# PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

One-Day Workshop

*Andrey Vladimirov and Ryo Asai*
*Colfax International — colfaxresearch.com*

October 2016


© Colfax International, 2013–2016

# 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

# COMPUTING APPLICATIONS

## Just some examples

**Computing**

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

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

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

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

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

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

**Environment**
- Weather prediction
- Earthquake simulation
- Pollutant propagation

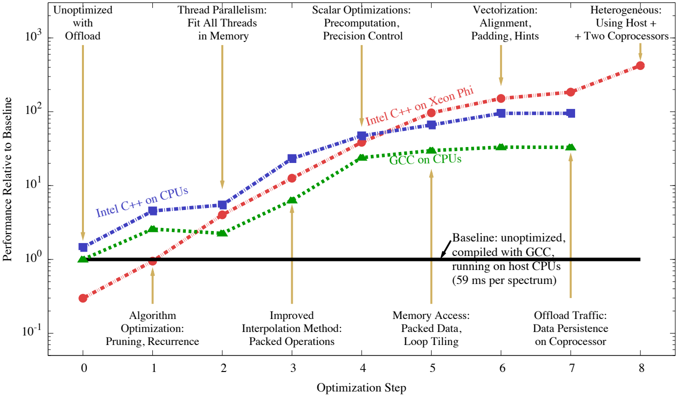
**Energy**
- Oil reservoir simulation
- Climate change science
- Nuclear fusion research



© Colfax International, 2013–2016

# PROGRAMMING MODEL CONTINUITY

Common story for many applications:




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



© Colfax International, 2013–2016

# OPTIMIZATION AREAS

**Scalar Tuning**
what goes on in the pipeline?

**Vectorization**
is SIMD parallelism used well?

**Memory**
is cache usage maximized or RAM access streamlined?

**Threading**
do cores cooperate efficiently?

**Communication**
can coordination in a distributed or heterogeneous system be improved?


© Colfax International, 2013–2016

# §2. PROGRAMMING, OPTIMIZATION BY EXAMPLE

# DIRECT N-BODY SIMULATION

# N-BODY SIMULATION ON CPU AND COPROCESSOR

**N-body simulation on...**

| Two Intel® Xeon® CPUs | One Intel® Xeon Phi™ coprocessor | Two Intel® Xeon Phi™ coprocessors |
|---|---|---|
|  |  |  |

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

Demo: click here


© Colfax International, 2013–2016

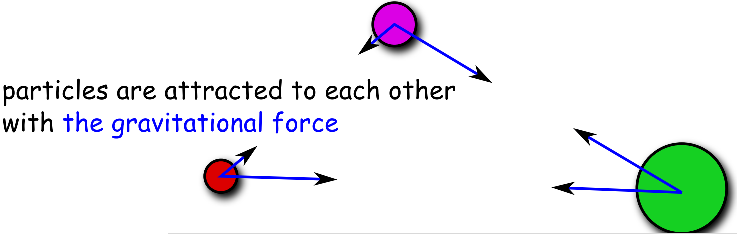
## Gravitational N-body dynamics:

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:

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




 © Colfax International, 2013–2016

# 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

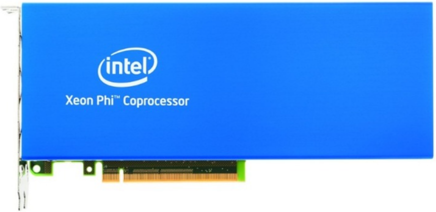
# PARTICLE UPDATE ENGINE

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


© Colfax International, 2013–2016

# INTEL ARCHITECTURE

# COMPUTING PLATFORMS

| Intel Xeon Processor | Intel Xeon Phi Coprocessor, 1st generation | Intel Xeon Phi Processor, 2nd generation* |
| --- | --- | --- |
|  |  | <br>* socket and coprocessor versions |
| Current: Broadwell<br>Upcoming: Skylake | Knights Corner (KNC) | Knights Landing (KNL) |

Multi-Core Architecture

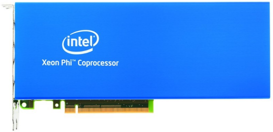
Intel Many Integrated Core (MIC) Architecture



© Colfax International, 2013–2016

# BIRD'S EYE VIEW


- C/C++/Fortran
- Linux/Windows
- ≤3 TiB DDR4
- ≤44 cores (2-way)
- ≈3 GHz
- 2 HT/core
- 256-bit AVX




- C/C++/Fortran
- Special Linux
- ≤ 16 GiB GDDR5
- 57-61 cores
- ≈1.2 GHz
- 4 HW THR/core
- 512-bit IMCI


- C/C++/Fortran
- Linux
- MCDRAM+DDR4
- 64-72 cores
- 1.3-1.5 GHz
- 4 HT/core
- 512-bit AVX-512

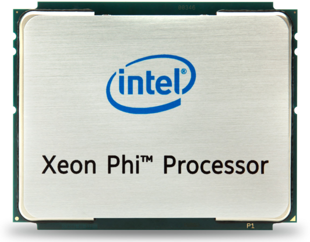

© Colfax International, 2013–2016

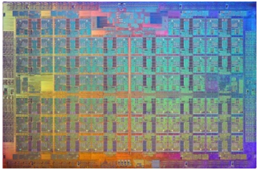
# CORES AND VECTORS

# Intel Xeon Phi Processors (2nd Gen)

Specialized platform for demanding computing applications.

- ▷ Socket version or coprocessor
- ▷ 64-72 cores × 4 HT at 1.3-1.5 GHz
- ▷ 3+ TFLOP/s in DP (FMA)
- ▷ 6+ TFLOP/s in SP (FMA)
- ▷ ≤ 384 GiB DDR4 (> 90 GB/s)
- ▷ 16 GiB HBM (MCDRAM, > 400 GB/s)
- ▷ Binary-compatible with Xeon
- ▷ Common OS (RHEL/CentOS/SUSE/Windows)

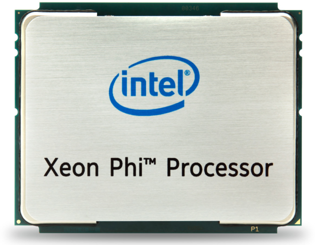


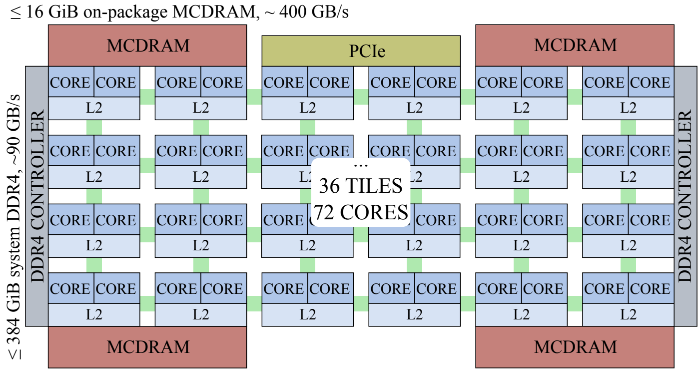




© Colfax International, 2013–2016

# KNL DIE ORGANIZATION

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



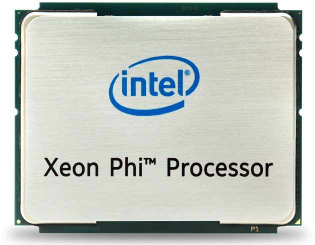



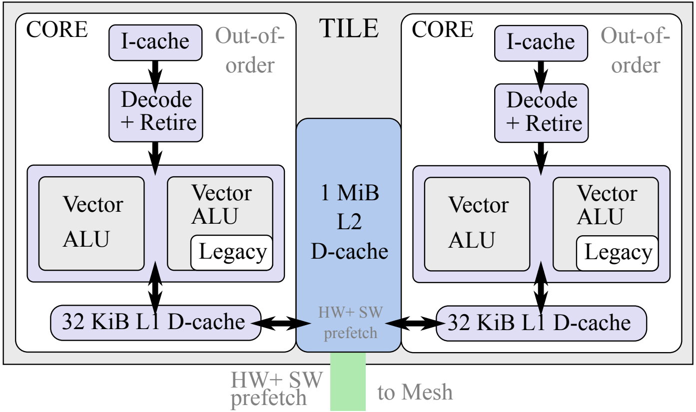


© Colfax International, 2013–2016

# KNL CORES

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





© Colfax International, 2013–2016

# INCORPORATING THREAD PARALLELISM

Before:

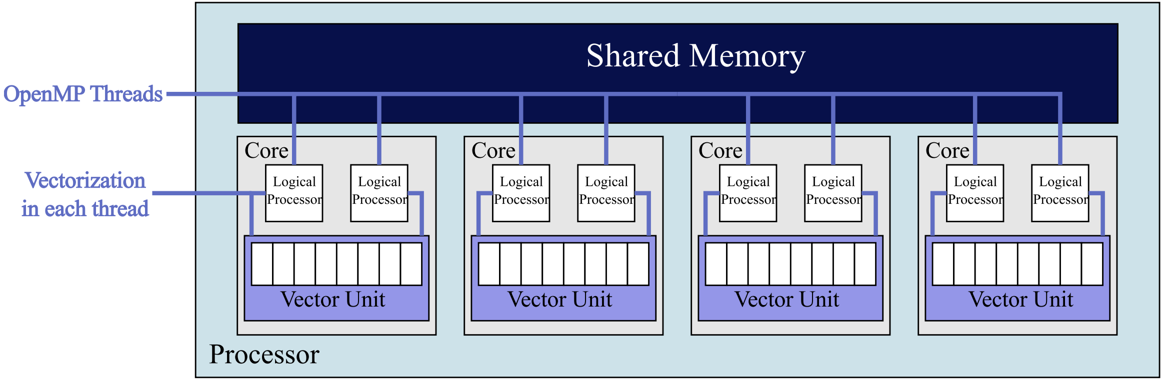
```
  for (int i = 0; i < nParticles; i++) {   // Particles that experience force
    float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
    for (int j = 0; j < nParticles; j++) { // Particles that exert force
      // Newton's law of universal gravity
      ...
```

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

# CO-EXISTENCE WITH VECTORS




**Utilize cores**: run multiple threads/processes (MIMD)

**Utilize vectors**: each thread (process) issues vector instructions (SIMD)


© Colfax International, 2013–2016

# SIMULTANEOUS THREADING AND VECTORIZATION

This approach often works:

```
#pragma omp parallel for
for (int i = 0; i < n; i++) // Thread parallelism in outer loop
#pragma simd
  for (int j = 0; j < m; j++) // Vectorization in inner loop
    DoSomeWork(A[i][j]);
```

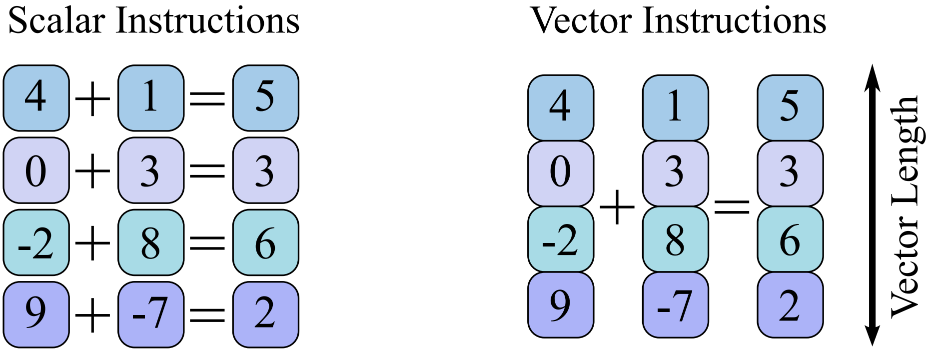
That works as well:

```
#pragma omp parallel for simd
for (int i = 0; i < n; i++) // If the problem is all data-parallel
    DoSomeWork(A[i]);
```


© Colfax International, 2013–2016

# SHORT VECTOR SUPPORT

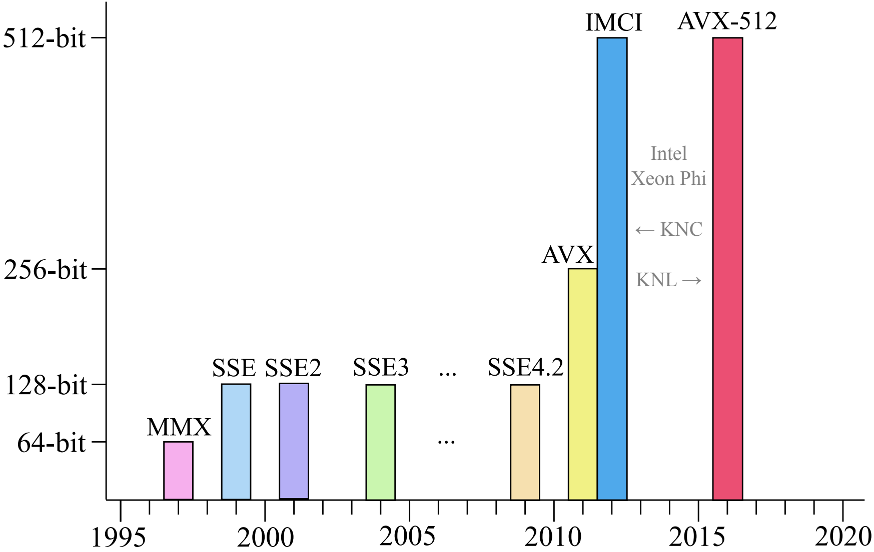
Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.





