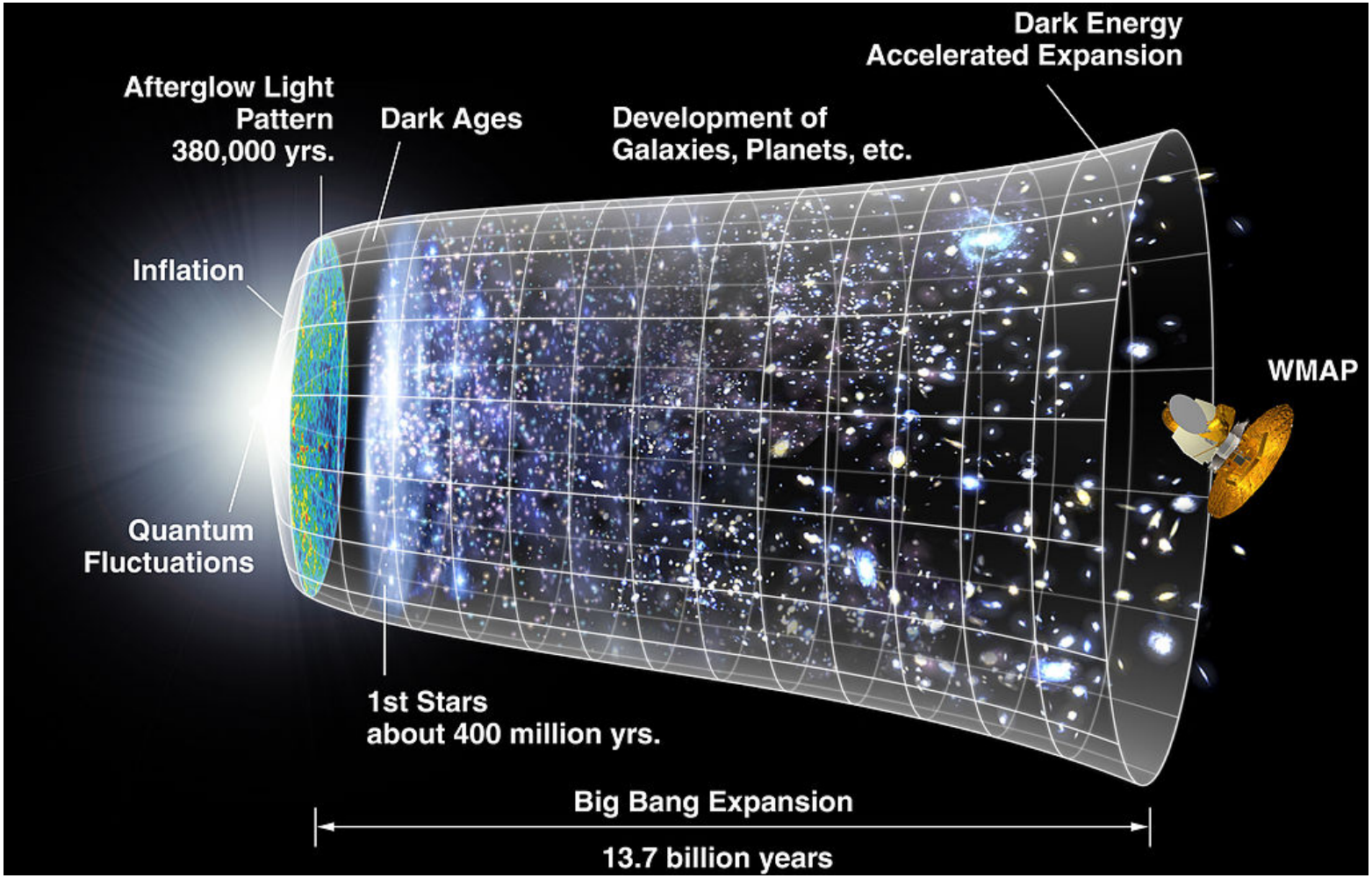
Portable Performance in the HACC cosmology framework

Hal Finkel

Salman Habib, Katrin Heitmann, Adrian Pope, Vitali Morozov, Nicholas Frontiere, et al.

