Advanced Multiprocessor programming
Group Project

Nick Mayerhofer (0726179), Konrad Pozniak (0925996)

July 1, 2015

1 Project Description

In this project we are going to implement four different list-based sets and try to find out which one has the best performance properties, in a highly parallel program. This includes testing the implementations for correctness and creating our own meaningful benchmarks.

We assume the reader is familiar with the lecture [5].

1.1 Programming language and environment

For this project, we use C++ 11 and its native thread implementation. Our benchmark tests are run on two shared memory multicore machines of Vienna university of technologies parallel computing research group:

- Mars
  8 Intel Xeon E7-8850 CPUs with 10 Cores @ 2Ghz
  160 Threads
  1 TiB RAM
  Debian GNU/Linux 3.14-1-amd64
- Ceres
  4 SPARC T5 CPUs with 16 Cores @ 3.6Ghz
  512 Threads
  1 TiB RAM
  Oracle Solaris 11

All test and benchmark programms are compiled with the GNU C++ compiler `g++` (version 4.9.2 on Mars and 4.8.2 on Ceres) and full optimization (`-O3`) for 64 bit (`-m64`) with C++11 standard (`-std=c++11`).

2 The Set data type

A set is a data type that can contain certain values, without any particular order, and no repeated values. It is a computer implementation of the mathematical concept of a finite set. Unlike most other collection types, rather than retrieving a specific element from a set, one typically tests a value for membership in a set. [2]
In this project we will take a look at the three most important operations of a set:

- `bool add(long item)`
  Adds an item to the set.
  Returns true if successfull, false if the item already was in the set.
- `bool remove(long item)`
  Removes an item from the set.
  Returns true if successfull, false if the item was not in the set.
- `bool contains(long item)`
  Checks if an item is contained in a set.
  Returns true if the item is contained, false if not.

2.1 Reference Set

As reference implementation we used `std::set` from the C++ standard library and synchronized each call to the object with a global `std::mutex` lock, since it is not thread safe.

This construction is deadlock free, since there is only one global lock. `std::mutex` has no guarantees to be starvation free or fair, so this set also has no guarantees. The linearization point is trivially where the lock is acquired.

The c++11 implementation of `std::set` is based on a binary search tree, while our implementations will be based on simple linked lists. Thus it is interesting to see how much of a difference this makes. If the set is filled with many elements, the time needed to search for an element will certainly be noticeable. We also expect that this reference set will be much faster on a single thread than our implementations, but the global lock will probably slow it down very much in parallel mode.

Following are explanations of the four different implementations, introduced in the lecture [5]. After that we are going to explain how we are going to penetrate the implementation, in order to test them.

2.2 Fine-Grained Locking

Instead of a global lock this implementation has a lock for each individual node, s.t. multiple threads can operate at the same time on different locations of the list.

The set with fine grained locking is deadlock free, since locks are always acquired in the same order. But it has no progress guarantees. The linearization point with an item is in the set, is when the corresponding node is locked. Otherwise when the next higher node is locked.

Implementing this set was pretty straightforward, except we found a little mistake in the lecture slides, see 6.

2.3 Optimistic Synchronization

Optimistic synchronization does not lock any nodes when searching for an element. It locks just if element has been found, the searched element itself and its predecessor. This requires validation that the nodes are still linked into the list after they got locked. If not, the process has to be restarted - this might be a performance problem.

As with the fine grained locking set, this set has no progress guarantees. The operation linearize when the node or the next higher node is successfully validated.

2.4 Lazy Synchronization

Lazy synchronization does not acquire locks for a `contains` call, in contrast to optimistic synchronization. To counteract the possibility that `contains` returns a element that has been unlinked from the list, each node can be marked as removed with a flag. It is then logically removed from the set, while it may be still linked into the list. This leads to the problem, that nodes cannot be deleted safely, even after they have been unlinked - other threads might still be accessing them. See 5 for possible solutions to this.

We expect this implementation to have better performance than the previous ones, because `contains` does not need any locks and is wait free, and `validate` does not need to iterate through the whole list again.

2.5 Lock Free

The lock free implementation does not use any locks, but atomic operations with hardware support to update the pointers in the list. The marked flag and the pointer need to be treated as one atomic unit, though.