© Colfax International, 2013–2016

# INSTRUCTION SETS IN INTEL ARCHITECTURE





© Colfax International, 2013–2016

# VECTORIZING WITH UNIT-STRIDE MEMORY ACCESS

Before:

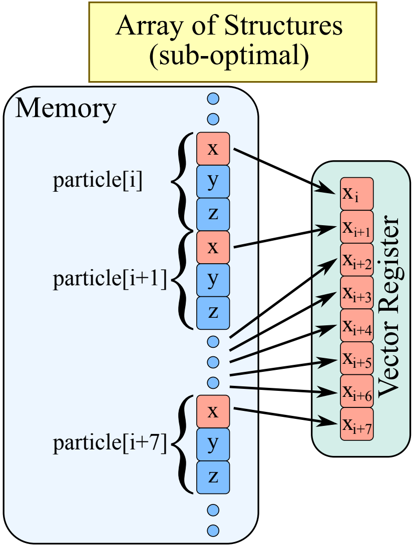
```
struct ParticleType {
  float x, y, z, vx, vy, vz;
};  // ...
      const float dx = particle[j].x - particle[i].x;
      const float dy = particle[j].y - particle[i].y;
      const float dz = particle[j].z - particle[i].z;
```

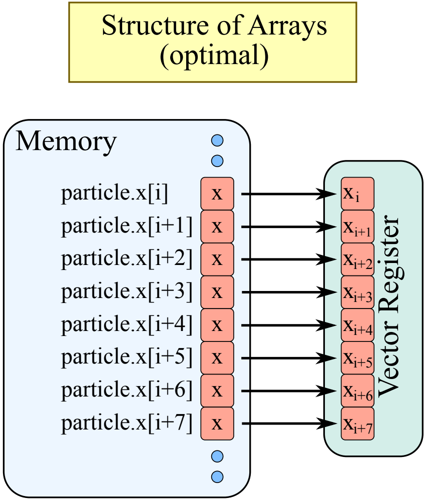
After:

```
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

# WHY AOS TO SOA CONVERSION HELPS: UNIT STRIDE





© Colfax International, 2013–2016

# IMPROVING SCALAR EXPRESSIONS

Before:

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

After:

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

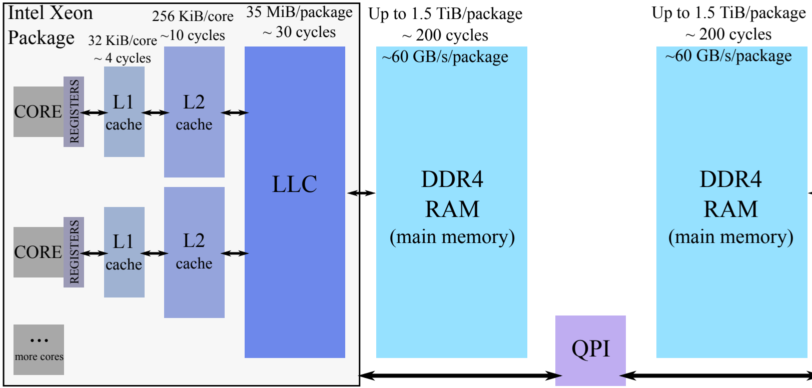
- ▷ Strength reduction (division → multiplication by reciprocal)
- ▷ Precision control (suffix `-f` on single-precision constants and functions)
- ▷ Reliance on hardware-supported reciprocal square root


© Colfax International, 2013–2016

# MEMORY ORGANIZATION

# INTEL XEON CPU: MEMORY ORGANIZATION

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

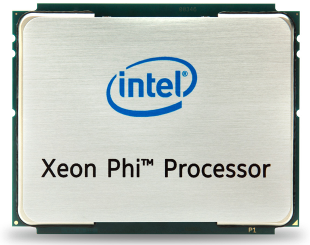



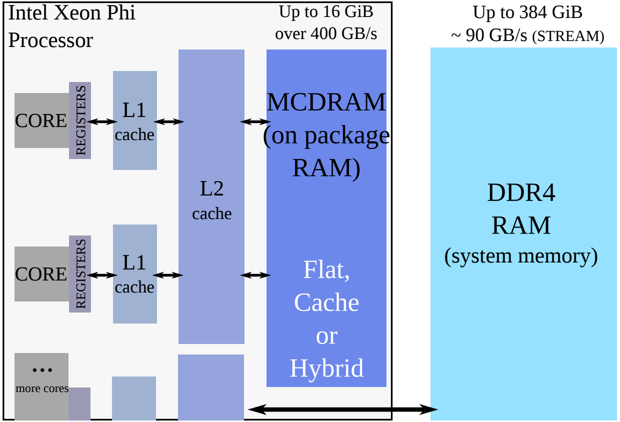


© Colfax International, 2013–2016

# KNL MEMORY ORGANIZATION (BOOTABLE)

- Direct access to on-platform RAM and on-package HBM
- Use HBM as cache, in flat mode, or as hybrid



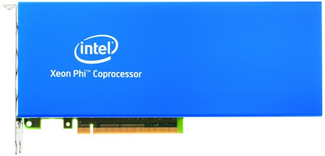



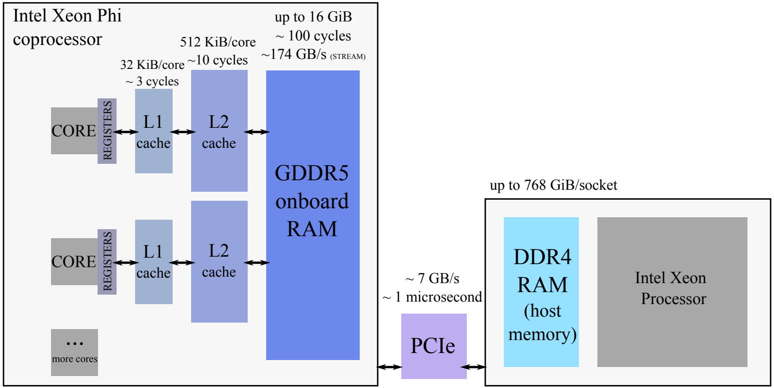


© Colfax International, 2013–2016

# KNC MEMORY ORGANIZATION

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



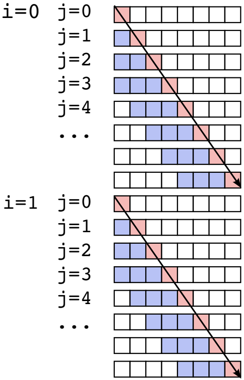



© Colfax International, 2013–2016

# LOOP TILING

**Original:**

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



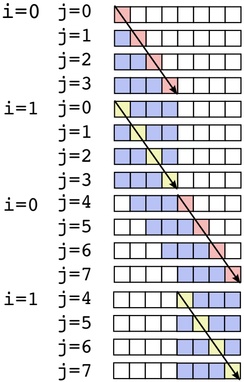

- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

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

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

**Tiled:**

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






© Colfax International, 2013–2016

# 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;
```

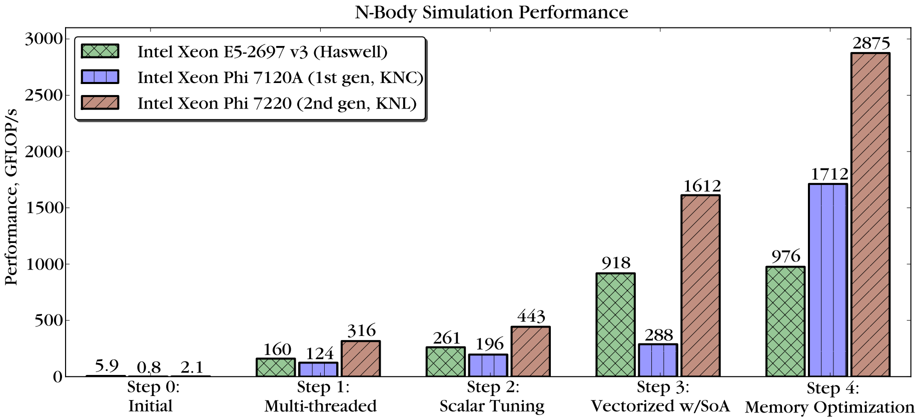
After: (tileSize = 16)

```
  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;
```



© Colfax International, 2013–2016

# IMPACT OF CODE OPTIMIZATION



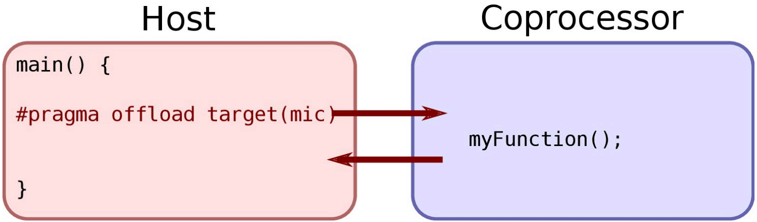

Contributed as Chapter 23 in “Intel Xeon Phi Processor High Performance Programming, Knights Landing Edition” (2016)


© Colfax International, 2013–2016

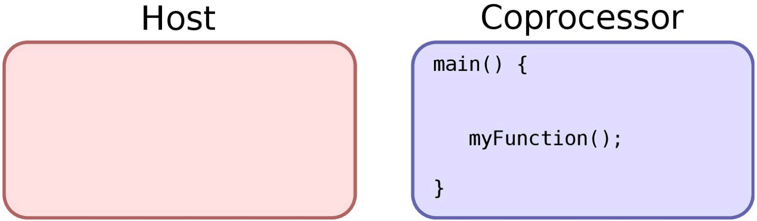
# COPROCESSORS AND CLUSTERS

# OFFLOAD AND NATIVE MODELS

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




- Native model (standalone application/MPI process):





© Colfax International, 2013–2016

"Hello World" application:

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

Compile and run on host CPU:

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



© Colfax International, 2013–2016

# NATIVE EXECUTION ON AN INTEL XEON PHI COPROCESSOR (KNC)

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

```
vega@lyra% icpc hello.cc -mmic # Cross-compile
vega@lyra% scp a.out mic0:~/ # Put executable on coprocessor
a.out 100% 10KB 10.4KB/s 00:00
vega@lyra% ssh mic0 # Log in to coprocessor
vega@mic0% pwd
/home/lyra
vega@mic0% ls
a.out
vega@mic0% ./a.out # Launch application
Hello world! I have 244 logical processors.
vega@mic0%
```

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


© Colfax International, 2013–2016

# 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: PRAGMA-BASED APPROACH

"Hello World" in the explicit offload model:

```
#include <cstdio>
int main() {
    printf("Hello World from host!\n");
#pragma offload target(mic)
    {
        printf("Hello World from coprocessor!\n"); fflush(stdout);
    }
    printf("Bye\n");
}
```

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

Detailed syntax in the Intel C++ Compiler Reference.