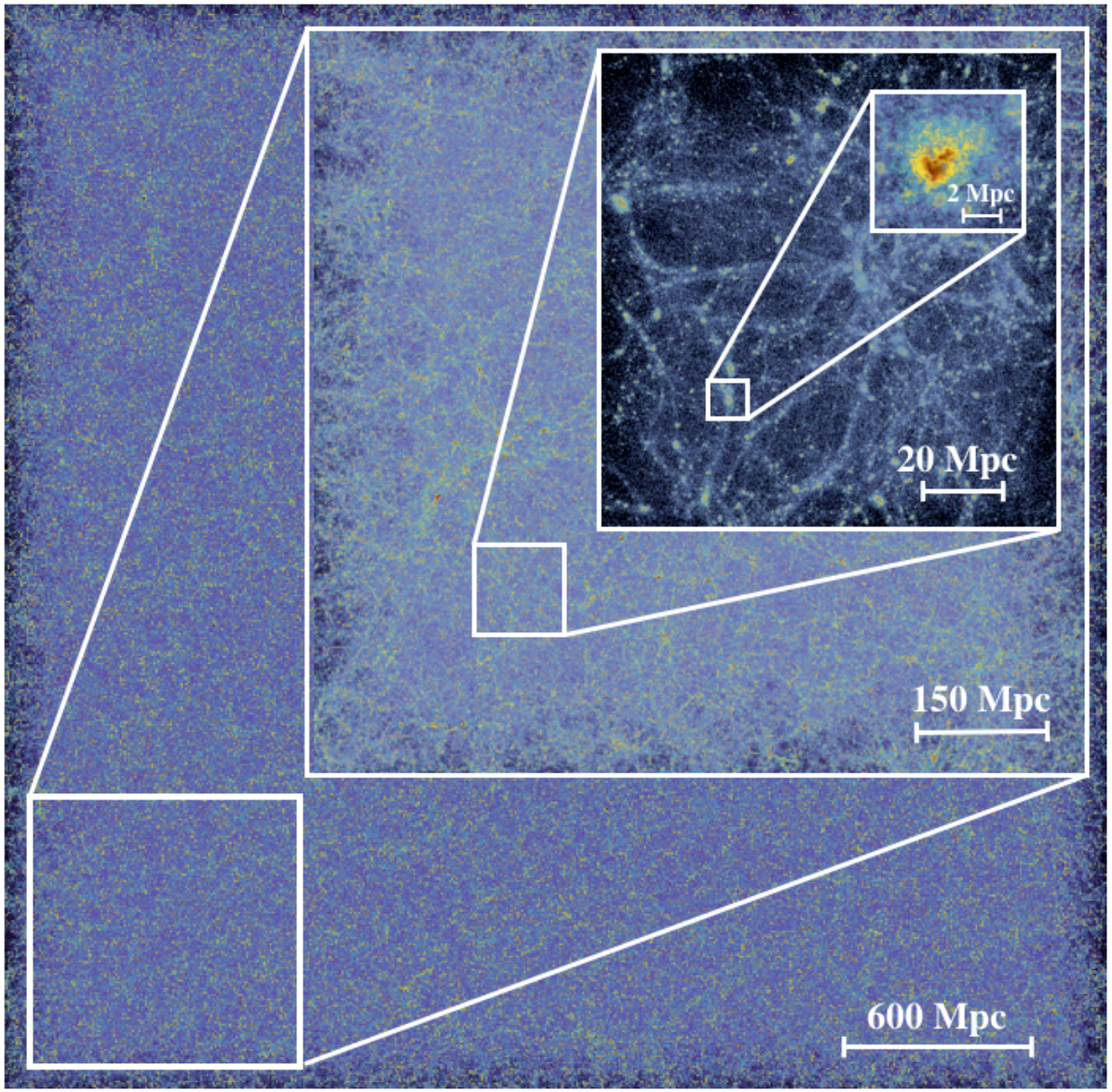
Aug 13, 2014

Cosmology

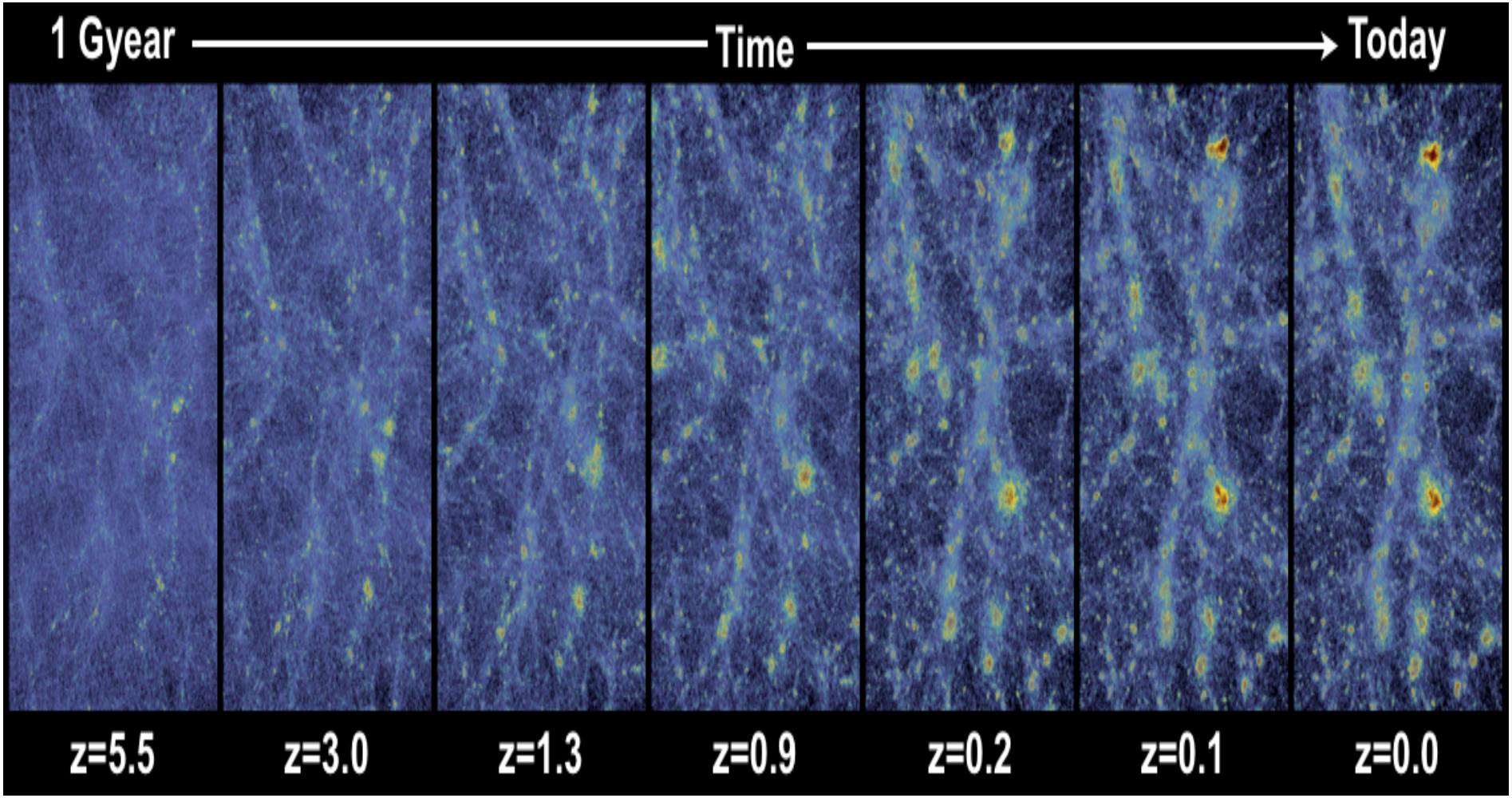


(graphic by: NASA/WMAP Science Team)

The Big Picture



Structure Evolution



Large-Scale Structure Formation

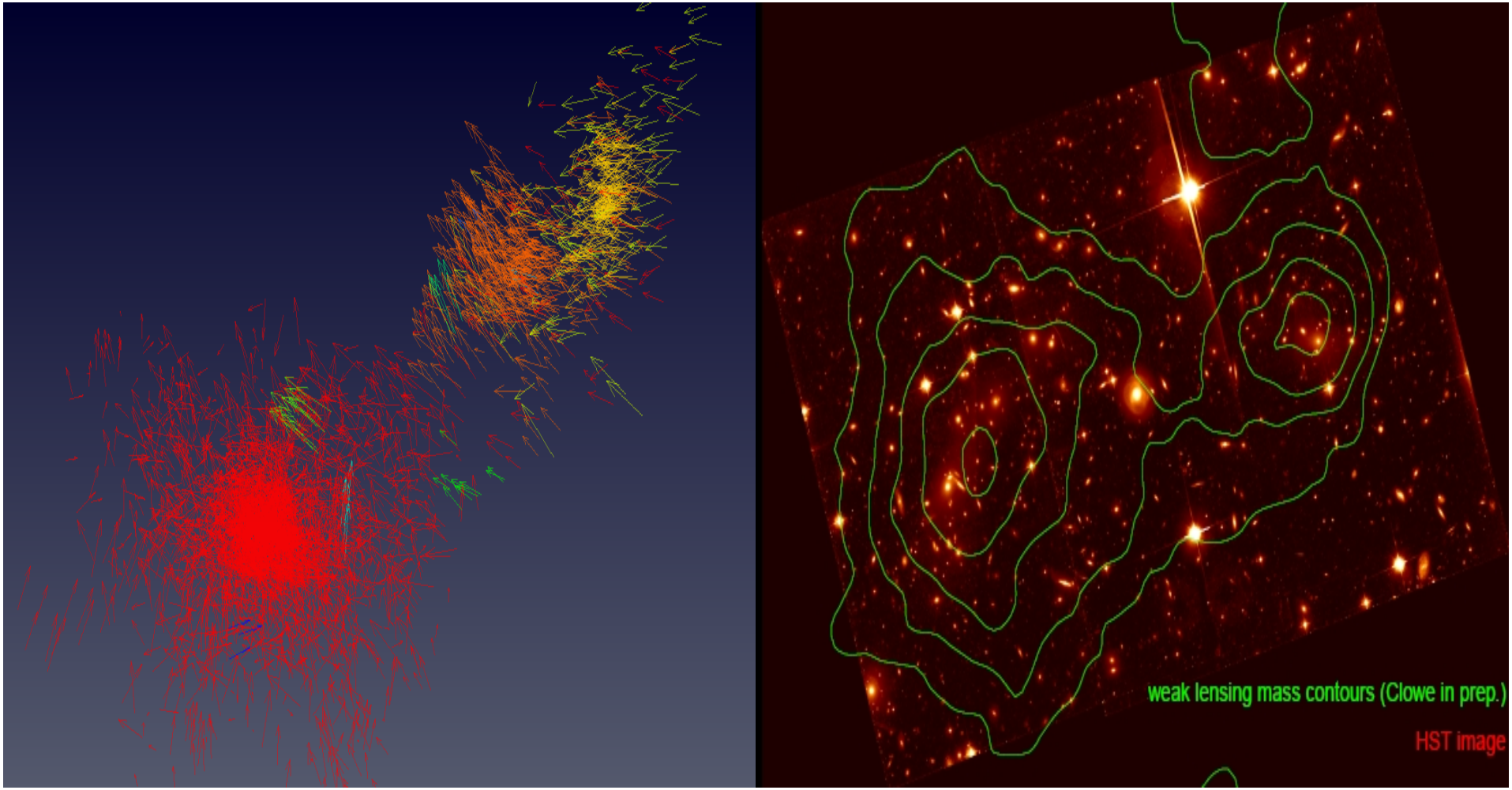
Why study large-scale structure formation?

- Cosmology is the study of the universe on the largest measurable scales.
- Dark matter and dark energy compose over 95% of the mass of the universe.
- The statistics of large-scale structure help us constrain models of dark matter and dark energy.
- Predictions from simulations can be compared to observations from large-scale surveys.
- Observational error bars are now approaching the sub-percent level, and simulations need to match that level of accuracy.

What We See

Image of galaxies covering a patch of the sky roughly equivalent to the size of the full moon (from the Deep Lens Survey)

What We See (cont.)



