



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.

© Colfax International, 2013-2015

colfaxresearch.com/how-series

Parallel Programming Boot Camp (1-Day) / Workshop (4-Days)



Instructor-led 1-day or 4-days training, at your office or at Colfax facility in Sunnyvale, CA

[Click here to learn more](#)

1-Day Parallel Programming Boot Camp

For software engineers and architects, providing an overview of parallel programming frameworks and optimization guidelines for multi-core CPUs (Intel® Xeon®) and many-core coprocessors (Intel® Xeon Phi™):

- Discussions about three layers of parallelism: SIMD, Threads, Cluster environment
- Tips for quick porting/development of HPC software applications
- Real-life examples of code and optimization techniques
- Hardware solution and corresponding software implementations, APIs, and frameworks

4-Days Parallel Programming Workshop

For the developer who wants to hit the ground running with the modern multi-core CPUs (Intel® Xeon®), many-core coprocessors (Intel® Xeon Phi™) and leading software development tools:

- Hardware installation
- MPSS tools and the Linux environment on the Intel® Xeon Phi™ coprocessor
- Exploring differences in serial vs. parallel programming / processing / hardware usage
- Accelerated clusters
- Optimizations of vector arithmetics, memory traffic, thread parallelism and communication
- Using the Intel® Math Kernel Library

Register Now!

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

- 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



THE “HOT” (HANDS ON TUTORIAL) SERIES

FREE ONLINE WEBINAR

EFFICIENT PROGRAMMING FOR INTEL® ARCHITECTURE

MARCH 21, 23 & 25

3 webinar series | Filling up fast, register now!

colfaxresearch.com/hot-16-03/

HOW Series, April 2016

A blue rectangular graphic with white and light blue text. The background features faint, light blue geometric patterns including circles and squares. The text is centered and reads: 'THE "HOW" (HANDS ON WORKSHOP) SERIES' in light blue, 'FREE ONLINE TRAINING' in large white letters, 'PARALLEL PROGRAMMING AND OPTIMIZATION' in white, 'FOR INTEL® ARCHITECTURE' in white, and 'STARTS APR 18' in light blue. At the bottom, a line of small white text provides details: '*10 2-hour sessions | 24-hour 3-week access to a system | Filling up fast, register now!'.

THE "HOW" (HANDS ON WORKSHOP) SERIES

FREE ONLINE TRAINING

PARALLEL PROGRAMMING AND OPTIMIZATION

FOR INTEL® ARCHITECTURE

STARTS APR 18

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

colfaxresearch.com/how-16-04/

§2. Expressing Data Parallelism

Handling Multiple Cores

Computing Platforms

Intel Xeon Processor



Current: Haswell
Upcoming: Broadwell

Multi-Core Architecture

Intel Xeon Phi Coprocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*



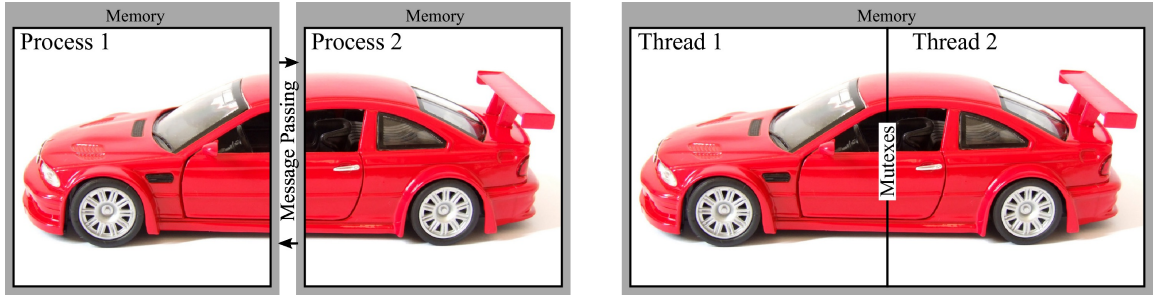
* socket and coprocessor versions

Upcoming: Knights Landing (KNL)

