CS6771 Assignment Five 2017

Parallel Bucket Sort

Due Date: 11:59pm Sunday 29 October 2017

Worth: 10%

Aims:

- To learn how to write multithreaded programs in C++
- To learn how to use mutex locks to avoid race conditions in multiple threads
- To learn about potential performance bottlenecks and performance evaluation of parallel programs

Your task is to take a slow single threaded sort algorithm and redesign it into a parallel radix sort algorithm that sorts a vector of numbers as quickly as possible.

The sorting algorithm takes a vector of unsigned ints and sorts them by using a MSD (Most Significant Digit) radix sort, which uses lexicographic order.

For example, given a vector [32, 53, 3, 134, 643, 3, 5, 12, 52, 501, 98], the sorted output would be [12, 134, 3, 3, 32, 5, 501, 52, 53, 643, 98] (as if a shorter number were conceptually left-justified and padded on the right with "-1" to make it as long as the longest number for the purposes of determining sorted order).

A MSD radix sort can be done easily through a parallel bucket sort. A bucket sort takes the most significant digit of each number and groups the list of numbers with the same digit into one bucket. For example, the vector given above may be split into the following buckets according their most significant digits:

```
Bucket 1: [ 134, 12 ]
Bucket 3: [ 32, 3, 3 ]
Bucket 5: [ 53, 5, 52, 501]
Bucket 6: [ 643 ]
Bucket 9: [ 98 ]
```

Afterwards, each bucket is sorted recursively, starting with the next most significant digit:

```
Bucket 1: [ 12, 134 ]
Bucket 3: [ 3, 3, 32 ]
Bucket 5: [ 5, 501, 52, 53]
Bucket 6: [ 643 ]
Bucket 9: [ 98 ]
```

Finally, all the buckets are concatenated together in order: [12, 134, 3, 3, 32, 5, 501, 52, 53, 643, 98]

Requirements:

In this assignment, you will need to create two files: `BucketSort.cpp` and `BucketSort.h.` These files can contain as many classes, functions and structs as you require. However, because this assignment is about speed, there is no requirement that any structure must strictly adhere to object oriented design principles (e.g., structure member fields do not need to be private). *For any single-threaded solution without using multiple threads, a mark of 0 will be awarded.*

`BucketSort.h` must contain the following struct (which can be added to):

```
#ifndef BUCKET_SORT_H
#define BUCKET_SORT_H

#include <vector>

struct BucketSort {

        // vector of numbers
        std::vector<unsigned int> numbersToSort;

        void sort(unsigned int numCores);
};


#endif /* BUCKET_SORT_H */
```

The member field `numbersToSort` will be modified by user code to add numbers to be sorted to a BucketSort object. The same member field will then be read from after the sort method has been called to confirm that the numbers have been sorted correctly. Your main task is to write a multithreaded definition of the sort member function in your `BucketSort.cpp` file.

The sort method will be passed an unsigned int containing the number of CPU cores that are available to be used. You do not need to adhere to this number when creating threads, however, it can be useful as it is not sensible to create more threads than there are CPU cores available. Secondly, in some test cases, the number of cores available may be less than the true number of CPU cores available. This is to ensure that the system running the application is able to keep other operating system tasks running. *If at any point during execution the number of threads your application is using exceeds the numCores limit for a given test case (used in `pbs.sort(numCores)`) you program may be terminated by the marking system and you will score 0 for that test case.*

Single Threaded Starter Code

The following definition of the sort member function is the reference solution for the correct sort output. Your multithreaded implementation should produce the same output as this function, and should be much faster.

```
#include "BucketSort.h"

#include <algorithm>
#include <cmath>

bool aLessB(const unsigned int& x, const unsigned int& y, unsigned int pow) {

        if (x == y) return false; // if the two numbers are the same then one is not less than
the other

        unsigned int a = x;
        unsigned int b = y;

        // work out the digit we are currently comparing on.
        if (pow == 0) {
                while (a / 10 > 0) {
                        a = a / 10;
                }
                while (b / 10 > 0) {
                        b = b / 10;
                }
        } else {
                while (a / 10 >= (unsigned int) std::round(std::pow(10,pow))) {
                        a = a / 10;
                }
                while (b / 10 >= (unsigned int) std::round(std::pow(10,pow))) {
                        b = b / 10;
                }
        }

        if (a == b)
                return aLessB(x,y,pow + 1);  // recurse if this digit is the same
        else
                return a < b;
}

// TODO: replace this with a parallel version.
void BucketSort::sort(unsigned int numCores) {
        std::sort(numbersToSort.begin(),numbersToSort.end(), [](const unsigned int& x, const
unsigned int& y){
                return aLessB(x,y,0);
        } );
}
```

Sample User Code

Test cases for this assignment will involve user code creating large vectors of random numbers and calling the sort method. Your code will be limited by time and memory with tests increasing in difficultly through decreasing time and memory limits and increasing vector sizes.

