Advanced Parallel Computing: Exercise #4

Due on Tuesday, May 19, 2015

Svend Dorkenwald, Günther Schindler

Inhaltsverzeichnis

Reading

Efficient Synchronization: Let Them Eat QOLB

In the paper 'Efficient Synchronization: Let Them Eat QOLB' the authors analyse the effectivity of synchronization primitives which enable exclusive access to shared data and critical sections of code. They show the five sources of overhead that locking synchronization primitives can incur and describe four mechanisms (local spinning, queue-based locking, collocation, and synchronized prefetch) that reduce these synchronization overheads. Instead of focusing on the individual latencies associated with mutually exclusive accesses to critical sections, they focused on the global throughput of critical section accesses.
The scientists expose that local spinning consistently aids performance but not very much, queue-based locking was very effective, collocation of the lock and locked data in the same cache line showed wildly different - depending on the benchmark - effects, and synchronized prefetching is the least effective of any. Their most important result is the consistent and large performance gain that the locking construct QOLB achieves, which is further increased by collocation. Finally, they claim that the inherent cost requirements of QOLB are not prohibitive.
In our view the authors give with their results a nice overview of the performance of different locking constructs as well as various combinations with the four improvement mechansims. Their work is of high importance because high latency in sequential synchronizations can easily lead into the bottleneck of high performance applications. Therefore, we strongly accept this paper.

GLocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs

Abellan, Fernandez and Acacio propose a new hardware-supported method for thread synchronization on chip-multiprocessors (CMPs). Since the current trend points towards more and more cores on one die (eg. Intel Knights Bridge) thread synchronization on one die becomes increasingly important.
Their approach requires hardware implemented G-Lines for communication between controllers at each core. Logically they form a tree of depth two for a 2D-mesh leading to only four cycles for Acquiring a lock (worst-case). Thereby, the number of required G-Lines only grows linear with the number of cores making it also suitable of larger many-core dies in terms of hardware implementation. Performance-wise they proof to be ahead of the curve by running micro-benchmarks and also real-world applications.
Nevertheless, it remains unclear how their method performs for a huge number of cores on one die (1 primary lock manager). We agree that on-chip communication requires hardware-support to be efficient, but expect it to be user-friendly. Whether their proposed method is worse or better than other methods will be shown as soon as it is or is not adapted by one of the manufacturers.
We accept this paper.

List based QLocks (aka MCS Locks)

Following the lock implementations from the last exercise we implemented the MCS lock algorithm from Mellor-Crummey and Scott (Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, 1991). The advantage of this implementation is that every thread spins on its own variable while only the demanded amount of variables are created. This lowers traffic due to cache coherence and only allocates a limited amount of memory. Mellor-Crummey and Scott accomplish this by ordering the lock-takers where each of them frees the next thread when finishing his job.
Following the datastructure sQnode_t is shown which is created by each thread. It simply contains a variable lock to spin on and a pointer to the next thread.

```
typedef struct sQnode_t sQnode_t;
struct sQnode_t {
    struct sQnode_t *next;
    int locked;
};
```

Each thread acquires a lock when entering the scene which returns as soon as the thread holds the lock. The critical section here is solved via an atomic operation (xchg_64 == fetch_and_store).

```
static void acquire_lock(lock *l, sQnode_t *thisQnode){
    sQnode_t *prenode;

    thisQnode->next = NULL;
    thisQnode->locked = 1;

    prenode = (sQnode_t*) xchg_64(l, thisQnode);

    if (!prenode) return;

    prenode->next = thisQnode;

    barrier();

    while (thisQnode->locked) cpu_relax();

    return;
}
```

After finishing its job a thread passes the lock to the next thread in line (if available). The critical section in this function is also solved via an atomic operation(cmpxchg == compare_and_swap).

```
static void release_lock(lock *l, sQnode_t *thisQnode){
    if (!thisQnode->next){
        if (cmpxchg(l, thisQnode, NULL) == thisQnode) return;

        while (!thisQnode->next) cpu_relax();
    }
    thisQnode->next->locked = 0;
}
```

