N-body Simulation

Physics

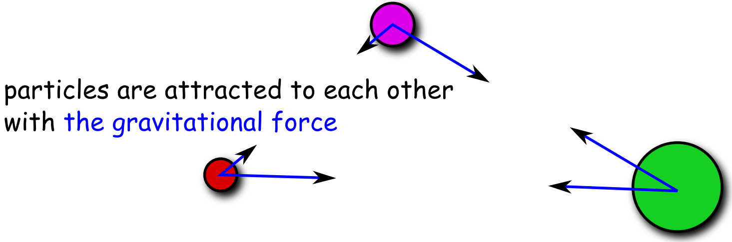
Gravitational N-body dynamics:

Newton's law of universal gravitation:

$$M_i \vec{R}_i''(t) = G \sum_j \frac{M_i M_j}{\left|\vec{R}_i - \vec{R}_j\right|^3} \left(\vec{R}_j - \vec{R}_i\right)$$

where:

$$\left|\vec{R}_i - \vec{R}_j\right| = \sqrt{(R_{i,\,x} - R_{j,\,x})^2 + (R_{i,\,y} - R_{j,\,y})^2 + (R_{i,\,z} - R_{j,\,z})^2}$$



Application

1. Astrophysics:
   - planetary systems
   - galaxies
   - cosmological structures
2. Electrostatic systems:
   - molecules
   - crystals

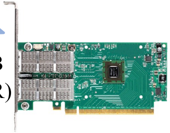
This work: "toy model" with all-to-all $O(n^2)$ algorithm. Practical N-body simulations may use tree algorithms with $O(n \log n)$ complexity.

Source: APOD, credit: Debra Meloy Elmegreen (Vassar College) et al., & the Hubble Heritage Team (AURA/ STScI/ NASA)

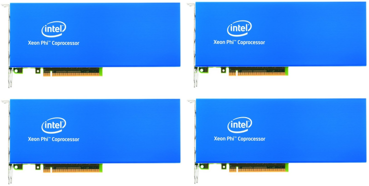
Comparative Benchmarks and System Configuration

Colfax ProEdge SXP8600p
rack-mountable workstations (cluster of 4)

Mellanox Connect-IB
InfiniBand HCA (FDR)



Intel Xeon Phi 7120P coprocessors
(4 per system)



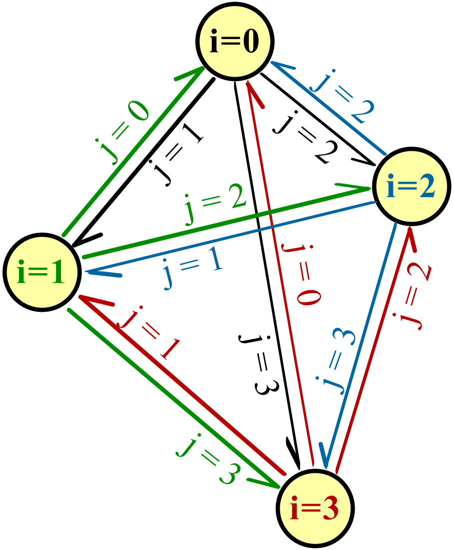
Dual-socket Intel Xeon E5-2697 v2
processor

http://xeonphi.com/workstations

Initial Implementation of the N-Body Simulation

Illustration of "Toy Model" Calculation Pattern

- All-to-all interaction
- $O(n^2)$ complexity
- All particles fit in memory of each compute node
- No multipole approximation, tree algorithms, Debye screening, etc.
- Basis for more efficient real-life models
- Good educational example



All-to-All Approach ($O\left(n^2\right)$ Complexity Scaling)

Each particle is stored as a structure:

```
struct ParticleType {
  float x, y, z;
  float vx, vy, vz;
};
```

`main()` allocates an array of `ParticleType`:

```
ParticleType* particle = new ParticleType[nParticles];
```

