Final Project Report

by Shang Wang, Taolun Chai and Xiaoming Jia

Concurrent Counting using Combining Tree

1. Introduction

Counting is one of the very basic and natural activities that computers do. However, for concurrent programs running on multi-core/multi-processor machines, designing a counting algorithm that scales well is not an easy task. In particular, the following three aspects are general concerns when doing concurrent counting on very large shared-memory processors:

1.1 Synchronizations

Synchronizations can be very essential in ensuring correctness. For example, some worker threads may need to be waiting or suspend until other threads finish their calculations, and some threads' activities may even depend on other threads' results. Although most of the times, approximation may be allowed when collecting result, more often than not, we still need concrete results with the help of barriers. In our algorithm, the effect of synchronization and coordination of threads to the program is very well demonstrated, which will be discussed in detail in later chapters.

1.2 Lock contentions

Another part that concurrent counting should take care of is lock contentions. In traditional ways of countings, usually multiple threads are competing with each other for accessing a shared variable, each of them tries to get into the critical section. Thus, tremendous amount of running time is wasted on lock contentions. Combining tree algorithm avoids this problem elegantly by changing the critical region from one hotspot to multiple nodes in the combining tree, which decreases the lock contention very efficiently.

1.3 Real parallelism and scalability

As we explained in 1.2, traditional countings may allow multiple threads to access the same hotspot. However, only one worker is able to get into the critical region at a particular time. This in essence lose the advantage of concurrency, because in fact, all threads are still forced to do sequential operations in the critical region. Combining tree algorithm allows each thread to concentrate on separate working fields, so they can start working at the same time without worrying about each other. This method can achieve very good throughput result despite the fact that one single operation maybe slower.

2. Basic Concurrent Counting

To better understand how memory contentions can effect the parallelism of a program, we will discuss several basic counting techniques in concurrent cases for comparison. Each of them has advantages and disadvantages, and we will use their performance to evaluate our combining tree algorithm in Test and Benchmark section.

2.1 Single Lock approach

When a thread wants to increase the counter, it will need to grab the lock which protects the counter being accessed by more than one thread. The sample code is he following:

```
int Single_Lock_Counter::getAndIncrement(){
        ScopedLock l(&lock_);
        counter_++;
        return counter_;
}
```

2.2 Atomic Operation approach

When a thread want to make an increment on the counter, it uses an atomic operation to perform the increase. The sample code is in the following:

```
int Atomic_Counter::getAndIncrement(){
        return __sync_fetch_and_add(&counter, 1);
}
```

2.3 Thread-based variables approach

The idea thread based variables is that each thread contribute to the counting will have a local counter which is owned by the thread. When a thread want to increase the counter, it must register itself as a contributor on this counter and then directly increase the local counter. But when the read operation is called, there will be a lock to protect the global counter, which will prevent new thread from registering/unregistering to this counter. Then it will collect every local counter owned by every registered thread then sum these up as contribution then add the contribution to global counter. The read operation will return the summed global counter. This approach has four methods:

Register Thread :

If new thread want to increase, it must register to the the counter and the counter will have a reference to that thread's local counter.

***Unregister Thread*:**

If a thread will no longer contribute to the counter, the counter will add this thread's local count to global counter, then remove the reference which point to this local count.

Increment:

The thread increases its local counter.

Read:

Read from the global counter, then read from local counts for threads which are not unregistered, then return the sum.

3. Combining Tree

In the basic counting algorithms we explored in section 2, single lock based and atomic operation based counting both will largely degrade performance in large scale concurrent counting due to the fact that both algorithms will generate large memory contention when concurrency increases. Thread based variable counting scales well on update operation, however, since each read operation requires the counter to iterate all threads' local counter and sum local counts, which takes O(n) time, this algorithm does not scale well when frequent read operation is required. A combining tree is a complete binary tree data structure which can be used to implement a simple fetch_and_Ø(in counting this will be fetch_and_add) operation on a shared variable. Instead of having one hot-spot(memory access point) that becomes a performance bottleneck when concurrency increases, a combining tree will have many smaller hot-spots. We will discuss combining tree in details in this section.

