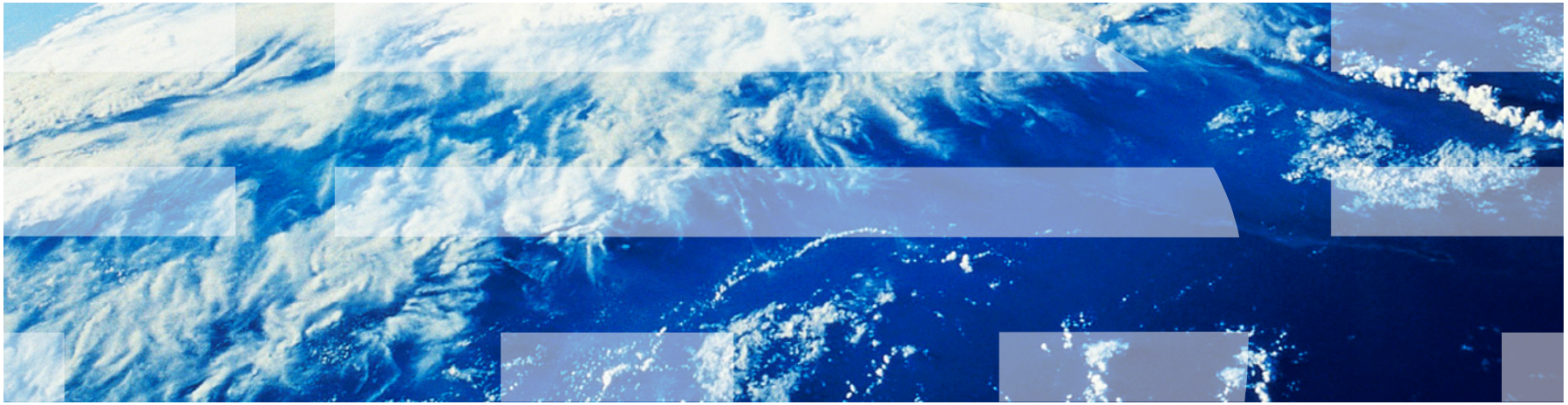
BG/Q Application Tuning

data prefetching, L2-atomic operations

Dr. Thilo Maurer, IBM Research & Development Böblingen
tmaurer@de.ibm.com

JUQUEEN Porting and Tuning Workshop, Feb 6th 2013



Stream Prefetching

- Recognizes consecutive memory accesses
- ensures early availability of data to processing unit
- 16 streams per core; 4 modes; mode valid for all threads on core
  - Off
  - Confirmed: new stream on two consecutive lines
  - Optimistic: new stream on every new address
  - Confirmed-or-DCBT (default)
- Application may switch mode anytime

```
#include <spi/include/l1p/sprefetch.h>

L1P_SetStreamPolicy(L1P_stream_optimistic);
L1P_SetStreamPolicy(L1P_stream_disable);
L1P_SetStreamPolicy(L1P_stream_confirmed);
L1P_SetStreamPolicy(L1P_confirmed_or_dcbt);
```


List-Based Prefetching

- Alternative to stream prefetching for non-consecutive but repetitive memory access patterns.
- Non-consecutive-ness may be due to indirect addressing or index calculation

Consecutive memory access:

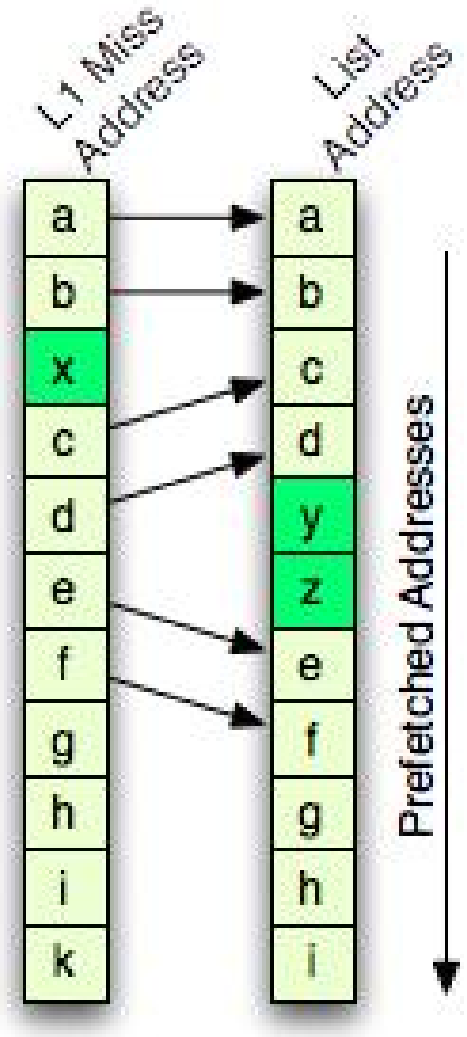
```
for (j = 0; j < M ; j++) {
   for (i = 0; i < N; i++) b[i] = a[i]*c[i];
   other_work(a, b, c);
}
```

*Non-consecutive (**but repetitive**) memory access:*

```
for (j = 0; j < M ; j++) {
   for (i = 0; i < N; i++) b[i] = a[index[i]]*c[i mod K];
   other_work(a, b, c);
}
```


List-Based Prefetching – Mechanism

- Idea: Use knowledge of access pattern to prefetch data in time
- User inserts START and STOP calls
- Hardware
  - records cache-miss addresses during first iteration
  - prefetches addresses in list in later iterations
  - Match miss-addresses against the list and prefetch to a depth N ahead of the present list-position.



Minimalistic Example:

```
for (j = 0; j < M ; j++) {
   L1P_PatternStart(j);
   for (i = 0; i < N; i++)
      b[i] = a[index[i]]*c[i mod K];
   L1P_PatternStop();
   other_work(a, b, c);
}
```


List-Based Prefetching – Syntax

- See spi/include/l1p/pprefetch.h for more and detailed descriptions

```
//simple usage model
int L1P_PatternConfigure(uint64_t n)
int L1P_PatternUnconfigure()
int L1P_PatternStart(int record)
int L1P_PatternStop()
int L1P_PatternPause()
int L1P_PatternResume()

//obtain status
int L1P_PatternStatus(L1P_Status_t* status)
int L1P_PatternGetCurrentDepth(uint64_t* fetch_depth, uint64_t* generate_depth)

//extended usage model
int L1P_AllocatePattern(uint64_t n, L1P_Pattern_t** handle)
int L1P_DeallocatePattern(L1P_Pattern_t* ptr)
int L1P_SetPattern(L1P_Pattern_t* ptr)
int L1P_GetPattern(L1P_Pattern_t** handle)
```


List-Based Prefetching – Full Example

```
L1P_PatternConfigure(n);
for (int iter=0;iter<iterations;iter++)
{
    int record=(iter==0?1:adaptive);

    L1P_PatternStart(iter||always);
    for (int i=0;i<n;i++)
    {
        sum+=a[i%iter]*b[index[i]];
        L1P_PatternPause();
        printf("sum=%lf\n",sum);
        L1P_PatternResume();
    }
    L1P_PatternStop();

    L1P_Status_t st;
    L1P_PatternStatus(&st);
    printf("iter=%d, RecordOverflow=%d, PlaybackAbandoned=%d,
        PlaybackEnded=%d, ReachedEndOfPlaybackList=%d\n",
        it, st.s.maximum, st.s.abandoned, st.s.finished, st.s.endoflist);
}
L1P_PatternUnconfigure();
```


List-Based Prefetching Limitations & Tuning

- recorded address-list depends on cache-composition before & during first iteration (only missed are recorded)
  - recording gives inreproducible list in shared-cache situations
- hardware looks 8 entries ahead
  - Playback unreliable in shared-cache situations
- Can work well when
  - no work-sharing among threads
  - no cache-line-sharing among threads

