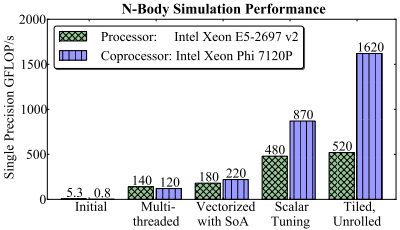
Primer on Computing with Intel Xeon Phi Coprocessors

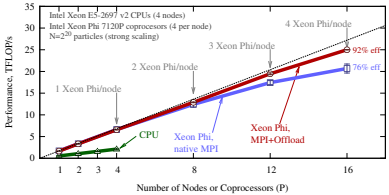
Part 2 of 2: Optimization Example

Andrey Vladimirov and Vadim Karpusenko
Colfax International

Intel HPC Developer Conference at SC14 in New Orleans, LA — November 16, 2014

Contents



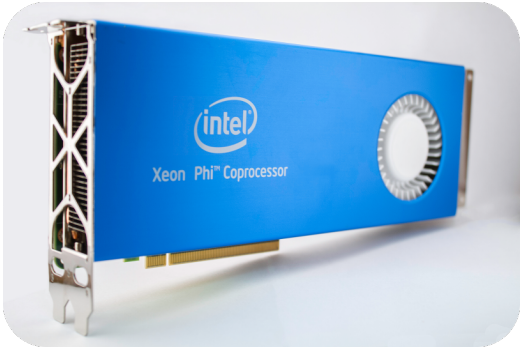


§1. Recap of Part 1

Purpose of Xeon Phi

Intel Xeon Phi Coprocessors and the MIC Architecture

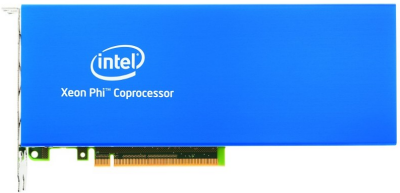
- PCIe end-point device
- High Power efficiency
- ~ 1 TFLOP/s in DP
- Heterogeneous clustering



For highly parallel applications which reach the scaling limits on Intel Xeon processors

Intel Xeon Phi Coprocessors and the MIC Architecture

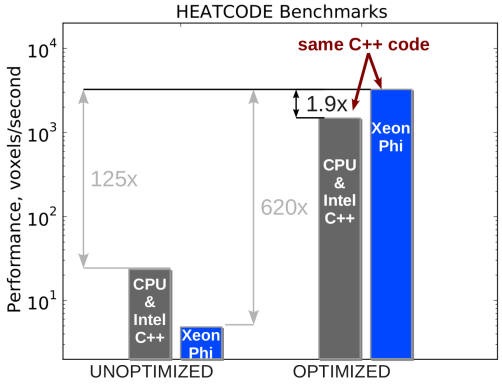
- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GB of DDR3 RAM
- ≤12 cores/socket ≈3 GHz
- 2-way hyper-threading
- 256-bit AVX vectors



- C/C++/Fortran; OpenMP/MPI
- Special Linux μOS distribution
- 6–16 GB cached GDDR5 RAM
- 57 to 61 cores at ≈1 GHz
- 4 hardware threads per core
- 512-bit IMCI vectors

Performance Expectations: “Two Birds with One Stone”

- Performance will be disappointing if code is not optimized for multi-core CPUs
- Optimized code runs better on the MIC platform *and* on the multi-core CPU
- Single code for two platforms + Ease of porting = Incremental optimization



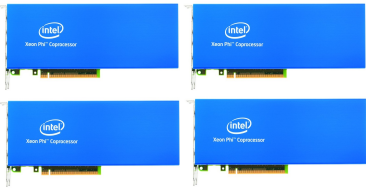
Case study:
xeonphi.com/papers/heatcode

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)



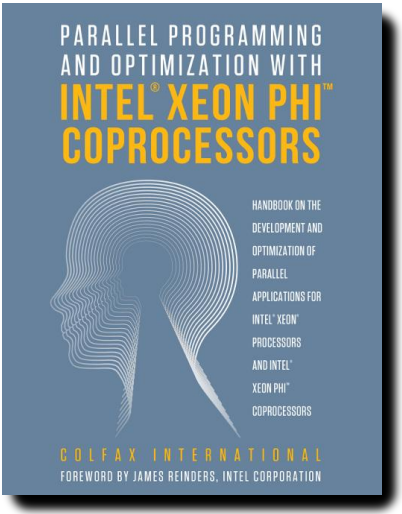
Dual-socket Intel Xeon E5-2697 v2
processor

xeonphi.com/workstations

Optimization Essentials

Optimization Checklist

1. Multi-threading
2. Vectorization
3. Scalar component
4. Memory and cache traffic
5. Communication



§2. Example: N-body Simulation

Problem Statement

Physics

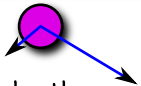
Gravitational N-body dynamics:

