Atomics in UPC

A memory operation is atomic in case the only observable states for the memory are the state before the operation began or the state after the operation has completed.

It is as if the operation happens at the memory location, but legally we don't care.

Atomic is a funny word:

can't be split? small? very powerful!

Definition

- Link Load / Store Conditional
- Fetch_and_Add
  - afadd(&p, v)
  - return p, set p = p + v
- Compare and Swap
  - acswap(&p, cmp, replace)
  - return p, if(cmp==p) { p = replace }

Link Lists

Standard use of compare and swap

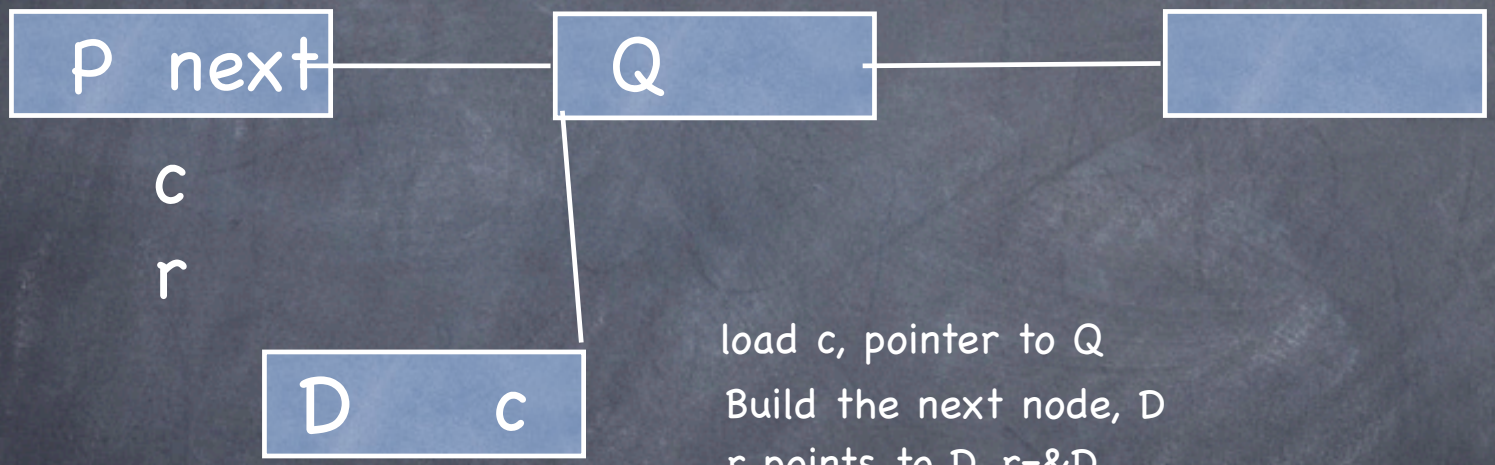
Link Lists

Standard use of compare and swap

load c, pointer to Q

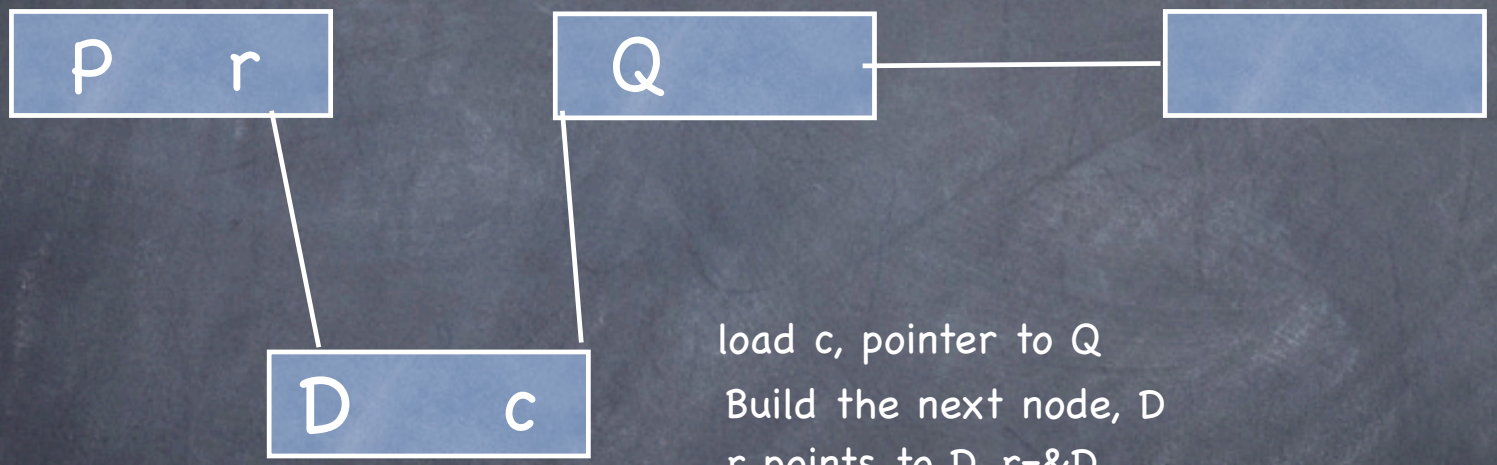
Link Lists

Standard use of compare and swap



Link Lists

Standard use of compare and swap



load c, pointer to Q

Build the next node, D

r points to D, r=&D

cswap(&next, c, r)

UPC machines with AMOS

- Cray T3E:
  - fetch_and_add(&p,v)
  - cswap(&p,c,r)
- Cray X1:
  - _amo_aadd(&p, v) //atomic add (long v)
  - _amo_afadd(&p,v) // fetch and add
  - _amo_aax(&p, A, X) // p=p&A^X
  - _amo_afax(&p,A,X)
  - _amo_acswap(&p, c, r) //compare and swap

UPC machines with AMOS

- MuPC:
  - _upc_faop(opcode, &p, v)
  - _upc_ffaop(opcode, &p, v)
  - _upc_cas(&p, &c, &r)
  - _upc_casv(&p, c, r)
  - _upc_dcas(&p, &c1, &r1, &q, &c2, &r2)
  - _upc_maskcswap(&p, mask, c, r)MuPC

Why not just use locks?

```
old = _amo_cswap(&A[k], C, R);
```

Is sort of equivalent to:

```
_upc_lock( lockforA[k]);
    old = A[k];
    if(C==old)
        A[k] = R;