Pointer on AMD64 have 48 bit, but are aligned to 64 bit in memory [4, p. 120]. The same is true for the SPARC T5 architecture. It is possible to use one of the surplus bits as marked flag, so the flag and the pointer can be moved together in a single compare-and-swap operation.

This was quite tricky to implement, since dereferencing a pointer when the flag is set leds to an memory access violation. So before every pointer dereferentiation the flag has to be removed, but it must not be forgotten when linking nodes in the list.

We used the GCC builtin `bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...)`. This builtin directly translates to a single assembler instruction and should be the fastest compare and swap available. Of course it is still is a memory fence and can be very costly.

Since the marked flag is part of the pointer to the next node, one must be very careful when unlinking nodes in `remove` or `find`: Just swapping the pointers can override the flag of the predecessor node. See 6 for a detailed description.

The lock free set is completely lock free, `contains` is even wait free. We expect the lock free set to perform best in our benchmarks because of this properties of lock freenes.

3 Testing for correctness

To test our implementations, we designed some testcases to make sure they run correct.

The first test case lets all threads `insert` the same elements multiple times, then checks that for each elements `add` returned `true` only once, and that all elements are really in the set. Then all threads `remove` the same elements again, and for each element, `remove` must only return `true` once. After the tests, the set must be empty.

The second test case mixes the three operations and aims to find any bugs caused by side-effects between the operations.

With these testcases we were able to find mistakes in the example code from the lecture slides and correct them. See 6 for a detailed description.

All tests have been run correctly at least 100 times each on both testing machines, Saturn and Ceres, with all our set implementations and different number of elements and threads.

4 Benchmark

We tried to construct the Benchmark environment as generic as possible, to fulfill each of our following requirements:

- Encapsulate as much as possible in reusable code
- Be as precise as possible (e.g. give the threads themselves the ability to execute their time measurements to keep the overhead low)
- Benchmark each operation separately, namely: add, contains, remove
- Output the results to files to keep the possibility for log outputs alive

To make sure we use the most efficient time measurement, we tested all available functions for their performance, see figure 3. We then decided to use the standard C++ high precision clock, because its short run time and high precision. Even when calling it very often with many threads, we measured an overhead less than 50ns.

There are endless possibilities for benchmarks: All combinations of the five sets, three operations and two machines with different thread counts, iterations, datatypes, on empty or prefilled sets etc.

We created four benchmarks to measure how long a certain number of operations on a set takes, one for `add`, one for `remove`, one for `contains` and one for a combination of them. Basically every thread executes the same operations in a loop and we measured the overall time it took the threads to complete the operations.

The `add` benchmark starts with an empty set and then every set performs 1000 insert operations on the set simultaneously. The values inserted are evenly distributed over the input space to avoid that one thread inserts only big numbers and another only small ones.

The `remove` benchmark starts with an half filled set (every second number is in the set) and then every thread performs 1000 remove operations, distributed like in the `add` benchmark, so every second `remove` fails and the set is empty at the end.

The `contains` benchmark also starts with an half filled set and then every thread performs 1000 contain operations.

In the mixed benchmark, every thread calls `add`, `contains`, `remove` and `contains` again with its thread id, so the max number of elements in the set at any time is the number of threads.

We chose 1000 as the number of iterations, because then the running time was short enough to repeat the benchmarks many times, but long enough to get meaningful measurements.

With this benchmarks we want to get a good metric for the sets performance and try to prove our hypotheses from the above set descriptions.

The results for Mars are listed in figure 1 and for Ceres in 2. We visualized them in figures 4, 5, 6 and 7.

The most obvious result is, that the fine grained locking set performs very badly on the first three benchmarks. Apparently the overhead for locking each node is much higher than the performance gain in allowing multiple threads to operate on the set at the same time. We knew the fine grained locking set would have bad performance, but we were surprised how bad it really is.

The difference between the reference set's tree implementation and our list implementation becomes visible in: the benchmarks that operate on a filled list and the advantage of the reference set is huge. On the mixed benchmark, that operates on an almost empty set, our implementations are much faster on Mars, while on Ceres only the lock free set is faster.

The throughput of the lazy synchronization set is much higher than the one of the optimistic synchronization one, especially for remove. The improvement of not locking nodes in `contains` and not iterating through the whole list again in `validate` really shows.

