National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

CS213: Advanced Programming

Class: BESE – 4B

Lab 4: Multithreading

Date: February 29, 2016

Time: 10:00 AM to 12:50 PM

Instructor: Mr. Numair Khan

Lab 3: Multithreading

Introduction

If you read the article “The Free Lunch is over,” by Herb Sutter (as you were required to do), you are probably aware that programmers will need to incorporate some level of parallelism into their code to take advantage of future multi-core microprocessors. Writing parallel programs is not always easy for a number of reasons: a parallel counterpart to your serial algorithm may not be known, parallel threads may interact unpredictably, or the portion of your program that can be effectively serialized may be very small. We will discuss these issues further in class. The purpose of this lab is to make you appreciate both the advantages of using parallel threads in your program, and some of their limitations. In addition, we will also look at how the usage of the **`restrict`** keyword affects our programs’ performance.

Objectives

After performing this lab students will be able to:

- Use the Pthreads library to write multithreaded C programs
- Parallelize a simple serial program

Tools/Software Requirement

- Pthreads

Description

Pthreads, or Posix threads, is a commonly used implementation of the IEEE Posix standard library found on Linux systems. Read the following tutorial on using Pthreads (read up to section 6):

https://computing.llnl.gov/tutorials/pthreads/

Make sure you understand how to create, run, terminate, and join threads. You should also understand how to compile a program that uses the Linux Pthreads library.

Lab Task 1

For your first task you are required to use Pthreads to parallelize the matrix/vector multiplication functions you wrote in the last lab. How you go about doing this is up to you. Matrix multiplication is what would be described as an *embarrassingly parallel* task, meaning it is extremely easy to parallelize.

Keeping the size of the matrices and vectors fixed at 1500x1500 and 1500x1, respectively, run each of your parallel functions with 2, 4, 16, and 32 threads. For each case record the *parallel speedup*. This is the ratio of the running time of the parallel program to the serial version. That is

$$Speedup = \frac{T_p}{T_s}$$

where T_p is the running time of your parallel version, and T_s the running time of the serial version.

Draw two bar graphs: one for matrix-matrix multiplication, and one for matrix-vector multiplication. Keep the number of threads (2, 4, 16, and 32) on the horizontal axis, and the speedup on the vertical axis. What trend do you observe in each graph? How can you explain this trend?

Lab Task 2

Consider the following two prototypes for the matrix-matrix multiplication function:

```
Matrix mult1 (Matrix A, Matrix B);

Matrix mult2 (Matrix *A, Matrix *B);
```

How are the two different? The first function passes the arguments to the function *by value*. This means each time the function is called, copies of the arguments are created on the stack. For a 1500x1500 floating point matrix, this would mean copying 8.5MB of data (depending on how the array of matrix elements is declared). Why copy the data when it already exists in memory? The second function passes only a pointer to the memory locations of the matrices. This is expected to be much faster since only a pointer is copied (the pointer variable is till passed by value; we will study *passing by reference* later in the course).

Create two variants of your matrix multiplication function: one that accepts arguments by value, and another that accepts pointer arguments. Measure their running time when passed matrices of size 1500x1500, 5000x5000, and 10000x10000. We expect the second variant to perform much faster. Do you notice a significant difference in their running times? Why or why not?

Lab Task 3

Remember how the compiler plays it safe and does not assume strict aliasing in case of pointers to the same data type. We can inform the compiler that two pointers never overlap by using the `restrict` keyword alongside the pointer declaration:

```
int *restrict arrayPtr;
```

This allows the compiler to optimize the generated machine code by telling it that `arrayPtr` can never overlap the memory pointed to by another `int` pointer. (Note, that it is the responsibility of the programmer to make sure that the pointer never overlaps.)

Use the `restrict` keyword in your classes where you declare the pointer to the dynamic array of matrix/vector elements, and repeat Task 1. Plot the graphs. How do these graphs compare to the previous graphs where `restrict` was not used?

Deliverables

1. Your source code
2. A PDF or Word file with the graphs and your explanation for Tasks 1 and 3, and your results and explanation for Task 2.