Vlasov-Poisson Equation

Cosmic structure formation follows the gravitational Vlasov-Poisson equation in an expanding Universe – a 6-D PDE for the Liouville flow (1) of the phase space PDF with the Poisson equation (2) imposing self-consistency:

$$\partial_t f(\mathbf{x}, \mathbf{p}) + \dot{\mathbf{x}} \cdot \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{p}) - \nabla\phi \cdot \partial_{\mathbf{p}} f(\mathbf{x}, \mathbf{p}) = 0, \tag{1}$$

$$\nabla^2 \phi(\mathbf{x}) = 4\pi G a^2(t) \Omega_m \delta_m(\mathbf{x}) \rho_c. \tag{2}$$

The expansion of the Universe is encoded in the time-dependence of the scale factor $a(t)$. Parameters:

- the Hubble parameter, $H = \dot{a}/a$
- G is Newton's constant
- ρ_c is the critical density
- Ω_m, the average mass density as a fraction of ρ_c

Vlasov-Poisson Equation (cont.)

- $\rho_m(\mathbf{x})$ is the local mass density
- $\delta_m(\mathbf{x})$ is the dimensionless density contrast

$$\rho_c = 3H^2/8\pi G, \quad \delta_m(\mathbf{x}) = (\rho_m(\mathbf{x}) - \langle\rho_m\rangle)/\langle\rho_m\rangle, \tag{3}$$

$$\mathbf{p} = a^2(t)\dot{\mathbf{x}}, \quad \rho_m(\mathbf{x}) = a(t)^{-3} m \int d^3\mathbf{p} f(\mathbf{x}, \mathbf{p}). \tag{4}$$

- The Vlasov-Poisson equation is very difficult to solve directly
- Structure develops on fine scales driven by the gravitational Jeans instability
- N-body methods: use tracer particles to sample $f(\mathbf{x}, \mathbf{p})$

Requirements

The requirements for our simulation campaigns are driven by the state of the art in observational surveys:

- Survey depths are of order a few Gpc (1 pc=3.26 light-years)
- To follow typical galaxies, halos with a minimum mass of $\sim 10^{11}$ $M_\odot$ ($M_\odot$=1 solar mass) must be tracked
- To properly resolve these halos, the tracer particle mass should be $\sim 10^8$ $M_\odot$
- The force resolution should be small compared to the halo size, i.e., $\sim$kpc.

This implies that the dynamic range of the simulation is a part in 10^6 ($\sim$Gpc/kpc) everywhere! With $\sim 10^5$ particles in the smallest resolved halos, we need hundreds of billions to trillions of particles.

Separation of Scales

The problem: Computing the particle-particle forces using an FFT-based particle-mesh technique is the most computationally efficient, but... we'd need an $\approx (10^6)^3$ grid capture the full dynamic range of the simulation!

The answer: A separation of scales: use the FFT-based particle-mesh technique for as much as possible, use some less-memory-hungry technique for the smaller scales. Plus, longer spatial scales have longer characteristic time scales, so we can "subcycle" the smaller scale computations relative to the long-range force computations. The short scale computations are now rank-local!

We can write $f(r_1 - r_2)$ as $f_{long}(r1 - r2) + f_{short}(r1 - r2)$.

Separation of Scales (cont.)

The problem: What are $f_{long}(r1 - r2)$ and $f_{short}(r1 - r2)$?

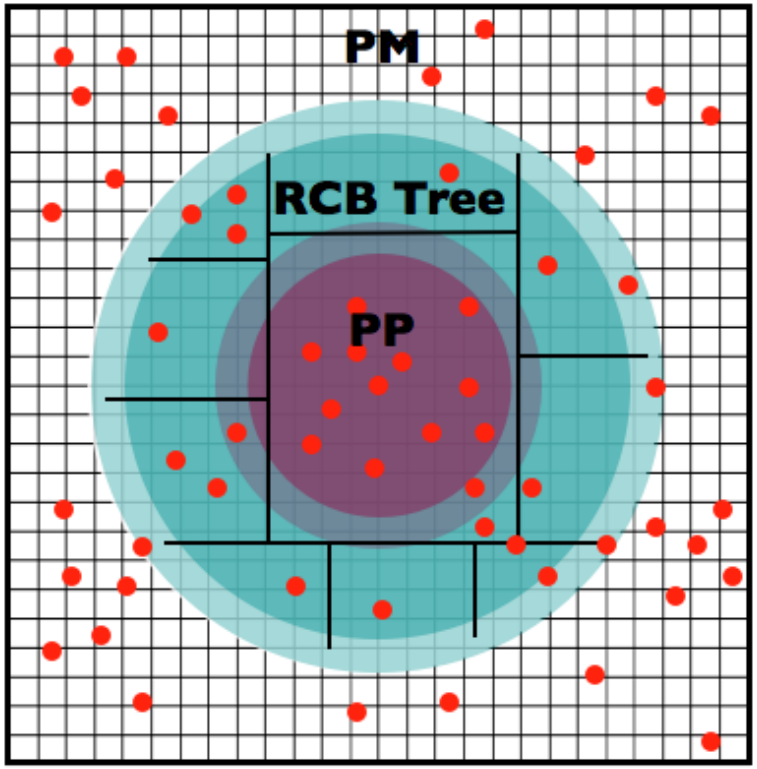
The answer: $f_{long}(r1 - r2)$, the "grid softened force", can be determined empirically. The force computed by the particle-mesh technique is sampled for many particle separations, and the resulting samples are fit by a polynomial. $f_{short}(r1 - r2)$ is then trivially determined by subtraction.