3.1 Data Structure

A combining tree is a complete binary tree. Each working thread is assigned to a leaf node and each leaf node can have at most 2 threads been assigned to it. The counter variable is stored at the root of the tree. Figure 3-1 shows a combining tree with 8 leaf nodes, which can be concurrently accessed by maximum 16 threads.

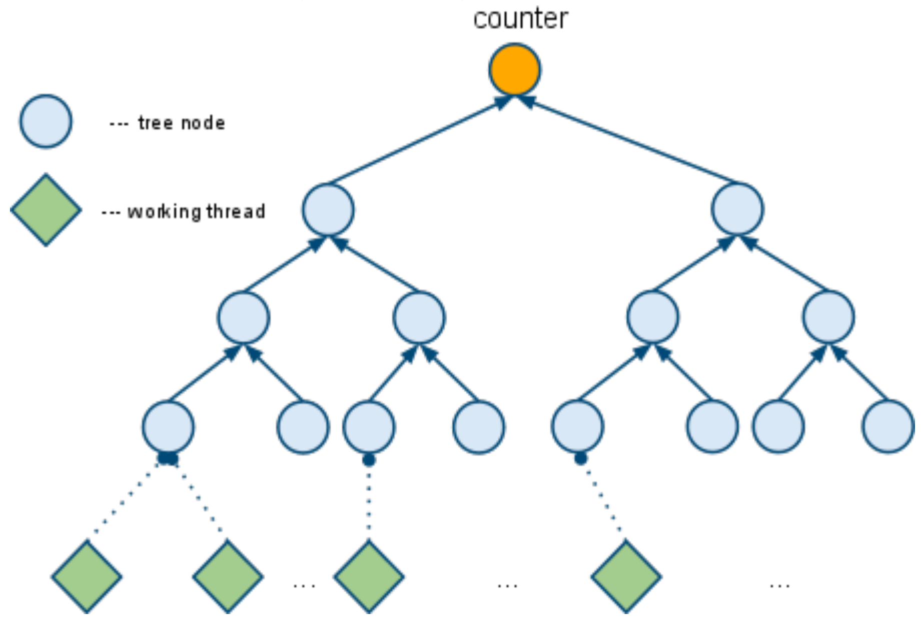


Figure 3-1: a combining tree with 8 leaf nodes

3.2 Combining Algorithm

In combining tree, if a thread wants to increase the counter, it starts from a leaf node and works its way up the tree to the root. If two threads reach a node at approximately the same time, the first arrived thread becomes the ***active*** thread and the second arrived thread becomes the ***passive*** thread. The ***active*** thread will combine its own increment request with the ***passive*** thread's request and carry the combined requests up to the next level while the ***passive*** thread waits for the ***active*** thread to return with fetched result. An ***active*** thread might become a ***passive*** thread later while climbing up the tree. When a thread reaches the root of the tree, it will increase the counter's value and fetch the old value and pass this value down the tree. Thus if multiple threads are increasing the counter at approximately the same time, the maximum number of threads that will compete for accessing the counter variable is 2, no matter how many threads are making increment requests. In this way, combining tree distributes a single hot-spot among all the nodes in the tree and each node becomes a much smaller hot-spot, thus reduces memory contention.

3.3 Implementation

Phases:

There are four different phases a thread has to go though when making an increment, which are ***pre_combine***, ***combine***, ***operation*** and ***distribute***. In ***pre_combine*** phase, a thread will start from the leaf node and climb up the tree until it either reaches the root or becomes a passive thread(the second thread that arrived at the node), a combining path is determined in this phase. If a thread stops at a node other than the root during ***pre_combine***, it will lock that node to indicate that it will return with combined requests, this will keep the active thread from combining the node without the passive thread's combined requests. After ***pre_combine***, the working thread will combine all the node's request along the combining path, if a node is locked, the working thread will have to wait at that node until it is unlocked then combine the requests from the passive thread and so on. After ***combine*** phase, a thread either reaches the root of the tree or stops at a node. In *operation* phase, a passive thread will puts its combined increment requests in the node and unlock the node and starts waiting for the result from the active thread while an active thread increases the counter at the root. Finally, during ***distribute*** phase, the active thread will go down its combining path and distribute the result to the passive thread waiting at the node(if any) until it reaches the leaf node. After 4 phases, a thread will get the counter's old value and finished an increase operation on the counter.