Intel Many Integrated Core (MIC) Architecture

Threads versus Processes

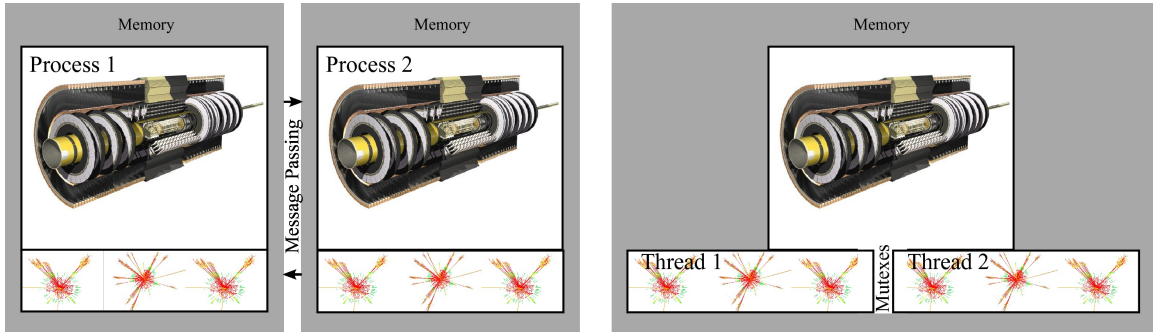
Option 1: Partitioning data set between threads/processes



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

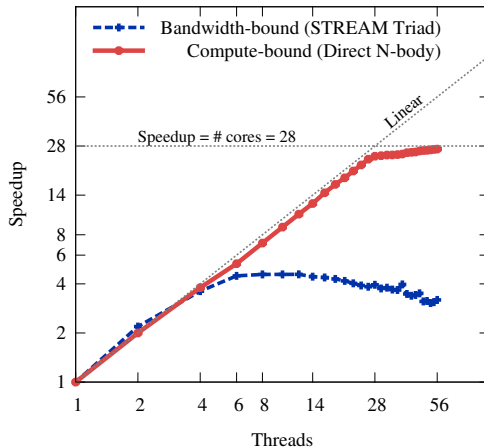
$$\text{Speedup}(T) = \frac{\text{Performance}(T)}{\text{Performance}(1)}$$

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

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

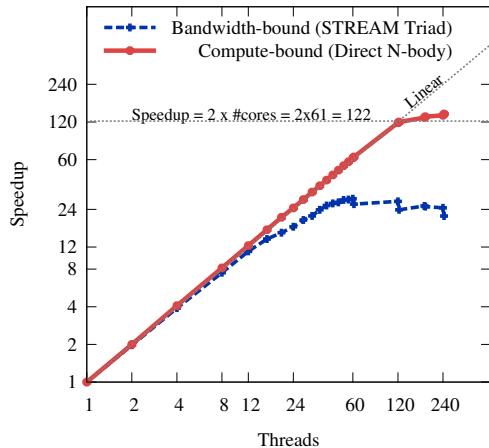
$$\text{Speedup}(T) = T$$

Performance on the CPU architecture

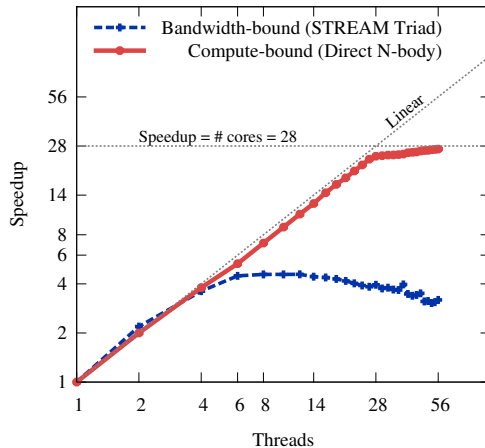


Scalability Expectations: MIC versus CPU

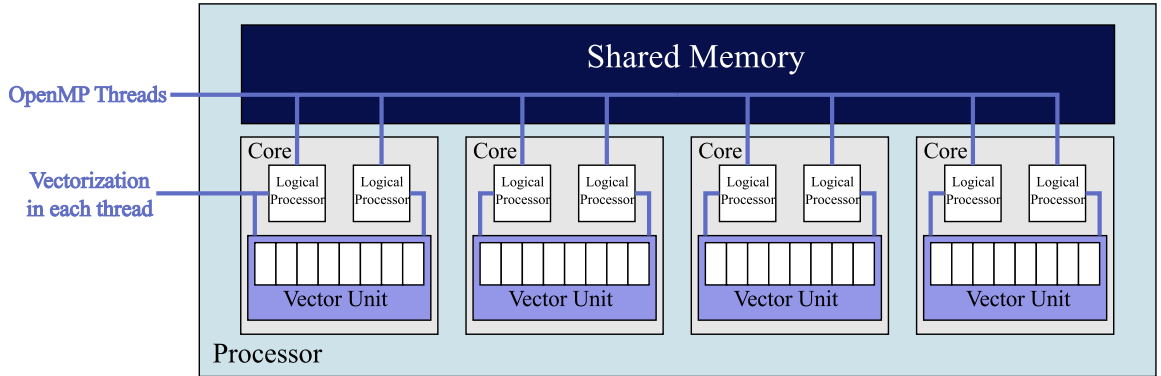
Performance on the MIC architecture



Performance on the CPU architecture



Co-Existence with Vectors



Simultaneous Threading and Vectorization

This approach often works:

