a+i*h) 3. write sum to integral	1. read value of integral 2. add f(a+i*h) 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.

To Do

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. Other scenarios will work (you may have seen some of these during testing of your code). Write down one of these.

2.2 Avoiding Race Conditions

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.

To Do

Can you re-explain this situation in your own words?

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

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

will

1. cause the variable integral to be private (local) during the execution of each thread, and
2. add the results of all those private variables, then finally
3. store that sum of private variables in the global variable named integral.

2.3 Reduction

This code example contains a very common *pattern* found in parallel programs: **reducing several values down to one**. This is so common that the designers of OpenMP chose to make it simple to declare that a variable was involved on a reduction. The code here represents one way that reduction takes place: during a step-wise calculation of smaller pieces of a larger problem. The other common type of reduction is to accumulate all of the values stored in an array.

See Also:

You can use other arithmetic operators besdides plus in the reduction clause– see the OpenMP tutorial section on reduction

According to the OpenMP Tutorial, here is how the reduction is being done inside the compiled OpenMP code (similar to how we described it above, but generalized to any reduction operator):

> "A private copy for each listed variable is created for each thread. At the end of the reduction, the reduction operation is applied to all private copies of the shared variable, and the final result is written to the global shared variable."

Thus, a variable such as `integral` that is decalred in a reduction clause is both private to each thread and ultimately a shared variable.

To Do

Add the above reduction clause to your OpenMP pragma and remove integral from the list of shared variables, so that the pragma in your code appears as shown below. Then recompile and test your program many times with different numbers of threads. You should now see the correct answer of 2.0 every time you execute it with multiple threads – a correct multi-core program!

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

Tricky Business

Note that with the incorrect version, you sometimes got lucky (perhaps even many times) as you tried running it over and over. This is one of the most perplexing and difficult aspects of parallel programming: **it can be difficult to find your bugs because many times your program will apear to be correct**. Remember this: you need to be diligent about thinking through your solutions and questioning whether you have coded it correctly.

2.4 Thread-safe Code

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; we were able to make it so by indicating that a *reduction* was taking place on the variable `integral`.

Some C/C++ 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 our example, is thread-safe, but the body of that loop is not thread-safe.

CHAPTER

THREE

TIMING AND PERFORMANCE ON INTEL MANYCORE TESTING LAB

3.1 Timing performance

We would like to know how long it takes to run various versions of our programs so that we can determine if adding additional threads to our computation is worth it.

There are several different ways that we can obtain the time it takes a program to run (we typically like to get this time in milliseconds or less).

3.1.1 Simple, less accurate way: linux time program

We can obtain the running time for an entire program using the *time* Linux program. For example, the line

```
/usr/bin/time -p trap-omp
```

might display the following output:

```
OMP defined, threadct = 1
With n = 1048576 trapezoids, our estimate of the integral from 0 to 3.14159 is 2

real 0.04

user 0.04

sys 0.00
```

Here, we use the full path /usr/bin/time to insure that we are accessing the time program instead of a shell built-in command. The -p flag produces output in a format comparable to what we will see in the MTL.

The real time measures actual time elapsed during the running of your command trap-omp. user measures the amount of time executing user code, and sys measures the time executing in Linux kernel code.

To Do

Try the time command using your linux machine, and compare the results for different thread counts. 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 increase considerably on some machines when increasing from 2 to 3 or more threads. What might explain that?

3.1.2 Additional accuracy: Using OpenMP functions for timing code

The for loop in your trap-omp.C code represents the parallel portion of the code. The other parts are the 'sequential parts' where one thread is being used (these portions of code are quite small in this simple example). Using functions to get current time at points in your program, you can begin to examine how long the sequential port takes in relation to the parallel portion. You can also use these functions around all the code to determine how long it takes with varying numbers of processors.

We can use an OMP library function whose 'signature' looks like this:

```
#include <omp.h>

double omp_get_wtime( void );
```

We can use this in the code in the following way:

```
// Starting the time measurement

double start = omp_get_wtime();

// Computations to be measured

...

// Measuring the elapsed time

double end = omp_get_wtime();

// Time calculation (in seconds)

double time1 = end - start;

//print out the resulting elapsed time
cout << "Time for paralel computation section: "<< time1 << "  milliseconds." << endl;
...
```

To Do

Try inserting these pieces code and printing out the time it takes to execute portions of your trap-omp.C code. You will use this for later portions of this activity. You can do this on your local linux machine now to test it out and make sure it is working.

3.1.3 Basic C/C++ timing functions

You could use basic linux/C/C++ timing functions: See the end of this activity for an explanation of the way in which we can time any C/C++ code, even if you are not using OpenMP library functions or pragmas. This will come in handy in cases where you are not using OpenMP (such as CUDA, for example).

3.2 Using the MTL

Let's try using many more threads and really experiment with multicore programming! You will need to use a 'terminal' on Macs or 'Putty' on PCs. If you are off campus, you will need to ssh into a machine on your campus before then logging into the MTL machine at Intel's headquarters in Oregon.

Note: Macalester's local machine that you can use is selkie.

You can login to the MTL computer, as follows

```
ssh accountname@192.55.51.81
```

Use one of the special MTL student account usernames provided to you, together with the password distributed to the class.

