CS3210 – 2017/18 Semester I

Lab 3
Running Parallel Programs and Performance Instrumentation

Objectives:

1. Learn shared-memory parallel programming via OpenMP
2. Learn and use performance instrumentation
3. Learn to measure and quantify performance of parallel programs

Updated Lab Machine Setup

To facilitate your own working for future labs and assignments, we have created individual user account for you on all machines (node1/node2/jetson). The user id is the last 4 digits and the check alphabet of your matrix number, e.g. student with matric no **`A01234567X`** has the user id **`4567X`**. The password is a simple 5-digit numbers given to you in IVLE gradebook. Note that the accounts are independent on each node (i.e. your files saved on one node is not visible on other machines).

To ensure minimal overhead, we will stick to a text based interface for the lab machines. We have added a couple of packages to help:

Text Terminal

You can have multiple screens even with text based terminal. Type in the following command:

```
$ byobu-enable
```

You will now be able to create additional screens by pressing F2, and move among them via F4. Exit each screen normally by "$ **`exit`**". Google "byobu" for more information.

USB support

USB drives are now automatically mounted. After you plug in the device, a message prompt will appear on screen after small delay, the drive will be mounted at the directory "**`/media/usb`**" automatically.

IMPORTANT: Use the following command to unmount the USB drive before unplugging it.

```
$ pumount /media/usb
```

Otherwise, your directory/file may be corrupted!

Git support

Alternatively, you can use "**`git`**" to manage your files for labs / assignments. The git packages have been installed for all students account.

No SUDO access

To ensure the setup of each machine is as homogenous as possible, we have removed the "**`sudo`**" access from all student accounts. Please let us know if you think certain packages should be added for the class.

Introducing the problem scenario

In lecture, we (very) briefly mention the **matrix multiplication** problem. This lab is based on the different ways of parallelizing this problem.

Given a **n x m matrix A** and a **m x p matrix B**, the product of the two matrices **(AB)** is a **n x p matrix** with entries define by: $(\boldsymbol{AB})_{\boldsymbol{ij}} = \sum_{\boldsymbol{k=1}}^{\boldsymbol{m}} \boldsymbol{A_{ik}B_{kj}}$

A straightforward sequential translation is given below:

Sequential: **mm-seq.c**

```
void mm( matrix a, matrix b, matrix result )
{
    int i, j, k;
    //assuming square matrices of (size x size)
    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];
}
```

To compile the sequential program using `gcc`, enter the command:

```
$ gcc mm-seq.c -o mm0
```

To multiply two square matrices of size 10, run:

```
$ ./mm0 10
```

You should try matrix sizes in the range of 1000+ for a longer execution time.

Note: All files in this lab can be found on IVLE workbin, or directly downloaded via wget:

```
$ wget www.comp.nus.edu.sg/~ccris/cs3210/lab3/<filename>

e.g.
$ wget www.comp.nus.edu.sg/~ccris/cs3210/lab3/mm-seq.c
```

Shared-Memory OpenMP Programs

One quick way to parallelize this problem is to create task to handle each row in the result matrix. As each row can be calculated independently, there is no need to worry about synchronization. Also, if the content of the matrices are shared between the tasks, there is no addition communication needed! We make use of this idea and translate it into an OpenMP program as given below:

OpenMP: **mm-omp.c**

```
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];
}
```

OpenMP is a set of compiler directives and library routine directly supported by GCC (GNU C-Compiler) to specify high level parallelism in C/C++ or Fortran programs. In this example, we use OpenMP to create multiple threads, each working on one iteration of the outer-most loop.

The line beginning with *#pragma* directs the compiler to generate the code that parallelizes the *for* loop. 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 `gcc` compiler command:

```
$ gcc –fopenmp mm-omp.c -o mm1
```

The `–fopenmp` flag enables the compiler to detect the #pragma commands, which would otherwise be ignored by the compiler.

To execute the OpenMP program, you run it as a normal program. To multiply two square matrices of size 10, using 4 threads, run:

```
$ ./mm1 10 4
```

Exercise 1: Compile and run the matrix multiplication program. Modify the number of threads and observe the trend in execution time. You may want to use a relatively large matrix to really stress the processor cores.

Processor Hardware Event Counters

Due to the multiple layers of abstraction in modern high level programming, it is sometime hard to understand performance at the hardware level. For example, a single line of code ("**`result[i][j] += a[i][k] * b[k][j]`**;") could be translated into a handful of machine instructions. In addition, this statement can take a wide range of execution duration to finish depending on cache / memory behavior.

