Using OpenMP: Timing and Performance on Intel Manycore testing lab

Timing performance

We can obtain the running time for a 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.

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?

Using the MTL

You will need to use a ‘terminal’ on Macs or ‘Putty’ on PCs. You can now login to the MTL computer, as follows

```
ssh accountname@192.55.51.81
```

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

Next, copy your program from your laptop to the MTL machine. One way to do this is to logout, then enter the following command:

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

After making this copy, login into the MTL machine `192.55.51.81` again.

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:

```
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?

An Alternative Method for Timing Code

The following code snippets can be used in your program to time sections of your program:

```
/* 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 timer_spent = timer_end.tv_sec - timer_start.tv_sec +
                     (timer_end.tv_usec - timer_start.tv_usec) / 1000000.0;
printf("Time spent: %.6f\n", timer_spent);
```

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 processor, or thread is being used. Using the above code snippets as a guide, you can begin to examine how long the sequential port takes in relation to the parallel portion.

Investigating '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 (htreads in this case) help you obtain an answer faster.

An interesting set of experiments to try is to increase the problem size by changing the number of trapezoids to values higher than 2^{20}. 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 know as weak scalability: you can keep using more processors (to a point) to tackle larger problems.