Parallel Programming and Optimization with Intel Xeon Phi Coprocessors

Colfax Developer Training — One-Day Seminar

Colfax International — @colfaxintl

February 2016, Rev. 20b

 © Colfax International, 2013–2016

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!

http://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.

© Colfax International, 2013–2016

List of Topics

1. Welcome
2. “Teaser”: N-body Simulation
3. Learn More
4. Hardware Orientation
5. Programming Coprocessors
6. Performance Fundamentals
7. Scalar Tuning
8. Vectorization
9. Threading
10. Memory Access
11. Distributed Computing with MPI
12. Additional Resources

 © Colfax International, 2013–2016

Remote Access for Hands-On Exercises

- Visit URL shown on your invitation and enter the token
- **First**, set up terminal access
- **Second**, set up virtual desktop (optional)
- Follow along or use access after the class (till end of week)

© Colfax International, 2013–2016

Where to Get These Slides

xeonphi.com/training/slides

§2. "Teaser": N-body Simulation

© Colfax International, 2013–2016

Physics

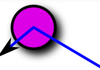
Gravitational N-body dynamics:

Newton's law of universal gravitation:

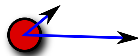
$$M_i \vec{R}_i''(t) = G \sum_j \frac{M_i M_j}{\left|\vec{R}_i - \vec{R}_j\right|^3} \left(\vec{R}_j - \vec{R}_i\right)$$

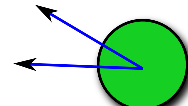
where:

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



particles are attracted to each other
with the gravitational force





Application

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

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

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

© Colfax International, 2013–2016

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
```

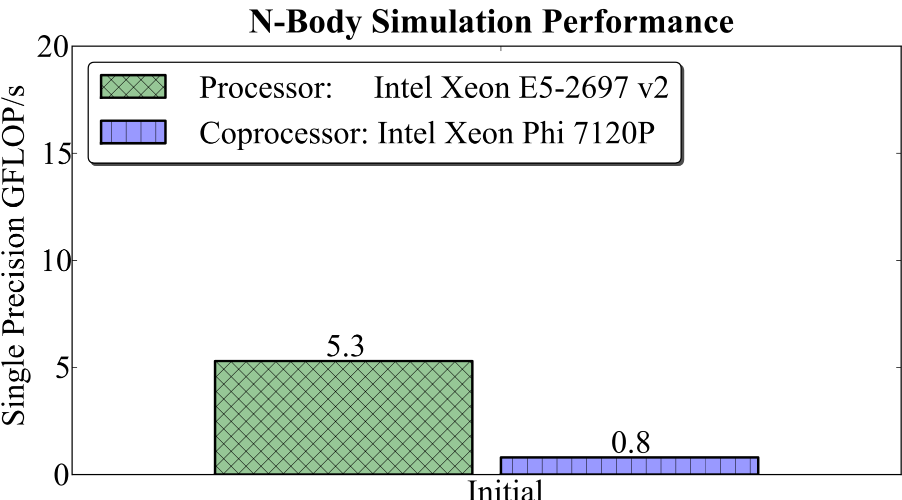
 © Colfax International, 2013–2016

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   ...
```


© Colfax International, 2013–2016

Performance of Initial Implementation



Incorporating Thread Parallelism

Before:

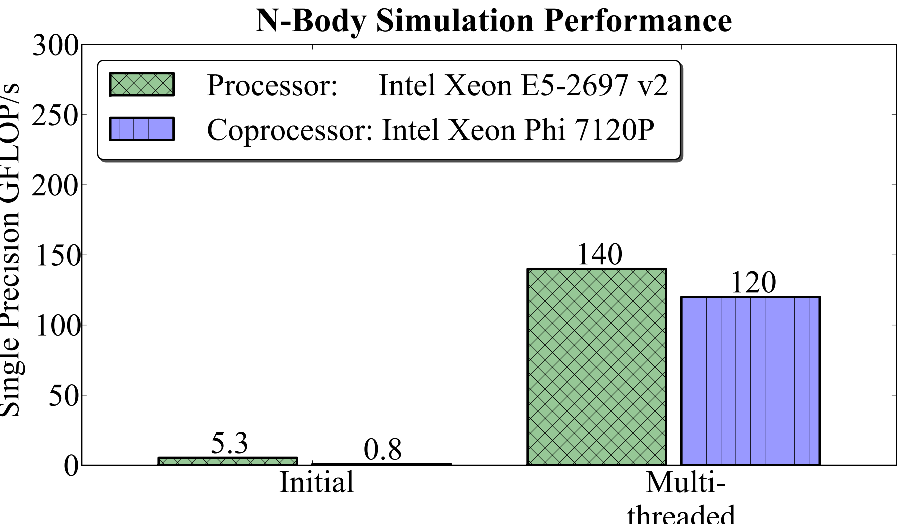
```
  for (int i = 0; i < nParticles; i++) {   // Particles that experience force
    float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
    for (int j = 0; j < nParticles; j++) { // Particles that exert force
      // Newton's law of universal gravity
      ...
```