Newton's law of universal gravitation:

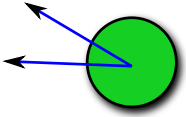
$$M_i \vec{R}_i''(t) = G \sum_j \frac{M_i M_j}{\left|\vec{R}_i - \vec{R}_j\right|^3} \left(\vec{R}_j - \vec{R}_i\right)$$

where:

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



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)

Prior Work on Toy Model

At SC'12 (Xeon Phi released) we showed a simple N-body demo later described in `xeonphi.com/papers/nbody-basic`

Test-driving Intel® Xeon Phi™ coprocessors with a basic N-body simulation

Andrey Vladimirov
Stanford University
and
Vadim Karpusenko
Colfax International

January 7, 2013

Abstract

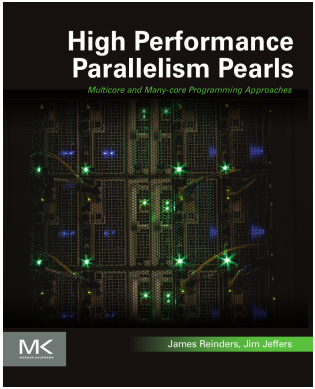
Intel® Xeon Phi™ coprocessors are capable of delivering more performance and better energy efficiency than Intel® Xeon® processors for certain parallel applications[1]. In this paper, we investigate the porting and optimization of a test problem for the Intel Xeon Phi coprocessor. The test problem is a basic N-body simulation, which is the foundation of a number of applications in computational astrophysics and biophysics. Using common code in the C language for the host processor and for the coprocessor, we benchmark the N-body simulation. The simulation runs 2.3x to 5.4x times faster on a single Intel Xeon Phi coprocessor than on two Intel Xeon E5 series processors. The performance depends on the accuracy settings for transcendental arithmetics. We also study the assembly code produced by the compiler from the C code. This allows us to pinpoint some strategies for designing C/C++ programs that result in efficient automatically vectorized applications for Intel Xeon family devices.

Contents

Follow-Ups on Toy Model

Andreas Stiller followed up on our paper in 2013 and compared coprocessor and GPU performance in an article in the German magazine CT.

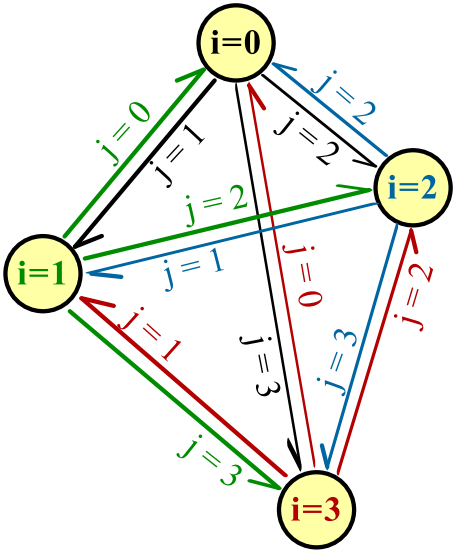
Alejandro Duran and Larry Meadows published in 2014 on improving the performance of our N-body code in Chapter 9 of “High Performance Parallelism Pearls”.



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 ($O(n^2)$ 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   ...
```


Compilation and Execution

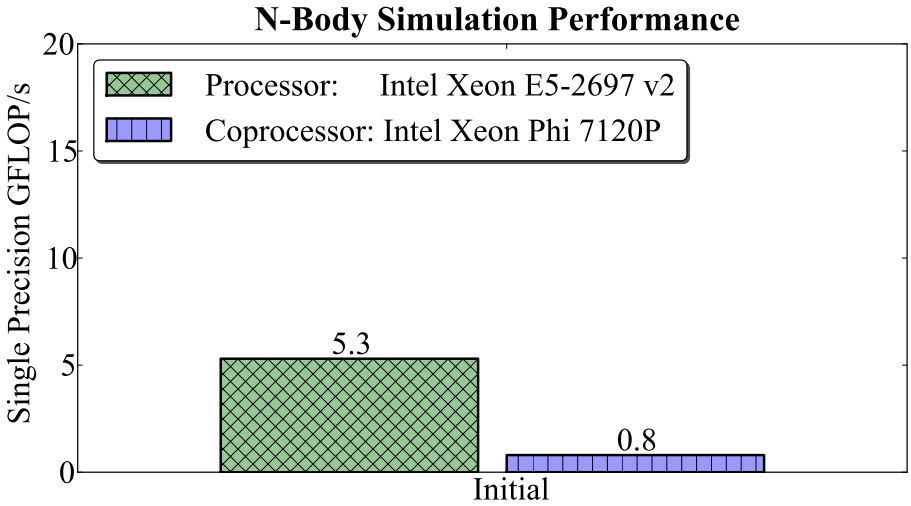
For the CPU architecture (Intel Xeon E5-2697 v2 processor):