© Colfax International, 2013–2016

# COMPILING AND RUNNING AN OFFLOAD APPLICATION

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

- No additional arguments (for Intel compiler)
- Launch on host as a regular application
- Code inside of `#pragma offload` is offloaded automatically
- Console output on coprocessor buffered, mirrored to the host
- If no coprocessor available, default behavior is error; may be overridden to 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

# 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

```
#pragma offload target(mic) optional
{
  printf("Hello World! I have %d logical processors.\n",
  sysconf(_SC_NPROCESSORS_ONLN )); fflush(stdout);
}
```

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


© Colfax International, 2013–2016

# 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.


© Colfax International, 2013–2016

# CLAUSES OF PRAGMA OMP TARGET

```
#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.


© Colfax International, 2013–2016

# SHARED VIRTUAL MEMORY MODEL

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

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

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

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



© Colfax International, 2013–2016

# 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

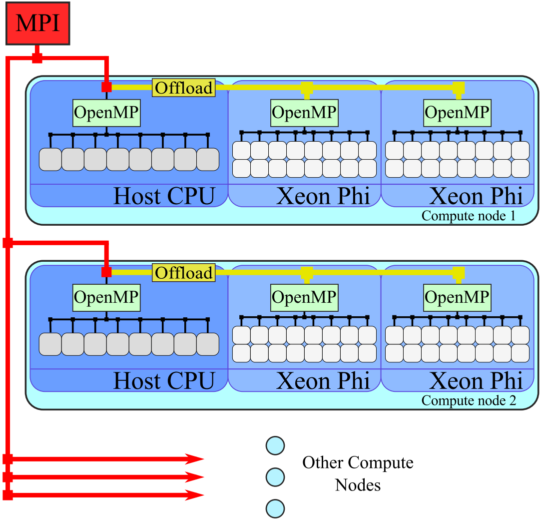


© Colfax International, 2013–2016

# 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.



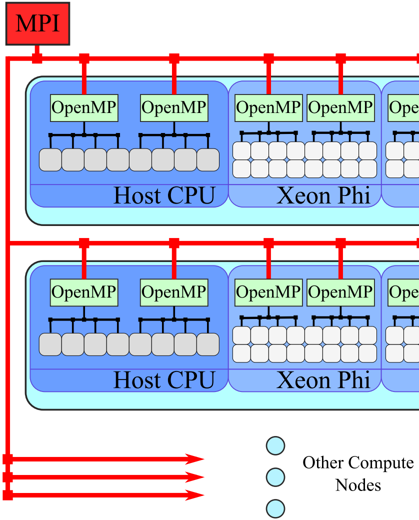


© Colfax International, 2013–2016

# HETEROGENEOUS DISTRIBUTED COMPUTING WITH XEON PHI

**Option 2: Symmetric hybrid MPI+OpenMP.**

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

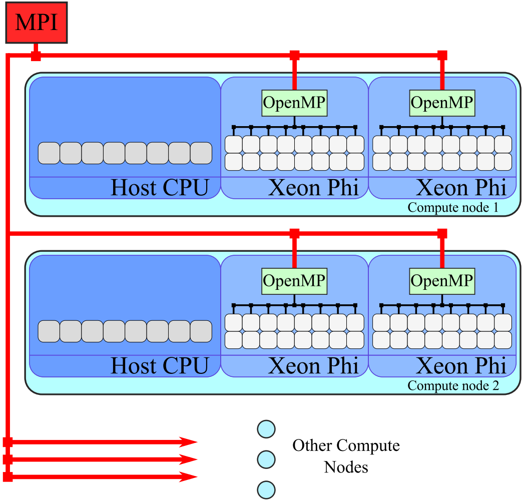




© Colfax International, 2013–2016

# SCALING ACROSS A CLUSTER WITH COPROCESSORS WITH MPI

- MPI processes only on CPUs
- Divide data between coprocessors
- Concurrent offload from multiple host threads
- Synchronize data between nodes with MPI





© Colfax International, 2013–2016

# CORE OF MPI-ONLY IMPLEMENTATION

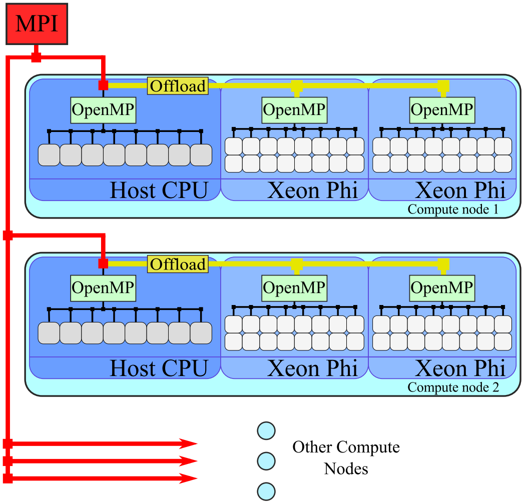
Simple: all particles on each compute node; exchange updated particle coordinates.

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


© Colfax International, 2013–2016

# SCALING ACROSS A CLUSTER WITH COPROCESSORS

- MPI processes only on CPUs
- Divide data between coprocessors
- Concurrent offload from multiple host threads
- Synchronize data between nodes with MPI





© Colfax International, 2013–2016

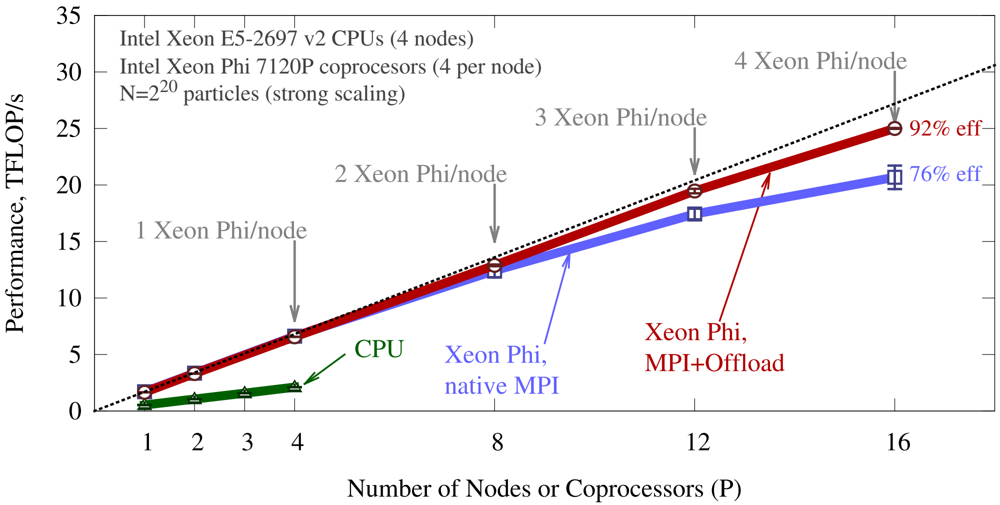
# MPI WITH OFFLOAD IMPLEMENTATION

```
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

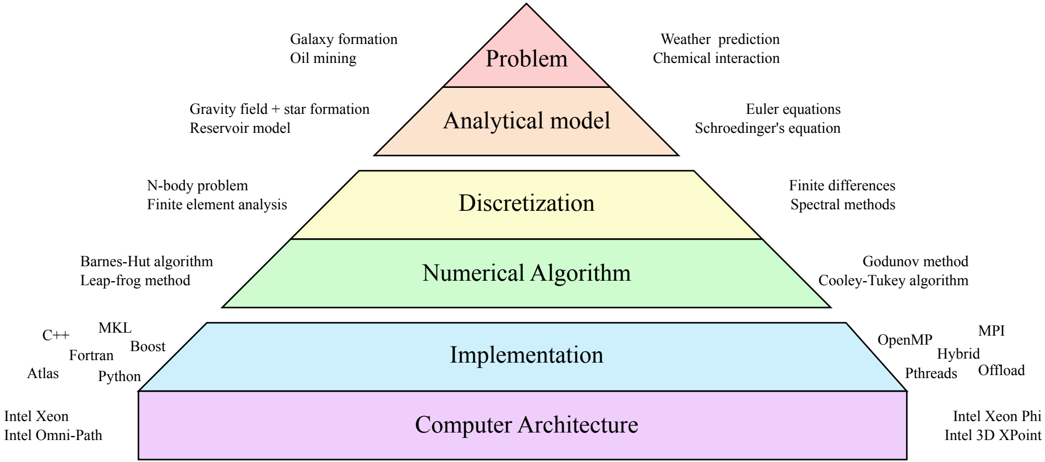




© Colfax International, 2013–2016

# §3. OPTIMIZATION POINTERS

# 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

# SCALAR TUNING

# 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% icpc -o mycode -O3 source.cc
```

For a specific function:

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


© 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

```
// 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);
```

```
// 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

# MOVE BRANCHES OUTSIDE OF LOOPS

```
// Elegant, but bad for performance
for (i = 0; i < n; i++) {
  if (i == 0) {
    // Absorbing boundary
    B[i] = 0.0;
  } else if (i == n - 1) {
    // Injection at boundary
    B[i] = B[i] + 1.0;
  } else {
    // Diffusion between boundaries
    B[i] = 0.25*(A[i-1] +
                 2.0*A[i] + A[i+1]);
  }
}
```

```
// Moving branches out of loops


// Absorbing boundary
B[i] = 0.0;

for (i = 1; i < n - 1; i++) {
  // Diffusion between boundaries
  B[i] = 0.25*(A[i-1] + 2.0*A[i] +
                            A[i+1]);
}

// Injection at boundary
B[n-1] = B[n-1] + 1.0;
```



© Colfax International, 2013–2016

# REDUNDANT CODE IS OK

```
// Elegant, but bad for performance
for (ii = 0; ii < n; ii+=16) {
  for (i = ii; i < ii+16; i++)
    // Branch causes unnecessary
    // masking of vector iterations
    if (i < n) {
      A[k*n + i] = ...
    }
}
```

```
// Redundant code, but faster
const int nTrunc = n - 16;
for (ii = 0; ii < nTrunc; ii+=16) {
  for (i = ii; i < ii+16; i++)
      A[k*n + i] = ...

for (i = nTrunc; i < n; i++)
  A[k*n + i] = ...
}
```



© Colfax International, 2013–2016

# VECTORIZATION

# AUTOMATIC VECTORIZATION OF LOOPS

```
#include <cstdio>

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
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

# 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

# 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).


© Colfax International, 2013–2016

# 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.


© Colfax International, 2013–2016

# AUTO-VECTORIZED LOOPS MAY BE COMPLEX (EXAMPLE 1)

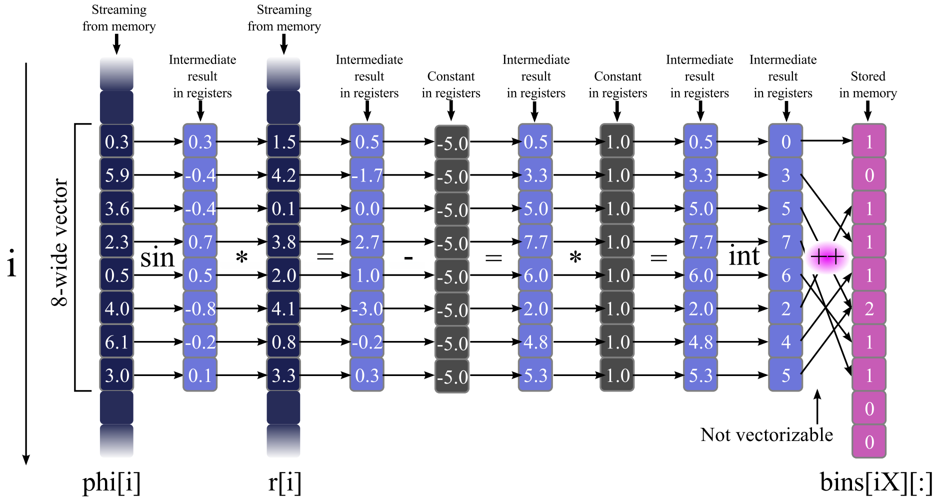
```
for (int i = ii; i < ii + tileSize; i++) { // Target for auto-vectorization

