Implementation of histogramming techniques in a parallel software environment

Orlando Garcia Feal
rodland@gmail.com
PH-SFT Group
Supervisor: Benedikt Hegner

August 23, 2013

Abstract

The aim of this document is to explain the procedures followed to develop a concurrent histogramming solution for parallel scientific software. The motivations, software development methodology, technologies and evaluation techniques used will be detailed in the following sections.

1 Introduction

Traditionally, histogramming was considered a sequential task. However since mid 2000 decade, the CPUs reached a practical limit in their clock frequencies so the manufacturers started to integrate several cores in the same chip [10][11]. Thereafter the performance upgrade obtained in the current software by simply switch the CPU for another faster was almost lost. It could be said that it was the beginning of a new *software crisis* era [12]. In order to take advantage of the brand new CPUs computing-power, all the software should be re-engineered to use parallelism. However while in some algorithms the parallelism is obvious in others it's still an object of study.

There's two motivations to make concurrent histogramming, the first of them and is to obtain a speed up in the histogram filling. The second one and probably the most important, is obtain thread-safe histograms that could be used in parallel software environments. This should lead us to have histogramming where before was not possible and be able to perform tasks that were not possible without a thread-safe implementation.

2 Technologies used

For the development of this project many technologies where used, some of the most important were the following ones:

C++11 the new C++ standard provides new memory model and concurrent programming facilities that were essential for the development of this project.

CPPUnit this popular unit testing library was used as a part of the Test-Driven development process [5].

Git was used as Source Code Management (SCM) system. It was an important part of the project to keep a track of the modifications, manage different branches and distribute the code to different users and machines [3].

igProf this profiler was used basically to measure the performance, try to identify problems and corroborate conclusions obtained from benchmarks [7].

CMake is the build system chosen for this project [4].

GDB the GNU Debugger was essential to debug the unit tests as well as the problems found with them [6].

3 Development methodology

The software development methodology chosen for this project was the Test-Driven Development

(TDD). The main reason to choose this methodology was the need of a solid testing suite to check the correctness of multiple histogram implementations of the same interface in sequential environments as well as parallel ones.

4 Design and development

In the first stage of the development a common interface to a generic histogram was designed. All the further tests and implementations were developed using the same interface. This interface was intentionally limited to one-dimension histograms as they already present all the problem of concurrent access.

The next step was the design and development of proper unit tests. This tests could be divided into two types. The first of them are the sequential tests. These tests are used to check that the histogram has the expected behaviour when used in a single-thread environment. For this, every function is checked and then, the histogram state is verified.

The second type of tests tries to check the validity of the results produced by the histogram when it is filled in a concurrent way by several threads. These tests compare the obtained results with those obtained by a histogram filled in a sequential way.

The multiple implementations developed will be explained in the following sections.

Note that in this first implementations, a worst-case in which each bin can have any arbitrary size was considered. Therefore, the process of finding the bin to be incremented is not constant-time as if the bins were fix-sized. More specifically, a binary search algorithm was used.

4.1 SeqHistogram

The SeqHistogram class was the first implementation. It was made as a non-thread safe reference. The main purposes of this class are to check that the concurrent tests work correctly (it should fail when run these tests), compare the results with more complex histograms and check how much faster or slower is a concurrent histogram in comparison with a non-thread safe one.

4.2 LockHistogram

The LockHistogram class was developed as simple approach to a concurrent histogram based on locks. More precisely, it uses a *std::recursive_mutex* to protect the data, the differences with a normal mutex is that this one can be acquired by many functions but these must be called from the same thread. On the other hand, to acquire the lock it is used a *std::lock_guard* function that automatically releases the mutex when the function is over, even if an exception is thrown.

4.3 AtomicHistogram

This implementation is based on atomic data types provided by the new standard C++11. These types ensure the atomicity of certain operations. If this atomicity is provided by some kind of internal lock or not depends on the platform in which is executed. However C++11 only provides atomic arithmetic with integer data types, meanwhile the content of the bins of a histogram are usually floating point values. For this case C++11 still provides some atomic operations like assignment and *std::atomic_compare_exchange_weak*. With this last one, it is possible to load the value, perform some arithmetic, and if the original value hasn't changed in this period of time, exchange it for the new value. With this kind of operation some *collisions* are expected when two or more threads tries to modify the same bin at the same time, and consequently, some of them should start the entire process again.

