



Programming and Optimization for Intel[®] Architecture

The Hands-On Workshop (HOW) Series

Andrey Vladimirov, PhD, and Ryo Asai
Colfax International — @colfaxintl

March 2016 , Rev. 02a

About This Document

This document represents the materials of a Web-based training “Programming and Optimization with Intel Architecture” developed and run by Colfax International.

© Colfax International, 2013-2015

colfaxresearch.com/how-series

Parallel Programming Boot Camp (1-Day) / Workshop (4-Days)



Instructor-led 1-day or 4-days training, at your office or at Colfax facility in Sunnyvale, CA

[Click here to learn more](#)

1-Day Parallel Programming Boot Camp

For software engineers and architects, providing an overview of parallel programming frameworks and optimization guidelines for multi-core CPUs (Intel® Xeon®) and many-core coprocessors (Intel® Xeon Phi™):

- Discussions about three layers of parallelism: SIMD, Threads, Cluster environment
- Tips for quick porting/development of HPC software applications
- Real-life examples of code and optimization techniques
- Hardware solution and corresponding software implementations, APIs, and frameworks

4-Days Parallel Programming Workshop

For the developer who wants to hit the ground running with the modern multi-core CPUs (Intel® Xeon®), many-core coprocessors (Intel® Xeon Phi™) and leading software development tools:

- Hardware installation
- MPSS tools and the Linux environment on the Intel® Xeon Phi™ coprocessor
- Exploring differences in serial vs. parallel programming / processing / hardware usage
- Accelerated clusters
- Optimizations of vector arithmetics, memory traffic, thread parallelism and communication
- Using the Intel® Math Kernel Library

[Register Now!](#)

Disclaimer

While best efforts have been used in preparing this training, Colfax International makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The publisher shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials.

Course Roadmap

- 1 Why Intel Parallel Architectures?
 - ▶ Parallelism and specialization – March 7
 - ▶ Programming model continuity – March 7
- 2 Programming models for Xeon Phi coprocessors
 - ▶ Native programming – March 7
 - ▶ Offload programming – March 8
- 3 Expressing Parallelism
 - ▶ Introduction to vectorization – March 9
 - ▶ Crash-course on OpenMP – March 10
- 4 Optimization – intro on March 11
 - ▶ Vectorization tuning – March 14
 - ▶ Multi-threading – March 15, 16
 - ▶ Memory traffic – March 17
- 5 Tools: MKL and VTune – March 18

March 2016						
S	M	T	W	H	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		
■ — Lecture+remote access						
■ — Self-study/remote access						

HOW Online

Course page: colfaxresearch.com/how-16-03

- Slides (including this one), code downloads
- Video of recorded sessions
- Chat (during webinars or offline)



Additional resources:

- More workshops like this one: colfaxresearch.com/how-series
- Video courses: colfaxresearch.com/video-courses
- [Intel Many Integrated Core Architecture Forum](#)

HOT Series

A blue rectangular banner with white and yellow text. The word "HOT" is stylized with yellow flames. The background features faint white geometric patterns. At the bottom of the banner, there is a yellow and orange flame graphic.

THE “**HOT**” (HANDS ON TUTORIAL) SERIES
FREE ONLINE WEBINAR
EFFICIENT PROGRAMMING FOR INTEL® ARCHITECTURE
MARCH 21, 23 & 25
3 webinar series | Filling up fast, register now!

colfaxresearch.com/hot-16-03/

A blue rectangular banner with white and light blue text. The background features faint, light blue geometric patterns of circles and lines. The text is centered and reads: "THE 'HOW' (HANDS ON WORKSHOP) SERIES" in light blue, "FREE ONLINE TRAINING" in large white letters, "PARALLEL PROGRAMMING AND OPTIMIZATION" in white, "FOR INTEL® ARCHITECTURE" in white, and "STARTS APR 18" in light blue. At the bottom, a line of smaller white text says: "*10 2-hour sessions | 24-hour 3-week access to a system | Filling up fast, register now!".