  // 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



© Colfax International, 2013–2016

# AUTO-VECTORIZED LOOPS MAY BE COMPLEX (EXAMPLE 2)




See this paper for more details


© 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
```

```
void mycopy(int n,
        float* a, float* b) {
#pragma ivdep
  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
...
LOOP BEGIN at vdep.cc(4,1)
<Multiversioned v2>
remark #15300: LOOP WAS VECTORIZED
LOOP END
```



© Colfax International, 2013–2016

# 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]`).



© Colfax International, 2013–2016

# UNIT-STRIDE ACCESS

Unit-stride access is optimal:

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

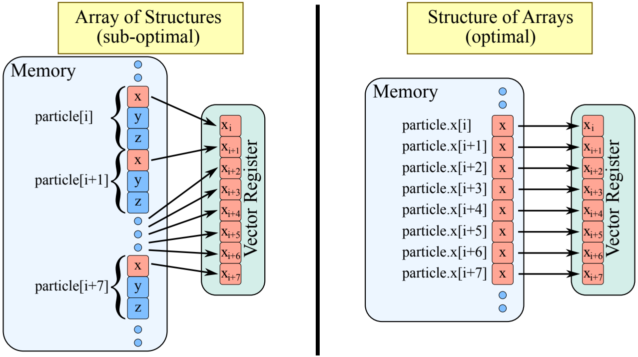
Non-unit stride is slower:

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

Stochastic access may be vectorized (but not efficient):

```
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:





© 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) |

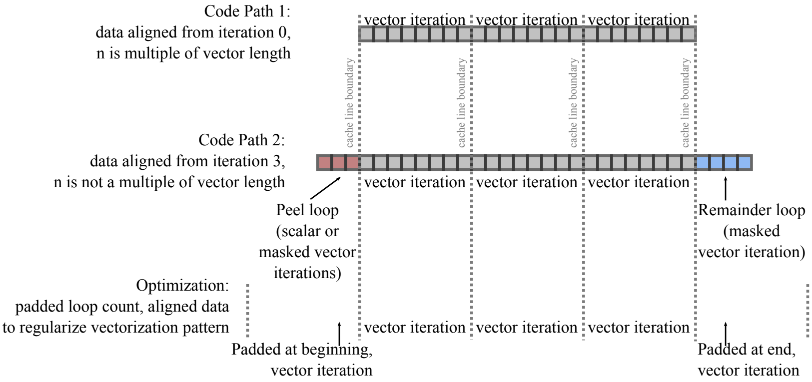
Why align: speed up vector load/stores, avoid false sharing (see Session 7), accelerate RDMA.


© Colfax International, 2013–2016

# WHAT HAPPENS WITHOUT ALIGNMENT

Compiler may implement peel and remainder loops:

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






© Colfax International, 2013–2016

# CREATING ALIGNED DATA CONTAINERS

- Data alignment on the stack

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

- Data alignment on the heap

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

- `A[0]` is aligned on a 64-byte boundary.
- Very high alignment value may lead to wasted virtual memory.
- Fortran: directive or compiler argument `-align array64byte`



© Colfax International, 2013–2016

# Padding Multi-Dimensional Containers 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.

Incorrect:

```
// A - matrix of size (n x n)
// n is not a multiple of 16
float* A =
 _mm_malloc(sizeof(float)*n*n, 64);

for (int i = 0; i < n; i++)
  // A[i*n + 0] may be unaligned
  for (int j = 0; j < n; j++)
    A[i*n + j] = ...
```

Correct:

```
// ... Padding inner dimension
int lda=n + (16-n%16); // lda%16==0
float* A =
 _mm_malloc(sizeof(float)*n*lda, 64);

for (int i = 0; i < n; i++)
  // A[i*lda + 0] aligned for any i
  for (int j = 0; j < n; j++)
    A[i*lda + j] = ...
```


© 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]`


© Colfax International, 2013–2016

# MULTI-THREADING

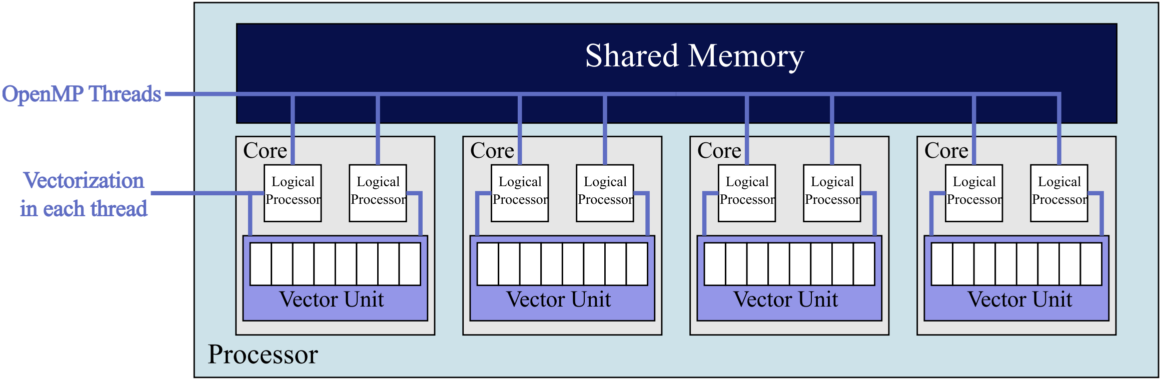
# "Hello World" OpenMP Program

```
#include <omp.h>
#include <cstdio>

int main(){
  // This code is executed by only 1 thread
  const int nt=omp_get_max_threads();
  printf("OpenMP with %d threads\n", nt);

#pragma omp parallel
  {
    // This code is executed in parallel
    // by multiple threads
    printf("Hello World from thread %d\n",
                                  omp_get_thread_num());
  }
}
```

- OpenMP = "Open Multi-Processing" = computing-oriented framework for shared-memory programming
- Threads – streams of instructions that share memory address space
- Distribute threads across CPU cores for parallel speedup


© Colfax International, 2013–2016

# CO-EXISTENCE WITH VECTORS



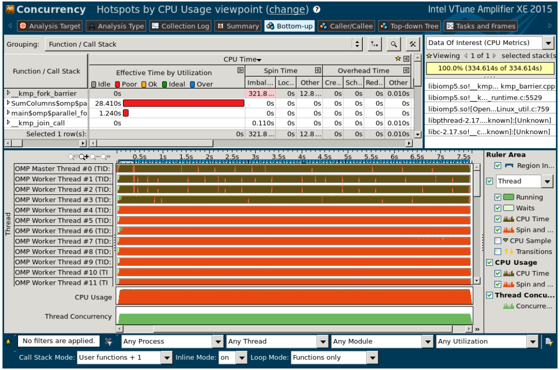

**Utilize cores**: run multiple threads/processes (MIMD)

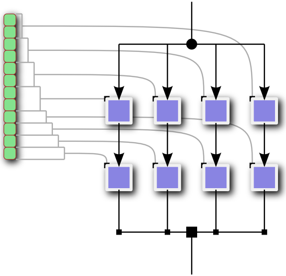
**Utilize vectors**: each thread (process) issues vector instructions (SIMD)


© Colfax International, 2013–2016

# INSUFFICIENT PARALLELISM

## Analysis in Intel VTune Amplifier XE



▷ Occurs when there are not enough iterations or parallel work-items exposed to the parallel loop in OpenMP.


© Colfax International, 2013–2016

# EXAMPLE: DEALING WITH INSUFFICIENT PARALLELISM

$$S_i = \sum_{j=0}^{n} M_{ij}, \; i = 0 \ldots m. \tag{1}$$

- `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 total=0;
#pragma vector aligned
    for (int j=0; j<n; j++) // n=100000000
      total+=M[i*n+j];
  s[i]=total; }}
```

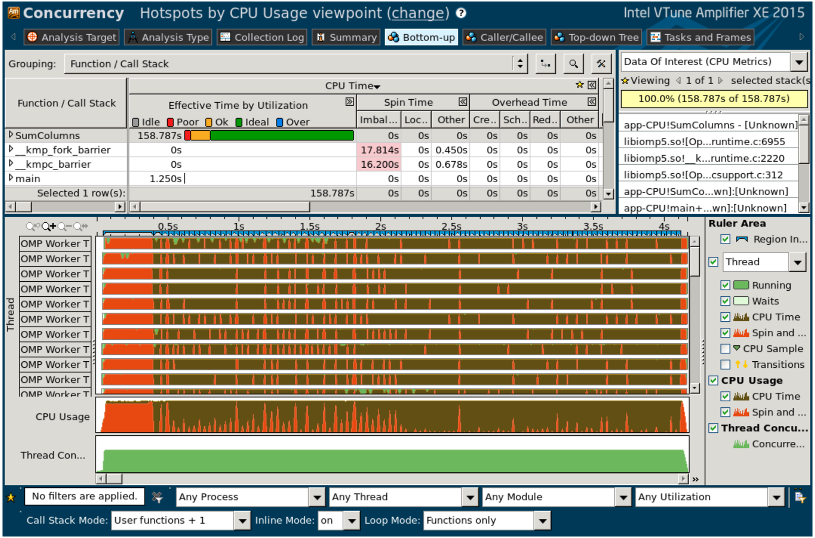

© Colfax International, 2013–2016

# 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 total[m];   total[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++)
          total[i]+=M[i*n+j];
    for (int i=0; i<m; i++)          // Reduction
#pragma omp atomic
      s[i]+=total[i];
}   }
```


© Colfax International, 2013–2016

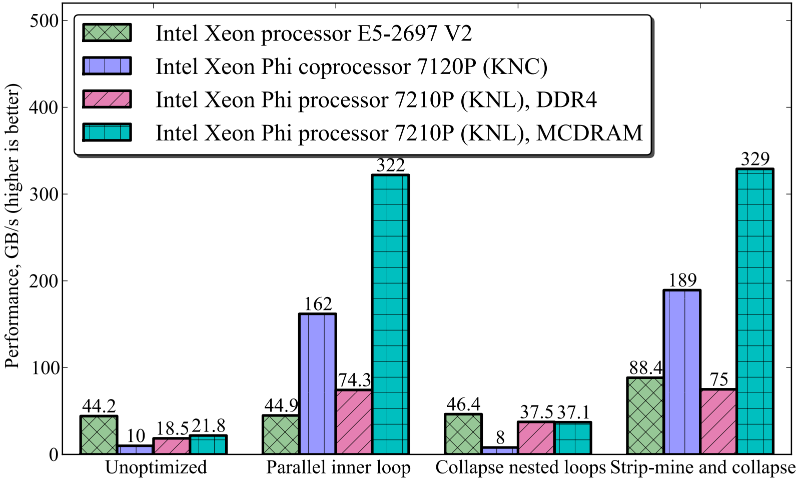
# EXPOSING PARALLELISM: STRIP-MINING AND LOOP COLLAPSE





© Colfax International, 2013–2016

# DEALING WITH INSUFFICIENT PARALLELISM



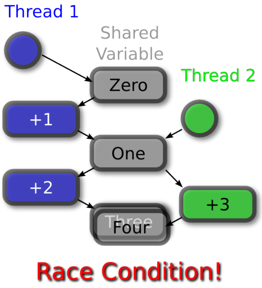



© Colfax International, 2013–2016

# RACE CONDITIONS AND UNPREDICTABLE PROGRAM BEHAVIOR

```
#include <omp.h>
#include <cstdio>
int main() {
  const int n = 1000;
  int total = 0;
#pragma omp parallel for
  for (int i = 0; i < n; i++) {
    total = total + i; // Race condition
  }
  printf("total=%d (must be %d)\n", total,
                                 ((n-1)*n)/2);
}
```

