SeanCooke8892@gmail.com

Parallel and Distributed Systems

README

Purpose

When running parcount, a given number of threads (t) will increment a counter a given number of times (i). After running parcount, the counter should read t * i.

t and i are provided to parcount as command line arguments according to the convention "./parcount –t [t] –i [i]". The argument directly following "–t" (if any) will be t. The argument directly following "–i" (if any) will be i. If multiple "–t" or "–i" flags are found, the last one will be used. If "–t" or "–i" is the last command line argument, it will be ignored.

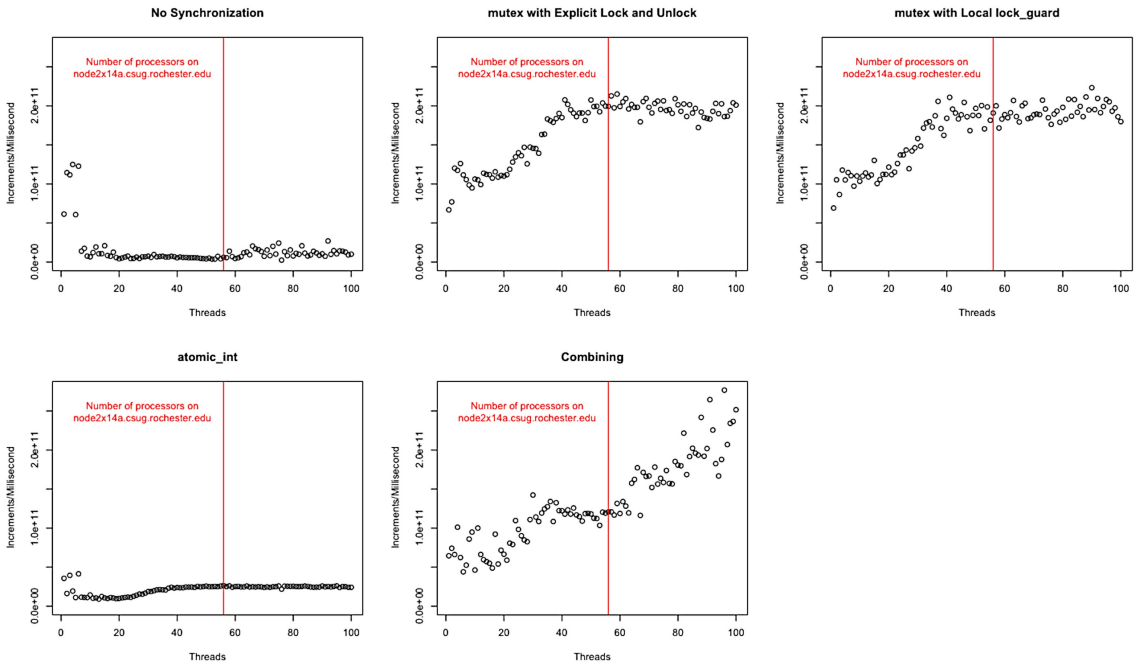
Experiment

The counter is incremented 5 different ways on each run of parcount.

1. No Synchronization – a data race ensues between the threads
2. mutex with Explicit Lock and Unlock - A thread explicitly locks and unlocks a mutex protecting the a counter
3. mutex with Local lock_guard – A thread locks and unlocks a mutex with a local lock_guard
4. atomic_int – The counter is assigned as type "std::atomic<int>"
5. Combining – Each thread increments a local counter i times and provides the local counter to the main thread. After joining the threads, the main thread calculates the sum of all local counters

Results

The results after running parcount on node2x14a.csug.rochester.edu (56 processors), varying t between 1 and 100, and fixing i at 10,000 are shown below.



We will refer to Increments/Millisecond as throughput.

"No Synchronization" creates a data race that results in undefined behavior and low throughput because the shared counter is incremented a number of times less than or equal to t * i.

The throughput of "mutex with Explicit Lock and Unlock" and "mutex with Local lock_guard" increase linearly until the number of processors is reached. After reaching the number of processors, increasing the number of threads fails to increase the throughput. This is because a mutex gives a processor atomic access to critical section variables. After the number of threads exceeds the number of processors, there is no increased benefit to using a mutex. The similarities between these two graphs imply the lock_guard data structure is an abstraction that explicitly locks and unlocks a mutex behind-the-scenes.

Low throughput is seen when using the "atomic_int" data structure because of the large overhead and no concurrency associate with an atomic data structure.

It can be seen that the truly parallel and concurrent solution of "Combining" is the only way throughput increases as the number of threads exceeds the number of processors. This is because at any time, any number of threads less than or equal to t can simultaneously increment their local counter, allowing the paradigm to scale linearly after t exceeds the number of processors.

Conclusions

Data races result in undefiled behavior and some data structures create significant overhead that suppress throughput. There are solutions where hardware will provide increased throughput though there can be a "leveling-off" point. In order to exploit the parallel and concurrent nature of problems, a parallel and concurrent solution must be implemented.