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
      ...
```


© Colfax International, 2013–2016

Performance with Thread Parallelism



Vectorizing with Unit-Stride Memory Access

Before:

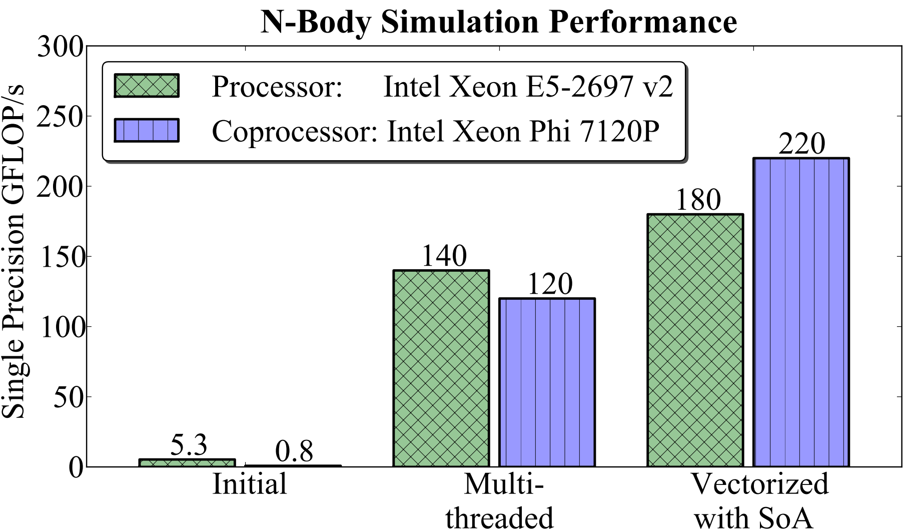
```
struct ParticleType {
  float x, y, z, vx, vy, vz;
};  // ...
      const float dx = particle[j].x - particle[i].x;
      const float dy = particle[j].y - particle[i].y;
      const float dz = particle[j].z - particle[i].z;
```

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, 2013–2016

Performance with Improved Vectorization



© Colfax International, 2013–2016

Improving Scalar Expressions

Before:

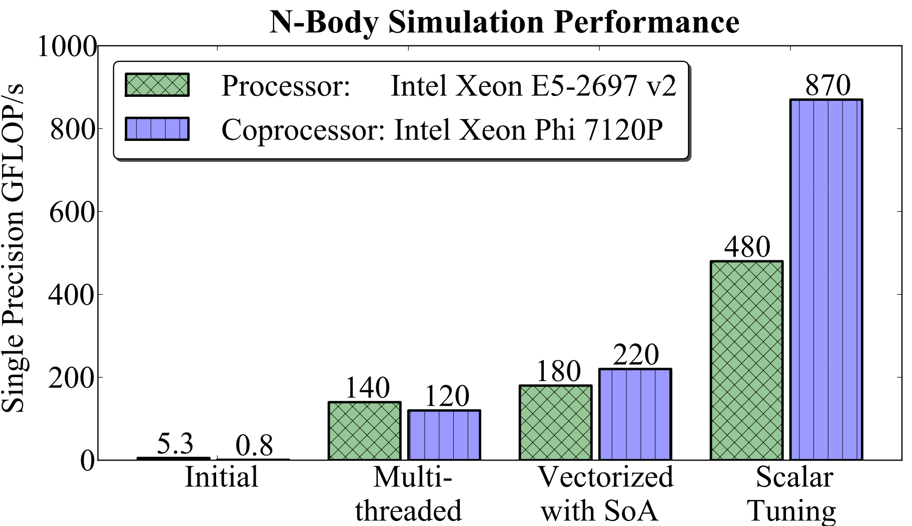
```
1        const float drSquared  = dx*dx + dy*dy + dz*dz + 1e-20;
2        const float drPower32  = pow(drSquared, 3.0/2.0);
3        // Calculate the net force
4        Fx += dx/drPower32;  Fy += dy/drPower32;  Fz += dz/drPower32;
```

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

Performance after Scalar Tuning



Improving Cache Traffic

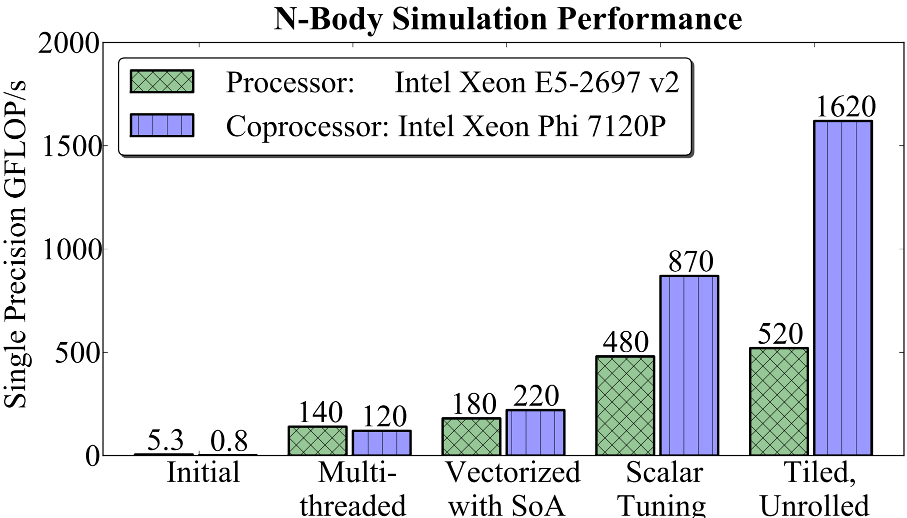
Before:

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

After: (tileSize = 16)

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

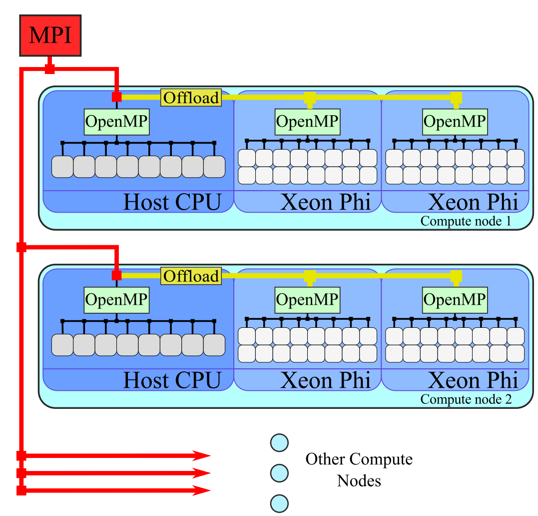
Performance with Cache Optimization (Loop Tiling)



© Colfax International, 2013–2016

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`



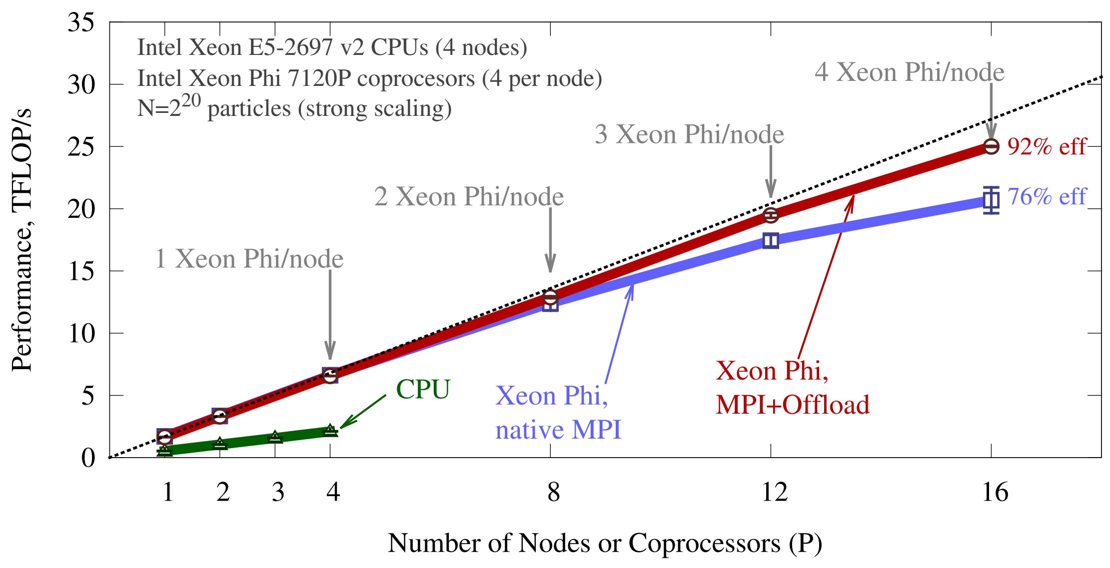
 © Colfax International, 2013–2016

MPI with offload implementation

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


© Colfax International, 2013–2016

Results with MPI+Offload



§3. Learn More

 © Colfax International, 2013–2016

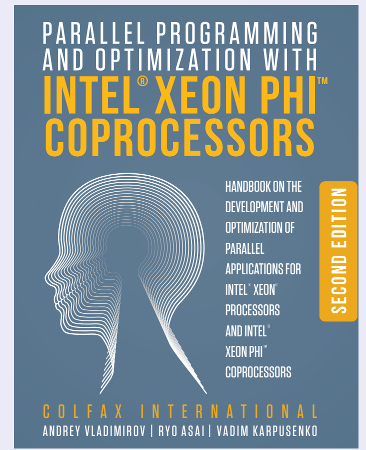
Textbook

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

Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors

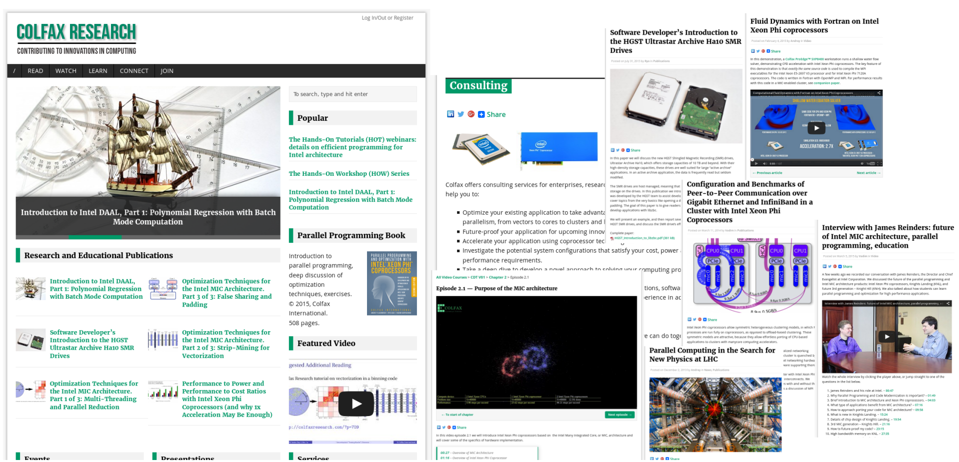
Handbook on the Development and Optimization of Parallel Applications for Intel® Xeon® Processors and Intel® Xeon Phi™ Coprocessors

© Colfax International, 2015



http://xeonphi.com/book

Colfax Research



http://colfaxresearch.com/

(already registered? get $10 off the book)

HOW Series

THE "HOW" (HANDS ON WORKSHOP) SERIES

FREE ONLINE TRAINING

PARALLEL PROGRAMMING AND OPTIMIZATION

FOR INTEL® ARCHITECTURE

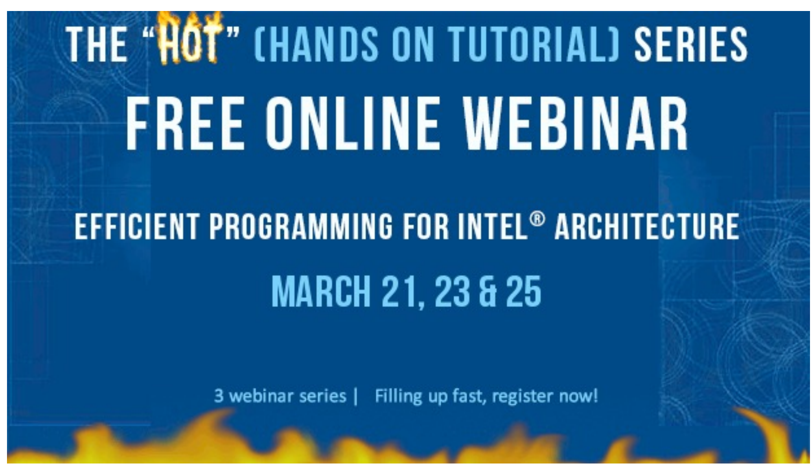
STARTS MARCH 7

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

colfaxresearch.com/how-series/

© Colfax International, 2013–2016

HOT Series



colfaxresearch.com/hot-16-03/

Intel Resources

1. Intel Xeon Phi Product Family: http://intel.com/xeonphi
2. Developer Portal: https://software.intel.com/mic-developer
3. Modern Code Program: https://software.intel.com/en-us/modern-code

© Colfax International, 2013–2016

§4. Hardware Orientation

 © Colfax International, 2013–2016

Intel Architecture Products

 © Colfax International, 2013–2016

Computing Applications

Just some examples

Computing

Finance
- Risk analysis
- Fraud prevention
- Real-time hedging

Internet
- Natural speech interaction
- Video streaming and archiving
- Facial recognition

Natural sciences
- Drug development
- Space weather
- New particle discovery

Medicine
- Medical imaging
- Radiation treatment simulation
- Human error detection

Aerospace
- Rocket engine simulation
- Radiation protection studies
- Aircraft aerodynamics

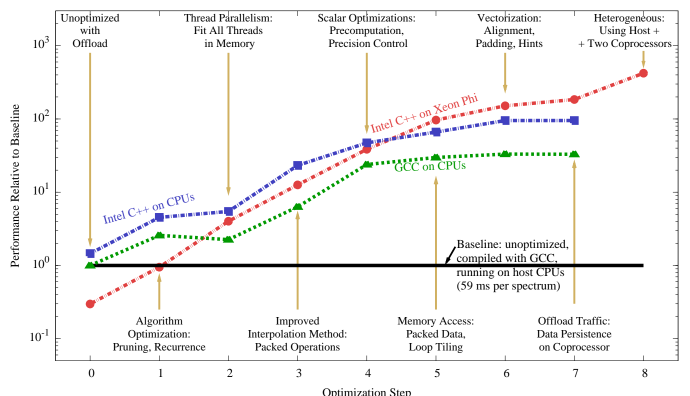
Energy
- Oil reservoir simulation
- Climate change science
- Nuclear fusion research

Manufacturing
- Simulation of material processing
- Development of new materials
- Robotic operations

Environment
- Weather prediction
- Earthquake simulation
- Pollutant propagation

Programming Model Continuity

Common story for many applications:



(see http://xeonphi.com/papers/heatcode)

© Colfax International, 2013–2016

Computing Platforms

Intel Xeon
Processor

Current: Haswell
Upcoming: Broadwell

Intel Xeon Phi
Corocessor, 1st generation

intel
Xeon Phi™ Coprocessor

Current: Knights Corner (KNC)

Intel Xeon Phi
Processor, 2nd generation*

* socket and coprocessor versions

Upcoming: Knights Landing (KNL)

Multi-Core Architecture

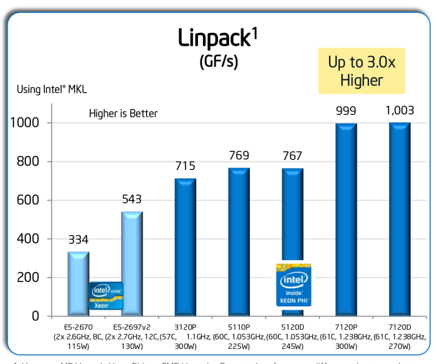
Intel Many Integrated Core (MIC) Architecture

 | © Colfax International, 2013–2016

Intel Architecture Product Performance

Many-core Coprocessors (Xeon Phi) vs Multi-core Processors (Xeon) —

- Better performance per system & performance per watt for parallel applications
- Same programming methods, same development tools.

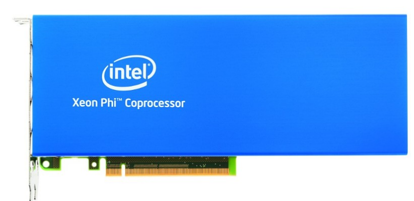


1. Xeon ran MP Linpack, Xeon Phi ran SMP Linpack. Expected performance difference between the two is estimated in the 3-5% range

Source: "Intel Xeon Product Family: Performance Brief"

Bird's Eye View

- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GiB of DDR3 RAM
- ≤18 cores/chip ≈3 GHz
- 2 hyper-threads per core
- 256-bit AVX vectors



- C/C++/Fortran; OpenMP/MPI
- Special Linux distribution
- 3–16 GiB cached GDDR5 RAM
- Up to 61 cores at ≈1.2 GHz
- 4 hardware threads per core
- 512-bit IMCI vectors

 © Colfax International, 2013–2016

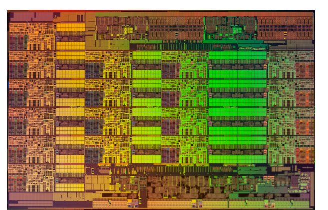
Intel Xeon E5 Series Processors

 © Colfax International, 2013–2016

Intel Xeon CPU: Purpose and Specifications

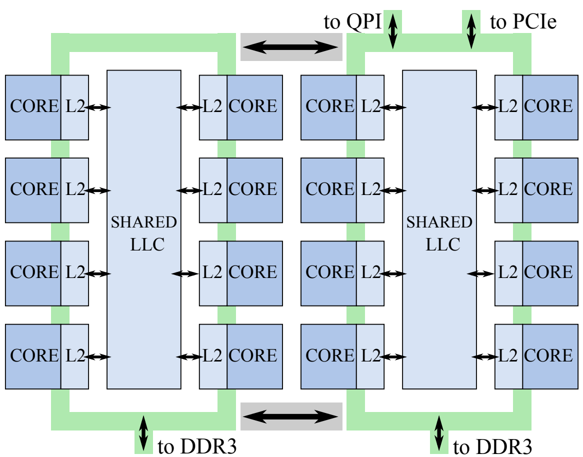
General-purpose platform for demanding computing applications.

- Up to ∼ 1 TFLOP/s in DP
- Up to ∼ 2 TFLOP/s in SP
- Up to 768 GiB DDR3 RAM
- ∼ 126 GB/s bandwidth
- Hardware-rich: forgiving of sub-optimal code



Intel Xeon CPU: Die Organization

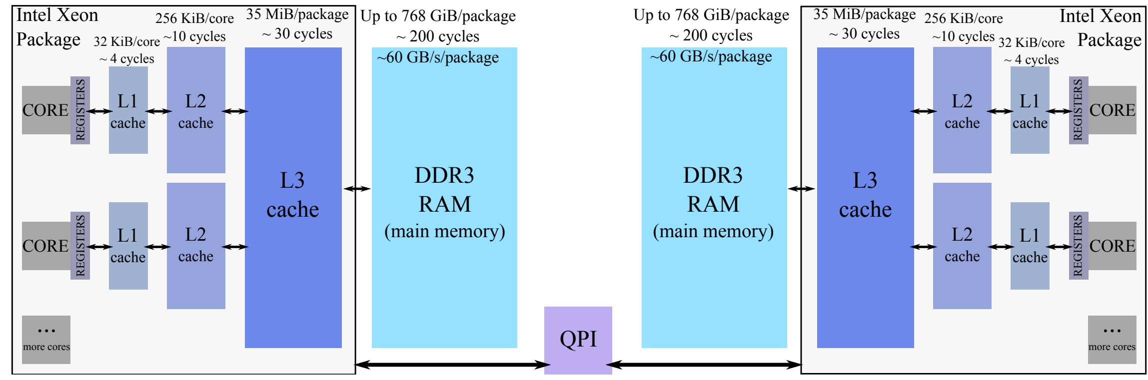
Likes data locality, but large LLC is forgiving.



© Colfax International, 2013–2016

Intel Xeon CPU: Memory Organization

- Hierarchical cache structure
- Two-way processors have NUMA architecture



 © Colfax International, 2013–2016

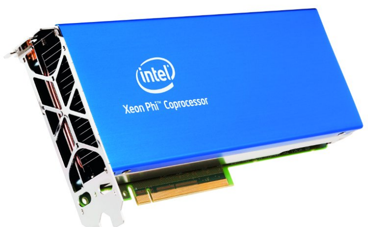
1st Generation Intel Xeon Phi Processors (KNC)

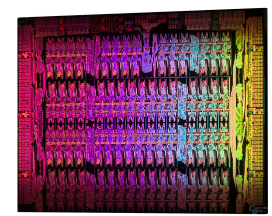
© Colfax International, 2013–2016

Intel Xeon Phi Processors (1st Gen)

Specialized platform for demanding computing applications.

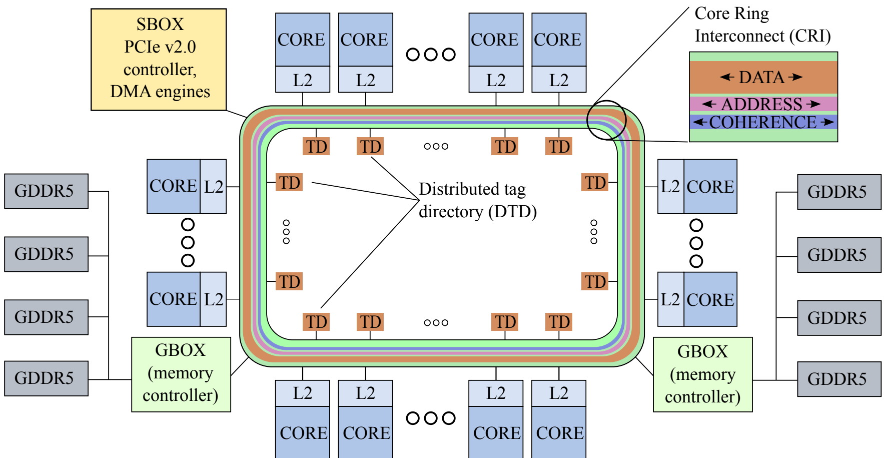
- PCIe end-point device
- $\sim$ 1.2 TFLOP/s in DP
- $\sim$ 2.4 TFLOP/s in SP
- Up to 16 GiB GDDR5 RAM
- $\sim$ 176 GB/s bandwidth
- Heterogeneous clustering
- Runs special Linux distribution





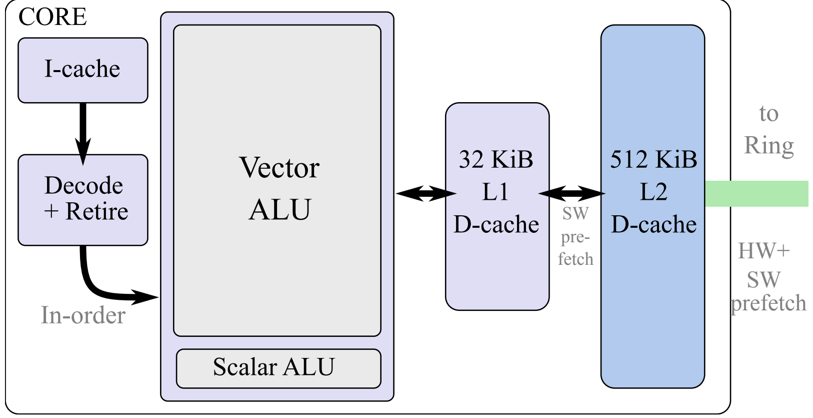
KNC Die Organization

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



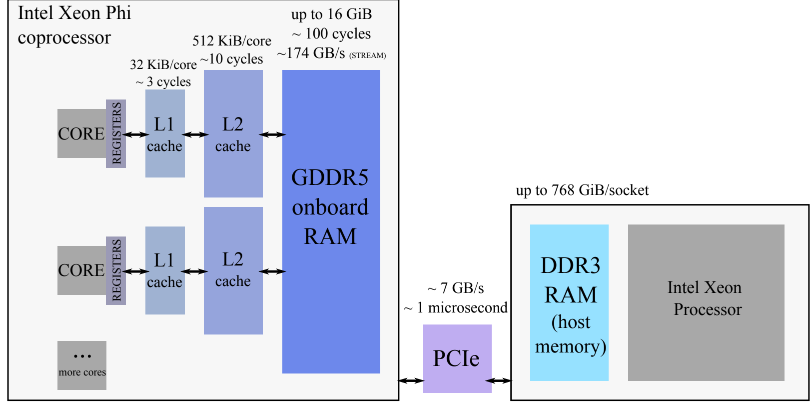
KNC Cores

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



KNC Memory Organization

- Direct access to ≤ 16 GiB of cached GDDR5 memory on board
- No access to system DDR3, connected to host via PCIe

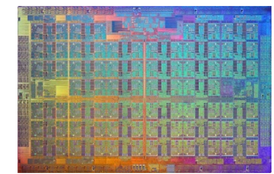


2nd Generation Intel Xeon Phi Processors (KNL)

Intel Xeon Phi Processors (2nd Gen)

Specialized platform for demanding computing applications.

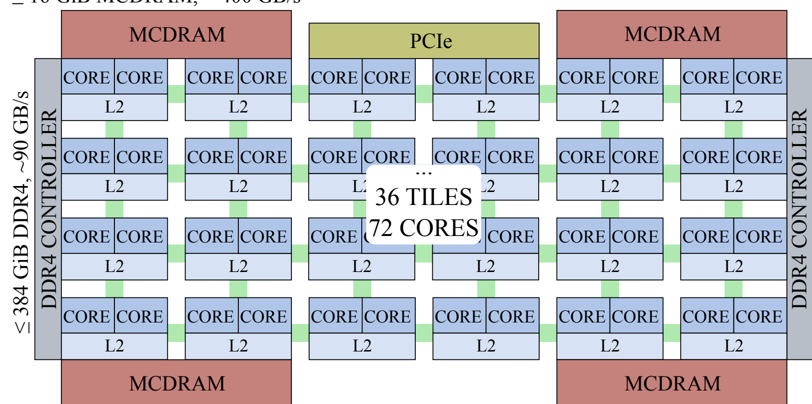
- Socket version or coprocessor
- 3+ TFLOP/s in DP
- 6+ TFLOP/s in SP
- Up to 16 GiB MCDRAM
- ~ 400 GB/s MCDRAM bandwidth
- Up to 384 GiB DDR4 RAM
- ~ 90 GB/s DDR4 bandwidth
- Supports common OS
- Public disclosures



© Colfax International, 2013–2016

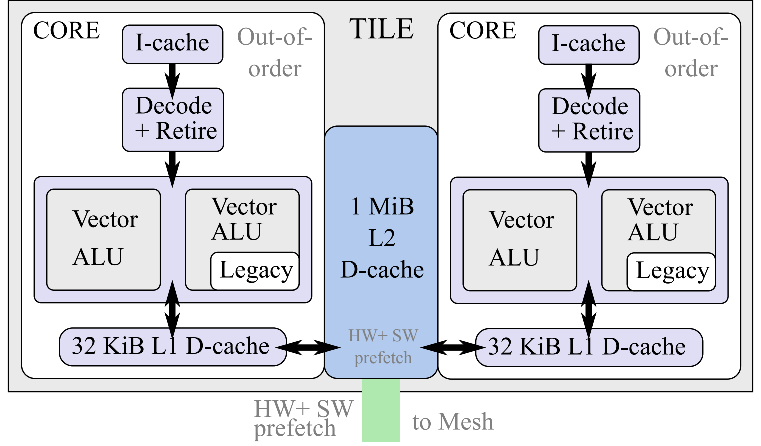
KNL Die Organization

- Mesh interconnect relaxes data locality requirement [somewhat]
- All-to-all, quadrant or sub-numa domain communication in mesh



KNL Cores

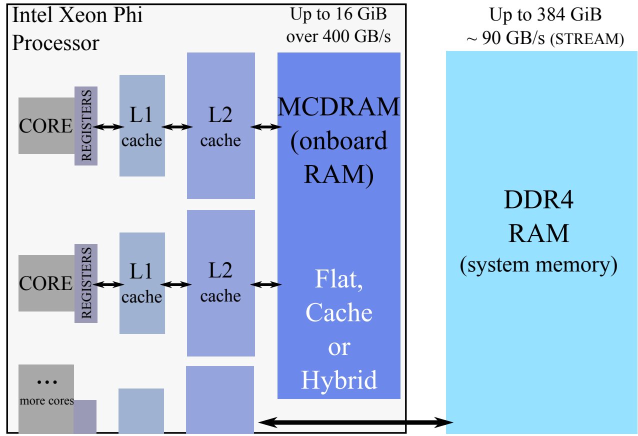
- Even more power in vector units
- Binary compatible with Xeon, but in legacy mode



 © Colfax International, 2013–2016

KNL Memory Organization

- Direct access to onboard MCDRAM *and* system DDR4
- Use MCDRAM as cache, in flat mode, or as hybrid



What's New in Programming for KNL

 © Colfax International, 2013–2016

KNL: What's New

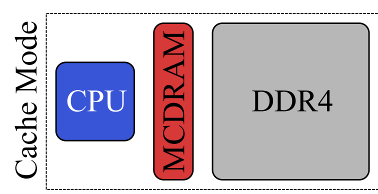
	Haswell	**KNC**	**KNL**
Form factor	Socket	PCIe	PCIe, Socket
Core	Out-of-order	In-order	Out of order
Vectors	≤ AVX2	IMCI	≤ AVX2 + AVX-512
Memory	DDR3	GDDR5	MCDRAM+DDR4
RDMA	Over PCIe	Over PCIe	PCIe or Integrated
Programming	Traditional	Native, Offload	Traditional, Offload
Performance, Optimization	High and forgiving	Higher; needs good code	3x KNC 1-threaded and parallel

 © Colfax International, 2013–2016

Using High-Bandwidth Memory (MCDRAM)

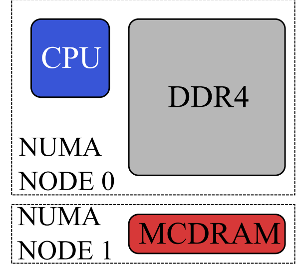
Option 1 : cache/hybrid mode

- Treat it as LLC
- Data locality techniques
- Miss latency 2x the direct DDR4 access



Option 2 : flat mode

- Application fits in 16 GiB? `numactl`
- More than 16 GiB data? Use special allocators (e.g., `memkind`)



 © Colfax International, 2013–2016

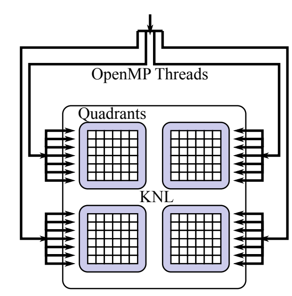
Using Cluster Modes

Option 1 : all-to-all

- Scale to 72×4 threads
- Data locality, enough parallelism

Option 2 : quadrants, domains

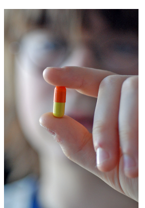
- Use nested parallelism in OpenMP
- Use hybrid MPI+OpenMP to pin processes to quadrants