5 Benchmarking

Once the different implementations were checked for their correctness using the unit test setup, their performance was compared. For that, a certain number of uniformly distributed random values was generated. And then, the task of filling a histogram is divided between 1 or more threads. This procedure is repeated with different sized histograms (number of bins) to measure the impact of this factor in the global performance.

Unless otherwise stated, all the benchmarks were done in a desktop computer with Scientific Linux 6 operating system and a 4-core AMD Phenom (code name Agena) processor at 2300Mhz . In all cases the same compiler version was used (GCC 4.8).

5.1 Single thread filling

In a first step, we compared the purely sequential performance of the different histogram implementations. As we can see in Figure 1 on page 4 the filling times strongly depend on the search-space given by the histogram size. Note that the non-thread safe implementation is the faster one and the concurrent implementations have a bit of overhead in comparison (over 13% for the *AtomicHistogram* and 20% for the *LockHistogram*). In other way we can also see that the *LockHistogram* is a bit slower than the *AtomicHistogram*. The overhead acquiring and releasing a lock causes a very significant overhead compared to using atomics.

5.2 Multithreaded filling

In a second step we measured the filling time with two threads and compared this time with the time spent with just one thread. The results can be seen in Figure 2 on page 5. We can easily spot that the *LockHistogram* solution is slower with two threads than with just a single thread. One of the main reasons is the lock contention, the *LockHistogram* uses just a lock to protect the critical data, the threads have to wait very frequently for the lock release. Even worse, due to the fact that the different threads (assigned to different cores) have to read and modify the lock, the caches of each CPU core have to be synchronised very often. Note that this process of cache coherence [13] is highly inefficient.

A different implementation utilising a fine-grained lock approach could be used, but even though the process of lock and release a lock is slower than the atomic operations.

Now we will analyse the results of *AtomicHistogram* in more detail. For this, we can take a look to Figure 3 on page 5. In this figure, the *LockHistogram* data was removed and data for three and four threads was added. As we can see, there is a small speed-up when using many threads with *AtomicHistogram* but only when dealing with large histograms. In contrast to single-threaded filling, the smaller the histogram, the slower filling speed is. The main cause for this is, once again, the data contention. As we get smaller histograms, different threads have to read and write the same data. Therefore we get from one hand the problem of the *std::atomic_compare_exchange_weak* collisions and from the other the problem of cache-coherence [13] protocols.

5.3 Fixed-bins Histogram

Up to now, we were using arbitrary-sized bins, so the searching times were rather large and non-constant. In order to verify how much influence this process has into the filling process, a new atomic histogram class was developed supporting fixed bins with constant search times. In the Figure 4 on page 6 the results with 1 and 4 threads for variable and fixed bin size are compared. Now the time spent by the fixed bin histogram with just one thread is (almost) constant with the number of bins due to the constant search times. Furthermore we can also see that there is still some speed improvements with multithreading even though the speed-up is not as high as in the variable-bin histograms.

5.4 Scaling tests

The benchmarks in this section were tested on a different machine, a 48-core at 2200Mhz in 4 sockets AMD Magny-Cours server running CentOS 5.

The purpose of the first benchmark is to test how well scales the *AtomicHistogram* with a high number of cores. The results can be seen in the Figure 5 on page 5. In this machine, the filling speed for two threads is clearly slower than with one thread. This phenomenon is more pronounced in some platforms than in others. It seems to be stronger in older and multiple socket machines, and is probably caused by some cache-coherence [13] protocols enabled in multithreaded applications that provoke this performance penalty. On the other hand we can see that even if the speedup is smaller as we increase the number of threads, we never reach a point in which the total filling time spent is higher than with less threads (except the previously mentioned 2-thread case).