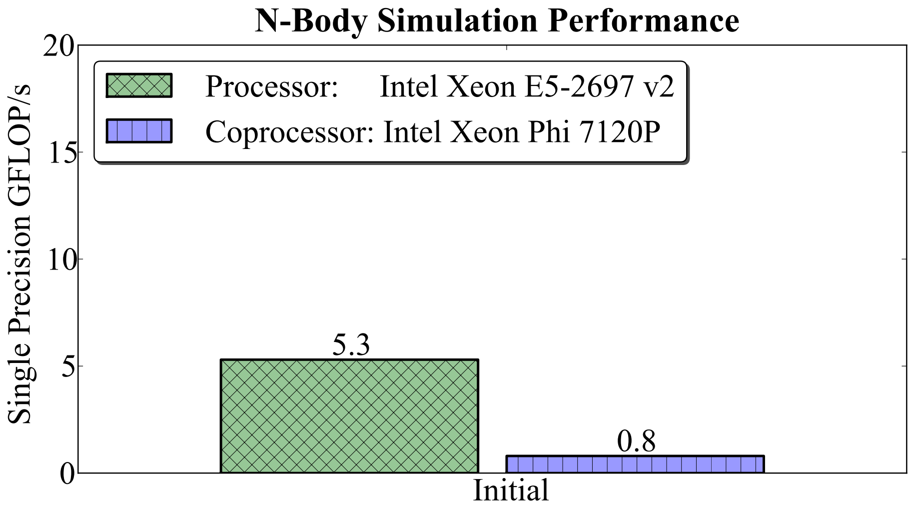
Particle propagation step is timed:

```
const double tStart = omp_get_wtime(); // Start timing
MoveParticles(nParticles, particle, dt);
const double tEnd = omp_get_wtime(); // End timing
```


Particle Update Engine

```
void MoveParticles(int nParticles, ParticleType* particle, float dt) {
  for (int i = 0; i < nParticles; i++) {   // Particles that experience force
    float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
    for (int j = 0; j < nParticles; j++) { // Particles that exert force
      // Newton's law of universal gravity
      const float dx = particle[j].x - particle[i].x;
      const float dy = particle[j].y - particle[i].y;
      const float dz = particle[j].z - particle[i].z;
      const float drSquared  = dx*dx + dy*dy + dz*dz + 1e-20;
      const float drPower32  = pow(drSquared, 3.0/2.0);
      // Calculate the net force
      Fx += dx/drPower32;  Fy += dy/drPower32;  Fz += dz/drPower32;
    }
    // Accelerate particles in response to the gravitational force
    particle[i].vx+=dt*Fx; particle[i].vy+=dt*Fy;  particle[i].vz+=dt*Fz;
  }
  ...
```


Performance of Initial Implementation



Optimization: Thread Parallelism

Incorporating Thread Parallelism

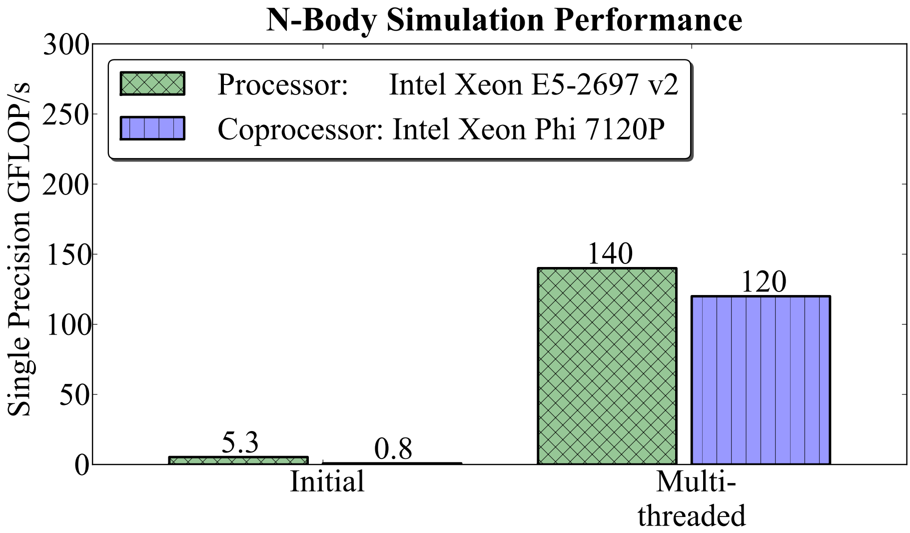
Before:

```
  for (int i = 0; i < nParticles; i++) {   // Particles that experience force
    float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
    for (int j = 0; j < nParticles; j++) { // Particles that exert force
      // Newton's law of universal gravity
      ...
```

