Assignment 3: C++ 11 Concurrency

This homework assignment, worth 20 marks and 5% of your final grade, is due by 4:00 PM on Friday, Oct. 3rd. Submit your solution (a printed and stapled/bound copy of `Edit.cpp`) to the ECE 312 LEC A1 assignment box outside the ECERF reception (2nd floor). Include your name, your student ID, the percentage of the submission (excluding code supplied with the assignment) that is original by you, and the names of all other contributors to your submission (indicate the percent contributed by each).

Electronic and wrong-box submissions will not be accepted. If any part of the assignment is submitted late, the whole assignment will be penalized 2 marks, i.e., 10% of the maximum mark, per business day. Submissions received, in whole or in part, after 4:00 PM on Tuesday, Oct. 7th, will receive 0 marks. Do not cheat, do not plagiarize, do not misrepresent facts, and do not participate in an offence!

Objectives

Concurrency need not entail parallel processing. With multi-core processors, however, such hardware concurrency is possible, which may lead to faster computation by concurrent algorithms compared to sequential ones. In this assignment, you will program, document, and test sequential and concurrent subroutines to multiply a matrix and vector. The program will use some concurrency features of the recent C++ 11 standard, which facilitates the porting of concurrent algorithms across platforms.

Getting Started

In Visual Studio (Express) 2012 or 2013, create a new one-project solution with `Main.cpp` and `Edit.cpp` as source files. Put `Input1.txt` in the project's working directory. Set it and `Output1.txt` as input and output files, respectively, as follows. Right click on your project in the Solution Explorer and select Properties. Choose Configuration Properties ≫ Debugging in the left pane. In the right pane, by Command Arguments, enter "`Input1.txt Output1.txt`" without quotes and click on OK.

Build and run the project. Verify that it creates an `Output1.txt` file successfully. If you cannot build and run the code, try the ETL E5-012 lab, where Visual Studio 2012 is available on Linux via VMware Player. Visual Studio 2012 may also be available on the Windows PCs in the ETL E4-013 lab.

In addition to this document, please review: the Wikipedia page on row-major order; the C++ 11 concurrency examples (see eClass) that were discussed in class; and all code and comments, also discussed in class, given in `Main.cpp` and `Edit.cpp`. Further information is linked below.

Technical Specifications

Write/edit the `seqMultiply` and `conMultiply` functions (`Edit.cpp`). Although some code is given there for Getting Started, you may delete/comment it except for the "`const uint THREADS`" statement. Do not change any other code, including the `seqMultiply` and `conMultiply` function interfaces. Do not add `include` statements or modify existing ones. Your entire solution must be programmed into the `Edit.cpp` file. You may add other (helper) functions if they help you develop a solution. Do not use global variables and, insofar as possible, do not use static local variables.

The `seqMultiply` and `conMultiply` functions have the same interface. `A`, `x`, and `y` are pointers to the base address of three existing arrays, i.e., where sufficient memory has been allocated. Moreover, `A` is a defined matrix of size `rows` × `cols`, in row-major order, and `x` is a defined vector of size `cols` × `1`. Both functions compute the matrix-vector product `A`·`x` and store the result in `y`, a vector of size `rows` × `1`.

According to linear algebra, where matrices are vectors are normally represented using a bold Roman font instead of a plain monospaced font as in programming, we can rewrite $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$ as follows:

$$\begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_4 \end{pmatrix} = \begin{pmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_4 \end{pmatrix} \cdot \mathbf{x} = \begin{pmatrix} \mathbf{A}_1 \cdot \mathbf{x} \\ \mathbf{A}_2 \cdot \mathbf{x} \\ \vdots \\ \mathbf{A}_4 \cdot \mathbf{x} \end{pmatrix},$$

where each $\mathbf{y}_k = \mathbf{A}_k \cdot \mathbf{x}$, with $1 \leq k \leq 4$, is independent of the others. Assuming $\mathbf{A}$ is an $m \times n$ matrix, where $m \geq 4$ does not have to be an exact multiple of four, we can partition it four-ways into three $m' \times n$ matrices, with $m' = \lfloor m/4 \rfloor$, and one $(m - 3m') \times n$ matrix. Although these equations have been written for a four-way partition, they may be easily rewritten for a `THREADS`-way partition.

Implement the `seqMultiply` function sequentially, i.e., without using any concurrency. If the number of rows, `rows`, is less than the given constant, `THREADS`, then the `conMultiply` function should also operate sequentially. Otherwise, it should operate concurrently using exactly `THREADS` threads, main thread included. Following the above pattern, each thread should compute one partition.

Avoid buffer overflows and memory leaks. Determine whether or not mutual exclusion is needed, and implement or do not implement it, as appropriate. Construct and terminate threads properly.

Presentation Specifications

Format your code professionally. Compare your placement of curly braces, i.e., `{` and `}`, to that of the given code. Use a similar approach or another professional approach. Code of the same code block should be left aligned and the indentation should consistently match the nesting level. In Visual Studio, selecting EDIT ≫ Advanced ≫ Format Document from the menu may correct alignment errors.

Ensure that long indented lines do not overflow without indentation when printed. Either break them into multiple indented lines (the compiler ignores line breaks) or print in landscape mode.

Write suitable comment headers for all functions you write/edit, which clearly describe the purpose, inputs, and outputs of each function. Functions that use static local and/or global variables are not fully defined this way. Describe, in the comment header, the state-driven behaviour of functions that use such variables. This is one good reason to minimize/simplify the use of state variables.

Review your working solution and look for ways to simplify it so that it would be easier to follow. For example, use refactoring to reduce code smell. This is another good reason to minimize/simplify the use of state variables. Furthermore, give helpful comments beyond those in the comment header.

Compared to a sequential algorithm, a concurrent algorithm incurs some overhead due to concurrency itself. As part of the above review, use refactoring to minimize/simplify the overhead.

Hints

As discussed in class, you can use the given `dispMatrix` function (`Main.cpp`) to help debug your code. See the Getting Started code for an example. The function outputs, in MATLAB format, a matrix, in row-major order, or a vector to a file called `Debugging.m`. Either analyze the file with a text editor and calculator, or run it in MATLAB where you can analyze the matrices and/or vectors further.

Do *not* submit the output file, `Output1.txt`, generated by your code (corresponding to the given input file, `Input1.txt`). Your code is marked against the specifications, irrespective of any submitted output. Your solution probably has a technical error if the `RMSE` is not always zero. Use the debugger and the advice given above. Zero `RMSE` does not mean your solution is free of technical errors.

Once you are confident that your code works, test it with larger matrices and vectors and more repetitions (delete/comment all `dispMatrix` statements first, otherwise you risk filling your hard disk and crashing your computer). Put `Input2.txt` in the project's working directory. Using the Getting Started instructions, set it and `Output2.txt` as input and output files, respectively.

Each line of the input file specifies the number of rows, the number of columns, and the number of repetitions to use in the simulation. These are echoed to each corresponding line of the output file, but the output file also reports the average time taken by the sequential and concurrent algorithms, as well as the root-mean-square error between the result vectors computed by each method.

On your instructor's dual-core PC, he observed a twofold speed-up with the concurrent algorithm, as compared to the sequential algorithm, for large-enough matrices, vectors, and repetitions. The speed-up factor may not match the hardware concurrency indicated in the output file, e.g., this was four for your instructor's dual-core PC because of hyper-threading on each core. Whereas the algorithm was unsuitable for hyper-threading, it did exploit multi-core processing successfully.