High Performance Computing
Exercise #4
Max Points: 20

You should save and rename this document using the naming convention `MUid_Exercise4.docx` (example: `raodm_Exercise4.docx`).

Objective: The objective of this exercise is to gain some familiarity with:

- Explore basic compiler optimizations
- Effectiveness of compiler assisted explicit parallelism (enabled via OpenMP)
- Reserve multiple cores on a compute node to run interactive jobs
- Measure and compare runtimes of programs.

Submission: Once you have completed this exercise, upload:

1. This MS-Word document saved as a PDF file named with the convention *`MUid_Exercise4`*`.pdf` (example: `raodm_Exercise4.pdf`)
2. Optionally: The C++ program developed as part of this exercise named with the convention *`MUid`*`_Rational.cpp`.

Fill in answers to all of the questions as directed. For some of the questions that require outputs to be indicated, you can simply copy-paste appropriate text from the shell/PuTTY window into this document. You are expected refer to `LinuxEnvironment.pdf` document available in Handouts folder off Niihka. You may discuss the questions with your instructor.

Name: **Yan Yu**

Preliminaries

1. Log onto the Linux server for this course via the following steps (that were covered in the previous exercises and as illustrated in the LinuxEnvironment.pdf):
   i. Run the X-Server `Xming`.
   ii. Use PuTTY to log into the Linux server `redhawk.hpc.muohio.edu`.
   iii. When you log onto the server, you will be presented with a shell (**$**) prompt. You need to perform various tasks by typing commands at the shell prompt and pressing the ENTER (↵) key.

Task 1: Copy and review supplied code

Expected time for completion 15 minutes

For this exercise you are supplied with a simple `Rational` class that represents a rational number consisting of a numerator and denominator value. Copy the starter code using the operations below:

1. Create a suitable directory to store the files for this exercise. Ensure your `pwd` (present working directory) is the new directory you have created for this exercise. In general it is a good idea to have your files using a meaningful organizational paradigm.

   NOTE: If you have to look-up commands to successfully complete the above step, then you are not doing justice to this course and you work on learning Linux commands.

2. Copy the supplied starter files (don't miss the period at the end of the command below):

```
$ pwd
$ /home/raodm/csex43/exercises/exercise4
$ cp /shared/raodm/csex43/exercises_14/exercise4/* .
$ mv Rational.cpp MUid_Rationa.cpp
```

3. Study (5 minutes) the supplied `Rational.h` and `MUid_Rational.cpp` files. The API methods provided by `Rational` class is summarized in the UML below:

Rational
- numerator: int - denominator: int
+ Rational() + Rational(int, int) + operator double() const: double + toPrimeFactorCount(): Rational + operator<(rhs: const Rational&): bool + customCompare(lhs: const Rational&, rhs: const Rational&) # isPrime(number: const int): bool # getPrimeFactorCount(number: const int): int + operator+(other: const Rational&) const: Rational + operator+=(other: const Rational&) const: Rational& + operator*(other: const Rational&) const: Rational + operator==(other: const Rational&) const: bool

   Note: The last four methods (in red) are to be implemented by you in **Part 7** of this exercise (outside of lab session). However, study the various methods, their implementation, and how they are used by the methods `partA_tests` and `partB_Tests`.

Task 2: Base case timing measurements

Expected time for completion 15 minutes

For this Task ensure that the `partA_Tests` is uncommented in the `main` method in the supplied `Rational.cpp` class. Review the code in `partA_Tests` to ensure that you understand what this simple method is accomplishing.

The next step of the process is to measure the runtime characteristics of the supplied program to obtain a base case timing against which impact of other optimizations will be measured and compared.

Reserve 1 cores on Red Hawk and measure the runtime of your program in the following manner:

1. Reserve 4 cores (or CPUs) on a single node using the following command (there is a lower case ell in the `-l` part of the command below):

```
$ qsub -I -l"nodes=1:ppn=1"
```

   NOTE: Once qsub starts up a job on Red Hawk you will notice the following changes:
   - You will be automatically moved from the head node to a compute node. So note that the host name changes in your shell prompt.
   - You will be dropped back to your home directory (on the compute node) and you will have to change to the appropriate working directory.