```
vega@lyra% # Compile for host CPU:
vega@lyra% icpc -o nbody-CPU -xhost nbody.cc
vega@lyra% # Launch executable on the host system:
vega@lyra% ./nbody-CPU
```

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

```
vega@lyra% # Compile for Xeon Phi coprocessor with flag -mmic:
vega@lyra% icpc -o nbody-MIC -mmic nbody.cc
vega@lyra% # Launch native executable directly on Xeon Phi:
vega@lyra% export SINK_LD_LIBRARY_PATH=$MIC_LD_LIBRARY_PATH
vega@lyra% micnativeloadex ./nbody-MIC # Launch on Xeon Phi
```

Performance of Initial Implementation



§3. Node-Level Optimization

Optimization of Thread Parallelism

Incorporating Thread Parallelism

Before:

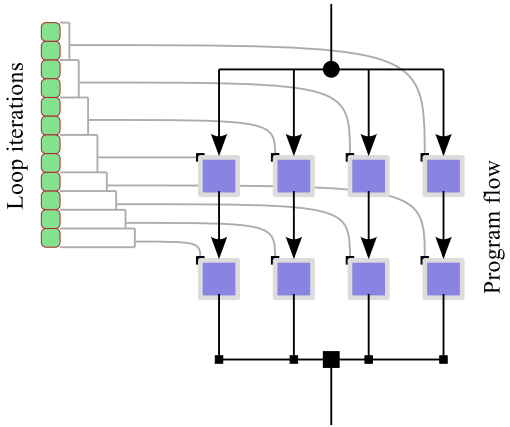
```
1  for (int i = 0; i < nParticles; i++) {   // Particles that experience force
2    float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
3    for (int j = 0; j < nParticles; j++) { // Particles that exert force
4      // Newton's law of universal gravity
5      ...
```

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      ...
```

How Parallel OpenMP Loops Work

- Program starts in *initial thread*
- `#pragma omp parallel` spawns multiple threads
- Threads may be bound to physical/logical cores
- `#pragma omp for` coordinates threads to process different loop iterations
- Scheduling of iterations across threads may be controlled by user



Compilation with OpenMP

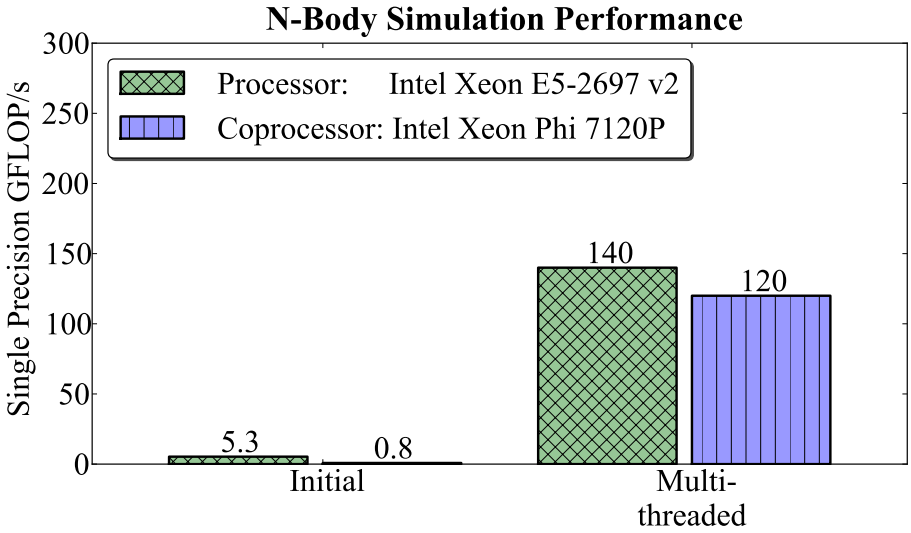
For the CPU architecture (Intel Xeon E5-2697 v2 processor):

```
vega@lyra% # Compile with OpenMP support (-qopenmp):
vega@lyra% icpc -o nbody-CPU -xhost -qopenmp nbody.cc
vega@lyra% # Prevent thread migration across cores:
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 OpenMP support (-qopenmp):
vega@lyra% icpc -o nbody-MIC -mmic -qopenmp nbody.cc
vega@lyra% export SINK_LD_LIBRARY_PATH=$MIC_LD_LIBRARY_PATH
vega@lyra% # Launch on Xeon Phi, preventing thread migration across cores:
vega@lyra% micnativeloadex ./nbody-MIC -e "KMP_AFFINITY=compact"
```

