PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

Hands-On Workshop (HOW) Series "Deep Dive"

Session 6

Colfax International — *colfaxresearch.com*

DISCLAIMER

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

COURSE ROADMAP

HOW SERIES ONLINE

Course page:
colfaxresearch.com/how-series

- ▷ Slides
- ▷ Code
- ▷ Video
- ▷ Chat

More workshops:
colfaxresearch.com/training

GET YOUR QUESTIONS ANSWERED: CHAT

colfaxresearch.com/how-series

GET YOUR QUESTIONS ANSWERED: FORUMS

READ | WATCH | LEARN | FORUMS | CONNECT | JOIN

Forum

Colfax Cluster

Discussion of Colfax Cluster usage policies, troubleshooting.

Developer Training, HOW Series

Questions about any of the Colfax trainings? Usage of training servers, experience with specific exercises, inquiries on what's inside, suggestions for future trainings - post them here.

Performance Optimization and Parallelism

Discuss with Colfax Research and colleagues any topics related to computational science, parallel programming, performance optimization and code modernization.

colfaxresearch.com/forum

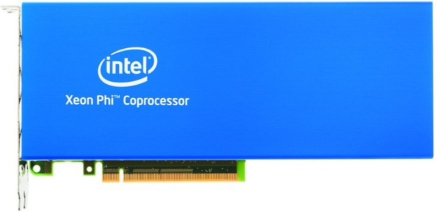
HANDS-ON EXERCISES AND REMOTE ACCESS

- All registrants receive an invitation from cluster@colfaxresearch.com
- Queue-based access to Intel Xeon E5, Intel Xeon Phi (KNC and KNL)
- Can access the cluster the entire 2 weeks of the workshop

§2. PERFORMANCE OPTIMIZATION

COMPUTING PLATFORMS

Intel Xeon Processor	Intel Xeon Phi Coprocessor, 1st generation	Intel Xeon Phi Processor, 2nd generation*



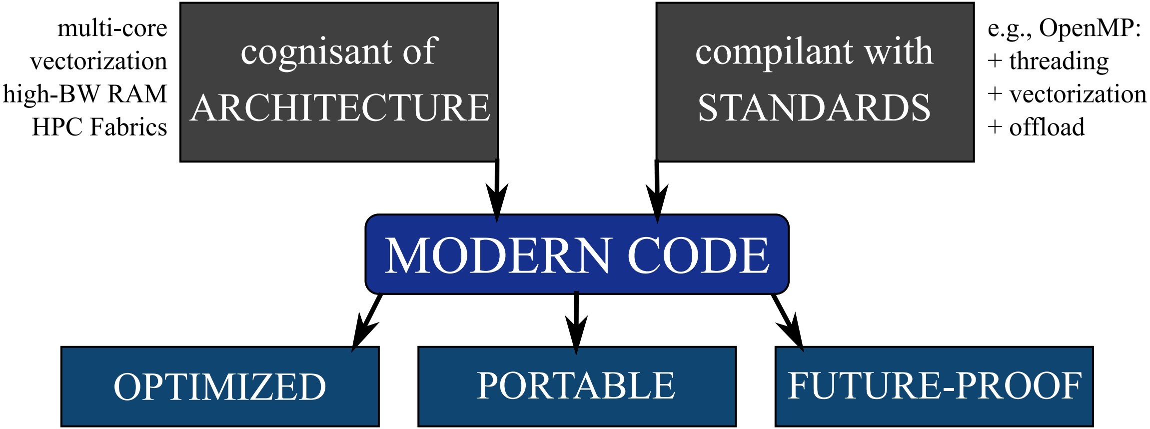
* socket and coprocessor versions

Current: Broadwell Upcoming: Skylake	Knights Corner (KNC)	Knights Landing (KNL)