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



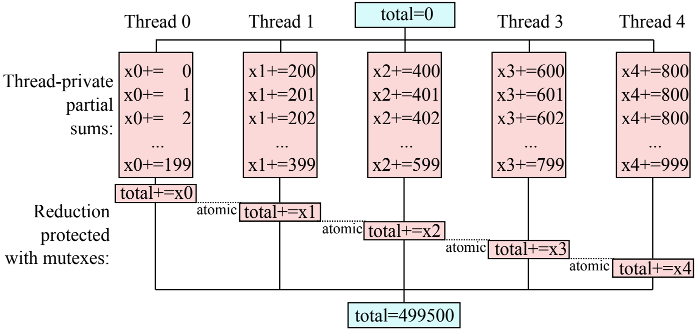

- Occurs when 2 or more threads access the same memory address, and at least one of these accesses is for writing


© Colfax International, 2013–2016

# 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;

}
```





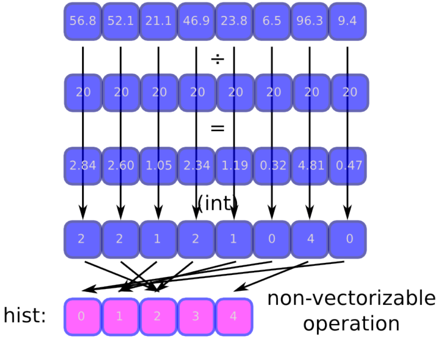
© Colfax International, 2013–2016

# EXAMPLE: BINNING PROBLEM

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,
  const float group_width) {
    for (int i = 0; i < n; i++) {
      const int j = int(age[i]/group_width);
      hist[j]++;
    }
}
```

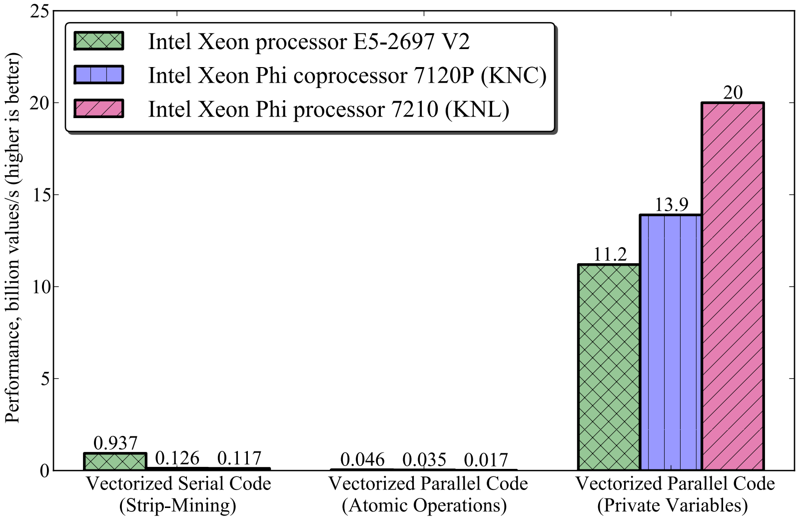
- Vector dependence in `hist[j]++`
- Strip-mine or use conflict detection






© Colfax International, 2013–2016

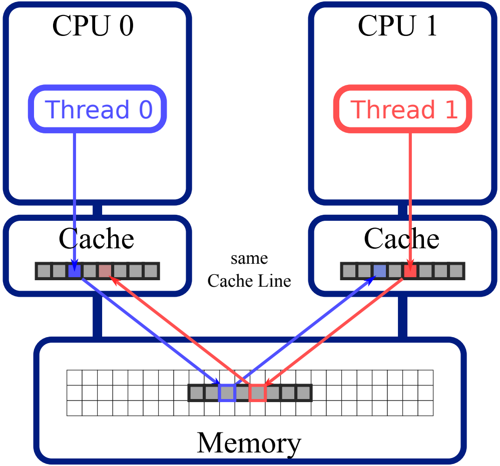
# USING REDUCTION INSTEAD OF SYNCHRONIZATION





© Colfax International, 2013–2016

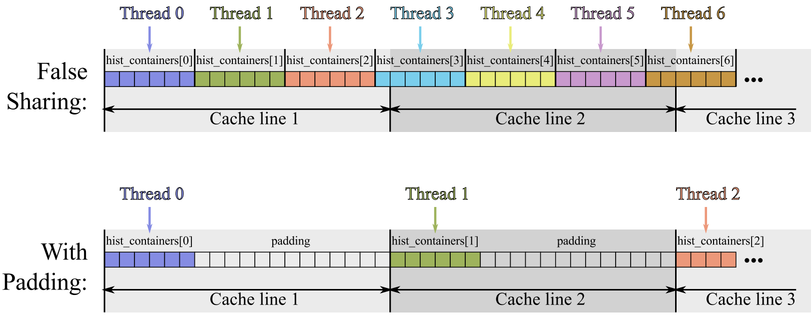
# FALSE SHARING. DATA PADDING AND PRIVATE VARIABLES




- Occurs when 2 or more threads acess the same cache line, and at least one of the accesses is for writing
- Caused by *coherent caches*
- Cache line is 64-byte wide (in modern Intel architectures)


© Colfax International, 2013–2016

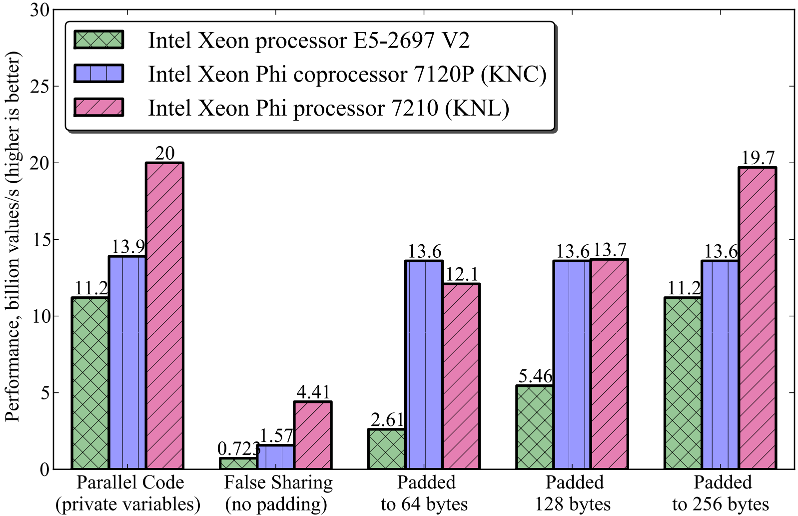
# PADDING TO AVOID FALSE SHARING




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


© Colfax International, 2013–2016

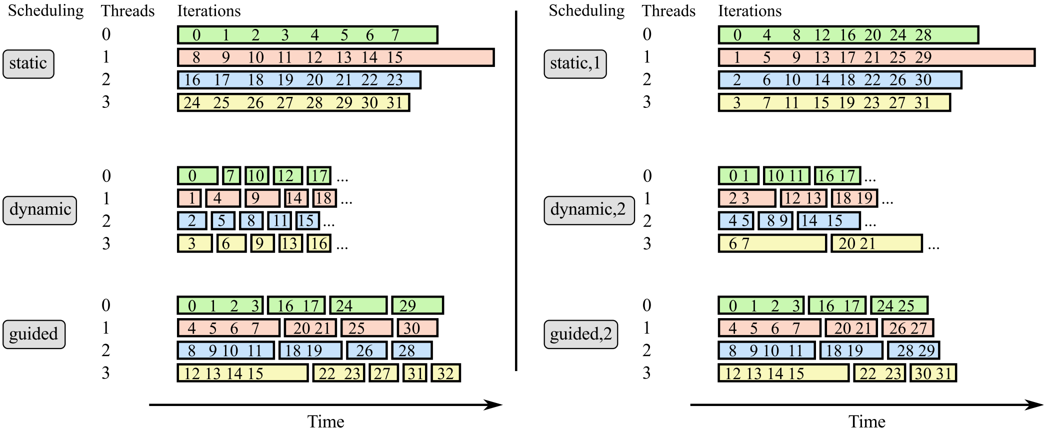
# PADDING TO AVOID FALSE SHARING






© Colfax International, 2013–2016

# LOOP SCHEDULING MODES IN OPENMP

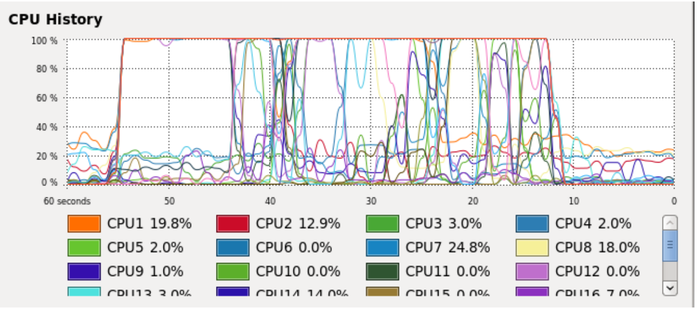




© Colfax International, 2013–2016

# What is Thread Affinity

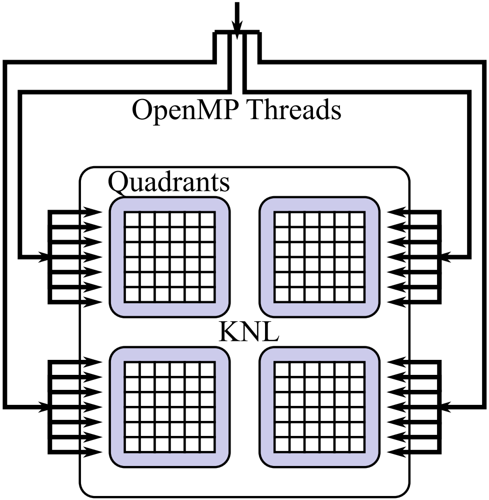
- OpenMP threads may migrate between cores
- Forbid migration — improve locality — increase performance
- Affinity patterns “scatter” and “compact” may improve cache sharing, relieve thread contention





© Colfax International, 2013–2016

# NESTED PARALLELISM WITH OPENMP




```
#pragma omp parallel
{
#pragma omp parallel
  {
    // ...
  }
}
```

- Tune granularity of parallelism
- Improve resource sharing in NUMA systems


© Colfax International, 2013–2016

# CACHE AND MEMORY ACCESS

# 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



© Colfax International, 2013–2016

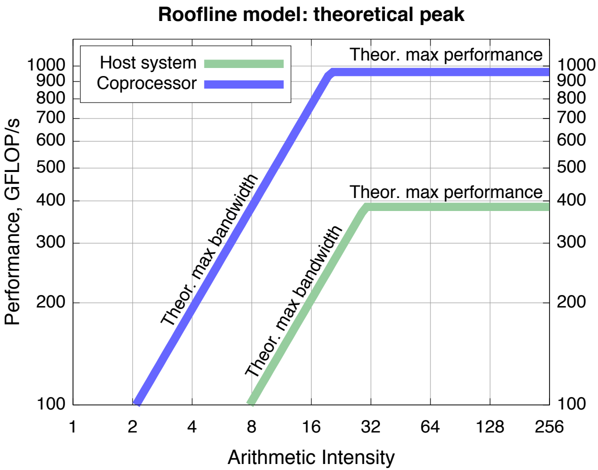
# ON COMPUTATIONAL COMPLEXITY OF ALGORITHMS

| 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 $\alpha > 1$. 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


© Colfax International, 2013–2016

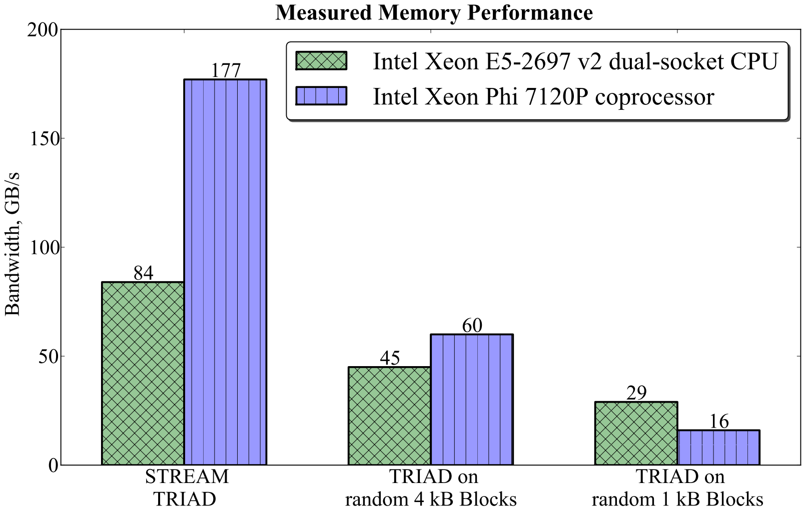
# ARITHMETIC INTENSITY AND ROOFLINE MODEL




More on roofline model: Williams et al.