```
1  #pragma omp parallel
2  {
3  #pragma omp parallel
4    {
5      // ...
6    }
7  }
```

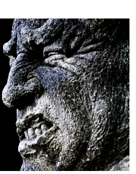
Using New Vector Features

Efficient gather/scatter, conflict detection, high-precision exponential and reciprocal, two vector units per core...



Option 1 : automatic vectorization

- No changes to code, possibly adjust to 512-bit vector width
- Recompile with `-xMIC-AVX512`



Option 2 : explicit vectorization

- Bite the bullet

See also this post

© Colfax International, 2013–2016

Bottom Line

The best way to prepare...

 © Colfax International, 2013–2016

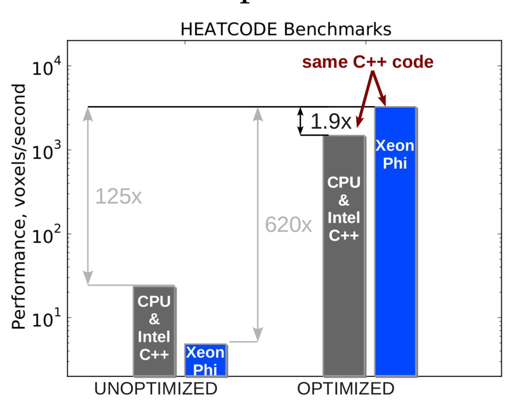
Performance

© Colfax International, 2013–2016

Will My Code Run Faster on KNL?

Performance on MIC architecture is a function of optimization Level

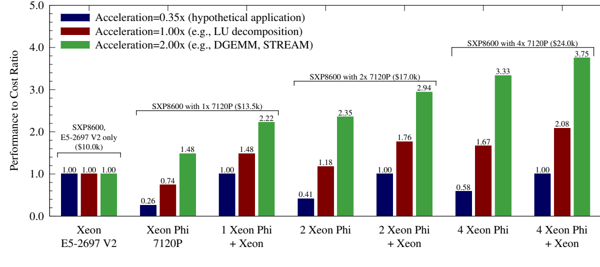
- Performance will be disappointing if code is not optimized for multi-core CPUs
- Optimized code runs better on the MIC platform *and* on the multi-core CPU
- Single code for two platforms + Ease of porting = Incremental optimization

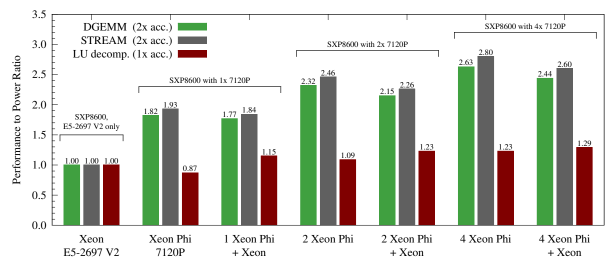


See this case study

What is Your Performance Metric?

- Performance per System
- Performance per Watt
- Performance/Cost Ratio

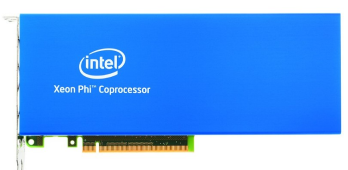




See this paper for details

 © Colfax International, 2013–2016

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"

§5. Programming Coprocessors

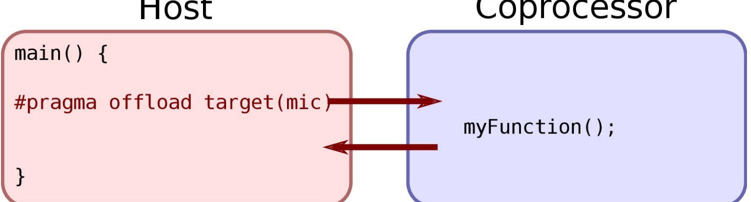
 © Colfax International, 2013–2016

Programming Options

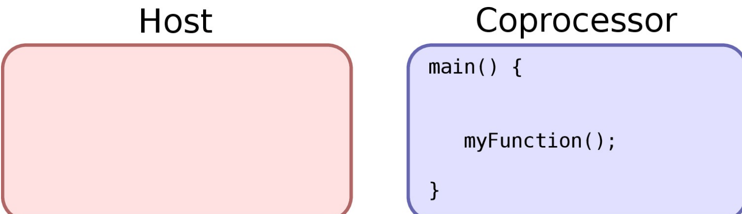
 © Colfax International, 2013–2016

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

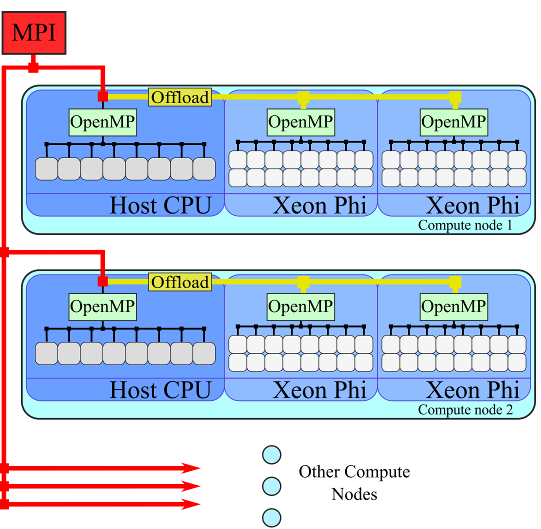
Multi-Core Hosted	Offload	Symmetric	Many Core Hosted
General serial and parallel computing	Code with highly-parallel phases	Codes with balanced needs	Highly-parallel codes

© Colfax International, 2013–2016

Heterogeneous Distributed Computing with Xeon Phi

Option 1: MPI+OpenMP with Offload.

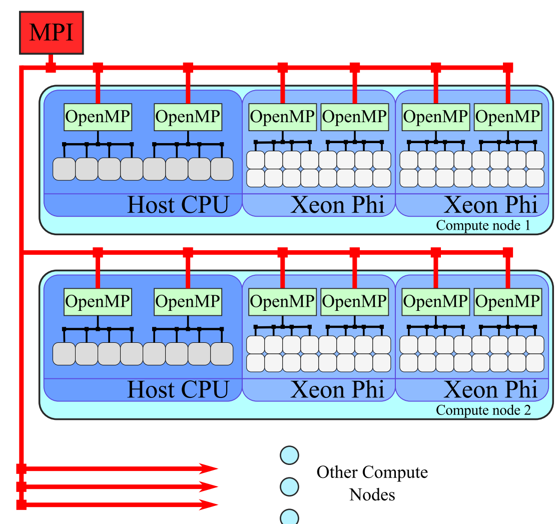
- MPI processes are multi-threaded with OpenMP.
- MPI runs only on CPUs.
- MPI processes offload to coprocessor(s).
- OpenMP in offload regions.



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.



Lab Time

Lab Time

Perform exercises in:

- `labs/2/2.01-native-basic` (recommended)
- `labs/2/2.03-offload-basic` (recommended)
- `labs/2/2.05-shared-virtual-memory-basic` (optional)
- `labs/2/2.07-benchmark-offload` (optional)

The rest of this section may help you with the exercises.
Lecture will resume in Section 12.

 © Colfax International, 2013–2016

Native Coprocessor Applications

© Colfax International, 2013–2016

Linux Environment on Intel Xeon Phi Coprocessors

```
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, 2013–2016

Native Execution

"Hello World" application:

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

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, 2013–2016

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)

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`
- Knights Landing: `-xMIC-AVX512`
- 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, 2013–2016

Explicit Offload

 © Colfax International, 2013–2016

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.

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

Offloading Functions

```
1  __attribute__((target(mic))) void MyFunction() {
2      // ... implement function as usual
3  }
4
5  int main(int argc, char * argv[] ) {
6  #pragma offload target(mic)
7      {
8          MyFunction();
9      }
10 }
```

- 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, 2013–2016

Offloading Multiple Functions

```
#pragma offload_attribute(push, target(mic))
void MyFunctionOne() {
// ... implement function as usual
}

void MyFunctionTwo() {
// ... implement function as usual
}
#pragma offload_attribute(pop)
```

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

 © Colfax International, 2013–2016

Offloading Data: Local Scalars and Arrays

```
void MyFunction() {
    const int N = 1000;
    int data[N];
#pragma offload target(mic)
    {
        for (int i = 0; i < N; i++)
          data[i] = 0;
    }
```

- 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

```
1  int* __attribute__((target(mic))) data;
2
3  void MyFunction() {
4      static int __attribute__((target(mic))) N;
5      // ...
6  }
7
8  int main() {
9      // ...
10 }
```

- 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

```
double *p1=(double*)malloc(sizeof(double)*N);
double *p2=(double*)malloc(sizeof(double)*N);

#pragma offload target(mic) in(p1 : length(N)) out(p2 : length(N))
{
    // ... perform operations on p1[] and p2[]
}
```

- `#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, 2013–2016

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, 2013–2016

Multiple Coprocessors with Explicit Offload

© Colfax International, 2013–2016

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);
}
```

Multiple Blocking Offloads Using Host Threads (Explicit Offload)

```
const int nDevices = _Offload_number_of_devices();
#pragma omp parallel num_threads(nDevices)
  {
    const int i = omp_get_thread_num();
#pragma offload target(mic: i)
      {
        MyFunction(/*...*/);
      }
  }
}
```

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

Blocking Explicit Offloads Using Threads: Dynamic Work Distribution Across Coprocessors

```
1  const int nDevices = _Offload_number_of_devices();
2  omp_set_num_threads(nDevices);
3  #pragma omp parallel for schedule(dynamic, 1)
4    for (int i = 0; i < nWorkItems; i++) {
5      const int iDevice = omp_get_thread_num();
6  #pragma offload target(mic: iDevice)
7      {
8        MyFunction(i);
9      }
10   }
```

- Up to 8 coprocessors, up to 32 host threads
- `nWorkItems` are dynamically scheduled on `nDevices`

 © Colfax International, 2013–2016

Memory Allocation Control

© Colfax International, 2013–2016

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 ≈ 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 */ }
```

Offload Latency With and Without Memory/Data Retention

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.

```
#pragma offload target(mic:0) in(p : length(N) alloc_if(1) free_if(0) )
{ /* 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.

Overlapping Communication and Computation

 © Colfax International, 2013–2016

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

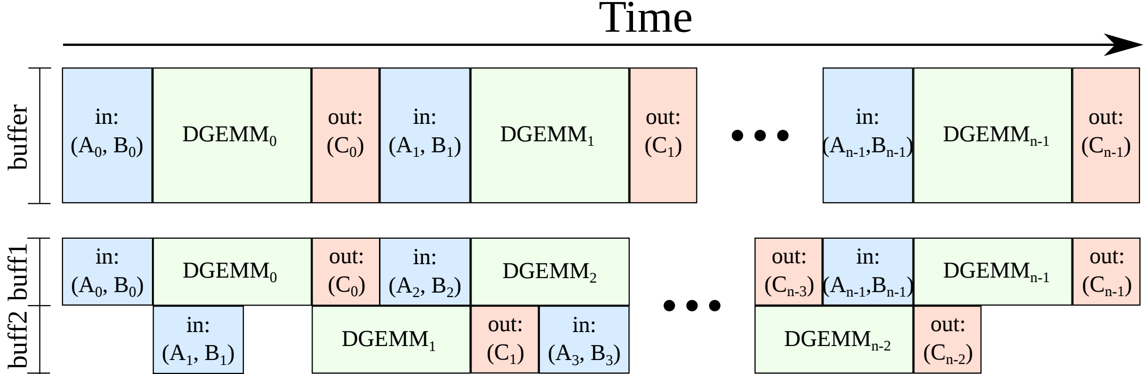
```
float* offload0 = &data[0]; // Any unique pointer value as signal
#pragma offload target(mic:0) signal(offload0) in(data : length(N))
{ /* ... will not block code execution because of clause "signal" */ }

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

 © Colfax International, 2013–2016

Overlapping Communication and Computation

Usage example: double buffering to mask communication latency



 © Colfax International, 2013–2016

Double Buffering with Asynchronous Offload

```
1  for(int i = 1; i < nMatrices-1; i++) {
2    double* A_T = MatrixA[i]; // Dataset to send next; ...same for B and C
3
4  #pragma offload target(mic:0)  signal(A_buff_C)  \
5    in(A_buff_C: length(0) alloc_if(0) free_if(0)) ...(same for B and C)
6    { cblas_dgemm(..., A_buff_C, ...); } // Asynchronous offload (COMPUTATION)
7
8  // Send next data set, retrieve previous results (COMMUNICATION):
9  #pragma offload_transfer target(mic:0) in(A_T[0:n*n]: into (A_buff_T[0:n*n]))...
10 #pragma offload_transfer target(mic:0) out(C_buff_T[0:n*n]: into (C_T[0:n*n]))
11 // Wait for asynchronous offload (SYNCHRONIZATION):
12 #pragma offload_wait target(mic:0) wait(A_buff_C)
13
14 if(i%2==1) // Swap Buffers
15   { A_buff_T=A_buff2; A_buff_C =A_buff1; /* ...same for B and C */ }
16 else
17   { A_buff_T=A_buff1; A_buff_C =A_buff2; /* ...same for B and C */ }
```

Additional Offload Controls

 © Colfax International, 2013–2016

Target-Specific Code

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

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

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

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.
```

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=48
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=240
```

 © Colfax International, 2013–2016

Offload in OpenMP 4.0

OpenMP 4.0 Target Offload

- Another API for offload: `#pragma omp target`
- Part of the OpenMP 4.0 standard
- Designed as portable solution (coprocessors, GPGPUs)
- On Xeon Phi, uses the same back-end as `#pragma offload`

```
#pragma omp target
  {
#pragma omp parallel for
    for(int i=0; i<size; i++)
      data[i] = 0;
  }
```

Application runs on the host, but some parts of code and data are moved ("offloaded") the coprocessor. Scope-local scalars and stack arrays offloaded automatically.

Clauses of pragma omp target

```
1 #pragma omp target [clause[, clause[, ...]]
```

- `device(int)` – offload to a specific device (coprocessor)
- `map([type:] variables)` – create data environment. `type` is `to`, `from`, `tofrom` or `alloc`
- `if(expr)` – optional offload

Link to reference manual.

OpenMP 4.0 Target Data Mapping

Use `#pragma omp target data` to create a device data environment. This allows to keep persistent data on coprocessor. Example:

```
#pragma omp target data map(from:data)
  {
#pragma omp target
#pragma omp parallel for
    for(int i=0; i<size; i++) data[i] = 0;

#pragma omp target
#pragma omp parallel for
    for(int i=0; i<size; i++) data[i] += 1;
  }
```

`data` array copied back from coprocessor only once at the end.
Link to reference manual.

Movement of Persistent Data

Use `#pragma omp target update` to force data movement within the data environment. Example:

```
#pragma omp target data map(from:data)
{
#pragma omp target
  { ... }

#pragma omp target update from(data)

#pragma omp target
  { ... }
}
```

`data` array copied from coprocessor between offloads, and at the end.

Link to reference manual.

 © Colfax International, 2013–2016

Offloading functions wtih #pragma omp target

Use #pragma omp declare target on functions that may be offloaded (similar to `__attribute__((target(mic)))`). Example:

```
#pragma omp declare target
void myinit(int* data, int size){
#pragma omp parallel for
  for(int i=0; i<size; i++) data[i] = 0;
}
#pragma omp end declare target

int main(int argv, char** argc){
...
#pragma omp target map(tofrom:data) map(to:size)
    myinit(data, size);
}
```

Link to reference manual.

 © Colfax International, 2013–2016

`#pragma offload target` vs. `#pragma omp target`

1. Different interfaces to the same offload library back-end
2. `#pragma offload target` is Intel-specific, `#pragma omp target` is part of a cross-platform standard (although, as of 2015, cross-platform support is not widespread).
3. `#pragma offload target` allows data/memory persistence outside of the scope of a pragma, `#pragma omp target` – only within the lexically structured scope (may change in Parallel Studio 2016).
4. `#pragma offload` is a more flexible model and will continue to be supported (see Intel's communication).

Additional information: webinar.

Shared Virtual Memory Offload Model

 © Colfax International, 2013–2016

Shared Virtual Memory Model

```
1  _Cilk_shared int arr[N]; // This is a virtual-shared array
2
3  _Cilk_shared void Compute() { // This function may be offloaded
4      // ... function uses array arr[]
5  }
6
7  int main() {
8      // arr[] can be initialized on the host
9      _Cilk_offload Compute(); // and used on coprocessor
10     // and the values are returned to the host
11 }
```

- 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, 2013–2016

Shared Virtual 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);
}
```

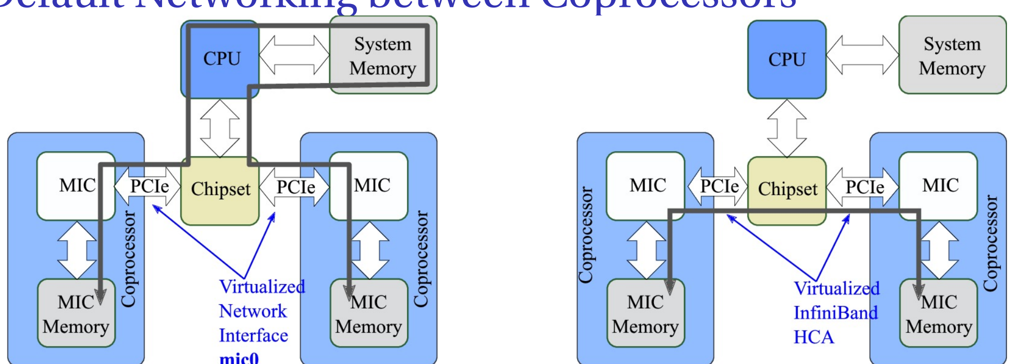
- Addresses of virtual-shared pointers identical on host and coprocessors
- Synchronized before and after offload, with page granularity

© Colfax International, 2013–2016

Networking with Coprocessors

 © Colfax International, 2013–2016

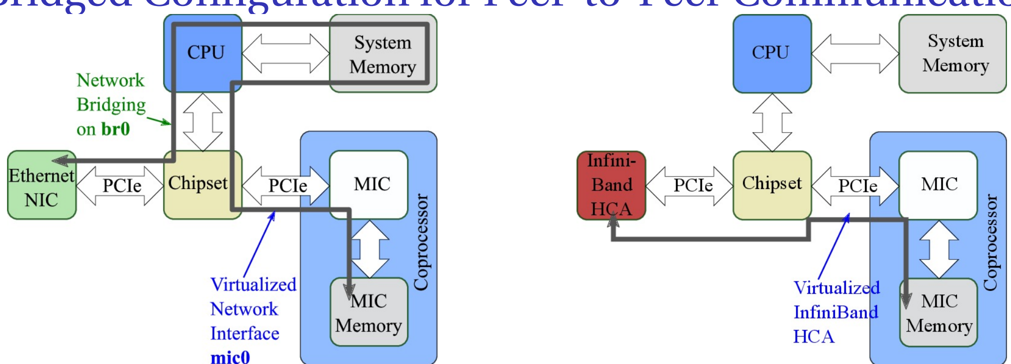
Default Networking between Coprocessors



- Left: by default, TCP/IP communication between cooprocessors within a system is enabled.
- Right: adding OFED enables DMA within a system over a virtual InfiniBand interface “ibscif”. See this paper.

© Colfax International, 2013–2016

Bridged Configuration for Peer-to-Peer Communication



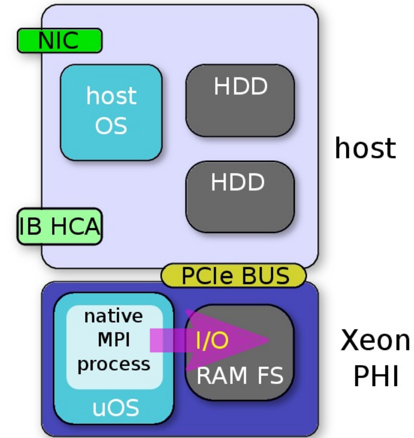
- Left: Gigabit Ethernet bridging on host allows to place coprocessors on the same subnet as hosts
- Right: Coprocessor Communication Link (CCL) – virtualization of an InfiniBand device on each coprocessor. See this paper.

Filesystem Access from Coprocessors

 © Colfax International, 2013–2016

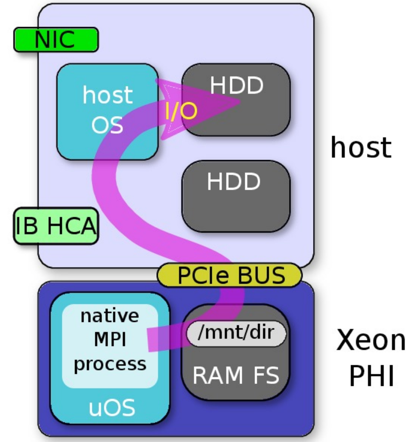
RAM Filesystem

- Files are stored in the coprocessor RAM
- Does not survive MPSS restart or host reboot
- Fastest method
- Good for local pre-staged input or runtime scratch data



Virtio Transfer to Local Host Drives

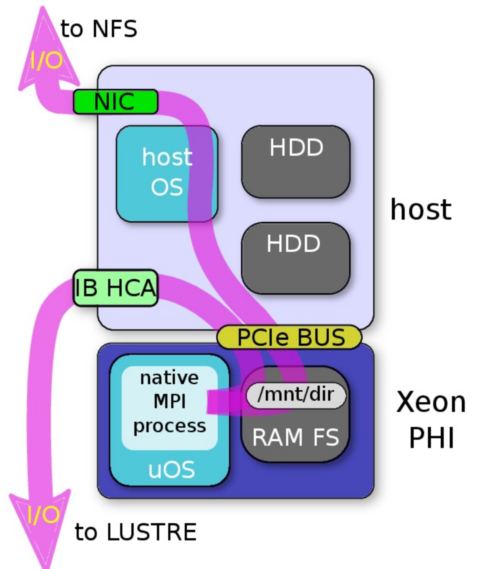
- Files are stored on a physical or virtual drive on the host
- Written data persistent across reboots
- Fast method
- Cannot share a drive between coprocessors
- Good for distributed checkpointing



© Colfax International, 2013–2016

Network Storage

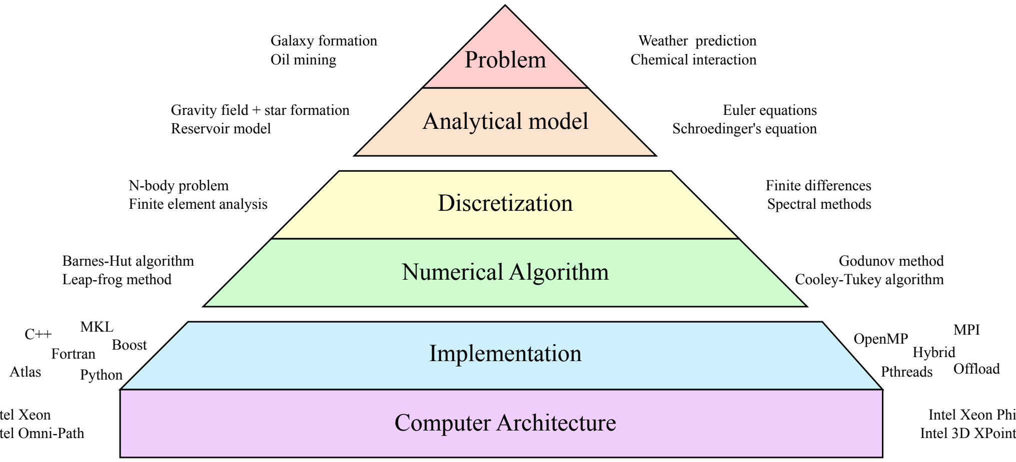
- Files are stored on a remote file server
- Can share a mount point across the cluster
- Lustre has scalable performance
- NFS is slow but easy to set up
- Details: http://xeonphi.com/papers/io



§6. Performance Fundamentals

© Colfax International, 2013–2016

Computing in Science and Engineering



 © Colfax International, 2013–2016

Scope of This Course

Areas of code optimization for Intel architecture:

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, 2013–2016

§7. Scalar Tuning

© Colfax International, 2013–2016

Compiler Arguments

 © Colfax International, 2013–2016

Optimization Level

Default optimization level `-O2`

- optimization for speed
- automatic vectorization
- inlining
- constant propagation
- dead-code elimination
- loop unrolling

Optimization level `-O3`

- aggressive optimization
- loop fusion
- block-unroll-and-jam
- `if`-statement collapse
- *may or may not be better than* `-O2`

© Colfax International, 2013–2016

Setting Optimization Level

For the entire file:

