Parallel Programming and Optimization with Intel Xeon Phi Coprocessors

Colfax Developer Training — One-Day Seminar

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

June 2015, Rev. 19b

About This Document

This document represents the materials of a one-day seminar “Parallel Programming and Optimization with Intel Xeon Phi Coprocessors“ developed and run by Colfax International.

© Colfax International, 2013-2015

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!

xeonphi.com/training

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.

Supplementary Materials

© Colfax International, 2013–2015

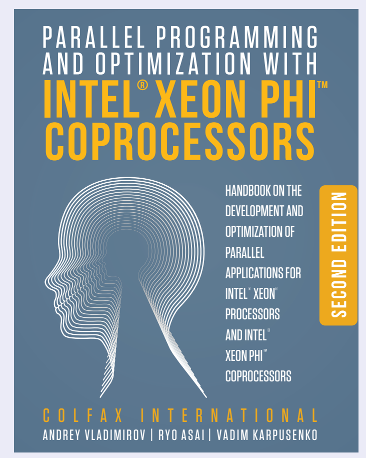
Supplementary Materials: Textbook

ISBN: 978-0-9885234-0-1 (508 pages, Electronic or Print)

Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors

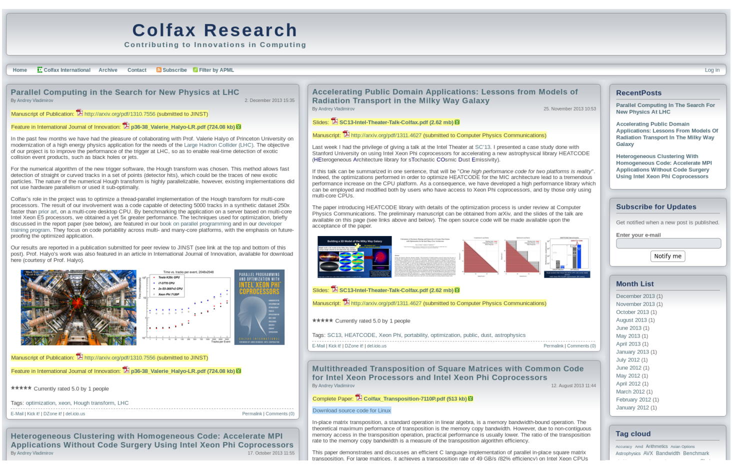
Handbook on the Development and Optimization of Parallel Applications for Intel® Xeon® Processors and Intel® Xeon Phi™ Coprocessors

© Colfax International, 2015



xeonphi.com/book

Research and Consulting



Colfax Research

Contributing to Innovations in Computing

Home | Colfax International | Archive | Contact | Subscribe | Filter by APML | Log in

Parallel Computing in the Search for New Physics at LHC

By Andrey Vladimirov — 2. December 2013 15:35

Manuscript of Publication: http://arxiv.org/pdf/1310.7556 (submitted to JINST)

Feature in International Journal of Innovation: p36-38_Valerie_Halyo-LR.pdf (724.08 kb)

In the past few months we have had the pleasure of collaborating with Prof. Valerie Halyo of Princeton University on modernization of a high energy physics application for the needs of the Large Hadron Collider (LHC). The objective of our project is to improve the performance of the trigger at LHC, so as to enable real-time detection of exotic collision event products, such as black holes or jets.

For the numerical algorithm of the new trigger software, the Hough transform was chosen. This method allows fast detection of straight or curved tracks in a set of points (detector hits), which could be the traces of new exotic particles. The nature of the numerical Hough transform is highly parallelizable, however, existing implementations did not use hardware parallelism or used it sub-optimally.

Colfax's role in the project was to optimize a thread-parallel implementation of the Hough transform for multi-core processors. The result of our involvement was a code capable of detecting 5000 tracks in a synthetic dataset 250x faster than prior art, on a multi-core desktop CPU. By benchmarking the application on a server based on multi-core Intel Xeon E5 processors, we obtained a yet 5x greater performance. The techniques used for optimization, briefly discussed in the report paper (see below), are featured in our book on parallel programming and in our developer training program. They focus on code portability across multi- and many-core platforms, with the emphasis on future-proofing the optimized application.

Our results are reported in a publication submitted for peer review to JINST (see link at the top and bottom of this post). Prof. Halyo's work was also featured in an article in International Journal of Innovation, available for download here (courtesy of Prof. Halyo).

Manuscript of Publication: http://arxiv.org/pdf/1310.7556 (submitted to JINST)

Feature in International Journal of Innovation: p36-38_Valerie_Halyo-LR.pdf (724.08 kb)

Currently rated 5.0 by 1 people

Tags: optimization, xeon, Hough transform, LHC

E-Mail | Kick it! | DZone it! | del.icio.us — Permalink | Comments (0)

Heterogeneous Clustering with Homogeneous Code: Accelerate MPI Applications Without Code Surgery Using Intel Xeon Phi Coprocessors

By Andrey Vladimirov — 17. October 2013 11:55

Accelerating Public Domain Applications: Lessons from Models of Radiation Transport in the Milky Way Galaxy

By Andrey Vladimirov — 25. November 2013 10:53

Slides: SC13-Intel-Theater-Talk-Colfax.pdf (2.62 mb)

Manuscript: http://arxiv.org/pdf/1311.4627 (submitted to Computer Physics Communications)

Last week I had the privilege of giving a talk at the Intel Theater at SC'13. I presented a case study done with Stanford University on using Intel Xeon Phi coprocessors for accelerating a new astrophysical library HEATCODE (HEterogeneous Architecture library for sTochastic COsmic Dust Emissivity).

If this talk can be summarized in one sentence, that will be "*One high performance code for two platforms is reality*". Indeed, the optimizations performed in order to optimize HEATCODE for the MIC architecture lead to a tremendous performance increase on the CPU platform. As a consequence, we have developed a high performance library which can be employed and modified both by users who have access to Xeon Phi coprocessors, and by those only using multi-core CPUs.

The paper introducing HEATCODE library with details of the optimization process is under review at Computer Physics Communications. The preliminary manuscript can be obtained from arXiv, and the slides of the talk are available on this page (see links above and below). The open source code will be made available upon the acceptance of the paper.

Slides: SC13-Intel-Theater-Talk-Colfax.pdf (2.62 mb)

Manuscript: http://arxiv.org/pdf/1311.4627 (submitted to Computer Physics Communications)

Currently rated 5.0 by 1 people

Tags: SC13, HEATCODE, Xeon Phi, portability, optimization, public, dust, astrophysics

E-Mail | Kick it! | DZone it! | del.icio.us — Permalink | Comments (0)

Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors

By Andrey Vladimirov — 12. August 2013 11:44

Complete Paper: Colfax_Transposition-7110P.pdf (513 kb)

Download source code for Linux

In-place matrix transposition, a standard operation in linear algebra, is a memory bandwidth-bound operation. The theoretical maximum performance of transposition is the memory copy bandwidth. However, due to non-contiguous memory access in the transposition operation, practical performance is usually lower. The ratio of the transposition rate to the memory copy bandwidth is a measure of the transposition algorithm efficiency.

This paper demonstrates and discusses an efficient C language implementation of parallel in-place square matrix transposition. For large matrices, it achieves a transposition rate of 49 GB/s (82% efficiency) on Intel Xeon CPUs

RecentPosts

Parallel Computing In The Search For New Physics At LHC

Accelerating Public Domain Applications: Lessons From Models Of Radiation Transport In The Milky Way Galaxy

Heterogeneous Clustering With Homogeneous Code: Accelerate MPI Applications Without Code Surgery Using Intel Xeon Phi Coprocessors

Subscribe for Updates

Get notified when a new post is published.

Enter your e-mail

Notify me

Month List

December 2013 (1)
November 2013 (1)
October 2013 (1)
August 2013 (1)
June 2013 (1)
May 2013 (1)
April 2013 (1)
January 2013 (1)
July 2012 (1)
June 2012 (1)
May 2012 (1)
April 2012 (1)
March 2012 (1)
February 2012 (1)
January 2012 (1)

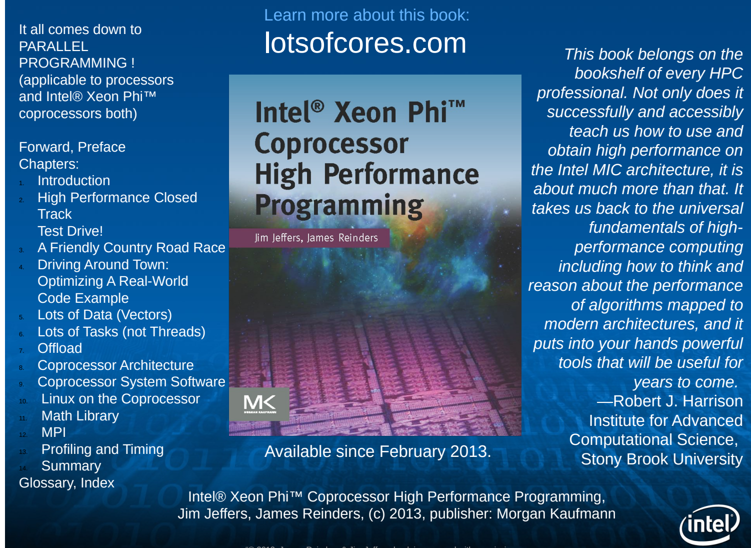
Tag cloud

Accuracy Amd Arithmetics Asian Options Astrophysics AVX Bandwidth Benchmark

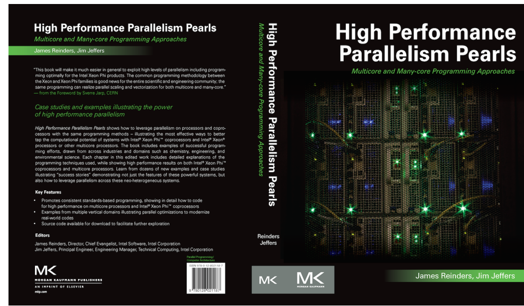
http://research.colfaxinternational.com/

Newsletter subscription – xeonphi.com/newsletter

Additional Reading



Additional Reading



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 code, load balancing, data structure and memory tuning, applicable to both processors and coprocessors!
- All figures, diagrams and code freely downloadable.

List of Topics

1. Welcome
2. Introduction to the Intel Many Integrated Core (MIC) Architecture
3. System Administration for Intel Xeon Phi Coprocessors
4. Models for Intel Xeon Phi Coprocessor Programming
5. Expressing Parallelism on Intel Architectures
6. Optimization Using Intel Software Development Tools
7. Optimization Roadmap
8. Optimization of Scalar Arithmetics
9. Optimization of Vectorization
10. Optimization of Thread Parallelism
11. Optimization of Memory Access
12. Optimization of Communication
13. Additional Topics on MPI
14. Additional Resources

What to Expect: N-body Simulation

Physics

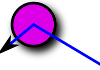
Gravitational N-body dynamics:

Newton's law of universal gravitation:

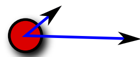
$$M_i \vec{R}_i''(t) = G \sum_j \frac{M_i M_j}{\left|\vec{R}_i - \vec{R}_j\right|^3} \left(\vec{R}_j - \vec{R}_i\right)$$

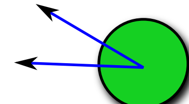
where:

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



particles are attracted to each other with the gravitational force





 © Colfax International, 2013–2015

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:

```
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:

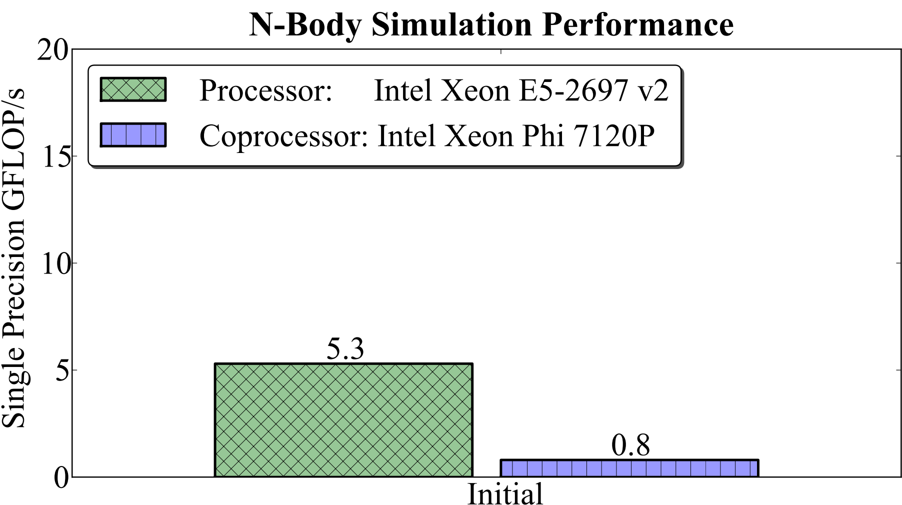
```
const double tStart = omp_get_wtime(); // Start timing
MoveParticles(nParticles, particle, dt);
const double tEnd = omp_get_wtime(); // End timing
```


© Colfax International, 2013–2015

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



 © Colfax International, 2013–2015

Incorporating Thread Parallelism

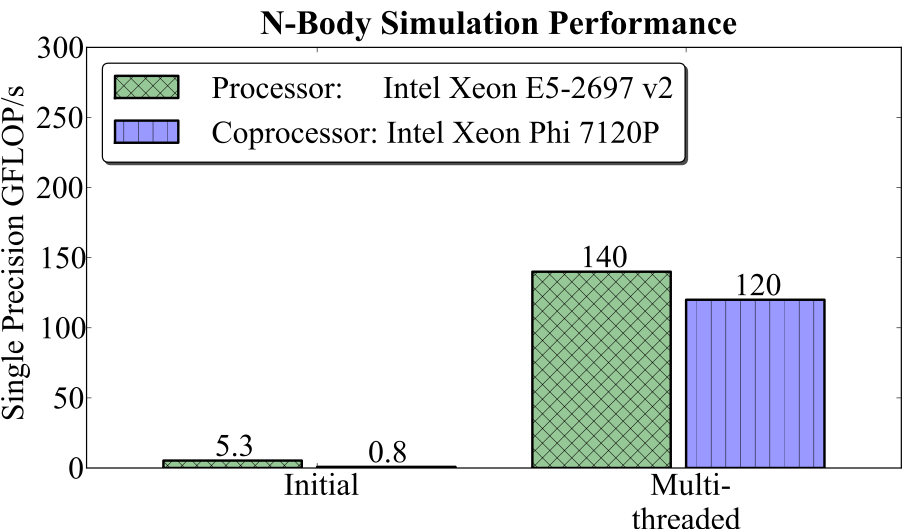
Before:

```
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



Vectorizing with Unit-Stride Memory Access

Before:

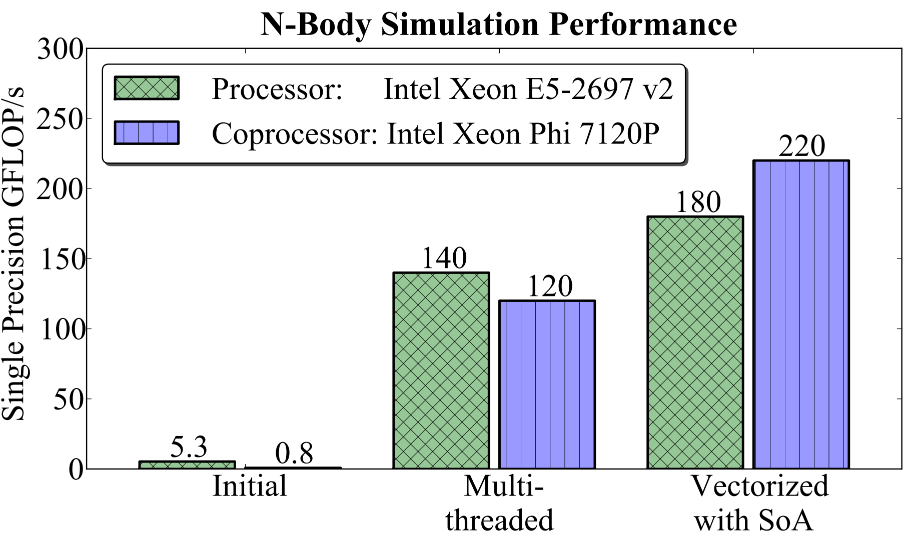
```
struct ParticleType {
  float x, y, z, vx, vy, vz;
}; // ...
      const float dx = particle[j].x - particle[i].x;
      const float dy = particle[j].y - particle[i].y;
      const float dz = particle[j].z - particle[i].z;
```

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];
```

 © Colfax International, 2013–2015

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;
```

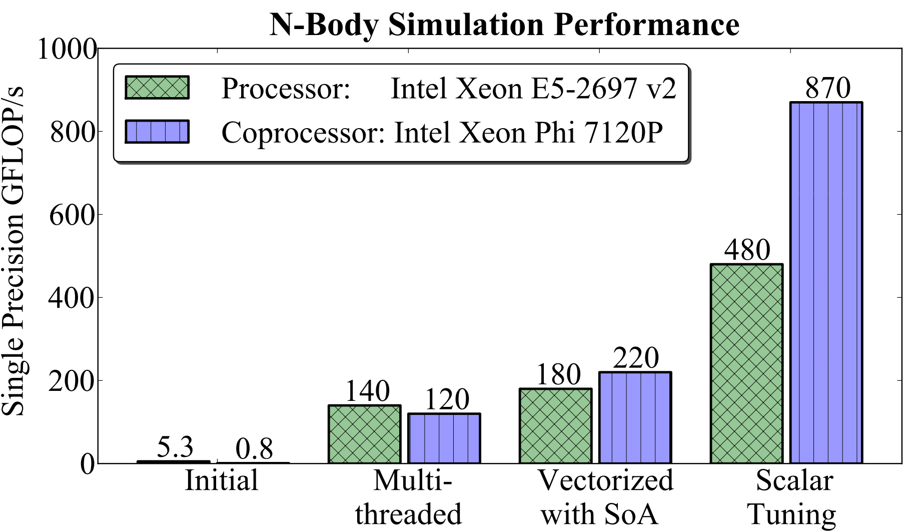
After:

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

© Colfax International, 2013–2015

Performance after Scalar Tuning



© Colfax International, 2013–2015

Improving Cache Traffic

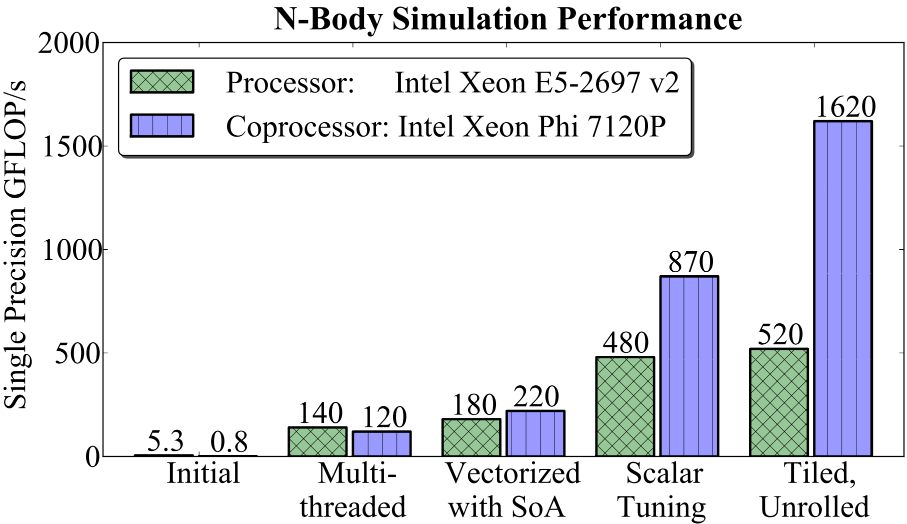
Before:

```
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;
```

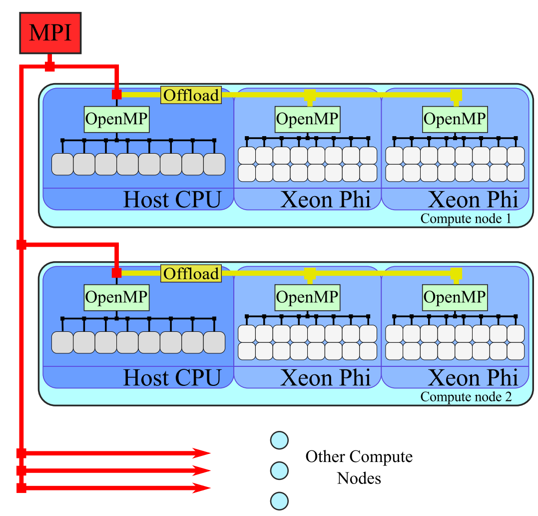
Performance with Cache Optimization (Loop Tiling)