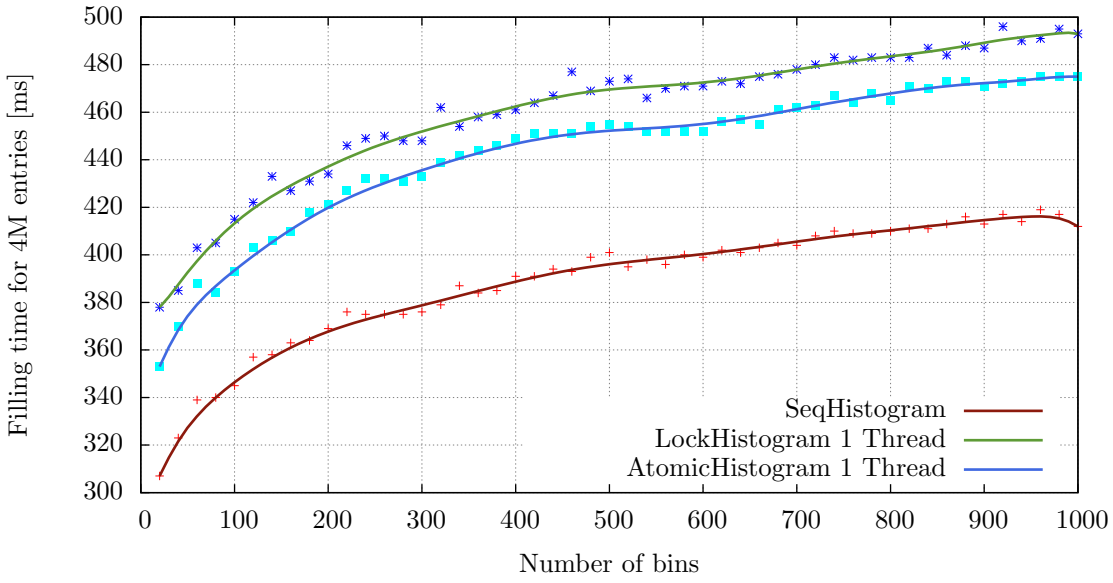

Figure 1: Comparison of multiple histogram classes in purely sequential use-case.

Note that in this case a slightly different implementation of the *AtomicHistogram* was used, with integer bin contents and fixed bin size.

In the next benchmark we tried to measure the impact of using multithreaded histogramming in multiple CPU sockets. At least in theory, using cores from CPUs in different sockets should result in a performance penalty due to the smaller bandwidth of the link between processors (e.g. Intel QuickPath or AMD HyperTransport). In this case, we used the *taskset* command from the util-linux package to distribute the different threads into specific cores belonging to specific sockets. Twelve threads where used, distributed from one to four sockets. The results presented in the Figure 6 on page 7 confirm the expected results. As we distribute the threads into more different sockets, the task becomes slower.

6 More complex histograms

In order to improve the performance of the previous implementations we started to consider new approaches. One of them was the lazy binning. This was made to obtain filling times independent from the histogram size and to try to increase the granularity of the operations over the histogram data structure.

In this implementation, basically, the filling methods insert the values into a temporary buffer. When a thread fills completely a buffer, this thread should *collapse* the buffer contents into the histogram. In order to avoid waiting times for the other threads a second buffer is used when the first one is being dumped.

In Figure 7 on page 7 the results for two preliminary implementations of this solution compared with the previous ones are shown. Even though it offers better performance than the *LockHistogram* it is still very poor in comparison with the *AtomicHistogram*. The profiling data obtained with igProf showed that the current implementation suffers from many fine-grained memory allocations. So based in this data can be concluded that there could be still enough room for a 50% of improvement.