Lock performance analysis

We compared our results to those from the last exercise, but recognized differences of a factor of 10^3 (MCS is that slower). This may be due to different programs and languages (C, C++ with C stuff (pthreads). We showed our results anyway in Abbildung 1 and visualized them in Abbildung 2. Time(T) is given in mu and the update-rate(U) in $10^9/s$ dependent on the Thread Count (TC)

As expected the MCS algorithm shows a comparably constant performance with larger TC (see also Abbildung 2). However, this may only be due to the fact that our maximum TC equals the number of cores (at least virtual ones) in our system. Since it is a fair algorithm it will suffer from inactive threads (system scheduler) which leads to a significant decrease in performance for TC >> number of cores.

TC	T[Mutex]	U[Mutex]	T[Atomic]	U[Atomic]	T[RMW]	U[RMW]	T[MCS]	U[MCS]
1	147	1.36	48	4.15	89	2.24	6413	0.03
2	314	0.64	82	2.43	1260	0.16	117000	0.002
4	1114	0.18	174	1.15	1494	0.13	76320	0.003
8	318	0.63	180	1.11	2706	0.07	76670	0.003
12	357	0.56	222	0.9	3781	0.05	71630	0.003
16	315	0.63	187	1.07	5051	0.04	72400	0.003
24	334	0.60	167	1.19	7281	0.03	125000	0.002
32	369	0.54	222	0.9	23962	0.01	134000	0.001
40	350	0.57	227	0.88	11956	0.02	461000	0.001
48	355	0.56	212	0.94	22812	0.01	3970000	0.0005

Abbildung 1: Performance comparison for different lock implementations (shared counter test)

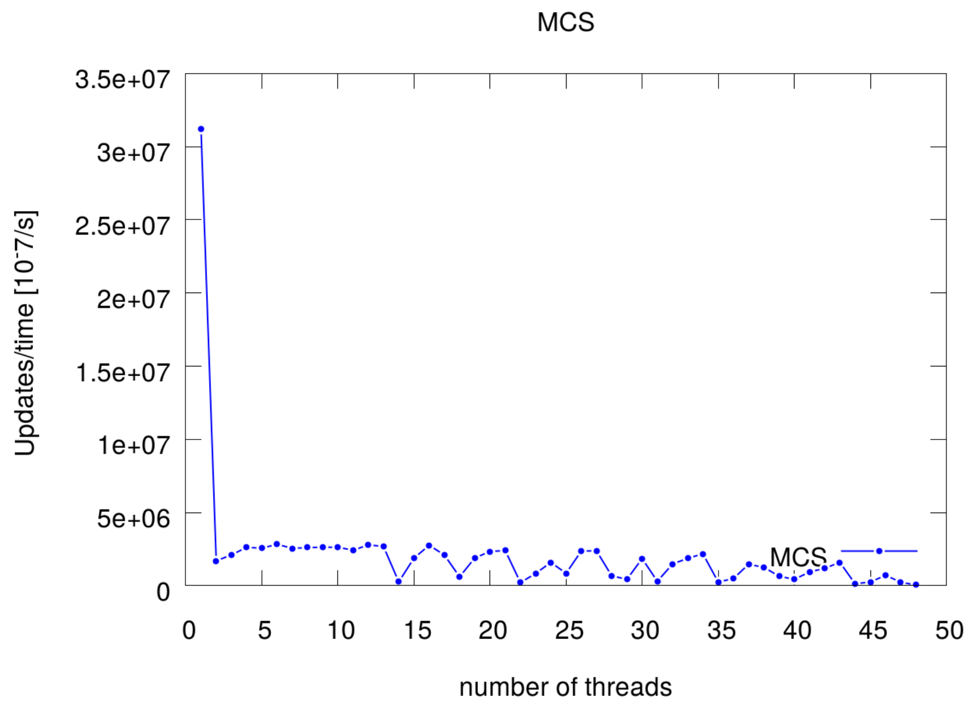

Abbildung 2: Update-rate as function of TC for MCS