date: 02/28/2020

My implementation

- I have assumed that the compiler would preserve the data dependency for a single thread, so I try to relax the memory order as much as possible when the correctness is not violated. To detect possible errors, I add a checker to check whether the final shared counter is as expected.
- On node2x18a, I use _mm_pause() for spinning which is said to improve performance (when there is no backoff).
- I have used barriers just as stated in the assignment page.

Experiment 1(consists of 1.1 each and 1.2 total)

1. no sync: nothing special
2. mutex: I use pthread's mutex.
3. tas: I use C++'s std::atomic_flag and its member function test_and_set(). The memory order is acquire; since the counter must be incremented after the lock is acquired.
4. tatas: since atomic_flag.test() is not supported by the g++ on the node2x18a machine (the version is old), I use std::atomic<bool> and its member functions exchange(for tas) instead (and can be test directly);
5. tatas with backoff: I use an additional while loop to serves as the pause function.
6. ticket: I use atomic<int> for the integer recording the next ticket number and its member function fetch_add with memory_order_relaxed for fai.
7. mcs: as stated in Shared-Memory Synchronization by Michael Scott, I use memory_order_release when requesting the predecessor to ensure that the initialization of waiting is prior to the release of the predecessor(otherwise, the predecessor may first release me and then my waiting is set to true).
8. fai: I use std::atomic<int> for the shared counter and its member function fetch_add with memory_order_relaxed.
9. local: nothing special.

Experiment 2

I test Intel TSX's HLE against normal tas with backoff. I don't add backoff for the tas lock.

Experiments

---

setting

1. Intel machine (node2x18a)
- x86_64
- 72 vCPUs, 2 sockets
- L1: 32K L2: 256K L3: 46080K

2. IBM machine (node-ibm-822)
- ppc64
- 160 vCPUs, 20 sockets
- L1: 64K L2: 512K L3: 8192K

Besides,
- I fix the number of processors as 36;
- The thread setting is 1, 2, 4, 6, 8, 12, 16, 32, 48, 64, 96, 128;
- i is the default setting, 10000;
- Same setting is run 3 times and the results are average;
- The experiment is conducted when no one else is running at the same time.

