Joseph Levesque
106907881
AMS 595

Multithreading, Inheritance, and Polymorphism in C++

Throughout the course, the focus has been on the introduction of three programming languages of particular use in mathematics, as well as various computer science concepts which enhance one's ability to utilize those languages. Thus, since the course focuses on the use of the programming languages rather than specific applications to mathematics, I chose a project which would increase my fluency in the most versatile of the three languages introduced this semester, C++. In particular, my intent for the project was to design and implement an object oriented class structure which would permit the comparison of execution time benchmarking between various levels of multithreading in Bucket Sort.

The class structure that I designed is illustrated in the elucidated UML class diagram shown in figure 1. While some dependencies were omitted in order to reduce clutter (e.g., vector.h, which would require a dependency arrow to every other custom class box), the diagram gives a fairly comprehensive overview of the underlying intricacies to the class structure. In particular, the virtual class AbstractSort was designed to be a general-purpose superclass, extensible to any sorting algorithm in order to create a subclass which would benchmark the execution time of that algorithm. This permitted the reuse of the accessor and mutator methods, as well as the class method isSortedAscending(), across all child classes. In addition, it ensured that any future implementations of sorting algorithm benchmarking classes would follow the same syntax with respect to member variables and methods.

Also of note is the Bucket class, which acts as a wrapper class for the vector of numeric values to be stored in an individual bucket. While the two member variables could have been simply stored in a pair, the definition of a class permitted the convenient grouping of various methods associated with bucket manipulation. For instance, the sortBucket() method—an implementation of Insertion Sort used to sort an individual bucket—fits better here than it would in the BucketSort class. The member method addToBucket(), together with the private nature of the member variables vector<int> bucketElements and int numElements, additionally prevents users from adding elements to the vector without incrementing the counter, which would not have been a possible restriction had a pair been used instead of a custom object class.