List-Based Prefetching - Abuse

```
!$omp parallel

!$omp do private(ix, iy, iz, i)
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    call other_code(...)
    do while ( i /= 0)
      cel(ix,iy,iz)%m      += slv(i)%m
      cel(ix,iy,iz)%vcm(1) += slv(i)%m * slv(i)%v(1)
      cel(ix,iy,iz)%vcm(2) += slv(i)%m * slv(i)%v(2)
      cel(ix,iy,iz)%vcm(3) += slv(i)%m * slv(i)%v(3)
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do

!$omp end parallel
```

16.9 sec
(14% of app)

```
!$omp parallel

!$omp do private(ix,iy,iz,i,...) ...
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      call heavy_compute(slv(i),...)
      call other_code()
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do

!$omp end parallel
```

23.7 sec
(20% of app)

```
!$omp parallel

!$omp do private(ix, iy, iz, i)
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      slv(i)%v_rel(1) = slv(i)%v(1) - cel(ix,iy,iz)%vcm(1)
      slv(i)%v_rel(2) = slv(i)%v(2) - cel(ix,iy,iz)%vcm(2)
      slv(i)%v_rel(3) = slv(i)%v(3) - cel(ix,iy,iz)%vcm(3)
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do

!$omp end parallel
```

19.6 sec
(17% of app)

List-Based Prefetching - Abuse

```
!$omp parallel

!$omp do private(ix, iy, iz, i)
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    call other_code(...)
    do while ( i /= 0)
      cel(ix,iy,iz)%m      += slv(i)%m
      cel(ix,iy,iz)%vcm(1) += slv(i)%m * slv(i)%v(1)
      cel(ix,iy,iz)%vcm(2) += slv(i)%m * slv(i)%v(2)
      cel(ix,iy,iz)%vcm(3) += slv(i)%m * slv(i)%v(3)
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do

!$omp end parallel
```

```
!$omp parallel

!$omp do private(ix,iy,iz,i,...) ...
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      call heavy_compute(slv(i),...)
      call other_code()
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do

!$omp end parallel
```

```
!$omp parallel

!$omp do private(ix, iy, iz, i)
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      slv(i)%v_rel(1) = slv(i)%v(1) - cel(ix,iy,iz)%vcm(1)
      slv(i)%v_rel(2) = slv(i)%v(2) - cel(ix,iy,iz)%vcm(2)
      slv(i)%v_rel(3) = slv(i)%v(3) - cel(ix,iy,iz)%vcm(3)
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do

!$omp end parallel
```

List-Based Prefetching – First Shot

```
!$omp parallel
  call L1P_PatternStart(1)
!$omp do private(ix, iy, iz, i)
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    call other_code(...)
    do while ( i /= 0)
      cel(ix,iy,iz)%m      += slv(i)%m
      cel(ix,iy,iz)%vcm(1) += slv(i)%m * slv(i)%v(1)
      cel(ix,iy,iz)%vcm(2) += slv(i)%m * slv(i)%v(2)
      cel(ix,iy,iz)%vcm(3) += slv(i)%m * slv(i)%v(3)
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do
  call L1P_PatternStop()
!$omp end parallel
```

also recording unwanted memory addresses

```
!$omp parallel
  call L1P_PatternStart(0)
!$omp do private(ix,iy,iz,i,...) ...
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      call heavy_compute(slv(i),...)
      call other_code()
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do
  call L1P_PatternStop()
!$omp end parallel
```

```
!$omp parallel
  call L1P_PatternStart(0)
!$omp do private(ix, iy, iz, i)
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      slv(i)%v_rel(1) = slv(i)%v(1) - cel(ix,iy,iz)%vcm(1)
      slv(i)%v_rel(2) = slv(i)%v(2) - cel(ix,iy,iz)%vcm(2)
      slv(i)%v_rel(3) = slv(i)%v(3) - cel(ix,iy,iz)%vcm(3)
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do
  call L1P_PatternStop()
!$omp end parallel
```