Node structure:

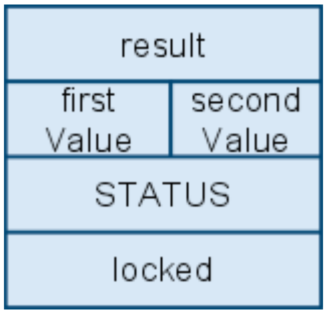


Figure 3-2: structure of a tree node

result: stores the result fetched by the active thread
firstValue: stores the combined increment requests by the first thread
secondValue: stores the combined increment requests by the second thread
status: used to indicated the combining stages this node is currently in, can be *IDLE,FIRST,SECOND,RESULT* and *ROOT*
locked: indicate if this node is locked by another thread

Node Status:

ROOT**:** indicate that this node is the root of the tree
IDLE**:** this node is currently idle
FIRST**:** this node has been visited by one thread
SECOND: this node has been visited by two thread
RESULT: the active thread has delivered the requests and returned with result

Synchronization:

There are two term of synchronization in combining tree: short term synchronization and long term synchronization.
Short-term synchronization**:** Since each node can be shared between two threads, synchronization between those two threads is needed to prevent two threads from modifying the same node at the same time.
Long-term synchronization**:** During the increment operation, an active thread need to wait on a locked node in combine phase and a passive thread need to wait for the active thread in operation phase. These two kinds of synchronization is long term synchronization.

Our first implementation uses a mutex to guard each method call on a node to achieve short term synchronization and uses conditional variable to achieve long term synchronization:

```
bool Node::pre_combine(){
  ScopedLock l(&node_lock_) // short term synchronization
```

```
while(this->locked_) {
  cond_var_.wait(&node_lock_); // long term synchronization
}
...
```

However, mutex and conditional variables are expensive to use, especially when a working thread have to repeatedly acquire/release the lock each time it enters/exits a node. So we adapted the idea behind spin lock and used one variable to indicate the synchronization status of the node and use compare and swap(CAS) to achieve both short-term and long-term synchronization:

```
bool Node::pre_combine(){
  while(!(__sync_bool_compare_and_swap(&lock_,UNOCCUPIED_AND_UNLOCKED,
    OCCUPIED_AND_UNLOCKED))){
    nanosleep(&SLEEP_TIME,NULL);
  }
  switch(this->cStatus_){
    case IDLE:
      cStatus_ = FIRST;
      __sync_fetch_and_sub(&lock_,2); // set lock_ to unoccupied&unlocked
      return true;
    case FIRST:
      cStatus_ = SECOND;
      __sync_fetch_and_sub(&lock_,1); // set lock_ to unoccupied&locked
      return false;
    case ROOT:
      __sync_fetch_and_sub(&lock_,1); // set lock_ to unoccupied&locked
      return false;
    default:
      std::cerr << "error! unexpected Node state in precombine!" << std::endl;
      ::exit(1);
  }
}
```

In this implementation, an integer lock_ is used to indicate the lock state of the node, which can be *UNOCCUPIED_AND_UNLOCKED,UNOCCUPIED_AND_LOCKED,OCCUPIED_AND_UNLOCKED_,OCCUPIED_AND_LOCKED* and *RESULT_READY.* An occupied node means one thread is already modifying the node and any other threads that try to enter the node will fail the **CAS** at the beginning of the method and will spin until the **CAS** succeeds. A locked node means that the node is in the process of being combined and any threads try to combine this node have to wait. When a thread exits a node, it uses *__sync_fetch_and_sub* to atomically change lock_ to the desired status.

4. Test and Benchmark

4.1 Test case design:

For all the counting algorithms, we test correctness under the following circumstances:
Different number of thread doing update. Thread number we chose is 1, 2, 4, 8, 16, 32, 64 and 128.
Different number of thread doing update while one thread do read. Update thread number we chose is 1, 2, 4, 8, 16, 32, 64 and 128. Read thread number is always 1.
Different number of thread doing read while one thread do update. Read thread number we chose is 1, 2, 4, 8, 16, 32, 64 and 128. Update thread number is always 1.
All the above test case has been test and passed under different machines with different core number and test with or without nanosleep intervals.