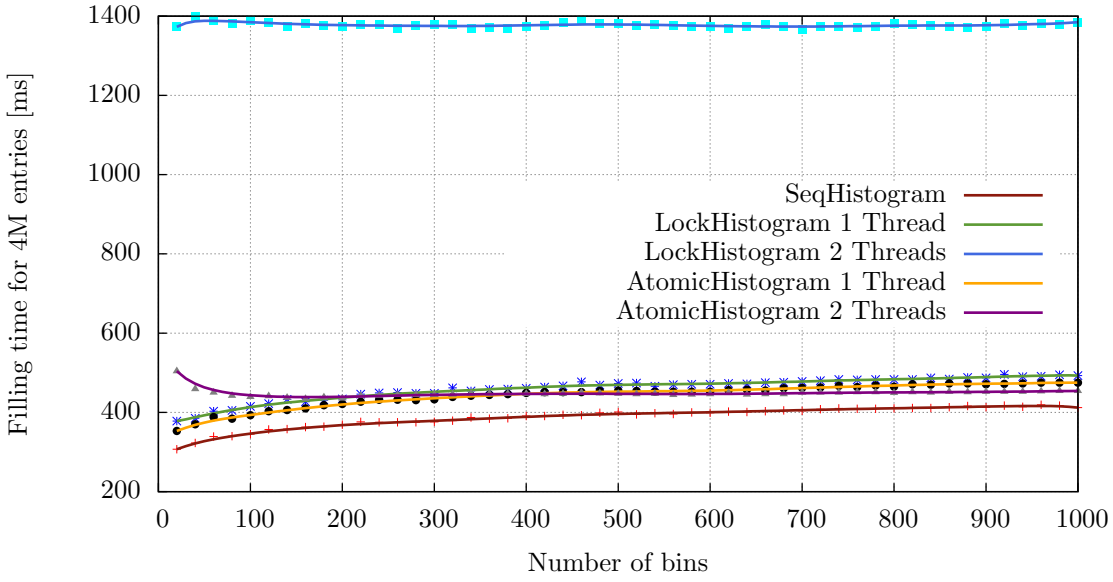

Figure 2: Comparison of multiple histogram classes both in sequential and concurrent use-cases.

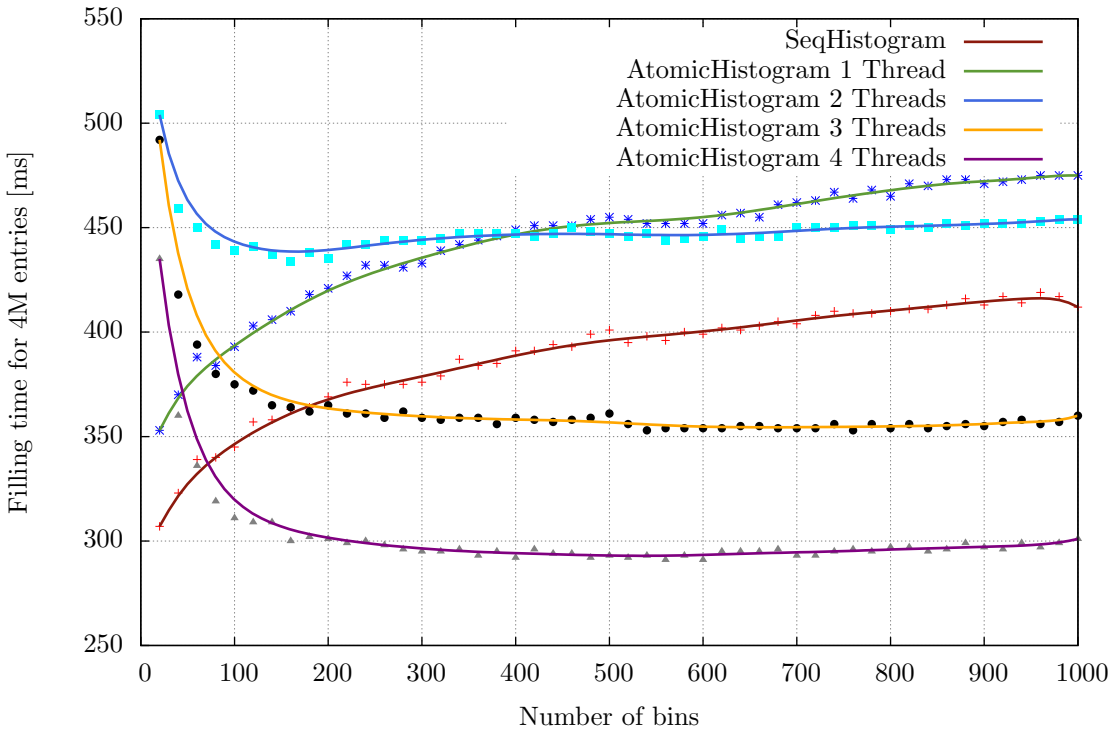

Figure 3: Detailed behaviour of atomic histogram working with one to four threads.