Performance with Thread Parallelism



Optimization of Vectorization

Vectorizing with Unit-Stride Memory Access

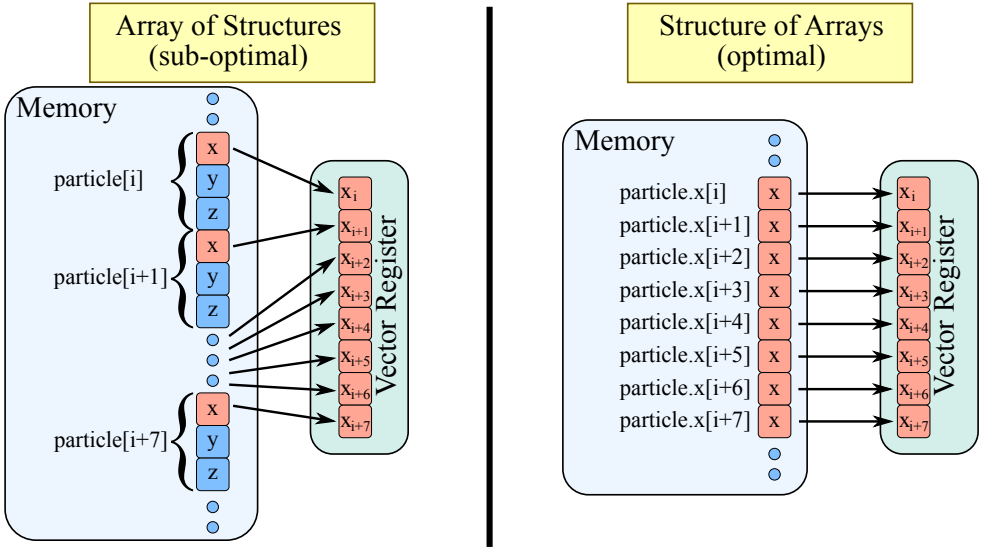
Before:

```
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;
```

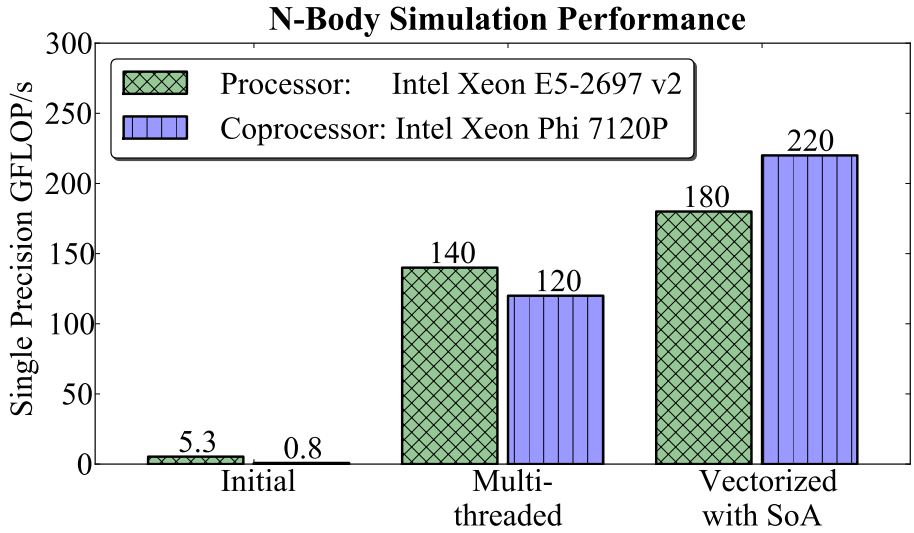
After:

```
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



Performance with Improved Vectorization



Scalar Tuning

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

Compilation with Relaxed Accuracy

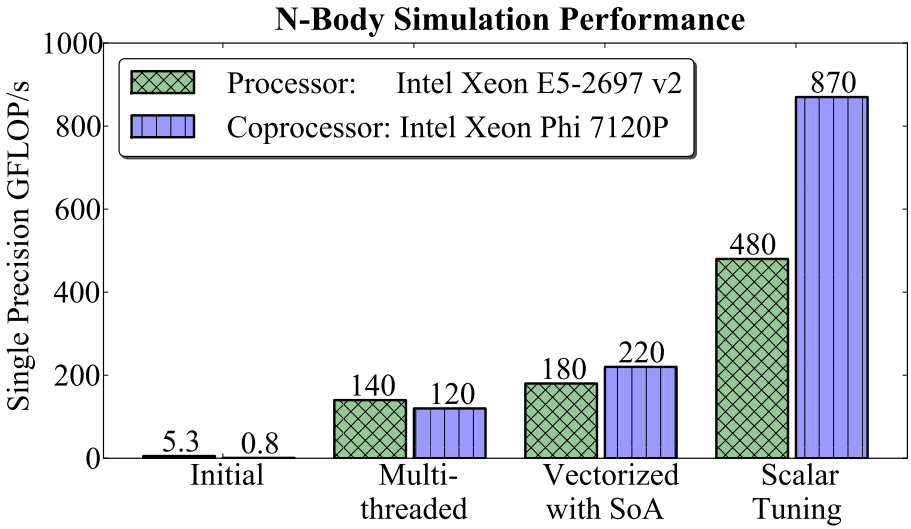
For the CPU architecture (Intel Xeon E5-2697 v2 processor):