THE "HOW" (HANDS ON WORKSHOP) SERIES

FREE ONLINE TRAINING

PARALLEL PROGRAMMING AND OPTIMIZATION

FOR INTEL® ARCHITECTURE

STARTS APR 18

*10 2-hour sessions | 24-hour 3-week access to a system | Filling up fast, register now!

colfaxresearch.com/how-16-04/

§2. Performance Optimization

Computing Platforms

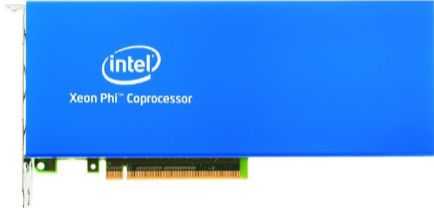
Intel Xeon Processor



Current: Haswell
Upcoming: Broadwell

Multi-Core Architecture

Intel Xeon Phi Coprocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*



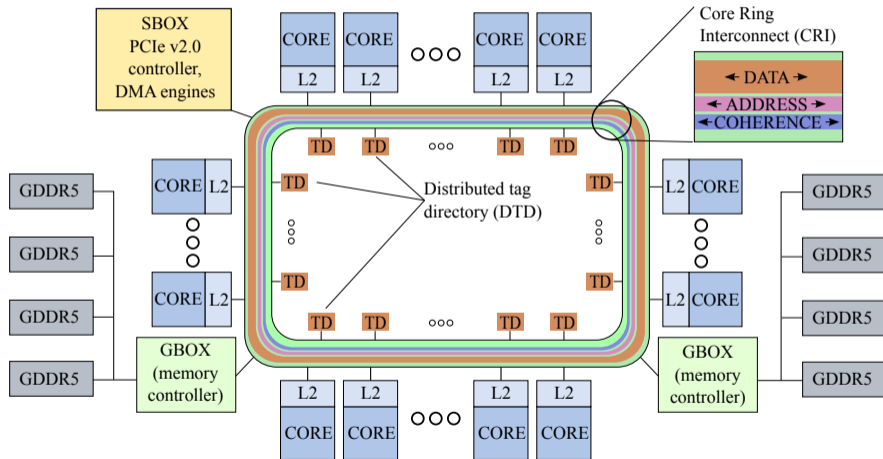
* socket and coprocessor versions

Upcoming: Knights Landing (KNL)

Intel Many Integrated Core (MIC) Architecture

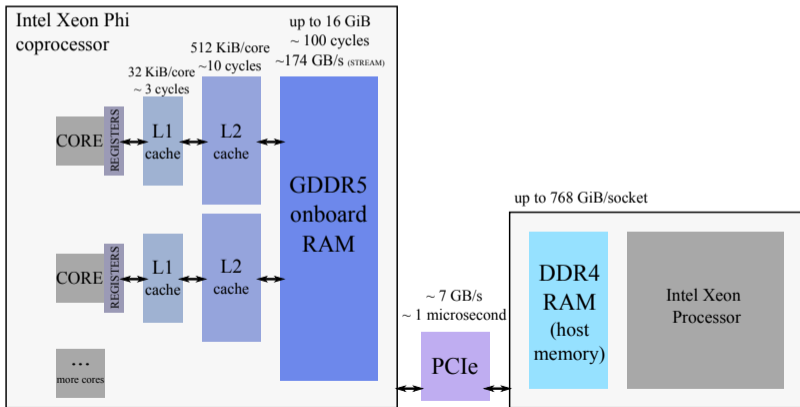
KNC Die Organization

In a ring bus with distributed cache, data access locality is key.



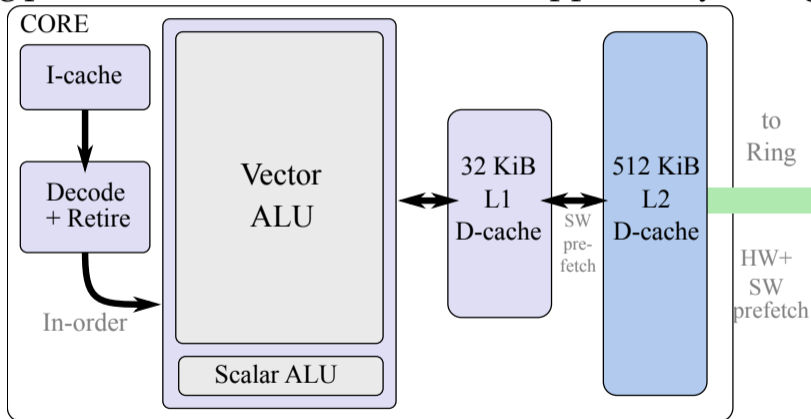
KNC Memory Organization

- Direct access to ≤ 16 GiB of cached GDDR5 memory on board
- No access to system DDR4, connected to host via PCIe



KNC Cores

Computing power is in vector units. Scalar support only for legacy usage.



