Atomic Memory Ops in UPC

Phil Merkey

May 18, 2005

Abstract

Atomic Memory Operations may make UPC even more productive. UPC allows one to write fine grained applications without worrying about organizing the communication into large messages and computation into large blocks. The use of AMOS along with the relaxed memory model in UPC may help eliminate the need for fine grained synchronization in such algorithms.

1 Introduction

There is a great body of theoretical work on Atomic Memory Operations. Much of it is from the 1980's when researchers were considering MIMD programming models with balanced architectures and large numbers of processors. There has also been some recent activity and in the literature. Most of this work concerns building complex shared data structures. The introduction of the X1 and its intrinsic AMOs has motivated the study of using AMOs as a means to reduce the need for synchronization in UPC.

2 Background

Atomic Memory Operations are formally defined as operations with the property that the operation can only be observed in the state before the operation began or in the state after the operation has completed. Note that this does not imply that AMOs must occur instantaneously nor does it imply that multiple threads need observe the operation in the same state.

The standard list of AMOS includes operations of the following types:

- `link load` / `store conditional` This is really a pair of operations. There are found on all modern processors as they are required for operating system code. This pair provides a primitive AMO in the sense that all others can be implemented using them. The idea is that one can issue a `link load` to load memory location and link it to some flag. If anything else touches the memory location in question that link is broken. The

`store conditional` then refers to this link and if the link is broken the store fails. In this way the operation can be repeated until we are certain that nothing has observe the operation in an intermediate state of the load modified store operation in question. This operation is too low level for most application codes and there are usually only a links that can be set at one time.

- `compare and swap` This is similar to `link load/store conditional`. This is operations takes three arguments: the address of the memory cell, a compare value, and replace value. The precise semantics, what is returned and precisely what arguments are allowed varies from one implementation, but the idea is the same. The `compare and swap` operation checks that the compare is equal to the contents of the memory and if so atomically swaps the contents with the replace value, if the comparison fails the swap does not occur. This is primary AMO used in building shared data structures. We will indicate how to use it to add a node into a link list in the next section.

- *fetch and add* This is the best known operations in the class of computational AMOs. This operation takes two arguments, the address of the memory cell in question and a value that is to be added to the contents of that memory cell. The fetch and add returns the contents of the memory location before the add occurred.

3 Availability

The machines that run UPC and have AMOs available to application programmers are limited and diverse. The only ones known to the author are the T3E, X1 and MuPC running on Beowulf clusters. On the T3E we have `upc_fetch_add` and the atomic add `upc_aadd`. The X1 provides:

- `_amo_aadd (&p, v);` an atomic `long int` add,
- `_amo_afadd (&p, v);` an atomic `long int` fetch and add,
- `_amo_aax (&p, A, X);` an atomic `long int`, `and` and `xor`,
- `_amo_afax (&p, A, X);` an atomic `long int`, fetch and `and` and `xor`,
- `_amo_acswap(&p, C, R);` an atomic `long int`, compare and swap.

The UPC implementation for Beowulf clusters developed at Michigan Tech uses two pthreads per UPC thread. One of the pthreads runs the UPC program, the other runs the *runtime system*. The runtime system is the layer of software that provides the illusion of shared memory. This system was developed as a research tool. It is highly portable and runs on affordable machines. There is no misunderstanding about the performance potential of the this setup. Programs written the usual UPC style, with random single word shared memory references generate lots of small gets and puts to memory all across the machine.

Beowulf clusters don't do this very well. Much of the research so far with UPC on machines with NIC based communications has revolved around using the relaxed memory model to enable caching non-affinity references. This should lead to message passing kind of behavior without requiring the programmer to explicitly form the messages.

The current implementation of AMOs within the MuPC is aimed at providing algorithm developer the ability to experiment with AMOs. We have copied the APIs and semantics of those provided by the X1 possible and have expanded the list to include:

- `upc_faop(opcode, &p, v)` atomic fetch and operation, where the operation can be add, multiple, or xor,
- `upc_cas(&p, c, r)` atomic compare and swap,
- `upc_dcas(&p, c1, r1, c2, r2)` atomic double compare and swap,
- `upc_maskswap(&p, v, m)` atomic swap bit under a mask,
- `upc_ffaop(opcode, &p, v)` floating point atomic fetch and op.

4 AMOs verses Locks

The UPC spec provides a locking mechanism that can be used to obtain behavior similar to AMOs. Suppose we want to use `compare_and_swap` on elements in the array, `array[]`. If we allocate a corresponding array of locks, `arraylock[]`, then we could use the following code instead of `a = upc_cas(&array[k],c,r);`

```
upc_lock(&arraylock[k]);
  a = array[k];
  if( a == c )
     array[k] = r;
upc_unlock(&arraylock[k]);
```

This technique is not very elegant but it does work at the cost of extra memory. Furthermore the memory overhead can be reduced by using a smaller table of locks and hashing the references into that smaller table. This trades memory for contention rate. We have found that we get reasonable performance if we use about as many locks as there are threads.

