README

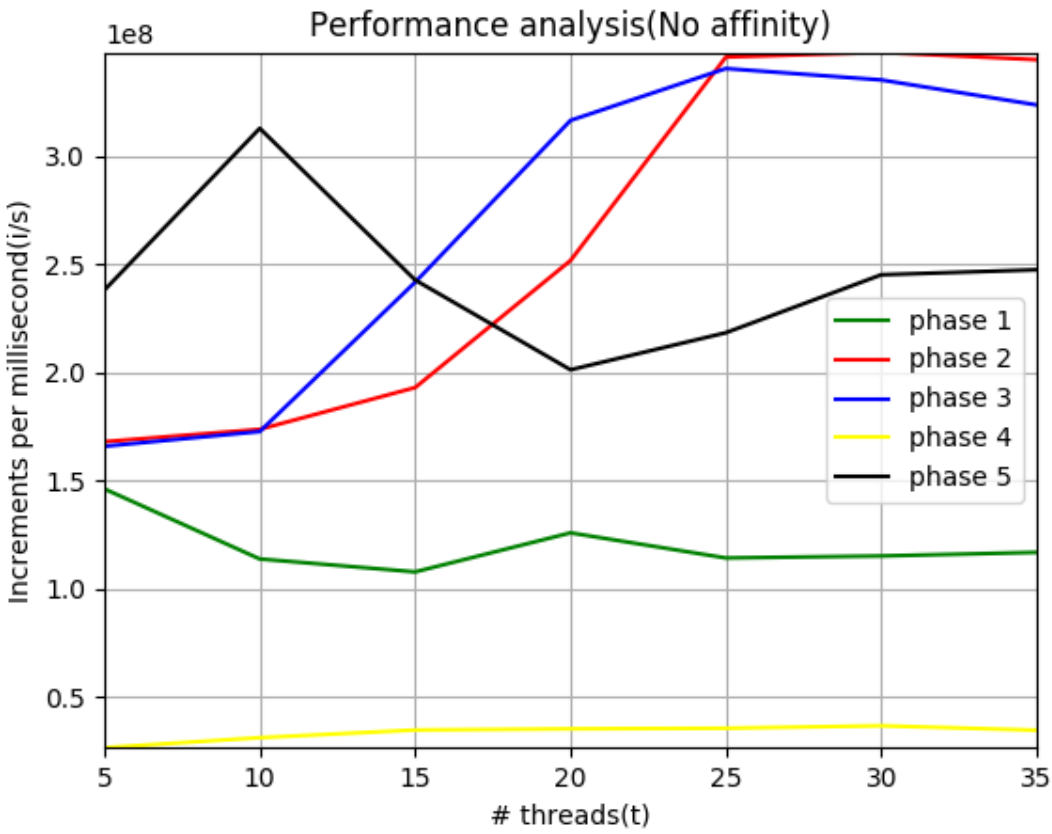
Zhenfei Ji
zji@ur.rochester.edu

Format of input parameters: ./parcount "-t 5" "-i 30000"

In assignment 1, we are required to implement C++ threads, and execute the following phases:

1) With no synchronization
2) Under protection of a mutex with explicit lock and unlock
3) Under protection of a mutex acquired and released via declaration of a local lock_guard
4) Without locking, but with the counter declared as an atomic_int and accessed with fetch_add operations
5) With a local, thread-specified counter, and the main thread adds the local counters together at the end of the phase

With increments equals 60000 and the number of threads range from 5 to 35 with interval of 5, the performance graph that shows increments per millisecond as a function of thread count for each of the phases is as following:



Now let me explain the result of each phase.

In phase 1, there is no synchronization so a data race will occur. Specifically,

- Two or more threads will assess the same memory location (shared counter) concurrently.
- At least one of the assesses is for writing (in this case, all accesses is for writing).

- And the threads are not using any exclusive locks to control their accesses to the counter.

When these three conditions hold, the order of accesses is non-deterministic, and the computation may give different results from run to run depending on that order, which means the final counter is not guaranteed to be t*i. We know that each thread increments the same shared counter in parallel. Although it seems a single line but "counter++" is actually converted into three instructions,

- Load "counter" variable value in register
- Increment register's value
- Store variable "counter" back to memory with register's value

Assuming in a special scenario, order of execution of above these instructions is as follows,

Thread 1: Order of Instructions	**Thread 2: Order of Instructions**
Load "counter" variable value in register	
	Load "counter" variable value in register
Increment register's value	
	Increment register's value
Store variable "counter" back to memory with register's value	
	Store variable "counter" back to memory with register's value

In this scenario one increment will get neglected because instead of incrementing the counter variable twice, different registers got incremented and counter's value was overwritten. Suppose prior to this scenario counter was 98 and as seen in above chart it is incremented 2 times, so the expected result is 100. But due to race condition in the phase 1, the final value of counter will be 99 only.