Optimization Areas

- 1 **Scalar optimization** (compiler-friendly practices)
- 2 **Vectorization** (must use 16- or 8-wide vectors)
- 3 **Multi-threading** (must scale to 100+ threads)
- 4 **Memory access** (streaming access or tiling)
- 5 **Communication** (offload, MPI traffic control)

§3. N-body Simulation

N-body Simulation on CPU and Coprocessor



N-body simulation on...

Two
Intel® Xeon®
CPUs



One
Intel® Xeon Phi™
coprocessor



Two
Intel® Xeon Phi™
coprocessors



Paper: <http://xeonphi.com/papers/nbody-basic>

Demo: [click here](#)

Physics

Gravitational N-body dynamics:

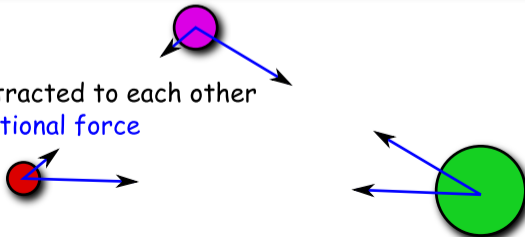
Newton's law of universal gravitation:

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

where:

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

particles are attracted to each other
with **the gravitational force**



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)

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

Each particle is stored as a structure:

```
1 struct ParticleType {  
2     float x, y, z;  
3     float vx, vy, vz;  
4 };
```

main() allocates an array of ParticleType:

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

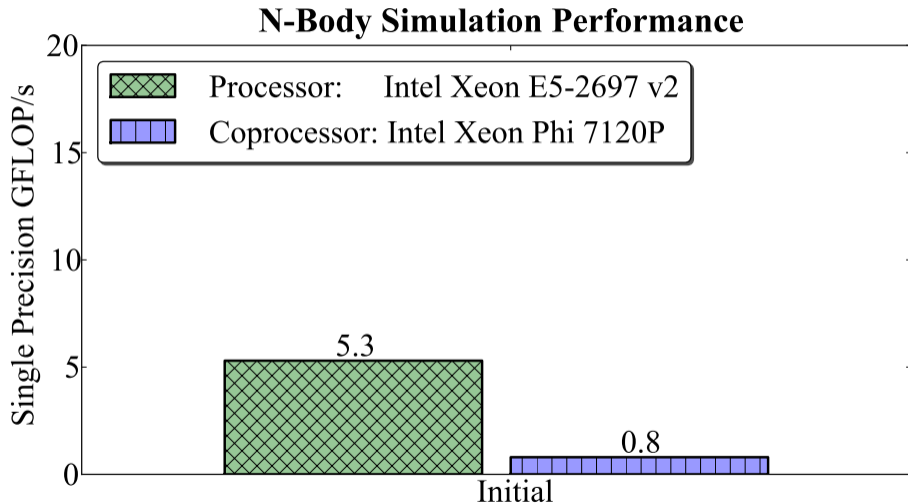
Particle propagation step is timed:

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

Particle Update Engine

```
1 void MoveParticles(int nParticles, ParticleType* particle, float dt) {
2     for (int i = 0; i < nParticles; i++) { // Particles that experience force
3         float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
4         for (int j = 0; j < nParticles; j++) { // Particles that exert force
5             // Newton's law of universal gravity
6             const float dx = particle[j].x - particle[i].x;
7             const float dy = particle[j].y - particle[i].y;
8             const float dz = particle[j].z - particle[i].z;
9             const float drSquared = dx*dx + dy*dy + dz*dz + 1e-20;
10            const float drPower32 = pow(drSquared, 3.0/2.0);
11            // Calculate the net force
12            Fx += dx/drPower32; Fy += dy/drPower32; Fz += dz/drPower32;
13        }
14        // Accelerate particles in response to the gravitational force
15        particle[i].vx+=dt*Fx; particle[i].vy+=dt*Fy; particle[i].vz+=dt*Fz;
16    }
17    ...
}
```

