CS3211 - Lab 1
openMP, and memory behaviour

January 19, 2017

In this tutorial you will learn how to compile, run and profile openMP applications. We will use the lab PCs (specifically the ones soctf-pdc-000 up to soctf-pdc-014), the teaching cluster tembusu, and an MP implementation called OpenMP.

To log in, you should use the NUSSTU\a00XXXXX account to the Ubuntu GNU+Linux OS on soctf-pdc-XXX, along with your NUSNET password. In this laboratory, Windows will not work for you. The objective of this lab is to apply basic performance analysis techniques on programs. First we will compile and run a shared-memory matrix multiplication program. Next, we will learn how to measure the performance of this program by using program instrumentation. Lastly, using the measured information, we analyze the results.

1 Compiling and Running shared-memory openMP Programs

The sequential matrix multiplication program below is converted into a shared-memory OpenMP program and will be used in this lab.

```
void mm(matrix a, matrix b, matrix result)
{
   int i, j, k;
   for (i = 0; i < size; i++)
      for (j = 0; j < size; j++)
         for (k = 0; k < size; k++)
            result[i][j] += a[i][k] * b[k][j];
}
```

This parallel program is almost identical to the sequential program. The only difference is in the multiplication function which is parallelized by splitting the iterations of the outermost loop using multiple threads. You will find the sample programs at http://www.comp.nus.edu.sg/~hugh/cs3211/

```
void mm(matrix a, matrix b, matrix result)
{
   int i, j, k;
   // Parallelize the multiplication
   // Each thread will work on one iteration of
   // the outer-most loop
   // Variables (a, b, result) are shared between threads
   // Variables (i, j, k) are private per-thread
   #pragma omp parallel for shared(a, b, result) private (i, j, k)
   for (i = 0; i < size; i++)
      for (j = 0; j < size; j++)
         for (k = 0; k < size; k++)
            result[i][j] += a[i][k] * b[k][j];
}
```

The line beginning with `#pragma` directs the compiler to generate the code that parallelizes the `for` loop. Each iteration of the `for` loop is independent of the others, and thus they can execute in parallel. The compiler will split the `for` loop iterations into different chunks (each chunk contains one or more iterations) which will be executed on different OpenMP threads in parallel.

To compile the program, you need to pass the flag `-fopenmp` to the compile instruction as below:

```
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab1/mm-shmem.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ gcc -fopenmp mm-shmem.c -o mm-shmem
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$
```

The `-fopenmp` flag enables the compiler to detect the `#pragma` commands, which would otherwise be illegal under C/C++ syntax.

To execute an OpenMP program, you run it as a normal program:

```
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ ./mm-shmem
   ...
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$
```

Task 1: Compile and run the matrix multiplication program. Observe the number of threads that the program is using. What[1] is the number? Why is it this number? You can change the number of threads using the function `omp_set_num_threads(int)` in your OpenMP code, or the environment variable `OMP_NUM_THREADS`.

```
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ export OMP_NUM_THREADS=2
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ ./mm-shmem
   ...
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$
```

Task 2: Tabulate the (unloaded) runtime for the program working on arrays of size 128 up to 2048, and for threads from 1 to 8. The size of the problem should go across the table, and one row for each number of threads (1..8). Note that the times will vary dramatically, dependant on other events in the particular computer you are using. In your table, should you use the minimum, average or maximum value for the times?

Task 3: Record and interpret your table. What do the rows and columns tell you? Can you relate any discontinuities in the results tabulated to things you might know or discover about the processor?

If you have not already done so, enable access to Tembusu by going to

`https://mysoc.nus.edu.sg/~myacct/services.cgi`

using your SoC UNIX id and enable the SoC Computer Cluster service for your account. To login to Tembusu using a secure shell (`ssh`), you use your SoC UNIX id to login to an access node - perhaps ssh a00123456@xcnc1. If you are using the lab machine soctf-pdc-001, then log in to xcnc1, if using the lab machine soctf-pdc-002, then log in to xcnc2 (and so on).

Task 4: Repeat your experiment from Tasks 1..3, only this time use threads from 1 to 40. Compile and run the program as before, tabulating the results. In your writeup, you should be able to explain why we are interested in up to 8 threads on the lab machines and 40 threads on the tembusu machines, and also explain the tabulated results in terms of speedup.

```
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ ssh a0123456@xcnc1
   ... enter your SoC UNIX password ...
-bash-4.2$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab1/mm-shmem.c
-bash-4.2$ gcc -fopenmp mm-shmem.c -o mm-shmem
-bash-4.2$ export OMP_NUM_THREADS=1
-bash-4.2$ ./mm-shmem
   ...
-bash-4.2$
```

[1] You will have to produce a short writeup for this lab, as part of your first project, and the answers to the questions embedded in these tasks will form the basis for your writeup.