On Mars, the lock free set performs noticeably better than the lazy synchronization set, except in the mixed test. On Ceres, the lock free set is slower in remove and add, and faster in contains and mixed. Probably a compare and swap operation is more expensive on a SPARC architecture than on a AMD64 one and a locking operation conversely. Note that `contains` in both sets does not use compare-and-swap or a lock.

To measure the fairness between the threads, we let each thread run for a certain amount of time and counted the number of operations performed per thread. Our findings are presented in figure 8 and 9.

Very interesting is the difference between the two machines. On Mars, the first threads are faster than the others, because the were started before them, but overall the distribution looks pretty fair for all sets. On Ceres the plots look different. Some are really unbalanced, others less. But it is safe to say that Mars is fairer than Ceres.

We think that this difference is either caused by the different architecture of the processor or the difference in thread scheduling by the operating system.

With this knowledge we think it would be very interesting to implement a lock free set over a tree data structure to beat the performance of the reference set even on big sets.

5 The problem with the memory leak

In the lazy synchronization and lock free sets, it is not possible to determine a point where unlinked nodes can be deleted savely, because it is always possible that other threads may still be accessing the node. This causes a memory leak. We have two ideas how this problem could be solved, but we had not enough times to try them:

- Using C++ smart pointer, e.g. `std::shared_ptr`. Smart pointer automatically keep track of their memory pointees and are releasing the memory with the last pointee going out of scope. They are very convenient, but - depending on the implementation - implemented with wither locks or atomic operations and we suspect they have a bad impact on performance.
- Having a global memory pool managing unused nodes and instead of creating new node objects every time, reusing the old ones. This does not really prevent a memory leak, but limits the memory usage to the maximum size the set ever had. It might be a problem if nodes get reused immediately.

6 Changes to the lecture slides

We found some implementations in the lecture slides, not working in practice:

The first one is in the find function of the fine grained locking set.

```
Window find (T item ) {
     Node* pred = head;
     Node* curr = pred->next;
     pred->lock();
     curr->lock();
     while (curr->item < item ) {
          pred->unlock ();
          pred = curr;
          curr = curr->next;
          curr->lock ();
     }
     return Window (pred, curr);
}
```

It is possible that a thread gets a invalid window back, because the pointer to the current node is acquired before the lock of the predecessor node. We fixed it like this:

```
Window find (T item ) {
     Node* pred = head;
     pred->lock();
     Node* curr = pred->next;
     [...]
}
```

Another problem is in the lock free set, when a node is unlinked from the list after it has been marked. If the predecessor node is also marked, its mark is overridden by the compare and swap operation because the mark is stored in the next pointer.

We fixed this by checking for the mark and setting it on the new pointer before the compare and swap if necessary:

```
if(isMarked(w.curr)) {
     next = mark(next);
}
__sync_bool_compare_and_swap(&(getPointer(w.pred)->next), w.curr, next);
```

Note: the compare_and_swap may be replaced through the C++11 compare and swap methods atomic::compare_exchange_weak [1]. Further research would

Another rare bug happened, when a node was removed from the lock free set by another thread just after `find` found it unmarked, but before it returned the node. This could cause `add` to insert an node after an removed one and return true, but the node would not be in the set. We fixed this problem by adding a check to `add`:

```
bool LockFreeSet::add(long item) {
     LfsNode *n = new LfsNode(item, nullptr);
     while (true) {
          LfsWindow w = find(item);
          if(isMarked(w.curr)) {
               continue;
          }
    [...]
```

7 Appendix

The source code of this project can be found at:
`https://github.com/nickma82/advanced_multiprocessor_prog`

	add	contains	remove	mixed
reference	418.53	231.50	470.04	272.85
fine grained lock.	17155.33	9360.36	8433.88	32.25
optimistic sync.	2609.88	418.71	3421.86	38.19
lazy sync.	1333.05	289.80	215.82	25.22
lock free	1128.28	161.50	115.61	29.39

Figure 1: Average time in milliseconds of 100 throughput benchmark runs on Mars, 80 threads, each thread ran the iteration 1000 times

	add	contains	remove	mixed
reference	29.75	25.17	25.77	27.94
fine grained lock.	5759.07	6059.24	5886.83	37.72
optimistic sync.	1348.86	634.34	2344.10	39.79
lazy sync.	635.12	328.49	307.21	30.92
lock free	687.51	320.21	358.25	16.03