2. Compile the program with necessary compiler flags.

```
$ icpc -std=c++11 -O0 -g -Wall MUid_Rational.cpp -o exercise4_noopt
```

3. Run, and measure the timing for the program from 5 independent runs as shown below without any compiler optimizations (by using `-O0` (dash big-Oh zero) to compile without any optimizations)

```
$ /usr/bin/time -v ./exercise4_noopt
…
```

   These timings indicate the following information (you are expected to memorize this and explain them in the exam):
   - User time: This time value indicates the total time spent executing code you developed. This value is proportional to the number of instructions in your program.
   - System time: The time spent by the OS performing tasks on behalf of the program
   - Elapsed time: The net wall clock time that the program took to finish.
   - Percentage CPU: The amount of CPU used by the program. Linux indicates 100% per core. Consequently, if a CPU has 4 cores, then this value can be as high as 400%.

4. Run the program five times and record your observations below in seconds. Note that each time the program is run, the value will change a bit (you should be able to explain why the values change a bit each time the program is run). Compute the average for each one of the columns in the last row of the table below:

Run #	User time	System time	Elapsed time	%CPU
1	21.10	0.03	21.22	99
2	21.11	0.03	21.31	99
3	20.73	0.04	21.15	98
4	21.05	0.06	21.80	97
5	21.07	0.03	22.01	95
Average	21.012	0.038	21.498	97.6

Task 3: Timing measurements with compiler optimizations

Expected time for completion 15 minutes

Background: Compiler optimizations

The `gcc/icc` compilers include many optimizations that can improve the runtime performance of a program. The manual page for `gcc/icc` includes a list of optimizations that can be enabled (http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html). You do not have to know what these optimizations do but in this course. However, some understanding of these optimizations falls under the category of "common sense" in computing. Consequently, you should invest some time reading about some of these optimizations outside the lab.

Typically, the `-O2` or `-O3` optimization levels are used for compiling validated programs. However, optimizations can make debugging programs harder. Consequently, it is best to compile programs without optimizations for troubleshooting and debugging purposes. Furthermore, `gcc/icc` take a bit longer to compile programs with optimizations enabled.

Experiments

Compile, run, and measure the timing for the program from 5 independent runs as shown below without compiler optimizations (by using `-O3` (dash Oh three) to compile with aggressive optimizations):

```
$ icpc -std=c++11 -O3 -g -Wall MUid_Rational.cpp -o exercise4_opt
```

Run the optimized executable (`exercise4_opt`) program five times and record your observations below in seconds. Note that each time the program is run, the value will change a bit (you should be able to explain why the values change a bit each time the program is run). Compute the average for each one of the columns in the last row of the table below:

Run #	User time	System time	Elapsed time	%CPU
1	20.83	0.10	21.75	96
2	20.72	0.01	21.35	97
3	20.83	0.01	20.97	99
4	21.41	0.00	21.48	99
5	20.34	0.02	20.44	99
Average	20.826	0.028	21.198	98

Note that on most CPUs and particularly the x86 CPUs, superscalar operations cannot be turned off. Consequently, in this specific exercise, the performance difference between `-O0` and `-O3` is expected to be negligible (but you should see a good performance improvement in the next task.

Task 4: Run the program with compiler-assisted parallelism

Expected time for completion 15 minutes

Background

The C++ Standard Library implementation provided by GNU (called `libstdc++`) also provides a "parallel mode". Using this mode enables existing serial code to take advantage of many parallelized standard algorithms that enables effective use of multi-core processors or multi-CPU machines which are common these days. The parallel mode operations can be enabled using compiler flags (namely: `-fopenmp -D_GLIBCXX_PARALLEL`) to enable use of parallel mode of operation whenever possible. This mode does not require any changes to the program. The compiler and standard library use heuristics to decide if an algorithm should be run in parallel. Consequently, some of the algorithms may not be run in parallel mode.