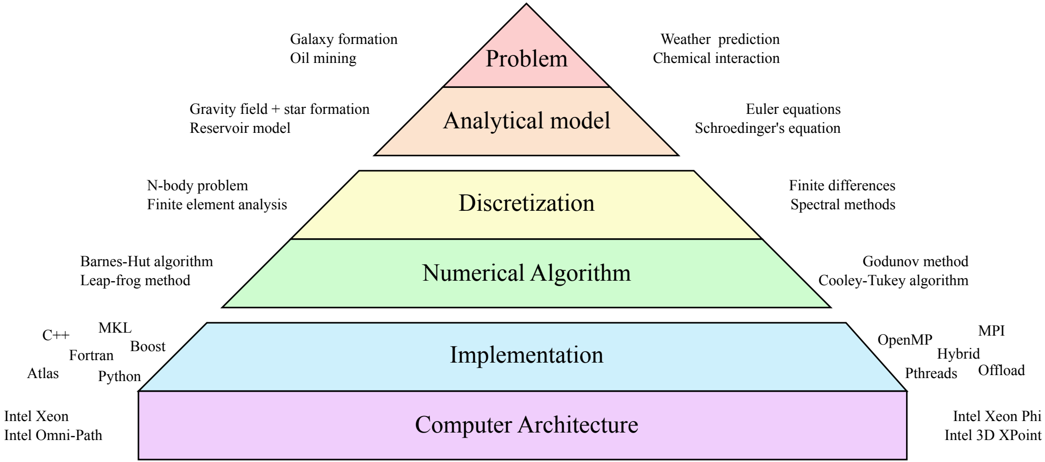
Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

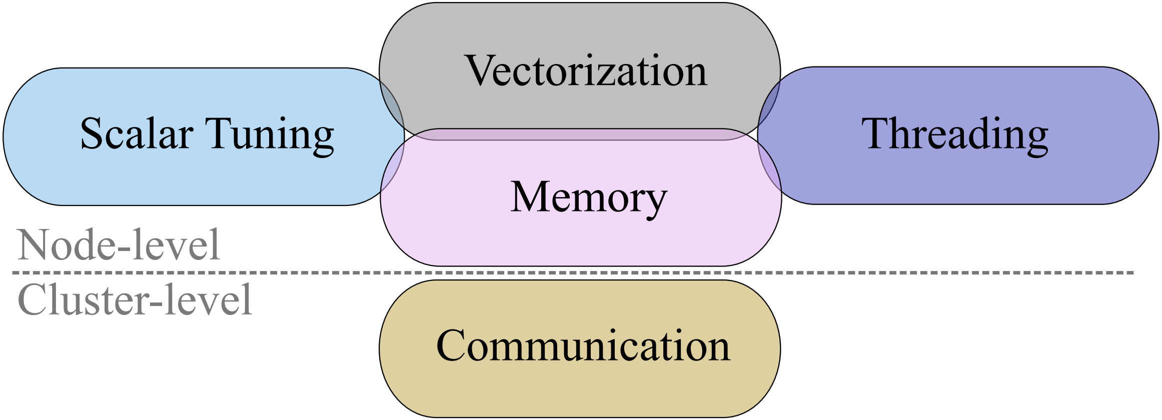
ONE CODE FOR ALL PLATFORMS



COMPUTING IN SCIENCE AND ENGINEERING

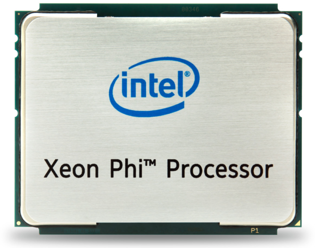


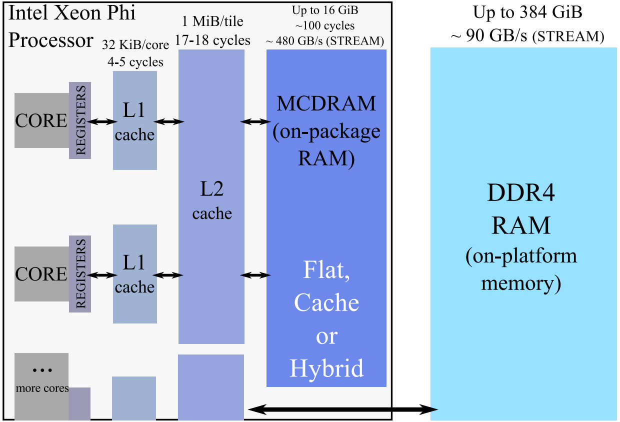
OPTIMIZATION AREAS



KNL MEMORY ORGANIZATION (BOOTABLE)

- On-package high-bandwidth memory (HBM) – MCDRAM
- Optimized for arithmetic performance and bandwidth (not latency)





§3. N-BODY SIMULATION

PHYSICS

N-BODY SIMULATION ON CPU AND COPROCESSOR

COLFAX Customized Solutions

N-body simulation on...