```
vega@lyra% icc -o mycode -O3 source.c
```

For a specific function:

```
1 #pragma intel optimization_level 3
2 void my_function() {
3   //...
4 }
```

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

© Colfax International, 2013–2016

Floating-Point Semantics

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

`-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.

Programming Practices

© Colfax International, 2013–2016

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, 2013–2016

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;
```

 © Colfax International, 2013–2016

Consistency of Precision: Functions

```c
// Bad: 3.14 is a double
float x = 3.14;

// Bad: sin() is a
// double precision function
float s = sin(x)

// Bad: round() takes double
// and returns double
long v = round(x);

// Bad: abs() is not from IML
// it takes int and returns int
int v = abs(x);
```

```c
// Good: 3.14f is a float
float x = 3.14f;

// Good: sin() is a
// single precision function
float s = sinf(x)

// Good: lroundf() takes float
// and returns long
long v = lroundf(x);

// Good: fabsf() is from IML
// It takes and returns a float
float v = fabsf(x);
```


© Colfax International, 2013–2016

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)=2.718281828459045l
exp (1.0 )=2.718281828459045l
Exact:    e=2.7182818284590452353 6...

Proof that expf() gives lower precision:
expf(1.0f)=2.718281745910644 5
expf(1.0 )=2.718281745910644 5
Exact:    e=2.7182818284590452353 6...

Overloading in namespace std:
std::exp(1.0f)=2.718281745910644 5
std::exp(1.0 )=2.718281828459045l
Exact:        e=2.7182818284590452353 6...
```

§8. Vectorization

 © Colfax International, 2013–2016

SIMD Instructions

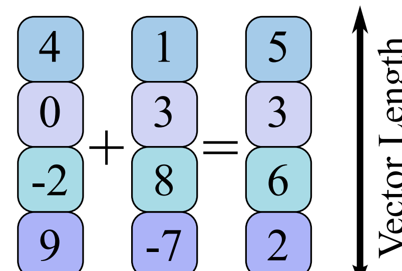
Short Vector Support

Vectors – one of forms of SIMD architecture (Single Instruction Multiple Data).

Scalar Instructions

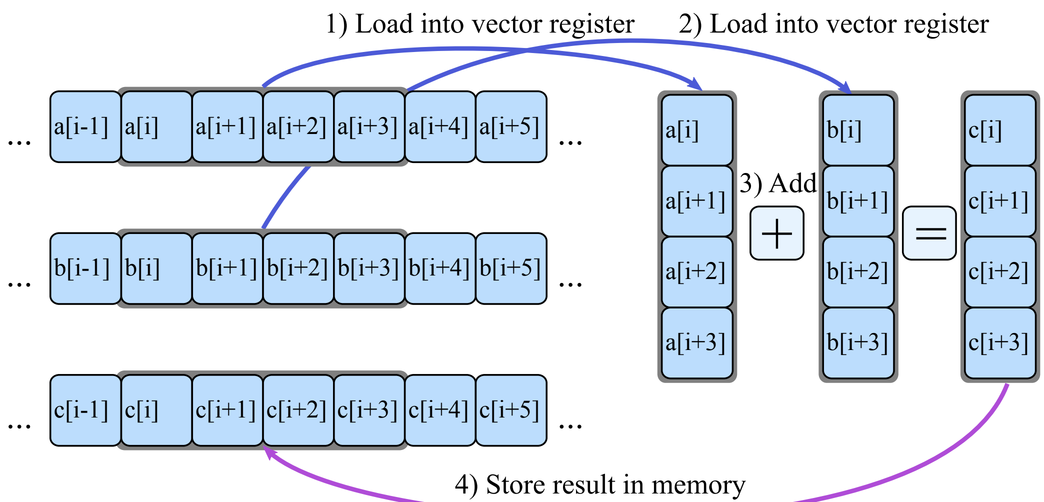
4 + 1 = 5
0 + 3 = 3
-2 + 8 = 6
9 + -7 = 2

Vector Instructions



 © Colfax International, 2013–2016

Workflow of Vector Computation



© Colfax International, 2013–2016

Instruction Sets in Intel Architectures

Instruction Set	Year and Intel Processor	Vector registers	Packed Data Types
MMX	1997, Pentium	64-bit	8-, 16- and 32-bit integers
SSE	1999, Pentium III	128-bit	32-bit single precision FP
SSE2	2001, Pentium 4	128-bit	8 to 64-bit integers; SP & DP FP
SSE3–SSE4.2	2004 – 2009	128-bit	(additional instructions)
AVX	2011, Sandy Bridge	256-bit	single and double precision FP
AVX2	2013, Haswell	256-bit	integers, additional instructions
IMCI	2012, Knights Corner	512-bit	32- and 64-bit integers; single & double precision FP
AVX-512	Knights Landing	512-bit	32- and 64-bit integers; single & double precision FP

© Colfax International, 2013–2016

Lab Time

© Colfax International, 2013–2016

Lab Time

Instructor probably demonstrated this lab live:

- `labs/4/4.04-threading-misc-histogram` (only steps 1, 2)

You can repeat steps in this lab and/or perform exercises in:

- `labs/3/3.01-vectorization` (recommended)
- `labs/4/4.02-vectorization-data-structures-coulomb` (optional)
- `labs/4/4.03-vectorization-tuning-lu-decomposition` (recommended)

The rest of this section may help you with the exercises.
Lecture will resume in Section 15.

Automatic Vectorization: Loops

 © Colfax International, 2013–2016

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
```

 © Colfax International, 2013–2016

What Can Be Automatically Vectorized

Limitations:

- Only `for`-loops can be auto-vectorized. Number of iterations must be known at a runtime and/or compilation time
- By default, compiler targets the innermost loop for vectorization
- Memory access in the loop must have regular pattern, ideally with unit stride

 © Colfax International, 2013–2016

What Cannot be Automatically Vectorized

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.

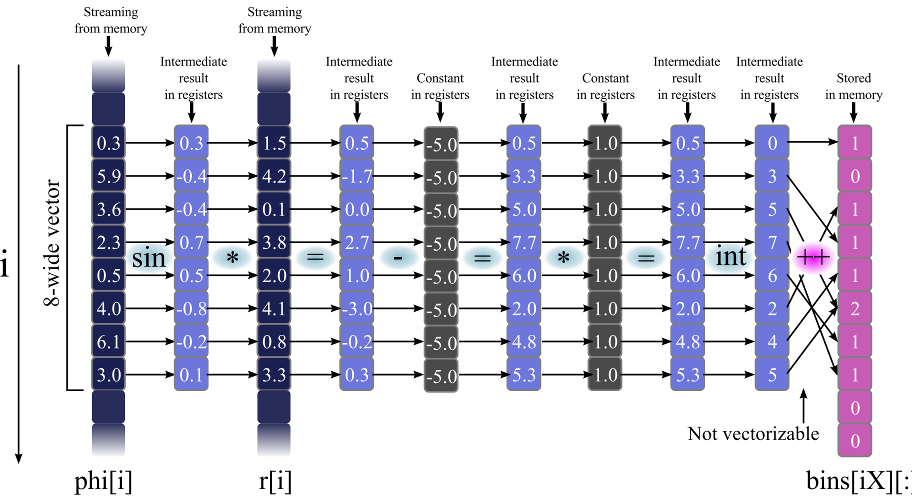
© Colfax International, 2013–2016

Auto-Vectorized Loops May Be Complex (Example 1)

```
for (int i = ii; i < ii + tileSize; i++) { // Target for auto-vectorization

  // Newton's law of universal gravity
  const float dx = particle.x[j] - particle.x[i]; // x[j] is a const
  const float dy = particle.y[j] - particle.y[i]; // x[i] makes SIMD vector
  const float dz = particle.z[j] - particle.z[i];
  const float rr = 1.0f/sqrtf(dx*dx + dy*dy + dz*dz + softening);
  const float drPowerN32  = rr*rr*rr;

  // Calculate the net force
  Fx[i-ii] += dx * drPowerN32;
  Fy[i-ii] += dy * drPowerN32;
  Fz[i-ii] += dz * drPowerN32;
}
```

See also this presentation

Auto-Vectorized Loops May Be Complex (Example 2)



See this paper for more details

Pragma simd

Vectorize more loops: `#pragma simd`

Statement `#pragma simd` is used to “enforce vectorization of loops”, which includes:

- Loops with SIMD-enabled functions (see below)
- Second innermost loops
- Failed vectorization due to compiler decision
- Loops where guidance is required (vector length, reduction, etc.)

See compiler reference on `#pragma simd` for more information.

© Colfax International, 2013–2016

Example for #pragma simd

```
const int N=128;
const int T=4;
float A[N*N], B[N*N], C[T*T];

for (int jj = 0; jj < N; jj+=T) // Tile in j
  for (int ii = 0; ii < N; ii+=T) // and tile in i
  // Using pragma simd to vectorize outer loop:
#pragma simd
  for (int k = 0; k < N; ++k) // long loop, vectorize it
    for (int i = 0; i < T; i++) { // Loop between ii and ii+T
      // Instead of a loop between jj and jj+T, unrolling that loop:
      C[0*T + i] += A[(jj+0)*N + k]*B[(ii+i)*N + k];
      C[1*T + i] += A[(jj+1)*N + k]*B[(ii+i)*N + k];
      C[2*T + i] += A[(jj+2)*N + k]*B[(ii+i)*N + k];
      C[3*T + i] += A[(jj+3)*N + k]*B[(ii+i)*N + k];
}
```

Loop Was Vectorized, Now What?

1. Ensure unit stride access
2. Align data
3. Pad multi-dimensional containers
4. Eliminate peel loops
5. Eliminate multiversioning
6. **Optimize data re-use in caches**

Good to Know

Vector FLOPs are cheap compared to memory access.

If your data is served by RAM and not caches, it does not matter if you have vectorization: you will be bottlenecked by memory access.

 © Colfax International, 2013–2016

Array Notation and SIMD-enabled Functions

© Colfax International, 2013–2016

Extensions for Array Notation

Array notation is a method for specifying

- slices of arrays (begin, length)

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

- a stride (begin, length, stride)

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

- Multi-dimensional arrays

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

Better than strided loops (e.g., this paper).

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, 2013–2016

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.

Pointer Disambiguation

© Colfax International, 2013–2016

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, 2013–2016

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
```

Multiversioning

```
user@host% icpc -c code.cc -qopt-report -qopt-report-phase:vec
user@host% cat code.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.

 © Colfax International, 2013–2016

Pointer Disambiguation to Prevent Multiversioning

Prevent multiversioning by using `#pragma ivdep`

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

```
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
...
```

When keyword `restrict` is used instead, may not disambiguate different offsets of same pointer (e.g, `A[i*n+j] += A[b*n+j]`).

Data Alignment

© Colfax International, 2013–2016

Data Alignment Requirements

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)

Data Alignment on Stack

- Data alignment on the stack

```
1 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

```
1 float *A = (float*)_mm_malloc(n*n*sizeof(float), 64);
2 // ...
3 _mm_free(A);
```

  - `_mm_malloc` and `_mm_free` are aligned version of `malloc` and `free`

© Colfax International, 2013–2016

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

  - `_mm_malloc` and `_mm_free` are aligned version of `malloc` and `free`

 © Colfax International, 2013–2016

Data Alignment Hints

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

```
1 // Promising that A[i*lda + 0] is aligned for every i
2 // and the same for every other array in this loop
3 #pragma vector aligned
4     for (int j = 0; j < n; j++)
5       A[i*lda + j] -= ...
```

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

© Colfax International, 2013–2016

Padding for Alignment

To use aligned instructions, you may need to pad inner dimension of multi-dimensional arrays to a multiple of 16 (in SP) or 8 (DP) elements.

```
// ... Padding inner dimension so that every row is aligned
int lda=n;
if (n % 16 != 0) lda += (16 - n%16);  // now lda%16==0
float* A = _mm_malloc(sizeof(float)*n*lda, 64);

// If A[0         ] is aligned AND lda%16==0, then
//    A[i*lda + 0] is aligned for any i
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    A[i*lda + j] = ...
```

Unit-Stride Data Access

 © Colfax International, 2013–2016

Unit-Stride Access

Unit-stride access is optimal:

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

Non-unit stride is slower:

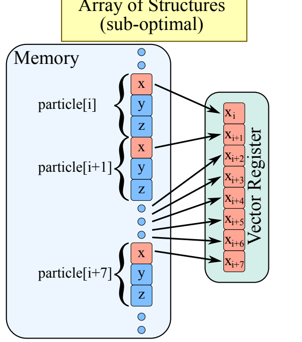
```
for (int i = 0; i < n; i++)
  A[i*stride] += B[i];
```

Stochastic access cannot be vectorized:

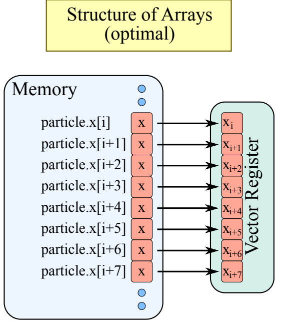
```
for (int i = 0; i < n; i++)
  A[offset[i]] += B[i];
```

It may be a question of changing the order of loop nesting, but sometimes you need to modify data structures:

Array of Structures (sub-optimal)



Structure of Arrays (optimal)



 © Colfax International, 2013–2016

Additional Vectorization “Tuning Knobs”

 © Colfax International, 2013–2016

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]`

Explicit Vectorization

© Colfax International, 2013–2016

Detecting Available Instructions

In the OS:

```
[student@cdt ~]% cat /proc/cpuinfo
...
fpu_exception   : yes
cpuid level     : 11
wp              : yes
flags           : fpu vme de pse tsc msr pae mce
cx8 apic mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ss ht syscall nx lm constant_tsc
unfair_spinlock pni ssse3 cx16 sse4_1 sse4_2
x2apic popcnt aes hypervisor lahf_lm fsgsbase
bogomips        : 5985.17
clflush size    : 64
cache_alignment: 64
address sizes   : 46 bits physical, 48 bits virtual
...
```

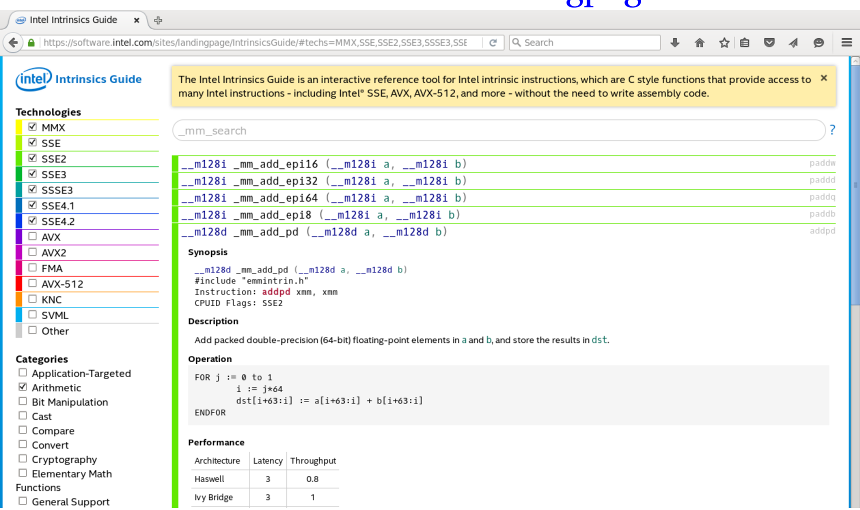
In code (see also):

```
// Intel compiler
// preprocessor macros:

#ifdef __SSE__
// ...SSE code path
#endif

#ifdef __SSE4_2__
// ...SSE code path
#endif

#ifdef __AVX__
// ...AVX code path
#endif
```


© Colfax International, 2013–2016

Intel Intrinsics Guide

https://software.intel.com/sites/landingpage/IntrinsicsGuide



Strip-Mining for Vectorization

© Colfax International, 2013–2016

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
}
```

Example 1: Binning, Strip-Mining

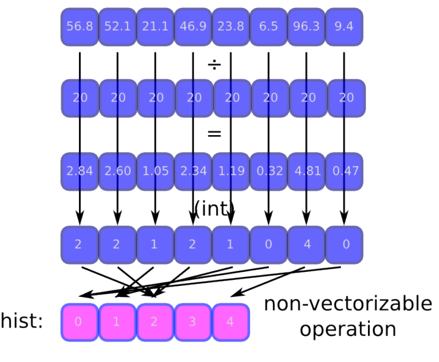
 © Colfax International, 2013–2016

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



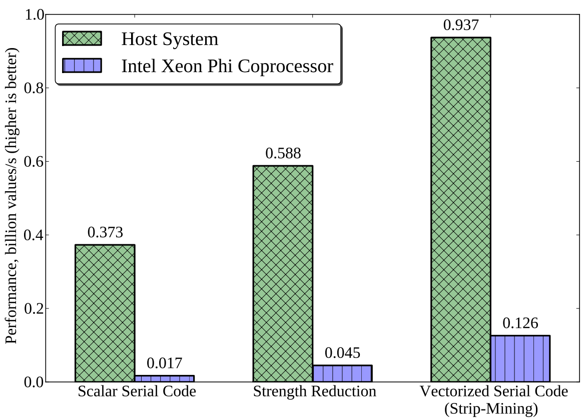
 © Colfax International, 2013–2016

The Same Calculation, Strip-Mined, Vectorized

```
1  void Histogram(const float* age, int* const hist, const int n,
2  const float group_width, const int m) {
3    const int vecLen = 16; // Length of vectorized loop
4    const float invGroupWidth = 1.0f/group_width; // Pre-compute the reciprocal
5    // Strip-mining the loop in order to vectorize the inner short loop
6    // Note: this algorithm assumes n%vecLen == 0.
7    for (int ii = 0; ii < n; ii += vecLen) { //Temporary store vecLen indices
8      int index[vecLen] __attribute__((aligned(64)));
9      // Vectorize the multiplication and rounding
10 #pragma vector aligned
11     for (int i = ii; i < ii + vecLen; i++)
12       index[i-ii] = (int) ( age[i] * invGroupWidth );
13     // Scattered memory access, does not get vectorized
14     for (int c = 0; c < vecLen; c++)
15       hist[index[c]]++;
16   }
17 }
```

Strip-Mining for Vectorization

Vectorization improves performance on both platforms. However, more work is needed to take advantage of the MIC architecture. Section 15 (multi-threading).



