Lab 2
Parallel Programming with OpenMP and Performance Instrumentation

CS3210 – 2021/22 Semester 1

Learning Outcomes

1. Implementing shared memory parallel programs with OpenMP
2. Learning how to do non-intrusive performance instrumentation
3. Learning basic measurement techniques for quantifying performance of parallel programs

Logging in & Getting Started

Use the username & password provided to you in your LumiNUS gradebook. The hostname and the type of node are shown below.

- `soctf-pdc-001` - `soctf-pdc-008`: Xeon Silver 4114
- `soctf-pdc-009` - `soctf-pdc-016`: Intel Core i7-7700
- `soctf-pdc-018` - `soctf-pdc-019`: Dual-socket Xeon Silver 4114
- `soctf-pdc-020` - `soctf-pdc-021`: Intel Core i7-9700
- `xcne3` - `xcne4`: Dual-socket Xeon Gold 6230 (SoC account)

Part 1: Shared-Memory Multi-Threaded Programming with OpenMP

Introducing the Problem Scenario: Matrix Multiplication

In the lectures, we (very) briefly mentioned the matrix multiplication problem. This lab explores two different ways of parallelizing this problem.

Given an n **x** m **matrix** A and an m **x** p **matrix** B, the product of the two matrices **(**AB**)** is a n **x** p matrix with entries defined by:

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

A straightforward sequential implementation 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];
}
```

- Compile the code in a terminal (console):

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

- Run the program in a terminal:

  ```
  $ ./mm0 10
  ```

Exercise 1

Compile and run `mm-seq.c`. Run with matrix sizes exceeding 1000 to obtain a longer execution time.

Shared-Memory OpenMP Programs

One quick way to parallelize this problem is to create tasks to handle each row in the result matrix. Since each row can be computed independently, there is no need to worry about synchronization. Also, if the content of the matrices are shared between the tasks, there is no 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
    // Iterations of the outer-most loop are divided
    //   amongst the threads
    // 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 routines 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 a set of iterations of the outer-most loop. Note that this default behaviour is implementation-defined, as the `schedule` clause was not supplied here.

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.

- Compile the code in a terminal:

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

- The `-fopenmp` flag enables the compiler to detect the `#pragma` directives, which would otherwise be ignored.
- To execute an OpenMP program, run it as a normal program.
- To multiply two square matrices of size 10, using 4 threads, run the program in a terminal:

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

Exercise 2

Compile and run the OpenMP matrix multiplication program `mm-omp.c`. 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.

Part 2: Performance Instrumentation

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];` typically translates into a handful of machine instructions. In addition, this statement can take a wide range of execution time 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, number of branch misses, etc. A modern processor such as a Core i5 or Core i7 supports a few hundred types of events.

In this section, we will learn how to read hardware events counters to measure 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.

- Use **perf stat** to produce a summary of program performance

```
$ perf stat -- ./mm0
```

- The output of perf is shown below (program output 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
```

- To count events of interest, you can specify exactly which events you wish to measure with the -e flag followed by a comma-delimited list of event names:

```
$ perf stat -e cache-references,cache-misses,cycles,instructions -- ./mm0
```

- The output of perf is shown below (program output not shown):

```
Performance counter stats for './mm0':

         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
```

- When you run perf with the OpenMP program, you will notice that time elapsed differs from user time. Time elapsed is the response time (perception of the user) when executing a program. User time represents the total time spent by all cores for the user program (user mode). If you run on 4 cores, and get elapsed time of 8 s, the user time might be around 4×8 s.
- You can list all events available on your platform with the command:

```
$ perf list
```

- The output of perf list is shown below:

```
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]…
 [ perf list output is truncated …]
```

- When using `perf` you might notice inexplicable values for `cache-references`. This value depends on the architecture and what the architecture reports through the event counters. `cache-references` do not count cache-hits in L1 cache. You can get detailed information about the cache using hardware counters such as:

```
L1-dcache-load-misses
L1-dcache-load
LLC-load-misses
LLC-store-misses
LLC-loads
LLC-stores
```

When taking performance measurements, you might notice that the numbers vary between different runs. This is because the execution of the code might be impacted by the other processes (tasks) that run at the same time on the same machine. As such, you are advised to take at least 3 measurements with the same settings and show the best result (lowest execution time) in your analysis. The best result happens when the run is the least impacted by other tasks running at the same time in the system.

Exercise 3

Use **perf** to profile the OpenMP matrix multiplication program (`mm-omp.c`) with a varying number of threads. Observe the variation of different performance event counts for different runs.