```
vega@lyra% # Compile with relaxed accuracy: (-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 accuracy: (-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



Cache Traffic Optimization

Improving Cache Traffic

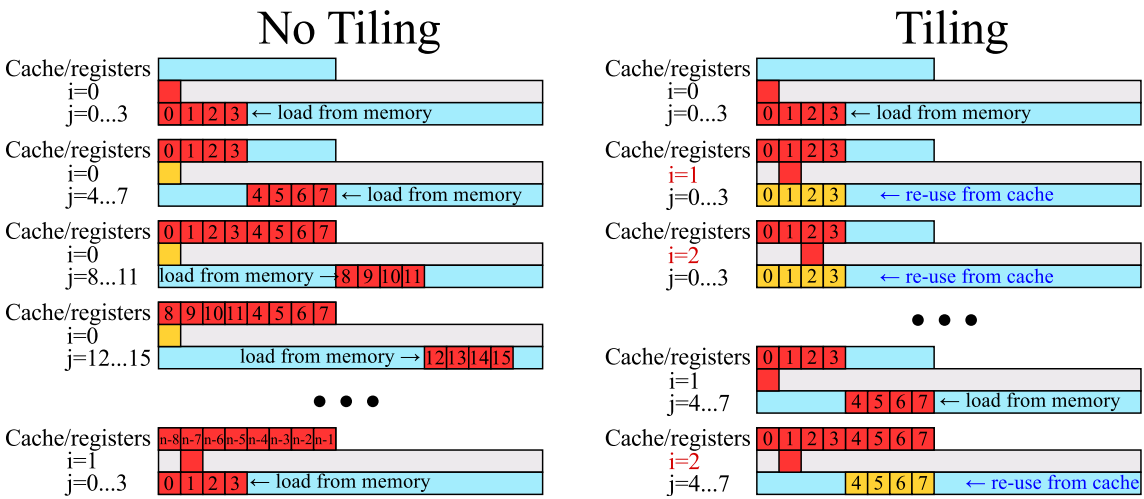
Before:

```
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;
```

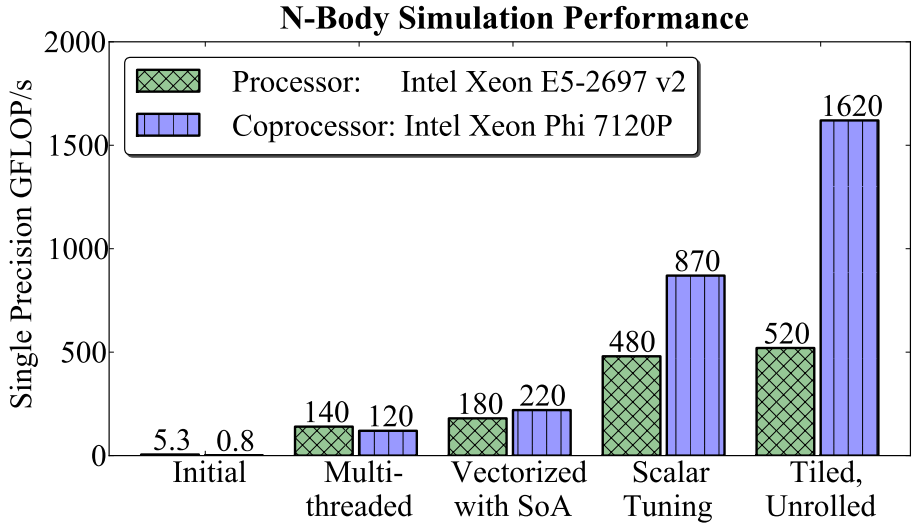
After: (`tileSize = 16`)

```
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;
```

Why Loop Tiling (Blocking) Helps



Performance with Cache Optimization (Loop Tiling)