The question: How to best compute $f_{short}(r1 - r2)$.

The answer: This depends on the architecture!

HACC

The HACC (Hybrid/Hardware Accelerated Cosmology Code) Framework uses 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).



Force Splitting

- The gravitational force calculation is split into long-range part and a short-range part
- A grid grid is responsible for largest 4 orders of magnitude of dynamic range
- particle methods handle the critical 2 orders of magnitude at the shortest scales

Complexity:

- PM (grid) algorithm: $\mathcal{O}(N_p)+\mathcal{O}(N_g \log N_g)$, where N_p is the total number of particles, and N_g the total number of grid points
- tree algorithm: $\mathcal{O}(N_{pl} \log N_{pl})$, where N_{pl} is the number of particles in individual spatial domains ($N_{pl} \ll N_p$)
- the close-range force computations are $\mathcal{O}(N_d^2)$ where N_d is the number of particles in a tree leaf node within which all direct interactions are summed

Force Splitting (cont.)

Long-Range Algorithm:

- The long/medium range algorithm is based on a fast, spectrally filtered PM method
- The density field is generated from the particles using a Cloud-In-Cell (CIC) scheme

The density field is smoothed with the (isotropizing) spectral filter:

$$\exp\left(-k^2\sigma^2/4\right)\left[(2k/\Delta)\sin(k\Delta/2)\right]^{n_s}, \tag{5}$$

where the nominal choices are $\sigma = 0.8$ and $n_s = 3$. The noise reduction from this filter allows matching the short and longer-range forces at a spacing of 3 grid cells.

- The Poisson solver uses a sixth-order, periodic, influence function (spectral representation of the inverse Laplacian)
- The gradient of the scalar potential is obtained using higher-order spectral differencing (fourth-order Super-Lanczos)

Force Splitting (cont.)

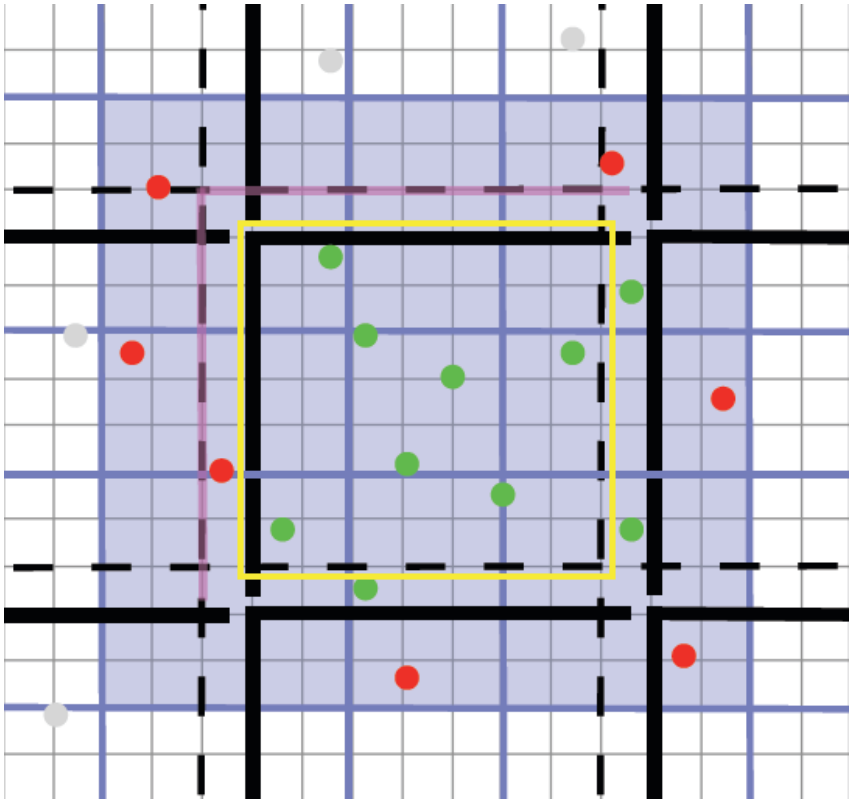
- The "Poisson-solve" is the composition of all the kernels above in one single Fourier transform
- Each component of the potential field gradient then requires an independent FFT
- Distributed FFTs use a pencil decomposition
- To obtain the short-range force, the filtered grid force is subtracted from the Newtonian force

Mixed precision:

- single precision is adequate for the short/close-range particle force evaluations and particle time-stepping
- double precision is used for the spectral component

Overloading

The spatial domain decomposition is in regular 3-D blocks, but unlike the guard zones of a typical PM method, full particle replication – termed 'particle overloading' – is employed across domain boundaries.



Overloading (cont.)

- Works because particles cluster and large-scale bulk motion is small
- Short-range force contribution is not used for particles near the edge of the overloading region
- The typical memory overhead cost for a large run is $\sim 10\%$
- The point of overloading is to allow sufficiently-exact medium/long-range force calculations with no communication of particle information and high-accuracy local force calculations

We use relatively sparse refreshes of the overloading zone! This is key to freeing the overall code performance from the weaknesses of the underlying communications infrastructure.