 © Colfax International, 2013–2016

 © Colfax International, 2013–2016

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;
#pragma simd
      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: http://xeonphi.com/papers/

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+1) - (b+1)%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`

Pointer-Disambiguation: Multiversioning

```
user@host% icpc -c code.cc -qopt-report -qopt-report-phase:vec
user@host% cat code.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
...
```

When keyword `restrict` is used instead, may not disambiguate different offsets of same pointer (e.g, `A[i*n+j] += A[b*n+j]`).

LU Decomposition: Compiler hints

- Data alignment hint: `#pragma vector aligned`

Before:

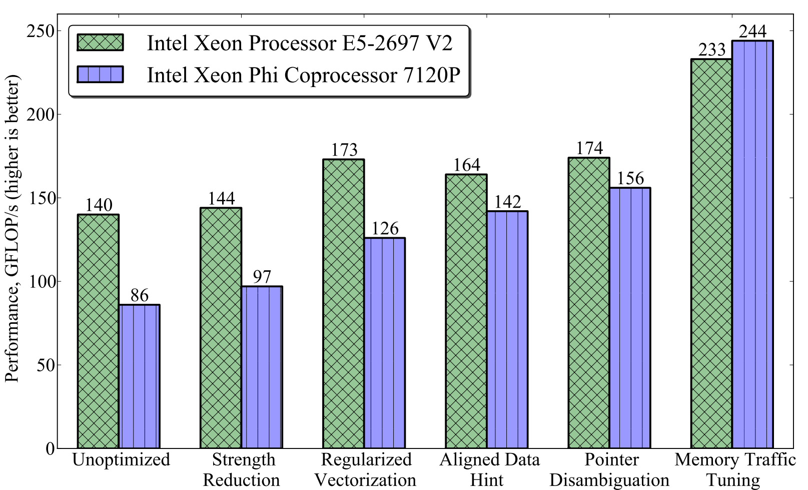
```
1  for (int b = 0; b < n; b++) {
2    const int jMin = b - b%tile;
3    const float recAbb = 1.0f/A[b*n+b];
4    for (int i = b+1; i < n; i++) {
5      L[i*n + b] = A[i*n + b]*recAbb;
6
7
8  #pragma simd
9      for (int j = jMin; j < n; j++)
10       A[i*n+j] -= L[i*n+b]*A[b*n+j];
11   }
12 }
```

After:

```
1  for (int b = 0; b < n; b++) {
2    const int jMin = b - b%tile;
3    const float recAbb = 1.0f/A[b*n+b];
4    for (int i = b+1; i < n; i++) {
5      L[i*n + b] = A[i*n + b]*recAbb;
6  #pragma vector aligned
7  #pragma ivdep
8  #pragma simd
9      for (int j = jMin; j < n; j++)
10       A[i*n+j] -= L[i*n+b]*A[b*n+j];
11   }
12 }
```


© Colfax International, 2013–2016

LU Decomposition: Performance



Paper: http://xeonphi.com/papers/lu

 © Colfax International, 2013–2016

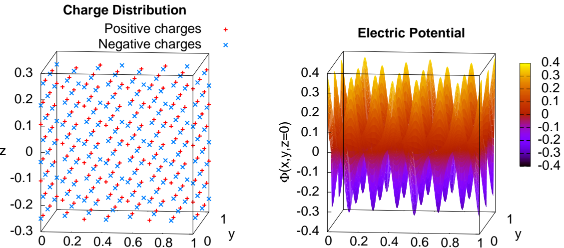
Example 3: Coulomb's Law

© Colfax International, 2013–2016

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

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

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



Paper: http://xeonphi.com/papers/autovec

© Colfax International, 2013–2016

Elegant, but Inefficient Solution: Array of Structures

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

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

 © Colfax International, 2013–2016

Arrays of Structures versus Structures of Arrays

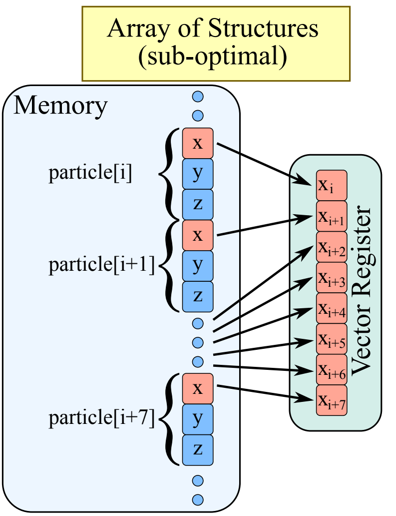
Array of Structures (AoS)

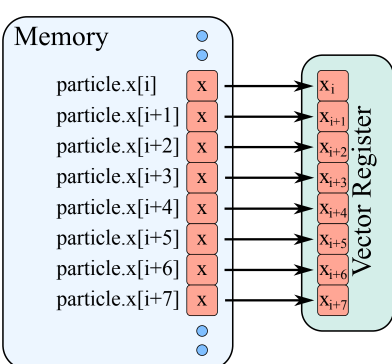
```
struct Charge { // Elegant, but ineffective data layout
  float x, y, z, q; // Coordinates and value of this charge
};
// The following line declares a set of m point charges:
Charge chg[m];
```

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, 2013–2016

Optimized Solution: Structure of Arrays, Unit-Stride Access

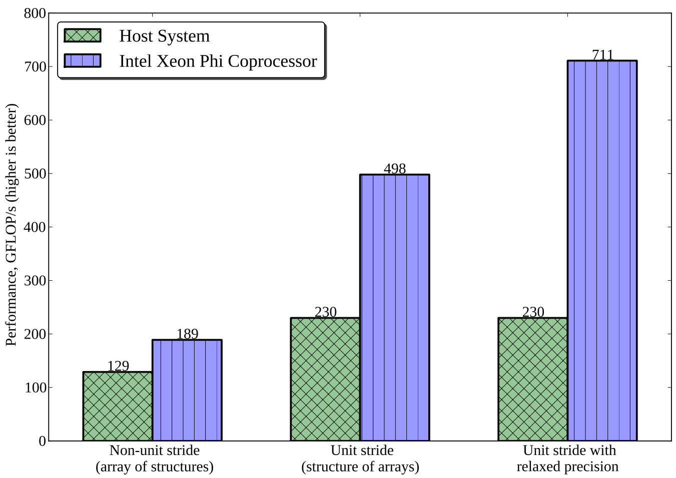
```
struct Charge_Distribution {
  // Data layout permits effective vectorization of Coulomb's law application
  const int m; // Number of charges
  float *x, *y, *z, *q; // Arrays of x-, y- and z-coordinates of charges
};
```

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

 © Colfax International, 2013–2016

Electric Potential Calculation with Coulomb's Law

Based on 10 FLOPs per interaction:



© Colfax International, 2013–2016

© Colfax International, 2013–2016

Example: Numerical Integration

$$I(a,b) = \int_a^b \frac{1}{\sqrt{x}} \mathrm{d}x$$

Rectangle method:

$$\begin{aligned} \Delta x &= \frac{b-a}{n}, \\ x_i &= (i+1)\Delta x, \\ I(a,b) &= \sum_{i=0}^{n-1} \frac{1}{\sqrt{x_i}} \Delta x + O(\Delta x). \end{aligned}$$

```
1   float Integrate(const float a,
2                   const float b,
3                   const int N) {
4     const float dx = (b-a)/float(n);
5     float S = 0.0f;
6     for (int i = 0; i < n; i++) {
7       const float xi = dx*float(i+1);
8       S += 1.0f/sqrtf(xi) * dx;
9     }
10    return S;
11  }
```


© Colfax International, 2013–2016

Implementation with SSE4.2

```
float Integrate(const float a,
                const float b, const int n) {
  __m128 dx = _mm_set1_ps((b - a)/float(n));
  __m128 S  = _mm_set1_ps(0.0f);
  for (int i = 0; i < n; i += 4) {
    __m128i ip1  =
             _mm_set_epi32(i+4, i+3, i+2, i+1);
    __m128  ip1f = _mm_cvtepi32_ps(ip1);
    __m128  xi = _mm_mul_ps(dx, ip1f);
    __m128  fi = _mm_rsqrt_ps(xi);
    __m128  dS = _mm_mul_ps(fi, dx);
    S  = _mm_add_ps(S, dS);
  }
  ConverterType c;
  c.v = S;
  return c.f[0] + c.f[1] + c.f[2] + c.f[3];
}
```

That is fine, *but*...

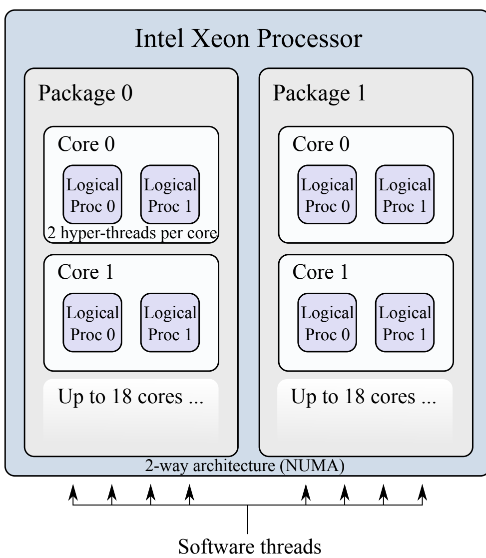
- Assuming `n` is a multiple of 4
- Only for SSE4.2 (circa 2011)
- No memory access. If we had some, peeling may be needed

© Colfax International, 2013–2016

§9. Threading

 © Colfax International, 2013–2016

Processor Hierarchy



Hierarchy:

Packages ->
Cores ->
Logical Processors

Core presents itself as 2 (4) logical processors due to:

a) 2-way hyper-threading (Xeon)

b) 4 hardware threads per core (Xeon Phi)

Terminology:

OS Proc - numeric ID of logical processor

Software thread - stream of instructions in a process

Jargon:

"physical core" = core

"logical core" = logical processor

Scalability Expectations (CPU)

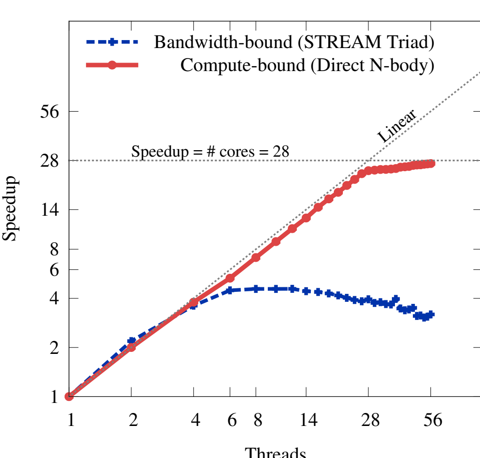
T = number of threads

$$\text{Speedup}(T) = \frac{\text{Performance}(T)}{\text{Performance}(1)}$$

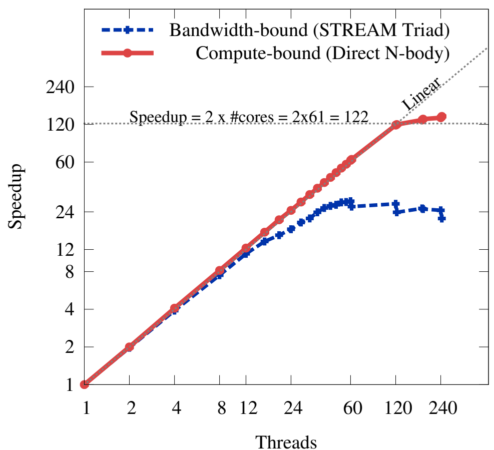
$$\text{Efficiency}(T) = \frac{\text{Speedup}(T)}{T}$$

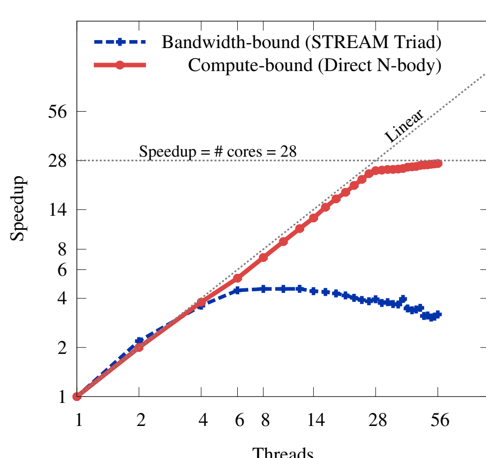
Linear scaling (ideal case, 100% parallel efficiency):

$$\text{Speedup}(T) = T$$



Scalability Expectations: MIC versus CPU



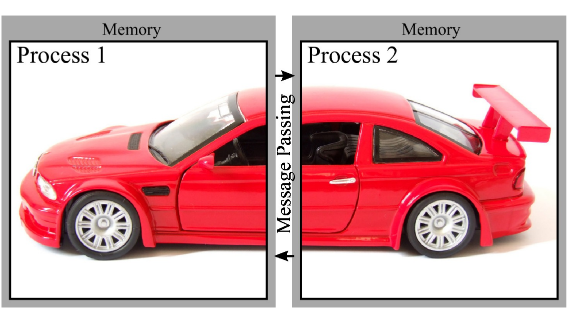


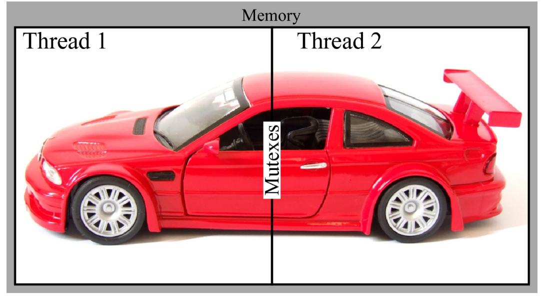
 © Colfax International, 2013–2016

Threads and Threading Frameworks

Threads versus Processes

Option 1: Partitioning data set between threads/processes



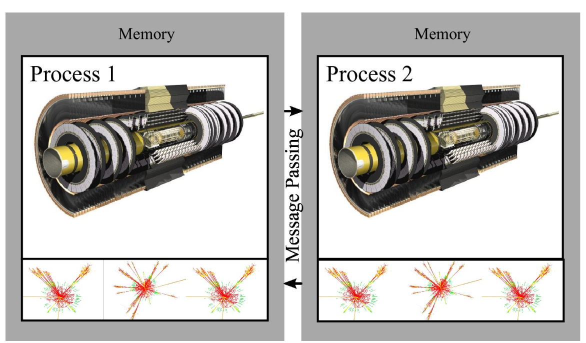


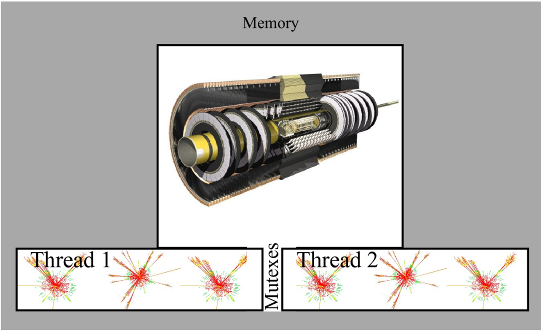
Examples: computational fluid dynamics (CFD), image processing.

 © Colfax International, 2013–2016

Threads versus Processes

Option 2: Sharing data set between threads/processes





Examples: particle transport simulation, machine learning (inference).

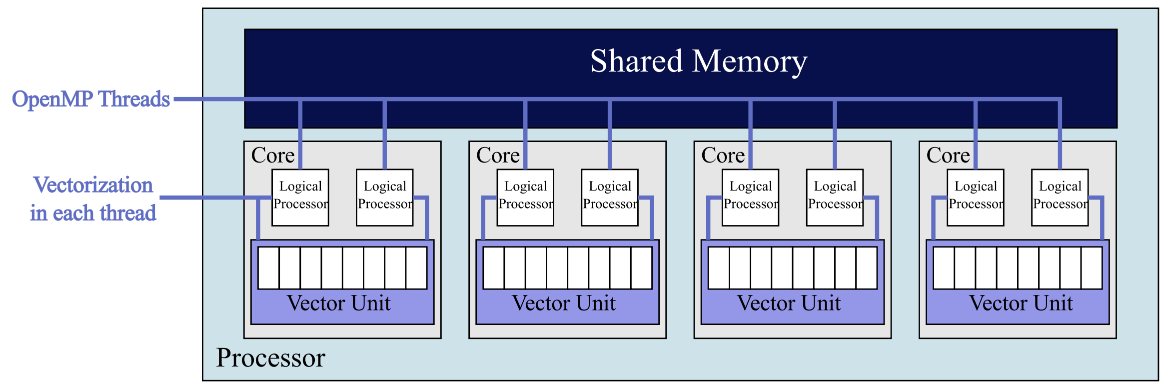
 © Colfax International, 2013–2016

Threading Frameworks

Framework	Implementation	Complexity	Functionality
POSIX Threads	Various	Simple	Manually control everything
Cilk Plus	Intel, Public	Very simple	Automatic loops and tasks, no user control
TBB	Intel, Public	Complex	Automatic trees of tasks, automatic scheduler
OpenMP	Various	Simple to Complex	HPC-specific functionality, automatic and manual control possible

© Colfax International, 2013–2016

Co-Existence with Vectors



 © Colfax International, 2013–2016

Lab Time

 © Colfax International, 2013–2016

Lab Time

Instructor will probably demonstrate OpenMP with the following labs:

- `labs/3/3.02-OpenMP-basics`
- `labs/4/4.04-threading-misc-histogram` (steps 3, 4)

Time permitting, also perform the following exercises:

- `labs/4/4.04-threading-misc-histogram` (step 5) (recommended)
- `labs/3/3.03-OpenMP-reduction` (optional)
- `labs/3/3.04-OpenMP-tasks` (optional)
- `labs/4/4.05-threading-insufficient-parallelism-sweep` (optional)
- `labs/4/4.05-threading-scheduling-jacobi` (optional)
- `labs/4/4.07-threading-affinity` (optional)

 © Colfax International, 2013–2016

OpenMP Basics

© Colfax International, 2013–2016

“Hello World” OpenMP Programs

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(){
5    const int nt=omp_get_max_threads();
6    printf("OpenMP with %d threads\n", nt);
7
8  #pragma omp parallel
9    {
10     printf("Hello World from thread %d\n", omp_get_thread_num());
11   }
12 }
```

“Hello World” OpenMP Programs

```
vega@lyra% icpc -qopenmp hello_omp.cc
vega@lyra% export OMP_NUM_THREADS=5
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_THREADS` controls number of OpenMP threads (default: logical CPU count)

Control of Variable Sharing

Method 1: using clauses in pragma omp parallel (C, C++, Fortran):

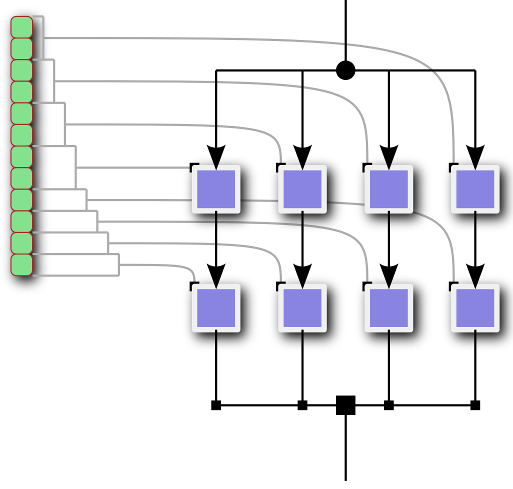
```
1 int A, B; // Variables declared at the beginning of a function
2 #pragma omp parallel private(A) shared(B)
3   {
4     // Each thread has its own copy of A, but B is shared
5   }
```

Method 2: using scoping (only C and C++):

```
1 int B; // Variable declared outside of parallel scope - shared by default
2 #pragma omp parallel
3   {
4     int A; // Variable declared inside the parallel scope - always private
5     // Each thread has its own copy of A, but B is shared
6   }
```

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.