_upc_unlock( lockforA[k]);
```

But there are issues.

Why not just use locks?

- Locks are ugly and not cool
- You might need a lock for each A[i]
- Locks have their own semantic issues:
  - T3E locks use cswap (not fifo, not fair)
  - HP has two versions (greedy | fair)
  - X1 ?
  - MuPC tries to minimize messages
- Not addressed by the spec.
- The two sets of issues should be separate.
- There are issues with the memory model.

Why not just use locks?

```
old = _amo_cswap(&A[k], C, R);
```

Is sort of equivalent to:

```
strict null reference
_upc_lock( lockforA[k]);
    old = A[k];
    if(C==old)
            A[k] = R;
_upc_unlock( lockforA[k]);
strict null reference
```

Relative Costs

No Head-to-Head Comparison is Meaningful

- T3E
  - afadd == aadd
  - a afadd is about the same as an add
- X1
  - amos don't vectorize
  - aadd twice as fast as afadd
- MuPC
  - the current caching strategies are disabled

Relative Costs

Random Access Lite:

```
for a long time:
    idx = random % N
    array[idx]++
```

Relative Costs

Pick a modest number of threads 16-64
For array sizes from 1 to 1024

```
loop                                        // baseline
    array[ random ]++
-----------
loop                                        // atomic
    afaad(&array[ random ] , 1)
-----------
loop                                        // locks
    upc_lock( lockarray[ random ] )
    array[random]++
    upc_unlock( lockarray[ random ] )
```

Relative Costs

T3E:

N	updates raw	slow down w/ locks
1024	98%	3x
32	50%	5x
1	20%	20x

afadd is 2x over raw +=

X1:

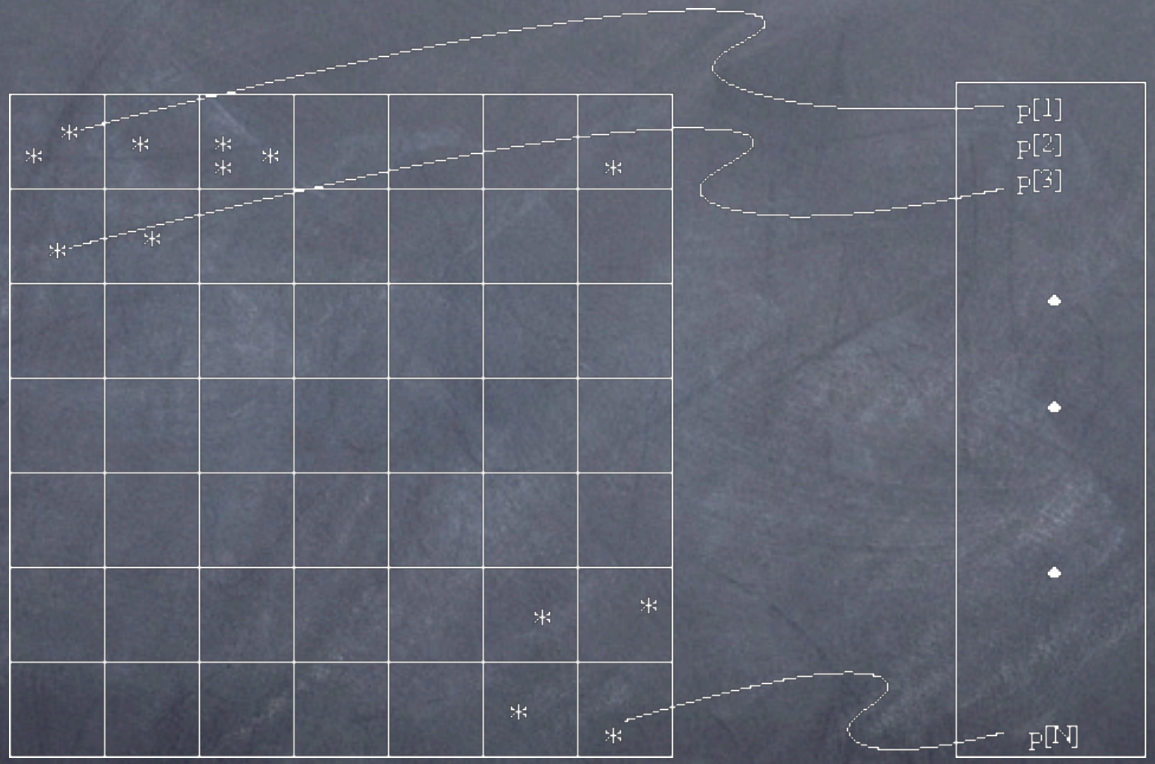
N	updates raw	slow down w/ locks
1024	98%	3x
32	50%	5x
1	20%	100x