Time Stepping

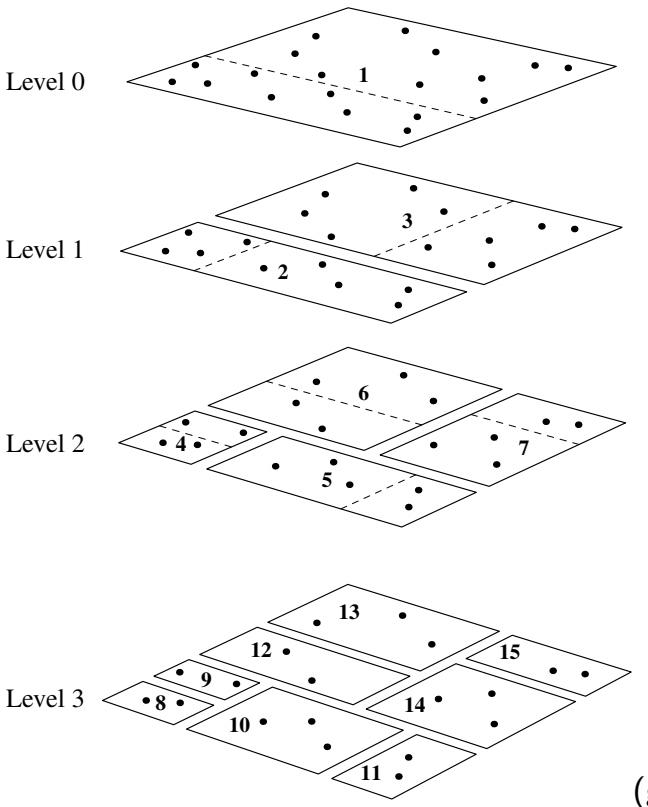
- The time-stepping is based on a 2nd-order split-operator symplectic SKS scheme (stream-kick-stream)
- Because the characteristic time scale of the long-range force is much smaller than that of the short-range force, we sub-cycle the short-range force operator
- The relatively slowly evolving longer range force is effectively frozen during the shorter-range sub-cycles

$$M_{full}(t) = M_{lr}(t/2)(M_{sr}(t/n_c))^{n_c} M_{lr}(t/2). \tag{6}$$

The number of sub-cycles is $n_c = 3 - 5$, in most cases.

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.



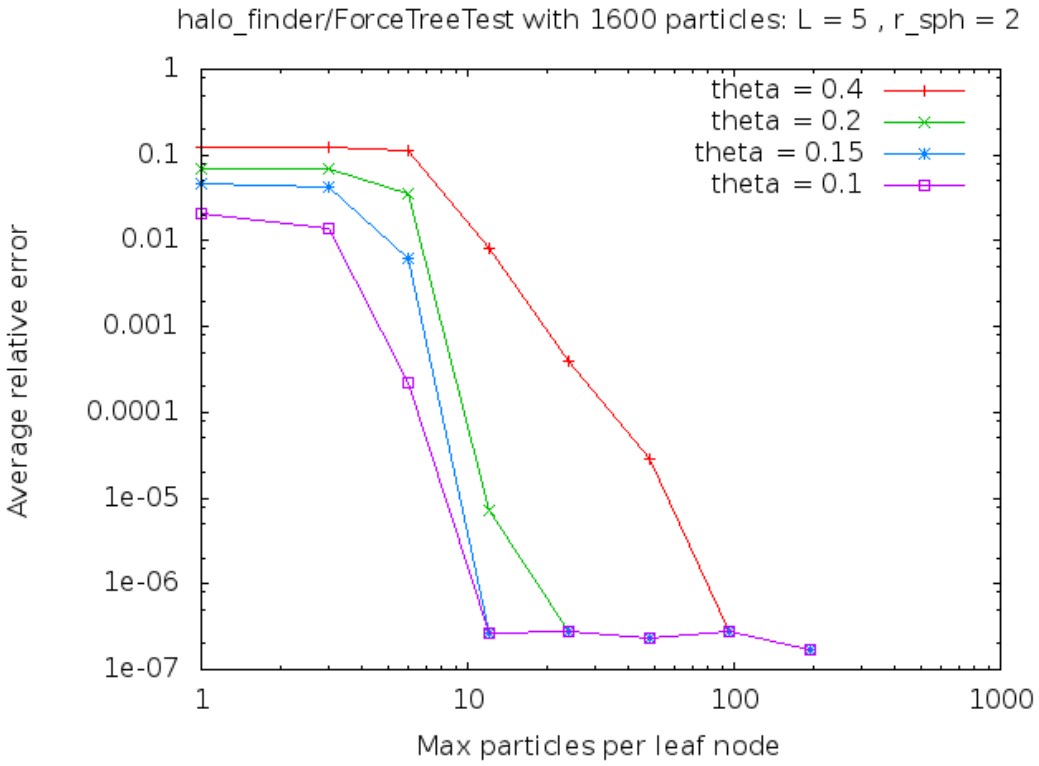
(graphic from Gafton and Rosswog: arXiv:1108.0028)

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).

RCB Tree (cont.)

Another benefit of using multiple particles per leaf node:



Force Kernel

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

$$f_{SR}(s) = (s + \epsilon)^{-3/2} - f_{grid}(s) \tag{7}$$

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/SSE 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

Force Kernel (cont.)

Remember:

- Memory motion is important! You may need to explicitly prefetch your data.
- Your compiler may not automatically pick the best loop unrolling factor.
- Modern super-computers are designed to compute low-order polynomials: do many FMAs!
- When possible, use estimates with refinement to get only the precision that you need (for reciprocals, reciprocal sqrt, etc.).
- When possible, use select and don't branch! The compiler may not always do this for you.