 © Colfax International, 2013–2015

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`



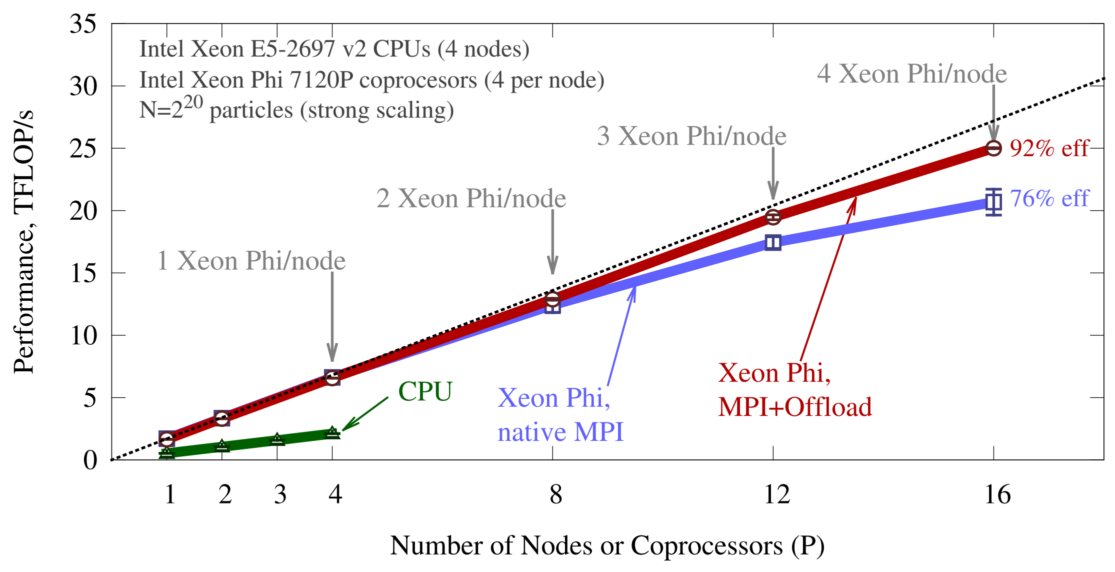
 © Colfax International, 2013–2015

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         // ...
```

 © Colfax International, 2013–2015

Results with MPI+Offload



© Colfax International, 2013–2015

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: xeonphi.com/papers/nbody-basic

Demo: http://www.youtube.com/watch?v=KxaSEcmkGTo

§2. Introduction to the Intel Many Integrated Core (MIC) Architecture

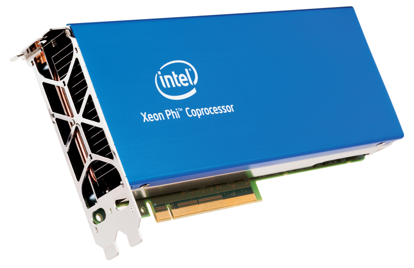
© Colfax International, 2013–2015

Purpose of the Intel MIC Architecture

© Colfax International, 2013–2015

Intel Xeon Phi Coprocessors and the MIC Architecture

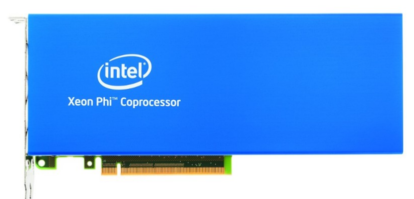
- PCIe end-point device
- High Power efficiency
- $\sim$ 1 TFLOP/s in DP
- $\sim$ 2 TFLOP/s in SP
- Up to 16 GiB onboard RAM ($\sim$ 170 GB/s bandwidth)



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

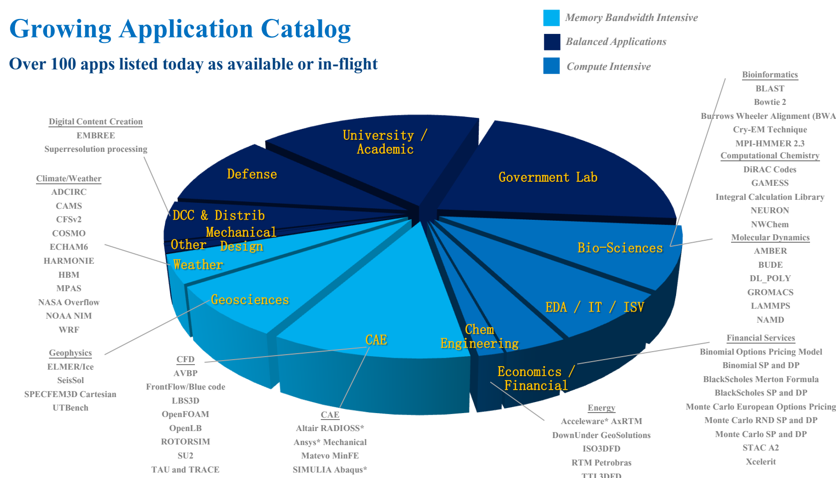
Intel Xeon Phi Coprocessors and the MIC Architecture

- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GB of DDR3 RAM
- ≤18 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

Intel Xeon Phi Coprocessor Application Catalog

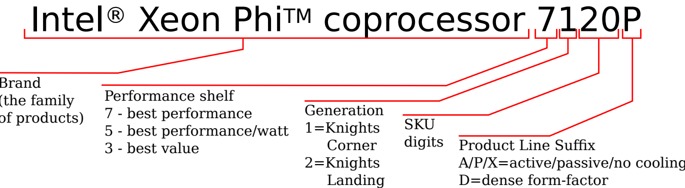


https://software.intel.com/en-us/xeonphionlinecatalog

SKUs and Solutions

© Colfax International, 2013–2015

Intel Xeon Phi Coprocessor SKU Lineup



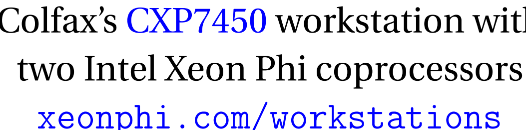
3100 series : price-optimal group (300 W TDP, 57 cores, 6 GB RAM, 240 GB/s, 1000 GFLOP/s, RCP $1695–1960).

5100 series : best performance per watt (225 W TDP, 60 cores, 8 GB RAM, 320 GB/s, 1000 GFLOP/s, RCP $2437–2649).

7100 series : top performance (300 W TDP, 61 cores, 16 GB RAM, 352 GB/s, 1200 GFLOP/s, RCP $4129–4235).

Performance reported above is theoretical peak values. In practice, ≈50% of RAM bandwidth and ≈70% of arithmetic throughput is achievable. RCP is price guidance for bulk purchases by direct Intel customers, not a formal pricing offer.

Examples of Solutions with the Intel MIC Architecture



Colfax's CXP7450 workstation with two Intel Xeon Phi coprocessors

xeonphi.com/workstations

Colfax's CXP9000 server with eight Intel Xeon Phi coprocessors

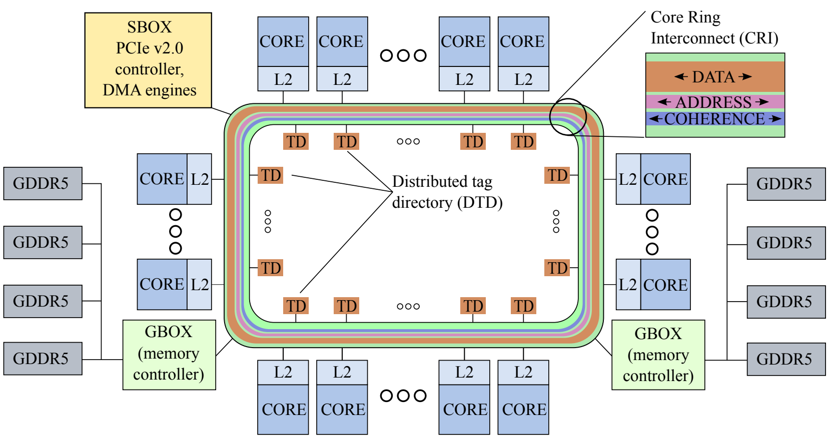
xeonphi.com/servers

© Colfax International, 2013–2015

Details of the MIC Architecture

 © Colfax International, 2013–2015

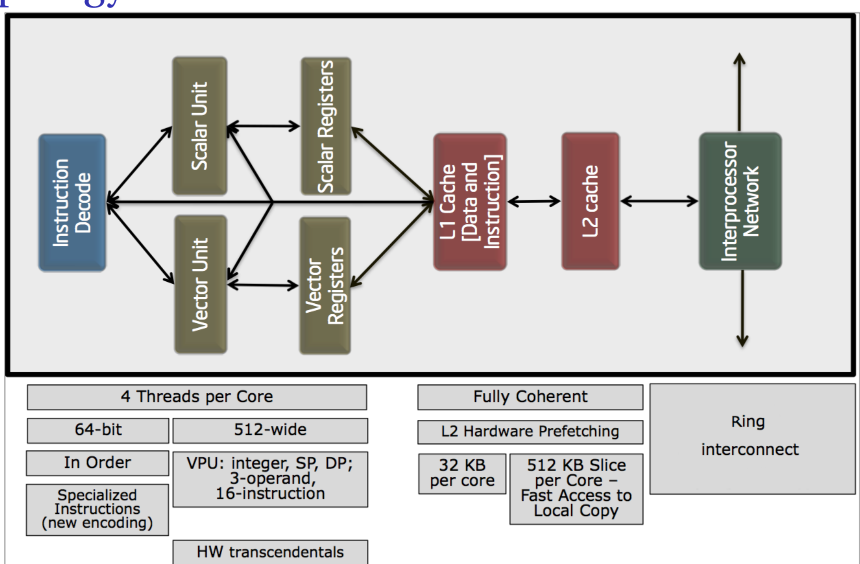
Die Organization Diagram



L1: 32 KiB/core, 8-way associative, ≈1 cycle latency, software prefetching only.
L2: 512 KiB/core, 8-way associative, ≈10 cycle latency, hardware+software prefethcing.

© Colfax International, 2013–2015

Core Topology



SIMD Operations

SIMD — Single Instruction Multiple Data

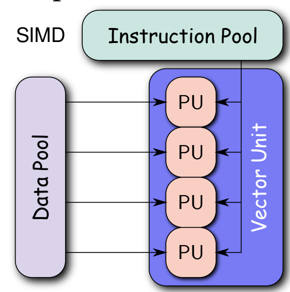
Scalar Loop

```
1  for (i = 0; i < n; i++)
2    A[i] = A[i] + B[i];
```

SIMD Loop

```
1  for (i = 0; i < n; i += 16)
2    A[i:(i+16)] = A[i:(i+16)] + B[i:(i+16)];
```

Each SIMD addition operator acts on 16 numbers at a time.



Features of the IMCI Instruction Set

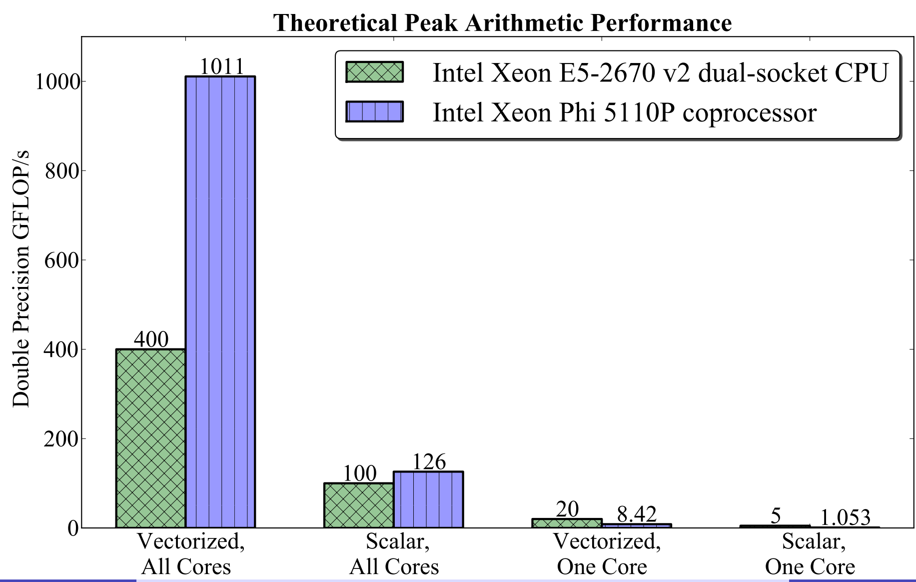
Knight's Corner uses Initial Many Core Instruction (IMCI) set.
512-bit wide registers: can hold 8 DP or 16 SP values.

IMCI Supports:

- Initialization, Load and Store, Gather and Scatter
- Arithmetic Instructions: Binary Operators, Transcendental Functions, etc.
- Comparison
- Conversion and type cast
- Bitwise instructions: `NOT, AND, OR, XOR, XAND`
- Reduction and minimum/maximum instructions
- Vector mask instructions

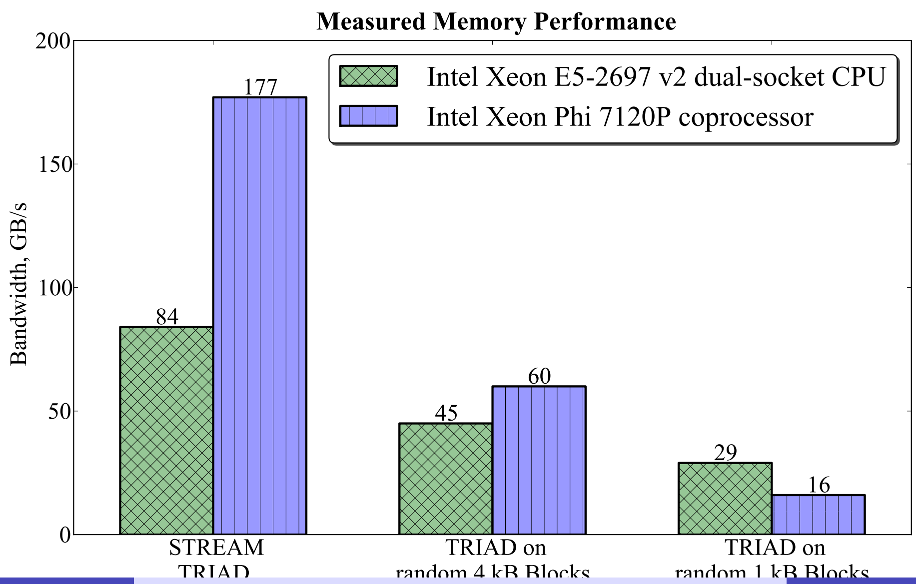
 © Colfax International, 2013–2015

Task and Data Parallelism

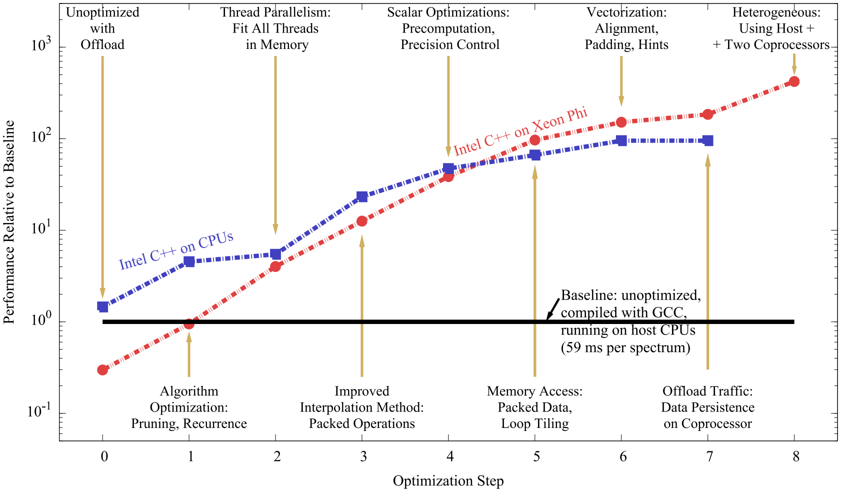


© Colfax International, 2013–2015

Memory Access Pattern



Performance on MIC is a Function of Optimization Level



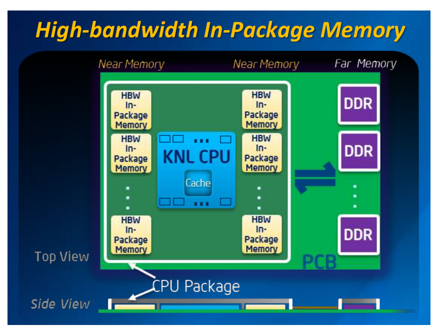
Paper: xeonphi.com/papers/heatcode

 © Colfax International, 2013–2015

Knights Landing, the Next Manycore Architecture

Next Generation MIC: Knights Landing (KNL)

- 2nd generation MIC product: code name Knights Landing (KNL)
- Intel's 14 nm manufacturing process
- A processor (running the OS) or a coprocessor (PCIe device)
- On-package high-bandwidth memory w/flexible memory models: flat, cache, & hybrid
- Intel Advanced Vector Extensions AVX-512 (public)



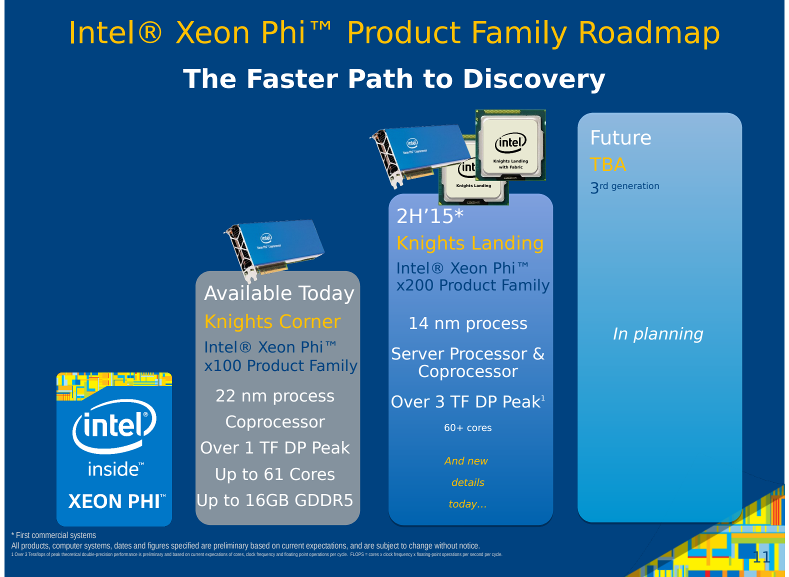
Source: *Intel Newsroom*

© Colfax International, 2013–2015

Getting Ready for the Future

- Porting HPC applications to today's MIC architecture makes them ready for future architectures, such as KNL
- Xeon, KNC and KNL are not binary compatible, therefore assembly-level tuning will not scale forward.
- Reliance on compiler optimization and using optimized libraries (such as Intel MKL) ensures future-readiness.

Source: Intel Newsroom



Source: https://www.brighttalk.com/webcast/10773/116329

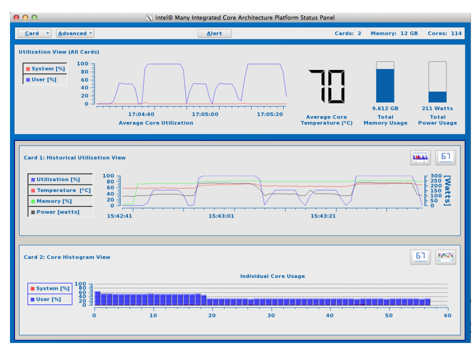
§3. System Administration for Intel Xeon Phi Coprocessors

 © Colfax International, 2013–2015

Software Tools for Intel Xeon Phi Coprocessors

Intel Manycore Platform Software Stack

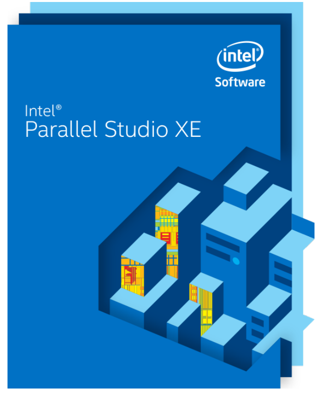
- `micinfo` – system information
- `micsmc` – monitor and modify the physical paramaters: temperature, power modes, core utilization, etc.
- `micctrl` – configure the Intel Xeon Phi coprocessor operating system
- `miccheck` – verify the Intel Xeon Phi coprocessor configuration
- `micrasd` – log of hardware errors reported by Intel Xeon Phi coprocessors
- `micflash` – flash memory agent



Monitoring MIC activity with `micsmc` (an MPSS tool)

Software Necessary to Build Xeon Phi Applications:

- **Compilers** : Intel C Compiler, Intel C++ Compiler, and Intel Fortran Compiler — mandatory
- **Optimization tools** : Intel VTune Amplifier XE and Intel Trace Analyzer and Collector (ITAC) — highly recommended
- **Mathematics support** : Intel Math Kernel Library (MKL) — highly recommended
- **Cluster Development** : Intel MPI — industry standard parallel framework
- **Development** : Intel Inspector XE, Intel Advisor XE — optional



All-in-One bundle,
Intel Parallel Studio XE

 © Colfax International, 2013–2015

Components of Intel Parallel Studio XE

Software	Composer	Professional	Cluster
Intel C, C++ and Fortran compilers	x	x	x
Intel Math Kernel Library (MKL)	x	x	x
Intel Threading Building Blocks (TBB)	x	x	x
Intel Performance Primitives (IPP)	x	x	x
Intel VTune Amplifier XE		x	x
Intel Inspector XE		x	x
Intel Advisor XE		x	x
Intel MPI Library			x
Intel Trace Analyzer and Collector (ITAC)			x

© Colfax International, 2013–2015

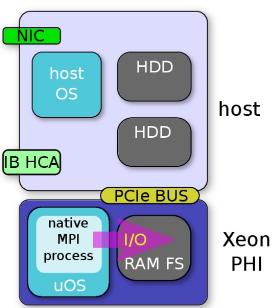
Operating System and Filesystem on Coprocessors

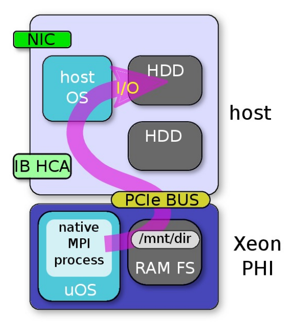
Linux μOS on Intel Xeon Phi coprocessors (part of MPSS)

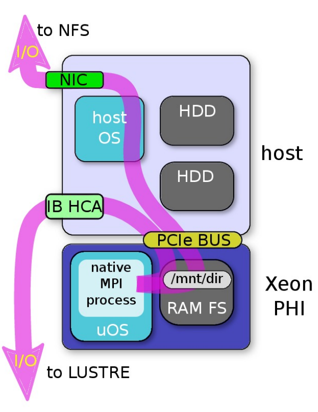
```
vega@lyra% lspci | grep -i "co-processor"
06:00.0 Co-processor: Intel Corporation Xeon Phi coprocessor 7120 series (rev 20)
82:00.0 Co-processor: Intel Corporation Xeon Phi coprocessor 7120 series (rev 20)
vega@lyra% sudo service mpss status
mpss is running
vega@lyra% cat /etc/hosts | grep mic
172.31.1.1  lyra-mic0 mic0
172.31.2.1  lyra-mic1 mic1
vega@lyra% ssh mic0

vega@mic0% cat /proc/cpuinfo | grep proc | tail -n 3
processor: 241
processor: 242
processor: 243
vega@mic0% ls /
amplxe   dev    home   lib64     oldroot   proc   sbin      sys   usr
bin      etc    lib    linuxrc   opt       root   sep3.10   tmp   var
```


© Colfax International, 2013–2015

Working with Files on Coprocessors







RAM Filesystem

VirtIO Transfer

Network Storage

Details: xeonphi.com/papers/io

© Colfax International, 2013–2015

§4. Models for Intel Xeon Phi Coprocessor Programming

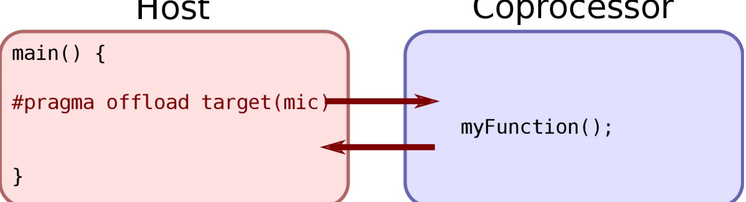
 © Colfax International, 2013–2015

Overview of Programming Options

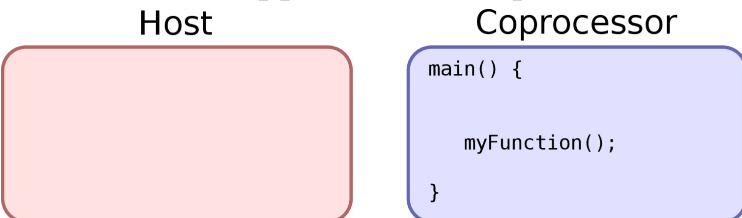
 © Colfax International, 2013–2015

Offload and Native modes

- Offload mode (explicit/virtual-shared memory/OpenMP 4.0):



- Native mode (standalone application/MPI process):



Teaming Xeon Processors with Xeon Phi Coprocessors

Programming models allow a range of CPU+MIC coupling modes

Xeon - Multi-Core Centric | *Breadth* | MIC - Many-Core Centric

Multi-Core Hosted	Offload	Symmetric	Many Core Hosted
General serial and parallel computing	Code with highly-parallel phases	Codes with balanced needs	Highly-parallel codes

Native Coprocessor Applications

 © Colfax International, 2013–2015

Native Execution

"Hello World" application:

```
1 #include <cstdio>
2 #include <unistd.h>
3 int main(){
4     printf("Hello world! I have %ld logical cores.\n",
5     sysconf(_SC_NPROCESSORS_ONLN ));
6 }
```

Compile and run on host:

```
vega@lyra% icpc hello.cc
vega@lyra% ./a.out
Hello world! I have 48 logical cores.
vega@lyra%
```

 © Colfax International, 2013–2015

Native Execution

Compile and run the same code on the coprocessor in the native mode:

```
vega@lyra% icpc hello.cc -mmic
vega@lyra% scp a.out mic0:~/
a.out 100% 10KB 10.4KB/s 00:00
vega@lyra% ssh mic0
vega@mic0% pwd
/home/lyra
vega@mic0% ls
a.out
vega@mic0% ./a.out
Hello world! I have 244 logical cores.
vega@mic0%
```

- Use `-mmic` to produce executable for MIC architecture
- Must transfer executable to coprocessor (or NFS-share) and run from shell
- Native MPI applications work the same way (need Intel MPI library)

© Colfax International, 2013–2015

Alternative Native Application Launcher: `micnativeloadex`

The tool `micnativeloadex` automatically transfers code and dependent libraries and runs the application.