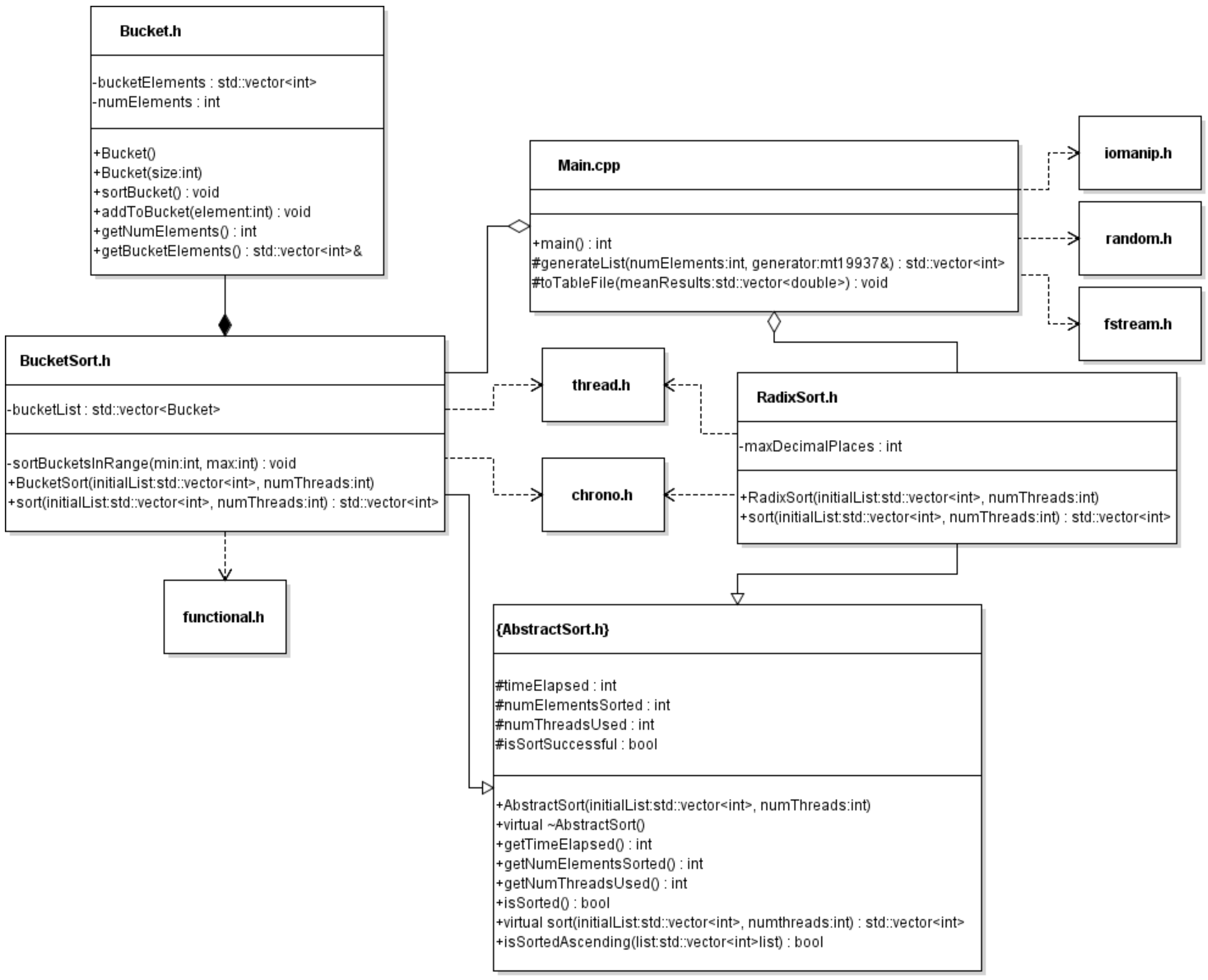
Bucket.h
-bucketElements : std::vector<int>
-numElements : int
+Bucket()
+Bucket(size:int)
+sortBucket() : void
+addToBucket(element:int) : void
+getNumElements() : int
+getBucketElements() : std::vector<int>&
Main.cpp
+main() : int
#generateList(numElements:int, generator:mt19937&) : std::vector<int>
#toTableFile(meanResults:std::vector<double>) : void
iomanip.h
random.h
fstream.h
BucketSort.h
-bucketList : std::vector<Bucket>
-sortBucketsInRange(min:int, max:int) : void
+BucketSort(initialList:std::vector<int>, numThreads:int)
+sort(initialList:std::vector<int>, numThreads:int) : std::vector<int>
thread.h
chrono.h
RadixSort.h
-maxDecimalPlaces : int
+RadixSort(initialList:std::vector<int>, numThreads:int)
+sort(initialList:std::vector<int>, numThreads:int) : std::vector<int>
functional.h
{AbstractSort.h}
#timeElapsed : int
#numElementsSorted : int
#numThreadsUsed : int
#isSortSuccessful : bool
+AbstractSort(initialList:std::vector<int>, numThreads:int)
+virtual ~AbstractSort()
+getTimeElapsed() : int
+getNumElementsSorted() : int
+getNumThreadsUsed() : int
+isSorted() : bool
+virtual sort(initialList:std::vector<int>, numthreads:int) : std::vector<int>
+isSortedAscending(list:std::vector<int>list) : bool

Figure 1: UML Class Diagram

The last class of note, Main.cpp, is the sole file without an accompanying header file. In addition to containing the main() method, this class contains helper methods which permit the analysis of vectors of AbstractSort objects. The generateList(int numElements, mt19937& generator) method utilizes the Mersenne twister engine in order to generate a list of integers from the Uniform(0, numElements) distribution, which is optimal for ensuring Bucket Sort runs in O(n) time. File I/O support is provided by the method toTableFile(), which takes as an argument a vector of results in milliseconds for 0, 2, and 4 threads and appends them as a new row of the table in the "results_table.txt" file.

In order to increase code readability and clarity, full Javadoc-style comments were used for all methods. This not only assists other people in reading the code, but also provides documentation which various IDEs may use to provide in-client support for the custom classes. It is particularly useful to be able to see the full documentation of a custom method simply by hovering over a line of code which calls it with the cursor, or to quickly see the correct argument order when typing out a method call. Portability and modularity is also increased, since code with full documentation is far simpler to utilize in future applications than undocumented code.

