**High Performance Computing**

**Exercise #5**

Max Points: 20

**Note: If you are using your personal machine then it is best to install a Linux virtual machine on your computer and use the Linux virtual machine.**

|  |
| --- |
| **You should save and rename this document using the naming convention MUid\_Exercise5.docx (example: raodm\_Exercise5.docx).**  **Objective**: The objective of this exercise is to gain some familiarity with:   * Explore basic compiler optimizations * Explore effectiveness of automatic parallelism (enabled via gcc’s parallel libraries) * Measure and compare runtimes of programs.   **Submission**: Once you have completed this exercise, upload:   1. This MS-Word document (duly filled with the necessary information) named and saved as a PDF file using the convention *MUid\_Exercise5*.pdf (example: raodm\_Exercise5.pdf) 2. 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](https://niihka.muohio.edu/access/content/group/7995c7ce-a855-4a0f-908c-ab8fac306f6a/Handouts%20_%20Tutorials/LinuxEvironment.pdf) document available in Handouts folder off Niihka. You may discuss the questions with your instructor. |

|  |  |
| --- | --- |
| **Name:** |  |

# Task #1: Copy and review supplied code

***Expected time for completion 20 minutes***

For this exercise you are supplied with a simple Rational class that represents a rational number consisting of a numerator and denominator value. Download the supplied starter code to your working directory.

1. Study 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 **Task 6** of this exercise. 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.

1. Compile, run, and measure the timing for the program from 5 independent runs as shown below without any compiler optimizations (by using –O0 (dash Oh zero) to compile without any optimizations)

|  |
| --- |
| $ g++ -std=c++11 **–O0** -g -Wall MUid\_Rational.cpp –o exercise5\_noopt  $ /usr/bin/time ./exercise5\_noopt  …  …  23.11user 0.00system 0:23.26elapsed 99%CPU (0avgtext+0avgdata 5792maxresident)k  3584inputs+0outputs (5major+402minor)pagefaults 0swaps |

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%.

1. 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 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |
| 5 |  |  |  |  |
| Average |  |  |  |  |

# Task 3: Timing measurements with compiler optimizations

***Expected time for completion 15 minutes***

## Background: Compiler optimizations

The gcc compiler includes many optimizations that can improve the runtime performance of a program. The manual page for gcc 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 takes 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 (pronounce as dash Oh three) to compile with aggressive optimizations:

|  |
| --- |
| $ g++ -std=c++11 **–O3** -g -Wall MUid\_Rational.cpp –o exercise5\_opt |

Run the optimized executable (exercise5\_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 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |
| 5 |  |  |  |  |
| Average |  |  |  |  |

# Task 4: Run the program with automatic multithreading

*Expected time for completion 15 minutes*

## Background

The C++ Standard Library implementation provided with the free GNU C++ compiler 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 implicitly enable use of parallel mode of operation whenever possible. The implicit 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

1. Compile with automatic multithreading enabled, run, and measure the timing for the program from 5 independent runs as shown below:

|  |
| --- |
| $ g++ -std=c++11 **-fopenmp -D\_GLIBCXX\_PARALLEL -O3** -g -Wall MUid\_Rational.cpp –o exercise5\_mt  $ export OMP\_NUM\_THREADS=2  $ /usr/bin/time ./exercise5\_mt |

1. Run the multithreaded and optimized executable (exercise5\_mt) 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 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |
| 5 |  |  |  |  |
| Average |  |  |  |  |

# Task 5: Record your inference

***Expected time for completion 15 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.

|  |
| --- |
| Use as much space as you need. |

# Task 6: 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 |

# Task 7: Submit files to Niihka

Once you have completed this exercise, upload:

1. This MS-Word document (duly filled with the necessary information) named with the convention *MUid\_Exercise5*.docx (example: raodm\_Exercise5.docx) save the document as a PDF file to upload it to Niihka.
2. The C++ program developed as part of this exercise named with the convention *MUid*\_Rational.cpp.

# Task 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.