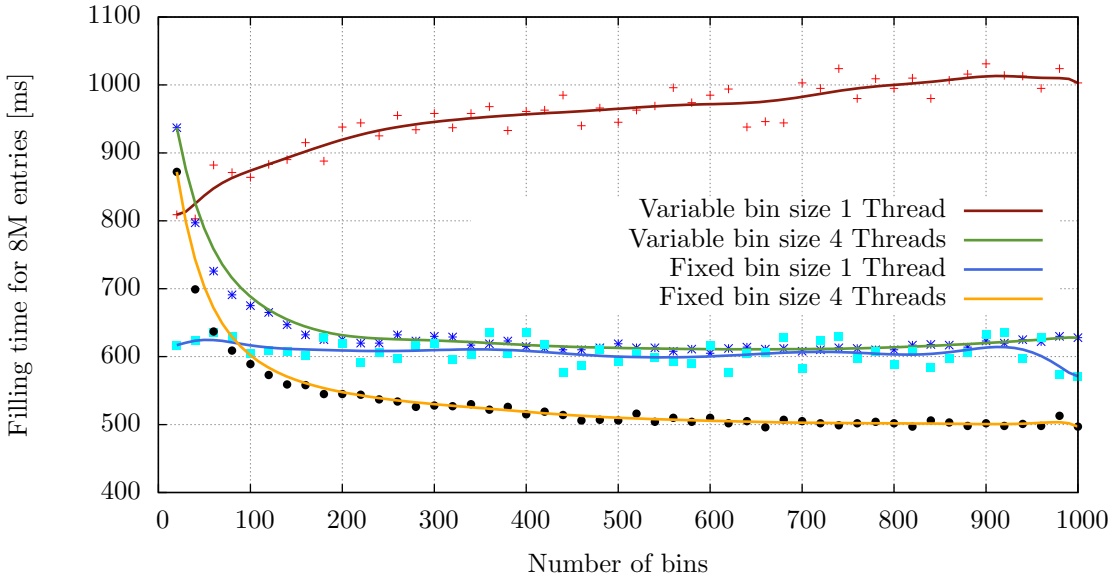

Figure 4: Comparison of the atomic histogram using fixed and non-fixed binning.

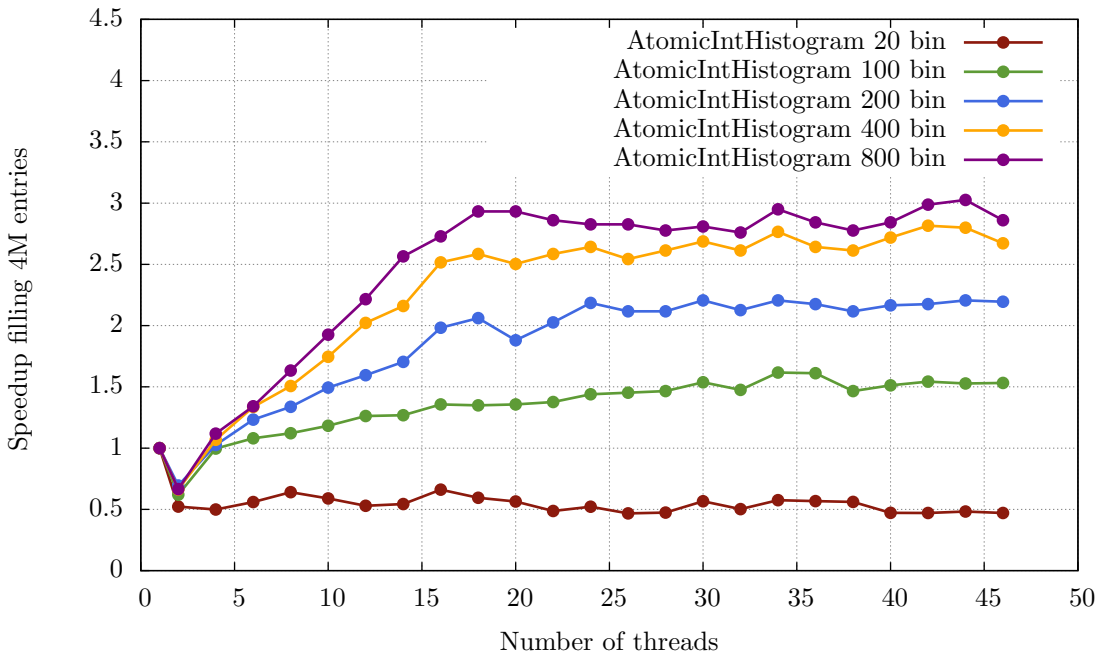

Figure 5: Speedup obtained working with a high amount of threads and different number of bins.