---

result

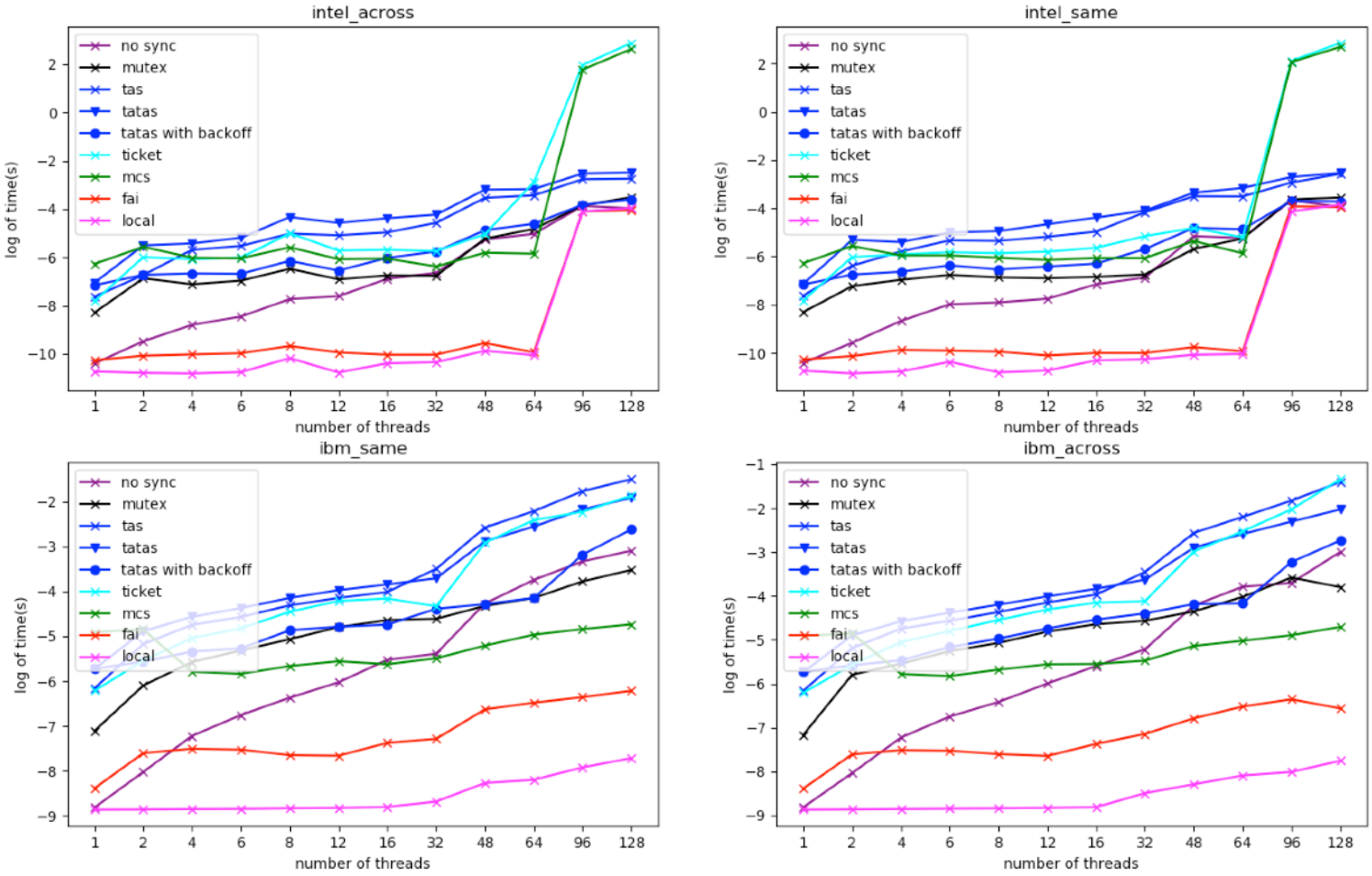


Figure 1. The result of Experiment 1.1 (i=10,000)