Performance of Initial Implementation



Incorporating Thread Parallelism

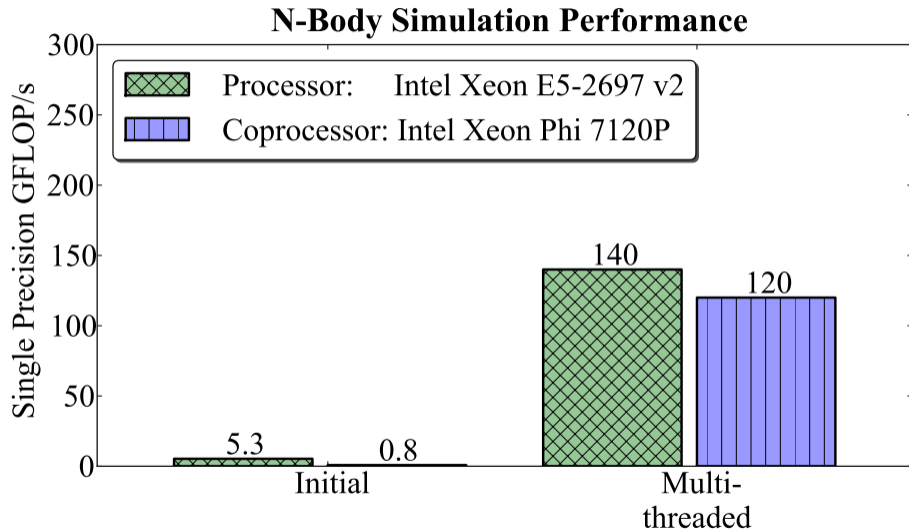
Before:

```
1  for (int i = 0; i < nParticles; i++) { // Particles that experience force
2      float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
3      for (int j = 0; j < nParticles; j++) { // Particles that exert force
4          // Newton's law of universal gravity
5          ...
```

After:

```
1  #pragma omp parallel for
2      for (int i = 0; i < nParticles; i++) { // Particles that experience force
3          float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
4          for (int j = 0; j < nParticles; j++) { // Particles that exert force
5              // Newton's law of universal gravity
6              ...
```

Performance with Thread Parallelism



Vectorizing with Unit-Stride Memory Access

Before:

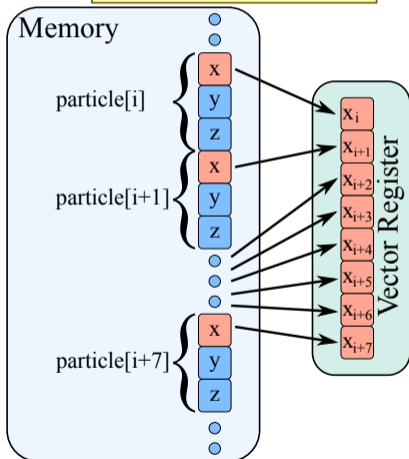
```
1 struct ParticleType {  
2     float x, y, z, vx, vy, vz;  
3 }; // ...  
4     const float dx = particle[j].x - particle[i].x;  
5     const float dy = particle[j].y - particle[i].y;  
6     const float dz = particle[j].z - particle[i].z;
```

After:

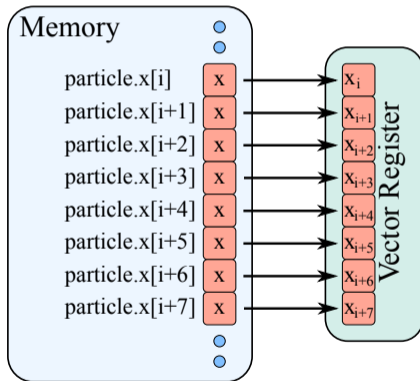
```
1 struct ParticleSet {  
2     float *x, *y, *z, *vx, *vy, *vz;  
3 }; // ...  
4     const float dx = particle.x[j] - particle.x[i];  
5     const float dy = particle.y[j] - particle.y[i];  
6     const float dz = particle.z[j] - particle.z[i];
```

Why AoS to SoA Conversion Helps: Unit Stride

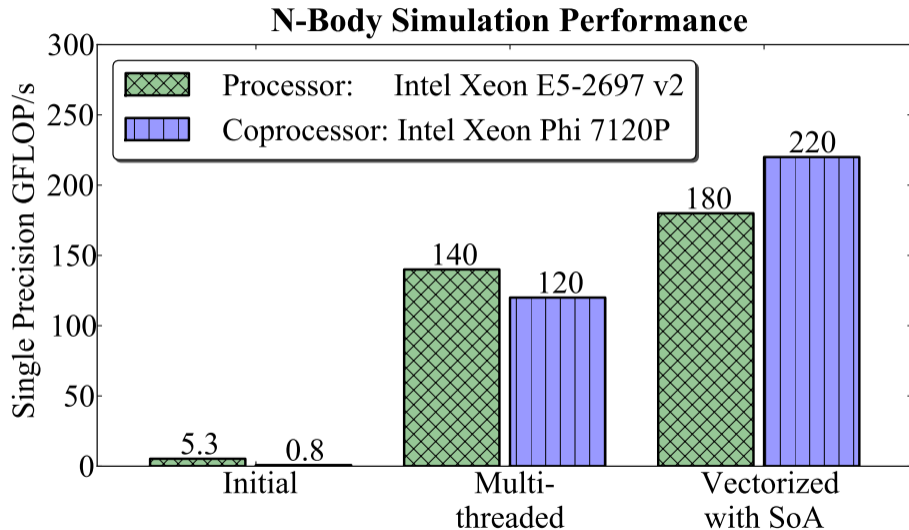
Array of Structures
(sub-optimal)



