CX 4010 / CSE 6010
Assignment 7
Parallel Computing

Due Dates:

- Due: 11:59 PM, Monday, November 18, 2019 (no revisions for this assignment)
- No late submissions will be accepted

1. Introduction

This assignment involves an introduction to parallel programming, i.e., software designed to execute on multiple processors of a computer. Here, you will use a software library called *OpenMP* to write your parallel code. OpenMP uses a shared memory programming model. You will work on an individual basis for this assignment (no teams).

The assignment involves writing parallel code to perform matrix multiplication, i.e., compute `C=A*B` where `A`, `B`, and `C` are matrices and `*` is matrix multiplication. You may assume the matrices are all square with N rows and N columns.

Parallelization can be accomplished by observing that `C[i,j]` is simply the dot product of row i of `A` and column j of `B`. The N*N dot products could all be performed in parallel by mapping each to a different thread. Here, you will use computation of a single *row* of the result matrix (N dot products) as the unit of computation distributed among the threads, resulting in a maximum of N-fold parallelism.

Your code must be designed to work so the number of threads can be easily changed (e.g., using `#define` or a command line argument without making significant changes to the code.

2. Static Mapping

Assume there are K threads performing the computation. In this approach, each thread is statically assigned a set of rows of the result matrix `C` for which it is responsible for computing results. Specifically, thread 0 is responsible for computing values for rows 0, K, 2K, … of `C`. Thread 1 computes values for rows 1, K+1, 2K+1, … In other words, the rows of the matrix are assigned in round robin fashion to the different threads.

This approach is called a static mapping because the assignment of rows (computations) to threads is determined in advance, before the program begins to execute.

3. Experiments

Construct a set of experiments to measure the performance (execution time) of the parallel implementation and a sequential implementation. Note that in general, an execution using one thread is not the same as a sequential execution because it will have parallel processing overheads. However, for this problem, these will likely result in similar run times. *SpeedUp(K)* is defined as the execution time of the sequential code divided by the execution time of the parallel code using K threads, and indicates now many times faster the parallel code executes.

Create two different "sizes" of the matrix multiplication computation. Define a "small" matrix as one with 50 rows and 50 columns. Define a "large" matrix by setting the number of rows and columns (N) to a value where the execution of the sequential code is approximately a minute or two.

Your code should fill the matrix with random numbers in the interval [0.0, 1.0]. All values should be double precision floating point numbers. You will need to verify the program computes correct results (e.g., you can use matlab for this purpose) and document this in your report. Measure the runtime of your codes for the small and large matrices varying the number of threads. Show plots of speedup as the number of threads is varied for the small and large matrices. Explain your results in the project report.

You will need a multiprocessor computer to perform your experiments. Use your account on the Georgia Tech cluster for this purpose.

4. Dynamic Mapping (Extra Credit)

This part of the assignment is optional. You can receive up to 20% of the total value of the assignment as extra credit for completing this part.

To complete this part, uses a dynamic approach to assign computations to threads. In the dynamic approach a pool of “worker threads” is used. This means you create K threads to perform the computation. Each worker thread repeatedly accesses a global data structure to allocate a single row of the result matrix to compute. It then performs this computation, and then goes back to the global data structure to allocate another piece of work to do. This process continues until the entire matrix computation is complete. Each row of the result matrix must be assigned to exactly one thread.

In other words, each worker thread executes the following loop:

```
While (rows of the result matrix have not been computed) {
     Allocate a row of the result matrix to compute
     Compute results for this row of the result matrix
}
```

Note that when using this approach, the specific rows of the matrix assigned to a certain thread is not known until during the execution of the computation itself. Further, in repeated executions of the same code a particular thread may compute different rows of the result matrix. However, the same results should always be computed independent of how the rows are mapped to threads.

To implement the dynamic allocation approach, you can define a single global variable, initialized to zero, that indicates the number of the row that should be allocated next to a worker thread. Each worker thread can simply read this variable to allocate a row to compute, and then increment the variable so the next worker is allocated a different row. Be sure to place accesses to this global variable in a critical section to avoid race conditions which may lead to erroneous results, e.g., a row being computed twice!

Compare the performance of this implementation by comparing its execution time with the static mapping and sequential implementations. Explain the results you obtained from these experiments.

5. Implementation Notes

You will need a timer to determine the execution time of your code. The C library `<time.h>` can be used for this purpose. This will enable your program to read wallclock time, i.e., time on the computer, enabling you to compute elapsed time values at very high precision. This will be important, especially for the small matrix computation.

You may allocate memory for the arrays statically as global variables, or dynamically using `malloc()`. Note the arrays must be defined as shared variables for your program to work correctly.

6. Report

Turn in a report including a brief description of your software and explanation how you verified it computes correct results. Turn in your software in a single zip file. Your software must be well documented and include comments so the code is easy to understand. You must include a README file with instructions on how to compile and run your program.

Write up the results of your timing experiments and show graphs depicting the speedup results you obtained. Explain any anomalous, unexpected results. If you completed the extra credit portion of the assignment, which parallel version yields better performance? Explain why. Further, speculate in what situations (in general) a dynamic mapping approach might perform better than the static approach, and what situations the opposite might be true.

7. Collaboration, Citing, and Honor Code

As a reminder, please refer to the course syllabus regarding rules and expectations regarding collaborating with other students and use of other resources, including materials available on the web.