Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Understanding and Avoiding Memory Issues with Multi-core Processors

When programming for multiple thread or multiple core systems, it is important to understand memory allocation and access

December 11, 2008
URL:http://www.drdobbs.com/parallel/understanding-and-avoiding-memory-issues/212400410

Shameem Akhter, a platform architect at Intel, and Jason Roberts, a senior software engineer at Intel, are the authors of Multi-Core Programming: Increasing Performance through Software Multithreading on which this article is based. Copyright (c) 2008 Intel Corporation. All rights reserved.

For multi-core programs, working within the cache becomes trickier, because data is not only transferred between a core and memory, but also between cores. As with transfers to and from memory, mainstream programming languages do not make these transfers explicit. The transfers arise implicitly from patterns of reads and writes by different cores. The patterns correspond to two types of data dependencies:

- **Read-write dependency**. A core writes a cache line, and then a different core reads it.
- **Write-write dependency**. A core writes a cache line, and then a different core writes it.

An interaction that does not cause data movement is two cores repeatedly reading a cache line that is not being written. Thus if multiple cores only read a cache line and do not write it, then no memory bandwidth is consumed. Each core simply keeps its own copy of the cache line.

To minimize memory bus traffic, minimize core interactions by minimizing shared locations. Hence, the same patterns that tend to reduce lock contention also tend to reduce memory traffic, because it is the shared state that requires locks and generates contention. Letting each thread work on its own local copy of the data and merging the data after all threads are done can be a very effective strategy.

Consider writing a multi-threaded version of the function **CacheFriendlySieve**. A good decomposition for this problem is to fill the array **factor** sequentially, and then operate on the windows in parallel. The sequential portion takes time **O()**, and hence has minor impact on speedup for large **n**. Operating on the windows in parallel requires sharing some data. Looking at the nature of the sharing will guide you on how to write the parallel version.

- The array **factor** is read-only once it is filled. Thus each thread can share the array.
- The array **composite** is updated as primes are found. However, the updates are made to separate windows, so they are unlikely to interfere except at window boundaries that fall inside a cache line. Better yet, observe that the values in the window are used only while the window is being processed. The array **composite** no longer needs to be shared, and instead each thread can have a private portion that holds only the window of interest. This change benefits the sequential version too, because now the space requirements for the sieve have been reduced from **O($^{\mathbf{n}}$)** to **O()**. The reduction in space makes counting primes up to 10^{11} possible on even a 32-bit machine.
- The variable **count** is updated as primes are found. An atomic increment could be used, but that would introduce memory contention. A better solution, as shown in the example, is to give each thread perform a private partial count, and sum the partial counts at the end.
- The array **striker** is updated as the window is processed. Each thread will need its own private copy. The tricky part is that striker induces a loop-carried dependence between windows. For each window, the initial value of striker is the last value it had for the previous window. To break this dependence, the initial values in **striker** have to be computed from scratch. This computation is not difficult. The purpose of **striker[k]** is to keep track of the current multiple of **factor[k]**.
- The variable base is new in the parallel version. It keeps track of the start of the window for which **striker** is valid. If the value of base differs from the start of the window being processed, it indicates that the thread must recompute **striker** from scratch. The recomputation sets the initial value of **striker[k]** to the lowest multiple of **factor[k]** that is inside or after the window.

Figure 1 shows the multi-threaded sieve. A further refinement that cuts the work in half would be to look for only odd primes. The refinement was omitted from the examples because it obfuscates understanding of the multi-threading issues.

```
long ParallelSieve( long n ) {
    long count = 0;
    long m = (long)sqrt((double)n);
    long n_factor = 0;
    long* factor = new long[m];
#pragma omp parallel
    {
       bool* composite = new bool[m+1];
       long* striker = new long[m];
#pragma omp single
       {
          memset( composite, 0, m );
          for( long i=2; i<=m; ++i )
             if( !composite[i] ) {
                ++count;
                Strike( composite, 2*i, i, m );
                factor[n_factor++] = i;
             }
       }
       long base = -1;
#pragma omp for reduction (+:count)
       for( long window=m+1; window<=n; window+=m ) {
          memset( composite, 0, m );
          if( base!=window ) {
             // Must compute striker from scratch.
             base = window;
             for( long k=0; k<n_factor; ++k )
                striker[k] = (base+factor[k]-1)/factor[k] *
```

```
                        factor[k] - base;
            }
            long limit = min(window+m-1,n) - base;
            for( long k=0; k<n_factor; ++k )
               striker[k] = Strike( composite, striker[k],
                                    factor[k], limit ) - m;
            for( long i=0; i<=limit; ++i )
               if( !composite[i] )
                  ++count;
               base += m;
         }
         delete[] striker;
         delete[] composite;
      }
      delete[] factor;
      return count;
}
```