Basically, in phase 1 the performance of multithreads is almost equal to one single thread because every thread finishes its three-step instructions independently with other threads, and don't need to wait for accessing the shared counter.

In phase 2, under protection of a mutex with explicit lock and unlock operations, the final value of counter is guaranteed to be t*i. Mutex lock in phase 2 makes sure that once one thread finishes the modification of counter then only any thread modifies the counter. That is, the value of counter can be modified by only one thread within a mutex lock (critical section), other threads have to wait for the unlock operation. Therefore, the performance is basically proportional to the number of threads.

There is one problem with explicit lock and unlock operations. If we forgot to unlock the mutex at the end of function or incorrect order of locks acquiring, deadlocks may occur.

In phase 3, instead of using explicit lock and unlock operations, we acquire and release a mutex via declaration of a local lock_guard. The lock_guard wraps the mute inside its object and locks the attached mutex in its constructor. When it's destructor is called it releases the mutex. Therefore, it solves the problems of explicit lock and unlock operations. In the performance graph, the result of phase 3 is similar to phase 2.
In phase 4, it is obviously efficient than other phases. Atomic operations leverage processor support(CAS) and don't use locks at all, whereas locks are more OS-dependent and perform differently on.
Locks actually suspend thread execution, freeing up cpu resources for other tasks, but incurring in obvious context-switching overhead when stopping/restarting thread. On the contrary, thread attempting atomic operations don't wait and keep trying until success, so they don't incur in context-switching overhead, but neither free up cpu resources.
The atomic will lock the memory bus on most platforms. However, there are two ameliorating details:

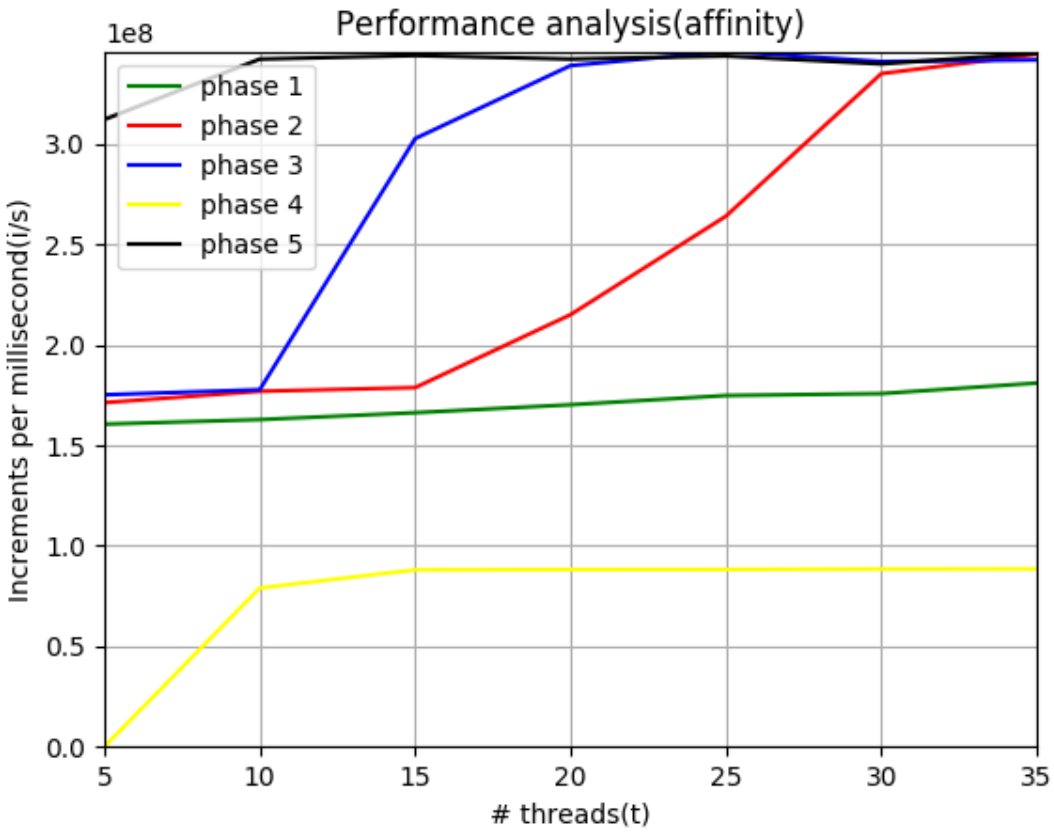
- It is impossible to suspend a thread during the memory bus lock, but it is possible to suspend a thread during a mutex lock. This is what lets us get a lock-free guarantee.
- Mutexes eventually end up being implemented with atomics. Since we need at least one atomic operation to lock a mutex, and one atomic operation to unlock a mutex, it takes at least twice long to do a mutex lock, even in the best of cases.