```
vega@lyra% export SINK_LD_LIBRARY_PATH=/opt/intel/composerxe/compiler/lib/mic
vega@lyra% icpc hello.cc -mmic
vega@lyra% micnativeloadex a.out
Hello world! I have 244 logical cores.
vega@lyra%
```

- Set `SINK_LD_LIBRARY_PATH` to help the tool find libraries
- Do not have to SSH into the coprocessor
- Runs under `micuser` on coprocessor

Native Applications with Autotools

- Use the Intel compiler with flag `-mmic`
- Eliminate assembly and unncecessary dependencies
- Use `--host=x86_64` to avoid “program does not run” errors

Example, the GNU Multiple Precision Arithmetic Library (GMP):

```
vega@lyra% wget https://ftp.gnu.org/gnu/gmp/gmp-5.1.3.tar.bz2
vega@lyra% tar -xf gmp-5.1.3.tar.bz2
vega@lyra% cd gmp-5.1.3
vega@lyra% ./configure CC=icc CFLAGS="-mmic" --host=x86_64 --disable-assembly
...
vega@lyra% make
...
```

 © Colfax International, 2013–2015

Explicit Offload

 © Colfax International, 2013–2015

Explicit Offload: Pragma-based approach

"Hello World" in the explicit offload model:

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    printf("Hello World from host!\n");
#pragma offload target(mic)
    {
        printf("Hello World from coprocessor!\n"); fflush(0);
    }
    printf("Bye\n");
}
```

Application runs on the host, but some parts of code and date are moved ("offloaded") the coprocessor.

Detailed syntax in the Intel C++ Compiler Reference.

© Colfax International, 2013–2015

Compiling and Running an Offload Application

```
vega@lyra% icpc hello_offload.cpp -o hello_offload
vega@lyra% ./hello_offload
Hello World from host!
Bye
Hello World from coprocessor!
```

- No additional arguments if compiled with an Intel compiler
- Run application on host as a regular application
- Code inside of `#pragma offload` is offloaded automatically
- Console output on Intel Xeon Phi coprocessor is buffered and mirrored to the host console
- If coprocessor is not installed, code inside `#pragma offload` runs on the host system

Optional Offload, Fall-Back to Host

```
1  #pragma offload target(mic) optional
2  {
3    printf("Hello World! I have %d logical cores.\n",
4    sysconf(_SC_NPROCESSORS_ONLN )); fflush(0);
5  }
```

```
vega@lyra% icpc Offload-Fallback.cc -o Offload-Fallback
vega@lyra% ./Offload-Fallback
Hello World! I have 244 logical cores.
vega@lyra% sudo systemctl stop mpss # Disabling coprocessors
vega@lyra% ./Offload-Fallback
Hello World! I have 48 logical cores.
```

 © Colfax International, 2013–2015

Offloading Functions

```
__attribute__((target(mic))) void MyFunction() {
    // ... implement function as usual
}

int main(int argc, char * argv[] ) {
#pragma offload target(mic)
    {
        MyFunction();
    }
}
```

- Functions used on coprocessor must be marked with the specifier `__attribute__((target(mic)))`
- Compiler produces a host version and a coprocesor version of such functions (to enable fall-back to host)

© Colfax International, 2013–2015

Offloading Multiple Functions

```
1 #pragma offload_attribute(push, target(mic))
2 void MyFunctionOne() {
3 // ... implement function as usual
4 }
5
6 void MyFunctionTwo() {
7 // ... implement function as usual
8 }
9 #pragma offload_attribute(pop)
```

- To mark a long block of code with the offload attribute, use `#pragma offload_attribute(push/pop)`

© Colfax International, 2013–2015

Offloading Data: Local Scalars and Arrays

```
1  void MyFunction() {
2      const int N = 1000;
3      int data[N];
4  #pragma offload target(mic)
5      {
6          for (int i = 0; i < N; i++)
7            data[i] = 0;
8      }
```

- Scope-local scalars and known-size arrays offloaded automatically
- Data is copied from host to coprocessor at the start of offload
- Data is copied back from coprocessor to host at the end of offload
- Bitwise-copyable data only (arrays of basic types and scalars)
  C++ classes, etc. should use virtual-shared memory model

Offloading Data: Global and Static Variables

```
int* __attribute__((target(mic))) data;

void MyFunction() {
    static int __attribute__((target(mic))) N;
    // ...
}

int main() {
    // ...
}
```

- Global and static variables must be marked with the offload attribute
- `#pragma offload_attribute(push/pop)` may be used as well

Data Marshalling for Dynamically Allocated Data

```
1  double *p1=(double*)malloc(sizeof(double)*N);
2  double *p2=(double*)malloc(sizeof(double)*N);
3
4  #pragma offload target(mic) in(p1 : length(N)) out(p2 : length(N))
5  {
6      // ... perform operations on p1[] and p2[]
7  }
```

- `#pragma offload` recognizes clauses `in`, `out`, `inout` and `nocopy`
- Data size (length), alignment, redirection, and other properties may be specified
- Marshalling is required for pointer-based data

 © Colfax International, 2013–2015

Multiple Coprocessors with Explicit Offload

 © Colfax International, 2013–2015

Multiple Coprocessors with Explicit Offload

- Querying the number of coprocessors:

```
const int numDevices = _Offload_number_of_devices();
printf("Number of available coprocessors: %d\n" , numDevices);
```

- Specifying offload target:

```
#pragma offload target(mic: 0)
{ /* ... */ }
```

- Query the device number from within Offload:

```
#pragma offload target(mic)
{
  const int deviceNum = _Offload_get_device_number();
  printf("Hello from coprocessor %d!\n" , deviceNum);
}
```

 © Colfax International, 2013–2015

Multiple Blocking Offloads Using Host Threads (Explicit Offload)

```
1  const int nDevices = _Offload_number_of_devices();
2  #pragma omp parallel num_threads(nDevices)
3    {
4      const int i = omp_get_thread_num();
5  #pragma offload target(mic: i)
6        {
7          MyFunction(/*...*/);
8        }
9    }
10 }
```

- Up to 8 coprocessors, up to 56 host threads
- All offloads start simultaneously and block the respective thread

Memory Allocation Control

© Colfax International, 2013–2015

Memory retention and data persistence on coprocessor

- By default, memory on coprocessor is allocated before, deallocated after offload
- Specifiers `alloc_if` and `free_if` allow to avoid allocation/deallocation
- Data transfer across the PCIe bus rate is 6-7 GB/s
- To allocate memory on the coprocessor – 0.5-2.0 GB/s

```
1  #pragma offload target(mic:0) in(p : length(N) alloc_if(1) free_if(0) )
2  { /* allocate memory for array p on coprocessor, do not deallocate */ }
3
4  #pragma offload target(mic:0) in(p : length(N) alloc_if(0) free_if(0) )
5  { /* re-use previously allocated memory buffer on coprocessor */ }
6
7  #pragma offload target(mic:0) in(p : length(0) alloc_if(0) free_if(0) )
8  { /* re-use previously transferred data on coprocessor */ }
9
10 #pragma offload target(mic:0) out(p : length(N) alloc_if(0) free_if(1) )
11 { /* re-use memory and deallocate at the end of offload */ }
```

Precautions with persistent data

- Use explicit zero-based coprocessor number (e.g., `mic:0` as shown below)
- With multiple coprocessors, if target number is unspecified, any coprocessor can be used, which will result in runtime errors if persistent data cannot be found.

```
1 #pragma offload target(mic:0) in(p : length(N) alloc_if(1) free_if(0) )
2 { /* allocate memory for array p on coprocessor, do not deallocate */ }
```

- Do not change the value of the host pointer to a persistent array: the runtime system finds the data on coprocessor using the host pointer value, not variable name.

 © Colfax International, 2013–2015

Overlapping Communication and Computation

 © Colfax International, 2013–2015

Asynchronous Offload

- By default, `#pragma offload` blocks until offload completes
- Use clause "`signal`" with any pointer to avoid blocking
- Use `#pragma offload_wait` to block where needed

```
char* offload0;
#pragma offload target(mic:0) signal(offload0) in(data : length(N))
{ /* ... will not block code execution because of clause "signal" */ }

DoSomethingElse();

/* Now block until offload signalled by pointer "offload0" completes */
#pragma offload_wait target(mic:0) wait(offload0)
```

- Use the target number to avoid hanging

Virtual-Shared Memory Offload Model

© Colfax International, 2013–2015

Virtual-shared Memory Model

```
_Cilk_shared int arr[N]; // This is a virtual-shared array

_Cilk_shared void Compute() { // This function may be offloaded
    // ... function uses array arr[]
}

int main() {
    // arr[] can be initialized on the host
    _Cilk_offload Compute(); // and used on coprocessor
    // and the values are returned to the host
}
```

- Alternative to Explicit Offload
- Data synced from host to coprocessor before the start of offload
- Data synced from coprocessor to host at the end of offload

 © Colfax International, 2013–2015

Virtual-shared Memory Model

```
int* _Cilk_shared data; // Pointer to a virtual-shared array

int main() {
    // Working with pointer-based data is illustrated below:
    data = (_Cilk_shared int*)_Offload_shared_malloc(N*sizeof(float));
    _Offload_shared_free(data);
}
```

- Addresses of virtual-shared pointers identical on host and coprocessors
- Synchronized before and after offload, with page granularity

 © Colfax International, 2013–2015

Additional Offload Controls

© Colfax International, 2013–2015

Offload diagnostics

```
vega@lyra% export OFFLOAD_REPORT=2
vega@lyra% ./offload-application
Transferring some data to and from coprocessor...
Done. Bye!
[Offload] [MIC 0] [File]            offload-application.cpp
[Offload] [MIC 0] [Line]            6
[Offload] [MIC 0] [CPU Time]        0.505982 (seconds)
[Offload] [MIC 0] [CPU->MIC Data]   1024 (bytes)
[Offload] [MIC 0] [MIC Time]        0.000409 (seconds)
[Offload] [MIC 0] [MIC->CPU Data]   1024 (bytes)
vega@lyra%
```

- Set environment variable `OFFLOAD_REPORT` to 1 or 2 for automatic collection and output of offload information.
- Unset or set `OFFLOAD_REPORT=0` to disable offload diagnostics

Target-Specific Code

- During MIC architecture compilation, preprocessor macro `__MIC__` is defined.
- Allows to fine-tune application performance or output where necessary

```
1  __attribute__((target(mic))) void MyFunction() {
2  #ifdef __MIC__
3      printf("I am running on a coprocessor.\n");
4      const int tuningParameter = 16;
5  #else
6      printf("I am running on the host.\n");
7      const int tuningParameter = 8;
8  #endif
9      // ... Proceed, using the variable tuningParameter
10 }
```

Offload Devices, Specifying Available Coprocessors

- Specify coprocessors to use; For example (using 0 and 1),

```
vega@lyra% export OFFLOAD_DEVICES=0,1
```

- Disable Offloading

```
vega@lyra% export OFFLOAD_DEVICES=none
```

Disabling Offload is useful for debugging. For example;

```
vega@lyra% icpc Offload-Fallback.cc -o Offload-Fallback
vega@lyra% ./Offload-Fallback
Hello from offload on MIC with 244 logical cores.
vega@lyra% export OFFLOAD_DEVICES=none # Coprocessors disabled
vega@lyra% ./Offload-Fallback
Hello from offload on CPU with 48 logical cores.
```

 © Colfax International, 2013–2015

Environment variable forwarding with offload

- By default, all host environment variables on the host will be copied to the coprocessor when offload starts.
- In order to have different values for an environment variable on host and coprocessor, set `MIC_ENV_PREFIX`
- The prefix is dropped when variables are copied to coprocessor

```
vega@lyra% # This sets the value of OMP_NUM_THREADS on the host:
vega@lyra% export OMP_NUM_THREADS=32
vega@lyra%
vega@lyra% # This enables special rules for variable copying:
vega@lyra% export MIC_ENV_PREFIX=XEONPHI
vega@lyra%
vega@lyra% # This sets the value of OMP_NUM_THREADS on the coprocessor:
vega@lyra% export XEONPHI_OMP_NUM_THREADS=236
```

 © Colfax International, 2013–2015

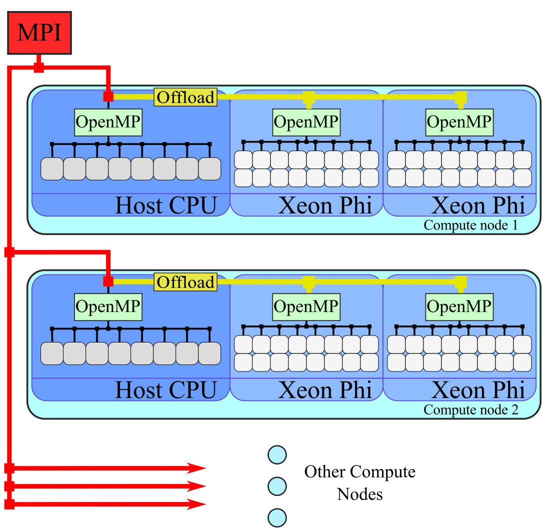
Coprocessors in Clusters

 © Colfax International, 2013–2015

Heterogeneous Distributed Computing with Xeon Phi

Option 1: MPI+OpenMP with Offload.

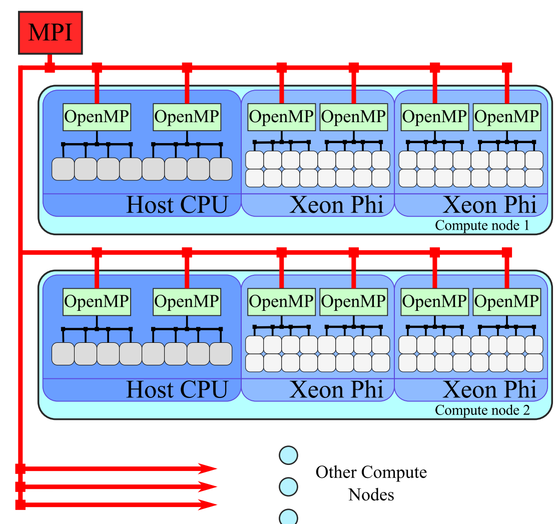
- MPI processes are multi-threaded with OpenMP.
- MPI runs only on CPUs.
- MPI processes offload to coprocessor(s).
- OpenMP in offload regions.



Heterogeneous Distributed Computing with Xeon Phi

Option 2: Symmetric hybrid MPI+OpenMP.

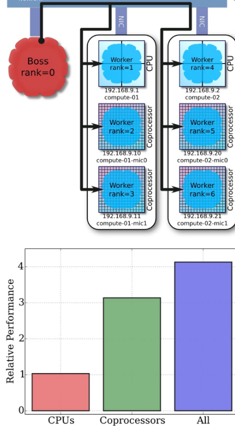
- MPI processes on hosts
- Native MPI processes on the coprocessor.
- Multi-threading with OpenMP.



 © Colfax International, 2013–2015

Heterogeneous Clustering with Homogeneous Code: Asian Option Pricing

- Monte Carlo method
- MPI + OpenMP + automatic vectorization
- The same C code for clusters of
  a) CPUs
  b) Coprocessors
  c) CPUs+Coprocessors (heterogeneous)
- **No code modification** to run on the Intel MIC architecture
- No platform-specific tunning
- Bridged network configuration

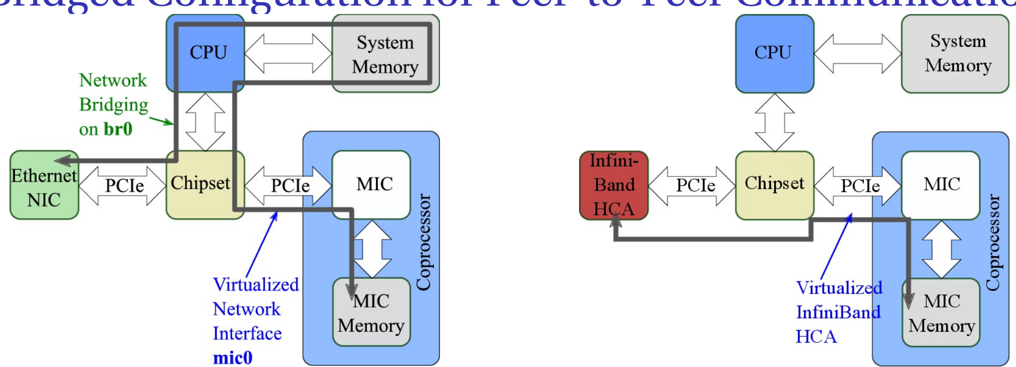


Paper: xeonphi.com/papers/heterogeneous
Demo: http://youtu.be/GffmChTcWf8

 © Colfax International, 2013–2015

© Colfax International, 2013–2015

Bridged Configuration for Peer-to-Peer Communication



- Left: Gigabit Ethernet bridging on host allows to place coprocessors on the same subnet as hosts
- Right: Coprocessor Communication Link (CCL) – virtualization of an InfiniBand device on each coprocessor

 © Colfax International, 2013–2015

§5. Expressing Parallelism on Intel Architectures

© Colfax International, 2013–2015

SIMD Parallelism and Automatic Vectorization

SIMD Operations

SIMD — Single Instruction Multiple Data

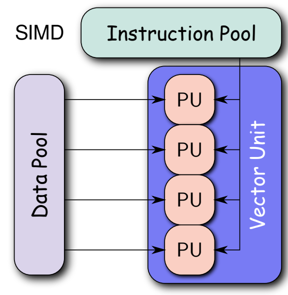
Scalar Loop

```
1 for (i = 0; i < n; i++)
2   A[i] = A[i] + B[i];
```

SIMD Loop

```
1 for (i = 0; i < n; i += 16)
2   A[i:(i+16)] = A[i:(i+16)] + B[i:(i+16)];
```

Each SIMD addition operator acts on 16 numbers at a time.



Explicit Vectorization: Compiler Intrinsics

Scalar code

```
for (int i=0; i<n; i++) {
  // This loop must be transformed
  // to a vector loop
  // by programmer or by compiler
  A[i] = A[i] + B[i];
}
```

IMCI Intrinsics

```
for (int i=0; i<n; i+=16) {
    __m512 Avec=_mm512_load_ps(A+i);
    __m512 Bvec=_mm512_load_ps(B+i);
    Avec=_mm512_add_ps(Avec, Bvec);
    _mm512_store_ps(A+i, Avec);
}
```

- The arrays `float A[n]` and `float B[n]` are aligned on 64-byte boundary
- n is a multiple of 16
- Variables `Avec` and `Bvec` are $512 = 16 \times$ `sizeof(float)` bits for the Intel Xeon Phi architecture

Automatic Vectorization of Loops

```
#include <stdio.h>

int main(){
  const int n=8;
  int i;
  int A[n] __attribute__((aligned(64)));
  int B[n] __attribute__((aligned(64)));

  // Initialization
  for (i=0; i<n; i++)
    A[i]=B[i]=i;

  // This loop will be auto-vectorized
  for (i=0; i<n; i++)
    A[i]+=B[i];

  // Output
  for (i=0; i<n; i++)
    printf("%2d %2d %2d\n", i, A[i], B[i]);
}
```

```
vega@lyra% icpc autovec.cc \
> -qopt-report -qopt-report-phase:vec
vega@lyra% cat autovec.optrpt
...
LOOP BEGIN at autovec.cc(14,3)
remark #15399: vectorization support:
unroll factor set to 2 [autovec.cc(14,3)]
remark #15300: LOOP WAS VECTORIZED
[autovec.cc(14,3)]
LOOP END
...
vega@lyra% ./a.out
0  0  0
1  2  1
2  4  2
3  6  3
4  8  4
5 10  5
6 12  6
7 14  7
```

Automatic Vectorization of Loops on MIC architecture

Compilation and runtime output of the code for Intel Xeon Phi execution

```
vega@lyra% icpc autovec.cc -mmic -qopt-report -qopt-report-phase:vec
vega@lyra% cat autovec.optrpt
...
LOOP BEGIN at autovec.cc(14,3)
remark #15300: LOOP WAS VECTORIZED  [ autovec.cc(14,3) ]
LOOP END
...
vega@lyra% micnativeloadex a.out
0  0  0
1  2  1
2  4  2
3  6  3
4  8  4
5 10  5
6 12  6
7 14  7
```

 © Colfax International, 2013–2015

Automatic Vectorization of Loops

Limitations:

- Only `for`-loops can be auto-vectorized. Number of iterations must be known at a runtime and/or compilation time
- Memory access in the loop must have a regular pattern, ideally with unit stride
- Non-standard loops that cannot be automatically vectorized:
  - loops with irregular memory access pattern
  - calculations with vector dependence
  - `while`-loops, for-loops with undetermined number of iterations
  - outer loops (unless `#pragma simd` overrides this restriction)
  - loops with complex branches (i.e., `if`-conditions)
  - anything else that cannot be, or is very difficult to vectorize.