After:

```
#pragma omp parallel for
  for (int i = 0; i < nParticles; i++) {   // Particles that experience force
    float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
    for (int j = 0; j < nParticles; j++) { // Particles that exert force
      // Newton's law of universal gravity
      ...
```

Performance with Thread Parallelism



Optimization: Vectorization

Vectorizing with Unit-Stride Memory Access

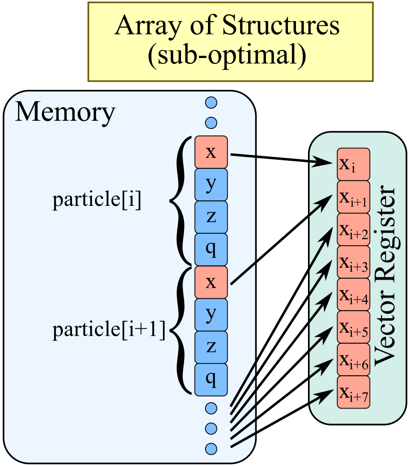
Before:

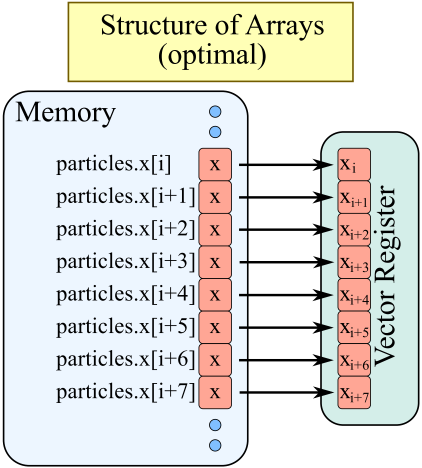
```
struct ParticleType {
  float x, y, z, vx, vy, vz;
};  // ...
      const float dx = particle[j].x - particle[i].x;
      const float dy = particle[j].y - particle[i].y;
      const float dz = particle[j].z - particle[i].z;
```

After:

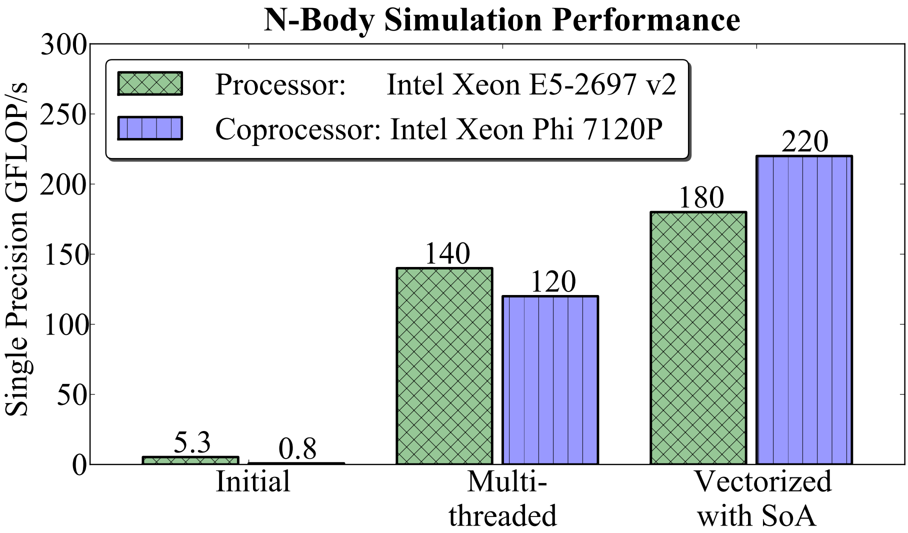
```
struct ParticleSet {
  float *x, *y, *z, *vx, *vy, *vz;
}; // ...
      const float dx = particle.x[j] - particle.x[i];
      const float dy = particle.y[j] - particle.y[i];
      const float dz = particle.z[j] - particle.z[i];
```


Why AoS to SoA Conversion Helps: Unit Stride





Performance with Improved Vectorization



Optimization: Scalar Tuning

Improving Scalar Expressions

Before:

```
const float drSquared  = dx*dx + dy*dy + dz*dz + 1e-20;
const float drPower32  = pow(drSquared, 3.0/2.0);
// Calculate the net force
Fx += dx/drPower32;  Fy += dy/drPower32;  Fz += dz/drPower32;
```