Performance Analysis

In this section, we will use hardware event counters to analyze the performance of a parallel program. For the exercises below, run the OpenMP matrix multiplication program on **both** the Desktop PC (Intel Core i7) and the Server Tower (Intel Xeon) with $n = 1, 2, 4, 8, 16, 32, 64, 128, 256$ threads. You should choose a reasonable matrix size (execution should take a few seconds). You may have a few different test scenarios, but there is no need to show results for all.

Exercise 4

Determine (i) the number of instructions executed per cycle (IPC) and (ii) the number (in millions) of floating-point operations per second (MFLOPS). Comment on how the IPC and MFLOPS change with increasing number of threads (maximum one paragraph).
[Hint: To obtain the MFLOPS, you need to estimate how many floating-point operations are executed during program execution. Alternatively you may use perf to find the exact number.] Investigate what happens with the execution time when increasing the number of cores - does it get faster? How much faster? Is it proportional with the number of threads? We use floating point operations as a proxy because their number should not change when we increase the number of threads.

Exercise 5

In `mm-omp.c`, the elements of the first matrix A is stored in row-major order whilst those of the second matrix B is stored in column-major order (you can read more about row-major and column-major ordering here).

When we multiply elements of one row in A pairwise with elements of one column in B, we access the elements in A **row-wise** and the elements in B **column-wise**. Modify `mm-omp.c` to allow B's elements to be accessed row-wise when a cell in the output matrix is computed. Briefly explain your approach for implementation (maximum one paragraph). Note that there is no need to transpose B (since the values are randomly generated). You just need to access the elements in a different way.

Exercise 6

Compile and run `mm-omp.c` and `mm-omp-col.c` with a varying number of threads and record the execution time for both versions. Explain your observations from comparing the runs of the original implementation (where B is accessed column-wise) with that of your modified implementation (where B is accessed row-wise).

Lab sheet (2% of your final grade):
You are required to produce a write-up with the results for exercises 4 to 6. Submit the lab report (in PDF form with file name format A0123456X.pdf) via LumiNUS before Fri, 11 Sep. The document must contain

- explanation on how you have solved each exercise (maximum 3 paragraphs in total)
- your results and the raw measured data - use graph(s) and try to justify your observations

Write down the hostnames of the nodes you used for your experiments. When you run your experiments, check if any other student is using the same node, and change the node if you observe that someone else is already running their programs on that node.

For full credit, attempting the lab and writing down your observations in your submission is sufficient.

Appendix: OpenMP Programming

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 a thread id of `0`.

Shown on the following page is the OpenMP version of the canonical hello world program (`hello-omp.c`). Each parallel region starts with a `#pragma` directive that signals to the compiler that the following code block will be executed in parallel.

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

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

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

        /* Only master thread executes 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 i7-7700 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 by setting the environment variable `OMP_NUM_THREADS`.

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 amongst the threads. 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` clause. Example:

```c
#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 are 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();
        }
    }
}
```

Within 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 a single section, only that the master thread executes the code.

Synchronization Constructs

OpenMP provides multiple directives to coordinate threads and manage critical sections (as we have learned in Lab 1) in code.

- `barrier` directive: synchronizes all threads (threads wait until all threads arrive at the barrier).

```
#pragma omp barrier
```

- `master` directive: Specifies a region that must be executed only by the master thread.

```
#pragma omp master

    structured_block
```

- `critical` directive: Specifies a critical region that must be executed only by one thread at a time (example on following page)

```c
#include <omp.h>

int main(int argc, char *argv[])
{
    int x;
    x = 0;

    #pragma omp parallel shared(x)
    {

        #pragma omp critical
        x = x + 1;

    }  /* end of parallel region */

 }
```

- `atomic` directive: Works like a mini-critical section; specifies that a specific memory location must be updated atomically.

```c
#pragma omp atomic

    statement_expression
```

Resources

- For more details on OpenMP constructs, please refer to the LLNL OpenMP documentation at

  https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

- The Microsoft Visual C/C++ compiler supports OpenMP as well. Most of the examples you find there should work on Linux with the `gcc`, `g++` or `clang` compilers as well. You can learn more at

  https://msdn.microsoft.com/en-us/library/tt15eb9t.aspx

- perf reference

  https://perf.wiki.kernel.org/index.php/Main_Page

- perf manual: `$ man perf`

- Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processor

  http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

- OpenMP Reference Sheet for C/C++

  http://www.plutospin.com/files/OpenMP_reference.pdf