 © Colfax International, 2013–2015

Array Notation for Automatic Vectorization

Extensions for Array Notation

Array notation is a method for specifying

- slices of arrays (begin, length)

```
1 A[0:16] += B[32:16]; // B[32]...B[47] added to A[0]...A[15]
```

- a stride (begin, length, stride)

```
1 A[0:16:2] += B[32:16:4]; // B[32],B[36]...B[92] added A[0],A[2]...A[30]
```

- whole arrays (without specifying boundaries)

```
1 A[:][:] += B[:][:]; // Add B to A; arrays are of the same shape
```

Better than loops with strides (e.g., `xeonphi.com/papers/efft`).

© Colfax International, 2013–2015

Assumed Vector Dependence

© Colfax International, 2013–2015

Assumed Vector Dependence

- True vector dependence makes vectorization impossible:

```
float *a, *b;
for (int i = 1; i < n; i++)
  a[i] += b[i]*a[i-1]; // dependence on the previous element
```

- *Assumed vector dependence*: when compiler cannot determine wheter vector dependence exists, auto-vectorization fails:

```
void mycopy(int n,
            float* a, float* b) {
  for (int i = 0; i < n; i++)
    a[i] = b[i];
}
```

```
vega@lyra% icpc -c vdep.cc -qopt-report \
> -qopt-report-phase:vec
vega@lyra% cat vdep.optrpt
...
remark #15304: loop was not
vectorized: non-vectorizable loop
instance from multiversioning
...
```

 © Colfax International, 2013–2015

Ignoring Assumed Vector Dependence

To ignore assumed vector dependence

```
#pragma ivdep
```

```
1 void mycopy(int n,
2             float* a, float* b) {
3 #pragma ivdep
4   for (int i = 0; i < n; i++)
5     a[i] = b[i];
6 }
```

```
vega@lyra% icpc -c vdep.cc -qopt-report \
> -qopt-report-phase:vec
vega@lyra% cat vdep.optrpt
...
LOOP BEGIN at vdep.cc(4,1)
<Multiversioned v2>
remark #15300: LOOP WAS VECTORIZED
LOOP END
```


© Colfax International, 2013–2015

Pointer Disambiguation (alternative to #pragma ivdep)

- `restrict` keyword applies to each pointer variable qualified with it
- `restrict` declares that the object accessed by the pointer is **only accessed by that pointer** in the given scope
- Must compile with the argument `-restrict`

```
1 void mycopy(int n, float* restrict a, float* restrict b) {
2   for (int i = 0; i < n; i++)
3     a[i] = b[i];
4 }
```

```
vega@lyra% icpc -qopt-report -qopt-report-phase:vec -restrict -c vdep.cc
vega@lyra% cat vdep.optreport
LOOP BEGIN at vdep.cc(2,1)
remark #15300: LOOP WAS VECTORIZED
LOOP END
```

 © Colfax International, 2013–2015

SIMD-Enabled Functions

© Colfax International, 2013–2015

SIMD-Enabled Functions

(formerly "elemental functions")

What if the implementation of a function is in a separate source code file (e.g., a library function)?

```
float my_simple_add(float x1, float x2){
  return x1 + x2;
}
```

```
// ...in a separate source file:
for (int i = 0; i < N, ++i) {
  output[i] = my_simple_add(inputa[i], inputb[i]);
}
```

Compiler will refuse to automatically vectorize this loop.

© Colfax International, 2013–2015

SIMD-Enabled Functions

The solution is to design and declare the function as *SIMD-enabled*:

```
__attribute__((vector)) float my_simple_add(float x1, float x2) {
  return x1 + x2;
}
```

When using SIMD-enabled functions, use `#pragma simd`.

```
// ...in a separate source file:
#pragma simd
for (int i = 0; i < N, ++i) {
  output[i] = my_simple_add(inputa[i], inputb[i]);
}
```

In this case, automatic vectorization succeeds.

© Colfax International, 2013–2015

SIMD-Enabled Functions

- Written as a regular C/C++ functions
- Operating on scalar numbers with scalar syntax
- Can be used in data- and thread-parallel contexts with automatic parallelization across multiple threads
- Every SIMD-enabled function must be *pure*, i.e., without side-effects: must not modify global data that other instances of that function depend on
- Paper: `xeonphi.com/papers/simd-lib`

© Colfax International, 2013–2015

Thread Parallelism and OpenMP

 © Colfax International, 2013–2015

“Hello World” OpenMP Programs

```c
#include <omp.h>
#include <stdio.h>

int main(){
  const int nt=omp_get_max_threads();
  printf("OpenMP with %d threads\n", nt);

#pragma omp parallel
  {
    printf("Hello World from thread %d\n", omp_get_thread_num());
  }
}
```


© Colfax International, 2013–2015

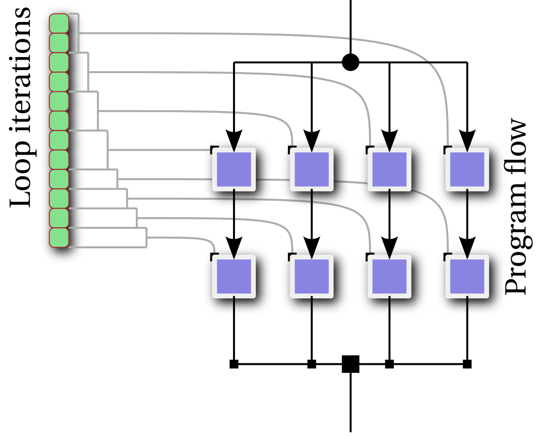
“Hello World” OpenMP Programs

```
vega@lyra% export OMP_NUM_THREADS=5
vega@lyra% icpc -qopenmp hello_omp.cc
vega@lyra% ./a.out
OpenMP with 5 threads
Hello World from thread 0
Hello World from thread 3
Hello World from thread 1
Hello World from thread 2
Hello World from thread 4
```

`OMP_NUM_TREADS` controls number of OpenMP threads. (default: logical core count)

Loop-Centric Parallelism: For-Loops in OpenMP

- Simultaneously launch multiple threads
- Scheduler assigns loop iterations to threads
- Each thread processes one iteration at a time



Parallelizing a `for`-loop.

 © Colfax International, 2013–2015

Loop-Centric Parallelism: For-Loops in OpenMP

The OpenMP library will distribute the iterations of the loop following the `#pragma omp parallel for` across threads.

```
int A, B;

#pragma omp parallel for private(A) shared(B)
for (int i = 0; i < n; i++) {
  printf("Iteration %d is processed by thread %d\n",
         i, omp_get_thread_num());
  // ... iterations will be distributed across available threads...
  // Each thread has a private copy of variable A
  // All threads access the same memory location for variable B
}
```

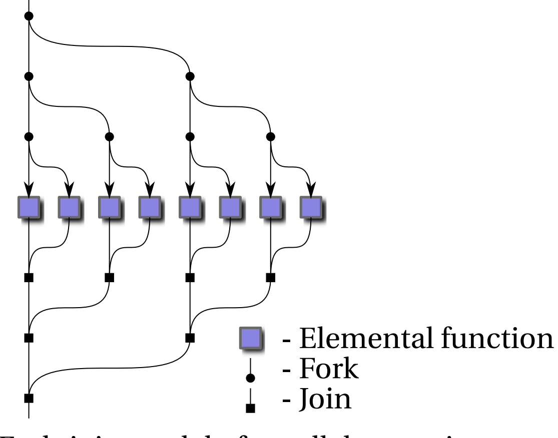

© Colfax International, 2013–2015

Loop-Centric Parallelism: For-Loops in OpenMP

```
#pragma omp parallel
{
  // Code placed here will be executed by all threads.

  // Alternative way to specify private variables:
  // declare them in the scope of pragma omp parallel
  int private_number=0;

#pragma omp for schedule(dynamic, 4)
  for (int i=0; i<n; i++) {
    // ... iterations will be distributed across available threads...
  }
  // ... code placed here will be executed by all threads
}
```

Fork-Join Model of Parallel Execution

- Each thread can spawn daughter threads
- Available threads pick up queued tasks
- Expresses algorithms that cannot be expressed in the loop model (e.g., parallel recursion)



Fork-join model of parallel execution.

(`#pragma omp task` functionality)

Tasks in OpenMP: Example

```
// Starting the first task:
#pragma omp parallel
{ // Enter a parallel region
#pragma omp single
  { // Start the first task
    // from only one thread
    RecursiveWorkload(args);
  }
}
```

```
// Recursive task spawning:
void RecursiveWorkload(Arg* args) {
  if (args->size > threshold) {
    // Split work
    Arg* args1=args->FirstHalf();
    Arg* args2=args->SecondHalf();

    // Parallel divide-and-conquer
#pragma omp task firstprivate(args1)
    { RecursiveWorkload(args1); }
#pragma omp task firstprivate(args2)
    { RecursiveWorkload(args2);  }
  } else {
    // End of recursion
    args->ProcessSmallestSubTask();
  }
}
```

Thread Synchronization in OpenMP

© Colfax International, 2013–2015

Race Conditions and Unpredictable Program Behavior

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main() {
4    const int n = 1000;
5    int total = 0;
6  #pragma omp parallel for
7    for (int i = 0; i < n; i++) {
8      // Race condition
9      total = total + i;
10   }
11   printf("total=%d (must be %d)\n", total, ((n-1)*n)/2);
12 }
```

Thread 1
Shared Variable
Thread 2
Zero
+1
One
+1
+1
Two

Race Condition!

```
vega@lyra% icpc -o omp-race omp-race.cc -qopenmp
vega@lyra% ./omp-race
total=208112 (must be 499500)
```

Protecting Race Conditions with a Critical Section

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main() {
4    const int n = 1000;
5    int total = 0;
6  #pragma omp parallel for
7    for (int i = 0; i < n; i++) {
8  #pragma omp critical
9      { // Only one thread at a time can execute this section
10       total = total + i;
11     }
12 } }
```

Thread 1
Shared Variable
Thread 2
Zero
+1
One
wait
+1
Two
+1
Three

```
vega@lyra% icpc -o omp-critical omp-critical.cc -qopenmp
vega@lyra% ./omp-critical
total=499500 (must be 499500)
```

Avoiding Races with Atomic Operations

This parallel fragment of code has predictable behavior, because the race condition was eliminated with *an atomic operation*:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3 { // Lightweight synchronization
4 #pragma omp atomic
5   sum += i;
6 }
```

Limitations of Atomic Operations

Read : operations in the form `v = x`

Write : operations in the form `x = v`

Update : operations in the form `x++`, `x--`, `--x`, `++x`, `x` *binop*= *expr* and `x = x` *binop expr*

Capture : operations in the form `v = x++`, `v = x-`, `v = -x`, `v = ++x`, `v = x` *binop expr*

- Here `x` and `v` are scalar variables
- *binop* is one of `+`, `*`, `-`, `- /`, `&`, `^`, `|`, `«`, `»`.
- No “trickery” is allowed for atomic operations:
  - no operator overload,
  - no non-scalar types,
  - no complex expressions.

© Colfax International, 2013–2015

 © Colfax International, 2013–2015

Reduction Clause in Parallel Region

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main() {
5    const int n = 1000;
6    int sum = 0;
7  #pragma omp parallel for reduction(+: sum)
8    for (int i = 0; i < n; i++) {
9      sum = sum + i;
10   }
11   printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
12 }
```

```
vega@lyra% icpc -o omp-reduction omp-reduction.cc -qopenmp
vega@lyra% ./omp-reduction
sum=499500 (must be 499500)
```

 © Colfax International, 2013–2015

Reduction using Private Variables

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main() {
5    const int n = 1000;
6    int sum = 0;
7  #pragma omp parallel
8    {
9      int sum_th = 0;
10 #pragma omp for
11     for (int i = 0; i < n; i++)
12       { sum_th = sum_th + i; }
13 #pragma omp atomic
14     sum += sum_th;
15   }
16   printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
17 }
```

 © Colfax International, 2013–2015

Distributed Memory Parallelism and MPI

© Colfax International, 2013–2015

Structure of MPI Applications: Hello World

```
1  #include "mpi.h"
2  #include <stdio.h>
3  int main (int argc, char *argv[]) {
4    MPI_Init (&argc, &argv);  // Initialize MPI environment
5    if (ret != MPI_SUCCESS) {
6      MyErrorLogger("...");
7      MPI_Abort(MPI_COMM_WORLD, ret);
8    }
9    int i, rank, size, namelen;
10   char name[MPI_MAX_PROCESSOR_NAME];
11   MPI_Comm_size (MPI_COMM_WORLD, &size);
12   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
13   MPI_Get_processor_name (name, &namelen);
14   printf ("Hello World from rank %d running on %s!\n", rank, name);
15   if (rank == 0) printf("MPI World size = %d processes\n", size);
16   MPI_Finalize (); // Terminate MPI environment
17 }
```

Compiling and Running MPI Applications on Host

```
vega@lyra% mpiicpc -o HelloMPI HelloMPI.cc
vega@lyra% mpirun -host localhost -np 2 ./HelloMPI
Hello World from rank 1 running on lyra!
Hello World from rank 0 running on lyra!
MPI World size = 2 processes
```

- Set up MPI environment variables
- Use wrapper script `mpiicpc` to compile
- Use automated tool `mpirun` to launch

Compiling and Running Native MPI Applications on Coprocessors

```
vega@lyra% export I_MPI_MIC=1
vega@lyra% mpiicpc -mmic -o HelloMPI.MIC HelloMPI.c
vega@lyra% scp HelloMPI.MIC mic0:~/
vega@lyra% mpirun -host mic0 -np 2 ~/HelloMPI.MIC
Hello World from rank 1 running on lyra-mic0!
Hello World from rank 0 running on lyra-mic0!
MPI World size = 2 processes
```

- Enable the MIC architecture in Intel MPI: `I_MPI_MIC=1`
- Copy or NFS-share MPI library & executables with coprocessor
- Use `mpiicpc` with `-mmic` to compile
- Launch as if `mic0` is a remote host

Heterogeneous MPI Applications: Host + Coprocessors

```
vega@lyra% mpirun \
> -host mic0 -n 2 ~/Hello.MIC : \
> -host mic1 -n 2 ~/Hello.MIC : \
> -host localhost -n 2 ~/Hello
Hello World from rank 5 running on localhost!
Hello World from rank 4 running on localhost!
Hello World from rank 2 running on mic1!
Hello World from rank 3 running on mic1!
Hello World from rank 1 running on mic0!
Hello World from rank 0 running on mic0!
MPI World size = 6 ranks
```

- Specify Xeon executable for host processes
- Specify Xeon Phi executable for coprocessor processes

Heterogeneous MPI Applications: Machine File

```
vega@lyra% cat hosts.txt
localhost:2
mic0:2
mic1:2
vega@lyra% export I_MPI_MIC_POSTFIX=.MIC
vega@lyra% mpirun -machinefile hosts.txt ~/Hello
Hello World from rank 0 running on localhost!
Hello World from rank 1 running on localhost!
Hello World from rank 2 running on mic1!
Hello World from rank 3 running on mic1!
Hello World from rank 4 running on mic0!
Hello World from rank 5 running on mic0!
MPI World size = 6 ranks
```

- Specify Xeon executable for host processes
- MIC executable obtained by appending `I_MPI_MIC_POSTFIX`

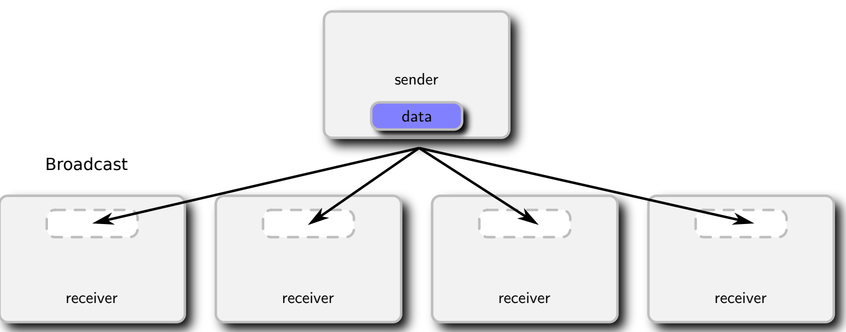
Point to Point Communication

```
1  if (rank == sender) {
2
3   char outgoingMsg[messageLength];
4   strcpy(outgoingMsg, "/Jenny");
5   MPI_Send(&outgoingMsg, messageLength, MPI_CHAR, receiver, tag, MPI_COMM_WORLD);
6
7
8  } else if (rank == receiver) {
9
10  char incomingMsg[messageLength];
11  MPI_Recv (&incomingMsg, messageLength, MPI_CHAR, sender,
12  tag, MPI_COMM_WORLD, &stat);
13  printf ("Received message with tag %d: '%s'\n", tag, incomingMsg);
14
15
16 }
```


© Colfax International, 2013–2015

Collective Communication: Broadcast

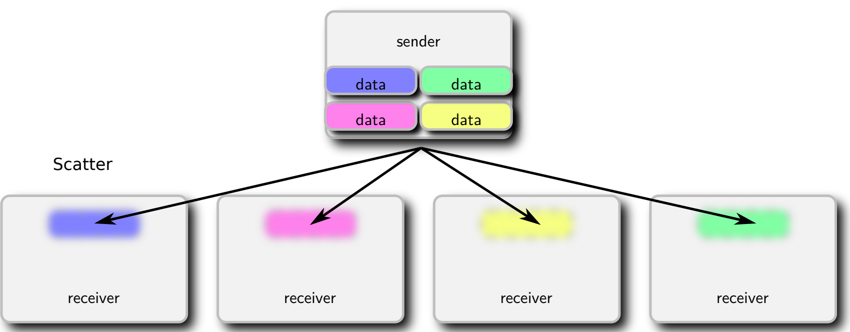
```
1 int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype,
2     int root, MPI_Comm comm );
```



© Colfax International, 2013–2015

Collective Communication: Scatter

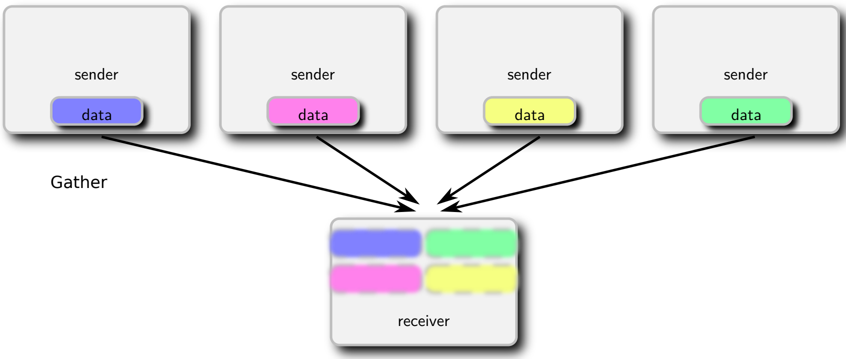
```
1 int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
2     int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
```



© Colfax International, 2013–2015

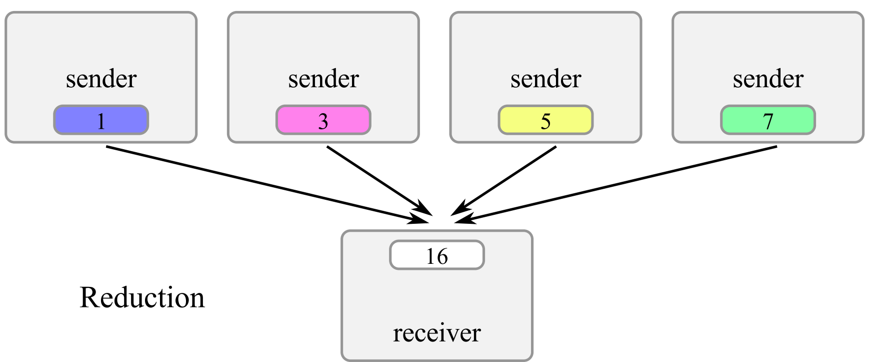
Collective Communication: Gather

```
1 int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
2     void *recvbuf, int recvcnt, MPI_Datatype recvtype,
3     int root, MPI_Comm comm);
```



Collective Communication: Reduction

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
    MPI_Op op, int root, MPI_Comm comm);
```



Available reducers: max/min, minloc/maxloc, sum, product, AND, OR, XOR (logical or bitwise).

 © Colfax International, 2013–2015

Summary and Additional Resources

 © Colfax International, 2013–2015

Expressing Parallelism

1. Data parallelism (vectorization)
   - Automatic vectorization by the compiler: portable and convenient
   - For-loops and array notation can be vectorized
   - Compiler hints (`#pragma simd`, `#pragma ivdep`, etc.) to assist the compiler
2. Shared-memory parallelism with OpenMP
   - Parallel threads access common memory for reading and writing
   - Parallel loops: `#pragma omp parallel for` — automatic work distribution
   - In OpenMP: private and shared variables; synchronization, reduction.
3. Distributed-memory parallelism with MPI
   - MPI processes do not share memory, but can send information to each other
   - All MPI processes execute the same code; role is determined by its rank
   - Point-to-point and collective communication patterns

Educational Materials on Parallel Programming

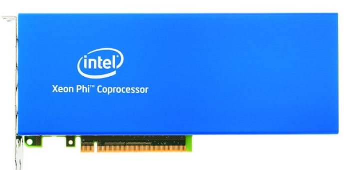
1. OpenMP Specifications
2. Intel's OpenMP Video Course
3. MPI Routines on the ANL Web Site
4. LLNL tuotorial: OpenMP, MPI
5. Book: "Intel Xeon Phi Coprocessor High Performance Programming" by Jeffers & Reinders
6. Book: "Parallel Programming and Optimization with Intel Xeon Phi Coprocessors" by Colfax.

© Colfax International, 2013–2015

§6. Optimization Using Intel Software Development Tools

 © Colfax International, 2013–2015

Coprocessor vs Processor Performance



vs.