4.2 Benchmark

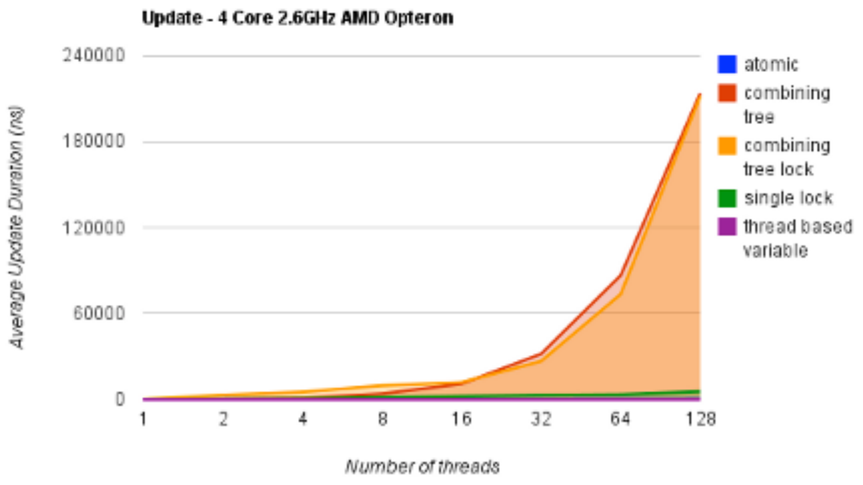
Benchmark method:

After insured the correctness of our implementation, we calculate the total operation time t and divided with thread_count*repeat_time to get average duration. In detail, we first pass a structure with a field called "total_time" into each method, and adopt the ticks_clock that we used before and wrap the operation loop with two time stamps, this gives us the total running time for each thread. Then, we assign total_time with the result and get the duration. In this way, we are able to get the precise time.
Following are the sample codes for the result collection:

```
void TreeTester_1::test1(TestCombo * tc_p){
  TicksClock::Ticks start = TicksClock::getTicks();
  for(int i = 0 ; i < tc_p->repeat_time; i++){
    tree_->getAndIncrement(tc_p->thread_id);
  }
  TicksClock::Ticks end = TicksClock::getTicks();
  tc_p->update_total = getTotal(start, end);
}
```

Performance analysis:

Following are the performance results for multiple updates with different amount of running threads under different platforms, each thread does update operation for 10,000 times:



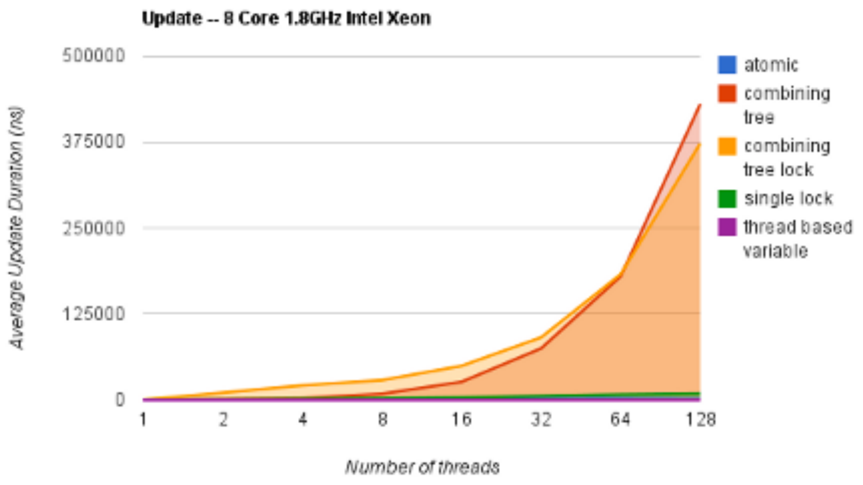


Figure 4-1 plots for multiple read operation on different machines

As we can see, when thread count is small, the running time for combining tree(red for busy wait, yellow for lock-based) is almost identical to other algorithms. However, as the thread number increases, especially when thread count is greater than 64, the running time is increased exponentially.