amos don't vectorize, afadd is 2x over aadd

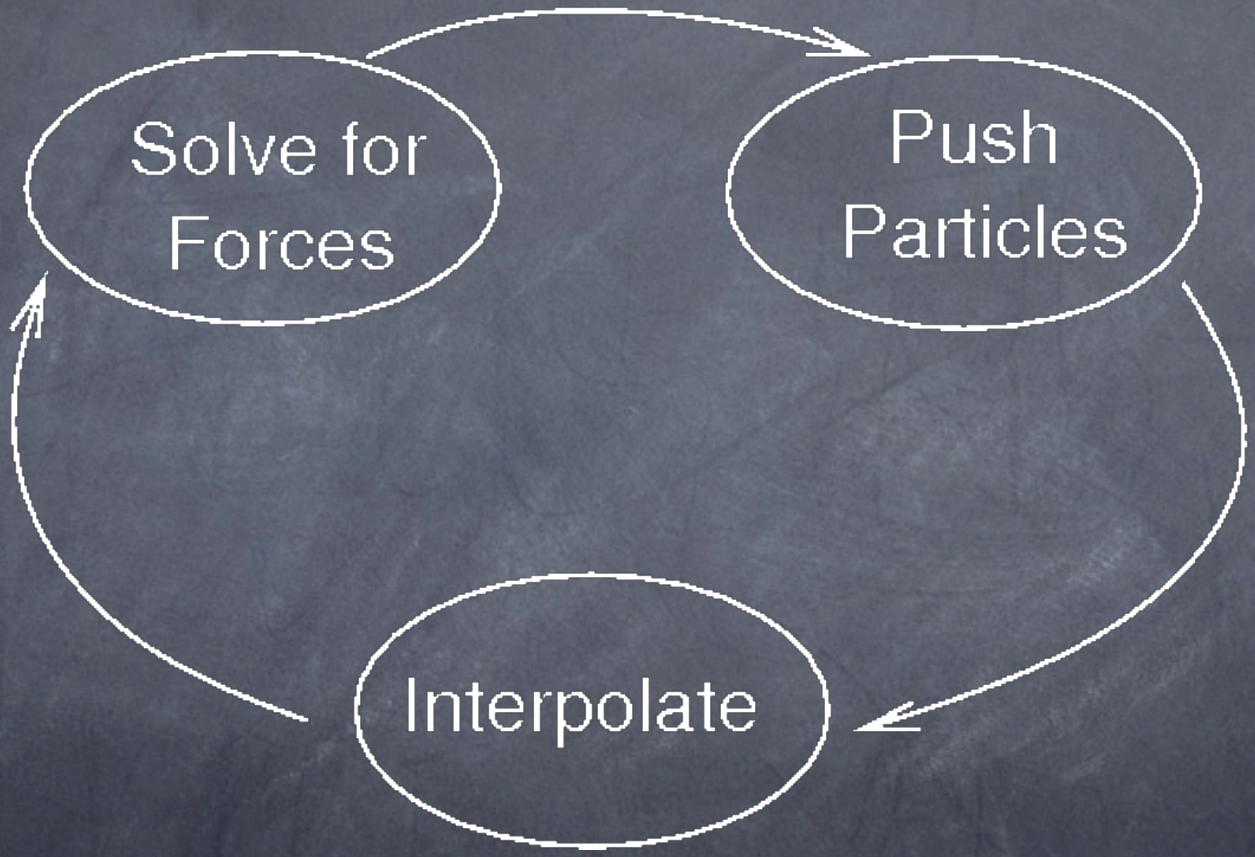
Relative Costs

- HP:
  - strict updates give the same update rates.
  - relaxed references don't.
    - caching effectively does the "sort trick"
    - but then the conflict rate is high at the end
- MuPC
  - strict updates give the same update rates.
  - afadd is a lot faster than using locks
  - afadd is single word messages