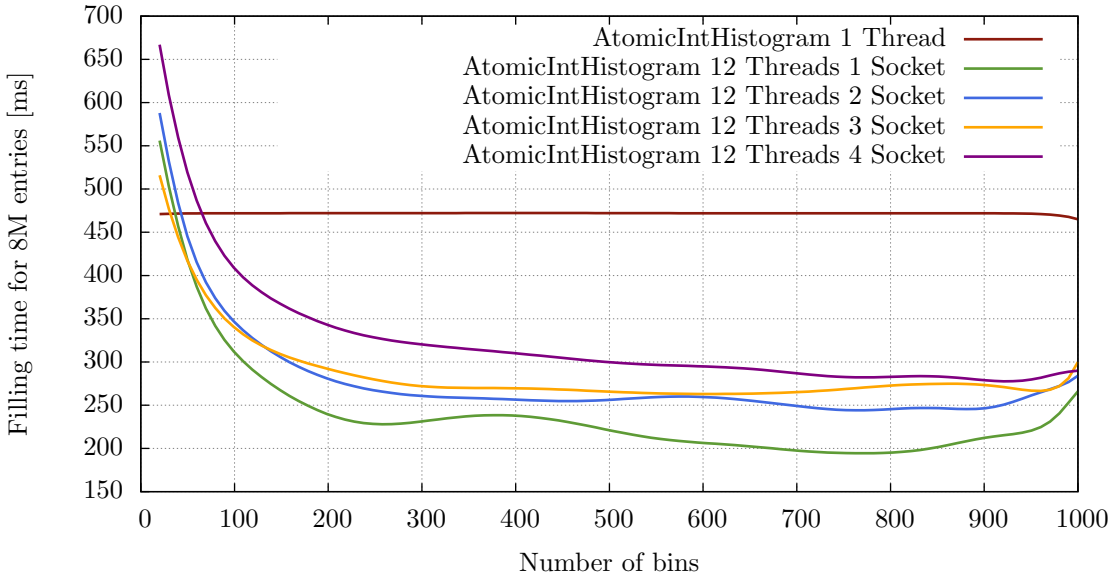

Figure 6: Performance differences filling a histogram from the same or different sockets.

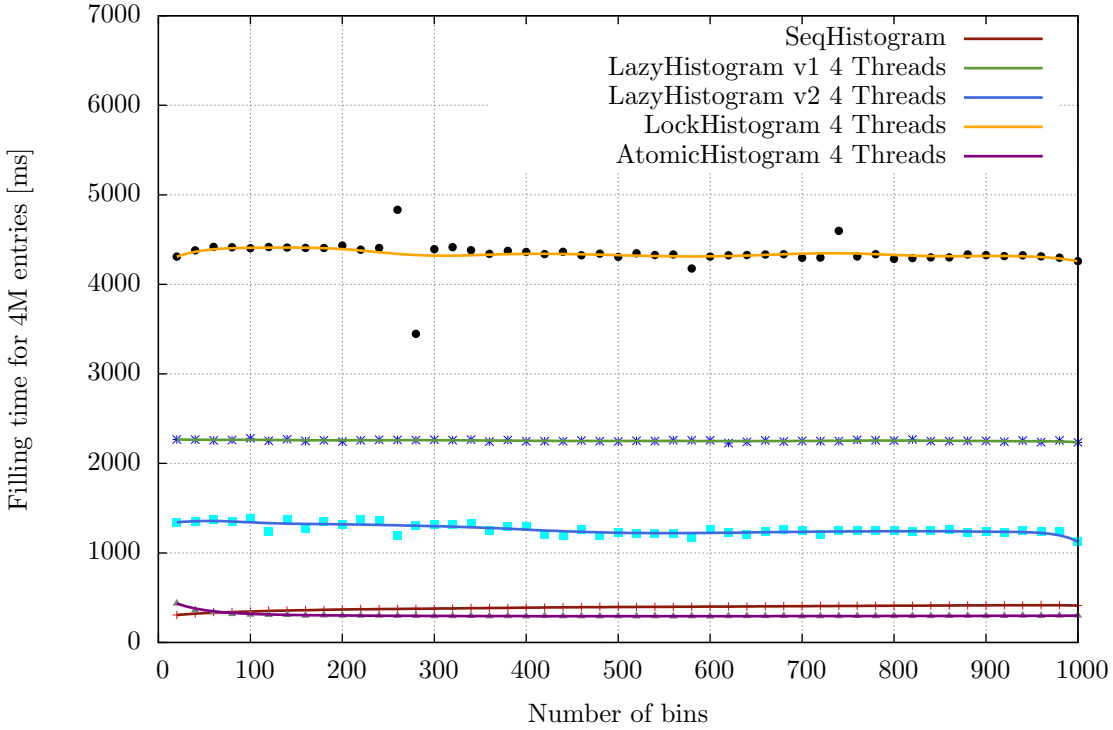

Figure 7: Comparison of LazyHistogram with the previous implementations.

