Programming and Optimization for Intel® Architecture

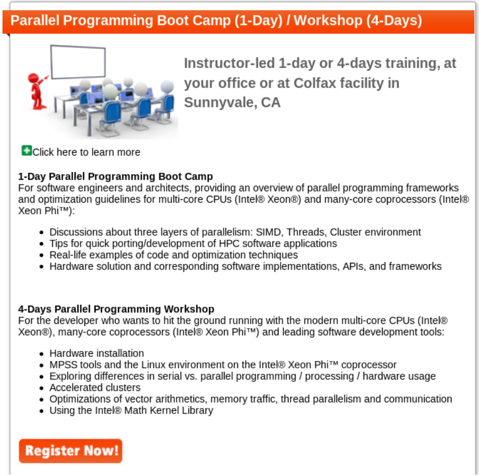
The Hands-On Workshop (HOW) Series

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

March 2016 , Rev. 02a

About This Document

This document represents the materials of a Web-based training "Programming and Optimization with Intel Architecture" developed and run by Colfax International.



colfaxresearch.com/how-series

Disclaimer

Course Roadmap

1. Why Intel Parallel Architectures?
   - Parallelism and specialization – March 7
   - Programming model continuity – March 7
2. Programming models for Xeon Phi coprocessors
   - Native programming – March 7
   - Offload programming – March 8
3. Expressing Parallelism
   - Introduction to vectorization – March 9
   - Crash-course on OpenMP – March 10
4. Optimization – intro on March 11
   - Vectorization tuning – March 14
   - Multi-threading – March 15, 16
   - Memory traffic – March 17
5. Tools: MKL and VTune – March 18

March 2016

S	M	T	W	H	F	S
		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		

— Lecture+remote access

— Self-study/remote access

HOW Online

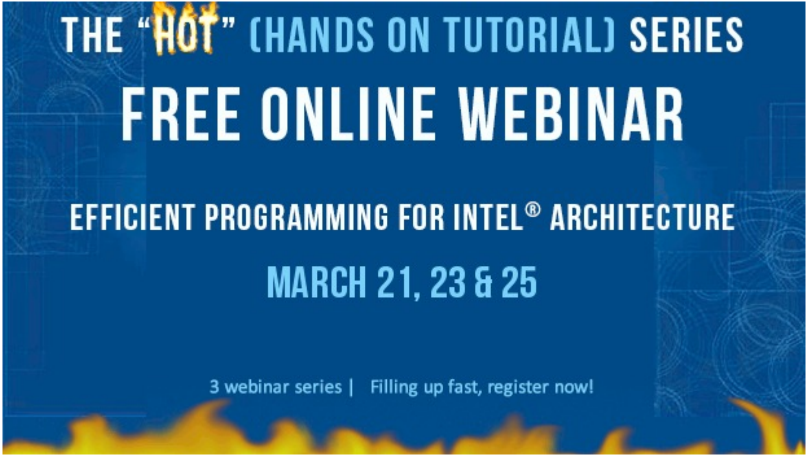
Course page: colfaxresearch.com/how-16-03

- Slides (including this one), code downloads
- Video of recorded sessions
- Chat (during webinars or offline)

Additional resources:

- More workshops like this one: colfaxresearch.com/how-series
- Video courses: colfaxresearch.com/video-courses
- Intel Many Integrated Core Architecture Forum

HOT Series



colfaxresearch.com/hot-16-03/

HOW Series, April 2016

colfaxresearch.com/how-16-04/

§2. Expressing Data Parallelism

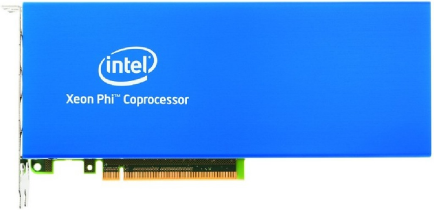
Handling Multiple Cores

Computing Platforms

Intel Xeon
Processor

Current: Haswell
Upcoming: Broadwell

Intel Xeon Phi
Corocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi
Processor, 2nd generation*

