Advanced Parallel Computing: Exercise #8

Due on Tuesday, June 16, 2015

Svend Dorkenwald, Günther Schindler

Reading

The AMD Opteron Northbridge Architecture

The authors introduce the upcoming Northbridge Architecture from AMD. Therefore, they discuss the latest features and present numbers supporting the abilities of thi chip.
Their new design is all about increasing bandwidth and while keeping the latency at a low level. They manage to do this by adding one extra hyperTransport port which clocks at the same frequency as one core. Compared to older generations their Cores are now connected by Direct Connect allowing bandwidths of 8 Gbytes/s between any neighbouring component. Finally, they optimized their system with regard to latency for one-, two-, and four-node systems.
Unfortunately, the authors do not manage to either proof the capabilities of their chip by comparing it to a latest Intel chip nor are they able to introduce anything brand-new. This paper seems like an update on the latest developments at AMD and includes a surprisingly large part about their future plans. Hence, we reject this paper (strongly!).

The SGI Origin: a ccNUMA highly scalableserver

This paper introduces the 'SGI Origin 2000', a cache-coherent non-uniform memory access multiprocessor, which is scaling to small and large processor counts without any bandwidth, latency, or cost cliffs. It uses the distributed shared memory architecture with a crossbar based I/O subsystem. The basic building block of the Origin system is the dual-processor node with 4 GB of main memory. The architecture supports up to 512 nodes, for a maximum configuration of 1024 processors and 1 TB of main memory. A non-blocking bristled fat hypercube network is used to provide a high bisection bandwidth with low latency to local and remote memory. Origin also includes several features to help the performance of applications including hardware and software support for page migration and fast synchronization.
The introduced architecture surprised us with its good scalability and performance results. We think that the use of distributed shared memory combined with the cache-coherent processors leads to a highly productivity from programmer perspective. The support for page migration and fast synchronization is also a plus for most applications. Therefor, we strongly accept this paper.

Red-Black Tree – Coarse grain locking

Our coarse grain locking is based on a shared read lock and a exclusive write lock. We implemented the lock using the PThread reader-writer-lock which allows:

- Any number of threads hold a reader lock at the same time
- Only one thread may hold a writer lock
- When a thread holds a writer lock no reader locks are held by any other thread

If we performe a search-operation, we simply lock this section shared so any other thread can search concurrent.

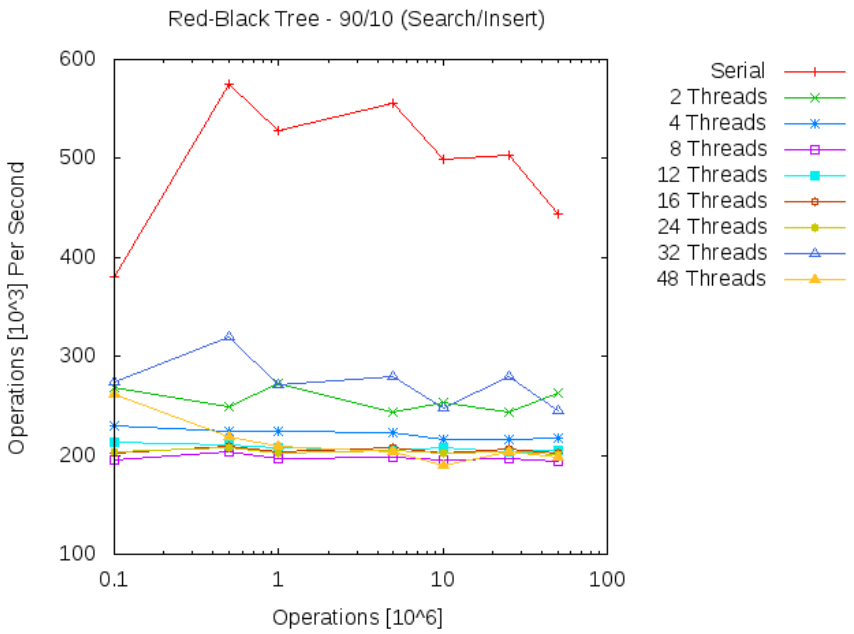
```
void* rbtree_lookup() {
    /* Shared lock */
    pthread_rwlock_rdlock(&rwlock);
    ...
    /* Unlock */
    pthread_rwlock_unlock(&rwlock);
}
```

If we performe a add-operation, we lock the whole tree so no other thread can search or add concurrent.