Why the performance instead of growing better, actually going worse? After some careful observations, we get the following facts:

Thread coordination latency

Just as we said at the beginning, combining tree algorithm relies heavily on thread synchronizations. In the ***combine*** phase, only the active thread is able to successfully combine the value and go to the upper level, while the ***passive*** thread should only wait

until the ***active*** thread comes back with acknowledgement. However, the ***active*** thread in one level may become ***passive*** thread for the upper level, which means that the ***passive*** thread at current level will wait for even longer time before the ***active*** thread comes back. Therefore, even at most 2 threads will compete for lock at the same time, the combining tree has an inherent disadvantage with respect to locks and synchronizations: each increment will have higher latency. Even if one thread is able to finish its task without any disturbance, the travel and operation on each node also creates additional latency compared to the algorithms with simpler operation.
Another fact is that, for multi-processor programs, a running process A which is holding the lock maybe rescheduled by CPU; At the mean time, another thread B, which is waiting for A to release the lock, will be blocked for longer time because A is not yet awakened by CPU.

Latency of using mutex and conditional variable

Another fact that may slows down the program is the mutex lock/unlock and wait/signal operations. Following is one part of the profiling plot that we derived using google profiler:

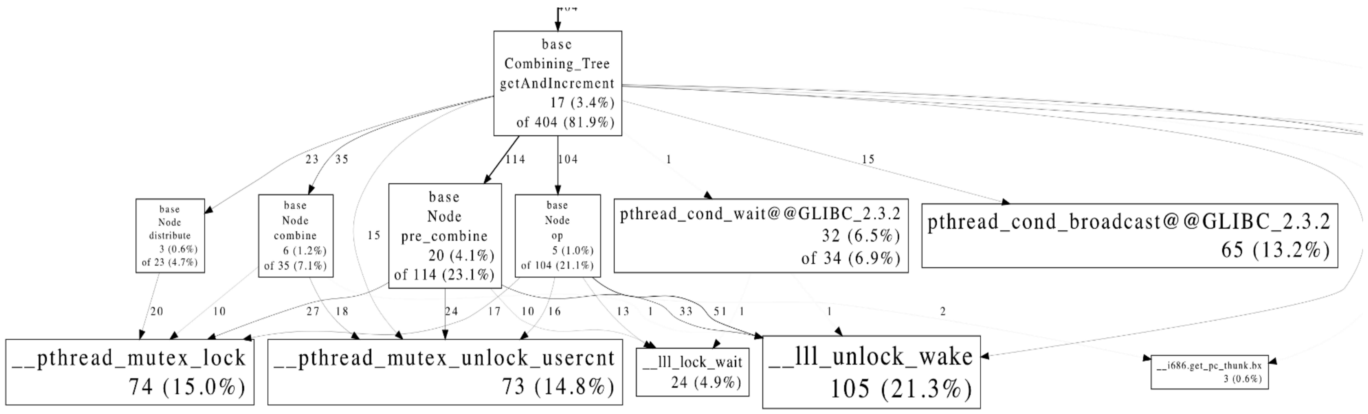


Figure 4-2 profiler plot for combining tree using lock

As shown on the above plot, almost 70% of the time for the getAndIncrement() method is spent on these mutex operations. That's also the fact that inspired us to switch from mutex to busy waiting method.

Latency using busy waiting

Due to the fact that most of the time is spent on the mutex operation, we decide to switch mutex operation to busy waiting approach which using sync_compare_and_swap. When the thread number is under some threshold we do observed the boost on average update speed, however when the thread number exceeds the threshold, the performance decreases dramatically. This decrease is due to fact that nanosleep() brings in lots of context switch time. And this is busy waiting approach, there must be some case that the running thread spin and wait for the spinning variable switched to the desired mode. However, the thread holding the spinning variable is already rescheduled.

