# CSCI 463 Assignment 6 – C++ Multithreading

10 Points

**Abstract**

In this assignment, you will implement a C++ multithreaded application that will sum the elements of a 2-dimensional matrix using either *static* or *dynamic* load balancing. Dynamic load balancing will demonstrate the use of a mutex lock in the critical section of code that selects the next row (work) for each thread to process. Static load balancing will demonstrate that when a workload only contains items requiring uniform time to process, some advance planning can, in the best of cases, eliminate the need to create critical sections of code in the first place (thus simplifying a solution.)

## 1 Problem Description

Sum the contents of a 2D matrix in a multithreaded application that uses static or dynamic load balancing based on a command-line argument.

## 2 Files You Must Write

You will write a C++ program suitable for execution on `hopper.cs.niu.edu` (or `turing.cs.niu.edu`.)

Your source file *MUST* be named exactly as shown below or it will fail to compile and you will receive zero points for this assignment.

Create a directory named `a6` and place within it the following file:

- `reduce.cpp` Your entire application is implemented in this file.

### 2.1 `reduce.cpp`

- To keep this assignment simple, create these (and only these) global variables for use by the threads in the application:

```
1  constexpr int rows = 1000;      ///< the number of rows in the work matrix
2  constexpr int cols = 100;       ///< the number of cols in the work matrix
3
4  std::mutex stdout_lock;         ///< for serializing access to stdout
5
6  std::mutex counter_lock;        ///< for dynamic balancing only
7  volatile int counter = rows;    ///< for dynamic balancing only
8
9  std::vector<int> tcount;        ///< count of rows summed for each thread
10 std::vector<uint64_t> sum;      ///< the calculated sum from each thread
11
12 int work[rows][cols];           ///< the matrix to be summed
```

- `void sum_static(int tid, int num_threads)`

  Implement the logic needed to sum the rows of the matrix using *static* load balancing to determine which rows will be processed by each thread.

  Use the thread ID (passed in from `main()`) to determine the first row for each thread and then advance the row number by `num_threads` to determine the next row to process.



/home3/projects/niu/NIU/courses/463/2022-fa/assignments/a6/handout.tex
jwinans@niu.edu 2022-08-18 14:53:29 -0500 v2.1-dirty

- `void sum_dynamic(int tid)`

  Implement the logic needed to sum the rows of the matrix using *dynamic* load balancing to determine which rows will be processed by each thread.

  Each thread must use a mutex lock to access the global (and volatile) `counter` variable in the critical section to determine the next row to process. Do not hold the lock for any more of the thread logic than is absolutely necessary!

- `int main(int argc, char **argv)`

  Provide a `main()` function so that it accepts the command-line parameters (and reflect them in a proper Usage statement) as discussed below. See the on-line manual for getopt(3) for details on how to use it to parse command-line arguments.

  The command-line arguments you must provide are:

  - `[-d]` Use dynamic load-balancing. (By default, use static load balancing.)
  - `[-t num]` Specifies the number of threads to use. (By default, start two threads.) Use:

    ```
    std::thread::hardware_concurrency()
    ```

    to determine the number of cores in the system. DO NOT start more threads than the system has cores!

  If any command-line arguments are invalid then print appropriate error and/or Usage messages and terminate the program in the traditional manner. (See https://en.wikipedia.org/wiki/Usage_message.)

# 3 Input

This program has no input.

Initialize the data in the global `work` matrix using the `rand()` function from the standard C library. See `rand(3)` for more information.

Note that `rand()` will always generate the same values, in the same order, if it is seeded to the same initial value. (Note that it is possible that `rand()` could work differently on different systems. The numbers shown below are those you will see when running on `hopper`.)

Seed your random number generator like this:

```
srand(0x1234);
```

You must initialize work matrix in the same order as the reference key to get the same output! Specifically, you must initialize your matrix from left to right, top-down, starting from the top. That is, set all the columns for row 0, then row 1,...

# 4 Output

Your program will be tested with a combination of the command-line arguments and will be `diff`'d against the output from a reference implementation.

Note that due to the varying load on the machine, your threads may start and complete in a different order than the reference output below. Your dynamic load balancing may differ in the number of rows summed by each thread between runs of your program as shown below. However, the sums of your static threads and the gross sum value in all cases must match the reference output to be considered correct.

```
winans@hopper:~$ ./reduce
8 concurrent threads supported.
Thread 0 starting
Thread 1 starting
Thread 1 ending tcount=500 sum=53670497890830
Thread 0 ending tcount=500 sum=53678649666216
main() exiting, total_work=1000 gross_sum=107349147557046
winans@hopper:~$ ./reduce -d
8 concurrent threads supported.
Thread 0 starting
Thread 1 starting
Thread 1 ending tcount=528 sum=56512359755886
Thread 0 ending tcount=472 sum=50836787801160
main() exiting, total_work=1000 gross_sum=107349147557046
winans@hopper:~$ ./reduce -d
8 concurrent threads supported.
Thread 0 starting
Thread 1 starting
Thread 1 ending tcount=545 sum=58565707641474
Thread 0 ending tcount=455 sum=48783439915572
main() exiting, total_work=1000 gross_sum=107349147557046
winans@hopper:~$ ./reduce -d -t942
8 concurrent threads supported.
Thread 0 starting
Thread 1 starting
Thread 2 starting
Thread 3 starting
Thread 5 starting
Thread 7 starting
Thread 4 starting
Thread 6 starting
Thread 4 ending tcount=88 sum=9457033734061
Thread 1 ending tcount=175 sum=18832858281021
Thread 6 ending tcount=54 sum=5779547738442
Thread 2 ending tcount=125 sum=13292384416291
Thread 0 ending tcount=224 sum=24120680043779
Thread 3 ending tcount=109 sum=11701029247359
Thread 7 ending tcount=110 sum=11811841106533
Thread 5 ending tcount=115 sum=12353772989560
main() exiting, total_work=1000 gross_sum=107349147557046
winans@hopper:~$ ./reduce -t3
8 concurrent threads supported.
Thread 0 starting
Thread 1 starting
Thread 2 starting
Thread 1 ending tcount=333 sum=35774388649170
Thread 0 ending tcount=334 sum=35960930241047
Thread 2 ending tcount=333 sum=35613828666829
main() exiting, total_work=1000 gross_sum=107349147557046
```

# 5 How To Hand In Your Program

When you are ready to turn in your assignment, make sure that the only files in your a6 directory is/are the source files defined and discussed above. Then, in the parent of your a6 directory, use the mailprog.463 command to send the contents of the files in your a6 project directory in the same manner as we have used in the past.

# 6 Grading

The grade you receive on this programming assignment will be scored according to the syllabus and its ability to compile and execute on the Computer Science Department's computer.

*It is your responsibility to test your program thoroughly.*

When we grade your assignment, we will compile it on `hopper.cs.niu.edu` using these exact commands:

```
g++ -g -ansi -pedantic -Wall -Werror -std=c++14 reduce.cpp -pthread -o reduce
```



/home3/projects/niu/NIU/courses/463/2022-fa/assignments/a6/handout.tex
jwinans@niu.edu 2022-08-18 14:53:29 -0500 v2.1-dirty