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.

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++) {
3   printf("Iteration %d is processed by thread %d\n",
4          i, omp_get_thread_num());
5   // ... iterations will be distributed across available threads...
6 }
```


© Colfax International, 2013–2016

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

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
+2
+3
Three
Four

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 } }
```



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

 © Colfax International, 2013–2016

Avoiding Races with Atomic Operations

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

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
{ // Lightweight synchronization
#pragma omp atomic
  total += i;
}
```


© Colfax International, 2013–2016

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*

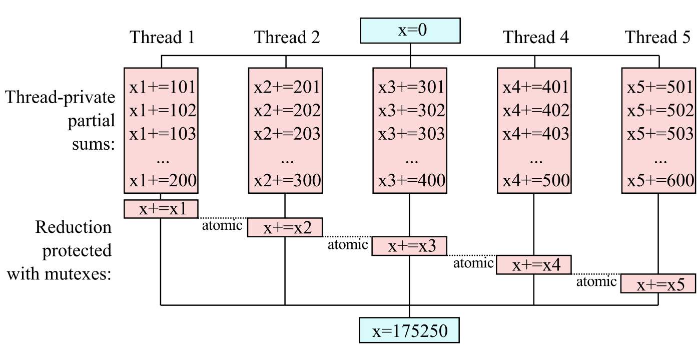
Capture : operations in the form `v = x++`, `v = x-`, `v = -x`, `v = ++x`, `v = x` *binop expr*

- 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.

Avoiding Races with Thread-Private Storage

Correct and efficient code:

```
int total = 0;
#pragma omp parallel
{
  int total_thr = 0;
#pragma omp for
  for (int i=0; i<n; i++)
    total_thr += i;

#pragma omp atomic
  total += total_thr;

}
```

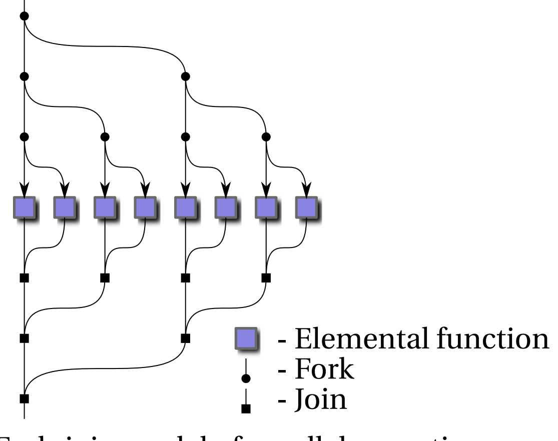


Tasks in OpenMP

 © Colfax International, 2013–2016

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);
  }
}
```

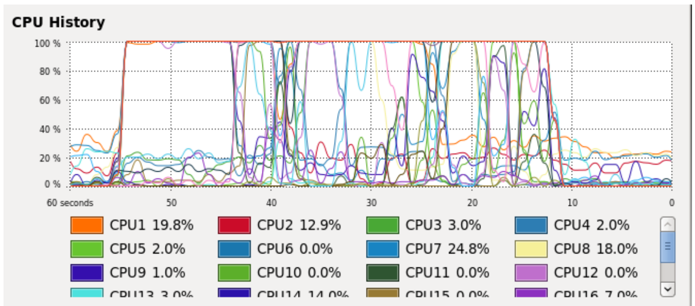
```
// Recursive task spawning:
void RecursiveWorkload(Arg* args) {
  if (args->size > threshold) {
    // Split work
    Arg* args1=args->FirstHalf();
    Arg* args2=args->SecondHalf();

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

 © Colfax International, 2013–2016

Controlling Thread Affinity

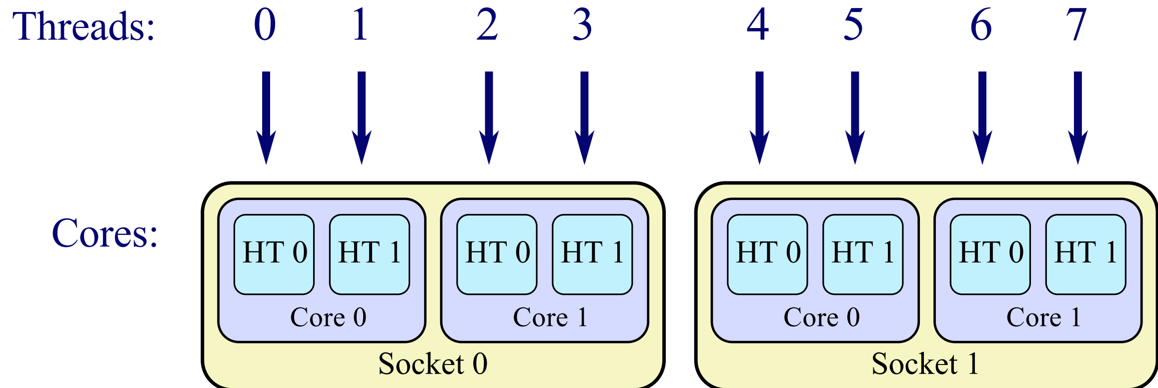
What is Thread Affinity

- OpenMP threads may migrate between cores according to OS decisions.
- Forbid migration — improve locality — increase the performance.



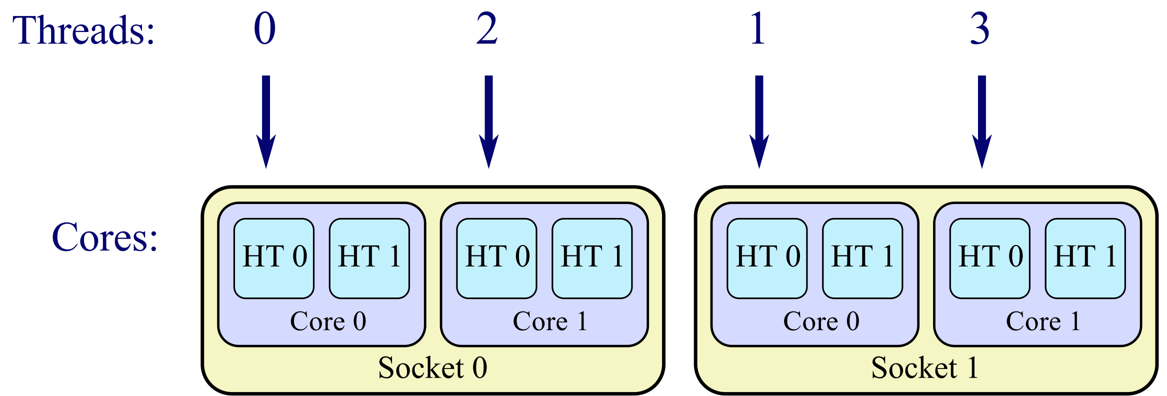
Thread Affinity: Compact Pattern

Generally beneficial for compute-bound applications.



Thread Affinity: Scatter Pattern

Generally beneficial for bandwidth-bound applications.



© Colfax International, 2013–2016

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.

The most important argument is type:

- compact: place threads as *close to each* other as possible
- scatter: place threads as *far from each* other as possible

© Colfax International, 2013–2016

Example 1: Binning, Parallel Reduction

 © Colfax International, 2013–2016

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 }
```



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

 © Colfax International, 2013–2016

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, 2013–2016

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 critical
    { hist[index[c]]++; }
}
```

Histogram Calculation Example: Adding Thread Parallelism

Correct, but inefficient solution:

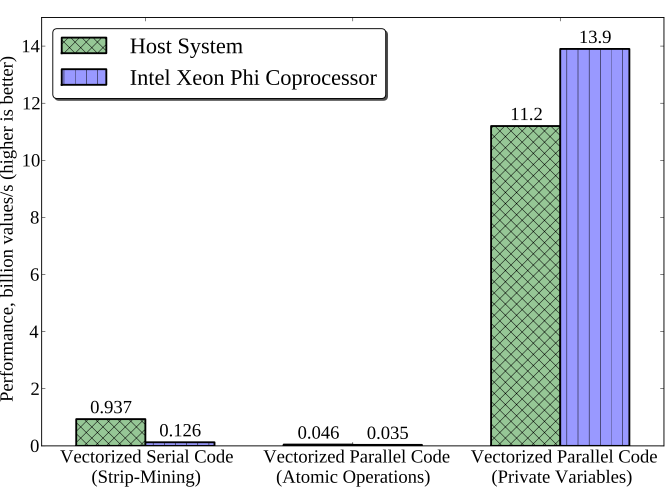
```
#pragma omp parallel for schedule(guided)
for (int ii = 0; ii < n; ii += vecLen) {
  int index[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
  for (int i = ii; i < ii + vecLen; i++)
    index[i-ii] = (int) ( age[i] * invGroupWidth );
  for (int c = 0; c < vecLen; c++)
    // Protect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

 © Colfax International, 2013–2016

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 } }
```

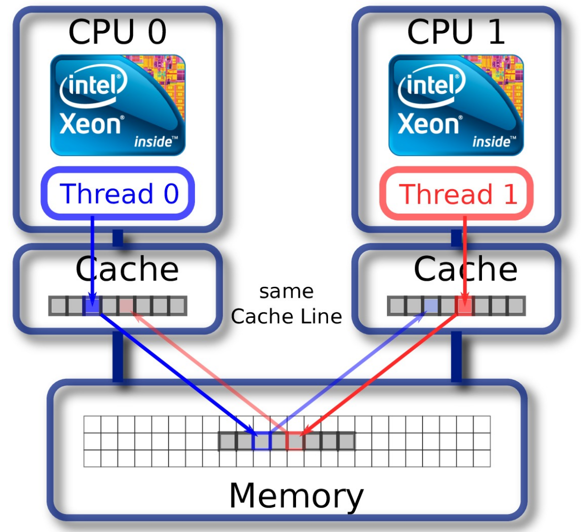
Using Reduction instead of Synchronization



Example 2: Binning, False Sharing

 © Colfax International, 2013–2016

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, 2013–2016

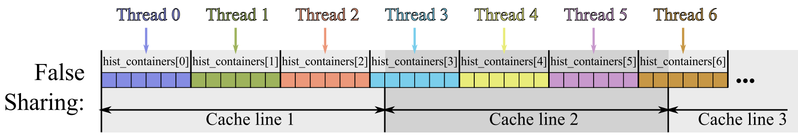
False Sharing. Data Padding and Private Variables

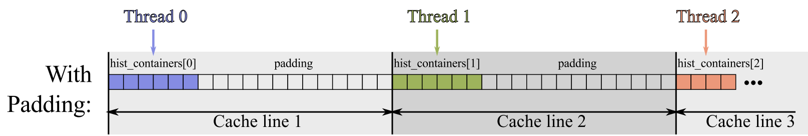
```
const int m = 5;
int hist_thr[nThreads][m];
#pragma omp parallel for
for (int ii = 0; ii < n; ii += vecLen) {
  // ...
  // False sharing occurs here
  for (int c = 0; c < vecLen; c++)
    hist_thr[iThread][index[c]]++;
}
// Reducing results from all threads to the common histogram hist
for (int iThread = 0; iThread < nThreads; iThread++)
  hist[0:m] += hist_thr[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][:]`

 © Colfax International, 2013–2016

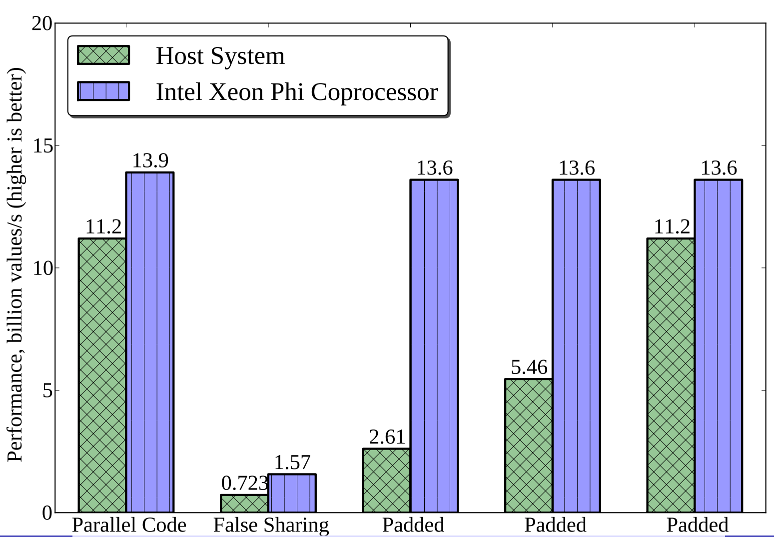
Padding to Avoid False Sharing






```
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
```

Padding to Avoid False Sharing



© Colfax International, 2013–2016

Example 3: Stencil, Expanding Iteration Space

 © Colfax International, 2013–2016

Example: Dealing with Insufficient Parallelism

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

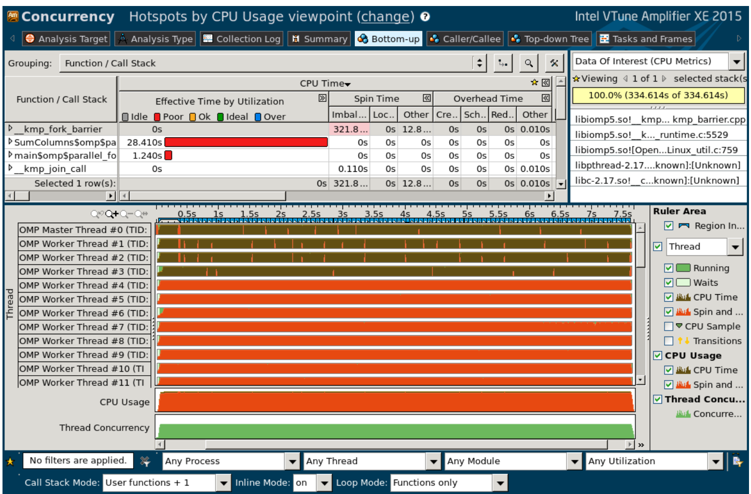
- m=4 is small, smaller than the number of threads in the system
- n $\approx 10^8$ is large enough so that matrix does not fit into cache

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

 © Colfax International, 2013–2016

Dealing with Insufficient Parallelism

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

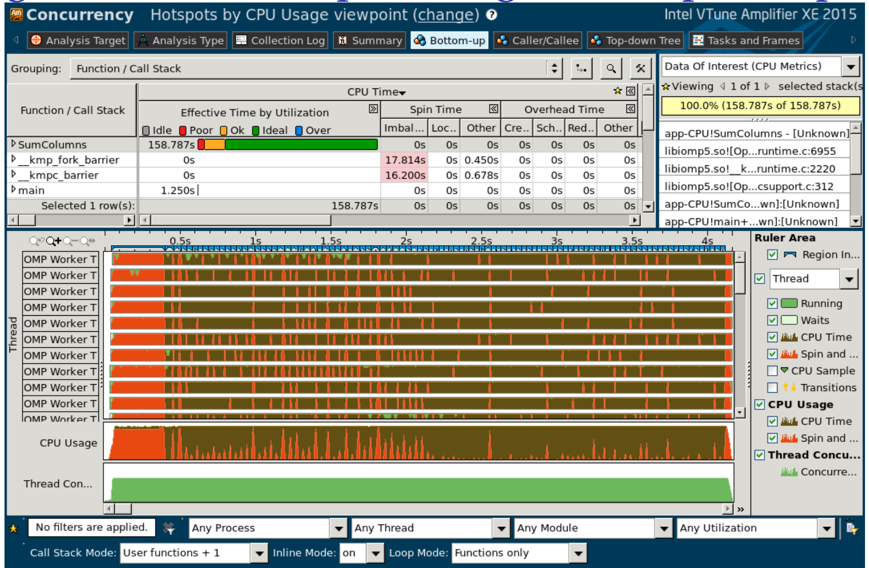


© Colfax International, 2013–2016

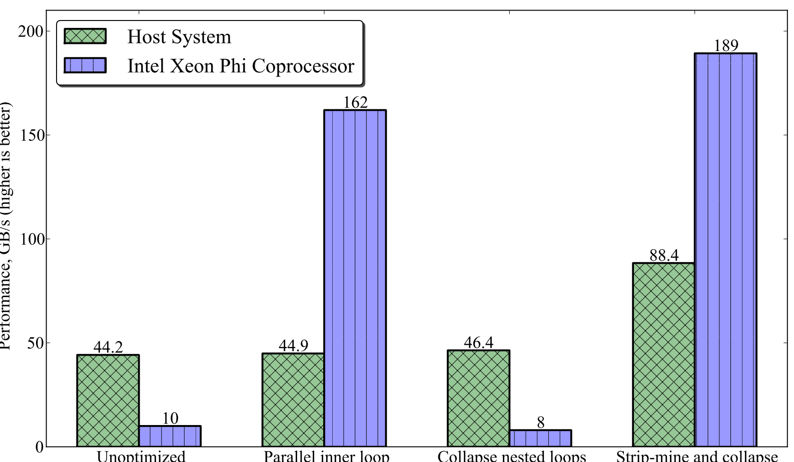
Exposing Parallelism: Strip-Mining and Loop Collapse

```
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, 2013–2016

Example 4: Thread Affinity

© Colfax International, 2013–2016

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)
```