PIC Code



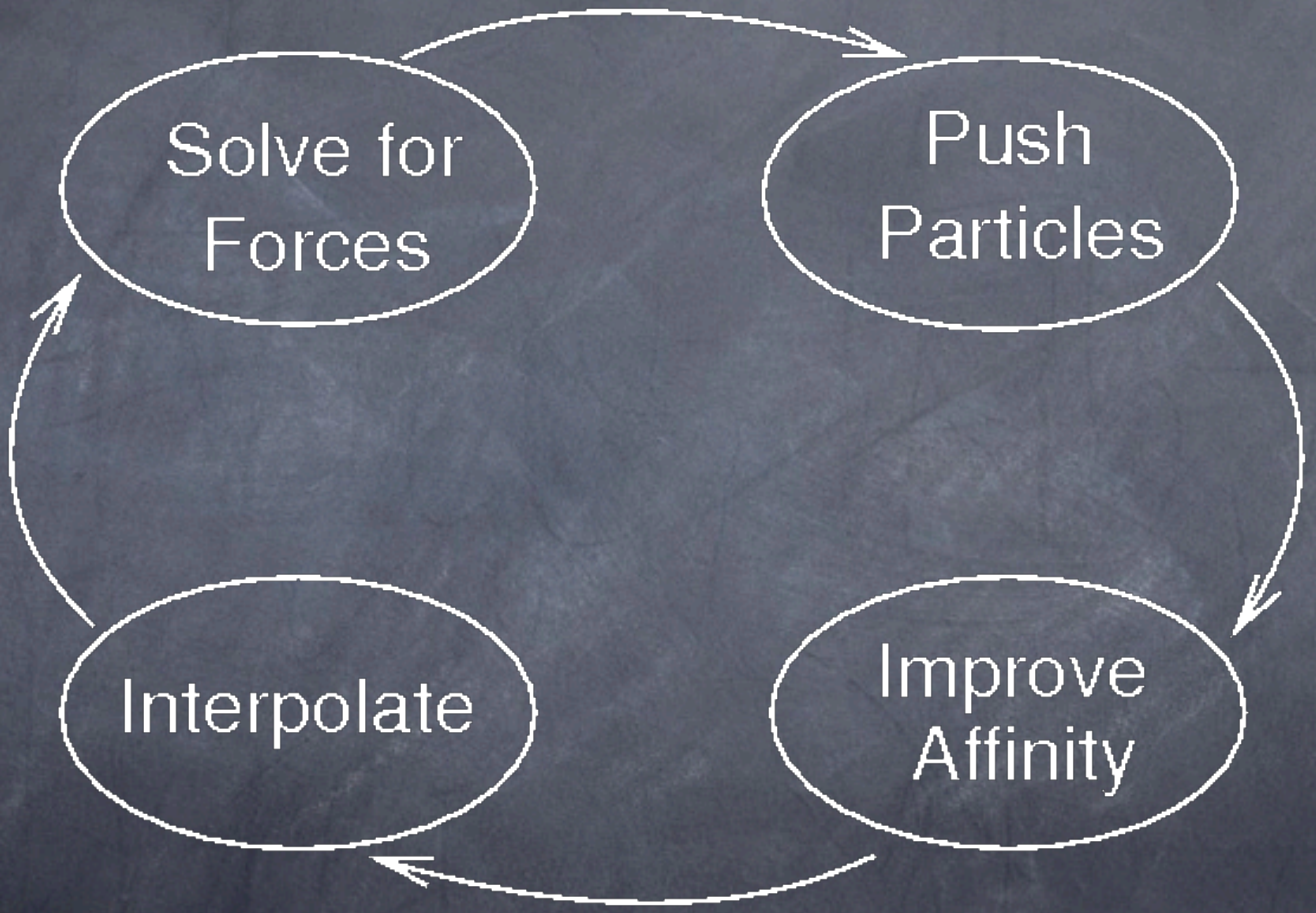
PIC Code



PIC Code

As the particles move their affinity changes

PIC Code


Solve for Forces
Push Particles
Improve Affinity
Interpolate

Improving Affinity

n_T = desired affinity of particle new position

newindex[0:THREADS-1] = counter for each thread

```
for all my particles, P
   n_T =  desired affinity of P
   n_idx[P] = amo_afadd( &newindex[n_T], 1 )
   if out of bounds, change n_T and try again

for all my particles, P
    move P to new home
```

PIC Code

Each particle contributes (mass or charge) to the cell that contains it.

Interpolation

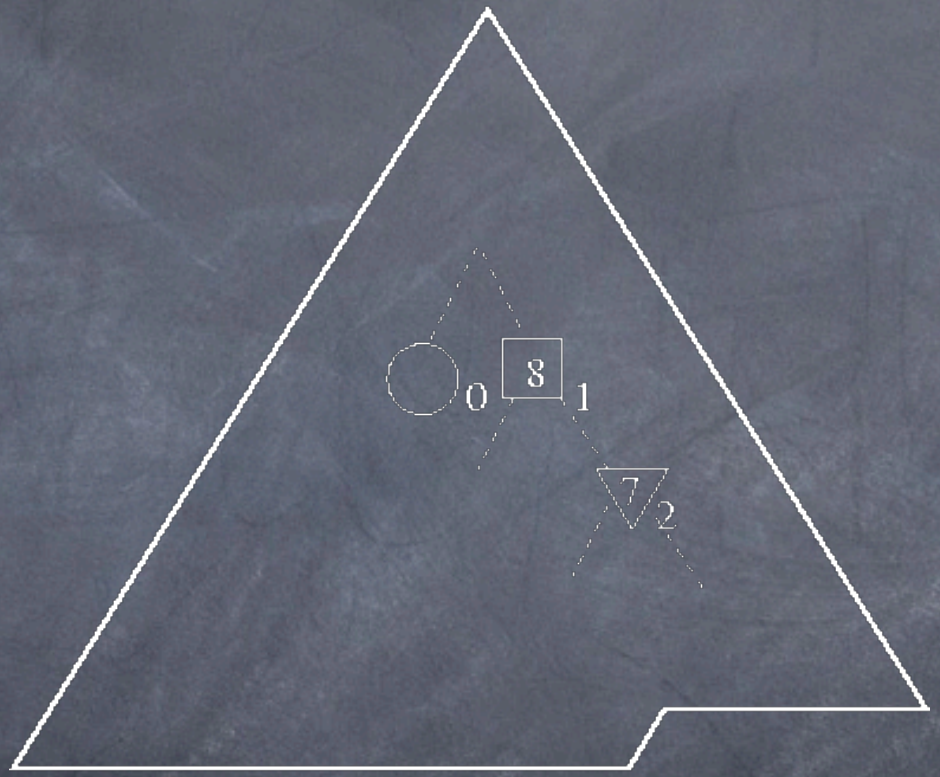
Fract(corner, particle) = the contribution to said corner