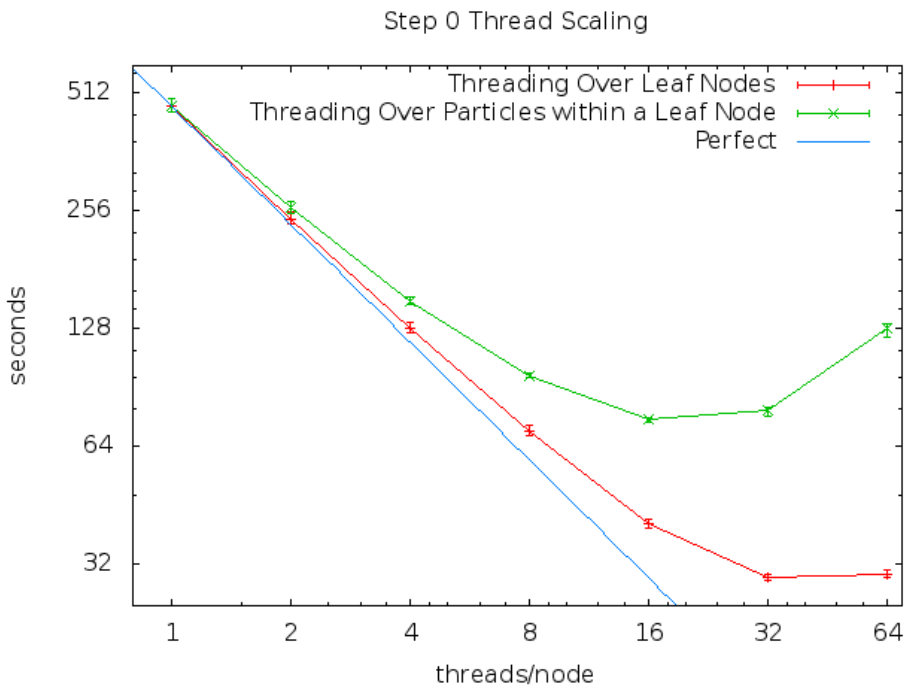
Running Configuration: Fine-Grained Threading

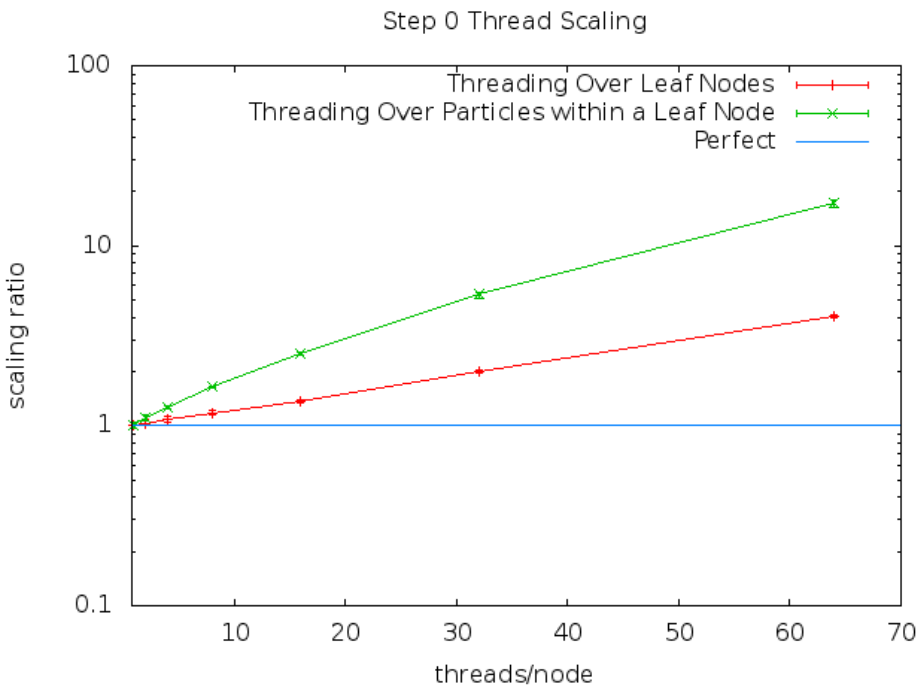
- Using OpenMP, the particles in the leaf node are assigned to different threads: all threads share the interaction list (which automatically balances the computation)
- We use either 8 threads per rank with 8 ranks per node, or 4 threads per rank and 16 ranks per node
- The code spends 80% of the time in the highly optimized force kernel, 10% in the tree walk, and 5% in the FFT, all other operations (tree build, CIC deposit) adding up to another 5%.

This code achieves over 50% of the peak FLOPS on the BG/Q!

Running Configuration: Threading over Leaf Nodes

- A work queue is formed of all leaf nodes, and this queue is processed dynamically using all available threads.
- Not limited by the concurrency available in each leaf node (which has only a few hundred particles with a collective interaction list in the thousands).





Take-Away Message on Threading

Balanced Concurrency!

Divide the problem into as many computationally-balanced work units as possible, and distribute those work units among the available threads. These units need to be large enough to cover the thread-startup overhead.

When using OpenMP, don't forget to use dynamic scheduling when the work unit size is only balanced on average:

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < WQS; ++i) {
  WorkQueue[i].execute();
}
```

And Think like a Compiler

On almost all HPC-relevant architectures, the compiler will never autovectorize this (without some special directives)...

```
void foo(double * restrict a, const double * restrict b, const double * restrict c) {
    for (i = 0; i < 2048; ++i) {
        if (c[i] > 0) { // for example: is the particle in range?
            a[i] = b[i];
        } else {
            a[i] = 0.0;
        }
    }
}
```

And Think like a Compiler

On almost all HPC-relevant architectures, the compiler will never autovectorize this (without some special directives)...