Structure of Arrays
(optimal)



Performance with Improved Vectorization



Improving Scalar Expressions

Before:

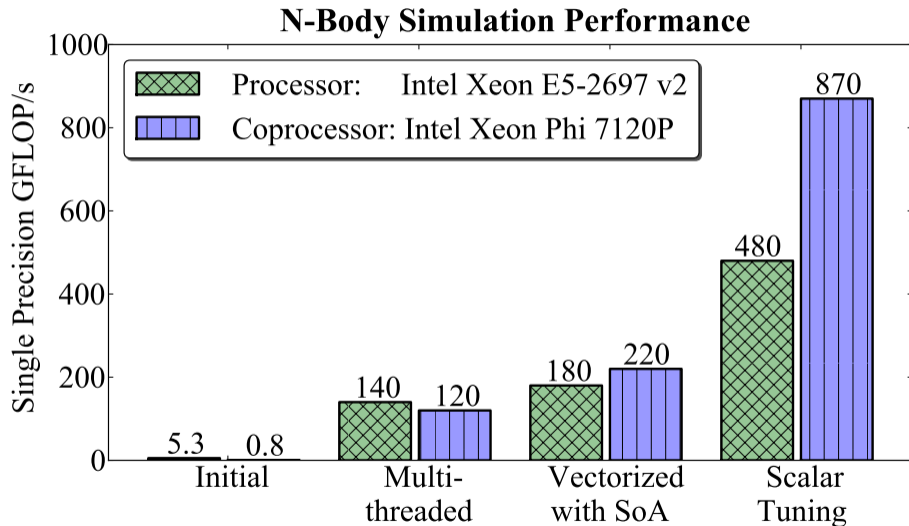
```
1  const float drSquared = dx*dx + dy*dy + dz*dz + 1e-20;  
2  const float drPower32 = pow(drSquared, 3.0/2.0);  
3  // Calculate the net force  
4  Fx += dx/drPower32;  Fy += dy/drPower32;  Fz += dz/drPower32;
```

After:

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

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

Performance after Scalar Tuning



Improving Cache Traffic

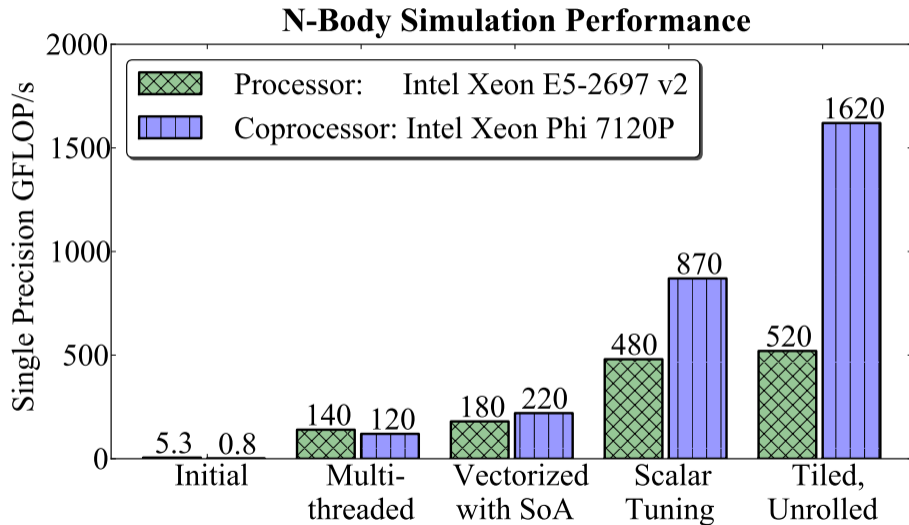
Before:

```
1  for (int i = 0; i < nParticles; i++) { // Particles that experience force
2      float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
3      for (int j = 0; j < nParticles; j++) { // Particles that exert force
4          // ...
5          Fx += dx*drPowerN32; Fy += dy*drPowerN32; Fz += dz*drPowerN32;
```