* socket and coprocessor versions

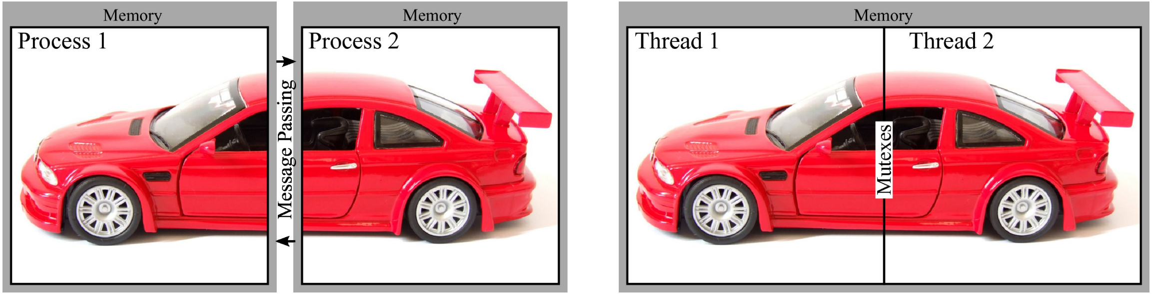
Upcoming: Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

Threads versus Processes

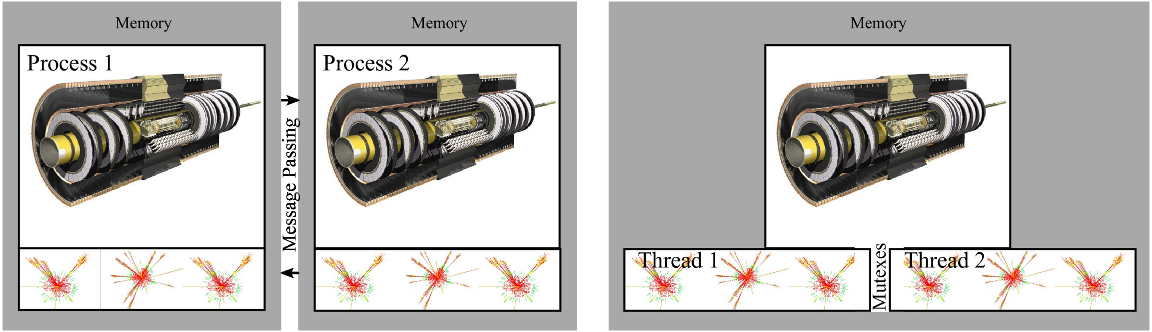
Option 1: Partitioning data set between threads/processes



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

Threads versus Processes

Option 2: Sharing data set between threads/processes



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

Scalability Expectations (CPU)