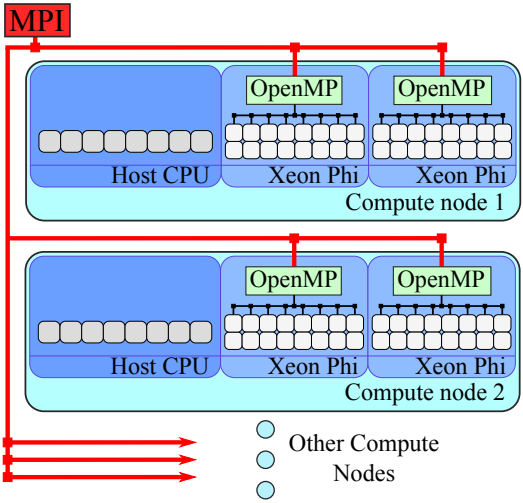


§4. Cluster-Level Optimization

Native MPI

Scaling Across a Cluster with Coprocessors

- We put MPI processes directly on coprocessors
- All particle coordinates on each coprocessor
- Synchronize data after every step using `MPI_Allgather`

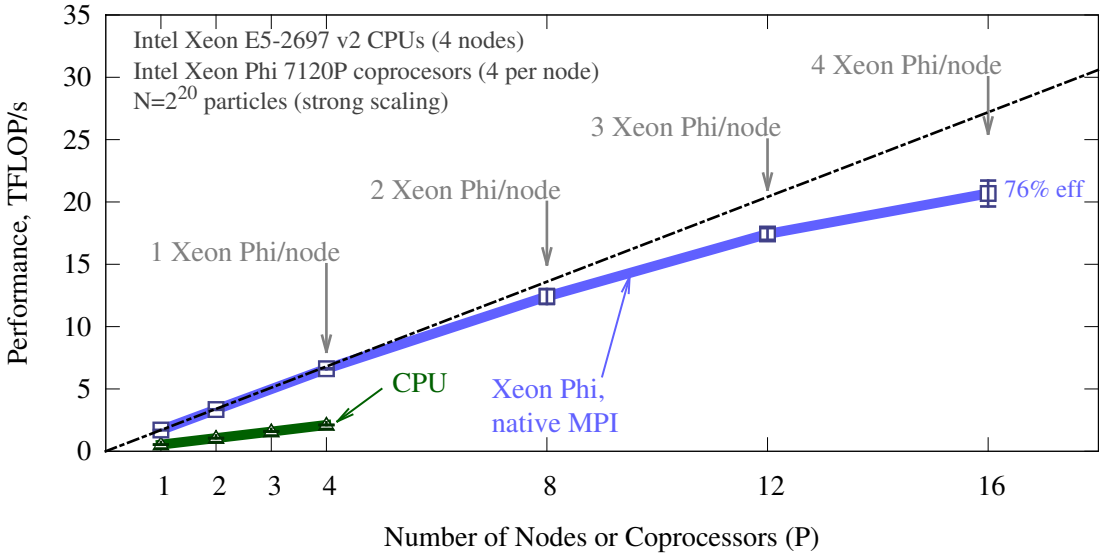


Core of MPI implementation

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

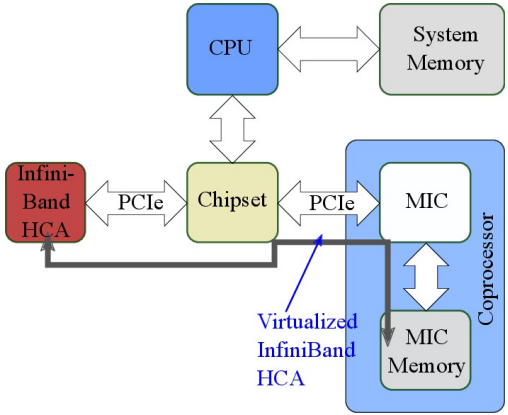
```
1  void MoveParticles(int nParticles, ParticleSet& particle, float dt,
2                     int mpiRank, int mpiWorldSize) {
3    const int myParticles   = nParticles/mpiWorldSize;
4    const int startParticle = (mpiRank    )*myParticles;
5    const int endParticle   = (mpiRank + 1)*myParticles;
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   // ...
```

Results with Native MPI



How Peer to Peer Messaging Works with Xeon Phi

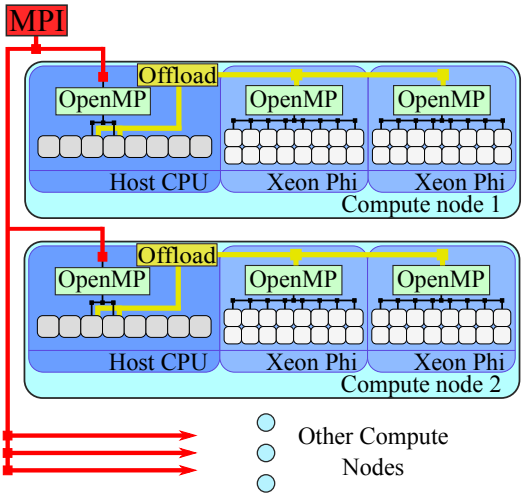
- InfiniBand requires additional software on top of MPSS
- Environment variable `I_MPI_FABRICS`
- More information in `xeonphi.com/papers/p2p`