In phase 5, each thread adds into a local, thread-specific counter, and the main thread adds the local counters together at the end of the phase. From the graph we see that the trend of phase 5 is not very clear, I think I can explain it from the perspective of processor. The type of multiprocessors in the server is NUMA, which means the location of memory will affect the performance of thread. In the implementation of phase 5, I create an array in which the number of elements is equal to the number of threads and the value of elements are all zero initially. However, the location of the elements in the array is not continuous on the physical memory, which makes the speed of accessing that element dependent on the distance from the thread accessing it to the current element. Therefore, the performance does not have a clear trend.

Experiment with processor affinity

1.All threads are restricted to cores of a single socket.

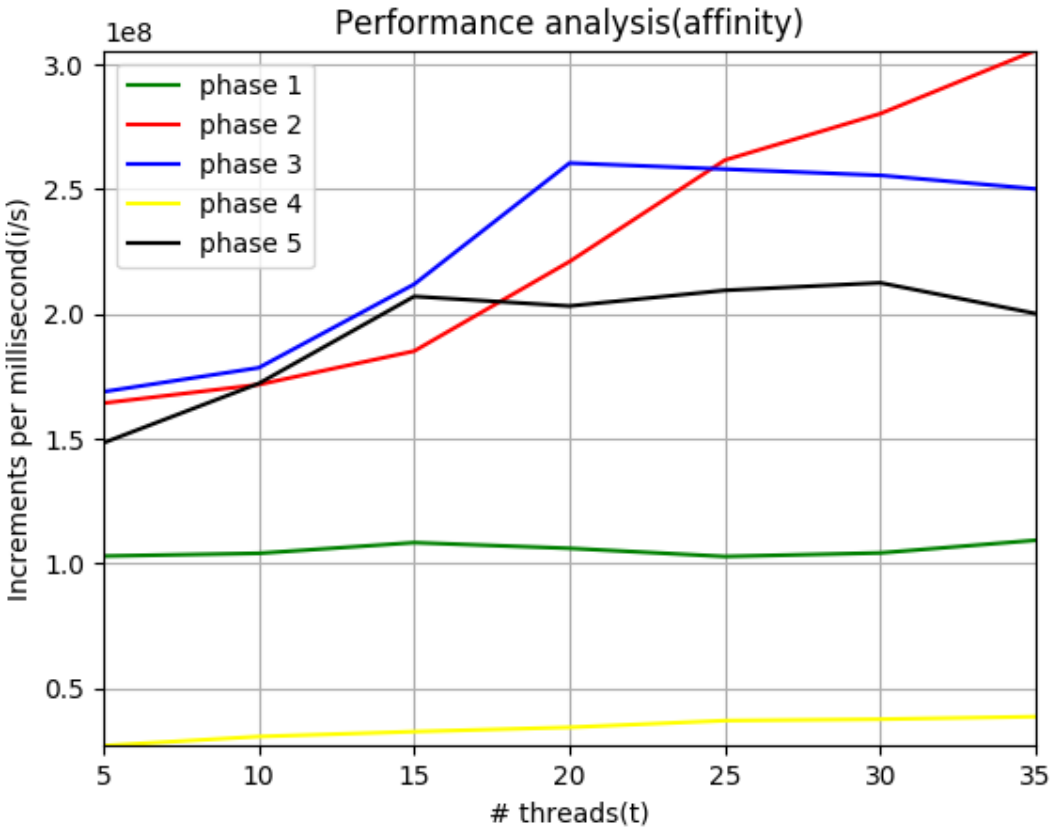
The performance graph is as follows:



Compared with the above performance graph(no affinity), there are some apparent differences.

- Average performance in phase 1 is lower than the previous one without affinity.
- Average performance in phase 3 keeps steady after some points.
- Average performance in phase 4 is lower than the previous one without affinity.
- The trend of phase 4 is more even and steady than previous one.

2.All threads are scattered 50/50 across the two sockets of a two-socket machine. The performance graph is as follows:



There are some obvious differences: performance of phase 3 and phase 2 is higher compared with above pictures. Additionally, average time in phase is lower than the previous one which restricts all threads in one socket.