Even though no current compiler takes advantage of it, there is a technical reason why we should continue to think of AMOs as being different than the above locking code. All shared references in UPC are either *strict* or *relaxed*. Roughly speaking, strict references follow the semantics of sequential consistency and relaxed references follow weak consistency semantics. Being a mixture of these two models, the UPC memory model gives the programmer the control of sequential consistency with respect to certain variables and the ability to express the order independence of others.

One working definition of strict and relaxed is the following. References that are strict must be appear to be preformed in program order. This affectively disables any ILP for code

involving strict references. Weak references only have to follow the usual serial C program constraints and additional constraint that order of weak references can not appear to move with respect to the strict references in that thread. The semantics of a lock require an implicit strict null reference before the `upc_lock` and after the `upc_unlock`. Therefore locks should be reserved for critical sections of code and should not be used as part of the memory model.

It is important to keep in mind that the purpose of this research is to help determine the potential of using AMOS in UPC, not necessarily obtaining the highest performance with the current implementations.

5 Overhead of Atomic Memory Operations

It is difficult to evaluate the overhead associated with atomic memory operations. There is no way to get a head to head comparison because no two machines implement the same operations in the same way. To get a sense of how the operations work we implemented a simple loop that adds one to random locations in an array. By varying the size of the array and the number of threads writing to the array we can control the content rate.

We have developed the following intuition about `fetch_and_add` on the T3E and the X1. The implementation of UPC on the T3E uses `fetch_and_add` to implement `upc_lock` and remote references are not cached, so there is little surprise in the behavior of the T3E. For low contention rates a `fetch_and_add` costs about twice as much as a regular add and using a lock as in the previous section costs about 3 to 4 times as much. At high contention rates, say 32 threads all trying to update a single value, the cost of a `fetch_and_add` is about 5 times that of a regular add and the cost of using locks is about 20 times a simple add.

On the X1, the AMOs don't vectorize so it is not a fair comparison to measure the speed of the AMO to a vectorized add operation. The atomic add instruction is "fire and forget" operation so it takes about half the time of a fetch and add. The lock construct about was about 5 times slower than an atomic add with no contention and about 100 times slower at high contention rates. Why this slow down is so much worst then the T3E is not yet known.

6 Applications

To demonstrate how AMOs can be used to reduce the need for locks and synchronization we will consider an implementation of a Particle in Cell code. This code has two dominate data structures. The force field is held as a multi-dimensional grid and the particle that move the this field are kept as list or array. At the highest level the code implements the following loop

```
loop over the time steps
   move the particles
   interpolate the new particle positions
   update the force field
```

The challenging thing from a parallelization point of view is that the force calculation would imply a regular data decomposition and then one would have to decompose the list of particles to fix that decomposition. As the simulation proceeds the particle move and eventually they lose affinity with the cell actually contain them. The code now becomes

```
loop over the time steps
   move the particles
   adjust affinity
   interpolate the new particle positions
   update the force field
```

In a message passing code one would have to send messages to update the new particle information. In UPC, one can ignore this lack of affinity, at the cost of performance, and run the loops without alternation. If one wants to adjust the affinity of the list of particle one can use a `fetch_and_add` trick to preform an insertion sort without locks. Begin by allocating a counter `next_entry[t]` for each thread. Suppose there is enough room for K particles to have affinity to each thread.

Then the code to adjust affinity looks like

```
for each of my particles, P
   n_T = the desired affinity for particle P
   new_index[P] = amo_afadd( &next_entry[n_T], 1 );
   if( new_index[P] > K )
      then try each thread
         new_index[P] = amo_afadd( &next_entry[next_thread], 1 );
      until new_index is in bounds
   write the particle information into new list at new_index[P]
```

After this reservation trick of using the atomic counter, one still has to decide exactly how to execute the realignment of the affinity of the particles. One could do it as described above, or one could brake the loop into two loops, so that moving the particles was a separate or a call to the UPC collective `upc_permute`. The important thing to note is that the use of AMO has made a reservation for each of the particles in an asynchronous way and has established a permutation of indices so the realignment can happen in a conflict free way.

There is another obvious place in the PIC code that would benefit from an AMO. Once the particles have been moved, their new positions are input for the next force calculation. This is the interpolation step. Each particle contributes an interpolated amount to the corners of the cell that contains it. Trying to update the grid in parallel leads to a race conditions when whenever two particles share the same corner. The usual solution the this problem is to parition the corners so that they can be updated with conflicts. If we had atomic floating point operations the race condition would be resolved and the pseudo code would be a simple as

```
Fract(corner, particle) = the contribution of particle to the given corner

for each particle, P
  ffaadd( Grid[nw], Fract(P) );
  ffaadd( Grid[ne], Fract(P) );
  ffaadd( Grid[se], Fract(P) );
  ffaadd( Grid[s], Fract(P) );
```

Of course, since atomic float point operations don't exist on any real hardware this point is mostly academic. We plan continue to use MuPC on our Beowulf cluster to examine the use of AMOs including float point AMOs to evaluate their impact on UPC as a high productivity language.