Compute-Bound, KMP_AFFINITY=compact/balanced

```
double* A = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);
double* B = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);
double* C = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);

for(int k = 0; k < nIter; k++) {

  dgemm(&tr, &tr, &N, &N, &N, &v, A, &Nld, B, &Nld, &v, C, &N);

  double flopsNow = (2.0*N*N*N+1.0*N*N)*1e-9/(t2-t1);
  printf("Iteration %d: %.1f GFLOP/s\n", k+1, flopsNow);
}
_mm_free(A); _mm_free(B); _mm_free(C);
```

 © Colfax International, 2013–2016

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
```


© Colfax International, 2013–2016

Other Optimization Topics for Thread Parallelism

Examples found in our book:

- Avoiding excessive synchronization with reduction
- Load balancing across threads
- Using thread affinity to partition a multi-socket NUMA system

 © Colfax International, 2013–2016

§10. Memory Access

 © Colfax International, 2013–2016

Vectorization is Worth Nothing Without Caches

© Colfax International, 2013–2016

How Cheap are FLOPs?

Theoretical estimates, Intel Xeon E5-2697 V3 processor

$$\text{Performance} = 28\,\text{cores} \times 2.7\,\text{GHz} \times (256/64)\,\text{vec.lanes} \times 2\,\text{FMA} \times 2\,\text{FPU} \approx 1.2\,\text{TFLOP/s}$$

$$\text{Required Data Rate} = 1.2\,\text{TFLOP/s} \times 8\,\text{bytes} \approx 10\,\text{TB/s}$$

$$\text{Memory Bandwidth} = \eta \times 2 \times 59.7 \approx 0.1\,\text{TB/s}$$

$$\text{Ratio} = 10/0.1 \approx 100\ (\text{FLOPs})/(\text{Memory Access})$$

If the Arithmetic Intensity is...

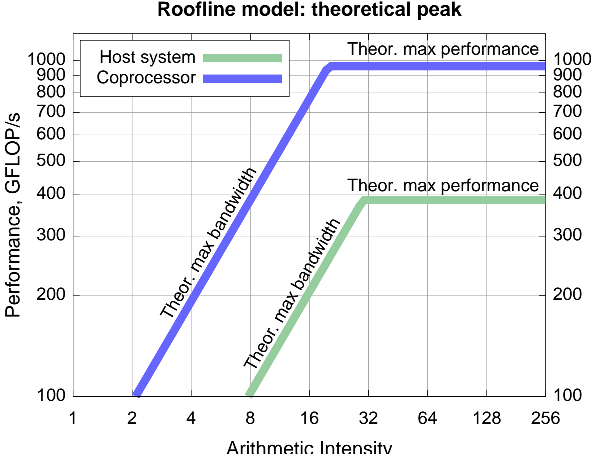
- $>$ 100 (FLOPs)/(Memory Access) — Compute Bound Application
- $<$ 100 (FLOPs)/(Memory Access) — Bandwidth Bound Application

On Computational Complexity of Algorithms

Type	Properties	Examples
$O(N)$	Each data element is used a fixed number of times. Memory-bound unless the number of times is large.	Array scaling, image brightness adjustment, vector dot-product.
$O(N^\alpha)$	Each element is used $N^{\alpha-1}$ times. A lot of data reuse for $\alpha \gtrsim 2$. Good implementation can be compute-bound, poor one – memory-bound.	Matrix-matrix multiplication: $O\left(N^{3/2}\right)$ (N = amount of data in matrix), direct N-body calculation: $O\left(N^2\right)$
$O(N \log N)$	Each element is used $\log N$ times. For small problems – memory-bound, for very large problems transitions to compute-bound	Fast Fourier transform, merge sort
$O(\log N)$	Always memory-bound.	Binary search

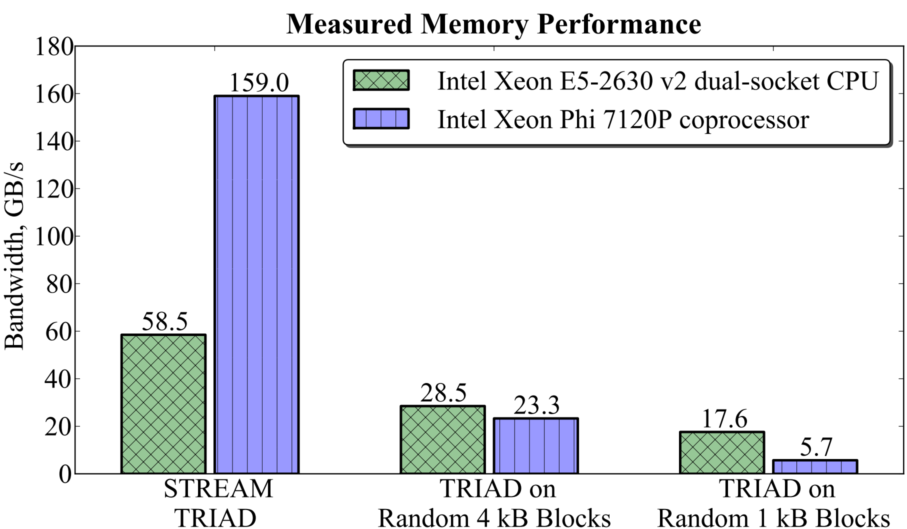
N = data size

Arithmetic Intensity and Roofline Model



More on roofline model: Williams et al.

Streaming Versus Random Access



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).

 © Colfax International, 2013–2016

Lab Time

 © Colfax International, 2013–2016

Lab Time

You may not have enough time for this during the 1-day seminar, but at home you can perform exercises in:

- `labs/4/4.08-memory-tiling-matrix_x_vector` (recommended)
- `labs/4/4.08-memory-loop-fusion-statistics` (recommended)

The rest of this section may help you with the exercises.
Consider the optional Section 10.

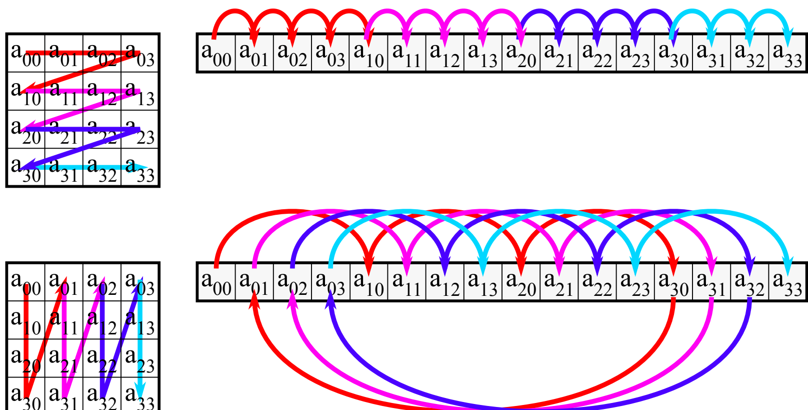
 © Colfax International, 2013–2016

Loop Permutation

 © Colfax International, 2013–2016

Principle

Choose loop order to maintain unit-stride memory access



Compiler may or may not be able to automate loop permutation.

Example: Over-simplified Matrix-Matrix Multiplication

$$C = AB \quad \Leftrightarrow \quad C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$$

Before:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int j = 0; j < n; j++)
4 #pragma vector aligned
5     for (int k = 0; k < n; k++)
6       C[i*n+j]+=A[i*n+k]*B[k*n+j];
```

After:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int k = 0; k < n; k++)
4 #pragma vector aligned
5     for (int j = 0; j < n; j++)
6       C[i*n+j]+=A[i*n+k]*B[k*n+j];
```

 © Colfax International, 2013–2016

Loop Fusion

 © Colfax International, 2013–2016

Principle

- The order of nested loops must be chosen for best locality of data access
- At `-O2` and above, the compiler automatically interchanges loops in some cases
- In other cases, loop interchange must be investigated manually

© Colfax International, 2013–2016

Loop Fusion Technique

Re-use data in cache by fusing loops in a data processing pipeline

```
MyDataType* data = new MyDataType(n);

for (int i = 0; i < n; i++)
  Initialize(data[i]);

for (int i = 0; i < n; i++)
  Stage1(data[i]);

for (int i = 0; i < n; i++)
  Stage2(data[i]);
```

```
MyDataType* data = new MyDataType(n);

for (int i = 0; i < n; i++) {

  Initialize(data[i]);

  Stage1(data[i]);

  Stage2(data[i]);
}
```

Potential positive side-effect: less data to carry between stages, reduced memory footprint, improved performance (see, e.g., this paper).

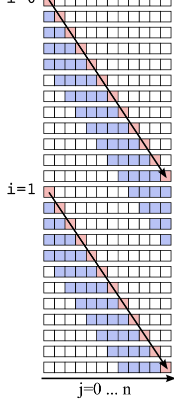
Loop Tiling

© Colfax International, 2013–2016

Loop Tiling

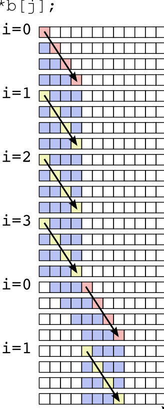
Original:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];
```



Tiled:

```
for (jj=0; jj<n; jj+=TILE)
  for (i=0; i<m; i++)
    for (j=jj; j<jj+TILE; j++)
      ...=...*b[j];
```



- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

Cache size: 4
TILE=4
(must be tuned to cache size)

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 75%

 © Colfax International, 2013–2016

Loop Tiling (Cache Blocking) – Procedure

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

```
1 // Step 1: strip-mine inner loop
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
```


© Colfax International, 2013–2016

Loop Tiling (Unroll-and-Jam/Register Blocking)

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

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

```
1  // Step 2: permute and vectorize outer loop
2  for (int ii = 0; ii < m; ii += TILE)
3  #pragma simd
4    for (int j = 0; j < n; j++)
5      for (int i = ii; i < ii + TILE; i++)
6        compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times
```


© Colfax International, 2013–2016

Loop Tiling (Unroll-and-Jam) – Alternative Implementation

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

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

```
// Step 2: permute middle two loops
for (int ii = 0; ii < m; ii += TILE)
  for (int jj = 0; jj < n; jj += VECLEN)
    for (int i = ii; i < ii + TILE; i++)
      for (int j = jj; j < jj + VECLEN; j++)
        compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times
```


© Colfax International, 2013–2016

Cache-Oblivious Recursion

© Colfax International, 2013–2016

Principle

Matrix A

j=0,1,2, ... ,n-1

i=0,1,2, ... ,m-1

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

j=0,1,2, ... ,n-1

i=0,1,2, ... ,m-1

0	2	8	10	32	34	40	42
1	3	9	11	33	35	41	43
4	6	12	14	36	38	44	46
5	7	13	15	37	39	45	47
16	18	24	26	48	50	56	58
17	19	25	27	49	51	57	59
20	22	28	30	52	54	60	62
21	23	29	31	53	55	61	63

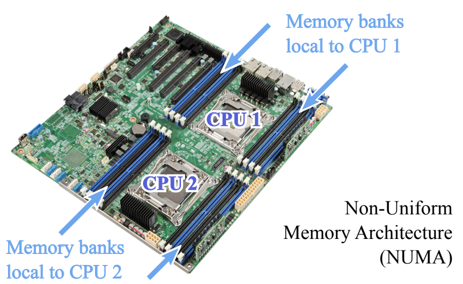
© Colfax International, 2013–2016

First-touch Allocation

© Colfax International, 2013–2016

NUMA Architectures

NUMA = Non-Uniform Memory Access. Cores have fast access to local memory, slow access to remote memory.



Examples:

- Multi-socket Intel Xeon processors
- Second generation Intel Xeon Phi

 | © Colfax International, 2013–2016

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

```
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;
```

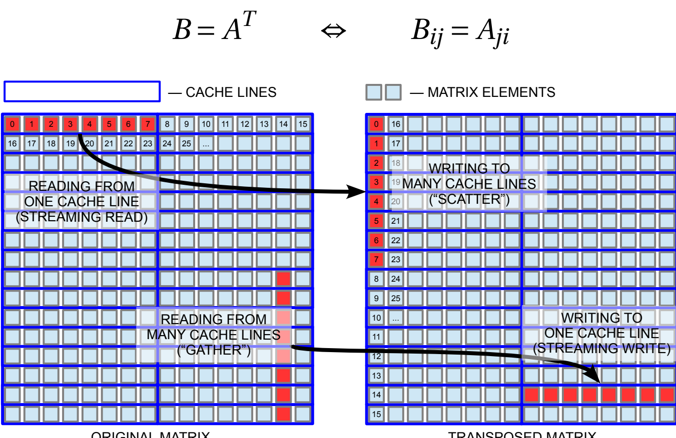
 © Colfax International, 2013–2016

Example 1: Matrix Transposition, Tiling

 © Colfax International, 2013–2016

Example: Matrix Transposition

$$B = A^T \quad \Leftrightarrow \quad B_{ij} = A_{ji}$$



See also this paper.

Matrix Transposition

Before:

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    B[i*n + j] = A[j*n + i];
```

After:

```
const int tile = 200;
if (n%tile != 0) exit(1);

#pragma omp parallel for
for (int ii=0; ii<n; ii+=tile)
 for (int jj=0; jj<n; jj+=tile)
  for (int i=ii; i<ii+tile; i++)
   for (int j=jj; j<jj+tile; j++)
    B[i*n + j] = A[j*n + i];
```

Example 2: Matrix-Vector Multiplication, Tiling

© Colfax International, 2013–2016

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++) // Each value of A[i*n+j] is used only once
8        c[i] += A[i*n+j] * b[j];   // Each value of b[j] is used a total of m times
9  }
```

Non-optimal performance due to inefficient cache use

© Colfax International, 2013–2016

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 } } }
```


© Colfax International, 2013–2016

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 } } }
```


© Colfax International, 2013–2016

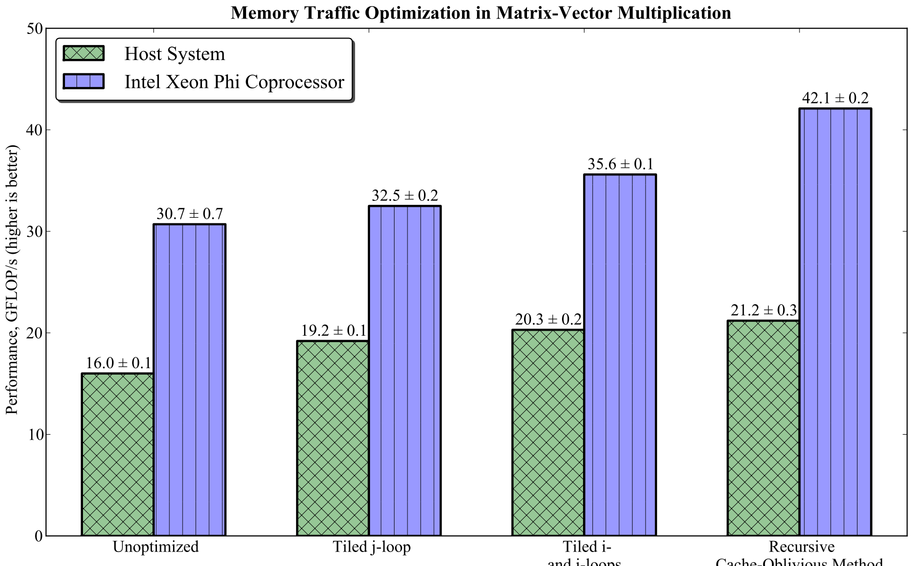
Example 3: Matrix-Vector Multiplication, Recursion

 © Colfax International, 2013–2016

Example: Matrix-Vector Multiplication

```
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 } }
```

Performance of Matrix Vector Multiplication



§11. Distributed Computing with MPI

© Colfax International, 2013–2016

Distributed Memory Parallelism and MPI

 © Colfax International, 2013–2016

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

 © Colfax International, 2013–2016

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`

Compiling and Running MPI applications

1. Compile and link with the MPI wrapper of the compiler:
   - `mpiicc` for C,
   - `mpiicpc` for C++,
   - `mpiifort` for Fortran 77 and Fortran 95.
2. Set up MPI environment variables *and* `I_MPI_MIC=1`
3. NFS-share or copy the MPI library and the application executable to the coprocessors
4. Launch with the tool `mpirun`
   - Colon-separated list of executables and hosts (argument `-host` *hostname*),
   - Alternatively, use the machine file to list hosts
   - Coprocessors have hostnames defined in `/etc/hosts`

© Colfax International, 2013–2016

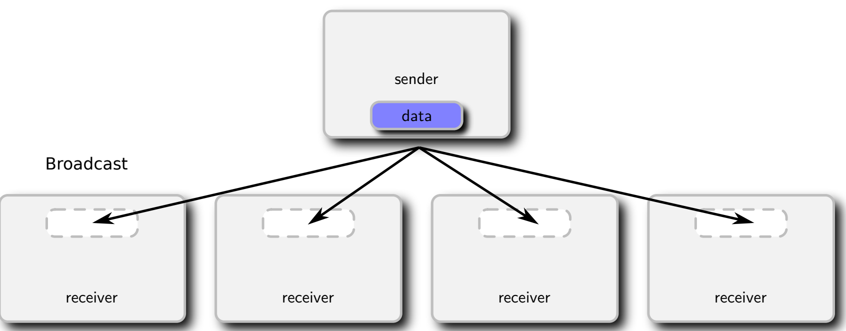
Point to Point Communication

```
if (rank == sender) {

 char outgoingMsg[messageLength];
 strcpy(outgoingMsg, "/Jenny");
 MPI_Send(&outgoingMsg, messageLength, MPI_CHAR, receiver, tag, MPI_COMM_WORLD);


} else if (rank == receiver) {

 char incomingMsg[messageLength];
 MPI_Recv (&incomingMsg, messageLength, MPI_CHAR, sender,
 tag, MPI_COMM_WORLD, &stat);
 printf ("Received message with tag %d: '%s'\n", tag, incomingMsg);


}
```

 © Colfax International, 2013–2016

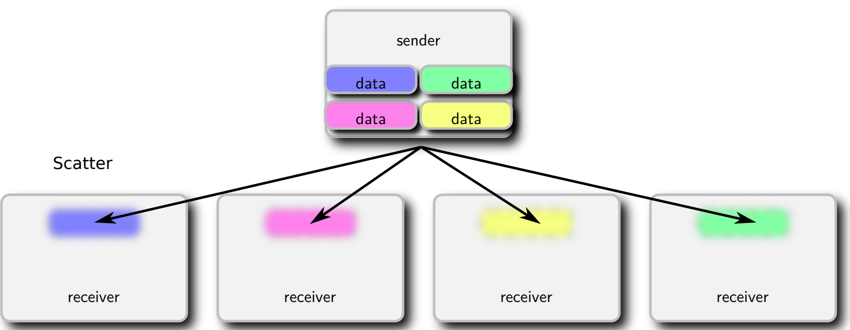
Collective Communication: Broadcast

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



Collective Communication: Scatter

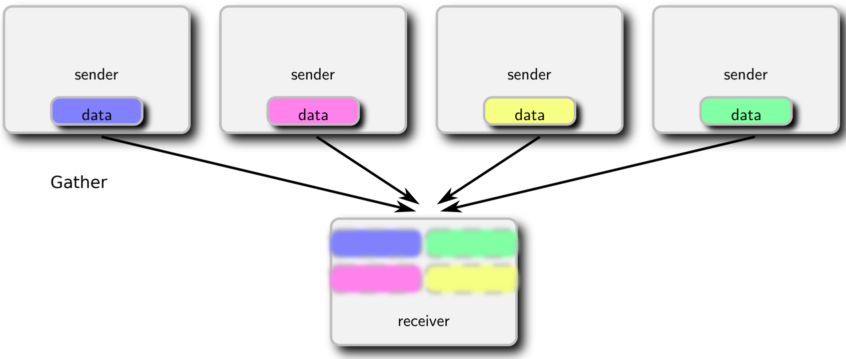
```
1 int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
2     int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
```



 © Colfax International, 2013–2016

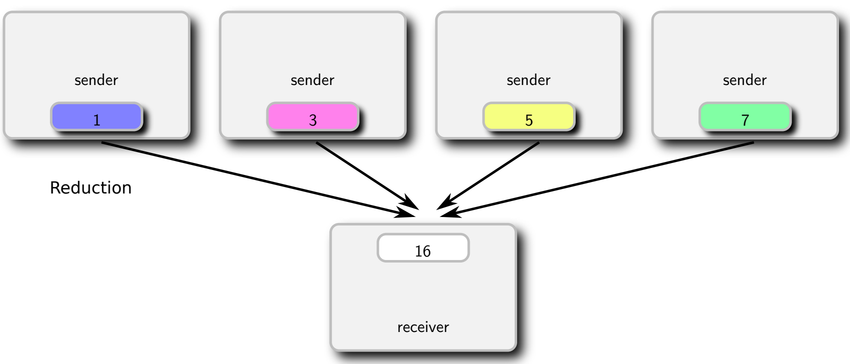
Collective Communication: Gather

```
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);
```



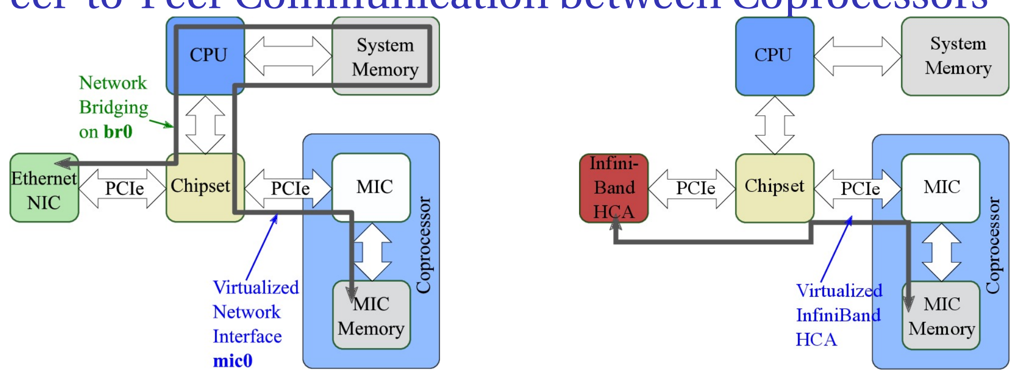
Available reducers: max/min, minloc/maxloc, sum, product, AND, OR, XOR (logical or bitwise).

