QPX and Parallel I/O in the HACC cosmology framework

Hal Finkel

Salman Habib, Katrin Heitmann, Adrian Pope, Vitali Morozov, Venkat Vishwanath, et al.

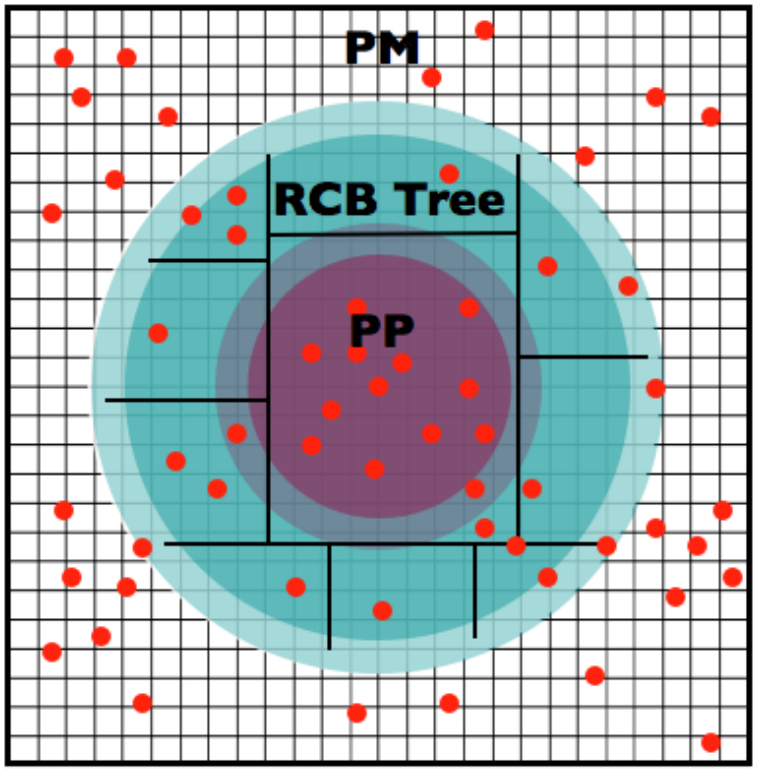
March 7, 2013

Cosmology

See part 1 (on threading) for the introduction.

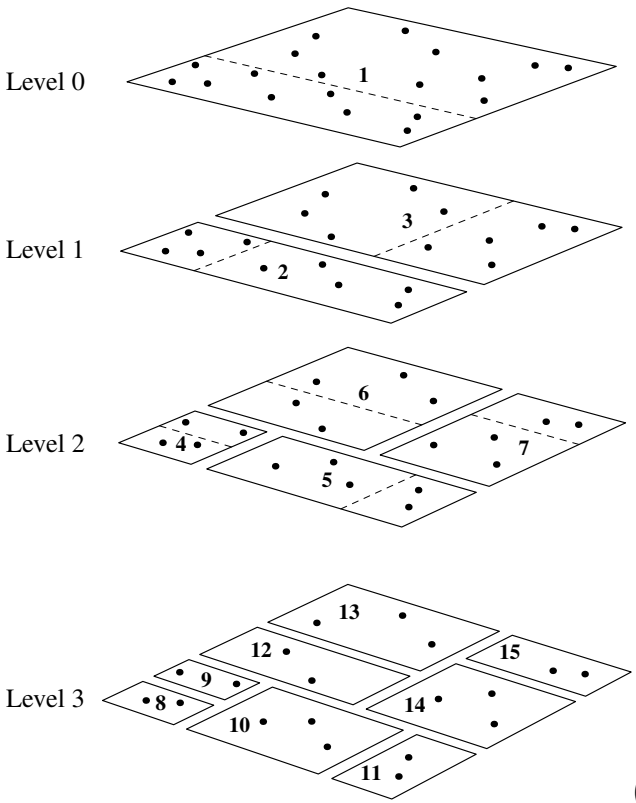
HACC

The HACC (Hybrid/Hardware Accelerated Cosmology Code) Framework meets these requirements using a P^3M (Particle-Particle Particle-Mesh) algorithm on accelerated systems and a Tree P^3M method on CPU-only systems (such as the BG/Q).