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 get hardware events counters for measuring the performance of a program using `perf`, a Linux OS utility. `perf` enables profiling of the entire execution of a program and produces a summary profile as output.

a. Use ***`perf stat`*** to produce a summary of program performance:

```
$ perf stat -- ./mm0
[… program output is not shown …]
Performance counter stats for './mm0':

      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
```

b. 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.0
[… program output is not shown …]
Performance counter stats for './mm.0 ':

      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
```

c. To list all the events available under your platform, you can use the 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 …]
```

Exercise 2: Use `perf` to profile the OpenMP version of matrix multiplication using different number of threads. Observe which events are profiled and comment on their variation among different runs.

Performance Analysis of the Matrix Multiplication Program

In this section, we will learn how to use hardware event counters to analyze the performance of a parallel program. For the exercises below, run the OpenMP version of matrix multiplication code on the both the tower PC (Intel Core i7) and the all-in-one PC (Intel Core i5) with n = 1, 2, 4, 8, 16, 32 threads. You should choose a reasonable matrix size (or have a few different test scenarios).

Exercise 3: Determine (i) the number of instruction executed per cycles (IPC) and (ii) MFLOPS Comment on how IPC and MFLOPS change with increasing number of threads.
[Hint: To get the MFLOPS, you need to estimate how many floating point operations are executed during program execution]

Exercise 4: Determine the execution time and derive the speedup of the program.

Exercise 5: Devise two methods to determine the sequential fraction and the maximum speedup achievable under Amdahl's Law.

[Hint: You can measure the sequential fraction, and you can also infer it using measured values of speedup and Amdahl's Law]

Homework:

You are required to produce a write-up with the results for exercises 3 to 5. Submit the lab report (in PDF form with file name format *A0123456X*.pdf) via IVLE Files by 20th September 2pm. The document must contain explanation on how you have solved the exercises, the results and the raw measured data.

References

1) perf reference: https://perf.wiki.kernel.org/index.php/Main_Page

2) perf manual: run `man perf`

3) OpenMP tutorial: https://computing.llnl.gov/tutorials/openMP/

4) Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors - http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

5) OpenMP Reference Sheet for C/C++ - http://www.plutospin.com/files/OpenMP_reference.pdf.

Appendix A - OpenMP Programming

A.1. Structure of an OpenMP Program

An OpenMP program consists of several parallel regions interleaved with sequential sections. In each parallel block, there is always one master thread and there may be several slave threads. The master thread always has thread id of zero.

Below is the OpenMP version of canonical hello world program. Each parallel region starts with a pragma command that tells the compiler that the following code block will be executed in parallel.

Sample program: **`hello-omp.c`**

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

int main (int  argc,  char *argv[])
{
      int thread_id, no_threads;

      /* Fork slave  threads,  each with its own  unique thread id */
      #pragma  omp  parallel
      {
            /* Obtain thread id */
            thread_id =  omp_get_thread_num();
            printf("Hello World from thread  =  %d\n",  thread_id);

            /* Only master thread  does this.
            Master thread always has id  equal to  0  */
            if (thread_id ==  0)
            {
                  no_threads = omp_get_num_threads();
                  printf("Number  of threads =  %d\n",  no_threads);
            }
      } /* All slave threads  join  master thread and are destroyed */
}
```

If you compile and run this program on the Intel Core i7 machine, you will see that there are 8 threads that echo the “Hello World” string. By default, OpenMP creates a number of threads equal to the number of processor cores of the machine. You can change this using the function `omp_set_num_threads(int)` in your OpenMP code, or the environment variable

`OMP_NUM_THREADS.`

A.2. Work-sharing Constructs

Inside the parallel region, there is some work that needs to be done. OpenMP provides four ways in which the work can be partitioned among the thread. These constructs are called work-sharing constructs:

- *Loop iterations*: Iterations within a *for* loop will be split among the existing threads. The programmer can control the *order* and the *number* of iterations assigned to each thread using the schedule directive. Example:

```
#pragma omp parallel
{
   #pragma omp for schedule (static, chunksize)
   for (i =  0; i <  n; i++)
   x[i] =  y[i];
}
```

  In this example, `n` iterations of the for loop is divided into pieces of size, chunksize, and assigned statically to the threads. There are other options for schedule, which you can read in the OpenMP quick-reference.

- *Sections*: The programmer manually defines some code blocks that will be assigned to any available thread, one at a time. Example:

```
#pragma omp parallel
{
   #pragma omp sections
   {
        #pragma omp  section
        {
             work1();
        }
        #pragma omp  section
        {
             work2();
        }
        #pragma omp  section
        {
             work3();
        }
   }
}
```

  In the *sections* region, you see the declaration of three work sections. The *sections* may be passed to different threads for execution.

- Single section: Only a single thread will execute the code. The runtime decides which thread will get to execute.
- Master section: Similar to single section, only that the master thread executes the code.