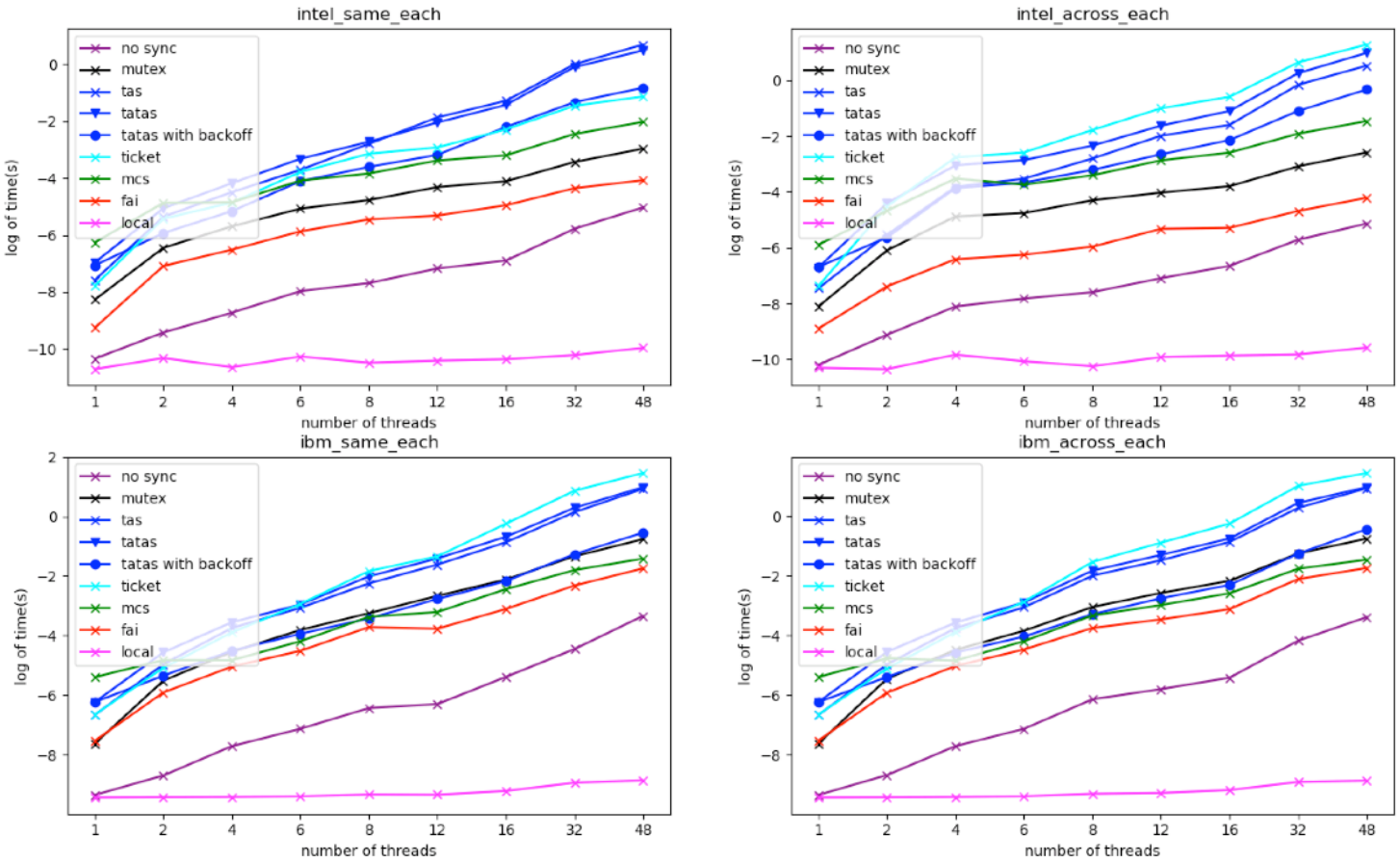


Figure 2. The result of Experiment 1.2 (i=1000)