Version control was also used for this project. Periodic uploads to git with detailed commit comments provided not only a back-up in case my hard drive spontaneously combusted, but also a means by which to roll back any changes which cause catastrophic failure. While no rollbacks were required for this project, it was certainly a boon to have the option. In addition, the use of git provides an avenue by which collaborators could contribute to the project via pull/merge/commit requests, had there been more than one person in this group.

In regard to the implementation of the multithreading, I utilized the new C++11 native multithreading support found in thread.h. Due to the list to be sorted being drawn from a uniform distribution, relatively even distribution of elements across the buckets can be assumed. Thus, to divide the buckets into two (or four) groups is to divide the elements into two (or four) roughly evenly sized subsets. Hence, an efficient and effective way to divide the work among the n threads is to simply distribute the elements into the buckets, and then assign the i^{th} thread to process the i/n^{th} subset of buckets. This prevents any concurrent access to variables, and thus eliminates the need for mutual exclusion objects to lock assets.

Since the focus of this project was on the implementation—that is, learning how to use object oriented design and multithreading in CPP—and not on the actual analysis of Bucket Sort, the "conclusions" of this project are many and varied, and to be found on almost every one of the 500 lines of code written for the project. To enumerate them in two pages would be an effort in futility (and would essentially amount to C++ documentation in very small font). However, the successful working implementations of virtual classes, inheritance, polymorphism, file I/O, chrono benchmarking, multithreading, and the sorting algorithms themselves stand testament to the validity of those "conclusions".

That said, the performance of the multithreading implementation warrants further discourse. From the results recorded in results_table.txt, wherein each row contains the mean execution time over 100 trials on a single initial list for zero, two, and four threads, it is readily apparent that the use of four threads reduces runtime by approximately 40% on a sample size of 2^{16}. This was as expected, since 2^{16} is sufficiently large to negate the majority of the overhead incurred by running and managing additional threads, and the actual concurrency of all four threads was unlikely due to hardware restrictions (quad core with hyperthreading has 8 available threads for all system processes, and I didn't run the program from MS-DOS or any similar minimal interface, so it is probable that less than four threads were available at various times throughout the execution of the sorting algorithm).

Threads Runtime (ms)	2 Threads Runtime (ms)	4 Threads Runtime (ms)
19274.7	14038.1	11234
19274.2	14417.9	10305.1
20182.3	13287.4	10693.7
18792.6	14056.3	9022.05
20468.4	13871.5	13080.7
21005.3	14095.2	12187
21832.8	13498.7	10558.2

Figure 2: Output from results_table.txt

Surprisingly, this increase in efficiency exceeds by far the amount that I found when reviewing the literature. In their thesis, "Performance Analysis of Multithreaded Sorting Algorithms", Jouper and Nordin recorded a mere 12.5% decrease in runtime for Bucket Sort utilizing 4 threads on a sample size of 2^{16}. While this may be due to more efficient use of multithreading in my implementation, it is not necessarily the case that this the cause. Indeed, the disparity is likely due to some combination of increased efficiency of implementation as well as the language's thread library, since Jouper and Nordin used Java whereas I used the (usually more efficient) CPP. Hardware differences are unlikely to have played a role, as while they would assuredly effect runtime, they are apt unto equally effecting runtime regardless of the number of threads used, provided that there are sufficiently many free threads for the program to make use of.

Joseph Levesque
106907881
AMS 595

References

Jouper, Kevin and Nordin, Henrik. *Performance Analysis of Multithreaded Sorting Algorithms*. Thesis. Blekinge Institute of Technology, Sweden. Web. 12/12/2018.

While not used for this report, the C++ documentation at the following websites was used extensively while writing the code for the project:

- http://www.cplusplus.com/reference/
- https://en.cppreference.com/w/

The following links were quite useful as a primer for basic multithreading in C++:

- https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/
- https://solarianprogrammer.com/2012/02/27/cpp-11-thread-tutorial-part-2/

Lastly, https://stackoverflow.com was quite useful—as always—in answering the weird little questions that come up while programming which the documentation doesn't suffice for.

The Eclipse CPP IDE with the MinGW64 compiler was used for writing and testing code.

Violet UML Editor was used to create the UML diagram provided as figure 1.

Atlassian Bitbucket (Git) was used for version control.