CSci 364
Spring, 2020
Programming Assignment #4
Due: April 24, 2020
75 Points + 10 Bonus points

OpenMP Matrix-Matrix Multiplication with written report

Phase I
Write a C++ program that multiplies two matrices and stores the result in a third matrix. The program should read the matrix size from the command line. A second but optional command line argument is used to signal whether or not to print the product matrix. The program should print the final product matrix if the second command line argument is a “Y”, “y”, or “1”. If the print flag argument is omitted, the product matrix should not be printed.

Ex. $ ./mm <matrix size> [<print flag>]

Dynamically allocate space for each matrix. The matrices should both be declared to store integers. Here are two examples of dynamic allocation of memory for a matrix of integers.

1) an array of pointers to separate arrays of integers
```
int **a = new int*[rows];
for (int i = 0; i < rows; i++)
      a[i] = new int[cols];
// access row 2, col 4 with a[2][4]
```

2) a contiguous array of integers
```
int *a = new int[rows * cols];
// access row 2, col 4 with a[2 * tot_columns + 4]
```

Deallocate as follows.

1) an array of pointers to separate arrays of integers
```
for (int i = 0; i < rows; i++)
      delete[] a[i];
delete[] a;
```

2) a contiguous array of integer
```
delete[] a;
```

Initialize each element of the input matrices to 1. This should be considered a test mode. The product matrix should be initialized to 0.

Next, perform the matrix-matrix multiplication.

Finally, if the print flag is true, print the product matrix to the console window one row per line. Test your work and confirm that the product is correct.

```
$ ./mm 3 y
3 3 3
```

3 3 3
3 3 3

Phase II
Enable a compile-time flag to control how the input matrices are initialized. In test mode initialize the input matrices as above. In normal mode initialize the matrices with random numbers in the range [0, 100]. DO NOT USE STANDARD C rand() FUNCTION. Use the C++11 random number facilities; an example follows. Please keep the engine seed = 1; this will make grading much easier.

```
#include <random>
    ...
    // initialize engine and distribution
    const int seed = 1;
    std::mt19937 engine(seed);
    std::uniform_int_distribution<int> dist(0, 100);
    …

#ifdef TEST
    matA[i][j] = 1;
#else
    matA[i][j] = dist(engine);
#endif
```

Compile for TEST mode as follows.
```
$ g++ std=c++11 -DTEST mm.cpp -o mm
```

Compile for normal mode by omitting the “-DTEST” string.

Phase III
Parallelize the outer loop of the matrix-matrix multiplication using OpenMP. Add pragma statement(s) to the source code and compile with the -fopenmp compiler option. Confirm that the product is still correct.

Print to the console the difference between the current time before and after the matrix multiplication (or use double **omp_get_wtime()**). This difference represents the runtime performance of the matrix multiplication and should include any OpenMP overhead for creating and scheduling the team of threads. Be sure not to include processing command line arguments, allocating memory, initializing matrices with values, or printing the multiplication results.

Phase IV
Run your program with **matrix size of 2000** using different schedule types, chunk sizes, and team sizes on the hodor-cpu partition. Use static and dynamic schedules, chunk sizes of 1, 4, and 8, and team sizes of 1, 4, and 8. (That’s 2 * 3 * 3 = 24 configurations.) Collect runtime performance data for 5 runs of each configuration. For reference, my results with 1, 4, and 8 cores and static(default) scheduling were approximately 12, 3.2, and 1.7 seconds, respectively.

Calculate the average runtime for each configuration. Use the averages to calculate speedup. Use the static, chunk size 1, and team size 1 configuration as the numerator in the speedup equation. Calculate the speedup of the remaining 11 configuration.

Write a report in PDF format that shows the raw data, the average runtimes, and the speedups. Which configuration had the best runtime performance and why? Given the best configuration, what matrix size can be processed in the same time as the slowest configuration at size 1200? Was schedule type or chunk size more significant in improving speedup?

Bonus (10 points): Write functions for allocating, deallocating, and printing matrices.

Submit your source code file and a PDF formatted report as a tar or zip file.