© Colfax International, 2013–2016

# STREAMING VERSUS RANDOM ACCESS

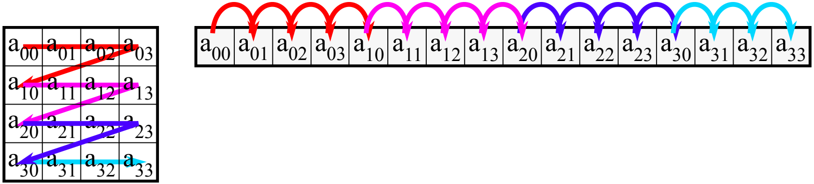


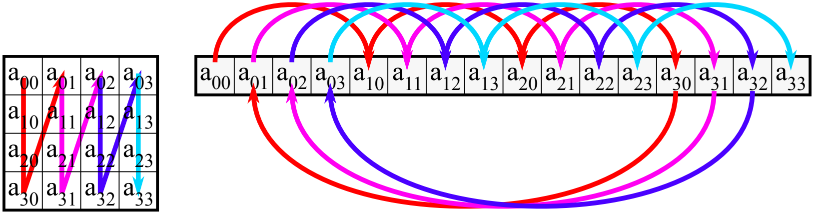


© Colfax International, 2013–2016

# PRINCIPLE

Choose loop order to maintain unit-stride memory access




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



© Colfax International, 2013–2016

# EXAMPLE: OVER-SIMPLIFIED MATRIX-MATRIX MULTIPLICATION

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

Before:

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

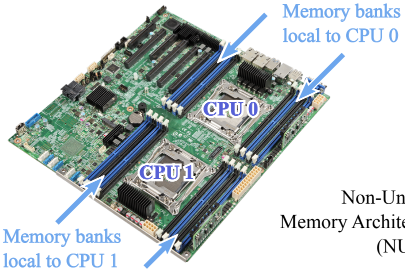
After:

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


© Colfax International, 2013–2016

# NUMA ARCHITECTURES

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




Non-Uniform Memory Architecture (NUMA)


Examples:

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


© Colfax International, 2013–2016

# ALLOCATION ON FIRST TOUCH

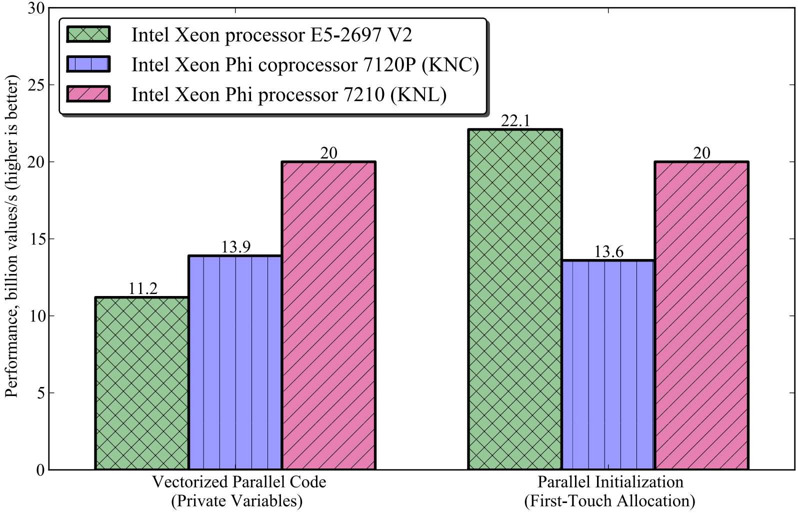
- Memory allocation occurs not during `_mm_malloc()`, but upon the first write to the buffer ("first touch")
- Default NUMA allocation policy is "on first touch"
- For better performance in NUMA systems, initialize data with the same parallel pattern as during data usage

```
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

# FIRST-TOUCH ALLOCATION POLICY






© 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).

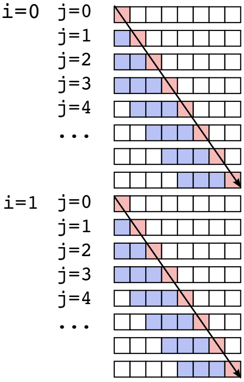


© Colfax International, 2013–2016

# LOOP TILING

**Original:**

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




- cached, LRU eviction policy

- cache miss (read from memory, slow)

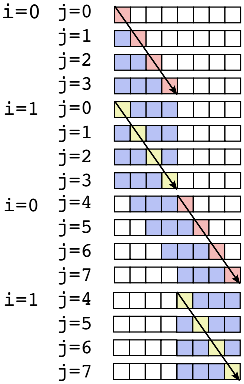
- cache hit (read from cache, fast)

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

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

**Tiled:**

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






© Colfax International, 2013–2016

# LOOP TILING (CACHE BLOCKING) -- PROCEDURE

```
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 inner loop
for (int i = 0; i < m; i++)
  for (int jj = 0; jj < n; jj += TILE)
    for (int j = jj; j < jj + TILE; j++)
      compute(a[i], b[j]); // Same order of operation as original
```

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


© Colfax International, 2013–2016

# EXAMPLE: MATRIX-VECTOR MULTIPLICATION

$$c_i = \sum_{j=0}^{n} A_{ij} b_j, \quad i = 0, 1, \ldots, (m-1). \tag{2}$$

```
void Multiply(const double* const A, const double* const b,
              double* const c, const long n, const long m){
  assert(n%64 == 0);
#pragma omp parallel for
  for (long i = 0; i < m; i++)
#pragma vector aligned
    for (long j = 0; j < n; j++) // Each value of A[i*n+j] is used only once
      c[i] += A[i*n+j] * b[j];   // Each value of b[j] is used a total of m times
}
```

Non-optimal performance due to inefficient cache use


© Colfax International, 2013–2016

# APPLYING TILING

```
  const long jTile = 4096L; assert(n%jTile == 0);
#pragma omp parallel
  {
    double temp_c[m] __attribute__((aligned(64)));
    temp_c[:] =0;
#pragma omp for
    for (long jj =0; jj < n; jj+=jTile) // Loop Tiling in j
      for (long i = 0; i < m; i++)
#pragma vector aligned
        for (long j =jj; j < jj+jTile; j++)
          temp_c[i] += A[i*n+j] * b[j];

    for(long i = 0; i < m; i++) { // Reduction
#pragma omp atomic
      c[i]+= temp_c[i];
} } }
```



© Colfax International, 2013–2016

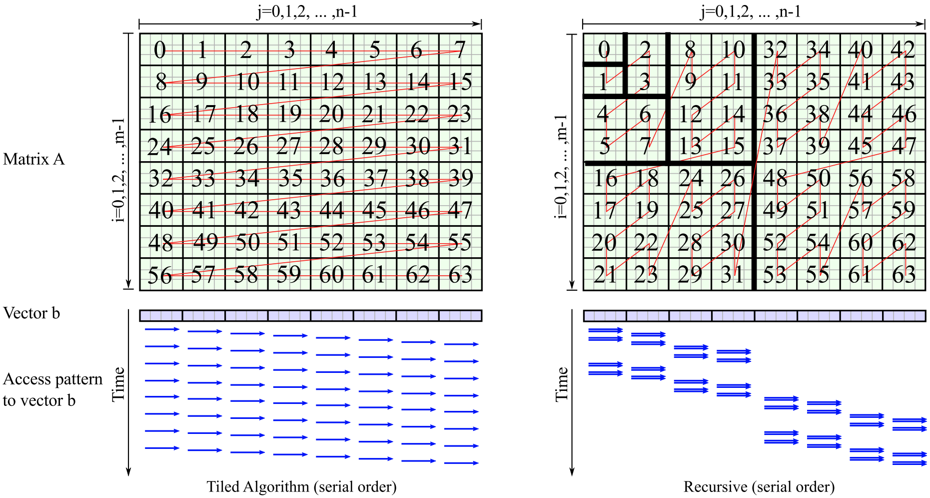
# CACHE BLOCKING + STRIP-MINE AND COLLAPSE

```
  const long iTile = 64L;   assert(m%iTile == 0);
  const long jTile = 4096L; assert(n%jTile == 0);
#pragma omp parallel
  {
    double temp_c[m] __attribute__((aligned(64))); temp_c[:] =0;
#pragma omp for collapse(2)
    for (long ii = 0; ii < m; ii += iTile)
      for (long jj = 0; jj < n; jj += jTile)
        for (long i = ii; i < ii+iTile; i++)
#pragma vector aligned
          for (long j =jj; j < jj+jTile; j++)
            temp_c[i] += A[i*n+j] * b[j];

    for(long i = 0; i < m; i++) {
#pragma omp atomic
      c[i]+= temp_c[i];
} } }
```



© Colfax International, 2013–2016

# PRINCIPLE





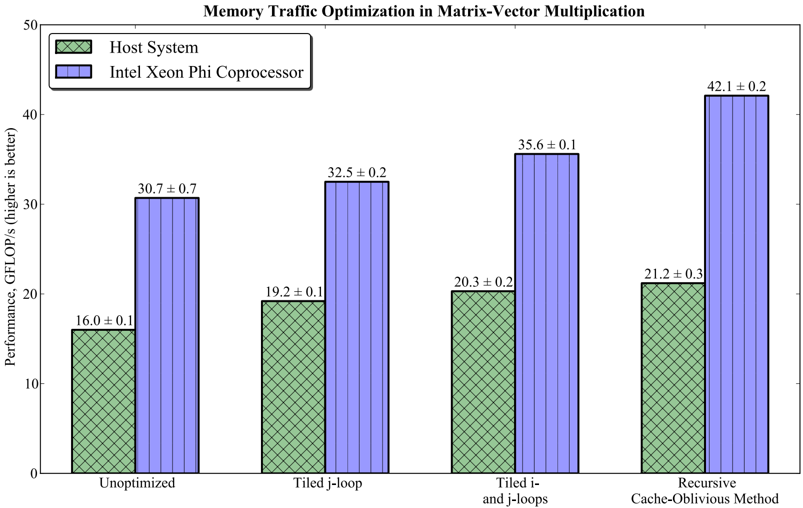

© Colfax International, 2013–2016

# EXAMPLE: MATRIX-VECTOR MULTIPLICATION

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


© Colfax International, 2013–2016

# PERFORMANCE OF MATRIX VECTOR MULTIPLICATION






© Colfax International, 2013–2016

# COMMUNICATION CONTROL

# STRUCTURE OF MPI APPLICATIONS: HELLO WORLD

```
#include "mpi.h"
#include <cstdio>
int main (int argc, char *argv[]) {
  MPI_Init (&argc, &argv);  // Initialize MPI envirnmnt
  int rank, size, namelen;
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Comm_rank (MPI_COMM_WORLD, &rank); // ID of current process
  MPI_Get_processor_name (name, &namelen); // Hostname of node
  MPI_Comm_size (MPI_COMM_WORLD, &size); // Number of processes
  printf ("Hello World from rank %d running on %s!\n", rank, name);
  if (rank == 0) printf("MPI World size = %d processes\n", size);
  MPI_Finalize (); // Terminate MPI environment
}
```


© Colfax International, 2013–2016

# 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`


© Colfax International, 2013–2016

# 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

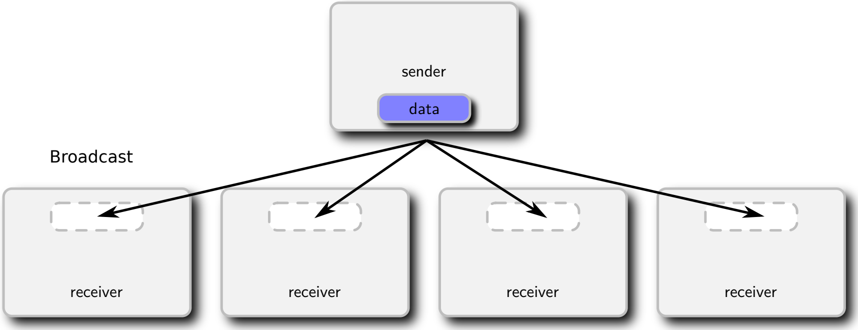
```
if (rank == sender) {

 char outgoingMsg[messageLength];
 strcpy(outgoingMsg, "Hi There!");
 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