One Intel Xeon Phi 7120P coprocessor

Two Intel Xeon E5-2697 v2 CPUs

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

See also "Intel Xeon Product Family: Performance Brief"

Library Solution: Intel Math Kernel Library (MKL)

 © Colfax International, 2013–2015

Intel Math Kernel Library (MKL)

Intel MKL offers variety of math functions that are optimized for Xeon Processors as well as for Xeon Phi coprocessors.

Linear Algebra	Fast Fourier Transform	Vector Math	Vector Random Number Generators	Summary Statistics	Data Fitting
BLAS LAPACK Sparse solvers ScaLAPACK	Multidimentional (up to 7D) FFTW interfaces Cluster FFT	Trigonometric Hyperbolic Exponential Logarithmic Power/Root Rounding	Congruential Recursive Wichmann-Hill Mersenne Twister Sobol Neiderreiter Non-deterministic	Kurtosis Variation coefficitent Quantiles, order statistics Min/max Variance-covariance	Splines Interpolation Cell search

© Colfax International, 2013–2015

Using Intel MKL

Three modes of usage:

- Native Execution

```
1 cblas_dgemm(..., ...);
```

```
vega@lyra% icpc -mkl -mmic Code.cc
vega@lyra% scp a.out mic0:~/
vega@lyra% ssh mic0 ./a.out
```

- Automatic Offload

```
1 cblas_dgemm(..., ...);
```

```
vega@lyra% icpc -mkl Code.cc
vega@lyra% MKL_MIC_ENABLE=1
vega@lyra% ./a.out
```

- Compiler-Assisted Offload

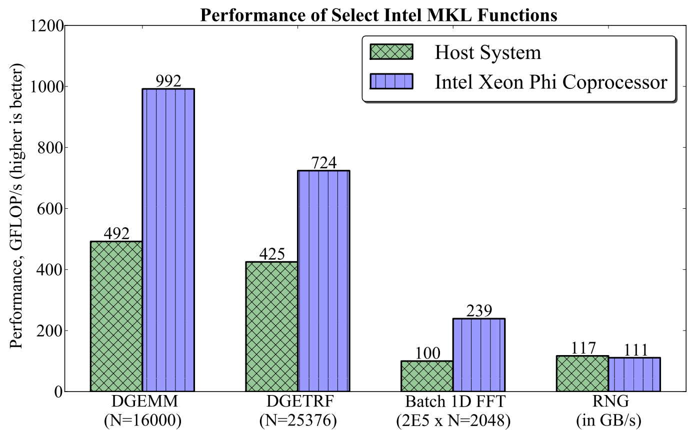
```
1 #pragma offload target(mic) ...
2 cblas_dgemm(..., ...);
```

```
vega@lyra% icpc -mkl Code.cc
vega@lyra% ./a.out
```


© Colfax International, 2013–2015

Performance of MKL in Native Mode

Performance of various MKL functions using Native execution.

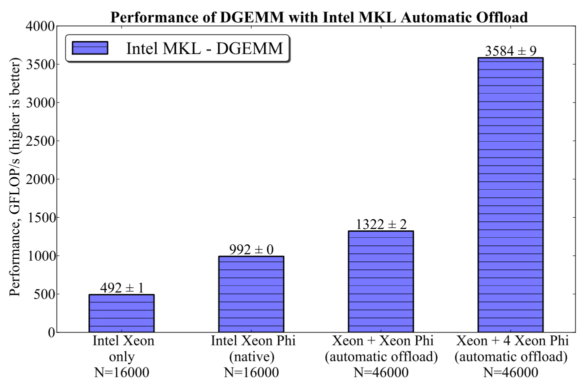


 © Colfax International, 2013–2015

Performance of MKL Automatic Offload Mode

```
1 cblas_dgemm(..., ...);
```

```
vega@lyra% icpc -mkl Code.cc
vega@lyra% OMP_NUM_THREADS=24
vega@lyra% KMP_AFFINITY=compact,1
vega@lyra% MKL_MIC_ENABLE=1
vega@lyra% MIC_KMP_AFFINIT=compact
vega@lyra% ./a.out
```



Automatic Offload vs. Compiler Assisted Offload

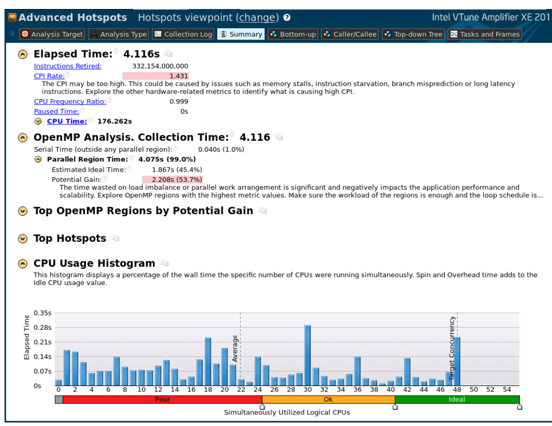
	Automatic Offload	Compiler Assisted Offload
Code Modification	No	Required
Data Marshalling	Automatic (no control)	Manual
Using Multiple Coprocessors	Automatic	Must be implemented by user
Functions Supported	Some	All

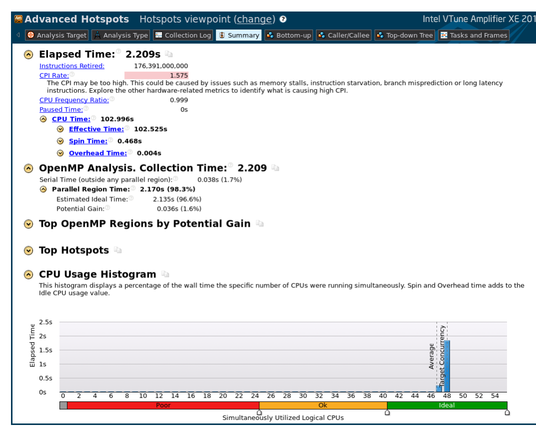
 © Colfax International, 2013–2015

Node-level Tuning with Intel VTune Amplifier XE

© Colfax International, 2013–2015

VTune: Hardware Events, Concurrency Evaluation



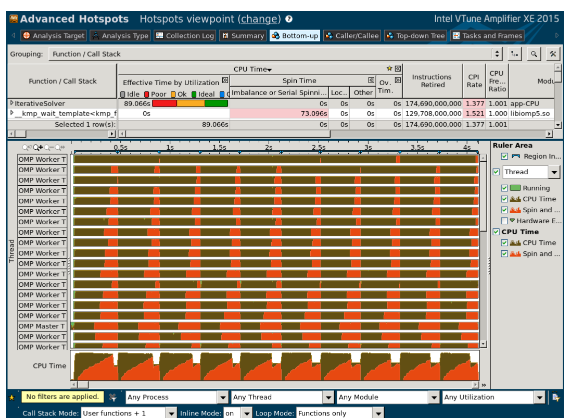


Not optimized: poor concurrency

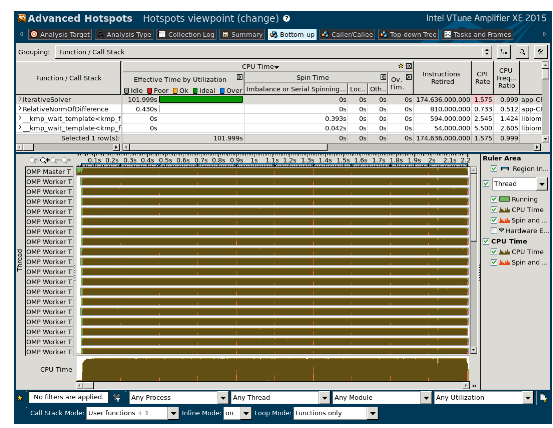
Optimized: ideal concurrency

 © Colfax International, 2013–2015

VTune: Timeline and Hotspots

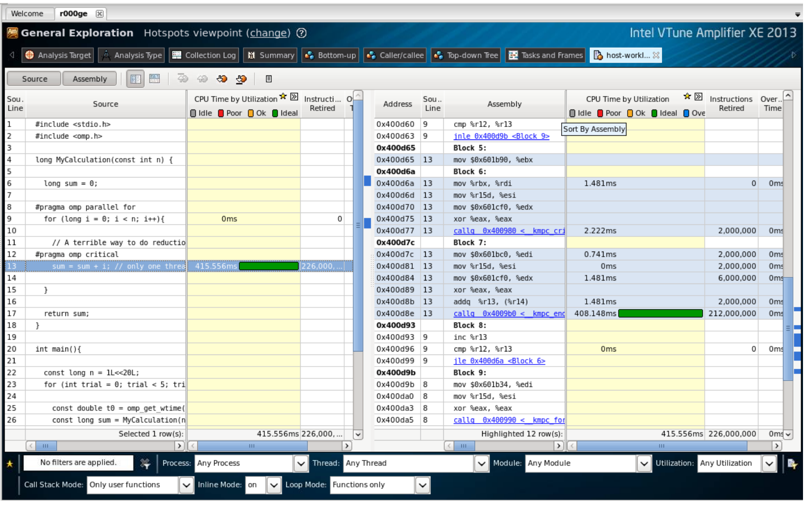


Not optimized: OpenMP functions in hotspots, idling in timeline



Optimized: user functions in hotspots, mostly CPU time in timeline

VTune: Drilling into Hotspots Down to a Single Line of Code:



§7. Optimization Roadmap

 © Colfax International, 2013–2015

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)

© Colfax International, 2013–2015

§8. Optimization of Scalar Arithmetics

© Colfax International, 2013–2015

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)

 © Colfax International, 2013–2015

Compiler-friendly Practices

© Colfax International, 2013–2015

Array Reference by Index instead of Pointer Arithmetics

```
1 for (int i = 0; i < N; i++)
2   for (int j = 0; j < N; j++) {
3     float* cp = c + i*N + j;
4     for (int k = 0; k < N; k++)
5       *cp += a[i*N+k]*b[k*N+j];
6 }
```

```
1 for (int i = 0; i < N; i++)
2   for (int j = 0; j < N; j++) {
3
4     for (int k = 0; k < N; k++)
5       c[i*N+j] += a[i*N+k]*b[k*N+j];
6 }
```

```
vega@lyra% icpc array_ptr.cc
vega@lyra% tmr ./a.out
Time: 0.733 s
```

```
vega@lyra% icpc array_index.cc
vega@lyra% tmr ./a.out
Time: 0.124 s
```

- With *Pointer arithmetics*, the code is 6x slower than with reference to array elements *by index*.

© Colfax International, 2013–2015

Strength Reduction

Common Subexpression Elimination.

```
for (int i = 0; i < n; i++)  {
  A[i] /= B;
}
```

```
const float Br = 1.0f/B;
for (int i = 0; i < n; i++)
  A[i] *= Br;
```

Replace division with multiplication.

```
for (int i = 0; i < n; i++)  {
  P[i] = (Q[i]/R[i])/S[i];
}
```

```
for (int i = 0; i < n; i++) {
  P[i] = Q[i]/(R[i]*S[i]);
}
```

Use functions with Hardware support.

```
double r = pow(r2, -0.5);
double v = exp(x);
double y = y0*exp(log(x/x0)*
              log(y1/y0)/log(x1/x0));
```

```
double r = 1.0/sqrt(r2);
double v = exp2(x*1.44269504089);
double y = y0*exp2(log2(x/x0)*
              log2(y1/y0)/log2(x1/x0));
```

 © Colfax International, 2013–2015

Precision Control

 © Colfax International, 2013–2015

Precision Control for Transcendental Functions

`-fimf-precision=value[:funclist]` Defines the precision for math functions. `value` is one of: `high`, `medium` or `low`

`-fimf-max-error=ulps[:funclist]` The maximum allowable error expressed in ulps (*units in last place*)

`-fimf-accuracy-bits=n[:funclist]` The number of correct bits required for mathematical function accuracy.

`-fimf-domain-exclusion=n[:funclist]` Defines a list of special-value numbers that do not need to be handled.
`int n` derived by the bitwise OR of types:
extremes: 1, NaNs: 2, infinites: 4, denormals[1]: 8, zeroes: 16.

[1] by default, on Intel Xeon Phi, denormals are flushed to zero in hardware, but supported in SVML

Precision Control for Transcendental Functions

```
1  const int N = 1000000;
2  const int P = 10;
3  double A[N];
4  const double startValue = 1.0;
5  A[:] = startValue;
6  for (int i = 0; i < P; i++)
7    for (int r = 0; r < N; r++)
8      A[r] = exp(-A[r]);
9  printf("Result=%.17e\n", A[0]);
```

```
vega@lyra% icpc -o precision-2\
> -fimf-precision=high\
> precision.cc
vega@lyra% tmr ./precision-2
Result=5.68428725029060722e-01
Time: 0.073 s
```

```
vega@lyra% icpc -o precision-1\
> -fimf-precision=low\
> precision.cc
vega@lyra% tmr ./precision-1
Result=5.68428725010313829e-01
Time: 0.046 s
```

Floating-Point Semantics

The Intel C++ Compiler may represent floating-point expressions in executable code differently, depending on the *floating-point semantics*.

Option	Description
`-fp-model strict`	Only value-safe optimizations
`-fp-model precise`	calculations are reproducible from run to run
	exceptions controlled using `-fp-model except`
`-fp-model fast=1`	(default) Value-unsafe optimizations are allowed
`-fp-model fast=2`	better performance at the cost of lower accuracy
`-fp-model source`	Intermediate arithmetic results are rounded to the precision defined in the source code.
`-fp-model double`	Intermediate arithmetic results are rounded to 53-bit (double) precision.
`-fp-model extended`	Intermediate arithmetic results are rounded to 64-bit (extended) precision.
`-fp-model [no-]except`	controls floating-point exception semantics.

Consistency of Precision: Constants

```
// Bad: 2 is "int"
long b=a*2;

// Bad: overflow
long n=100000*100000;

// Bad: excessive
float p=6.283185307179586;

// Bad: 2 is "int"
float q=2*p;

// Bad: 1e9 is "double"
float r=1e9*p;

// Bad: 1 is "int"
double t=s+1;
```

```
// Good: 2L is "long"
long b=a*2L;

// Good: correct
long n=100000L*100000L;

// Good: accurate
float p=6.283185f;

// Good: 2.0f is "float"
float q=2.0f*p;

// Good: 1e9f is "float"
float r=1e9f*p;

// Good: 1.0 is "double"
double t=s+1.0;
```

Consistency of Precision: Functions

```
1  // Bad: 3.14 is a double
2  float x = 3.14;
3
4  // Bad: sin() is a
5  // double precision function
6  float s = sin(x)
7
8  // Bad: round() takes double
9  // and returns double
10 long v = round(x);
11
12 // Bad: abs() is not from IML
13 // it takes int and returns int
14 int v = abs(x);
```

```
1  // Good: 3.14f is a float
2  float x = 3.14f;
3
4  // Good: sin() is a
5  // single precision function
6  float s = sinf(x)
7
8  // Good: lroundf() takes float
9  // and returns long
10 long v = lroundf(x);
11
12 // Good: fabsf() is from IML
13 // It takes and returns a float
14 float v = fabsf(x);
```

 © Colfax International, 2013–2015

Consistency of Precision: Functions

Transcendental functions are *not* overloaded (unless in namespace `std` in C++).

```
vega@lyra% ./Scalar-TestFOverload
Proof that exp() is not overloaded:
exp (1.0f)=2.718281828459045
exp (1.0 )=2.718281828459045
Exact:    e=2.7182818284590452353 6...

Proof that expf() gives lower precision:
expf(1.0f)=2.718281745910645
expf(1.0 )=2.718281745910645
Exact:    e=2.71828182845904523536...

Overloading in namespace std:
std::exp(1.0f)=2.718281745910645
std::exp(1.0 )=2.718281828459045
Exact:        e=2.71828182845904523536...
```


© Colfax International, 2013–2015

§9. Optimization of Vectorization

 © Colfax International, 2013–2015

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)

 © Colfax International, 2013–2015

Challenges with Optimizing Vectorization on Xeon Phi

- Must utilize 512-bit vector registers (16 `float` or 8 `double`)
- Must convince compiler that vectorization is possible
- Preferably unit-stride access to data
- Preferably align data on 64-byte boundary
- Avoid branches in vector loops
- Guide compiler regarding expected iteration count, memory alignment, outer loop vectorization, etc.

Diagnosing the Utilization of Vector Instructions

- Use `-qopt-report -qopt-report-phase:vec` to get information about automatic vectorization
- Benchmark regular compilation *vs.* `-no-vec -no-simd` case
- Intel VTune Amplifier XE can collect statistics on vector instruction usage

 © Colfax International, 2013–2015

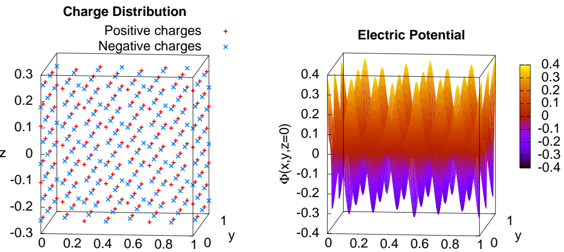
Vector-friendly Data Structures

 © Colfax International, 2013–2015

Example: Unit-Stride Access in Coulomb's Law Application

$$\Phi\left(\vec{R}_j\right) = -\sum_{i=1}^{m} \frac{q_i}{\left|\vec{r}_i - \vec{R}_j\right|}, \quad (1)$$

$$\left|\vec{r}_i - \vec{R}\right| = \sqrt{\left(r_{i,x} - R_x\right)^2 + \left(r_{i,y} - R_y\right)^2 + \left(r_{i,z} - R_z\right)^2}. \quad (2)$$



Paper: xeonphi.com/papers/autovec

© Colfax International, 2013–2015

Elegant, but Inefficient Solution: Array of Structures

```
1  struct Charge {                    // Elegant, but ineffective data layout
2    float x, y, z, q;
3  } chgs[m]; // Coordinates and value of this charge
```

```
1  for (int i=0; i<m; i++)  { // This loop will be auto-vectorized
2    // Non-unit stride: (&chg[i+1].x - &chg[i].x) != sizeof(float)
3    const float dx=chg[i].x - Rx;
4    const float dy=chg[i].y - Ry;
5    const float dz=chg[i].z - Rz;
6    phi -= chg[i].q / sqrtf(dx*dx+dy*dy+dz*dz); // Coulomb's law
7  }
```


© Colfax International, 2013–2015

Arrays of Structures versus Structures of Arrays

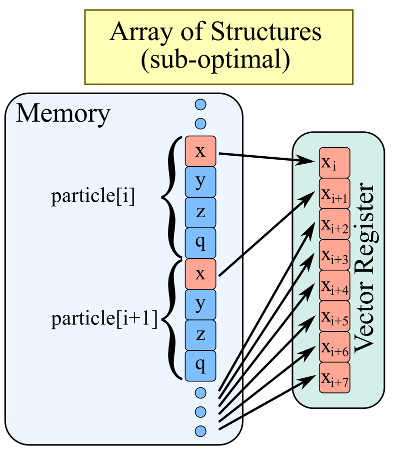
Array of Structures (AoS)

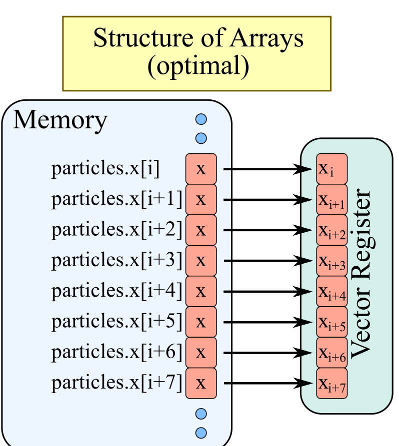
```
struct Charge { // Elegant, but ineffective data layout
  float x, y, z, q; // Coordinates and value of this charge
};
// The following line declares a set of m point charges:
Charge chg[m];
```

Structure of Arrays (SoA)

```
struct Charge_Distribution {
  // Data layout permits effective vectorization of Coulomb's law application
  const int m; // Number of charges
  float * x; // Array of x-coordinates of charges
  float * y; // ...y-coordinates...
  float * z; // ...etc.
  float * q; // These arrays are allocated in the constructor
};
```

Arrays of Structures versus Structures of Arrays





 © Colfax International, 2013–2015

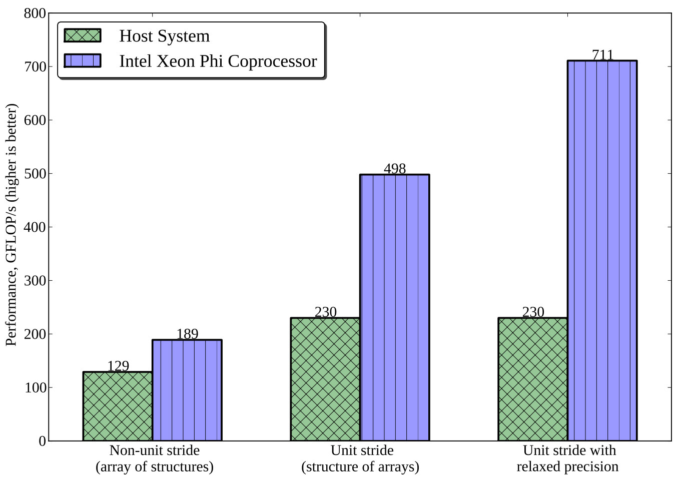
Optimized Solution: Structure of Arrays, Unit-Stride Access

```
1 struct Charge_Distribution {
2   // Data layout permits effective vectorization of Coulomb's law application
3   const int m; // Number of charges
4   float *x, *y, *z, *q; // Arrays of x-, y- and z-coordinates of charges
5 };
```

```
1 // This version vectorizes better thanks to unit-stride data access
2 for (int i=0; i<chg.m; i++)  {
3   // Unit stride: (&chg.x[i+1] - &chg.x[i]) == sizeof(float)
4   const float dx=chg.x[i] - Rx;
5   const float dy=chg.y[i] - Ry;
6   const float dz=chg.z[i] - Rz;
7   phi -= chg.q[i] / sqrtf(dx*dx+dy*dy+dz*dz);
8 }
```

Electric Potential Calculation with Coulomb's Law

Based on 10 FLOPs per interaction:



 © Colfax International, 2013–2015

Data Alignment for Vectorization

 | © Colfax International, 2013–2015

Data Alignment

Array `char* p` is `n`-byte aligned if `((size_t)p%n==0)`.