7 Gaudi integration

Until now we were measuring just the histogram-filling speed. Despite that this is an important characteristic of the histogram implementation, this is not the typical scenario in which is run. Usually the working threads are performing also other tasks along with the histogram filling. Hence we are looking for the least impact possible in the working threads when we require concurrent histogramming.

The next step of the project was to integrate the existing concurrent histogram classes into GaudiHive. The current histogram service of Gaudi only provides non-thread safe histograms, therefore it is necessary to modify the histogram service in order to provide methods to book concurrent histograms.

To benchmark the new histogram classes, a test case was configured in which every Gaudi event is processed by three algorithms and each of them just fills a histogram with three million of random values. The test case process 1000 events and a maximum of 20 events and 10 algorithms running in parallel. Note that the algorithm cloning was enabled. In this case the histograms tested had 10 and 200 bins, and all algorithms from all events fill the same histogram instance.

The results can be seen in Figure 8 on page 9, the behaviour is similar as we have seen previously, the *LockHistogram* becomes slower as we increased the number of threads and the *AtomicHistogram* is just slightly faster. Nevertheless the performance penalty suffered by the *AtomicHistogram* in the previous tests with small histograms and multiple threads is not so strong as before.

The next test case will consist in mixing histogram-filling algorithms with *CPUCrunch* algorithms. This algorithm was designed to simulate the CPU stress produced by the real workload. This test is run in the same configuration as before but adding four *CPUCrunch* algorithms interleaved with the histogram filling algorithms.

In the Figure 9 on page 9 we can see some interesting results. The overhead added by the histogramming is lesser than the time spent by just do the histogram filling, moreover, the *LockHistogram* does not perform as bad as in the previous cases. It seems that the data contention problems seen in the previous tests are not as significant when histogramming is mixed with another tasks.

8 Conclusions

After all this project we can see as a conclusion that histogramming is a lightweight task that presents data contention problems in parallel environments. For all this, it is difficult to obtain a performance gain via parallelism but it is easy to obtain a significant performance loss in some cases. Therefore we should design the necessary data structures with care to avoid possible drawbacks.

On the other hand, we could say that the most successful implementation developed is the *AtomicHistogram*. Despite its simplicity, it was a solid performer in all the setups tested. Even in the worst case, working with small histograms, it performed better than the other solutions. And even though the non-thread safe implementation is faster in single-thread filling, the *AtomicHistogram* is just a 13% slower in the tested platform.

9 Further work

After eight weeks of work in this project as a summer student, there is still work and research that could be done in this project. There is still many solutions that could be tested, like improve the *LazyHistogram* approach, implement a fine-grained lock solution, test the transactional memory support of the new compilers... It will be also interesting continue with the study of the behaviour of the concurrent histogramming under real-world cases with Concurrent Gaudi.

References

[1] Anthony Williams, *C++ Concurrency in action: Practical Multithreading*. Manning Publications, 2012.