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

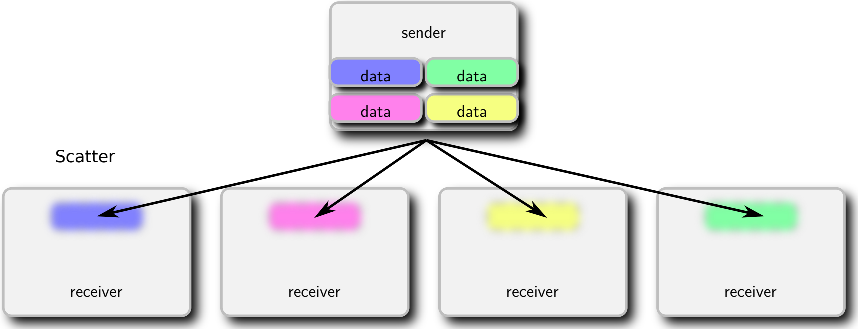




© Colfax International, 2013–2016

# COLLECTIVE COMMUNICATION: SCATTER

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

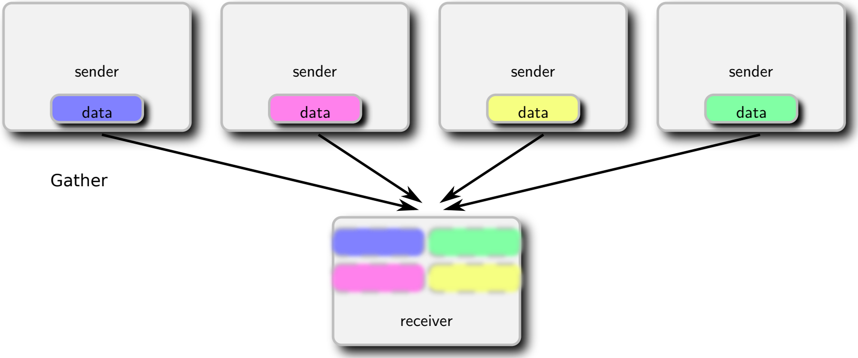




© Colfax International, 2013–2016

# COLLECTIVE COMMUNICATION: GATHER

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

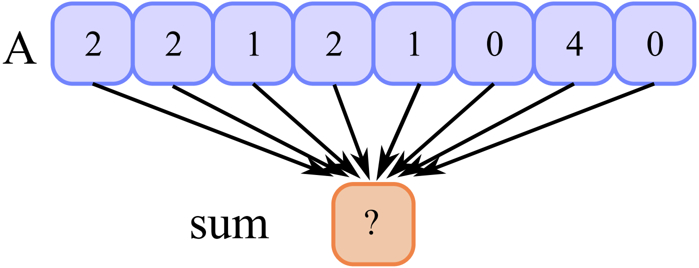




© Colfax International, 2013–2016

# COLLECTIVE COMMUNICATION: REDUCTION

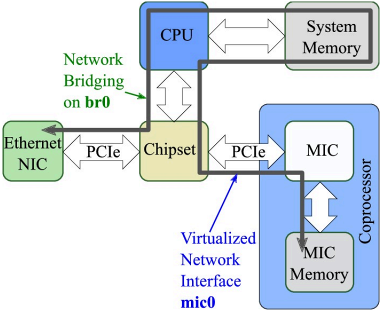
```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
    MPI_Op op, int root, MPI_Comm comm);
```

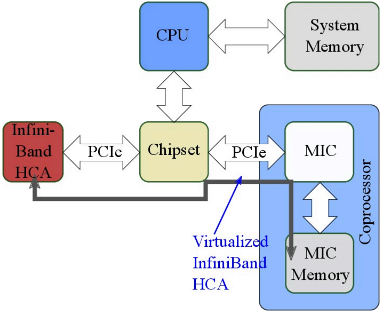



Available reducers: max/min, minloc/maxloc, sum, product, AND, OR, XOR (logical or bitwise).


© Colfax International, 2013–2016

# PEER-TO-PEER COMMUNICATION BETWEEN COPROCESSORS



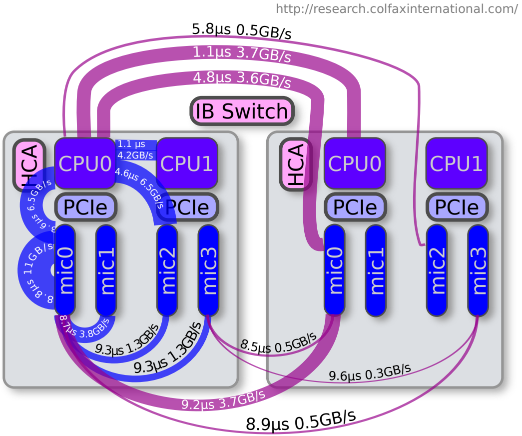

- Left: Gigabit Ethernet bridging on host allows to place coprocessors on the same subnet as hosts
- Right: Coprocessor Communication Link (CCL) – virtualization of an InfiniBand device on each coprocessor



© 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`



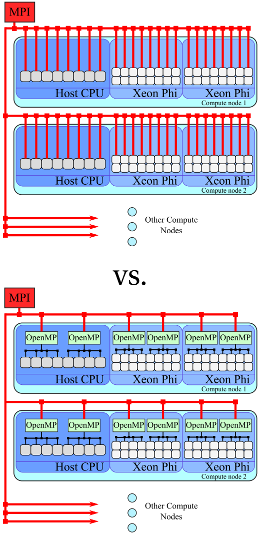

Our publication with details:
http://xeonphi.com/papers/p2p


© Colfax International, 2013–2016

# 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

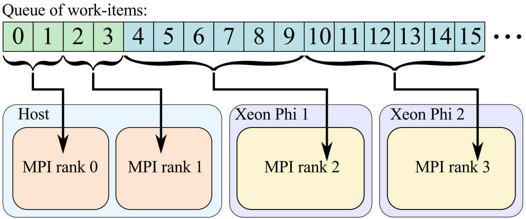




© Colfax International, 2013–2016

# STATIC LOAD BALANCING

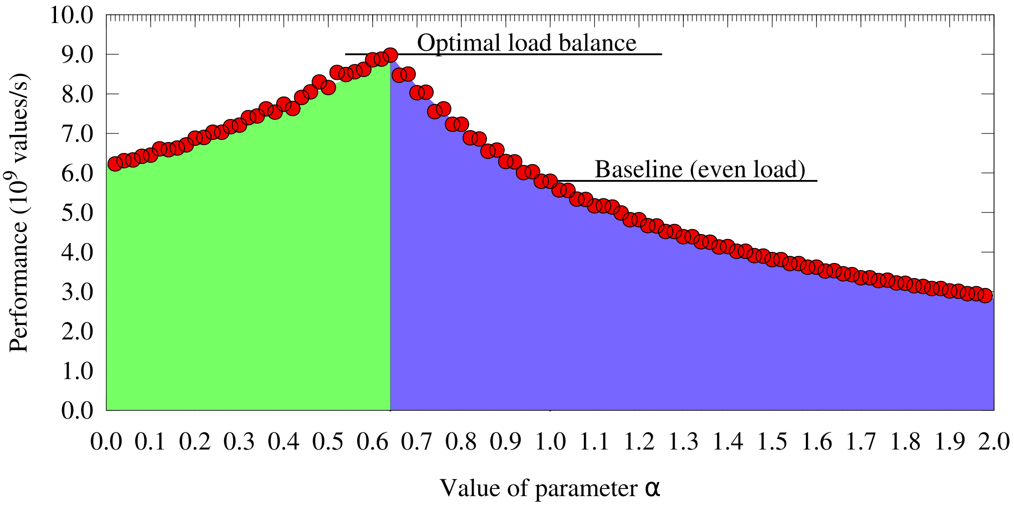
```
if (rankTypes[myRank] == 0) { // I am a MIC-based rank
  double optionsPerProc = double(lastOptForCPUs)/double(cpuRanks.size());
  myFirstOpt = int(optionsPerProc*(myGroupRank));
  myLastOpt  = int(optionsPerProc*(myGroupRank+1.0));
} else { // I am a CPU-based rank
  double optionsPerProc = double(nOpts-lastOptForCPUs)/double(micRanks.size());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





© Colfax International, 2013–2016

# STATIC LOAD BALANCING: PARAMETER TUNING

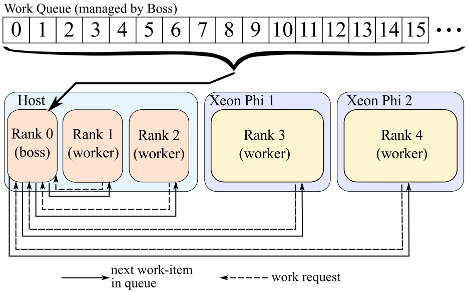




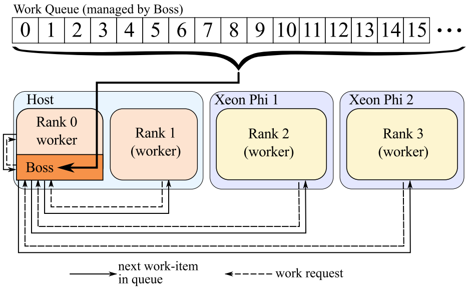
© Colfax International, 2013–2016

# DYNAMIC LOAD BALANCING

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




or





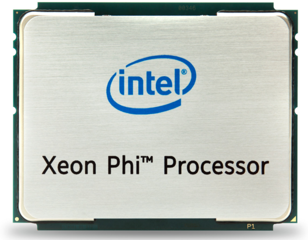
© Colfax International, 2013–2016

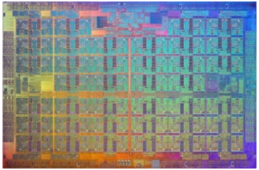
# §4. PREPARING FOR INTEL XEON PHI PROCESSORS

# INTEL XEON PHI PROCESSORS (2ND GEN)

Specialized platform for demanding computing applications.

- Socket version or coprocessor
- 64-72 cores × 4 HT at 1.3-1.5 GHz
- 3+ TFLOP/s in DP (FMA)
- 6+ TFLOP/s in SP (FMA)
- ≤ 384 GiB DDR4 (> 90 GB/s)
- 16 GiB HBM (MCDRAM, > 400 GB/s)
- Binary-compatible with Xeon
- Common OS (RHEL/CentOS/SUSE/Windows)



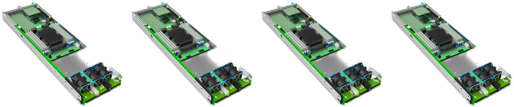

© Colfax International, 2013–2016

# BOOTABLE INTEL XEON PHI PROCESSORS

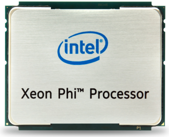
- Bootable Host Processor
- RHEL/CentOS/SUSE/Win
- 64 cores × 4 HT, 1.3 GHz
- ≤ 384 GiB DDR4, > 90 GB/s
- 16 GiB HBM, > 400 GB/s
- PCIe bus for networking

dap.xeonphi.com

Servers:



Workstations:





© Colfax International, 2013–2016

# GET READY FOR INTEL® XEON PHI PROCESSORS (CODENAMED: KNIGHTS LANDING)


colfaxresearch.com/knl-ready


colfaxresearch.com/how-knl


© Colfax International, 2013–2016

# COMPILING WITH AVX-512

# AVX-512 FEATURES

- AVX-512F (Fundamentals)
  - Extension of most AVX2 instructions to 512-bit vector registers.
- AVX-512CD (Conflict Detection)
  - Efficient conflict detection (application: binning).
- AVX-512ER (Exponential and Reciprocal)
  - Transcendental function (`exp`, `rcp` and `rsqrt`) support.
- AVX-512PF (Prefetch)
  - Prefetch for scatter and gather.

Learn more: colfaxresearch.com/avx-512


© Colfax International, 2013–2016

# INTEL COMPILER SUPPORT FOR AVX-512

Intel C, C++ and Fortran compilers ≥ 15.0 support AVX-512