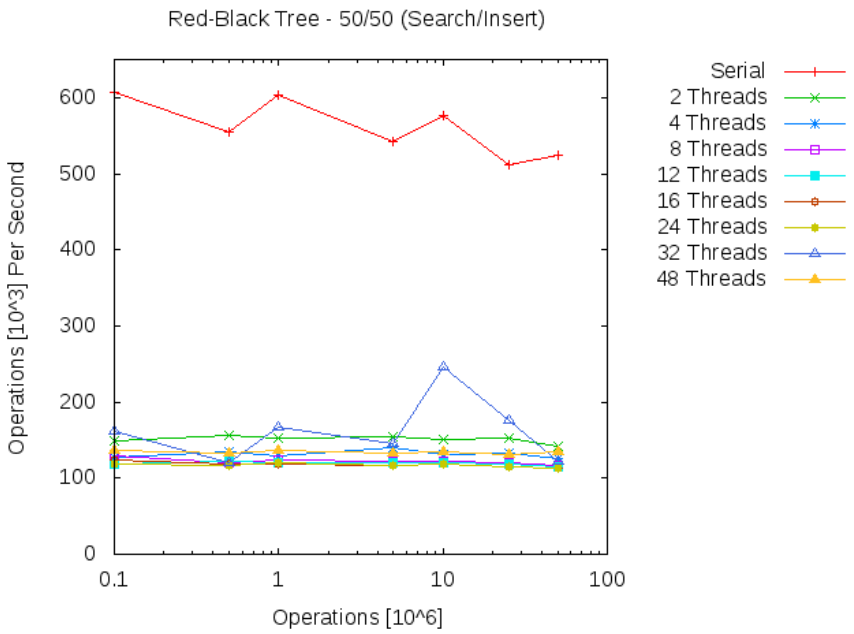
```
void rbtree_insert() {
    /* Lock the whole tree */
    pthread_rwlock_wrlock(&rwlock);
    ...
    /* Unlock the tree */
    pthread_rwlock_unlock(&rwlock);
}
```

Red-Black Tree – Performance of coarse-grain locking

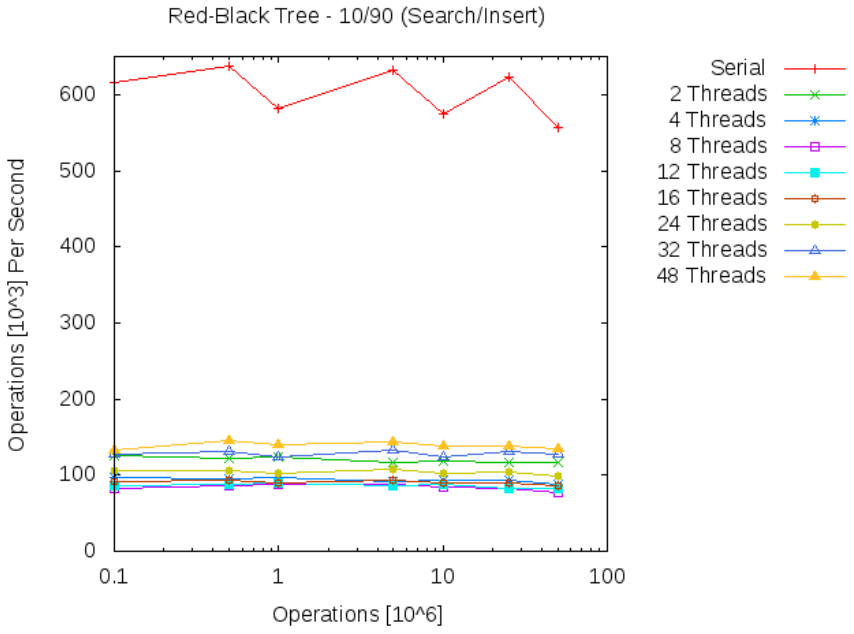
The performance analysis of our parallel implementation show quite bad results. We don't reache nearly the performance of our sequential implementation. This is not surprising, if we consider that only one thread at a time is able to performe an add-operation, whereas all other threads spin on the lock and do nothing. Due to that, the parallel implementation will be performed sequential with additional overhead for parallelisation and lock-spinning.



Surprisingly, we get the best results for a ratio of 90/10 with a thread count of 2 and 32. All other thread counts seem to be quite equal.



With increasing add-operation, the sequential part becomes bigger and it doesn't matter how many threads we use.

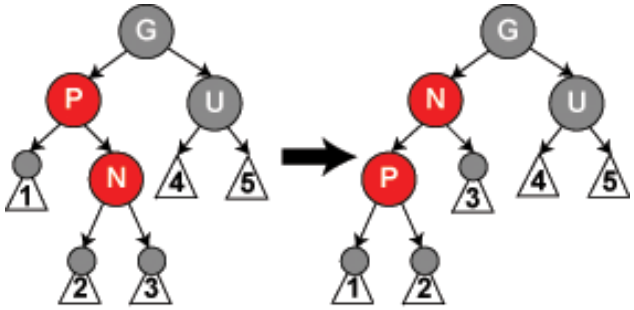


The only adventage of the parallel implementation seems to be the good scalability, since we don't get performance losses with increasing operation numbers.

Red-Black Tree – Considerations for fine-grain Locking

The basic idea of our fine-grain locking considerations is to store a lock in each node so we can lock only certain parts of the tree. If we performe a search operation we would simply read-lock every involved node, which allows us to search concurrent and ensure correctness of the data.

When inserting a new value it is going to be a little more complicated. In our implementation, we first insert a new node into the tree as we would into an ordinary binary search tree. So far we would just need to read-lock as long as we find the right place and then write-lock the destination node. The problem is that the resulting tree may not satify our five red-black tree properties, so we call to insert_case1() function which begins the process of correcting the tree so that it satifies the properties once more.
Our five insert cases need to access and modify the parent-, grandparent-, uncle-, left- and right-node like in the picture shown below.



Thus, we could implement a function which write-locks the whole family which is called if we call our insert_case functions. In the case of an insert conflict only one thread can access this family nodes whereas the others spin.
A huge disadvantage of this implementation would be the big memory amount which is needed to store a lock in each node. Also, there will be spent a lot of time with lock creation and spinning, but the amount of concurrency should strongly increase.