After: (tileSize = 16)

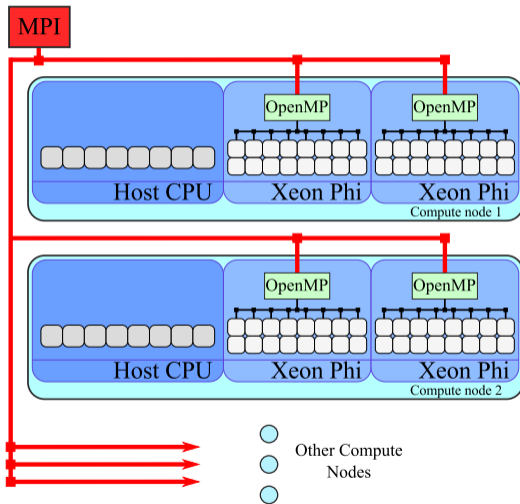
```
1  for (int ii = 0; ii < nParticles; ii += tileSize) { // Particle blocks
2      float Fx[tileSize], Fy[tileSize], Fz[tileSize]; // Force on particle block
3      Fx[:] = Fy[:] = Fz[:] = 0;
4      #pragma unroll(tileSize)
5      for (int j = 0; j < nParticles; j++) { // Particles that exert force
6          for (int i = ii; i < ii + tileSize; i++) { // Traverse the block
7              // ...
8          Fx[i-ii] += dx*drPowerN32; Fy[i-ii] += dy*drPowerN32; Fz[i-ii] += dz*drPowerN32;
```

Performance with Cache Optimization (Loop Tiling)



Scaling Across a Cluster with Coprocessors with MPI

- We put MPI processes only on CPUs
- Subdivide particles between coprocessors
- Concurrent offload from multiple host threads
- Synchronize data between CPUs MPI_Allgather



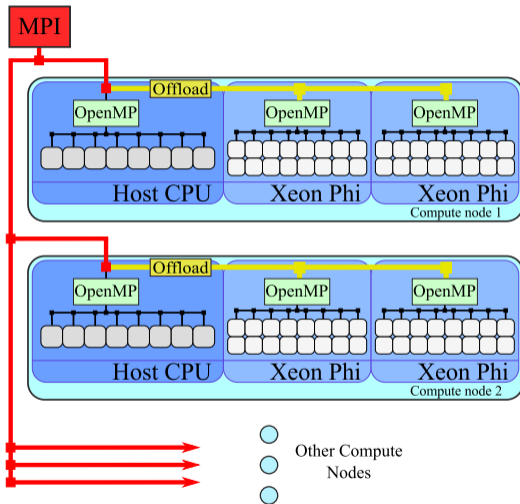
Core of MPI-only Implementation

Simple: all particles on each compute node; exchange updated particle coordinates.

```
1 void MoveParticles(int nParticles, ParticleSet& particle, float dt,
2                   int mpiRank, int mpiWorldSize) {
3     const int myParticles = nParticles/mpiWorldSize;
4     const int startParticle = (mpiRank    )*myParticles;
5     const int endParticle   = (mpiRank + 1)*myParticles;
6     // Outer loop over only the subset of particles processed by present process
7     #pragma omp parallel for schedule(guided)
8     for (int ii = startParticle; ii < endParticle; ii += tileSize) {
9         for (int j = 0; j < nParticles; j++) // ...But inner loop over all particles
10            //...
11    }
12    // ... Propagate results of time step across the cluster
13    MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, particle.x,
14                 myParticles, MPI_FLOAT, MPI_COMM_WORLD);
15    // ...
```