After:

```
const float drRecip     = 1.0f/sqrtf(dx*dx + dy*dy + dz*dz + 1e-20f);
const float drPowerN32  = drRecip*drRecip*drRecip;
// Calculate the net force
Fx += dx*drPowerN32;  Fy += dy*drPowerN32;  Fz += dz*drPowerN32;
```

- Strength reduction (division → multiplication by reciprocal)
- Precision control (suffix `-f` on single-precision constants and functions)
- Reliance on hardware-supported reciprocal square root

Compilation with Relaxed Precision

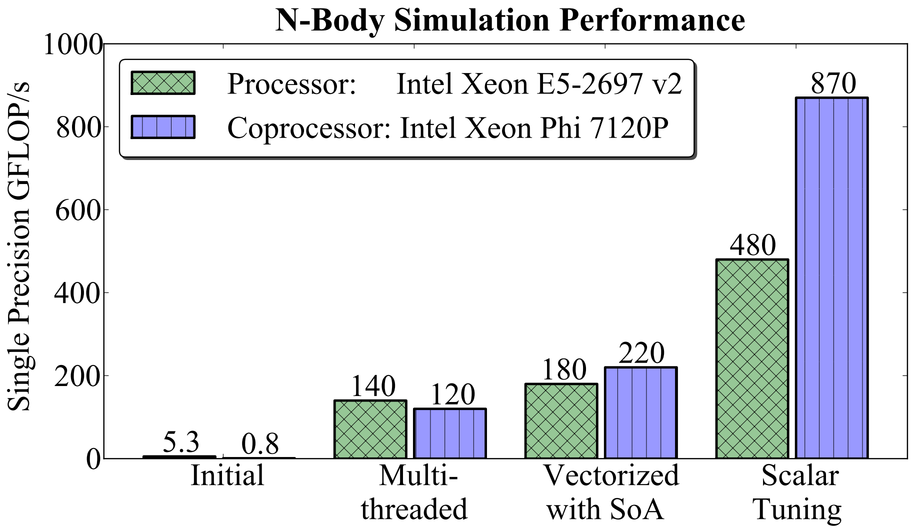
For the CPU architecture (Intel Xeon E5-2697 v2 processor):

```
vega@lyra% # Compile with relaxed precision: (-fp-model fast=2)
vega@lyra% icpc -o nbody-CPU -qopenmp -fp-model fast=2 nbody.cc
vega@lyra% export KMP_AFFINITY=compact
vega@lyra% ./nbody-CPU
```

For the MIC architecture (Intel Xeon Phi 7120P coprocessor):

```
vega@lyra% # Compile for Xeon Phi with relaxed precision: (-fp-model fast=2)
vega@lyra% icpc -o nbody-MIC -mmic -qopenmp -fp-model fast=2 nbody.cc
vega@lyra% export KMP_AFFINITY=compact
vega@lyra% export SINK_LD_LIBRARY_PATH=$MIC_LD_LIBRARY_PATH
vega@lyra% micnativeloadex ./nbody-MIC
```


Performance after Scalar Tuning



Optimization: Memory Traffic

Improving Cache Traffic

Before:

```
for (int i = 0; i < nParticles; i++) {   // Particles that experience force
  float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
  for (int j = 0; j < nParticles; j++) { // Particles that exert force
    // ...
    Fx += dx*drPowerN32;  Fy += dy*drPowerN32;  Fz += dz*drPowerN32;
```

After: (tileSize = 16)

```
for (int ii = 0; ii < nParticles; ii += tileSize) { // Particle blocks
  float Fx[tileSize], Fy[tileSize], Fz[tileSize]; // Force on particle block
  Fx[:] = Fy[:] = Fz[:] = 0;
#pragma unroll(tileSize)
  for (int j = 0; j < nParticles; j++) { // Particles that exert force
    for (int i = ii; i < ii + tileSize; i++) { // Traverse the block
      // ...
Fx[i-ii] += dx*drPowerN32; Fy[i-ii] += dy*drPowerN32; Fz[i-ii] += dz*drPowerN32;
```

Performance with Cache Optimization (Loop Tiling)