```
void foo(double * restrict a, const double * restrict b, const double * restrict c) {
    for (i = 0; i < 2048; ++i) {
        if (c[i] > 0) { // for example: is the particle in range?
            a[i] = b[i];
        } else {
            a[i] = 0.0;
        }
    }
}
```

No, it is not aliasing (that is what the 'restrict' is for)...

And Think like a Compiler

On almost all HPC-relevant architectures, the compiler will never autovectorize this (without some special directives)...

```
void foo(double * restrict a, const double * restrict b, const double * restrict c) {
    for (i = 0; i < 2048; ++i) {
        if (c[i] > 0) { // for example: is the particle in range?
            a[i] = b[i];
        } else {
            a[i] = 0.0;
        }
    }
}
```

No, it is not aliasing (that is what the 'restrict' is for)...
No, it has nothing to do with alignment...

And Think like a Compiler

On almost all HPC-relevant architectures, the compiler will never autovectorize this (without some special directives)...

```
void foo(double * restrict a, const double * restrict b, const double * restrict c) {
    for (i = 0; i < 2048; ++i) {
        if (c[i] > 0) { // for example: is the particle in range?
            a[i] = b[i];
        } else {
            a[i] = 0.0;
        }
    }
}
```

No, it is not aliasing (that is what the 'restrict' is for)...
No, it has nothing to do with alignment...
The compiler cannot prove that it is safe to speculatively dereference 'b' because 'b' could be NULL and c[i] could be always non-positive for all i.

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 );
...
```

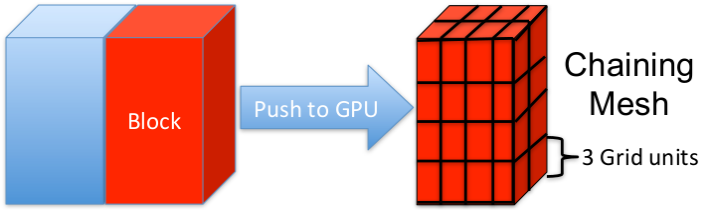
QPX Intrinsics (select and sqrt)

- 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 );
...
```

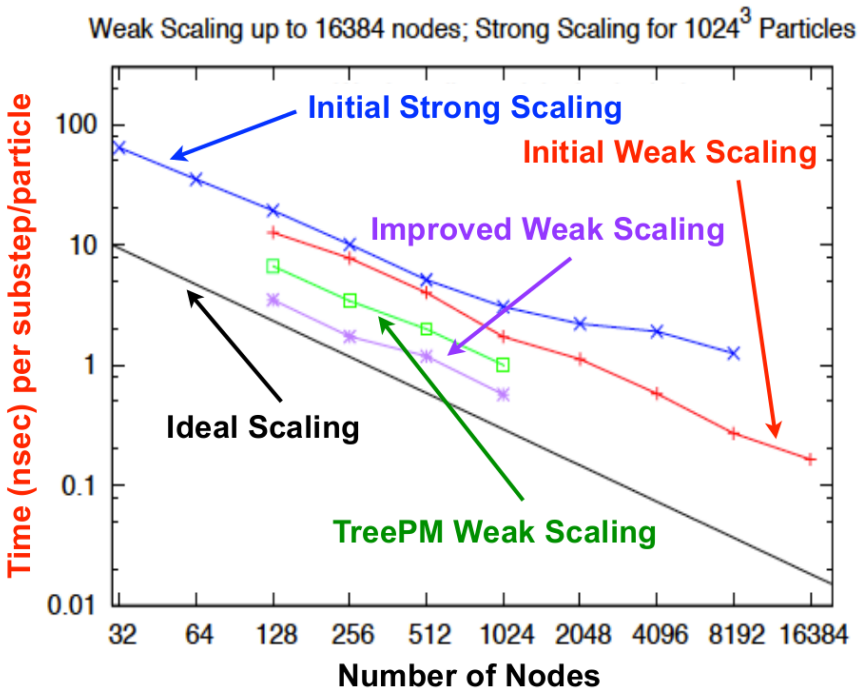
P^3M on GPUs

- Implemented in OpenCL
- Spatial data pushed to GPU in large blocks, data is sub-partitioned into chaining mesh cubes
- Forces are computed between particles in a cube and neighboring cubes
- Natural parallelism and simplicity leads to high performance
- Typical push size 2GB; large push size ensures computation time exceeds memory transfer latency by a large factor
- TreePM analog of BG/Q code written in CUDA, also produces high performance



GPU Scaling

- P^3M kernel runs at 1.6TFlops/node at 40.3% of peak
- TreePM kernel was run on 77% of Titan at 20.54 PFlops at almost identical performance on the card
- Because of less overhead, P^3M code is (currently) faster by factor of two in time to solution



GPU Programming

- CPU uses large caches, a few hardware threads, and for some, out-of-order dispatch, to hide latency - pipelines kept as short as possible.
- GPU uses many hardware threads, long pipelines, large memory-request reorder buffers, and fast DRAM. - You need lots of concurrency!
- On current systems, you must account for the cost of transferring data to the GPU and back again.
- Even for linear ("scan") memory-bandwidth-bound algorithms, the GPU can probably complete the task much faster than the CPU (it has higher internal bandwidth). [The BG/Q also has relatively high internal memory bandwidth, most other CPU systems in HPC do not!]

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.