Processor	Operation	Alignment
Xeon (Westmere and earlier)	SSE load, store	16-byte
Xeon (Sandy Bridge and later)	AVX load, store	32-byte (relaxed)
Xeon Phi (1st gen)	IMCI load, store	64-byte (strict)
Xeon Phi (1st gen)	DMA transfer in offload	4096-byte (preferred)
Xeon Phi (2nd gen)	AVX-512 load, store	64-byte (relaxed)

 © Colfax International, 2013–2015

Data Alignment

- Data alignment on the stack

```
float A[n] __attribute__((aligned(64))); // 64-byte alignment applied
```

  - The address of `A[0]` is a multiple of 64, *i.e.*, aligned on a 64-byte boundary.
  - Setting a very high alignment value may lead to wasted virtual memory.
- Alignment of memory blocks on the heap

```
float *A = (float*)_mm_malloc(n*sizeof(float), 64);
// ...
_mm_free(A);
```

  - `_mm_malloc` and `_mm_free` are aligned version of `malloc` and `free`:
  - the header file `malloc.h` must be included

 © Colfax International, 2013–2015

Data Alignment Hints

Programmer may promise to the compiler (under penalty of segmentation fault) that alignment has been taken care of:

```
float* matrix = _mm_malloc(sizeof(float)*m*n, 64);

// ...

for (int i = b; i < n; i++)
#pragma vector aligned
  for (int j = 0; j < m; j++)
    matrix[i*m + j] += factor*matrix[(i-1)*m + j];
```

This can lead to significant speedups, because compiler will not implement runtime checks for alignment situation.

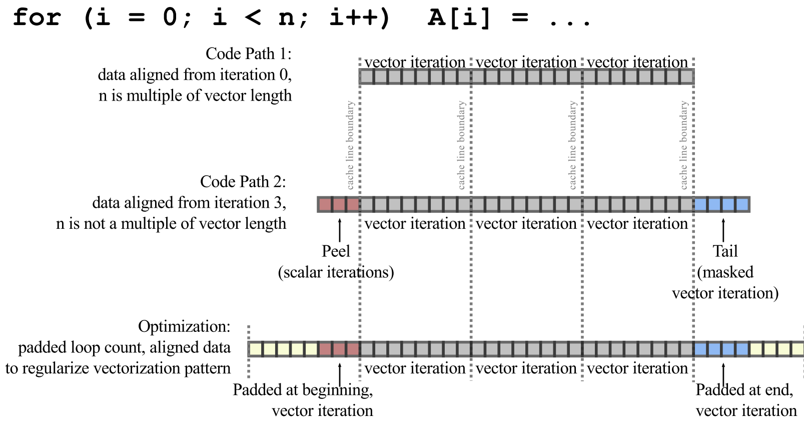
Data Alignment and Padding

To use aligned instructions, you may need to pad the inner dimension of 2-dimensional arrays. Pad to to a multiple of 16 (in single precision) or 8 (double precision) elements.

```
// ... Padding inner dimension so that every row is aligned
if (m % 16 != 0) m += (16 - m%16);

float* matrix = _mm_malloc(sizeof(float)*m*n, 64);

// ...

for (int i = b; i < n; i++)
#pragma vector aligned
  for (int j = 0; j < m; j++)
    matrix[i*m + j] += factor*matrix[(i-1)*m + j];
```

 © Colfax International, 2013–2015

Regularizing Vectorization Pattern



 © Colfax International, 2013–2015

Example: LU Decomposition

```
void LU_decomp(const int n, float* const A) {
  // LU decomposition (Doolittle algorithm)
  // In-place decomposition of form A=LU
  // L is returned below main diagonal of A
  // U is returned at and above main diagonal
  for (int b = 0; b < n; b++) {
    // Strength reduction:
    const float recAbb = 1.0f/A[b*n + b];
    for (int i = b+1; i < n; i++) {
      A[i*n + b] = A[i*n + b]*recAbb;
      for (int j = b+1; j < n; j++)
        A[i*n + j] -= A[i*n + b]*A[b*n + j];
    }
  }
}
```

LU decomposition for small matrices. ($n \approx 128$)

Based on publication: xeonphi.com/papers/lu

Non-optimal Vectorization Pattern.

- Unaligned
- Irregular loop count

LU Decomposition: Regularizing Vectorization

Before:

```
1 for (int b = 0; b < n; b++) {
2   // ...
3   // ...
4   for (int i = b+1; i < n; i++) {
5     // ...
6     for (int j = b+1; j < n; j++)
7       A[i*n+j] -= A[i*n+b]*A[b*n+j];
8   }
9 }
```

After:

```
1 for (int b = 0; b < n; b++) {
2   // ...
3   const int jMin = b - b%16;
4   for (int i = b+1; i < n; i++) {
5     // ...
6     for (int j = jMin; j < n; j++)
7       A[i*n+j] -= L[i*n+b]*A[b*n+j];
8   }
9 }
```

Loop in `j` always starts on a multiple of 64 → aligned access to `A` and `L`

© Colfax International, 2013–2015

Pointer Disambiguation

 © Colfax International, 2013–2015

Multiversioning

```
user@host% icpc -c code.cc -qopt-report -qopt-report-phase:vec
user@host% cat vdep.optrpt
...
LOOP BEGIN at code.cc(4,1)
<Multiversioned v1>
   remark #25228: LOOP WAS VECTORIZED
LOOP END
...
LOOP BEGIN at code.cc(4,1)
<Multiversioned v2>
   remark #15304: loop was not vectorized: non-vectorizable loop instance ....
LOOP END
```

Aliasing (true vector dependence) checked at *runtime* to choose the implementation.

Pointer Disambiguation to Prevent Multiversioning

Prevent Multiversioning by using #pragma ivdep

```
1 #pragma ivdep
2   for (int i = 0; i < n; i++)
3       // ...
```

```
user@host% icpc -c code.cc -qopt-report -qopt-report-phase:vec
user@host% cat vdep.optrpt
...
LOOP BEGIN at code.cc(4,1)
   remark #25228: LOOP WAS VECTORIZED
LOOP END
...
```

LU Decomposition: Compiler hints

- Data Alignment Hint: `#pragma vector aligned`
- Pointer Disambiguation: `#pragma ivdep`

Before:

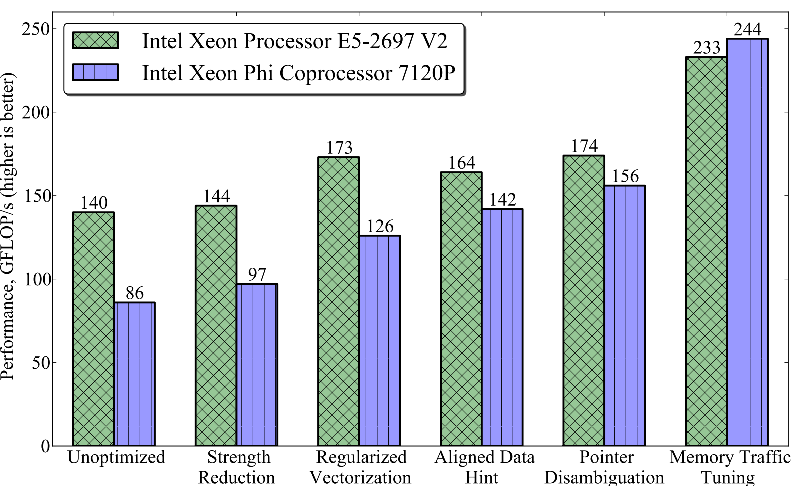
```
for (int b = 0; b < n; b++) {
  const int jMin = b - b%tile;
  const float recAbb = 1.0f/A[b*n+b];
  for (int i = b+1; i < n; i++) {
    L[i*n + b] = A[i*n + b]*recAbb;


    for (int j = jMin; j < n; j++)
      A[i*n+j] -= L[i*n+b]*A[b*n+j];
  }
}
```

After:

```
for (int b = 0; b < n; b++) {
  const int jMin = b - b%tile;
  const float recAbb = 1.0f/A[b*n+b];
  for (int i = b+1; i < n; i++) {
    L[i*n + b] = A[i*n + b]*recAbb;
#pragma vector aligned
#pragma ivdep
    for (int j = jMin; j < n; j++)
      A[i*n+j] -= L[i*n+b]*A[b*n+j];
  }
}
```


© Colfax International, 2013–2015

LU Decomposition: Performance



Paper: http://research.colfaxinternational.com/post/2015/01/27/LU.aspx

Strip-Mining for Vectorization

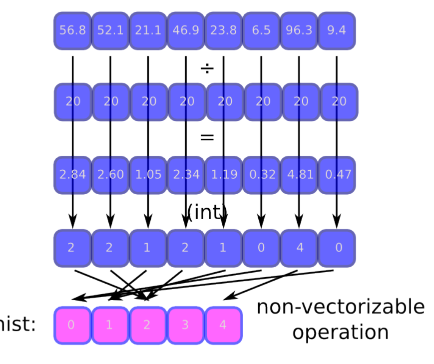
 © Colfax International, 2013–2015

Example: Strip-Mining for Vectorization

Computing a histogram ($m << n$):

```
void Histogram(
  // Ages, values from 0.0f to 100.0f:
  const float* age,
  // Size of array age, n=100000000:
  const int n,
  // Output: counts in groups:
  int* const hist,
  // Size of array hist, m=5:
  const int m,
  // group_width=20.0f
  const float group_width) {
    for (int i = 0; i < n; i++) {
      const int j = int(age[i]/group_width);
      hist[j]++;
    }
}
```

- Code cannot be auto-vectorized
- True vector dependence



Strip-Mining: Method

- Strip-mining is a programming technique that turns one loop into two nested loops.
- used to expose vectorization opportunities in the inner loop.

Original code:

```
for (int i = 0; i < n; i++) {
  // ... do work
}
```

Strip-mined implementation:

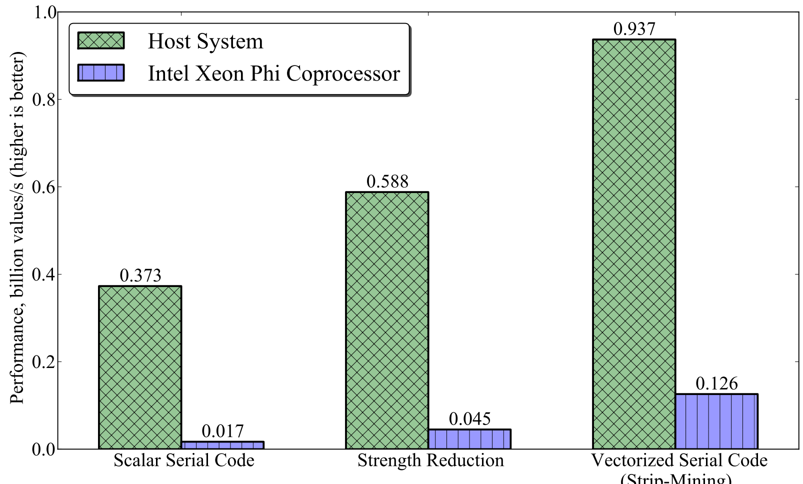
```
const int STRIP=1024;
for (int ii = 0; ii < n; ii += STRIP)
  for (int i = ii; i < ii+STRIP; i++) {
    // ... do work
}
```

The Same Calculation, Strip-Mined, Vectorized

```
void Histogram(const float* age, int* const hist, const int n,
const float group_width, const int m) {
  const int vecLen = 16; // Length of vectorized loop
  const float invGroupWidth = 1.0f/group_width; // Pre-compute the reciprocal
  // Strip-mining the loop in order to vectorize the inner short loop
  // Note: this algorithm assumes n%vecLen == 0.
  for (int ii = 0; ii < n; ii += vecLen) { //Temporary store vecLen indices
    int index[vecLen] __attribute__((aligned(64)));
    // Vectorize the multiplication and rounding
#pragma vector aligned
    for (int i = ii; i < ii + vecLen; i++)
      index[i-ii] = (int) ( age[i] * invGroupWidth );
    // Scattered memory access, does not get vectorized
    for (int c = 0; c < vecLen; c++)
      hist[index[c]]++;
  }
}
```

Strip-Mining for Vectorization

Vectorization improves performance on both platforms. However, more work is needed to take advantage of the MIC architecture. See next section (multi-threading).



 © Colfax International, 2013–2015

Additional Vectorization “Tuning Knobs”

 © Colfax International, 2013–2015

Vectorization Pragmas, Keywords and Compiler Arguments

- `#pragma simd`
- `#pragma vector always`
- `#pragma vector aligned | unaligned`
- `__assume_aligned` keyword
- `#pragma vector nontemporal | temporal`
- `#pragma novector`
- `#pragma ivdep`
- `restrict` qualifier and `-restrict` command-line argument
- `#pragma loop count`
- `-qopt-report -qopt-report-phase:vec`
- `-O[n]`
- `-x[code]`

§10. Optimization of Thread Parallelism

© Colfax International, 2013–2015

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)

 © Colfax International, 2013–2015

Challenges with Thread Parallelism on Xeon Phi

- Multi-core CPU: 4–48 threads, Xeon Phi: 228–244 threads.
- Must have enough parallelism to keep all cores busy
- Must have less synchronization than on CPU
- Must have lower per-thread memory overhead
- Must access core-local data whenever possible
- Must co-exist with vectorization in each core

 © Colfax International, 2013–2015

Reduction instead of Synchronization

 © Colfax International, 2013–2015

Histogram Calculation Example: Adding Thread Parallelism

Incorrect solution: unprotected data races

```
#pragma omp parallel for schedule(guided)
for (int ii = 0; ii < n; ii += vecLen) {
  int index[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
  for (int i = ii; i < ii + vecLen; i++)
    index[i-ii] = (int) ( age[i] * invGroupWidth );
  for (int c = 0; c < vecLen; c++)
    // Multiple threads will write into a single shared container
    // These data races lead to incorrect results!
    hist[index[c]]++;
}
```


© Colfax International, 2013–2015

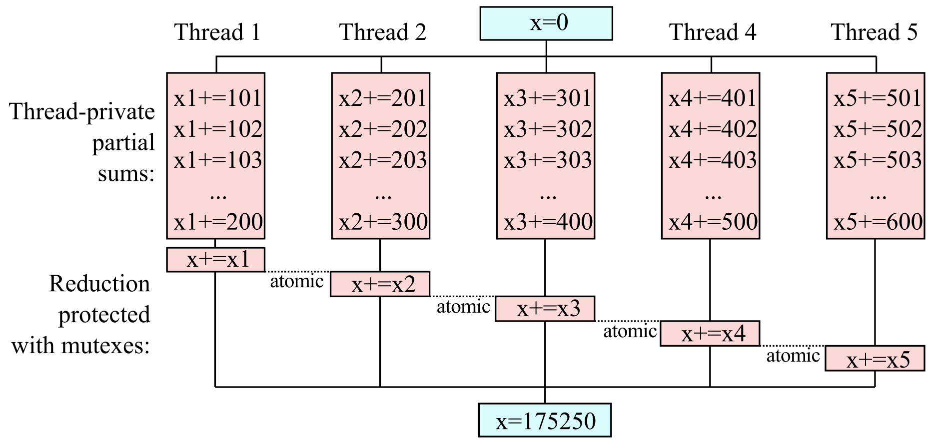
Histogram Calculation Example: Adding Thread Parallelism

Correct, but inefficient solution:

```
#pragma omp parallel for schedule(guided)
for (int ii = 0; ii < n; ii += vecLen) {
  int index[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
  for (int i = ii; i < ii + vecLen; i++)
    index[i-ii] = (int) ( age[i] * invGroupWidth );
  for (int c = 0; c < vecLen; c++)
    // Protect the ++ operation with the atomic mutex (inefficient!)
#pragma omp atomic
    hist[index[c]]++;
}
```

 © Colfax International, 2013–2015

Efficient Parallel Reduction

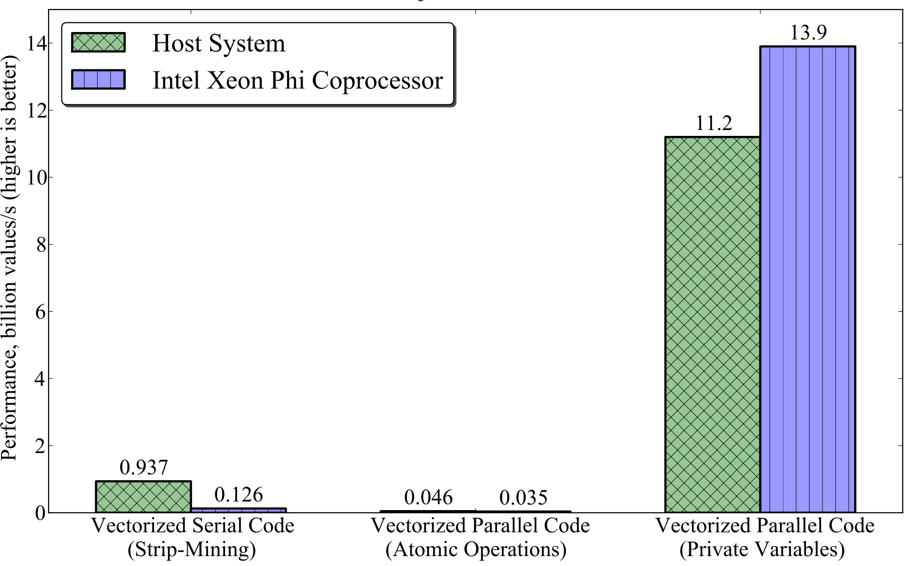


Correct and Efficient Solution with Reduction

```
1  #pragma omp parallel
2  {
3    int hist_priv[m]; // Better idea: thread-private storage
4    hist_priv[:] = 0;
5    int index[vecLen] __attribute__((aligned(64)));
6  #pragma omp for schedule(guided)
7    for (int ii = 0; ii < n; ii += vecLen) {
8  #pragma vector aligned
9      for (int i = ii; i < ii + vecLen; i++)
10       index[i-ii] = (int) ( age[i] * invGroupWidth );
11     for (int c = 0; c < vecLen; c++)
12       hist_priv[index[c]]++;
13   }
14   for (int c = 0; c < m; c++) {
15 #pragma omp atomic
16     hist[c] += hist_priv[c];
17 } }
```

 © Colfax International, 2013–2015

Using Reduction instead of Synchronization

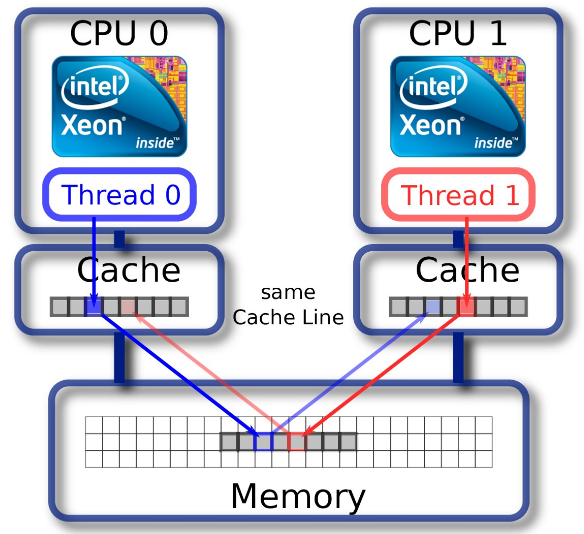


© Colfax International, 2013–2015

Elimination of False Sharing

© Colfax International, 2013–2015

False Sharing. Data Padding and Private Variables



- *False sharing* is similar to *race condition*
- Threads accessing the same *cache line*
- Caused by *coherent caches*
- Cache line is 64-byte wide (in modern Intel architectures)

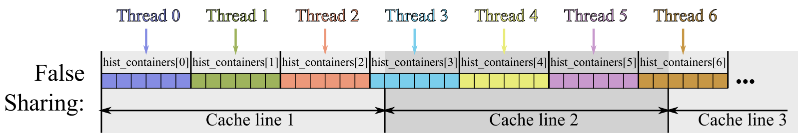
 © Colfax International, 2013–2015

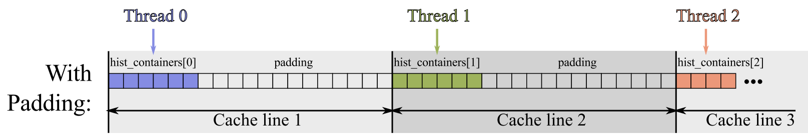
False Sharing. Data Padding and Private Variables

```
1  const int m = 5;
2  int hist_containers[nThreads][m];
3  #pragma omp parallel for
4  for (int ii = 0; ii < n; ii += vecLen) {
5    // ...
6    // False sharing occurs here
7    for (int c = 0; c < vecLen; c++)
8      hist_containers[iThread][index[c]]++;
9  }
10 // Reducing results from all threads to the common histogram hist
11 for (int iThread = 0; iThread < nThreads; iThread++)
12   hist[0:m] += hist_containers[iThread][0:m];
```

- The value of `m=5` is small
- Array elements `hist_thr[0][:]` are within `m*sizeof(int)=20` bytes of array elements `hist_thr[1][:]`

Padding to Avoid False Sharing

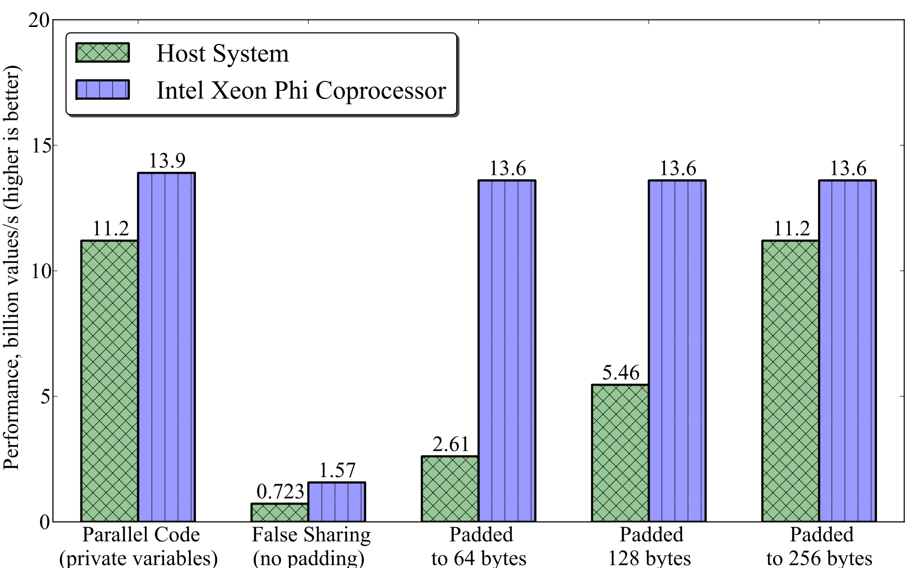





```
1 // Padding to avoid sharing a cache line between threads
2 const int paddingBytes = 64;
3 const int paddingElements = paddingBytes / sizeof(int);
4 const int mPadded = m + (paddingElements-m%paddingElements);
5 int hist_containers[nThreads][mPadded]; // New container
```