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

Paper: https://colfaxresearch.com/nbody-basic
Demo: click here

Gravitational N-body dynamics:

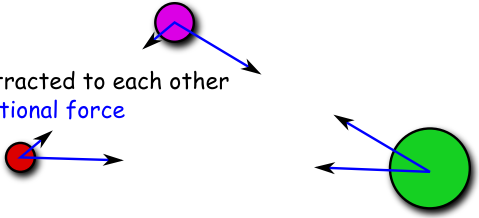
Newton's law of universal gravitation:

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

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)

ALL-TO-ALL APPROACH ($O\left(n^2\right)$ 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
```

OPTIMIZATION

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;
  }
```

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
      ...
```

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

VECTORIZING WITH UNIT-STRIDE MEMORY ACCESS

Before:

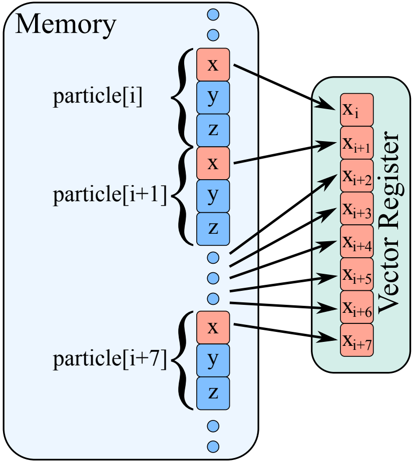
```
struct ParticleType {
  float x, y, z, vx, vy, vz;
};  // ...
      const float dx = particle[j].x - particle[i].x;
      const float dy = particle[j].y - particle[i].y;
      const float dz = particle[j].z - particle[i].z;
```

After:

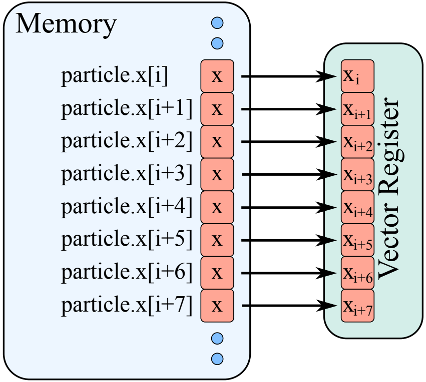
```
struct ParticleSet {
  float *x, *y, *z, *vx, *vy, *vz;
}; // ...
      const float dx = particle.x[j] - particle.x[i];
      const float dy = particle.y[j] - particle.y[i];
      const float dz = particle.z[j] - particle.z[i];
```

WHY AOS TO SOA CONVERSION HELPS: UNIT STRIDE

Array of Structures (sub-optimal)



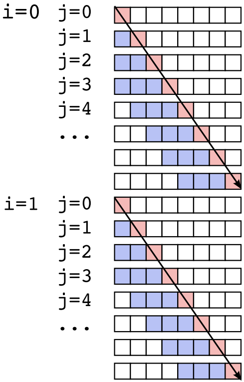
Structure of Arrays (optimal)



LOOP TILING: REGISTER BLOCKING

Original:

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



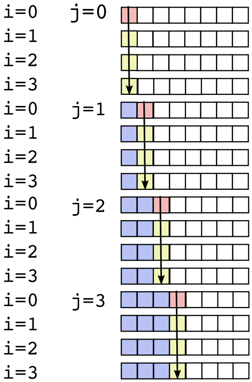
- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

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



IMPROVING CACHE TRAFFIC

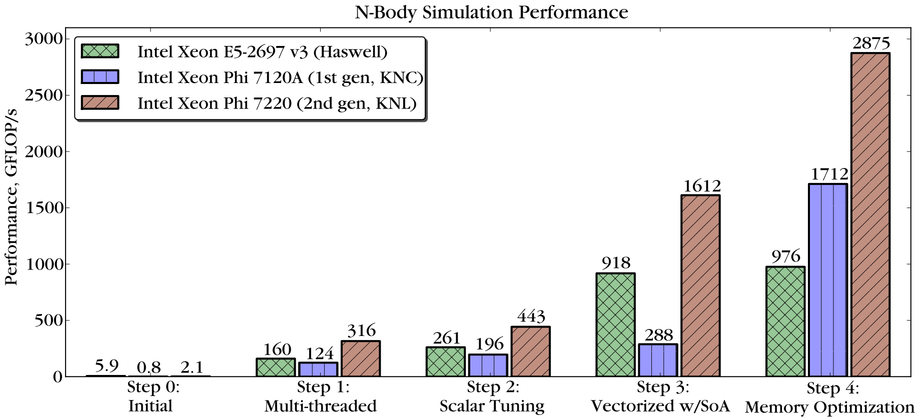
Before:

```
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
      // ...
^^IFx[i-ii] += dx*drPowerN32; Fy[i-ii] += dy*drPowerN32; Fz[i-ii] += dz*drPowerN32;
```

IMPACT OF CODE OPTIMIZATION

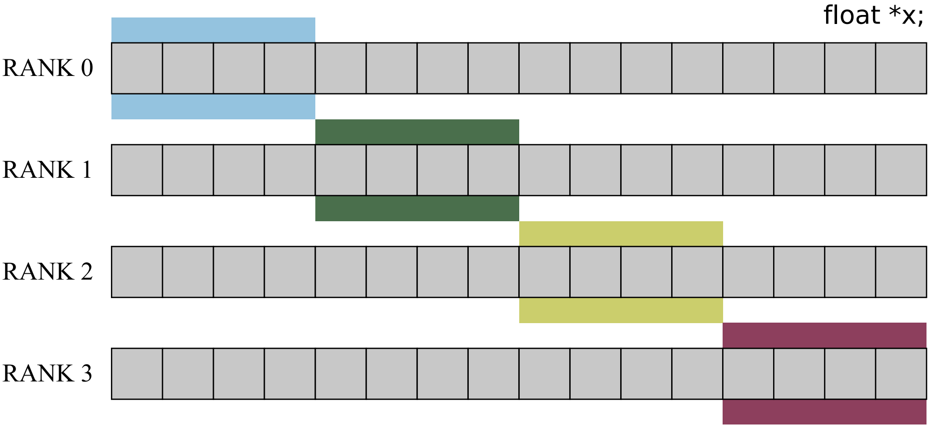


Contributed as Chapter 23 in "Intel Xeon Phi Processor High Performance Programming, Knights Landing Edition" (2016)

MPI IMPLEMENTATION

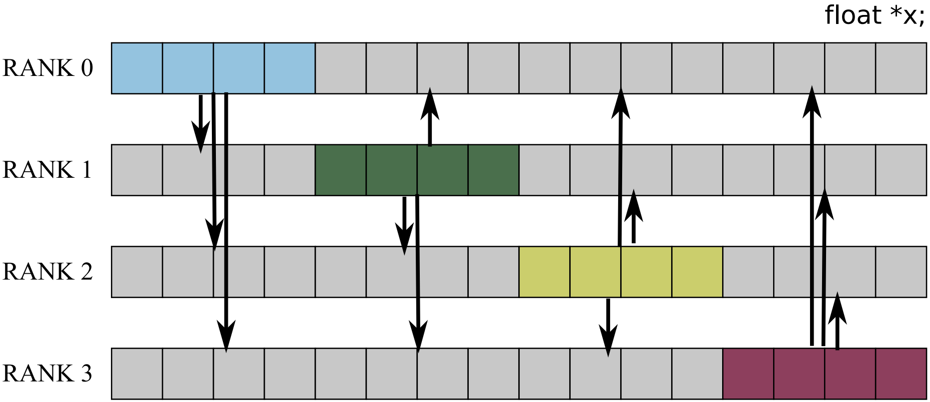
WORK DISTRIBUTION

- All processes start with same complete set of particle coordinates.
- Each processor moves only its share of particles



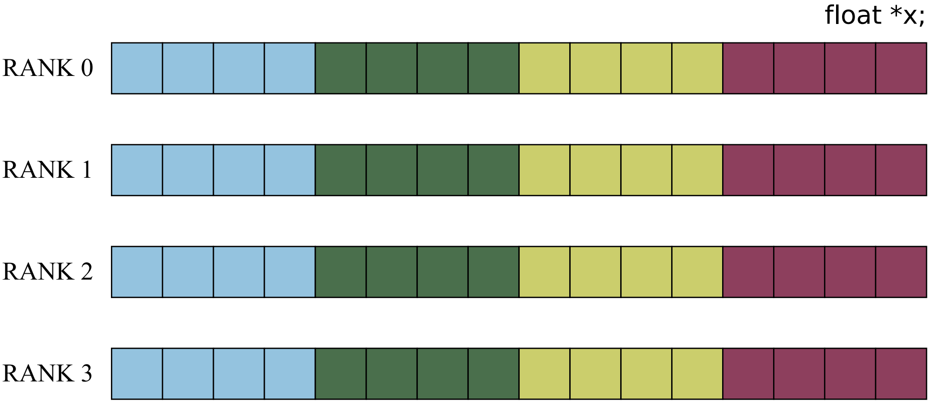
COMMUNICATION

- After time step, need to propagate modified particles to all peers
- Use the Allgather operation from MPI



COMMUNICATION

- All processors end up with the same data set again
- Only x, y, z need to be propagated. vx, vy, vz stay local



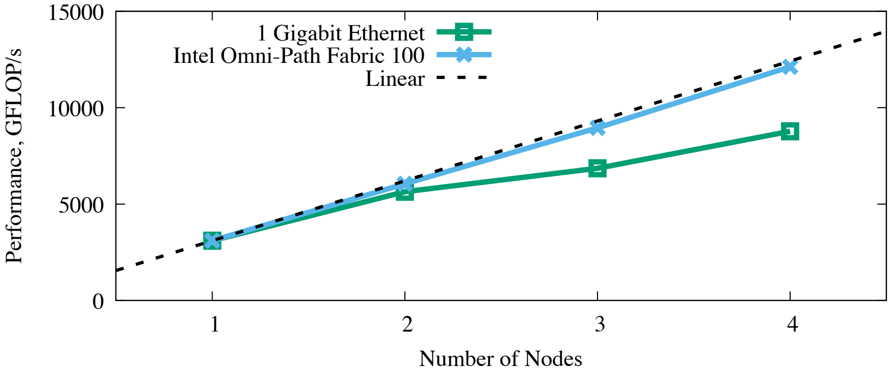
CORE OF MPI-ONLY IMPLEMENTATION

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);
  // ...
```