Figure 1: Parallel Sieve of Eratosthenes

Cache-related Issues

As remarked earlier in the discussion of time-slicing issues, good performance depends on processors fetching most of their data from cache instead of main memory. For sequential programs, modern caches generally work well without too much thought, though a little tuning helps. In parallel programming, caches open up some much more serious pitfalls.

False Sharing

The smallest unit of memory that two processors interchange is a cache line or cache sector. Two separate caches can share a cache line when they both need to read it, but if the line is written in one cache, and read in another, it must be shipped between caches, even if the locations of interest are disjoint. Like two people writing in different parts of a log book, the writes are independent, but unless the book can be ripped apart, the writers must pass the book back and forth. In the same way, two hardware threads writing to different locations contend for a cache sector to the point where it becomes a ping-pong game.

Figure 2 illustrates such a ping-pong game. There are two threads, each running on a different core. Each thread increments a different location belonging to the same cache line. But because the locations belong to the same cache line, the cores must pass the sector back and forth across the memory bus.

[Click image to view at full size]

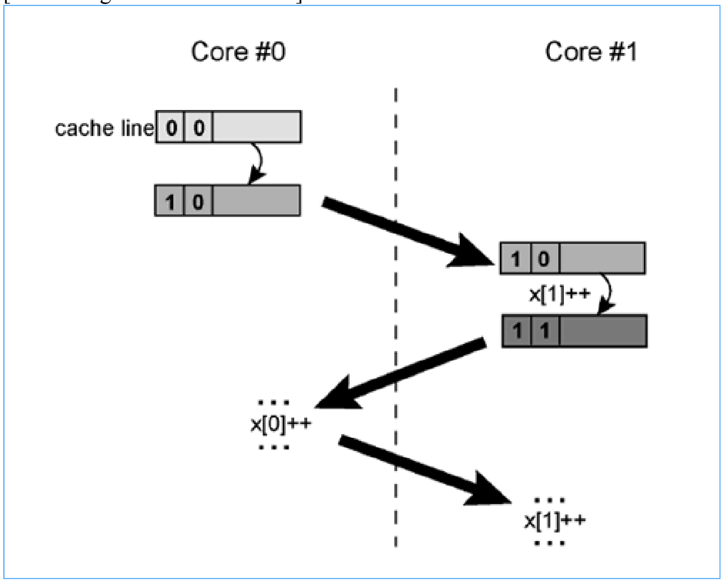

Figure 2: Cache Line Ping Ponging Caused by False Sharing

Figure 3 shows how bad the impact can be for a generalization of Figure 2. Four single-core processors, each enabled with Hyper-Threading Technology (HT Technology), are used to give the flavor of a hypothetical future eight-core system. Each hardware thread increments a separate memory location. The **ith** thread repeatedly increments **x[i*stride]**. The performance is worse when the locations are adjacent, and improves as they spread out, because the spreading puts the locations into more distinct cache lines. Performance improves sharply at a stride of 16. This is because the array elements are 4-byte integers. The stride of 16 puts the locations 16 W 4 = 64 bytes apart. The data is for a Pentium 4 based processor with a cache sector size of 64 bytes. Hence when the locations were 64 bytes part, each thread is hitting on a separate cache sector, and the locations become private to each thread. The resulting performance is nearly one hundredfold better than when all threads share the same cache line.