© Colfax International, 2013–2016

MPI Communication Controls

© Colfax International, 2013–2016

Peer-to-Peer Communication between Coprocessors

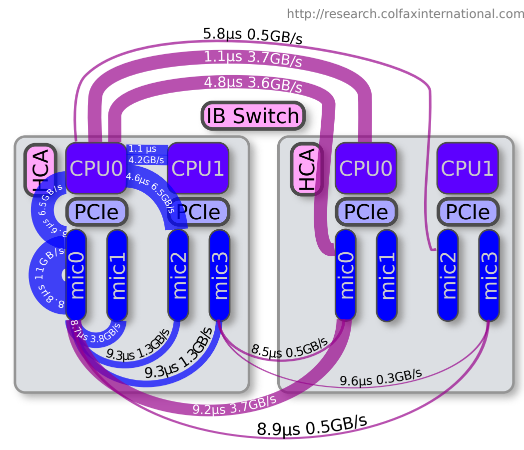


- Left: Gigabit Ethernet bridging on host allows to place coprocessors on the same subnet as hosts
- Right: Coprocessor Communication Link (CCL) – virtualization of an InfiniBand device on each coprocessor

© Colfax International, 2013–2016

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`

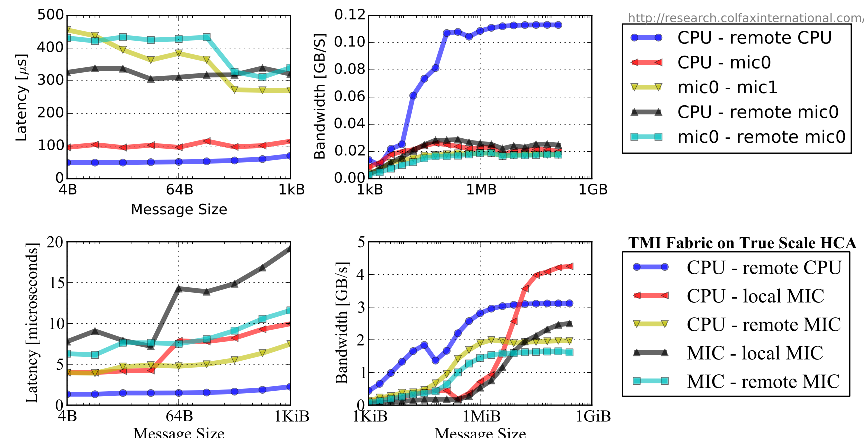


Our publication with details:
http://xeonphi.com/papers/p2p

 © Colfax International, 2013–2016

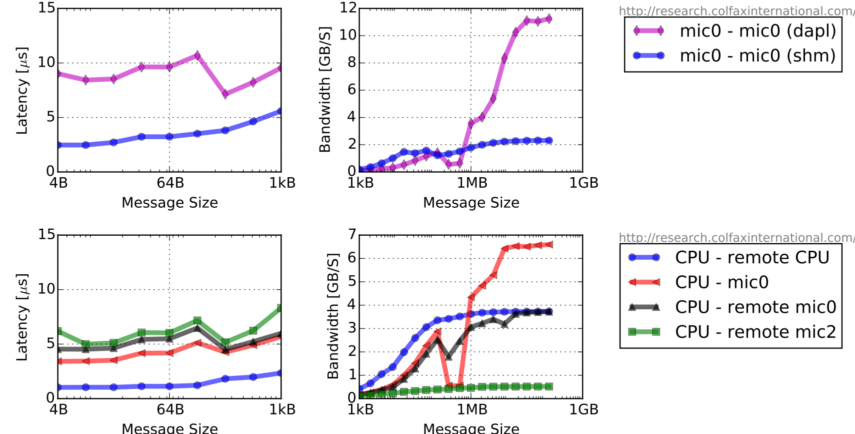
Gigabit Ethernet versus Intel True Scale Interconnects

- Ethernet+TCP between coprocessors slower than the hardware limit
- InfiniBand approaches the hardware limit from CPU to coprocessors



Intra-Device and Intra-Node Communication

- Fabrics for messages within a single coprocessor:
- `shm` provides better latency, `dapl` – greater bandwidth



 © Colfax International, 2013–2016

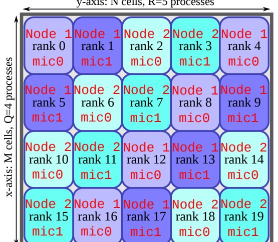
Communication Pattern Matters

© Colfax International, 2013–2016

Communication Pattern Control and MPI Rank Enumeration

```
vega@lyra% cat hosts.txt
node1-mic0
node1-mic1
node2-mic0
node2-mic1
vega@lyra% mpirun -np 20 \
>          -machine hosts.txt ...
```

```
vega@lyra% cat hosts.txt
node1-mic0:5
node1-mic1:5
node2-mic0:5
node2-mic1:5
vega@lyra% mpirun \
>          -machine hosts.txt ...
```



Publication: link, see also video.

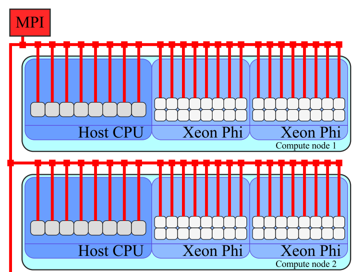
Inter-Operation with OpenMP

 © Colfax International, 2013–2016

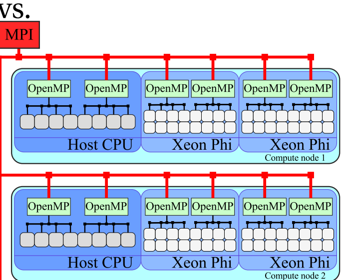
Hybrid MPI+OpenMP

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



vs.



 © Colfax International, 2013–2016

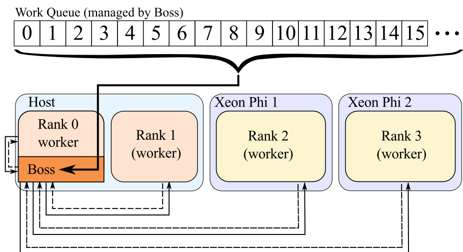
Hybrid MPI+OpenMP

- 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, 2013–2016

Multi-Threading within MPI Processes

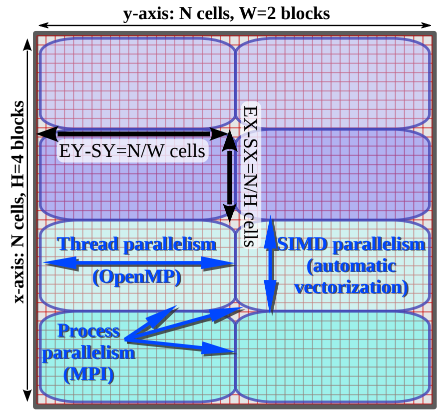
```
1  if(myRank == 0) { // Rank 0 has both a boss and a worker inside:
2      const int nThreads = omp_get_max_threads();  omp_set_nested(1);
3  #pragma omp parallel sections num_threads(2)
4      {
5  #pragma omp section
6        { DistributeWork(nOptions, option, mpiWorldSize); } // Boss
7  #pragma omp section
8        { omp_set_num_threads(nThreads-1); // Worker in rank 0:
9          ReceiveWork(option, payoff, myRank, optioncount);  } // ...
```

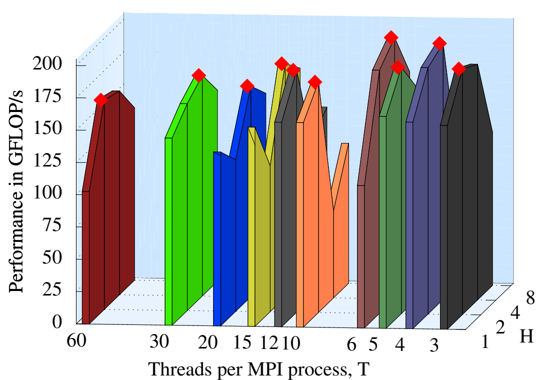


 © Colfax International, 2013–2016

Example of OpenMP and MPI Inter-Operation

Number of threads per process may be a tuning parameter:





Case study: this paper

© Colfax International, 2013–2016

Example 1: Asian Options, Heterogeneous Distributed Computing

© Colfax International, 2013–2016

Example: the Monte Carlo Method of Asian Option Pricing

1) Simulate Random-Walk of Asset Price

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

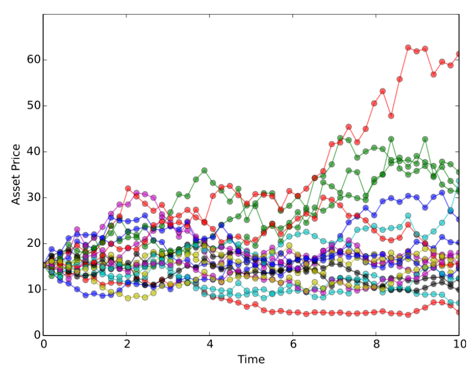
Using the Solution

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

2) Perform Asian Option Price Averaging

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

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



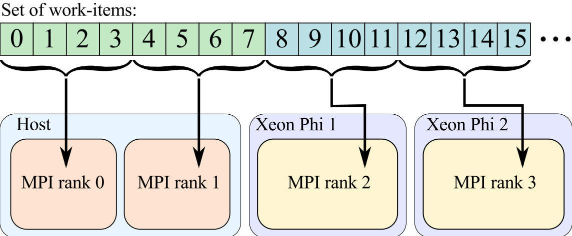
3) Compute Discounted Pay-off

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

© Colfax International, 2013–2016

Heterogeneous Calculation without Load Balancing

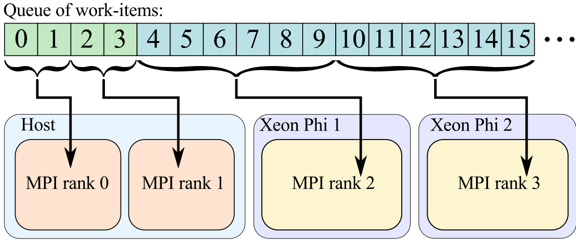
```
1  const double optionsPerProcess = double(nOptions)/double(mpiWorldSize);
2  const int myFirstOption = int(optionsPerProcess*(myRank));
3  const int myLastOption  = int(optionsPerProcess*(myRank+1));
4
5  // Static, even load distribution: assign options to ranks
6  for (int i = myFirstOption; i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



 © Colfax International, 2013–2016

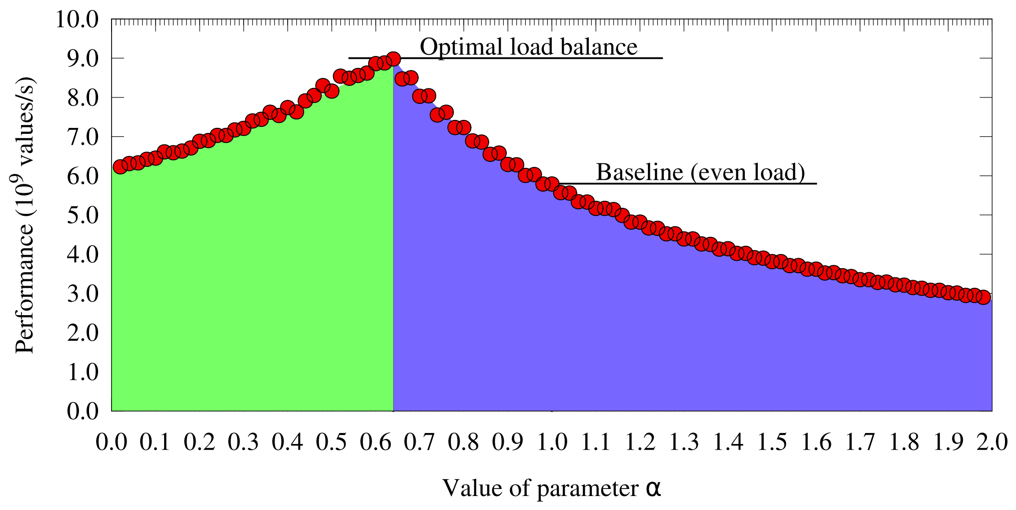
Static Load Balancing

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



 © Colfax International, 2013–2016

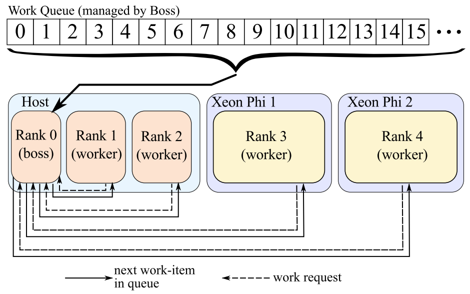
Static Load Balancing: Parameter Tuning



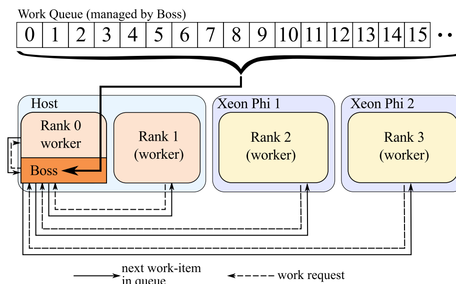
© Colfax International, 2013–2016

Dynamic Load Balancing

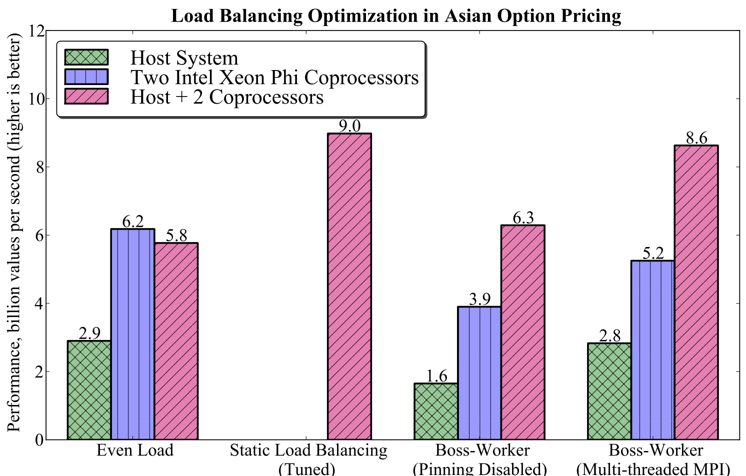
```
1  if (myRank == 0) // Boss's branch
2    DistributeWork(nOptions, option, mpiWorldSize);
3  else // Workers' branch
4    ReceiveWork(option, payoff, myRank);
```



or



Performance with Different Scheduling Modes



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

 © Colfax International, 2013–2016

§12. Additional Resources

 © Colfax International, 2013–2016

Course Recap

 © Colfax International, 2013–2016

Programming Models for Xeon Phi Coprocessors

1. Native coprocessor applications
   - Compile with `-mmic`
   - Run with `micnativeloadex` or `scp+ssh`
   - The way to go for MPI applications without offload
2. Explicit offload
   - Functions, global variables require `__attribute__((target(mic)))`
   - Initiate offload, data marshalling with `#pragma offload`
   - Only bitwise-copyable data can be shared
3. Clusters and multiple coprocessors
   - `#pragma offload target(mic:i)`
   - Use threads to offload to multiple coprocessors
   - Run native MPI applications

 © Colfax International, 2013–2016

Optimization Checklist

Areas of code optimization for Intel architecture:

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, 2013–2016

Double Rewards of MIC Programming

From the book “Parallel Programming and Optimization with Intel Xeon Phi Coprocessors” by Colfax:

It is not trivial to achieve good performance with Intel Xeon Phi coprocessors, especially when one compares it to the performance of modern Intel Xeon processors with the Sandy Bridge architecture. The new truth that HPC programmers must learn is: if a parallel code does not perform fast on Intel Xeon Phi coprocessors, it probably is not doing very well on Intel Xeon processors, either. The flip side of this truth is that **when developers invest time and effort into optimizing for the many-core architecture, they also reap performance benefits on multi-core processors**.

Book page: link

© Colfax International, 2013–2016

Colfax International

© Colfax International, 2013–2016

Colfax Complete Xeon Phi Ecosystem

- Servers: xeonphi.com/servers
- Workstations: xeonphi.com/workstations
- 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, 2013–2016

Thank you!

You have remote access until the end of the week.

```
vega@lyra% cowsay 'Have a wonderful journey to the parallel world!'
 _________________________________
/ Have a wonderful journey to the \
\ parallel world!                 /
 ---------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

 © Colfax International, 2013–2016