Scaling Across a Cluster with Coprocessors

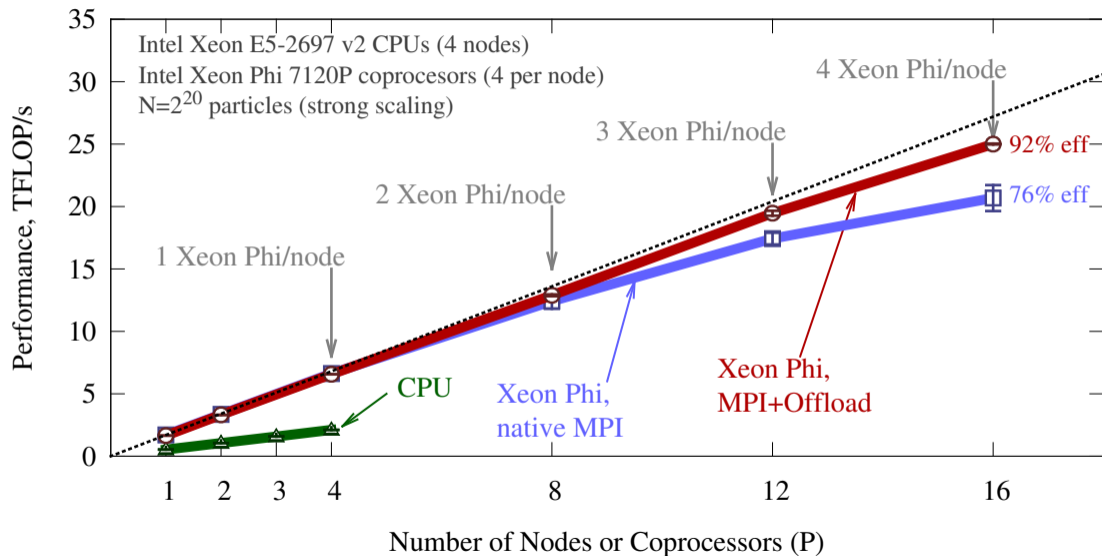
- We put MPI processes only on CPUs
- Subdivide particles between coprocessors
- Concurrent offload from multiple host threads
- Synchronize data between CPUs MPI_Allgather



MPI with offload implementation

```
1  const int nDevices = _Offload_number_of_devices();
2  const int particlesPerDevice=(nDevices==0 ? myParticles : myParticles/nDevices);
3  #pragma omp parallel num_threads(nDevices) if(nDevices>0)
4  {
5      const int iDevice = omp_get_thread_num();
6      const int startParticle = rankStartParticle + (iDevice )*particlesPerDevice;
7      #pragma offload target(mic:iDevice) if(nDevices>0)          \
8          in (x : length(nParticles)          alloc_if(alloc==1) free_if(0)) \
9          out(x [startParticle:particlesPerDevice] : alloc_if(0) free_if(alloc==-1)) \
10         in (vx: length(nParticles*alloc*alloc)          alloc_if(alloc==1) free_if(0)) \
11         //...
12         { // Loop over particles that experience force
13     #pragma omp parallel for schedule(guided)
14         for (int ii = startParticle; ii < endParticle; ii += tileSize) {
15             // ...
```

Results with MPI+Offload

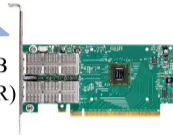


Comparative Benchmarks and System Configuration

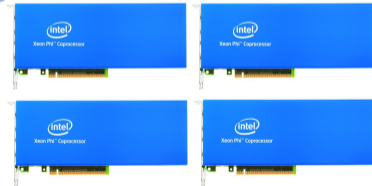
Colfax ProEdge SXP8600p
rack-mountable workstations (cluster of 4)



Mellanox Connect-IB
InfiniBand HCA (FDR)



Intel Xeon Phi 7120P coprocessors
(4 per system)

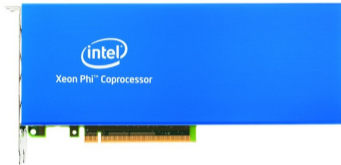


Dual-socket Intel Xeon E5-2697 v2
processor



<http://xeonphi.com/workstations>

Coprocessor vs Processor Performance



One Intel Xeon Phi 7120P
coprocessor

vs.



Two Intel Xeon E5-2697 v2
CPUs

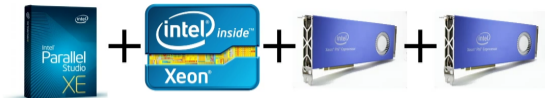
- Why compare 1 coprocessor against 2 processors?
Same thermal design power (TDP).

See also [“Intel Xeon Product Family: Performance Brief”](#)

Additional Case Studies

Astrophysics

Astrophysical Code HEATCODE: an Offload Story

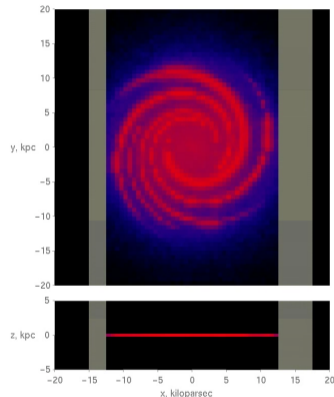


Porting to Intel® Xeon Phi™ coprocessors

- We ported Frankie code using explicit offload model
- Same code & optimization methods for Xeon Phi™
- Simultaneous calculations on CPU and coprocessors with automatic load balancing was easy to implement
- With two Intel® Xeon Phi™ coprocessors, performance for high-res calculations is 3.2x better than with two Intel® Xeon® E5 processors alone.
- **RESULT:** estimated target project calculation time is now 2 weeks (down from 6+ years)

