Lab 2
Parallel Programming with OpenMP and Performance Instrumentation

CS3210 – 2023/24 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

Login vs Compute Nodes

As a reminder, we have portioned the lab machines (without overlap) into two types.

1. **Login Nodes**: where you can run simple test scripts and submit Slurm jobs
2. **Slurm Compute Nodes**: nodes that will only run Slurm jobs (no direct SSH access!)

This is a common setup among compute clusters. You will only be allowed to interactively `ssh` into login nodes, and these are primarily for submitting Slurm jobs. You are free to run short commands on login nodes (rough guideline: programs that use 100% CPU for many seconds are not OK), but we reserve the right terminate any long running jobs to keep the nodes usable for all users.

For this lab, we will still run CPU-heavy tasks on login nodes in Part 1 and 2. The rest of the lab will use Slurm.

Remember that you can still refer to our **Student User Guide** for further information (`https://bit.ly/cs3210-student-guide`).

Logging in & Getting Started

1. Follow the instructions from previous labs / tutorials to `ssh` into one of our lab machines. Remember that you can now only `ssh` into login nodes!
2. **Change to your NFS directory first (i.e.,** `/nfs/home/your_username/`**)**
3. Get the lab code via `wget` `https://www.comp.nus.edu.sg/~srirams/cs3210/L2_code.zip`
4. Use `unzip` to extract the code.

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.cpp`):

```
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];
}
```

We are now going to run the above sequential matrix multiplication implementation.

- Compile the code:

  ```
  $ g++ -o mm0 mm-seq.cpp
  ```

- Run the program with a small matrix size in a terminal - observe that it completes immediately:

  ```
  $ ./mm0 10
  ```

Exercise 1

Run the sequential matrix multiplication implementation `mm0` with increasingly larger matrix sizes (try 500 – 1000) and observe how the runtime scales with matrix size.

You can prefix your commands with `/usr/bin/time -vvv` to observe a few statistics such as context switches, percent of CPU for the job, and user/system/wall 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.cpp`):

```
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:

  ```
  $ g++ -fopenmp -o mm1 mm-omp.cpp
  ```

- The `-fopenmp` flag enables the compiler to detect the `#pragma` directives, which would otherwise be ignored.
- Let's run our parallel version for matrix sizes that took many seconds in the serial version (matrix size 1000, using 8 threads).

  ```
  $ ./mm1 1000 8
  ```

Exercise 2

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. What determines the ideal number of threads to use?

Understanding OpenMP Fundamentals

Now that we've seen the power of OpenMP, let's understand the basics in further detail. We will only describe a small portion of OpenMP - you are free to read more about it in the OpenMP Quick Reference. Note that we use **OpenMP 4.5**.

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 worker 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 (included in this lab as hello-omp.cpp). Each parallel region starts with a #pragma directive that signals to the compiler that the following code block will be executed in parallel.

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

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

    /* Fork worker 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)
        {
            num_threads = omp_get_num_threads();
            printf("Number of threads = %d\n", num_threads);
        }
    } /* All worker threads join master thread and are destroyed */
}
```

Exercise 3

Read the contents (and comments) in hello-omp.cpp. Then, compile the hello-omp.cpp program as in the previous exercise. Use Slurm to run it on both an i7-7700 machine and an xs-4114 machine (srun should be enough to do this). Is there any difference in their output?

The parallel directive creates a team of threads, and each thread executes the code in the parallel region. The omp_get_thread_num() function returns the thread id of the calling thread. The omp_get_num_threads() function returns the number of threads in the current team. At the end of the parallel region, all threads join the 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 (note that without a parallel region, everything will execute sequentially), there is usually some work that needs to be done across the multiple threads, and different threads need to do different things. OpenMP provides a few 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. There are many *clauses* that the programmer can add on to the *for* directive to control its behavior.

  One such clause is the schedule clause: the programmer can control the order and the number of iterations assigned to each thread. 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 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.

This construct was used in the matrix multiplication exercise above, and is a very common way to parallelize programs easily.

Exercise 4

Compile and run omp-schedule.cpp. Do you notice the difference between the static and dynamic schedules?

Nesting Work-Sharing Constructs

Please note that OpenMP does not allow nesting of work-sharing constructs (including if the nested loop is called in a function). For example, the following code is not allowed:

```
#pragma omp parallel
{
  #pragma omp for
  for (i = 0; i < n; i++)
    #pragma omp for
    for (j = 0; j < m; j++)
      // can't nest the omp for directives
}
```

The compiler will likely throw an error, but even if it does not, the code will be silently serialized. **Please watch out for this common mistake!**. See this link for more info.