Next, copy your program from your laptop or local linux machine to the MTL machine. One way to do this is to use another window (to keep for copying your code), then enter the following command *from the directory where your code is located*:

```
scp trap-omp.C accountname@192.55.51.81:
```

After making this copy, login into the MTL machine 192.55.51.81 in another window.

On the MTL machine, compile and test run your program.

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

./trap-omp

./trap-omp 2

./trap-omp 16
```

Note: Since the current directory . may not be in your default path, you probably need to use the path name ./trap-omp to invoke your program.

Now, try some time trials of your code on the MTL machine. (The full pathname for time and the -p flag are unnecessary.) For example, using time:

```
time trap-omp

time trap-omp 2

time trap-omp 3

time trap-omp 4

time trap-omp 8

time trap-omp 16

time trap-omp 32
```

What patterns do you notice with the real and user times of various runs of trap-omp with various values of threadct?

Also try it without using the time command on the command line and instead using the OpenMP omp_get_wtime() function calls in your code.

To Do

It may be useful to change our problems size, n, to see how this affects the time and to observe the range of times that can occur for various problem sizes. We therefore should eliminate 'hard coding' of n.
Now update your code so that you submit the number of elements to compute as an additional command-line argument. Now the number of trapezoids, n should be set to the value in argv[2] (at the time that you set threadcnt to the value in argv[1]). **This also involves moving the declaration and assignment of the variable h, also**. The updated segment of code should look like this:

```
/* parse command-line arg for number of threads, n */
if (argc == 2) {
  threadct = atoi(argv[1]);
} else if (argc == 3) {
  threadct = atoi(argv[1]);
  n = atoi(argv[2]);
}
double h = (b - a) / n; /* width of subdivision */
```

Note: The best way to work is to change your code on your local campus machine and copy it to the MTL using scp. That way you have your own copy.

3.2.1 Submitting Batch Jobs for Timing Accurately

To submit a job on MTL and guarantee that you have exclusive access to a nod for timing purposes, you submit your job to a queuing system. You do this by creating a 'script' file that will be read by the queuing system. Then you use the qsub command to run that script. Here is an example of the contents of a script file (save it as submit.sh on your MTL account):

```
    #!/bin/sh

    #PBS -N LS\_trap

    #PBS -j oe

    #here is how we can send parameters from job submission on the command
    line:

    $HOME/240activities/trap-omp $p $n

# this is a shell script comment
    # the job gets submitted like this:
    #   qsub -l select=1:ncpus=32 -v 'p=32, n=10485760'

    /home/mcls/240activities/submit.sh

    ######### end of script file
```

Here is an example of how you run the script (change the path for your user account):

```
qsub -l select=1:ncpus=32 -v 'p=32, n=10485760'
/home/mcls/240activities/submit.sh
```

3.3 Investigating 'scalability'

Scalability is the ability of a parallel program to run ingreasingly larger problems. In our simple program, the *problem size* is the number of trapazoids whose area are computed. You will now conduct some investigations of two types of scalability of parallel programs:

- **stong scalability**
- **weak scalbility**

3.3.1 Strong Scalability

As you keep the same 'problem size', i.e. the amount of work being done, and increase the number of processors, you would hope that the time drops proportionally to the number of processors used. So in your case of the problem size being the number of trapezoids computed, 2^{20}, are you able to halve the time as you double the number of threads? When does this stop being the case, if at all? When this occurs, your program is exhibiting *strong scalability*, in that additional resources (threads in this case) help you **obtain an answer faster**. To truly determine whether you have *strong scalability*, you will likely need to try a larger problem size on the MTL.

3.3.2 Weak Scalability

Another interesting set of experiments to try is to both increase the problem size by changing the number of trapezoids to values higher than 2^{20} and to correspondingly increase the number of threads. Try this: if you double the problem size and double the number of threads, does the loop take the same amount of time? In high performance computation, this is known as *weak scalability*: you can keep using more processors (to a point) to **tackle larger problems**.

Note: Don't try more than the maximum 40 cores on the MTL for the above tests.

3.3.3 What happens when you have more threads than cores?

Another interesting investigation is to consider what happens when you 'oversubscribe' the cores by using more threads than cores available. Try this experiment and write down your results and try to explain them.

3.4 An Alternative Method for Timing Code (optional; for reference)

The following code snippets can be used in your program to time sections of your program. This is the traditional linux/C/C++ method, which is most likely what the implementation of the OMP function get_wtime() is using.

```
/* Put this line at the top of the file: */
#include <sys/time.h>

/* Put this right before the code you want to time: */
struct timeval timer_start, timer\_end;
gettimeofday(&timer_start, NULL);

/* Put this right after the code you want to time: */
gettimeofday(&timer_end, NULL);
double time_spent = timer_end.tv_sec - timer_start.tv_sec +
                    (timer_end.tv_usec - timer_start.tv_usec) / 1000000.0;
printf("Time spent: %.6f\\n", time_spent);
```

Note: This example uses C printf statements; feel free to use C++ cout syntax, perhaps like this:

```
cout << "Time for paralel computation section: "<< time_spent << "  milliseconds." << endl;
```