The following sample test case shows how this is done. For the given single threaded code, this sort takes around about 23 seconds to run on the CSE server williams (with 8 cores). In comparison the model multithreaded solution runs for about 6 seconds (without any optimisation being applied).

```
#include <iostream>
#include <random>
#include <thread>

#include "BucketSort.h"
```

```
int main() {

        unsigned int totalNumbers =     500000;
        unsigned int printIndex =       259000;

        // use totalNumbers required as the seed for the random
        // number generator.
        std::mt19937 mt(totalNumbers);
        std::uniform_int_distribution<unsigned int> dist(1, std::numeric_limits<unsigned
int>::max());

        // create a sort object
        BucketSort pbs;

        // insert random numbers into the sort object
        for (unsigned int i=0; i < totalNumbers; ++i) {
                pbs.numbersToSort.push_back(dist(mt));
        }

        // call sort giving the number of cores available.
        const unsigned int numCores = std::thread::hardware_concurrency();
        pbs.sort(numCores);

        std::cout << "number of cores used: " << numCores << std::endl;

        // print certain values from the buckets
        std::cout << "Demonstrating that all the numbers that start with 1 come first" <<
std::endl;
        std::cout << pbs.numbersToSort[0] << " " << pbs.numbersToSort[printIndex - 10000]
                << " " << pbs.numbersToSort[printIndex] << " " <<
pbs.numbersToSort[pbs.numbersToSort.size() - 1]
                << std::endl;

}
```

Tips:

- Consider carefully when you need shared and local memory.
- Consider carefully when you need to use mutexes around shared memory.
- Consider carefully when you need to use multithreaded code and when you need to use single threaded code.
- You may need to split the sort into multiple sections where threads are created, destroyed and later additional threads are created.
- You shouldn't need to use condition variables, futures or thread pools in this assignment, but you can use these if they help you.
- You shouldn't need any pointers or dynamic memory to complete this assignment.
- The lecture slides and tutorials have many code snippets that you may find helpful.
- You shouldn't need to create any additional classes or structs (but you may if you like).
- The reference solution is 220 lines including comments.
- Do not use other libraries (e.g., boost).
- The textbook: Wilkinson, B. and Allen, M., *Parallel Programming*, 2nd Edition (2005), Pearson Prentice Hall., provides further details and ideas on how to implement the parallel bucket sort.
  A six page pdf photocopied extract covering the relevant section is available here. Figure 4.10 is the best diagram of what you should be working your code towards.

Testing

Place all your code in a files called BucketSort.h and BucketSort.cpp

Ensure it compiles on the CSE machines using the following sample makefile

```
all: sortTester

sortTester : sortTester.o BucketSort.o
        g++ -std=c++14 -Wall -Werror -O2 -pthread -o sortTester sortTester.o BucketSort.o

sortTester.o: sortTester.cpp BucketSort.h
        g++ -std=c++14 -Wall -Werror -O2 -pthread -c sortTester.cpp

BucketSort.o : BucketSort.h BucketSort.cpp
        g++ -std=c++14 -Wall -Werror -O2 -pthread -c BucketSort.cpp
```

```
clean:
	rm *.o sortTester
```

To compile your code type:

```
make
```

To run your code type:

```
./sortTester
```

You should create your own test cases to check the full functionality of your code against the specifications.

If you like, you may use the benchmarking suite provided to compare performance with your previous result, or with other students (though if comparing with other students, run it on wagner)

Marking

Your submission will be given a mark out of 100 with 80% an automarked performance component for output correctness and 20% manually marked component for code style and quality.

As this is a third-year course we expect that your code will be well formatted, documented and structured. We also expect that you will use standard formatting and naming conventions.

A number of test cases will be used to mark your solution. To pass a test case, your solution must produce exactly the same output as the reference solution. The results from both will be compared by using the linux tool diff.

In a number of test cases, your solution will be tested under a time limit, which is set according to the reference solution that was quickly written with no optimisations being applied.

Submission Instructions

Copy your code to your CSE account and make sure it compiles without any errors or warnings. Then run your test cases. If all is well then submit using the command:

```
give cs6771 ass5 BucketSort.h BucketSort.cpp
```

If you submit and later decide to submit an even better version, go ahead and submit a second (or third, or seventh) time; we'll only mark the last one. Be sure to give yourself more than enough time before the deadline to submit.

Late submissions will be penalised unless you have legitimate reasons for an extension which is arranged before the due date. Any submission after the due date will attract a reduction of 20% per day to your individual mark. A day is defined as a 24-hour day and includes weekends and holidays. No submissions will be accepted more than three days after the due date.

Plagiarism constitutes serious academic misconduct and will not be tolerated.

Further details about lateness and plagiarism can be found in the Course Introduction.

Acknowledgement
Inspiration for this assignment is based on a 2009 Massey University assignment written by Dr Martin Johnson.