[2] C++ Reference: `http://cppreference.com`

[3] Git Source Control System: `http://git-scm.com/`

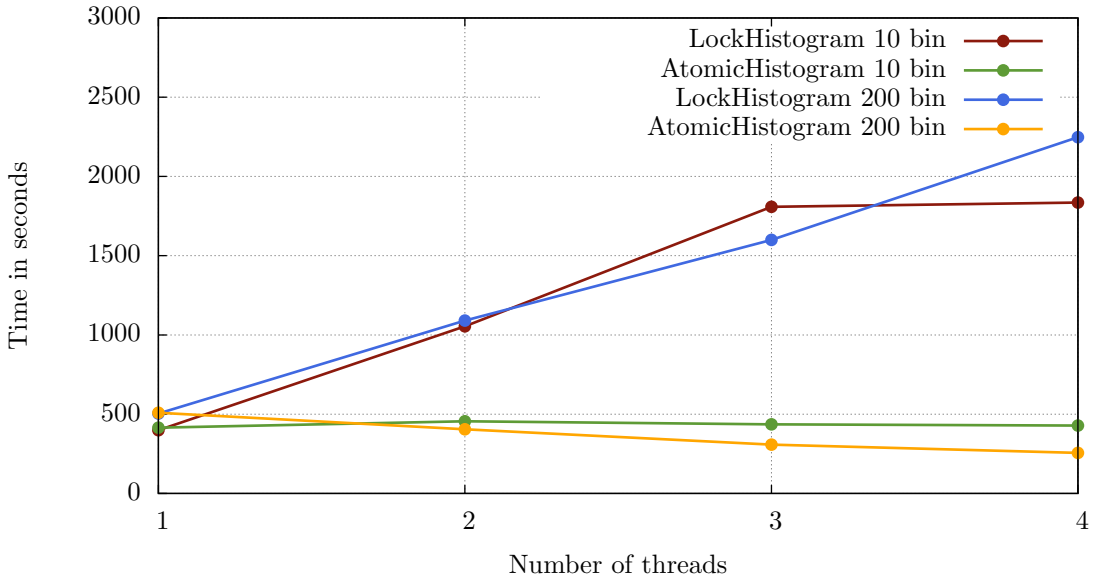

Figure 8: Runtime of GaudiHive test workflow when adding histogram usage.

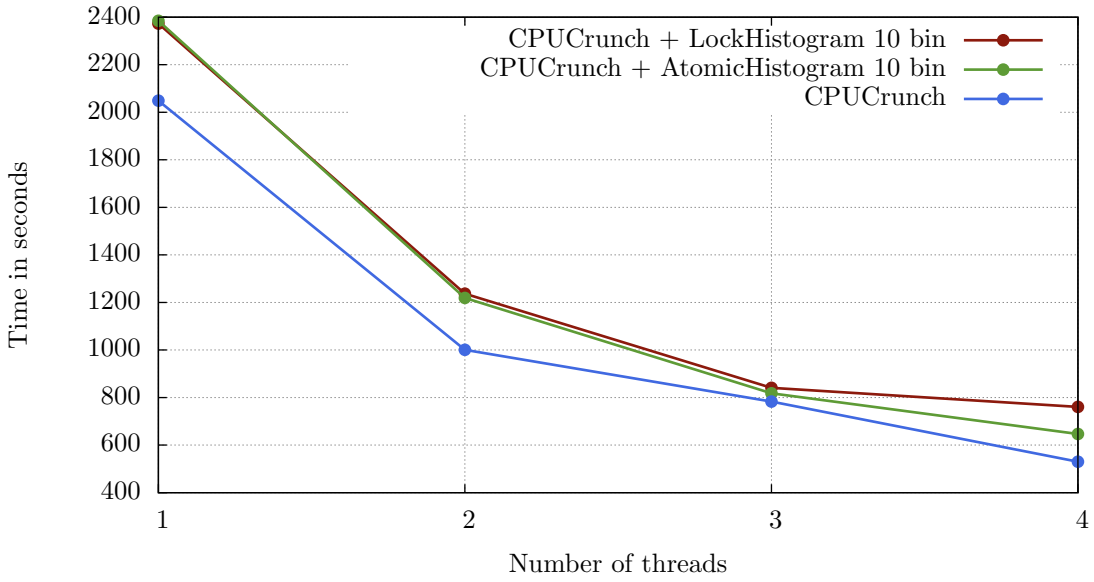

Figure 9: Runtime of GaudiHive test workflow when adding histogram usage along with CPUCrunch.

[4] CMake Build System: http://www.cmake.org/

[5] CPPUnit 1.11.6 Documentation: http://cppunit.sourceforge.net/doc/1.11.6/

[6] The GNU Project Debugger: https://www.gnu.org/software/gdb/

[7] igProf Documentation: http://igprof.org/

[8] C++11 Concurrency tutorial: http://www.baptiste-wicht.com/series/cpp11-concurrency-tutorial

[9] Concurrent Gaudi: https://twiki.cern.ch/twiki/bin/view/C4Hep/WebHome

[10] Herb Sutter, *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. http://www.gotw.ca/publications/concurrency-ddj.htm 2005.

[11] Julio Diez Ruiz, *Overcoming the embedded CPU performance wall*. http://www.embedded.com/design/mcus-processors-and-socs/4405280/1/Overcoming-the-embedded-CPU-performance-wall- 2013.

[12] Saman Amarasinghe, *The Looming Software Crisis due to the Multicore Menace* http://groups.csail.mit.edu/commit/papers/06/MulticoreMenace.pdf

[13] David Kanter, *An Introduction to Multiprocessor Systems*. http://www.realworldtech.com/coherency/ 2006.