- **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();
        }
    }
}
```

Within the `sections` region, you see the declaration of two work sections. The sections may be passed to different threads for execution.

- Single section (`#pragma omp single`): Only a single thread will execute the code. The runtime decides which thread will get to execute.
- Master section (`#pragma omp master`): Only the master thread executes the code.

Exercise 5

Compile and run `omp-sections.cpp`. What can we say about the assignment of the threads to different portions of the code? Try adding the word `nowait` at the end of the line `#pragma omp sections` - how does the behavior change?

Synchronization Constructs

OpenMP provides multiple directives to coordinate threads and manage critical sections (as we have learned in Lab 1) in code. **This makes synchronization easier!**

- `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)

```
#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.

```
#pragma omp atomic
    statement_expression
```

Exercise 6

Compile and run `omp-sync.cpp`. Notice that it has a race condition. Try using both the `critical` and `atomic` directives to solve the issue. Which is faster?

Please see Appendix: OpenMP / Performance Resources to learn more. There's also a really good visual OpenMP tutorial at this link.

Part 2: Performance Instrumentation

Now that we understand a bit more about OpenMP, we can explore how to extract the **maximum performance** out of such parallel programs using **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 few 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 this run 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
```

Perf Event Sampling

If you ask perf to sample too many events (i.e., too many arguments after -e), it will not give you an *exact* count for all of those events. Instead, it will *sample* the events and give you an estimate. This is because the hardware event counters are limited in number, so the hardware cannot count all events at the same time. The number of available counters depends on the CPU being used.

- 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 ...]
```

Perf Metric Groups

perf list also shows the *metric groups* you can measure. This specifies a group of events to measure with the -M flag followed by a comma-delimited list of metric group names. For example, if there was a metric group called example, you could do:

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

Note that if the group measures too many events, you might get sampling!

- 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 on a shared machine (without Slurm), you might notice that the numbers vary significantly 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, if you take performance measurement locally, 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 7

Use **perf** to profile the OpenMP matrix multiplication program (`mm-omp.cpp`) with a varying number of threads. Observe the variation of different events counts for different runs. You can try to use the `-r` flag to run the program multiple times and get the average.

Part 3: Running and profiling compute-heavy applications with Slurm

You may have noticed that the runtimes and perf statistics you are measuring are not entirely consistent. As you are using a shared node, every user's programs are fighting for a share of your node's resources. Therefore, everyone's performance measurements are both slower than expected, and have large variances.
To rectify this, we will use Slurm to get accurate performance measurements.

To make things easier, we have created two Slurm batch job files (`seq-job.sh` and `omp-job.sh`) to *help you get started*. These are likely not sufficient to complete the lab writeup, so **please modify them as necessary**. Take special note that you may need more **memory** or **time** than specified in the job script, or use a different **partition**.

Exercise 8

Open the file `seq-job.sh` and familiarize yourself with the contents.

You will run this script via sbatch, in the format `sbatch seq-job.sh <matrix size>`, e.g., for a matrix size of 1000, please run:

```
$ sbatch ./seq-job.sh 1000
```

When the job completes, you should see a file that contains your job ID and ending with `.slurmlog`. This will contain the stdout/err for your job and you can see the results.

Compare your results using Slurm to Exercise 1. Are your runs notably faster or more consistent?

Exercise 9

Use **perf** to profile the OpenMP matrix multiplication program (`mm-omp.c`) with a varying number of threads while **using omp-job.sh**.

Some notable points:

- omp-job.sh requires two arguments, not one, because those two arguments are passed to mm-omp which requires two arguments.
- You can change the partition (specified within the script) to run on a different machine type.

Notice that the job already runs `perf` for you, but you should change the events that are measured with `-e`. Observe the variation of different performance event counts for different runs (you could try `perf -r`). Consider things like cache performance, branch performance, and page faults, to name a few important events.

Compare your results to Exercise 7. Are your runs notably faster or more consistent?

Part 4: Accurate Performance Analysis (Lab Submission)

In this section, we will use hardware event counters to analyze the performance of a parallel program. For the exercises below, **use Slurm** to **run the OpenMP matrix multiplication program** on **both the Desktop PC (Intel Core i7, i7-7700 partition) and the Server Tower (Intel Xeon, xs-4114 partition)** with $n =$ 1, 2, 4, 8, 16, 32, 64, 128, 256 threads. You should choose a reasonable matrix size (execution should take at least a few seconds). You may have a few different test scenarios, but there is no need to show results for all.

Exercise 10

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, and across machine types.

[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 that your program uses - 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 11

In `mm-omp.cpp`, the elements of for matrices A and B are stored in **row-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.cpp` 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, we don't care about the result value). You just need to access the elements in a different way.

Exercise 12

Compile and run (1) `mm-omp.cpp` and (2) your row-wise implementation from the previous exercise `mm-omp-row.cpp` 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). You may just run on one hardware type for this part.

Lab sheet (2% of your final grade):
You are required to produce a write-up with the results for exercises 10, 11, and 12. Submit the lab report in **PDF** form with file name format A0123456X.pdf (**Do not zip your file!**) via Canvas before **2pm, Wednesday, 20th Sep**. The document must contain:

- explanations on how you have solved each exercise (keep this to a few paragraphs in total)
- your results and the raw measured data - use graph(s) and try to justify your observations

Please describe your experiments in a way that can be replicated. Specify exactly what partitions / nodes you ran your performance measurements on.

Appendix: OpenMP / Performance Resources

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