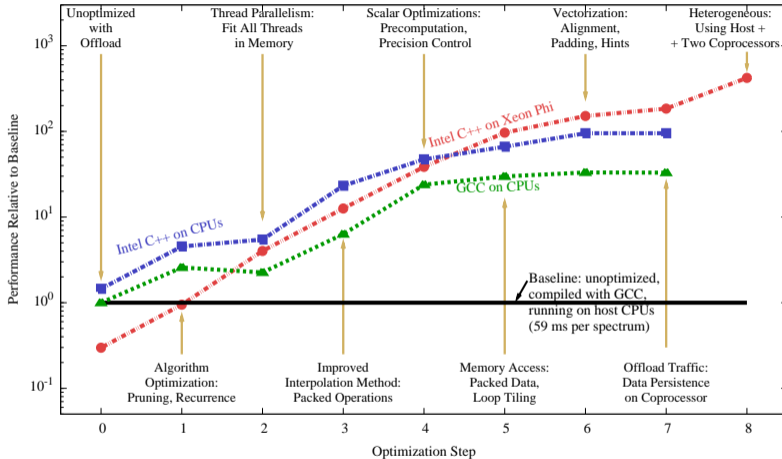
Goal achieved!

Transient Emission of Cosmic Dust Grains
in the Milky Way Galaxy,
Simulation with Frankie Code



<http://xeonphi.com/papers/heatcode>

Astrophysical Code HEATCODE: an Offload Story



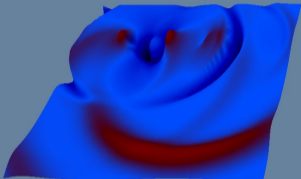
<http://xeonphi.com/papers/heatcode>

Fluid Dynamics

Computational Fluid Dynamics: Legacy Code

FLUID DYNAMICS WITH FORTRAN ON INTEL® XEON PHI™ COPROCESSORS

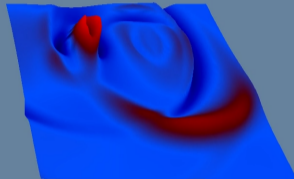
SHALLOW WATER EQUATION SOLVER



PERFORMANCE ON CPU: 19.5 GFLOP/S

SAME CODE FOR CPU AND XEON PHI
FORTRAN 90 + OPENMP + MPI

PUBLICATION:
XEONPHI.COM/PAPERS/SHALLOW



PERFORMANCE WITH COPROCESSORS: 52.5 GFLOP/S

SIMULATION SIZE: 9600X9600

ACCELERATION: 2.7X



INTEL XEON E5-2697 V3 PROCESSOR



INTEL XEON E5-2697 V3 PROCESSOR +
TWO INTEL XEON PHI 7120A COPROCESSORS



SERVERS WORKSTATIONS TRAINING CONSULTING RESEARCH

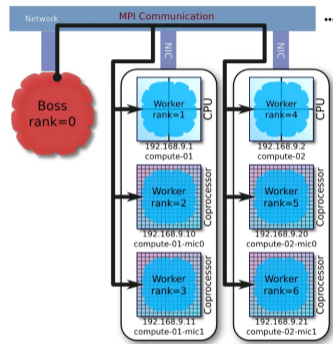
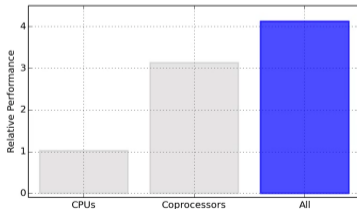
WWW.COLFAX-INTL.COM

<http://xeonphi.com/papers/shallow>

Financial Applications

Asian Option Pricing: Heterogeneous Clustering

Heterogeneous Clustering with Homogeneous Code:
Asian Option Pricing



<http://xeonphi.com/papers/heterogeneous>

Summary

Areas of optimization of applications for Intel Xeon and Intel Xeon Phi processors:

- 1 **Scalar optimization** (compiler-friendly practices)
- 2 **Vectorization** (must use 16- or 8-wide vectors)
- 3 **Multi-threading** (must scale to 100+ threads)
- 4 **Memory access** (streaming access or tiling)
- 5 **Communication** (offload, MPI traffic control)

Next session: optimization of vectorization.