for each particle, P
    ffaop(Op, Grid[nw], Fract(nw, P))
    ffaop(Op, Grid[ne], Fract(ne, P))
    ffaop(Op, Grid[sw], Fract(sw, P))
    ffaop(Op, Grid[se], Fract(se, P))

barrier

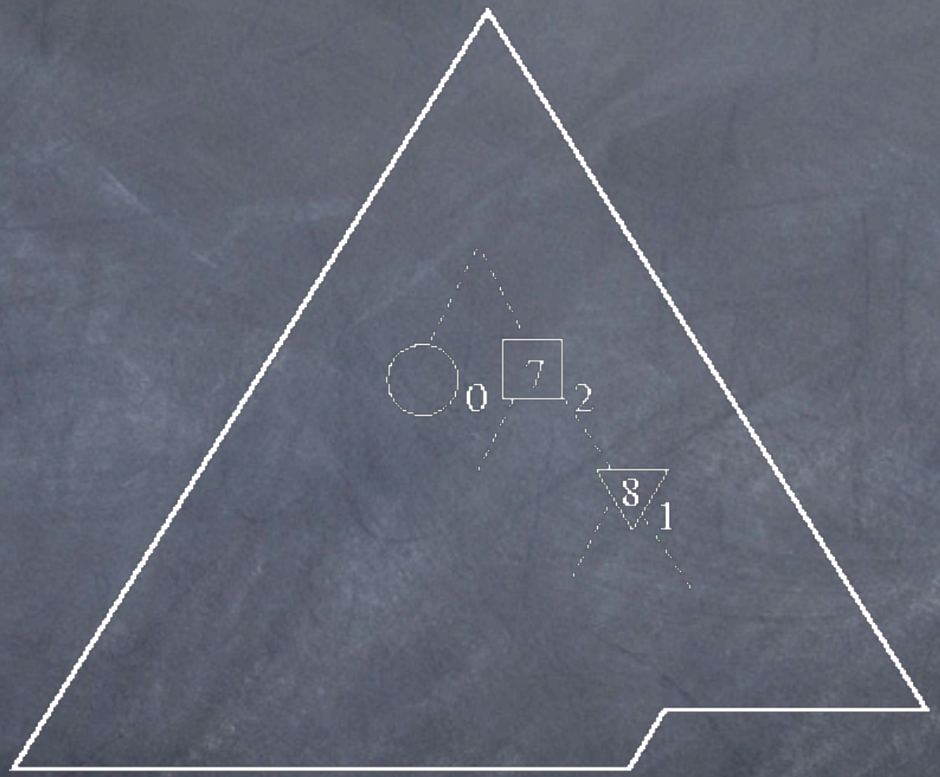
If these aren't relaxed then what's the point

Heap



0 = unlocked
1 = moving down
2 = moving up

Heap



Down has the Right of Way.
Down does the swap and swaps the lock flags

Issues

- How do we handle errors (1/x) if we want "fire and forget" AMO
- How do we feel about non-determinism?
- Which operations should we be looking at?
- Which references should we be looking at?
- How do we not lose all the optimization work?
- Do we need a new type?