Figure 2: Average time in milliseconds of 100 throughput benchmark runs on Ceres, 64 threads, each thread ran the iteration 1000 times

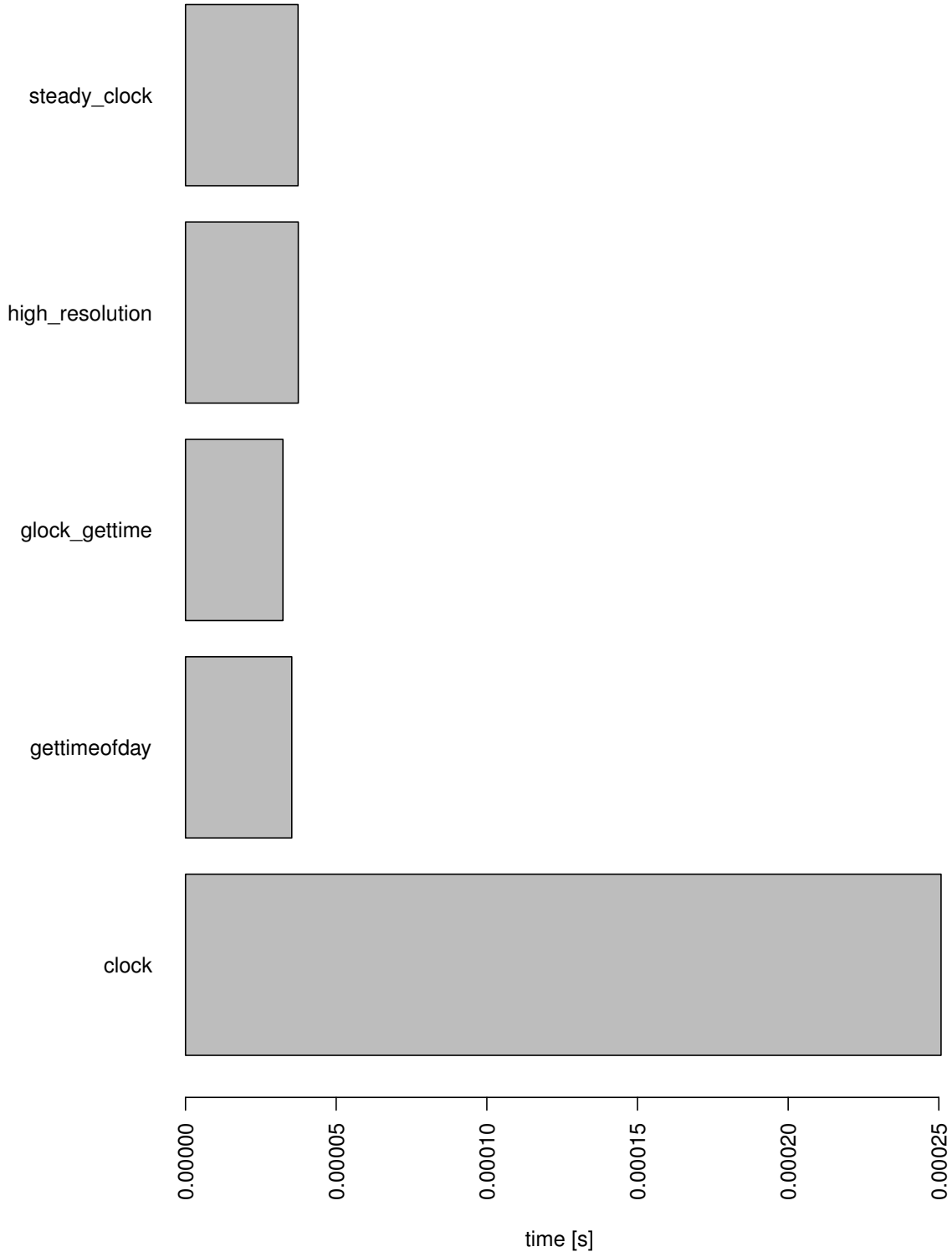

Figure 3: average execution time of 1000 calls to different time capture methods, measured with a wrapped *clock_gettime*