Memory allocation/deallocation

After we implemented the busy wait method, we generate the plot again, and this time we achieve the following plot:

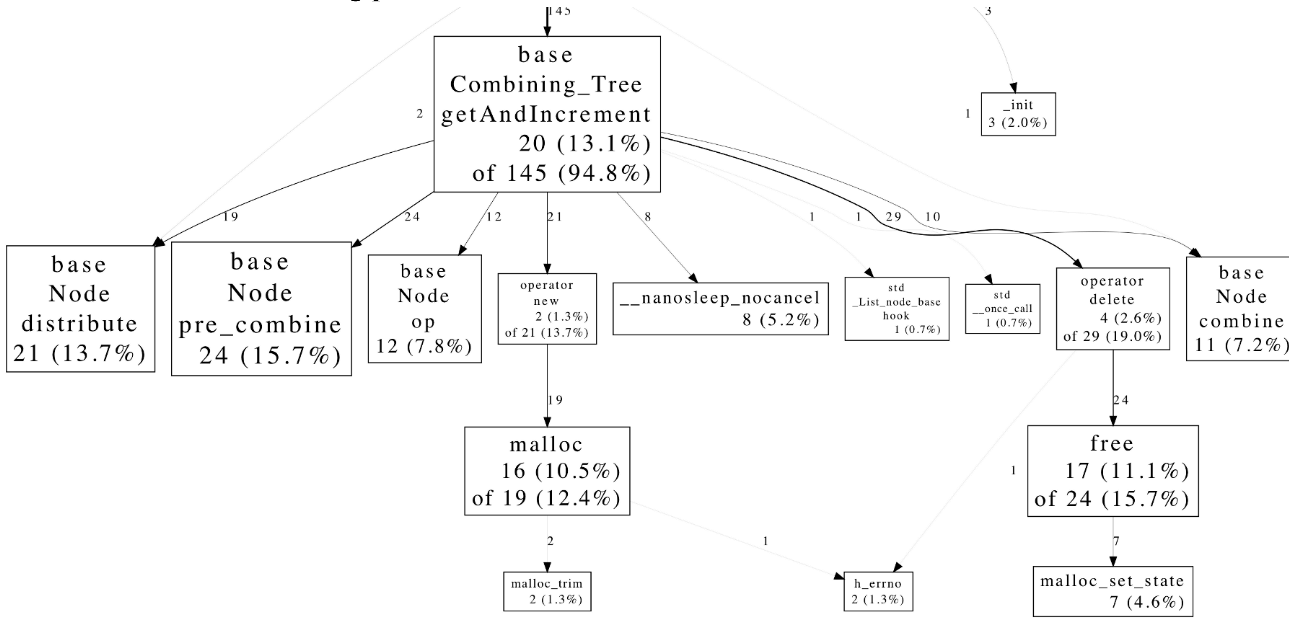
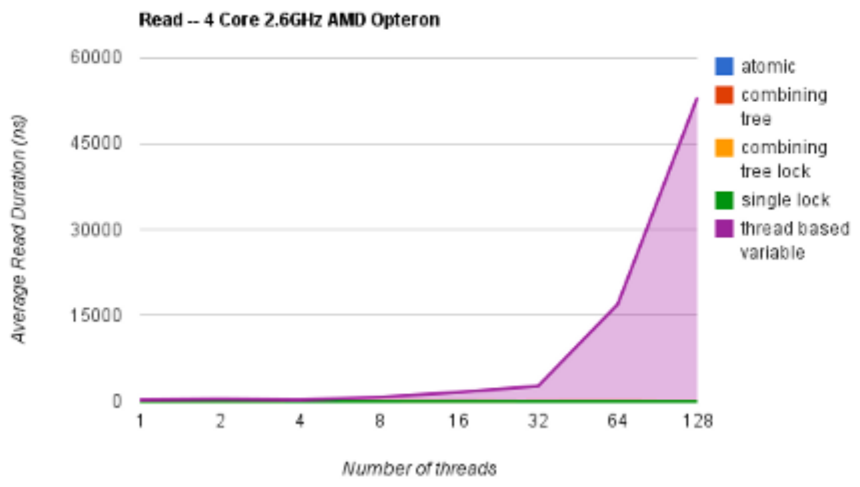

145
base Combining_Tree getAndIncrement 20 (13.1%) of 145 (94.8%)
3
_init 3 (2.0%)
2
1
19
24
12
21
8
1
1
29
10
base Node distribute 21 (13.7%)
base Node pre_combine 24 (15.7%)
base Node op 12 (7.8%)
operator new 2 (1.3%) of 21 (13.7%)
__nanosleep_nocancel 8 (5.2%)
std _List_node_base hook 1 (0.7%)
std __once_call 1 (0.7%)
operator delete 4 (2.6%) of 29 (19.0%)
base Node combine 11 (7.2%)
19
24
malloc 16 (10.5%) of 19 (12.4%)
1
free 17 (11.1%) of 24 (15.7%)
2
1
7
malloc_trim 2 (1.3%)
h_errno 2 (1.3%)
malloc_set_state 7 (4.6%)

Figure 4-3 profiling plot for combining tree using busy wait

It's easy to observe that the allocation and deallocation for temporary stack that records combining path is also a time-consuming part in the algorithm.