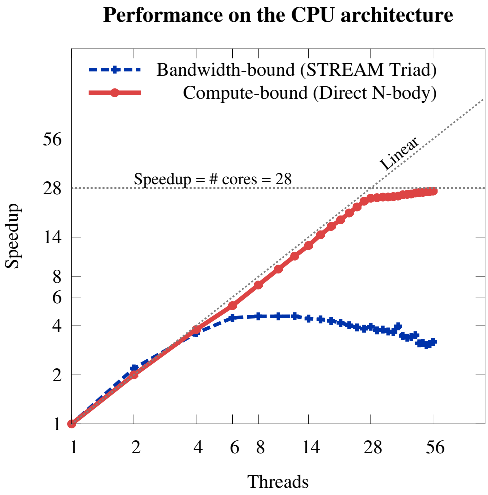
T = number of threads

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

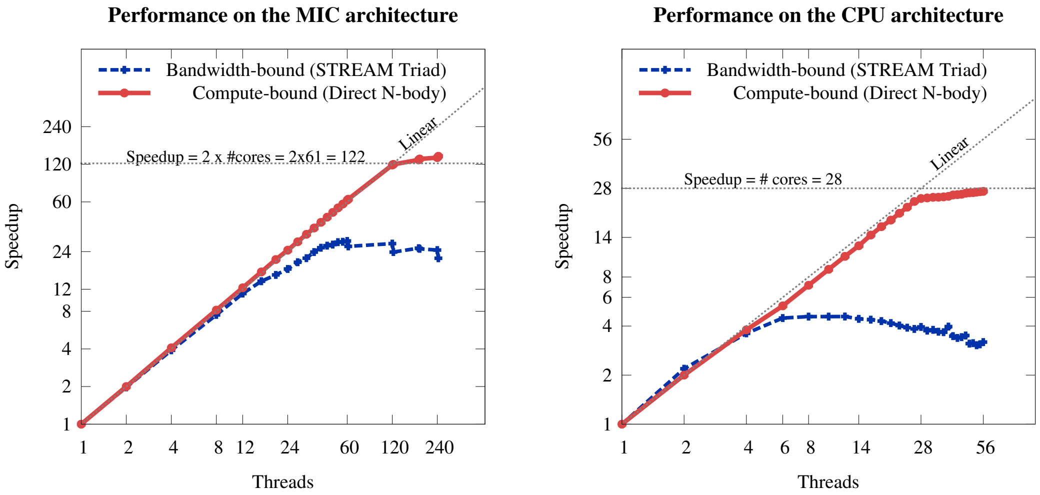
$$\text{Efficiency}(T) = \frac{\text{Speedup}(T)}{T}$$