List-Based Prefetching – Second Thought

```
!$omp parallel
  call L1P_PatternStart(0)
!$omp do private(ix, iy, iz, i)
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    call other_code(...)
    do while ( i /= 0)
      cel(ix,iy,iz)%m      += slv(i)%m
      cel(ix,iy,iz)%vcm(1) += slv(i)%m * slv(i)%v(1)
      cel(ix,iy,iz)%vcm(2) += slv(i)%m * slv(i)%v(2)
      cel(ix,iy,iz)%vcm(3) += slv(i)%m * slv(i)%v(3)
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do
  call L1P_PatternStop()
!$omp end parallel
```

```
!$omp parallel
  call L1P_PatternStart(0)
!$omp do private(ix,iy,iz,i,...) ...
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      call heavy_compute(slv(i),...)
      call other_code()
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do
  call L1P_PatternStop()
!$omp end parallel
```

```
!$omp parallel
  call L1P_PatternStart(0)
!$omp do private(ix, iy, iz, i)
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      slv(i)%v_rel(1) = slv(i)%v(1) - cel(ix,iy,iz)%vcm(1)
      slv(i)%v_rel(2) = slv(i)%v(2) - cel(ix,iy,iz)%vcm(2)
      slv(i)%v_rel(3) = slv(i)%v(3) - cel(ix,iy,iz)%vcm(3)
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do
  call L1P_PatternStop()
!$omp end parallel
```

List-Based Prefetching – Pragmatic?

- List not easily recordable?
- Willing to leave predefined paths?
- Construct the list on your own and pass to the hardware!

```
#define L1P_ME(a)  (3 + ISHFT(ISHFT(a,-7),3))
paslv=virt2phys(loc(slv(0)))
pacpl=virt2phys(loc(cp%list(0)))

uint32_t* L1P_PatternReadBuffer() {
return _L1P_Context[Kernel_ProcessorID()]
                        .currentPattern.ReadPattern;
}

!$omp parallel private(crp,rp)
  len = 2*size(cp%list)/numthreads
  call L1P_PatternConfigure(len)
  crp = L1P_PatternReadBuffer()
  call c_f_pointer(crp, rp, [len])
  rpp=1
!$omp do private(ix,iy,iz,i) schedule(static)
  do iz = 1, dd%nc(3)
  do iy = 1, dd%nc(2)
  do ix = 1, dd%nc(1)
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      rp(rpp) = L1P_ME(paslv+i*sizeof(slv(0)))
      rpp=rpp+1
      rp(rpp) = L1P_ME(pacpl+i*sizeof(cp%list(0)))
      rpp=rpp+1
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do
  rp(rpp)=-1
!$omp end parallel
```

List-Based Prefetching – Results

```
!$omp parallel
  call L1P_PatternStart(0)
!$omp do private(ix, iy, iz, i)
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    call other_code(...)
    do while ( i /= 0)
      cel(ix,iy,iz)%m      += slv(i)%m
      cel(ix,iy,iz)%vcm(1) += slv(i)%m * slv(i)%v(1)
      cel(ix,iy,iz)%vcm(2) += slv(i)%m * slv(i)%v(2)
      cel(ix,iy,iz)%vcm(3) += slv(i)%m * slv(i)%v(3)
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do
  call L1P_PatternStop()
!$omp end parallel
```

~~16.9 sec~~
8.5 sec

```
!$omp parallel
  call L1P_PatternStart(0)
!$omp do private(ix,iy,iz,i,...) ...
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      call heavy_compute(slv(i),...)
      call other_code()
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do
  call L1P_PatternStop()
!$omp end parallel
```