 © Colfax International, 2013–2015

Padding to Avoid False Sharing



Expanding Iteration Space

© Colfax International, 2013–2015

Example: Dealing with Insufficient Parallelism

$$S_i = \sum_{j=0}^{n} M_{ij}, \; i = 0 \ldots m. \quad (3)$$

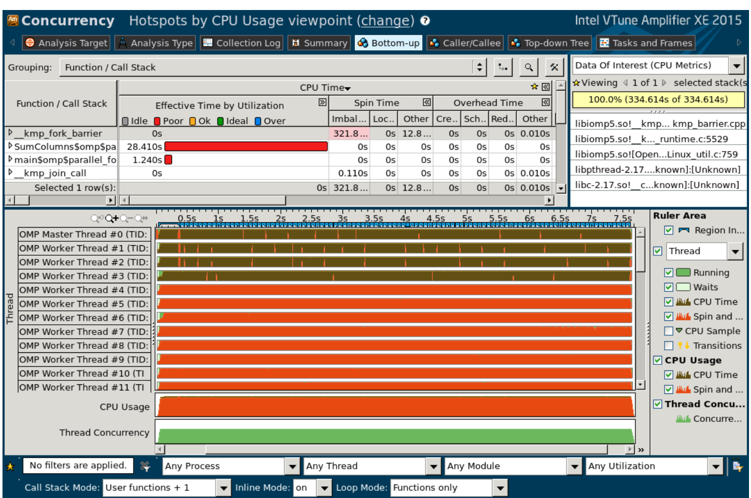
- m=4 is small, smaller than the number of threads in the system
- n $\approx 10^8$ is large enough so that matrix does not fit into cache

```
1 void sum_unoptimized(const int m, const int n, long* M, long* s){
2 #pragma omp parallel for
3   for (int i=0; i<m; i++) { // m=4
4     long sum=0;
5 #pragma vector aligned
6     for (int j=0; j<n; j++) // n=100000000
7       sum+=M[i*n+j];
8   s[i]=sum; }}
```

 © Colfax International, 2013–2015

Dealing with Insufficient Parallelism

VTune Analysis: Row-Wise Reduction of a Short, Wide Matrix

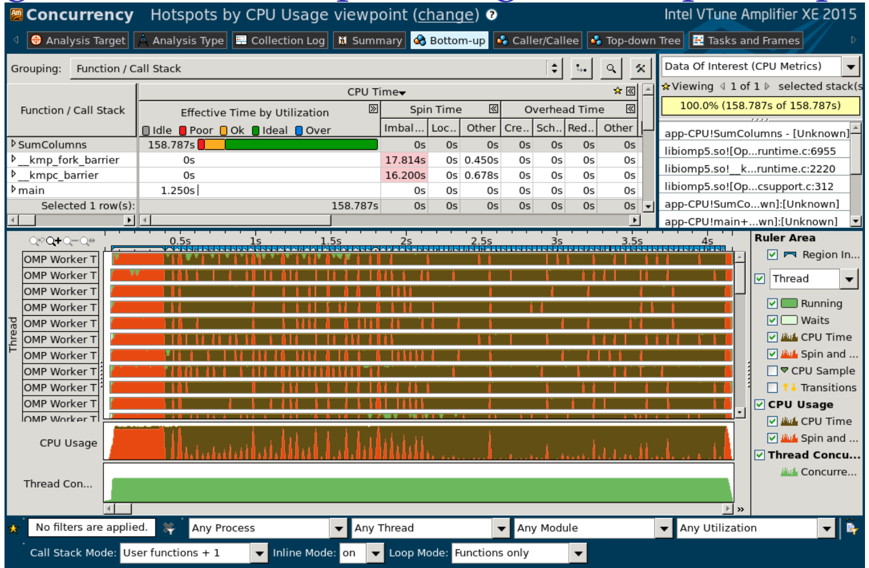


 © Colfax International, 2013–2015

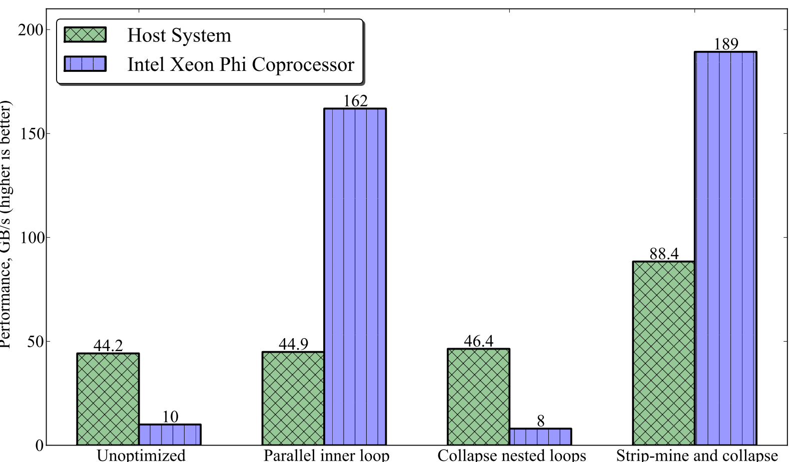
Exposing Parallelism: Strip-Mining and Loop Collapse

```
void sum_stripmine(const int m, const int n, long* M, long* s){
  const int STRIP=1024;
  assert(n%STRIP==0);
  s[0:m]=0;
#pragma omp parallel
  {
    long sum[m];   sum[0:m]=0;
#pragma omp for collapse(2) schedule(guided)
    for (int i=0; i<m; i++)
      for (int jj=0; jj<n; jj+=STRIP)
#pragma vector aligned
        for (int j=jj; j<jj+STRIP; j++)
          sum[i]+=M[i*n+j];
    for (int i=0; i<m; i++)          // Reduction
#pragma omp atomic
      s[i]+=sum[i];
}   }
```

Exposing Parallelism: Strip-Mining and Loop Collapse



Dealing with Insufficient Parallelism



Techniques that did not work well are discussed in the book.

 © Colfax International, 2013–2015

Controlling Thread Affinity

 © Colfax International, 2013–2015

Setting Thread Affinity

- OpenMP threads may migrate from one core to another according to OS decisions.
- Forbid migration — increase the performance.
- Control: environment variable `KMP_AFFINITY`

 © Colfax International, 2013–2015

The KMP_AFFINITY Environment Variable

```
KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]
```

modifier:

- verbose/nonverbose
- respect/norespect
- warnings/nowarnings
- granularity=core or thread

- type=compact, scatter or balanced
- type=explicit, proclist=[<proc_list>]
- type=disabled or none.

```
vega@lyra% export MIC_ENV_PREFIX=XEONPHI
vega@lyra% export KMP_AFFINITY=compact,granularity=thread
vega@lyra% export XEONPHI_KMP_AFFINITY=balanced,granularity=thread
```


© Colfax International, 2013–2015

Uses of Thread Affinity

- **Bandwidth-bound applications**: 1 thread per core + “scatter” affinity pattern.
- **Compute-bound applications**: 2 (Xeon) or up to 4 (Xeon Phi) threads per core + “compact” affinity (or “balanced” for 3 threads on Xeon Phi).
- **NUMA** (Non-Uniform Memory Access) systems: use “offset” + “compact” to pin processes to respective NUMA nodes.

 © Colfax International, 2013–2015

Bandwidth-bound, KMP_AFFINITY=scatter

```
vega@lyra% export OMP_NUM_THREADS=32
vega@lyra% export KMP_AFFINITY=none
vega@lyra% for i in {1..4} ; do ./rowsum_stripmine | tail -1; done
Problem size: 2.980 GB, outer dimension: 4, threads: 32
Strip-mine and collapse: 0.061 +/- 0.002 seconds (52.89 +/- 1.31 GB/s)
Strip-mine and collapse: 0.059 +/- 0.002 seconds (54.11 +/- 1.56 GB/s)
Strip-mine and collapse: 0.077 +/- 0.001 seconds (41.71 +/- 0.69 GB/s)
Strip-mine and collapse: 0.070 +/- 0.005 seconds (45.59 +/- 3.14 GB/s)
vega@lyra% export OMP_NUM_THREADS=16
vega@lyra% export KMP_AFFINITY=scatter
vega@lyra% for i in {1..4}; do ./rowsum_stripmine | tail -1 ; done
Problem size: 2.980 GB, outer dimension: 4, threads: 16
Strip-mine and collapse: 0.059 +/- 0.004 seconds (54.47 +/- 3.25 GB/s)
Strip-mine and collapse: 0.061 +/- 0.004 seconds (52.30 +/- 3.30 GB/s)
Strip-mine and collapse: 0.062 +/- 0.005 seconds (51.37 +/- 4.29 GB/s)
Strip-mine and collapse: 0.058 +/- 0.001 seconds (55.48 +/- 1.27 GB/s)
```

 © Colfax International, 2013–2015

Compute-Bound, KMP_AFFINITY=compact/balanced

```
vega@lyra% icpc -o bench-dgemm -mkl -mmic bench-dgemm.cc
vega@lyra% micnativeloadex ./bench-dgemm
Iteration 1: 312.7 GFLOP/s
Iteration 2: 346.5 GFLOP/s
Iteration 3: 348.5 GFLOP/s
Iteration 4: 347.2 GFLOP/s
Iteration 5: 348.3 GFLOP/s

vega@lyra% micnativeloadex ./bench-dgemm -e "KMP_AFFINITY=compact"
Iteration 1: 626.8 GFLOP/s
Iteration 2: 769.1 GFLOP/s
Iteration 3: 769.4 GFLOP/s
Iteration 4: 769.3 GFLOP/s
Iteration 5: 769.4 GFLOP/s
```

§11. Optimization of Memory Access

 © Colfax International, 2013–2015

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)

 © Colfax International, 2013–2015

Challenges with Memory Access on Xeon Phi

- More threads than CPU, same amount of Level-2 cache (~30 MB)
- No hardware prefetching from Level-2 to Level-1
- High penalty for data page walks

"Rule of Thumb" for memory optimization: locality of data access in space and in time.

Spatial locality = data structures (packing, reordering).

Temporal locality = order of operations (e.g., loop tiling).

Roofline Model

 © Colfax International, 2013–2015

Arithmetic Intensity and Roofline Model

Theoretical estimates, Intel Xeon Phi coprocessor(Double Precision)

$$\text{Arithmetic Performance} = 60 \times 1.0 \times (512/64) \times 2 \approx 1\,\text{TFLOP/s} = 8\,\text{TB/s}$$

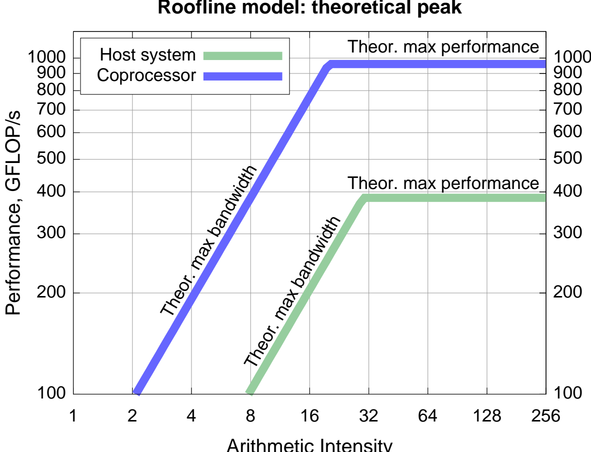
$$\text{Memory Bandwidth} = \eta \times 6.0 \times 8 \times 2 \times 4 = \eta \times 384\,\text{GB/s}$$

$$\text{Ratio} = 8\ /\ .384 \approx 20\ (\text{FLOPs})/(\text{Memory Access})$$

If the Arithmetic Intesity is...

- $>$ 20 (FLOPs)/(Memory Access) — Compute Bound Application
- $<$ 20 (FLOPs)/(Memory Access) — Bandwidth Bound Appliction

Arithmetic Intensity and Roofline Model



More on roofline model: Williams et al.

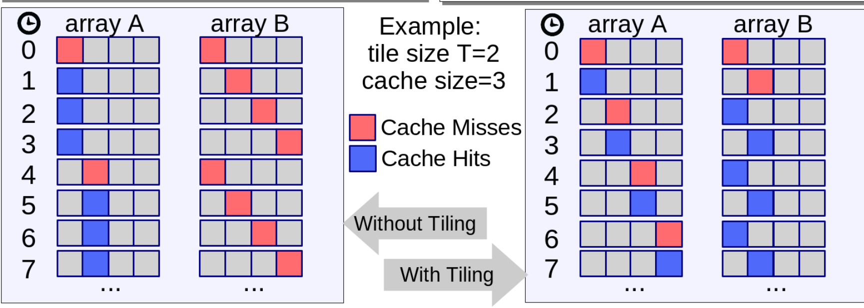
Memory Access and Cache Utilization

 © Colfax International, 2013–2015

Loop Tiling (Blocking)

```
/* Nested loops without tiling.
Array B[] does not fit into cache */
for (int i = 0; i < iMax; ++i)
 for (int j = 0; j < jMax; ++j)
  PerformWork(A[i], B[j]);
```

```
/* Tiled nested loops */
for (int jj = 0; jj < jMax;  jj+= T)
 for (int i = 0; i < iMax; ++i)
  for (int  j = jj;  j < jj+T; ++j)
   PerformWork(A[i], B[j]);
```



Cache Hit Rate = 6/16

SLOWER

Cache Hit Rate = 10/16

FASTER

Loop Tiling (Cache Blocking) – Procedure

```
1  // Original code:
2  for (int i = 0; i < m; i++)
3    for (int j = 0; j < n; j++)
4      compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
1  // Step 1: strip-mine
2  for (int i = 0; i < m; i++)
3    for (int jj = 0; jj < n; j += TILE)
4      for (int j = jj; j < jj + TILE; j++)
5        compute(a[i], b[j]); // Same order of operation as original
```

```
1  // Step 2: permute
2  for (int jj = 0; jj < n; j += TILE)
3    for (int i = 0; i < m; i++)
4      for (int j = jj; j < jj + TILE; j++)
5        compute(a[i], b[j]); // Re-use to j=jj sooner
```

Example: Matrix-vector Multiplication

$$c_i = \sum_{j=0}^{m} A_{ij} b_j, \quad i = 0, 1, \ldots, (n-1). \tag{4}$$

```
1 void Multiply(const double* const A, const double* const b,
2               double* const c, const long n, const long m){
3   assert(n%64 == 0);
4 #pragma omp parallel for
5   for (long i = 0; i < m; i++)
6 #pragma vector aligned
7     for (long j = 0; j < n; j++)
8       c[i] += A[i*n+j] * b[j];
9 }
```

Non-optimal performance due to inefficient cache use

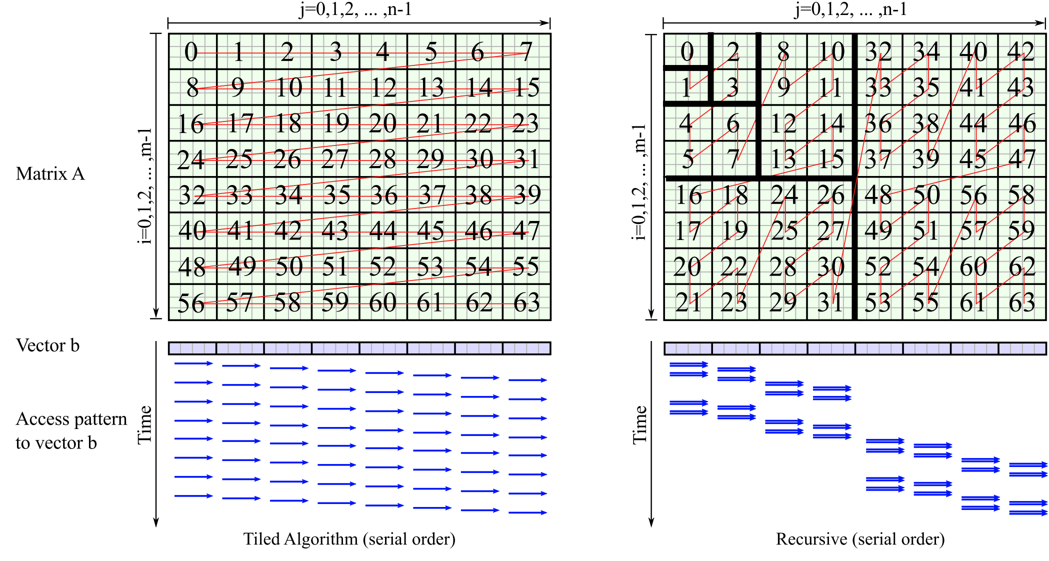
Applying Tiling

```
1    const long jTile = 4096L; assert(n%jTile == 0);
2  #pragma omp parallel
3    {
4      double temp_c[m] __attribute__((aligned(64)));
5      temp_c[:] =0;
6  #pragma omp for
7      for (long jj =0; jj < n; jj+=jTile) // Loop Tiling in j
8        for (long i = 0; i < m; i++)
9  #pragma vector aligned
10         for (long j =jj; j < jj+jTile; j++)
11           temp_c[i] += A[i*n+j] * b[j];
12
13     for(long i = 0; i < m; i++) { // Reduction
14 #pragma omp atomic
15       c[i]+= temp_c[i];
16 } } }
```

Cache Blocking + Strip-Mine and Collapse

```
1    const long iTile = 64L;   assert(m%iTile == 0);
2    const long jTile = 4096L; assert(n%jTile == 0);
3  #pragma omp parallel
4    {
5      double temp_c[m] __attribute__((aligned(64))); temp_c[:] =0;
6  #pragma omp for collapse(2)
7      for (long ii = 0; ii < m; ii += iTile)
8        for (long jj = 0; jj < n; jj += jTile)
9          for (long i = ii; i < ii+iTile; i++)
10 #pragma vector aligned
11           for (long j =jj; j < jj+jTile; j++)
12             temp_c[i] += A[i*n+j] * b[j];
13
14     for(long i = 0; i < m; i++) {
15 #pragma omp atomic
16       c[i]+= temp_c[i];
17 } } }
```

Cache-Oblivious Matrix Traversal Diagram



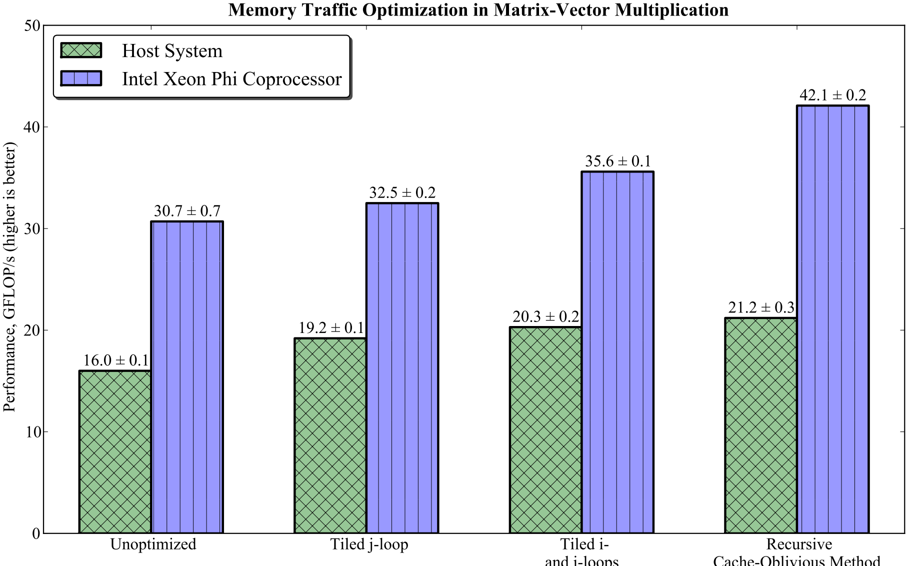
 © Colfax International, 2013–2015

Cache-Oblivious Algorithms

```
1  void RecursMultiply(const double* const A, const double* const b,
2         double* const c, const long n, const long m, const long lda){
3    const long jThreshold = 8192L; assert(n%jThreshold == 0);
4    const long iThreshold = 64L;   assert(m%iThreshold == 0);
5    if ((m<=iThreshold) && (n<=jThreshold)) { // Recursion threshold
6      // .... Base Case: Compute the result inside the tile ... //
7    } else { // Recursive divide-and-conquer
8      if (m*jThreshold > n*iThreshold) { // Split i-wise
9        double c1[m/2] __attribute__((aligned(64)));
10 #pragma omp task
11       { RecursMultiply(&A[0*lda + 0], &b[0], c1, n, m/2, lda); }
12       double c2[m/2] __attribute__((aligned(64)));
13       RecursMultiply(&A[(m/2)*lda + 0], &b[m/2], c2, n, m/2, lda);
14 #pragma omp taskwait
15       c[0:m/2] += c1[0:m/2]; c[m/2:m/2] += c2[0:m/2]; // Reduction
16     } else { // .... Split j-wise .... // }
17 } }
```


© Colfax International, 2013–2015

Performance of Matrix Vector Multiplication



First Touch Allocation

 © Colfax International, 2013–2015

Allocation on First Touch

- Memory allocation occurs not during `_mm_malloc()`, but upon the first write to the buffer ("first touch")
- Default NUMA allocation policy is "on first touch"
- For better performance in NUMA systems, initialize data with the same parallel pattern as during data usage
- Applies to multi-socket Xeon-based systems

```
float* A = (float*)_mm_malloc(n*m*sizeof(float), 64);

// Initializing from parallel region for better performance
#pragma omp parallel for
for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    A[i*m + j] = 0.0f;
```


© Colfax International, 2013–2015

§12. Optimization of Communication

© Colfax International, 2013–2015

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)

 © Colfax International, 2013–2015

Offload Traffic Optimization

© Colfax International, 2013–2015

PCIe Bandwidth Considerations

Does offload pay off?