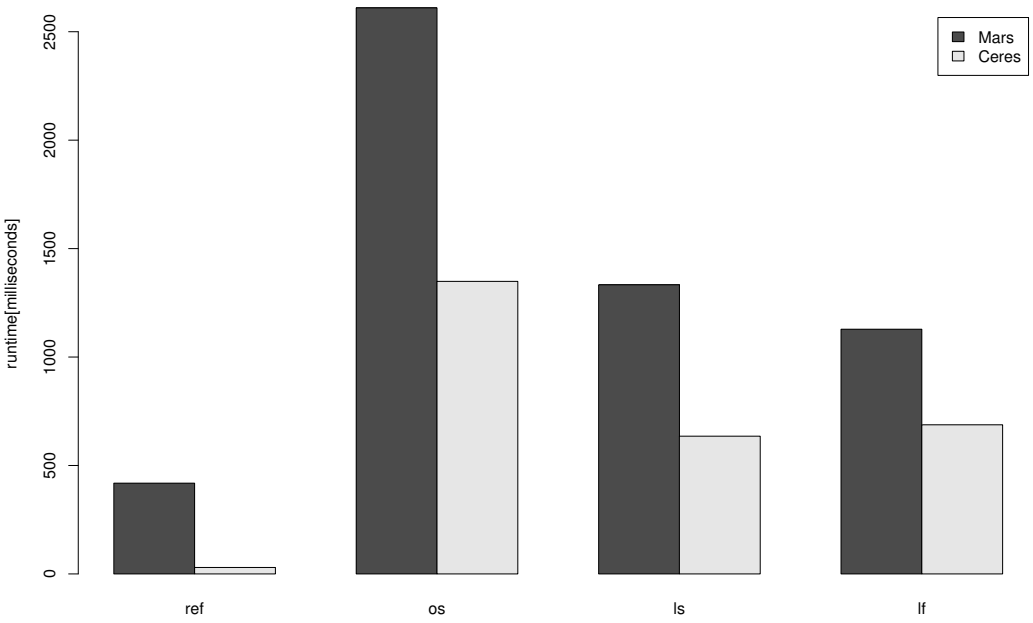

Figure 4: Comparison of `add` performance, on both machines

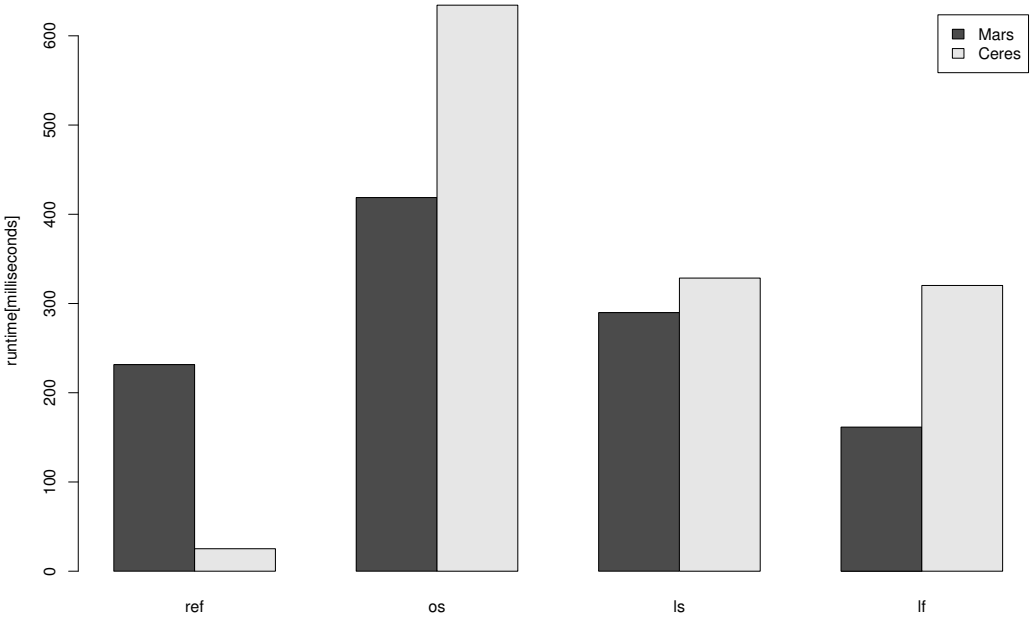

Figure 5: Comparison of `contains` performance, on both machines

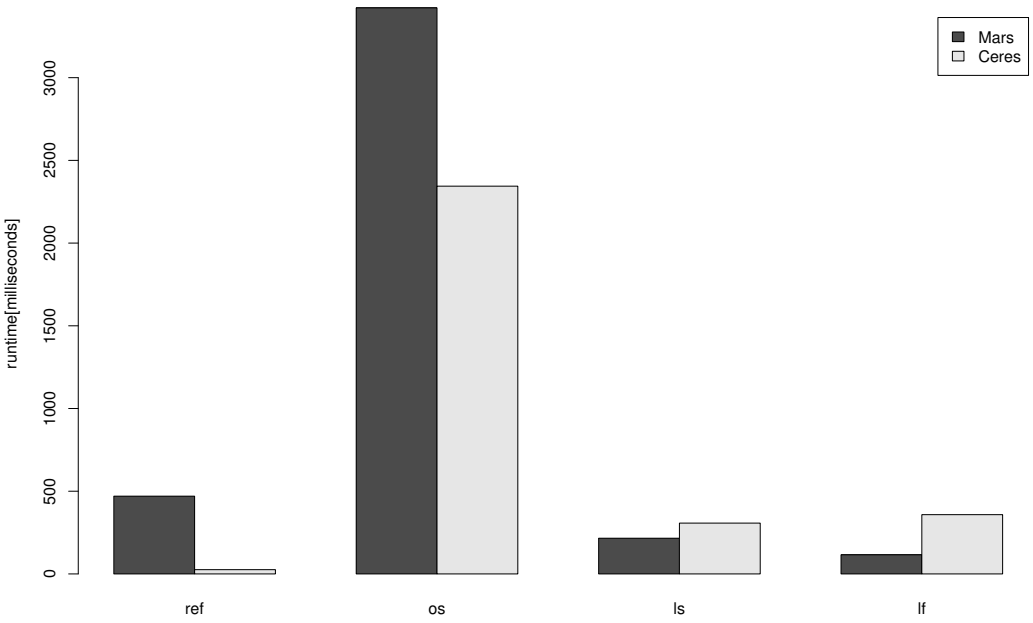

Figure 6: Comparison of `remove` performance, on both machines