```
user@knl% icc -v
icc version 16.0.1 (gcc version 4.8.5 compatibility)
user@knl% icc -help
// ... truncated output ... //
-x<code>
     ...
     MIC-AVX512
     CORE-AVX512
     COMMON-AVX512
```

- -xMIC-AVX512 : for KNL (supports F, CD, ER, PF)
- -xCORE-AVX512 : for future Xeon (supports F, CD, DQ, BW, VL)
- -xCOMMON-AVX512 : common to KNL and Xeon (supports F, CD)


© Colfax International, 2013–2016

# GCC SUPPORT FOR AVX-512

GCC ≥ 4.9.1 supports AVX-512 instruction set.

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

Basic automatic vectorization support: add -m flags and -O3:

```
user@knl% g++ -v
gcc version 4.9.2 (GCC)
user@knl% g++ foo.cc -mavx512f -mavx512er -mavx512cd -mavx512pf -O3
```

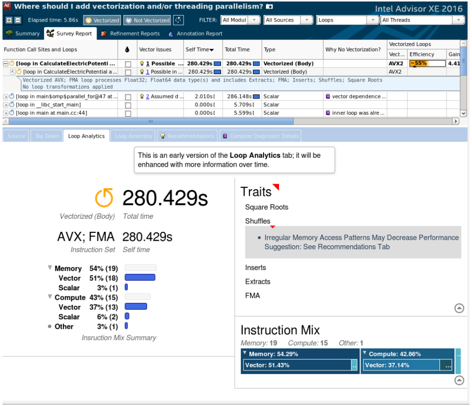
Get assembly:

```
user@knl% g++ -s foo.cc -mavx512f -O3
user@knl% cat foo.s
...
  vmovapd -16432(%rbp,%rax), %zmm0
  vaddpd -8240(%rbp,%rax), %zmm0, %zmm0
  vmovapd %zmm0, -8240(%rbp,%rax)
```

 © Colfax International, 2013–2016

# PERFORMANCE CONSIDERATIONS

## Even if your code is vectorized, tuning may unlock more performance.




download paper to learn more

- ▷ Providing enough parallelism.
  - More consecutive vector operations required to overcome vectorization latency.
- ▷ Loop pipelining and unrolling.
  - Double the pipeline stages to populate.
- ▷ Better vectorization patterns.
  - Avoid long latency operations with unit-stride and unmasked operations.



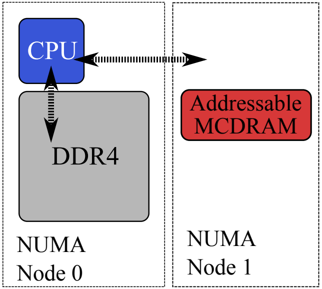
© Colfax International, 2013–2016

# USING HIGH-BANDWIDTH MEMORY

# MODES OF HBM OPERATION

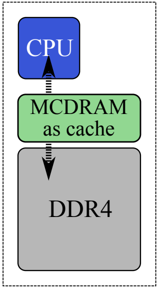
## Flat Mode

- MCDRAM treated as a NUMA node
- Users control what goes to MCDRAM



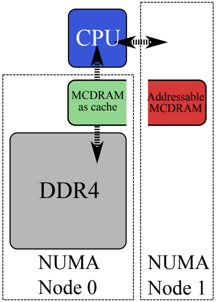

## Cache Mode

- MCDRAM treated as a Last Level Cache (LLC)
- MCDRAM is used automatically




## Hybrid Mode

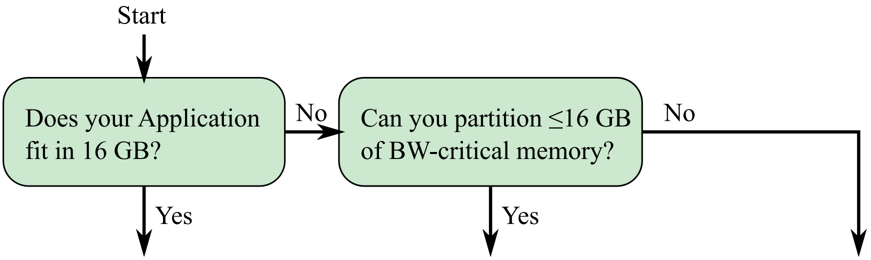
- Combination of Flat and Cache
- Ratio can be chosen in the BIOS





© Colfax International, 2013–2016

# FLOW CHART FOR BANDWIDTH-BOUND APPLICATIONS




| numactl | Memkind | Cache mode |
|---|---|---|
| ▷ Run the whole program in HBM<br>▷ No code modification | ▷ Selectively allocate data to HBM<br>▷ Add memkind calls | ▷ Allow the chip to figure out how to use HBM<br>▷ No code modification |


© Colfax International, 2013–2016

# RUNNING APPLICATIONS IN HBM WITH NUMACTL

- Finding information about the NUMA nodes in the system.

```
user@knl% # In Flat mode of MCDRAM
user@knl% numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ... 254 255
node 0 size: 98207 MB
node 0 free: 94798 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15991 MB
```

- Binding the application to HBM (Flat/Hybrid)

```
user@knl% gcc myapp.c -o runme -mavx512f -O2
user@knl% numactl --membind 1 ./runme
// ... Application running in HBM ... //
```


© Colfax International, 2013–2016

# ALLOCATION IN HBM WITH MEMKIND LIBRARY

Manual allocation to HBM possible with hbwmalloc and Memkind Library.

```
#include <hbwmalloc.h>

// Basic allocation in HBM
double* A = (double*) hbw_malloc(sizeof(double)*n);

// Allocation with alignment
double* B;
int ret = hbw_posix_memalign((void**) &B, 64, sizeof(double)*n);

hbw_free(A); hbw_free(b); // Special deallocator
```

In Fortran:

```
REAL, ALLOCATABLE :: A(:)
!DEC$ ATTRIBUTES FASTMEM :: A
ALLOCATE (A(1:N))
```


© Colfax International, 2013–2016

# COMPILATION WITH MEMKIND LIBRARY AND HBWMALLOC

To compile C/C++ applications:

```
user@knl% icpc -lmemkind foo.cc -o runme
user@knl% g++ -lmemkind foo.cc -o runme
```

To compile Fortran applications:

```
user@knl% ifort -lmemkind foo.f90 -o runme
user@knl% gfortran -lmemkind foo.f90 -o runme
```

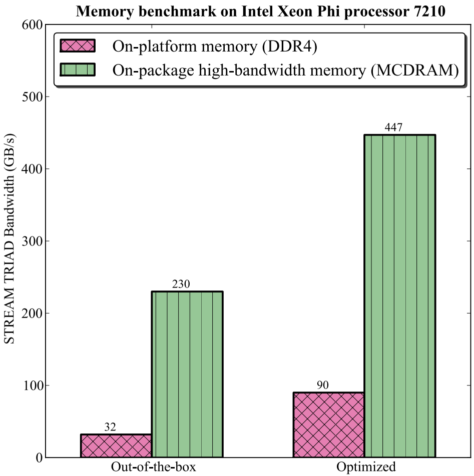
Open source distribution of Memkind library can be found at:
memkind.github.io/memkind

Learn more:
colfaxresearch.com/knl-mcdram


© Colfax International, 2013–2016

# STREAM BENCHMARK

- Industry-standard tool for memory bandwidth measurement
- 4 tests: COPY, ADD, SCALE and TRIAD
- Download from Dr. John McCalpin's site: www.cs.virginia.edu/stream/



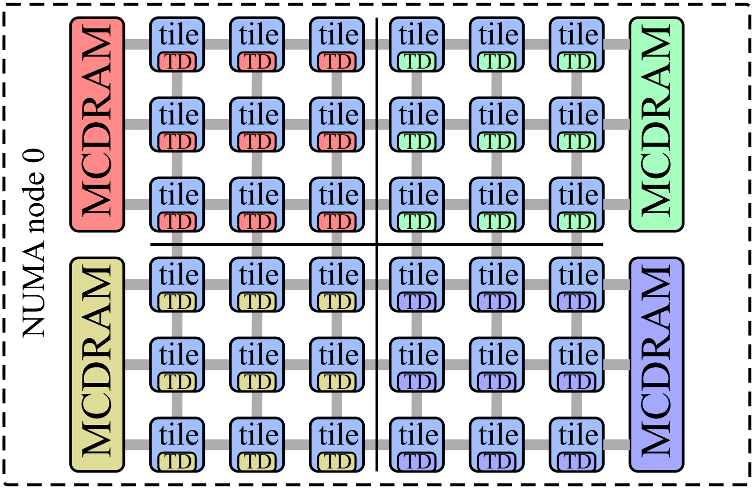


© Colfax International, 2013–2016

# LEVERAGING CLUSTERING MODES

# CLUSTERING MODES: QUADRANT/HEMISPHERE

Tag Directory (TD) and memory reside in the same quadrant.

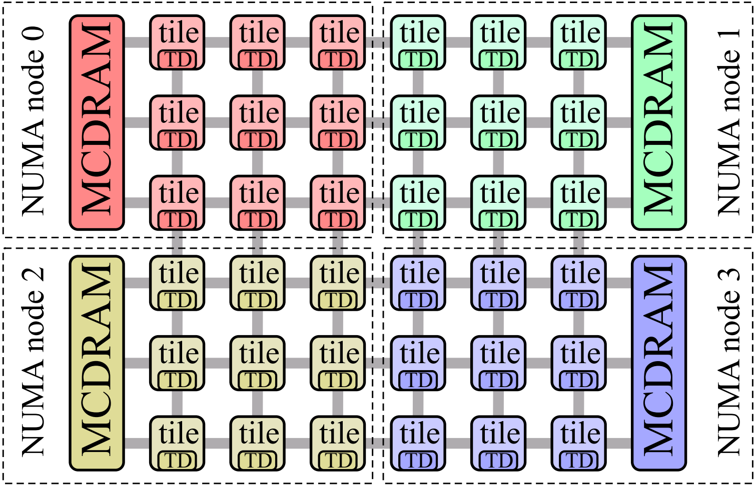




© Colfax International, 2013–2016

# CLUSTERING MODES: SNC-4/SNC-2

Cores appear as 4 (or 2) NUMA nodes.



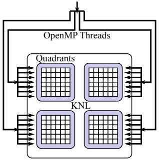


© Colfax International, 2013–2016

# HOW TO USE CLUSTERING MODES

## Nested OpenMP

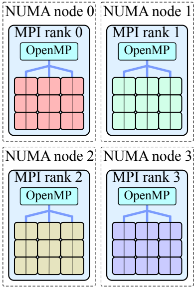
```
#pragma omp parallel
{
  // ...
#pragma omp parallel
  {
    // ...
  }
}
```




```
user@knl% OMP_NUM_THREADS=4,72
user@knl% OMP_NESTED=1
```

## MPI+OpenMP

```
stat = MPI_Init();
// ...
#pragma omp parallel
{
    // ...
}
// ...
MPI_Finalize();
```




```
user@knl% mpirun -host knl \
>   -np 4 ./myparallel_app
```

Learn more: colfaxresearch.com/knl-numa

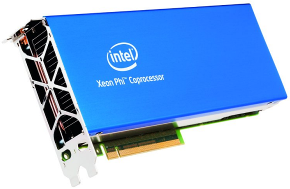

© Colfax International, 2013–2016

# COPROCESSOR AND KNL-F

# Future Form-Factors

## KNLF: KNL with Fabric

- Fabric integrated on CPU
  - Intel® Omni-Path Architecture
- Socket mount processor




*KNC image

## KNL Coprocessor

- PCIe add-in card
  - Requires host
- Multiple KNLs in a system


© Colfax International, 2013–2016