~~23.7 sec~~
11.5 sec

```
!$omp parallel
  call L1P_PatternStart(0)
!$omp do private(ix, iy, iz, i)
  do iz = 1, nz
  do iy = 1, ny
  do ix = 1, nx
    i = cp%head(ix,iy,iz)
    do while ( i /= 0 )
      slv(i)%v_rel(1) = slv(i)%v(1) - cel(ix,iy,iz)%vcm(1)
      slv(i)%v_rel(2) = slv(i)%v(2) - cel(ix,iy,iz)%vcm(2)
      slv(i)%v_rel(3) = slv(i)%v(3) - cel(ix,iy,iz)%vcm(3)
      i = cp%list(i)
    enddo
  enddo
  enddo
  enddo
!$omp end do
  call L1P_PatternStop()
!$omp end parallel
```

~~19.6 sec~~
11.0 sec

List-Based Prefetching – Improved Results

Additionally align
each slv(i) to 128B
to eliminate playback-stalls

~~16.9 sec~~
~~8.5 sec~~
7.1 sec
(2.3x)

~~19.6 sec~~
~~11.0 sec~~
8.5 sec
(2.3x)

~~23.7 sec~~
~~11.5 sec~~
8.6 sec
(2.8x)

L2-atomic Operations

- L2 cache provides atomic in-memory operations
  - only 64b integer
  - two-operation atomics (e.g. load, then increment)
  - three-operation atomics (e.g. load, then increment if not equal)
- Can be leveraged to do
  - efficient cross-thread communication (Locks, Queue, Dequeues, etc...)
  - Simple calculations: e.g. collaborative summing

L2-atomic Operations – Syntax

- User needs to explicitly allocate memory region
  uint64_t Kernel_L2AtomicsAllocate(void* ptr, size_t length);
- Loads involving adjacent locations need to be in same 32b line

```
#include <spi/include/l2/atomic.h>
#include <hwi/include/common/bgq_alignment.h>

typedef int64_t i6;
i6 *p, value, atbound=0x8000000000000000;

i6 L2_AtomicLoad(p);                    // load
i6 L2_AtomicLoadClear(p);               // load, clear
i6 L2_AtomicLoadIncrement(p);           // load, increment (wraps)
i6 L2_AtomicLoadDecrement(p);           // load, decrement (wraps)
i6 L2_AtomicLoadIncrementBounded(p);    // load, increment when *p!=*(p+1), else atbound
i6 L2_AtomicLoadDecrementBounded(p);    // load, decrement when *p!=*(p+1), else atbound
i6 L2_AtomicLoadIncrementIfEqual(p);    // load, increment when *p =*(p+1), else atbound

void L2_AtomicStore(p, value);              // store to p
void L2_AtomicStoreTwin(p, value);          // store to p and p+1 if they contain equal values
void L2_AtomicStoreAdd(p, value);           // add value to *p
void L2_AtomicStoreAddCoherenceOnZero(p, value); // add value to *p, only cohere if *p is zero
void L2_AtomicStoreOr(p, value);            // OR *p with value
void L2_AtomicStoreXor(p, value);           // XOR *p with value
void L2_AtomicStoreMax(p, value);           // store maximum to p
void L2_AtomicStoreMaxSignValue(p, value);  // store maximum to p (double floating point)
```

L2-atomic Operations

- A few synchronisation constructs available
  - Barrier: #include <spi/include/l2/barrier.h>
  - Lock: #include <spi/include/l2/lock.h>
  - Queue (contact me)
- Other complex constructs may be built
  - Dequeue
  - Stack
- FORTRAN interface (contact me)