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

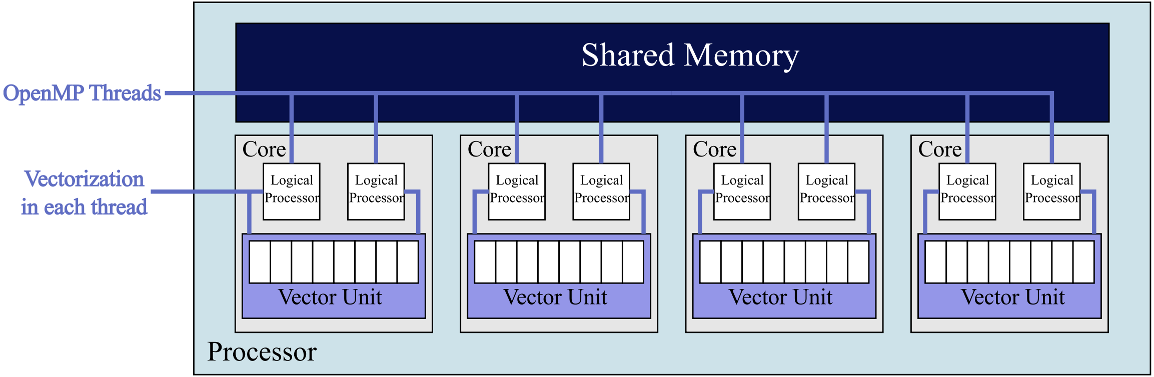
$$\text{Speedup}(T) = T$$



Scalability Expectations: MIC versus CPU



Co-Existence with Vectors



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]);
```

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

OpenMP Basics

“Hello World” OpenMP Programs

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

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

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

“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):

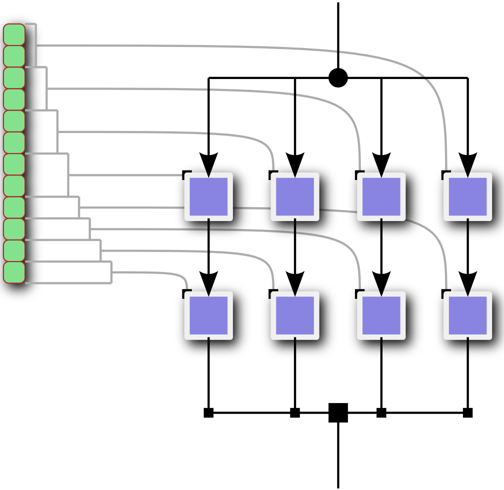
```
int A, B; // Variables declared at the beginning of a function
#pragma omp parallel private(A) shared(B)
  {
    // Each thread has its own copy of A, but B is shared
  }
```

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

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

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

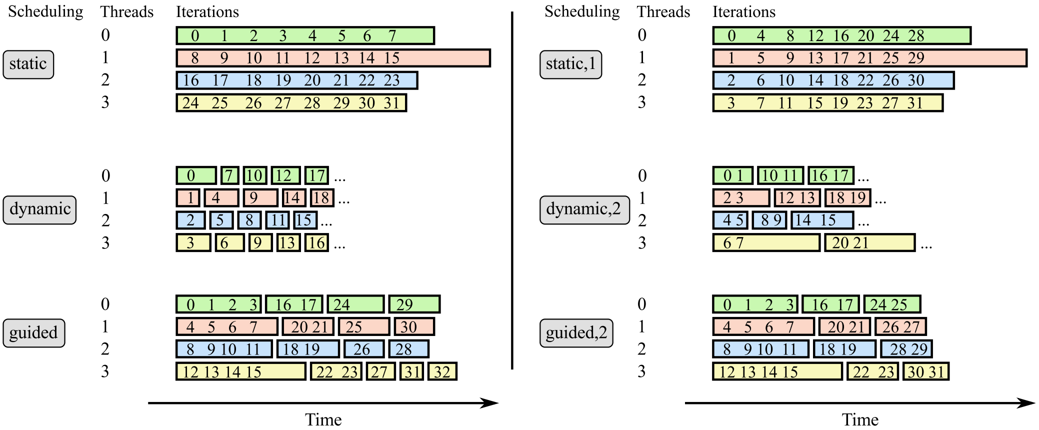
The OpenMP library will distribute the iterations of the loop following the `#pragma omp parallel for` across threads.

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


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
}
```

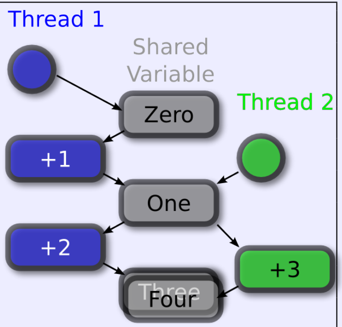
Loop Scheduling Modes in OpenMP



Thread Synchronization

Race Conditions and Unpredictable Program Behavior

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

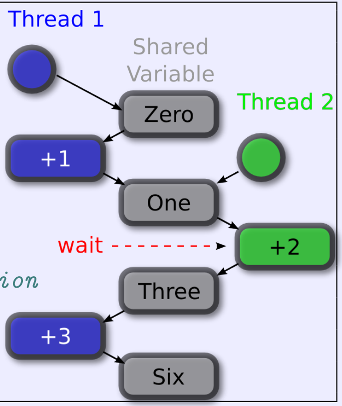


Race Condition!

```
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

```
#include <omp.h>
#include <stdio.h>
int main() {
  const int n = 1000;
  int total = 0;
#pragma omp parallel for
  for (int i = 0; i < n; i++) {
#pragma omp critical
    { // Only one thread at a time can execute this section
      total = total + i;
    }
} }
```




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

Avoiding Races with Atomic Operations

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

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

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.

Parallel Reduction

Reduction Clause in Parallel Region

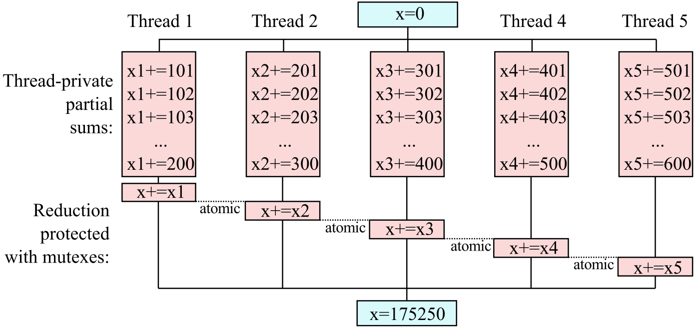
```
#include <omp.h>
#include <stdio.h>