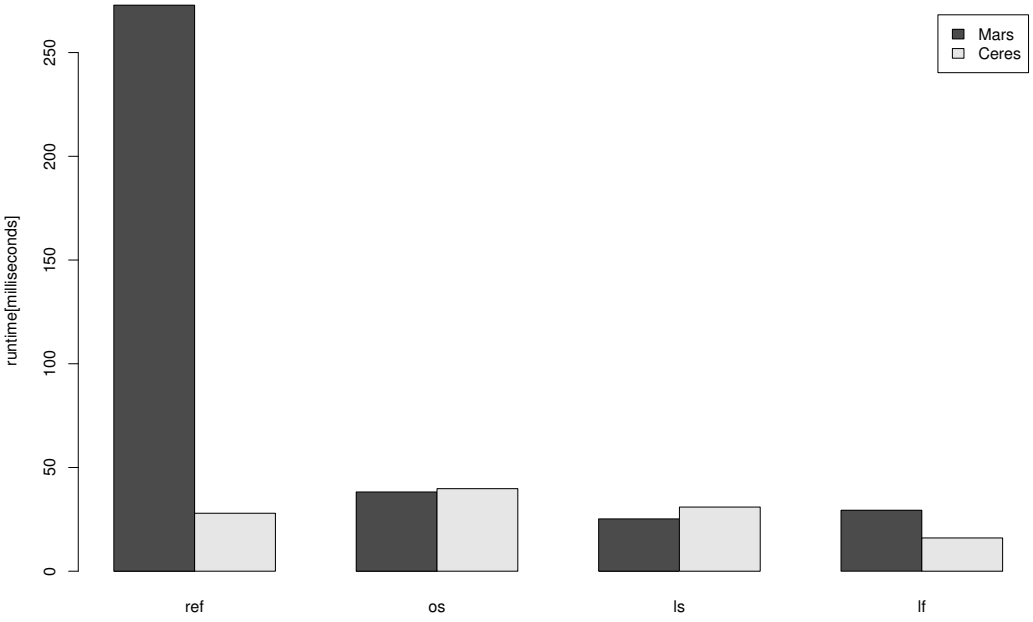

Figure 7: Comparison of `mixed` performance, on both machines

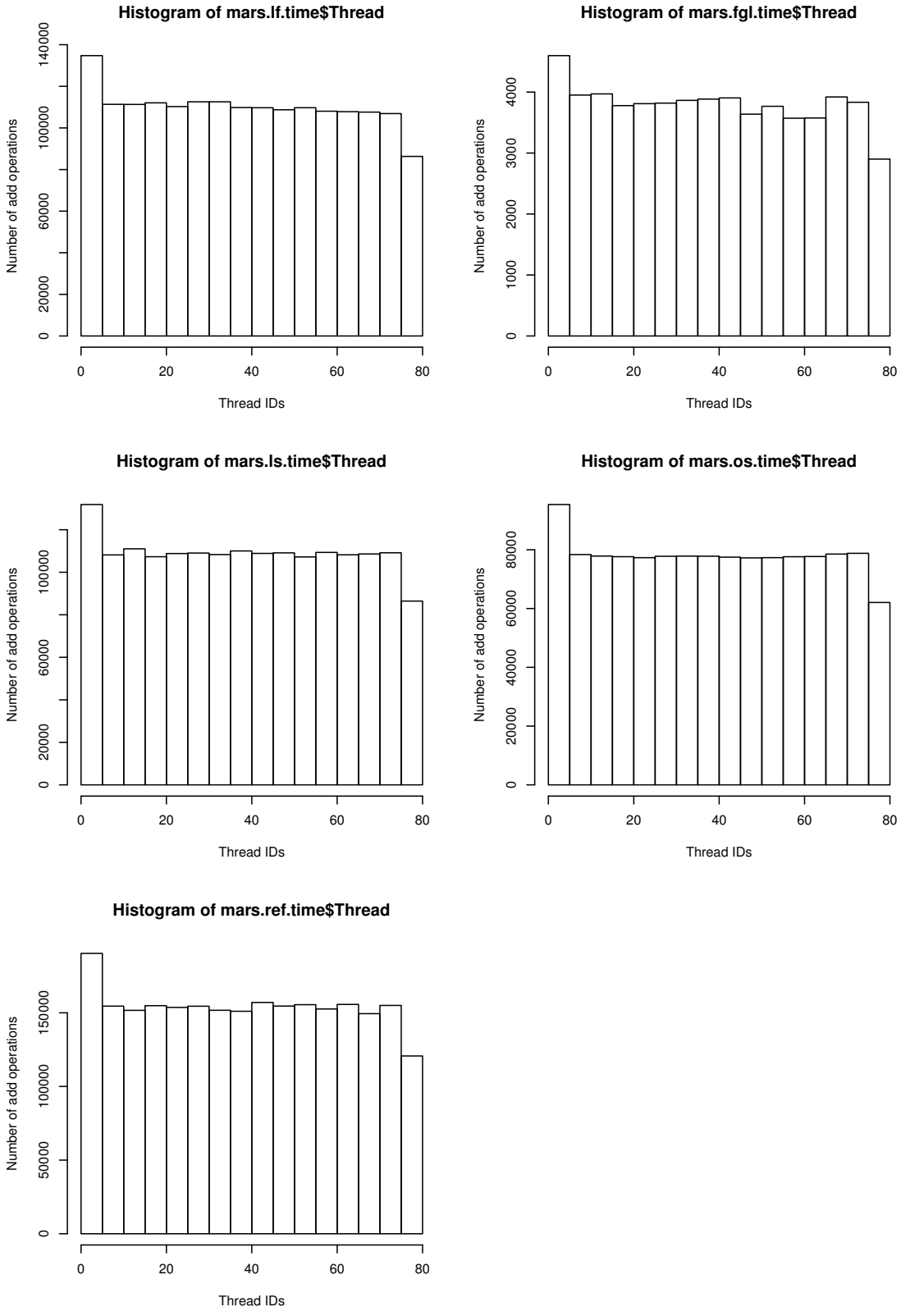

Figure 8: Histograms of 5 second runs of `insert`, on Mars, with 80 threads

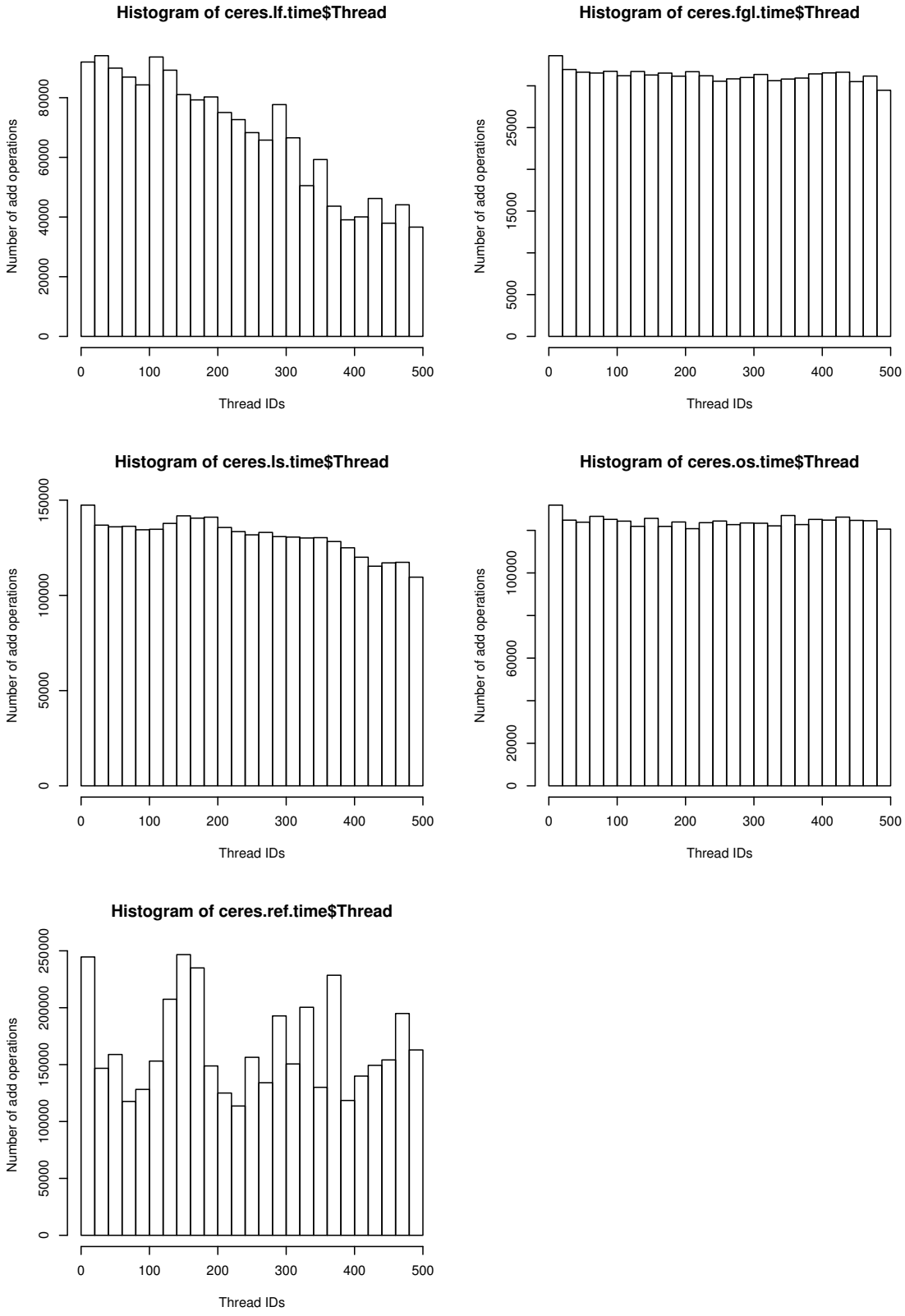

Figure 9: Histograms of 5 second runs of `insert`, on Ceres, with 500 threads

References

[1] cppreference.com. cppreferences std::atomic::compare_exchange. `http://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange`, June 2015.

[2] Eric C. R. Hehner. Bunch Theory: A Simple Set Theory for Computer Science. *Inf. Process. Lett.*, 12(1):26–30, 1981.

[3] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, April 2008.

[4] AMD64 Technology. AMD64 Architecture Programmer's Manual. `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v2.pdf`, September 2012.

[5] Jesper Larsson Träff. Advanced Multiprocessor Programming Slides, 2015.