Advanced Parallel Computing: Exercise #3

Due on Tuesday, May 12, 2015

Svend Dorkenwald, Günther Schindler

Inhaltsverzeichnis

Reading

Cost-Effective Parallel Computing

The authors David A. Wood and Mark D. Hill show in their paper 'Cost-Effective Parallel Computing' that parallel systems can be cost-effective at modest speedups when memory cost is significant fraction of system cost. They want to disprove the claim that parallel computing is only worthwile when applications achieve nearly linear speedup. To do so the researcher introduce the costup factor - the parallel system cost divided by uniprocessor cost - and compare it to the classical speedup.
Their results show that parallel computing can cost-effective maximize throughput whenever speedup exceeds costup. Furthermore they expose when memory is sufficiently large, more than one processor may be needed to efficiently utilize the memory.
In our opinion the reasoning of the authors is correct although their results are not groundbreaking. However, they results are solid arguments to disprove Amdahl's Law once again. Thus, we weakly accept this paper.

A Survey of Cache Coherence Schemes for Multiprocessor

In this paper from 1990 Stenström surveys different schemes for maintaining cache coherence which was an upcoming problem back then. The emerging shared-memory multiprocessors showed up to introduce three main problems: Memory contention, Communication contention and latency. Depending on the system architecture (bus, multistage interconnect) one or many of these problems are predominant and tackled with private caches.
The proposed solutions ranged from hardware-implemented protocols, eg. snoopy cache protocols and directory schemes, to software-implemented policies. He evaluated each of them carefully and argued which is best suited for different systems. However, since only snoopy protocols had been implemented it was shear impossible to evaluate those under real-world conditions.
Stenström discusses many aspects of cache coherency, but realizes that much experimental research is missing to rank different methods. Nevertheless, we are impressed by his contribution. Many of the discussed points are still valid today. Thus, we give this paper a strong accept.

Shared Counter

As part of this exercise we should implement a Pthread program which results in a counter. A mutex has to be used to ensure mutual exclustion when incrementing the counter.
The following listening gives the algorithm which we implemented for this.

```c
void vCountMutex(void *arg){
  int i, iID;
  iID = (intptr_t) arg;
  /* Synchronize threads */
  pthread_barrier_wait(&barrier);
  /* Start incrementation process */
  for(i = (iID*iC)/iN; i < ((iID+1)*iC)/iN; i++){
    /* Lock mutex */
    pthread_mutex_lock(&mutex);
    /* Increment count variable */
    vInc(&iCount);
    /* Unlock mutex */
    pthread_mutex_unlock(&mutex);
  }
}
```

Depending on the thread-ID (arg) every thread increments the count variable until the variable reaches the C value. If we count without the mutual exclusion, data races result in a mismatch between the counter variable and the C value.

Shared Counter Revisited

Next, we should implement two alternative counter update methods. First, we use the atomic operation __sync_add_and_fetch().

```c
void vCountAtom(void *arg){
  int i, iID;
  iID = (intptr_t) arg;
  /* Synchronize threads */
  pthread_barrier_wait(&barrier);
  /* Start incrementation process */
  for(i = (iID*iC)/iN; i < ((iID+1)*iC)/iN; i++){
    /* Increment count variable */
    __sync_add_and_fetch(&iCount,1);
  }
}
```

As we can see, with the atomic operation there is no need for locking like the PThread mutex anymore. This is because the fetch-and-add instruction allows any processor to automatically increment a value in memory, preventing multiple processor collisions.

Based on this operation we also implemented our own locking mechanism as followed:

```c
typedef struct {
  int ticket;
  int users;
} lock_t;

static void lock_rmw(lock_t *t){
  int me = __sync_add_and_fetch(&(t->users), 1);
  while (t->ticket != me)
    cpu_relax();
}

static void unlock_rmw(lock_t *t){
  __sync_add_and_fetch(&(t->ticket), 1);
}
```

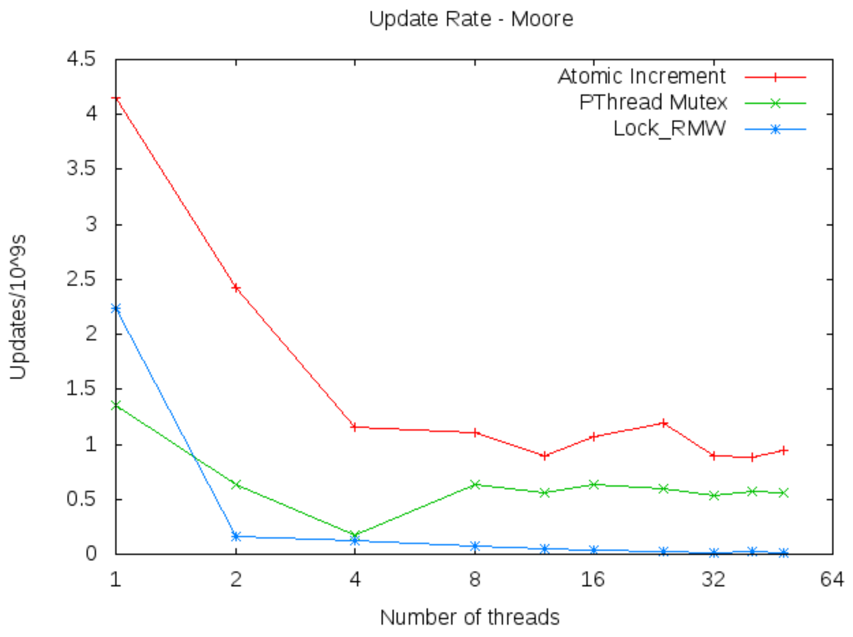
In this case we need a atomic operation because this CPU instruction automatically modifies the contents of a memory location instead of the cached value.

Shared counter performance analysis

As part of this exercise we should perform a performance analysis of the three different shared counter implementations. Following table shows the results of this performance analysis. Time is given in us and the update-rate is given in 10^9/s.

Thread Count	Time[Mutex]	Updates[Mutex]	Time[Atomic]	Updates[Atomic]	Time[RMW]	Updates[RMW]
1	147	1.36	48	4.15	89	2.24
2	314	0.64	82	2.43	1260	0.16
4	1114	0.18	174	1.15	1494	0.13
8	318	0.63	180	1.11	2706	0.07
12	357	0.56	222	0.9	3781	0.05
16	315	0.63	187	1.07	5051	0.04
24	334	0.60	167	1.19	7281	0.03
32	369	0.54	222	0.9	23962	0.01
40	350	0.57	227	0.88	11956	0.02
48	355	0.56	212	0.94	22812	0.01

The derived number of updates per seconds dependent on the number of threads gives a nice expression about the performance.



As the graphic shows the atomic increment implementation gives the best result. Second is the PThread Mutex implementation. We guess that this gap is due to the high overhead for the Mutex implementation compared to the simplicity of the atomic operation. The worst result is given by our own lock implementation.