int main() {
  const int n = 1000;
  int total = 0;
#pragma omp parallel for reduction(+: total)
  for (int i = 0; i < n; i++) {
    total = total + i;
  }
  printf("total=%d (must be %d)\n", total, ((n-1)*n)/2);
}
```

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

Avoiding Races with Thread-Private Storage

Correct and efficient code:

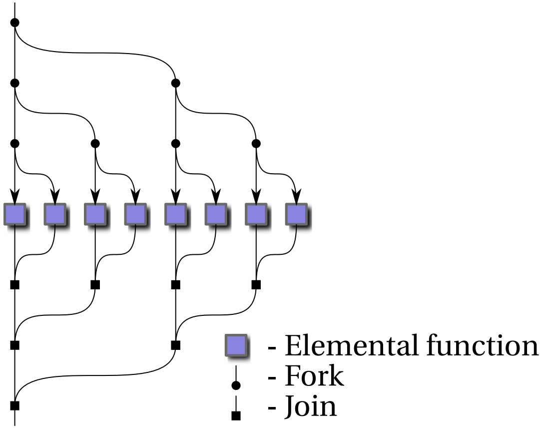
```
int total = 0;
#pragma omp parallel
{
  int total_thr = 0;
#pragma omp for
  for (int i=0; i<n; i++)
    total_thr += i;

#pragma omp atomic
  total += total_thr;

}
```



Tasks in OpenMP

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();
  }
}
```

Recipes for Success

 |

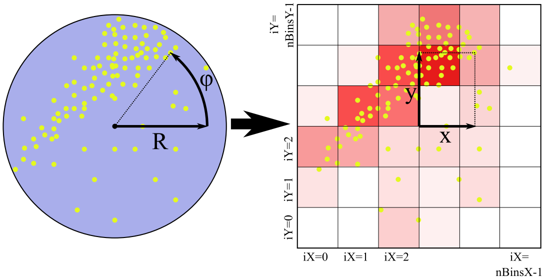
Recipe for Success: “Take a Deep Breath”

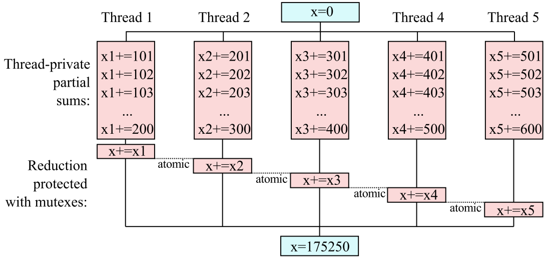
Where is parallelism in your application?

See also full interview with James Reinders at Colfax Research

Suggested Additional Reading

Colfax Research tutorial on multi-threading in a binning code

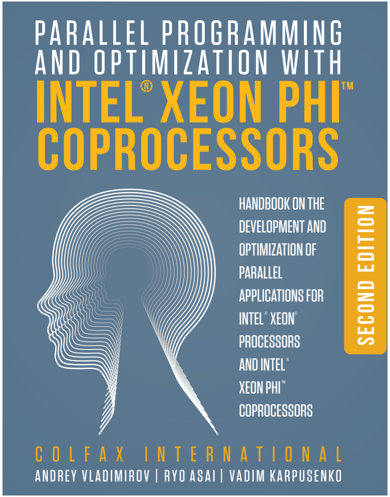




http://colfaxresearch.com/?p=6

Additional Materials on OpenMP

1. OpenMP Specifications
2. Intel's OpenMP Video Course
3. LLNL tuotorial: OpenMP
4. Book: "Parallel Programming and Optimization with Intel Xeon Phi Coprocessors" by Colfax.



Summary

Discussed today:

- Cores can run independent programs
- Application must use threads to scale across cores
- Race conditions lead to incorrect, unpredictable results
- Synchronization controls race conditions at cost of performance
- Must have vectorization in each thread
- OpenMP – well-established parallel framework for HPC

Next session: introduction into optimization for Xeon and Xeon Phi.