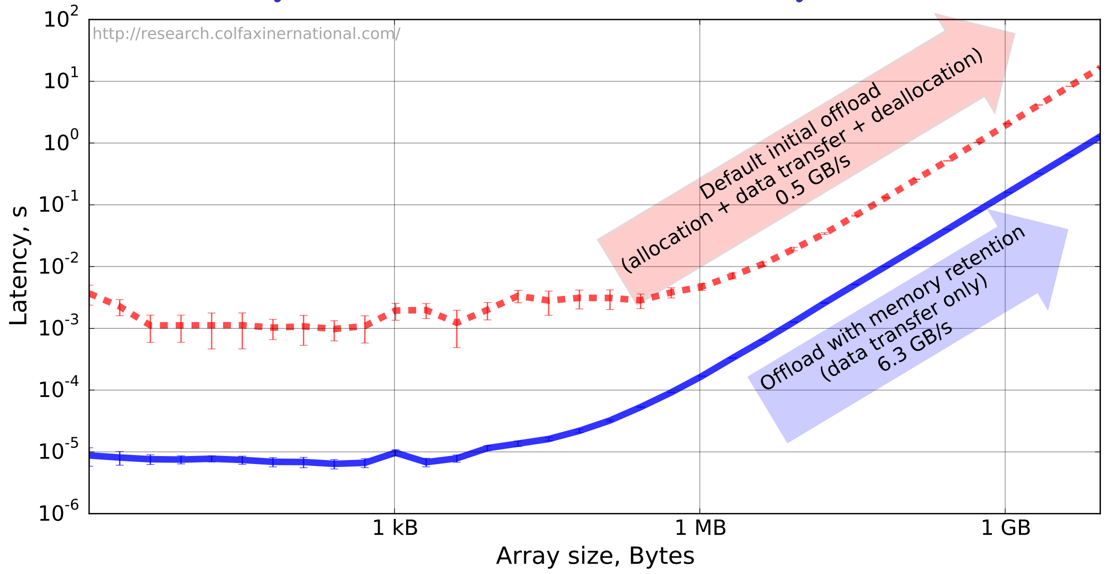
- PCIe bandwidth ≈7 GB/s
- 1 TFLOP/s in double precision is 8 TB/s

Conclusion: offload if algorithm performs $\gg$ 1000 operations per transferred word.

Good for offload:	**Bad for offload:**
Strong scaling (e.g., $O(n^2)$)	Weak scaling (e.g., $O(n)$, $O(n \log n)$)
Large problems ($n \gg 1000$)	Small problems ($n \lesssim 1000$)

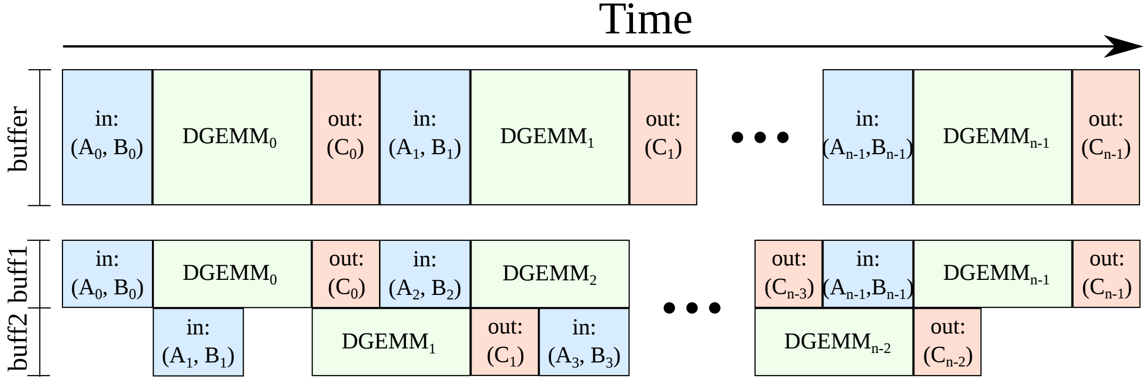
© Colfax International, 2013–2015

Offload Latency With and Without Memory/Data Retention



Overlapping Communication and Computation

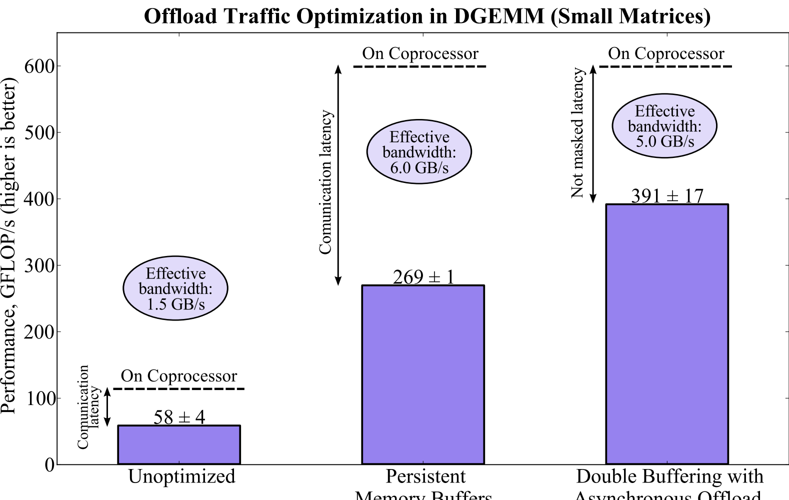
Double buffering may help to mask communication latency



 © Colfax International, 2013–2015

Performance Effect of Double Buffering

DGEMM on a set of small matrices (1024x1024):



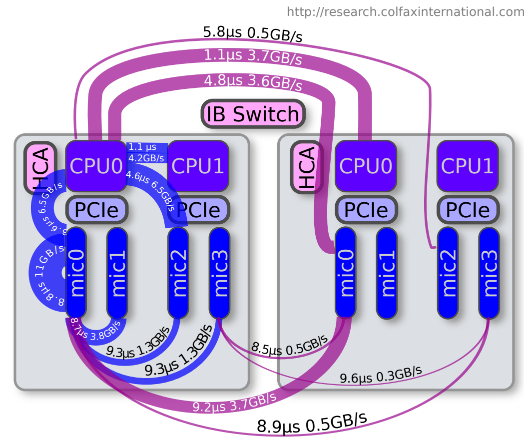
 © Colfax International, 2013–2015

MPI Communication Controls

 © Colfax International, 2013–2015

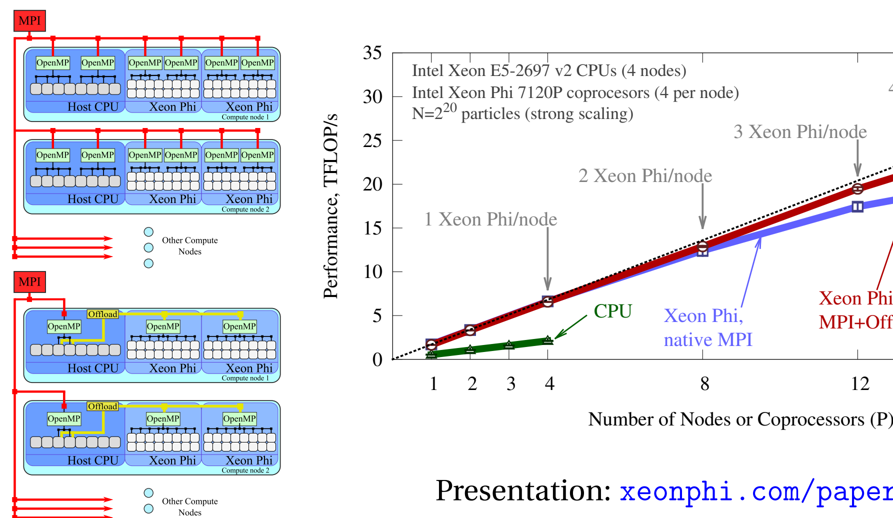
MPI Fabric Selection

- MPI communication between CPU and coprocessors as efficient as offload
- Peer-to-peer communication not uniform, but better than with Gigabit Ethernet
- Control: environment variable `I_MPI_FABRICS`



Our publication with details:
xeonphi.com/papers/p2p

Improving Communication with Offload+MPI



Presentation: xeonphi.com/papers/sc14

© Colfax International, 2013–2015

§13. Additional Topics on MPI

 © Colfax International, 2013–2015

Load Balancing in Heterogeneous Applications

 © Colfax International, 2013–2015

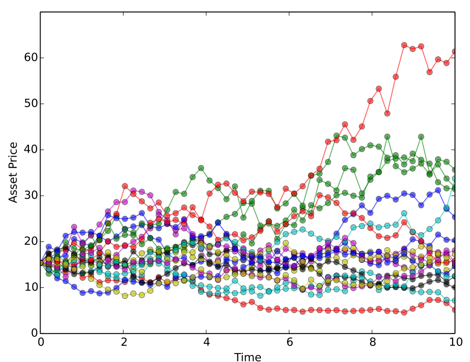
Example: the Monte Carlo Method of Asian Option Pricing

1) Simulate Random-Walk of Asset Price

$$dS(t) = \mu S(t)dt + \sigma S(t)dB(t)$$

Using the Solution

$$S(t) = S(0)e^{(\mu - \frac{1}{2}\sigma^2)t + \sigma\sqrt{t}N(0,1)}$$



2) Perform Asian Option Price Averaging

$$\langle S\rangle_{\text{arithm}} = \frac{1}{N}\sum_{i=0}^{N-1} S(t_i),$$

$$\langle S\rangle_{\text{geom}} = \exp\left(\frac{1}{N}\sum_{i=0}^{N-1}\log S(t_i)\right)$$

3) Compute Discounted Pay-off

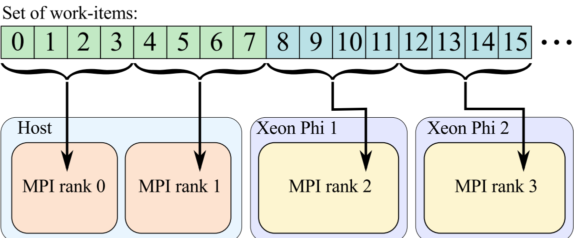
$$P_{\text{put}} = e^{-rT}\mathbb{E}\left(\max\left\{0; K - \langle S\rangle\right\}\right),$$

$$P_{\text{call}} = e^{-rT}\mathbb{E}\left(\max\left\{0; \langle S\rangle - K\right\}\right)$$

 © Colfax International, 2013–2015

Heterogeneous Calculation without Load Balancing

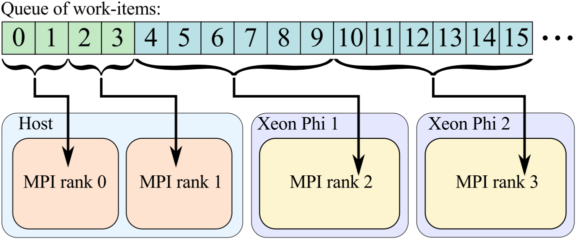
```
1  const double optionsPerProcess = double(nOptions)/double(mpiWorldSize);
2  const int myFirstOption = int(optionsPerProcess*(myRank));
3  const int myLastOption  = int(optionsPerProcess*(myRank+1));
4
5  // Static, even load distribution: assign options to ranks
6  for (int i = myFirstOption; i < myLastOption; i++)
7    ComputeOptionPayoffs(option[i], payoff[i]);
```



 © Colfax International, 2013–2015

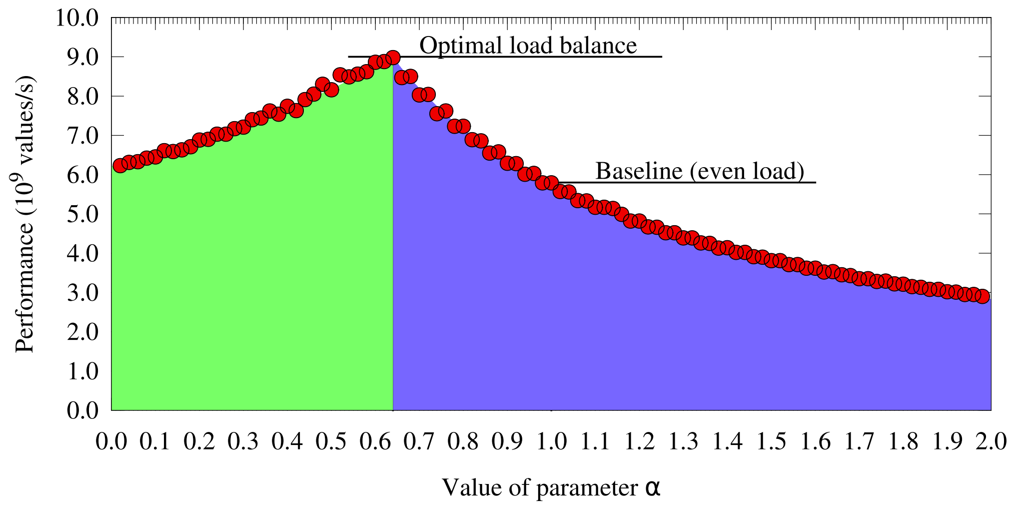
Static Load Balancing

```
1  if (rankTypes[myRank] == 0) { // I am a MIC-based rank
2    double optionsPerProc = double(lastOptForCPUs)/double(cpuRanks.size());
3    myFirstOpt = int(optionsPerProc*(myGroupRank));
4    myLastOpt  = int(optionsPerProc*(myGroupRank+1.0));
5  } else { // I am a CPU-based rank
6    double optionsPerProc = double(nOpts-lastOptForCPUs)/double(micRanks.size());
7    myFirstOpt=lastOptForCPUs+int(optionsPerProc*(myGroupRank));
8    myLastOpt=lastOptForCPUs+int(optionsPerProc*(myGroupRank+1.0)); }
```



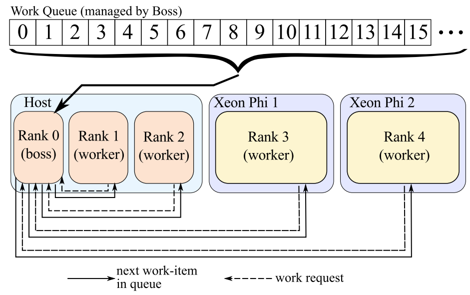
 © Colfax International, 2013–2015

Static Load Balancing: Parameter Tuning

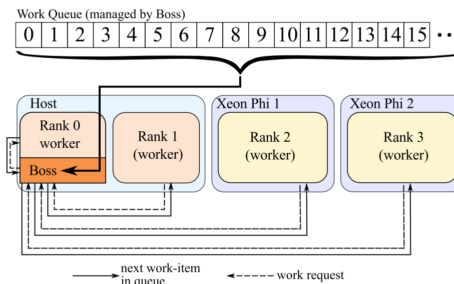


Dynamic Load Balancing

```
1  if (myRank == 0) // Boss's branch
2    DistributeWork(nOptions, option, mpiWorldSize);
3  else // Workers' branch
4    ReceiveWork(option, payoff, myRank);
```

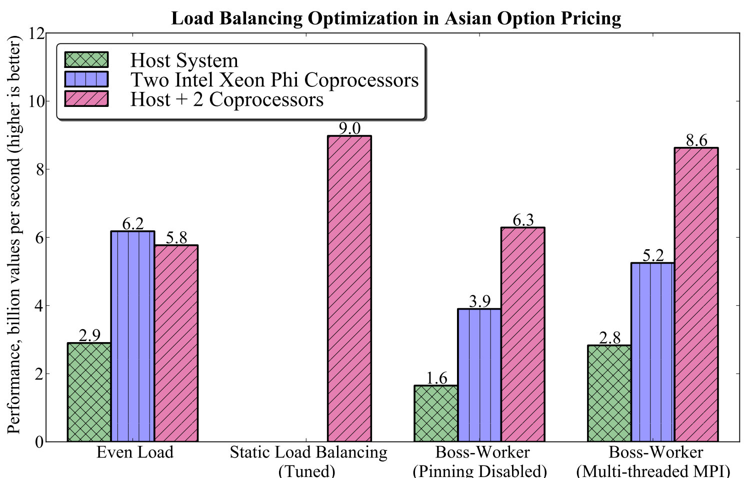


or



 © Colfax International, 2013–2015

Performance with Different Scheduling Modes



Refer to the book for explanation on the last two results.

© Colfax International, 2013–2015

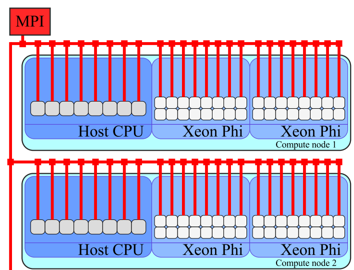
Inter-Operation with OpenMP

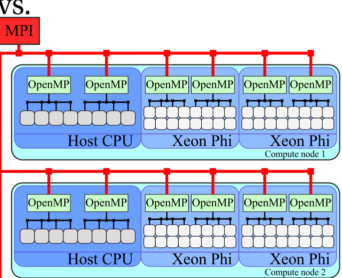
© Colfax International, 2013–2015

Hybrid MPI+OpenMP

Using OpenMP inside of MPI processes:

- Reduces the memory footprint
- Decreases the number of MPI ranks, which reduces communication
- May incur thread synchronization overhead
- Optimal number of threads in MPI processes must be established empirically





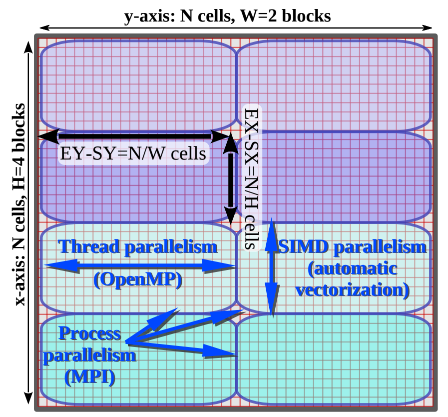
Hybrid MPI+OpenMP

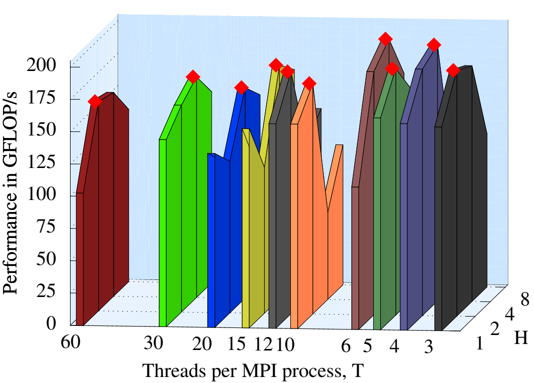
- For MPI calls from multiple MPI threads, use `-mt_mpi`
- MPI pins processes to cores and sets OpenMP affinity for them.
- To tune pinning: `I_MPI_PIN`, `I_MPI_PIN_DOMAIN`
- To diagnose process pinning: `I_MPI_DEBUG=4`
- More information in the MPI Reference Manual

 © Colfax International, 2013–2015

Example of OpenMP and MPI Inter-Operation

Number of threads per process may be a tuning parameter:





Case study: `xeonphi.com/papers/shallow`

§14. Additional Resources

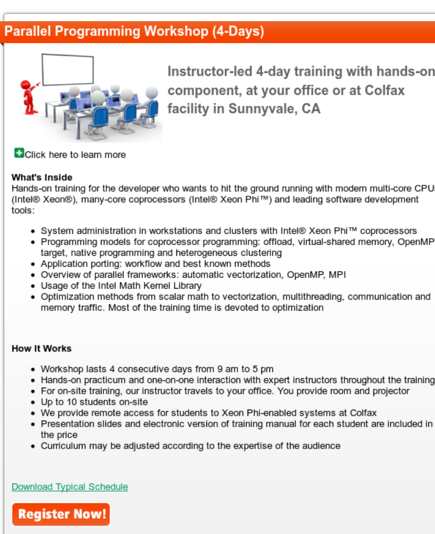
 © Colfax International, 2013–2015

Where to Get More Information

 © Colfax International, 2013–2015

4-Day Hands-on Training

- More in-depth version of this course is available as 4-day workshop
- Intructor-led hands-on practicum
- Access to systems with 2 or more coprocessors
- xeonphi.com/training



Reference Guides

- Intel C++ Compiler 15.0 User and Reference Guide
- Intel VTune Amplifier XE User's Guide
- Intel Trace Analyzer and Collector Reference Gude
- Intel MPI Library for Linux* OS Reference Manual
- Intel Math Kernel Library Reference Manual
- Intel Software Documentation Library
- MPI Routines on the ANL Web Site
- OpenMP Specifications

 © Colfax International, 2013–2015

Online Resources

Intel:

1. Developer Portal — general information, case studies, links
2. Intel MIC Forum — technical support by community
3. Top 10 List — getting started with Xeon Phi guide

Colfax:

1. Colfax Research — educational and research publications
2. Video Course — based on this training (work in progress)

 © Colfax International, 2013–2015

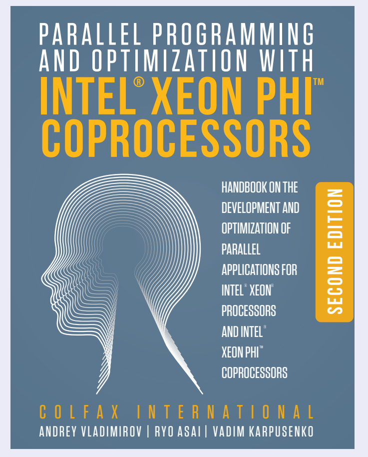
Supplementary Materials: Textbook

ISBN: 978-0-9885234-0-1 (508 pages, Electronic or Print)

Parallel Programming
and Optimization with
Intel® Xeon Phi™
Coprocessors

Handbook on the Development and
Optimization of Parallel Applications
for Intel® Xeon® Processors
and Intel® Xeon Phi™ Coprocessors

© Colfax International, 2015



xeonphi.com/book

Colfax International

 © Colfax International, 2013–2015

Colfax Complete Xeon Phi Ecosystem

- Servers: xeonphi.com/servers
- Workstations: xeonphi.com/workstations
- Starter kits: xeonphi.com/starterkit
- Storage: xeonphi.com/lustre
- Specials: xeonphi.com/promo195
- Book: xeonphi.com/book
- Training: xeonphi.com/training
- Research: xeonphi.com/research
- Consulting: xeonphi.com/consulting

Follow us on Twitter:
@colfaxintl

 © Colfax International, 2013–2015

Research and Consulting

Colfax offers consulting services for Enterprises, Research Labs, and Universities. We can help you to:

- Optimize your existing application to take advantage of all levels of hardware parallelism
- Future-proof for upcoming innovations in computing solutions.
- Accelerate your application using coprocessor technologies.
- Investigate the potential system configurations that satisfy your cost, power and performance requirements.
- Take a deep dive to develop a novel approach.

For more details, contact us at phi@colfax-intl.com to discuss what we can do together.

© Colfax International, 2013–2015

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

 © Colfax International, 2013–2015