PERFORMANCE WITH MPI

Intel Xeon Phi 7210 processors, $N = 2^{18}$ particles



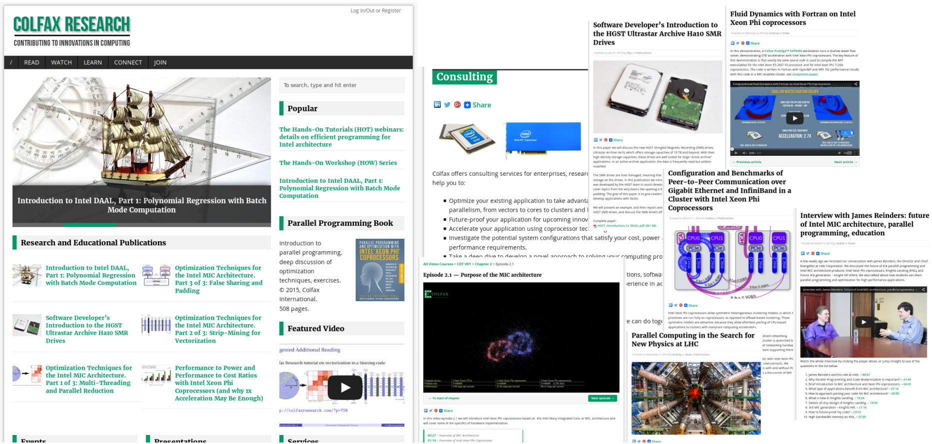
Impact of communication decreases with increasing N.

SUMMARY

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

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

Next session: scalar tuning, optimization of vectorization.



https://colfaxresearch.com/