MPI+Offload

Scaling Across a Cluster with Coprocessors

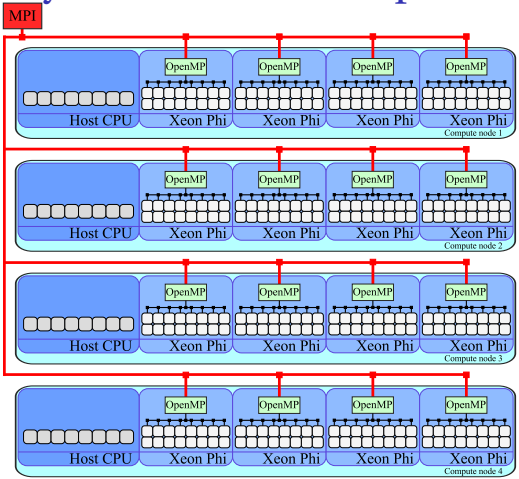
- We put MPI processes only on CPUs
- Subdivide particles between coprocessors
- Concurrent offload from multiple host threads
- Synchronize data between CPUs `MPI_Allgather`



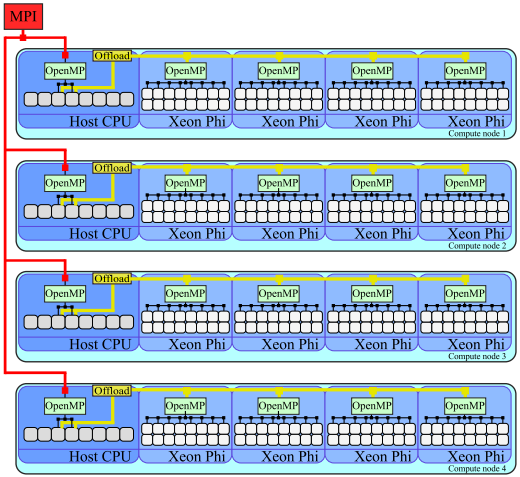
MPI with offload implementation

```
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          // ...
```

Why MPI+Offload Helps: Fewer MPI End-Points

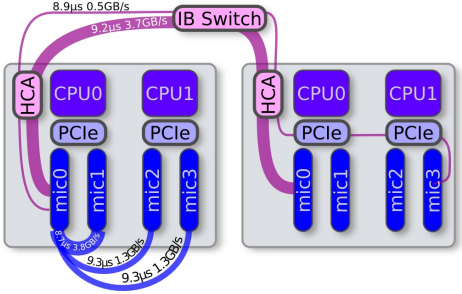


Native MPI: $4 \times P$ end-points for all-to-all `MPI_Allgather`

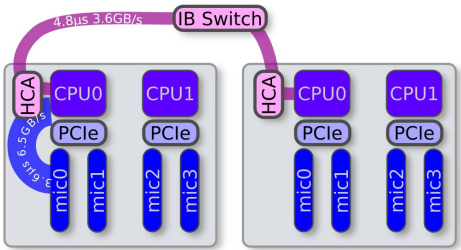


MPI+Offload: P end-points for all-to-all `MPI_Allgather`

Why MPI+Offload Helps: Optimal Communication Paths



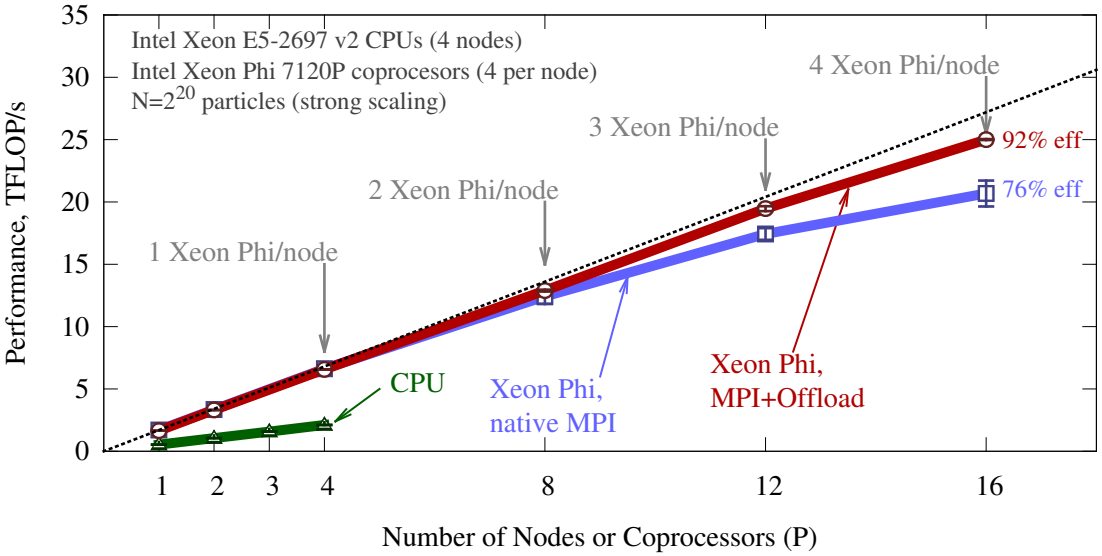
Native MPI: peer-to-peer communication across QPI is throttled (single-HCA configurations w/Ivy Bridge)