Exercise

Ensure you end the previous job (using the `exit` command) Reserve 4 cores on Red Hawk and measure the runtime of your program in the following manner:

1. Ensure you have ended the previous job. Reserve 4 cores (or CPUs depending on platform configuration) on a single node using the following command (there is a lower case ell in the `-l` part of the command below):

```
$ qsub -I -l"nodes=1:ppn=4"
```

2. Compile with OpenMP, run, and measure the timing for the program from 5 independent runs as shown below:

```
$ icpcg++ -std=c++11 -fopenmp -D_GLIBCXX_PARALLEL -O3 -g -Wall MUid_Rational.cpp -
o exercise4_omp
$ export OMP_NUM_THREADS=4
$ /usr/bin/time -v ./exercise4_omp ← only this command needs to be repeated 5 times
```

3. Run the Open MP enabled and optimized executable (`exercise4_omp`) program five times and record your observations below in seconds. Note that each time the program is run, the value will change a bit (you should be able to explain why the values change a bit each time the program is run). Compute the average for each one of the columns in the last row of the table below:

Run #	User time	System time	Elapsed time	%CPU
1	23.95	0.04	6.05	396
2	24.01	0.04	6.07	396
3	23.47	0.02	5.89	398
4	24.58	0.05	6.22	395
5	24.68	0.05	6.22	397
Average	24.138	0.04	6.09	396.4

Task 5: Record your inference

Expected time for completion 10 minutes

Using the average values computed in Task 2, Task 3, and Task 4, determine which one of the two approaches is faster and record your inference below. Why do you think (this is your opinion) the approach is faster? Ensure you contrast (1 or 2 sentences) the following aspects of the three versions of the program: elapsed time, user time, system time, and %CPU.

As observed from the charts above, task 4 with multi-threading is faster than the task 2 and task 3. The user, sys and elapsed time differences between task 2 and task 3 is not that huge, it's mainly because the CPU's superscalar operations cannot be turned off, but once when we switch to multi-threading optimization, there is a huge difference, the elapsed time is way shorter than the previous two, about 3.5 times faster, since there are 4 cores being reserved to coordinate with each other to perform the same task, each core can operate part of the job, so the real time is faster than 1 core. And since we used multi-core and multi-threading, the user time increases a little bit, because we signal compiler to use the multi-threading version of the library, so the code is much longer than the non multi-threading version, and because user time is proportional to lines of code, multi-threading version user time is longer than previous two. And also using multi-threading technique will use more resources, so the operation system will take time to do more sys calls, the system time also increases. The % of CPU usage indicates the sum of 4 cores' usage, so the number is about 400%.

DUE DATE: **TUE, Feb 17 2014, BY END OF CLASS**

Part 6: Submit files to Niihka

Once you have completed this exercise, upload:

1. This MS-Word document (duly filled with the necessary information) and saved as a PDF document named with the convention *MUid_Exercise4*.pdf (example: raodm_Exercise4.pdf)

Task 7: Implementing custom operators in `Rational` class

Expected time for completion 20 minutes

In the exams you will be expected to implement methods in various classes to implement necessary functionality. In this part of the exercise you are expected to implement additional operators in `Rational` class using the steps below:

1. End the interactive job, if you haven't already done so. Ensure you are on the head node on Red Hawk.
2. First comment out call to `partA_Test` in main method. Next uncomment `partB_Test` call and the `#define PARTB_TESTS` lines (in the `.cpp` files).
3. Compile the program (from `emacs`) and notice the error that the compiler generates when methods are declared but not defined.
4. Now implement the necessary operators in the source (`.cpp`) file using the functionality specified in the comments in the header (`.h`) file.
5. Once you have implemented the necessary operators, compile the program and ensure that the output is exactly as shown in the sample output below.

Sample Output

Once you have successfully implemented the necessary operators in Rational class, compile and run your program. The output should be exactly as shown below:

```
r1 + r2 : 1/2 +  1/2 = 4/4
r1 * r2 : 1/2 *  1/2 = 1/4
r1 * 10 : 1/2 *  10  = 10/20
r3      : 13/10
r1 == r2: 1/2 == 1/2 = true
r1 == r3: 1/2 == 13/10 = false
```

Part 8: Additional methods to practice for exams

Here are a few methods that you should be able to develop for this class. You should expect such question in exams:

- Develop a copy constructor for the `Rational` class.
- Develop a move constructor for the `Rational` class.
- Develop an assignment operator for the `Rational` class.
- Develop getter and setter methods for `numerator` and `denominator` instance variables.