[Click image to view at full size]

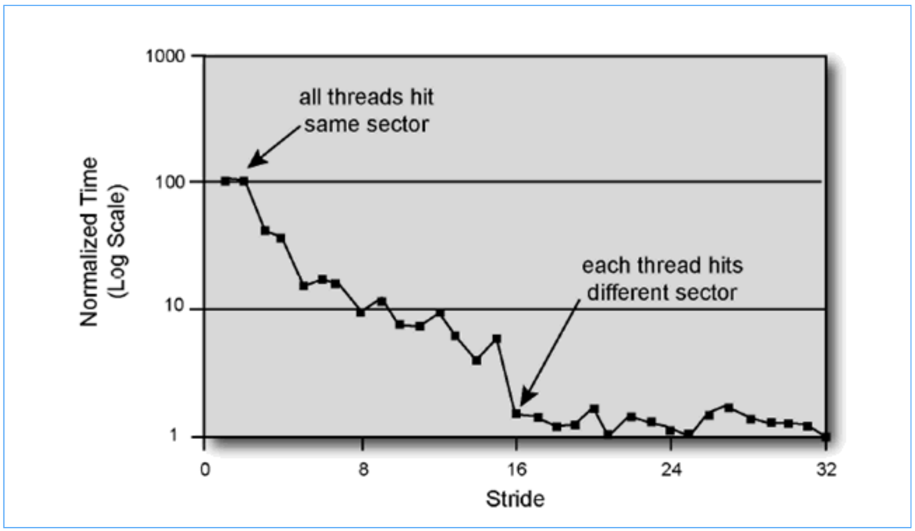

Figure 3: Performance Impact of False Sharing

Avoiding false sharing may require aligning variables or objects in memory on cache line boundaries. There are a variety of ways to force alignment. Some compilers support alignment pragmas. The Windows compilers have a directive **__declspec(align(n))** that can be used to specify *n*-byte alignment. Dynamic allocation can be aligned by allocating extra pad memory, and then returning a pointer to the next cache line in the block. Figure 4 shows an example allocator that does this. Function **CacheAlignedMalloc** uses the word just before the aligned block to store a pointer to the true base of the block, so that function **CacheAlignedFree** can free the true block. Notice that if **malloc** returns an aligned pointer, **CacheAlignedMalloc** still rounds up to the next cache line, because it needs the first cache line to store the pointer to the true base.

It may not be obvious that there is always enough room before the aligned block to store the pointer. Sufficient room depends upon two assumptions:

- A cache line is at least as big as a pointer.
- A **malloc** request for at least a cache line's worth of bytes returns a pointer aligned on boundary that is a multiple of **sizeof(char*)**.

These two conditions hold for IA-32 and Itanium-based systems. Indeed, they hold for most architecture because of alignment restrictions specified for **malloc** by the C standard.

```
// Allocate block of memory that starts on cache line
void* CacheAlignedMalloc( size_t bytes, void* hint ) {
    size_t m = (cache line size in bytes);
    assert( (m & m-1)==0 ); // m must be power of 2
    char* base = (char*)malloc(m+bytes);

    // Round pointer up to next line
    char * result = (char*)((UIntPtr)(base+m)&-m);

    // Record where block actually starts.
    ((char**)result)[-1] = base;

    return result;
}
// Free block allocated by CacheAlignedMalloc
void CacheAlignedFree( void* p ) {

    // Recover where block actually starts
    char* base = ((byte**)p)[-1];
    // Failure of following assertion indicates memory
    // was not allocated with CacheAlignedMalloc.
    assert( (void*)((UIntPtr)
            (base+NFS_LineSize)&-NFS_LineSize) == p);
    free( base );
}
```

Figure 4: Memory Allocator that Allocates Blocks Aligned on Cache Line Boundaries

The topic of false sharing exposes a fundamental tension between efficient use of a single-core processor and efficient use of a multi-core processor. The general rule for efficient execution on a single core is to pack data tightly, so that it has as small a footprint as possible. But on a multi-core processor, packing shared data can lead to a severe penalty from false sharing. Generally, the solution is to pack data tightly, give each thread its own private copy to work on, and merge results afterwards. This strategy extends naturally to task stealing. When a thread steals a task, it can clone the shared data structures that might cause cache line ping ponging, and merge the results later.