Although our result at update side is far beyond satisfaction, the reader side achieves very good result. The following are the reading side operation plots:



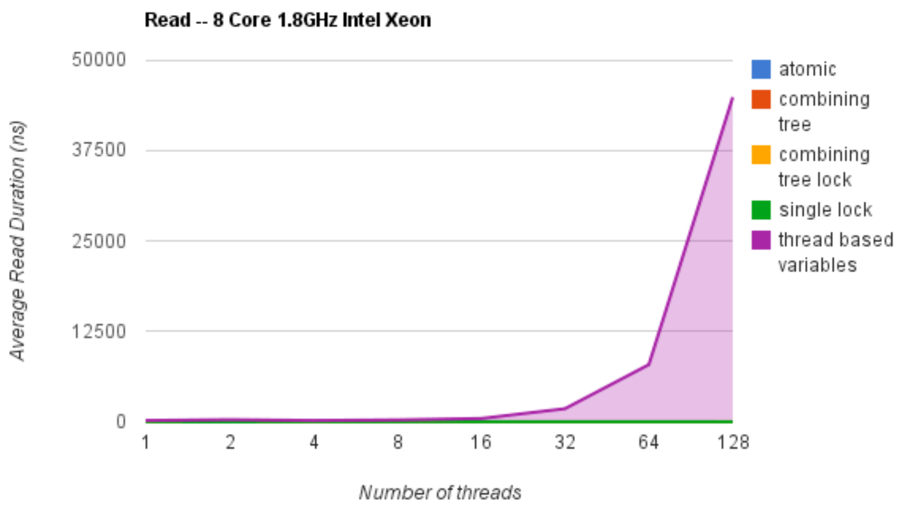


Figure 4-4 Plot for multiple reading operation on different machines

As we can see from above plots, the read operation efficiency for combining tree algorithm is almost as good as other basic algorithms, while the thread-based variable algorithm performs poorly as the number of thread increases. This is quite intuitive because the result for combining tree can be easily fetched from the root, but for thread based algorithm, it should also collect from those threads which are not yet unregistered, thus yields high latency in reading.

5. Conclusion

Combining tree can distribute the memory contention caused by multiple threads accessing a shared variable down the tree to all nodes in the tree. However, combining tree suffers high latency due to large overhead in a single update. Due to the testing environment limitation, we were only able to test our combining tree implementation on maximum 8 cores machine. With high overhead(acquiring/releasing lock and allocate/free memory) in combining tree counter and relatively low memory contention in single lock counter(the maximum number of concurrently running threads is limited by the number of cores available), single lock counter out performs combining tree when concurrency is low. Even if our experiment does not achieved expected result, ideally, as described in [3], when the concurrency increases dramatically, combining tree will result better performance.

Source code of this project can be found at:
https://github.com/markmarch/MCP-Counter

Reference

[1]. MCKENNEY,P.E., 2011, Is Parallel Programming Hard, And, If So, What Can You Do About It?, Chapter 4 section 2.
[2]. MAURICE HERLIHY, NIR SHAVIT, 2008, The Art of Mutiprocessor Programming, Chapter 12 section 3.
[3]. HERLIHY,M., LIM,B.H., AND SHAVIT,N., 1995, Scalable Concurrent Counting. ACM Transactions on Computer Systems, Vol 13, NO 4, Page 343-364.
[4]. GOODMAN, J. R., VERNON, M. K., AND WOEST, P. J. 1989. Efficient synchronization primitives for large-scale cache-coherent multiprocessors.In Proceedings of the 3rd ASPLOS. ACM,New York, 64-75.