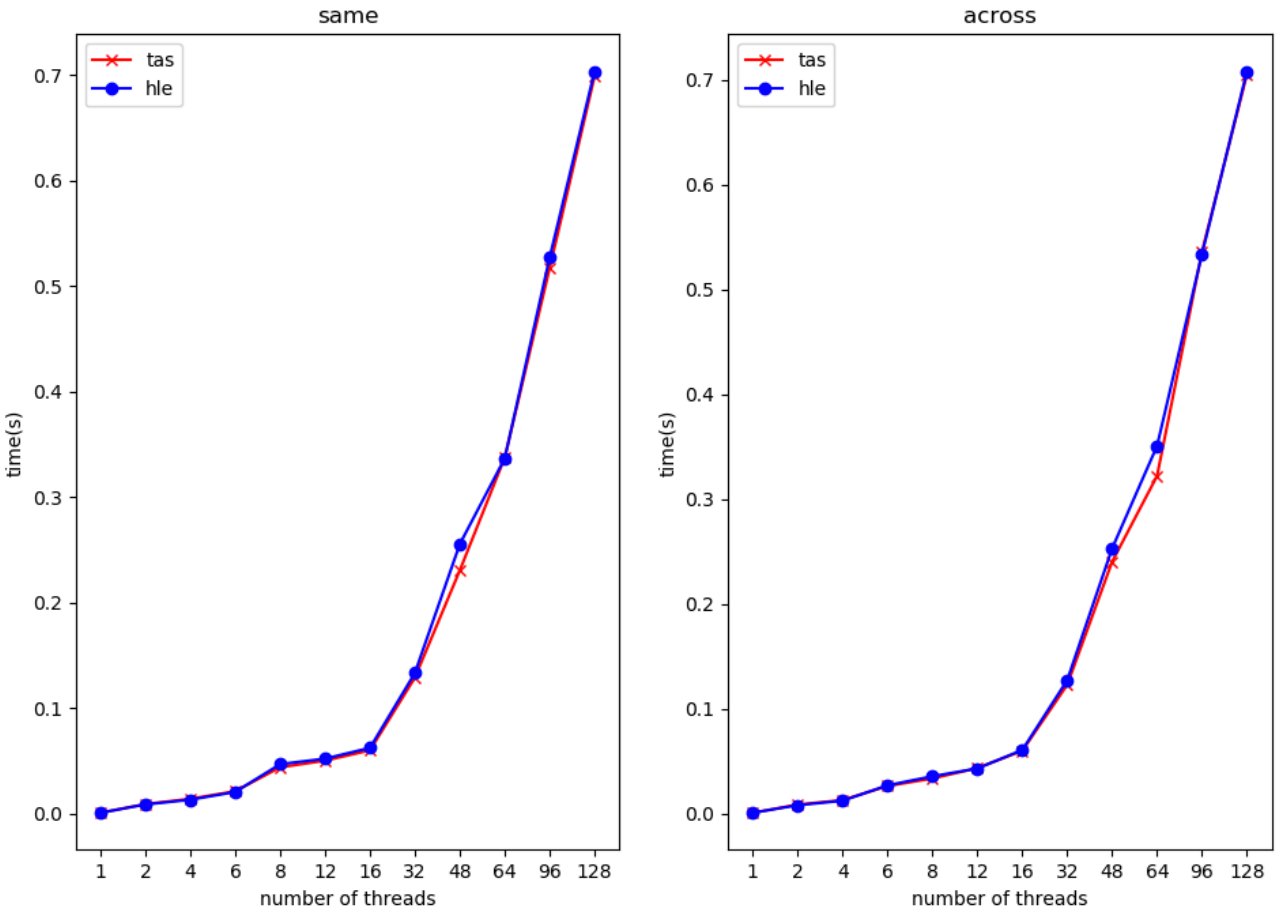
Experiment 1

- no sync
  - Though each thread has incremented the clock many times, the speed is still fast. That's probably because there is no spinning and each awaken thread can do something -- even though there are many useless increment.
  - 1.1: The thread count distribution is even but each thread counts nearly i times.
- Mutex
  - works and scales pretty well. That's probably because there is a scheduling behind the implementation and the thread will keep sleeping until the lock is released. So the wasted CPU time is little.
  - 1.1: The threads count evenly.
  - 1.2: The result shows that the fairness of mutex is pretty good; it's significantly faster than MCS lock on intel platform.
- tas/tatas/tatas with backoff
  - Surprisingly, tatas(without backoff) doesn't perform better than tas on both platforms. This makes me believe that there is some optimization

for the TAS when the lock is already set, so that a TAS on a set lock doesn't issue a set on it.
    - 1.1: the distribution of tas and tatas is even; while in tatas with backoff, certain threads increment the majority. This may explain why there are improvements for this method.
    - 1.2: as expected the performance degrades a lot when fairness is considered. TATAS with backoff demonstrates that contention in multithreading is worse than waiting(backoff).
- Ticket/MCS
    - The overhead for mcs and ticket is large(espcially for storage). When the number of threads is small, their performance is pretty bad.
    - For node2x18a, the time cost of queued spin locks (ticket, mcs) increases dramatically when the number of threads is larger than 64. Profiling like gprof shows that the major time cost is the spinning; so possibly due to the fairness nature of queued spin locks, when the thread to be served doesn't wake up, many other threads that wake up first just find that it's still not their turn and waste time spinning. But interestingly this doesn't happen on the IBM machine: the mcs lock maintains a good and stable performance when the number of threads grows.
    - 1.1: Because of fairness, each thread counts almost the same times.
    - 1.2: It demonstrates that spinning on the same shared variable totally offsets the benefits brought by fairness.
- fai
    - Can be really fast; however, there is always no correctness guaranteed. We can't know the value of the counter before we increment it. So when there are more and more threads, the final counter could be larger and larger.
    - 1.1: Each thread counts almost the same times.
- The local counter shows the benefits of privitization. Even though the total number of increments is much larger, it's still much faster than any other methods.

Experiment 2

As expected, judging from single run, the performace of hle is not stable; sometimes it's better than tas but other times it can be worse. The result here is on average, and suprisingly, their performance are really close. I have tested for larger is, but it only turned out that they are still very close(hle performs a bit better, but not significant). So here I just show the case for i = 10,000.



The experiment 2 is only conducted on node2x18a(must be a intel machine).
Figure 2. The result of experiment 2 (i = 10,000)