L2-atomic Operations Example

```
integer   :: i,ix,iy,iz,tmp
cp%list(:) = 0
cp%head(:,:,:) = 0
cel%(:,:,:)%n_slv = 0

!$omp parallel
!$omp do private(i, ix, iy, iz, tmp)
 do i = 1, dd%n_slv
   ix = calc_ix(...)
   iy = calc_iy(...)
   iz = calc_iz(...)

!$omp critical
   ! count particles
   cel(ix,iy,iz)%n_slv = cel(ix,iy,iz)%n_slv + 1

   ! and update head of list
   tmp = cp%head(ix,iy,iz)
   cp%head(ix,iy,iz) = i
!$omp end critical
   cp%list(i) = tmp
 enddo
!$omp end do
!$omp end parallel
```

```
integer*8 :: i,ix,iy,iz,tmp
cp%list(:) = 0
cp%head(:,:,:) = -1
cel%(:,:,:)%n_slv = 0

!$omp parallel
!$omp do private(i, ix, iy, iz, tmp)
 do i = 1, dd%n_slv
   ix = calc_ix(...)
   iy = calc_iy(...)
   iz = calc_iz(...)

   ! count particles
   call L2_AtomicStoreAdd(cel(ix,iy,iz)%n_slv,1)

   ! and update head of list
   ! aquire head; lock by set to 0
   do
     ! try aquire
     tmp = L2_AtomicLoadClear(cp%head(ix,iy,iz))
     ! 0 if already aquired
     if (tmp.ne.0) exit
   end do
   ! write new value, (implicitly releasing lock)
   call L2_AtomicStore(cp%head(ix,iy,iz),i)

   if (tmp.eq.-1) tmp = 0
   cp%list(i) = tmp
 enddo
!$omp end do

! clean up by replacing -1 by 0
!$omp do private(ix, iy, iz)
 do (iz,iy,iz) = (1,1,1), (nz,ny,nx)
   if (cp%head(ix,iy,iz).eq.-1) cp%head(ix,iy,iz) = 0
 enddo
!$omp end do
!$omp end parallel
```

L2-atomic Operations Example

```
integer   :: i,ix,iy,iz,tmp
cp%list(:) = 0
cp%head(:,:,:) = 0
cel%(:,:,:)%n_slv = 0

!$omp parallel
!$omp do private(i, ix, iy, iz, tmp)
 do i = 1, dd%n_slv
   ix = calc_ix(...)
   iy = calc_iy(...)
   iz = calc_iz(...)

!$omp critical
   ! count particles
   cel(ix,iy,iz)%n_slv = cel(ix,iy,iz)%n_slv + 1

   ! and update head of list
   tmp = cp%head(ix,iy,iz)
   cp%head(ix,iy,iz) = i
!$omp end critical
   cp%list(i) = tmp
 enddo
!$omp end do
!$omp end parallel
```

Takes 16.6s

4 threads per process
Speedup of 1.74

Takes 9.5s

```
integer*8 :: i,ix,iy,iz,tmp
cp%list(:) = 0
cp%head(:,:,:) = -1
cel%(:,:,:)%n_slv = 0

!$omp parallel
!$omp do private(i, ix, iy, iz, tmp)
 do i = 1, dd%n_slv
   ix = calc_ix(...)
   iy = calc_iy(...)
   iz = calc_iz(...)

   ! count particles
   call L2_AtomicStoreAdd(cel(ix,iy,iz)%n_slv,1)

   ! and update head of list
   ! aquire head; lock by set to 0
   do
     ! try aquire
     tmp = L2_AtomicLoadClear(cp%head(ix,iy,iz))
     ! 0 if already aquired
     if (tmp.ne.0) exit
   end do
   ! write new value, (implicitly releasing lock)
   call L2_AtomicStore(cp%head(ix,iy,iz),i)

   if (tmp.eq.-1) tmp = 0
   cp%list(i) = tmp
 enddo
!$omp end do

! clean up by replacing -1 by 0
!$omp do private(ix, iy, iz)
 do (iz,iy,iz) = (1,1,1), (nz,ny,nx)
   if (cp%head(ix,iy,iz).eq.-1) cp%head(ix,iy,iz) = 0
 enddo
!$omp end do
!$omp end parallel
```