2 Memory effects

The next exercise does not involve an openMP program (but it could...), but is supposed to show an interesting effect common to most modern computers. You should do this experiment back on the laboratory machines.

Task 5: Download, compile and run the program testmem.c:

```
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab1/testmem.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ gcc testmem.c -o testmem
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ ./testmem
   ...
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$
```

This program does a series of operations (64M!) on different sized arrays, and times each series. In each case the exact same number of operations are done. Confirm this by investigating the source code.

Task 6: Tabulate and explain the results you got in task 5. You could even plot them.

3 Processor Hardware Event Counters

Hardware event counters are special registers built into modern processors that can be used to count low-level events in a system such as the number of instructions executed by a program, number of L1 cache misses among others. A modern processor such as Core i5 or Core i7 supports a few hundred types of events.

In this section, we will learn how to use hardware events counters for measuring the performance of a program. The lab machines support two methods for accessing the hardware events counters under Linux OS: `perf` and `PAPI-C`. In this lab, you will just look at `perf`.

3.1 Perf

The program `perf` is a software tool shipped with the Linux OS. It enables profiling of the entire execution of a program and produces a summary profile as output.

Task 7: Use `perf stat` to produce a summary of program performance:

```
$ perf stat -- ./mm-shmem
   ...
Performance counter stats for './mm-shmem':

    45.595495 task-clock                #   0.657 CPUs utilized
          410 context-switches          #   0.009 M/sec
            0 CPU-migrations            #   0.000 M/sec
          362 page-faults               #   0.008 M/sec
   98,015,090 cycles                    #   2.150 GHz
   35,472,062 stalled-cycles-frontend   #  36.19% frontend cycles idle
   10,198,393 stalled-cycles-backend    #  10.40% backend cycles idle
  169,428,906 instructions              #   1.73  insns per cycle
                                        #   0.21  stalled cycles per insn
   21,957,507 branches                  # 481.572 M/sec
      167,305 branch-misses             #   0.76% of all branches
0.069367093 seconds time elapsed
```

To count the events of interest, you can specify exactly which events you wish to measure:

```
$ perf stat -e cache-references -e cache-misses -e cycles -e instructions -- ./mm-shmem
   ...
Performance counter stats for './mm-shmem':
       8,764,236 cache-references
          58,695 cache-misses              # 0.670 % of all cache refs
   5,766,321,978 cycles                    # 0.000 GHz
  10,542,104,914 instructions              # 1.83 insns per cycle
2.151194898 seconds time elapsed
```

To list all the events available under your platform, you can use the list command:

```
$ perf list
List of pre-defined events (to be used in -e):
   cpu-cycles OR cycles                             [Hardware event]
   stalled-cycles-frontend OR idle-cycles-frontend  [Hardware event]
   stalled-cycles-backend OR idle-cycles-backend    [Hardware event]
   instructions                                     [Hardware event]
   cache-references                                 [Hardware event]
[... perf list output is truncated ...]
```

The program `testmem_size.c` does the same sort of calculation as that done in `testmem.c`, except that it does it for a single array size, expressed as a power of two. For example, to test the memory using an array size of $1024 = 2^{10}$ integers (with each integer using 4 bytes, the array size is 4kB), you would do this:

```
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab1/testmem_size.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ gcc testmem_size.c -o testmem_size
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ ./testmem_size 10
Array size: 4 kB and time taken is 0.147257 secs
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$
```

Task 8: Use perf to measure (L1 and L3) cache loads and misses for different sizes of array. Tabulate and interpret the results.

4 Writeup

In this lab, we looked at how you can develop shared memory programs using openMP, and how you can do detailed timing analysis.

Your first project will include a write-up, with the results for the tasks in this, and the following laboratories, which explore issues related to memory, processors, and accuracy. The document must contain clear answers to each question asked in each lab. For this lab, we will be looking primarily for your clear explanations for the behaviour you observed in tasks 3,4, 6 and 8. For each task - why did you get the results you got? To get ahead on your project, you should work on this sooner rather than later.