MPI+Offload: offload traffic and InfiniBand between hosts are high-bandwidth communication paths

More information: xeonphi.com/papers/p2p

Results with MPI+Offload



§5. Net Total

Effort and Results

- Node-level optimization (→ 67% theoretical peak performance):
  a) Multi-threading (OpenMP)
  b) Vectorization (data structures of unit stride)
  c) Scalar tuning (functions, precision)
  d) Cache traffic (loop tiling, unrolling)

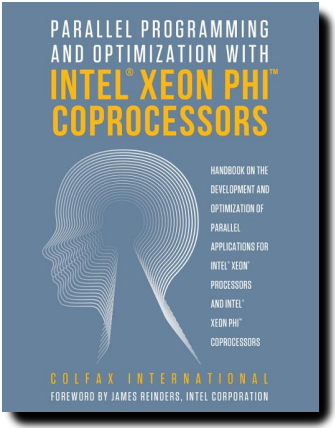
- Cluster-level optimization (→ 92% parallel efficiency):
  a) Scaling native application across cluster (MPI)
  b) Combining MPI traffic with offload

- **Single code optimized for both CPU and MIC arch.**
- **Ready for future generations of Intel parallel platforms**

§6. Additional Resources

Colfax Developer Training

Colfax runs a one-day and four-day trainings for organizations on parallel programming with Intel Xeon Phi co-processors.



xeonphi.com/training

Additional Reading

It all comes down to PARALLEL PROGRAMMING ! (applicable to processors and Intel® Xeon Phi™ coprocessors both)

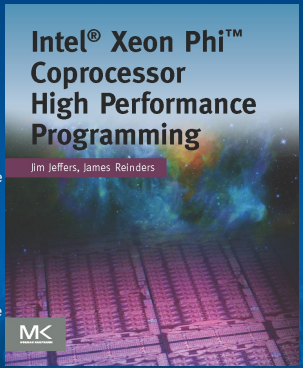
Forward, Preface
Chapters:

1. Introduction
2. High Performance Closed Track Test Drive!
3. A Friendly Country Road Race
4. Driving Around Town: Optimizing A Real-World Code Example
5. Lots of Data (Vectors)
6. Lots of Tasks (not Threads)
7. Offload
8. Coprocessor Architecture
9. Coprocessor System Software
10. Linux on the Coprocessor
11. Math Library
12. MPI
13. Profiling and Timing
14. Summary

Glossary, Index

Learn more about this book:

lotsofcores.com



Available since February 2013.

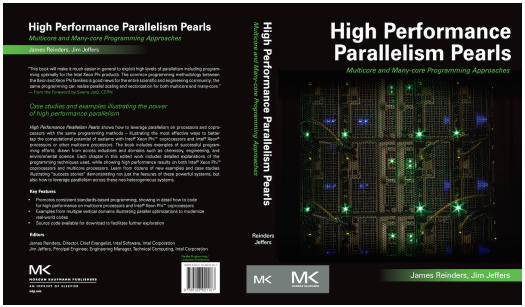
This book belongs on the bookshelf of every HPC professional. Not only does it successfully and accessibly teach us how to use and obtain high performance on the Intel MIC architecture, it is about much more than that. It takes us back to the universal fundamentals of high-performance computing including how to think and reason about the performance of algorithms mapped to modern architectures, and it puts into your hands powerful tools that will be useful for years to come.

—Robert J. Harrison
Institute for Advanced Computational Science, Stony Brook University

Intel® Xeon Phi™ Coprocessor High Performance Programming,
Jim Jeffers, James Reinders, (c) 2013, publisher: Morgan Kaufmann

Additional Reading

Available November 17, 2014



www.lotsofcores.com

- 28 chapters
- 69 expert contributors
- Numerous “Real World” Code “Recipes” and examples using OpenMP, MPI, TBB, C, C++, OpenCL, Fortran.
- Successful techniques, tips for vectorization, scalable parallel coding, load balancing, data structure and memory tuning, applicable to both processors and coprocessors!
- All figures, diagrams and code freely downloadable. (Nov’14)

Thank you for tuning in,
and
have a wonderful journey
to the Parallel World!

http://research.colfaxinternational.com/

P.S.: We are hiring! xeonphi.com/jobs