RCB Tree

The short-range force is computed using recursive coordinate bisection (RCB) tree in conjunction with a highly-tuned short-range polynomial force kernel.



(graphic from Gafton and Rosswog: arXiv:1108.0028)

RCB Tree (cont.)

- At each level, the node is split at its center of mass
- During each node split, the particles are partitioned into disjoint adjacent memory buffers
- This partitioning ensures a high degree of cache locality during the remainder of the build and during the force evaluation
- To limit the depth of the tree, each leaf node holds more than one particle. This makes the build faster, but more importantly, trades time in a slow procedure (a "pointer-chasing" tree walk) for a fast procedure (the polynomial force kernel).

Force Kernel

Due to the compactness of the short-range interaction, the kernel can be represented as

$$f_{SR}(s) = (s + \epsilon)^{-3/2} - f_{grid}(s) \quad (1)$$

where $s = \mathbf{r} \cdot \mathbf{r}$, $f_{grid}(s) = \text{poly}[5](s)$, and ϵ is a short-distance cutoff.

- An interaction list is constructed during the tree walk for each leaf node
- When using fine-grained threading: using OpenMP, the particles in the leaf node are assigned to different threads: all threads share the interaction list (which automatically balances the computation)
- The interaction list is processed using a vectorized kernel routine (written using QPX compiler intrinsics)
- Filtering for self and out-of-range interactions uses the floating-point select instruction: no branching required
- We can use the reciprocal (sqrt) estimate instructions: no library calls

Prefetching

- Prefetch all loads (but never prefetch the same 64-byte L1 cache line twice)!
- For stride-1 streams, data would otherwise be in the L1P (14-20 cycle access latency). For more complicated patterns, the data would otherwise be in the L2.

```
    for ( i = 0, j = 0; i < count1-7; i = i + 8, j = j + 32 )
    {
        __dcbt( (void *)&xx1 [i+offset] );
        __dcbt( (void *)&yy1 [i+offset] );
        __dcbt( (void *)&zz1 [i+offset] );
        __dcbt( (void *)&mass1[i+offset] );
...
```

QPX Intrinsics

- Use threads and unrolling to hide latency (but remember that there are only 32 floating-point registers).
- Most floating-point operations have a 6-cycle latency: yields an effective delay of 6/(threads per core) instructions.

```
    for ( i = 0, j = 0; i < count1-7; i = i + 8, j = j + 32 )
    {
...
        b0 = vec_sub( b0, a1 );
        c0 = vec_sub( c0, a1 );

        b0 = vec_mul( b0, b0 );
        c0 = vec_mul( c0, c0 );

        b1 = vec_ld( j, yy1 );
        c1 = vec_ld( j+16, yy1 );
...
```

QPX Intrinsics (FMA)

- Modern super-computers are designed to compute low-order polynomials: do many FMAs!

```
    for ( i = 0, j = 0; i < count1-7; i = i + 8, j = j + 32 )
    {
...
        b1 = vec_madd( b2, a15, a14 );
        c1 = vec_madd( c2, a15, a14 );

        b1 = vec_madd( b2, b1, a13 );
        c1 = vec_madd( c2, c1, a13 );

        b1 = vec_madd( b2, b1, a12 );
        c1 = vec_madd( c2, c1, a12 );
...
```

QPX Intrinsics (select and sqrt)

- Use estimates with refinement to get only the precision that you need.
- When possible, use select and don't branch!

```
    for ( i = 0, j = 0; i < count1-7; i = i + 8, j = j + 32 )
    {
...
        b1 = vec_rsqrte( b0 );
        c1 = vec_rsqrte( c0 );
...
        b0 = vec_sel( b1, a6, b2 );
        c0 = vec_sel( c1, a6, c2 );
...
```

Parallel I/O

- For large writes (many TB in total), use non-collective I/O. We can choose either MPI I/O or POSIX I/O.
- Write one separate file per I/O node (which corresponds to 128 compute nodes).
- Preallocate the extent of the file.
- Each rank writes into a disjoint space (without any kind of data reorganization).
- Protect all data with CRC64 ("checksum") codes! – Please contact me for the source code.