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

That works as well:

```
1 #pragma omp parallel for simd
2 for (int i = 0; i < n; i++) // If the problem is all data-parallel
3     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, au- tomatic scheduler
OpenMP	Various	Simple to Complex	HPC-specific functional- ity, automatic and manual control possible

OpenMP Basics

“Hello World” OpenMP Programs

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

“Hello World” OpenMP Programs

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

OMP_NUM_THREADS controls number of OpenMP threads (default: logical CPU count)

Control of Variable Sharing

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

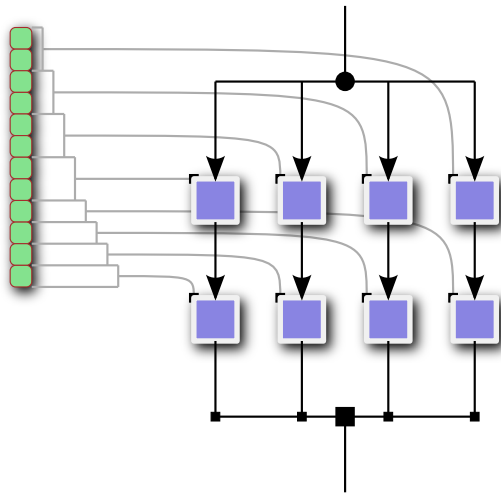
```
1 int A, B; // Variables declared at the beginning of a function
2 #pragma omp parallel private(A) shared(B)
3 {
4     // Each thread has its own copy of A, but B is shared
5 }
```

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

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

Loop-Centric Parallelism: For-Loops in OpenMP

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



Parallelizing a for-loop.

Loop-Centric Parallelism: For-Loops in OpenMP

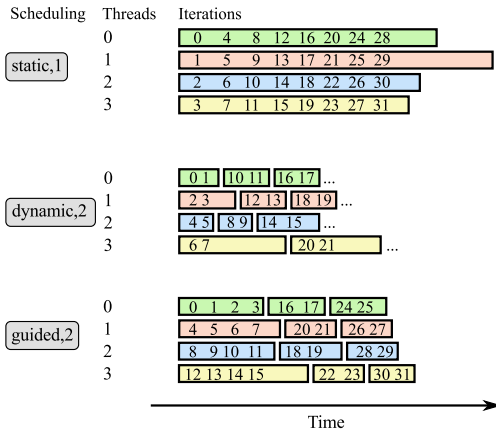
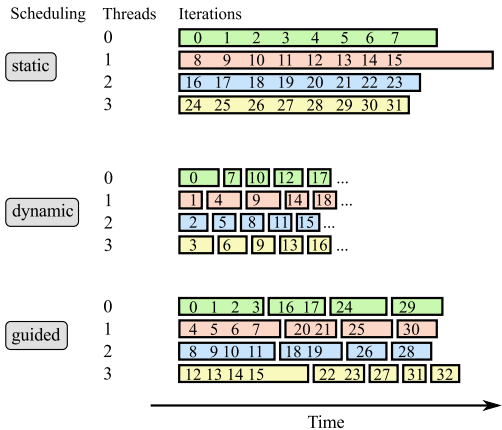
The OpenMP library will distribute the iterations of the loop following the `#pragma omp parallel for` across threads.

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

Loop-Centric Parallelism: For-Loops in OpenMP

```
1  #pragma omp parallel
2  {
3      // Code placed here will be executed by all threads.
4
5      // Alternative way to specify private variables:
6      // declare them in the scope of pragma omp parallel
7      int private_number=0;
8
9  #pragma omp for
10     for (int i = 0; i < n; i++) {
11         // ... iterations will be distributed across available threads...
12     }
13     // ... code placed here will be executed by all threads
14 }
```

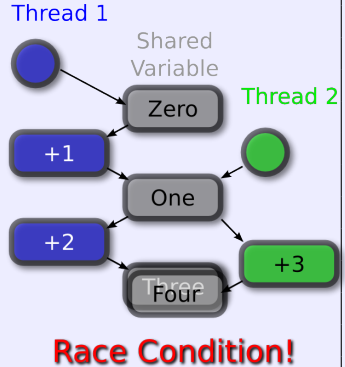
Loop Scheduling Modes in OpenMP



Thread Synchronization

Race Conditions and Unpredictable Program Behavior

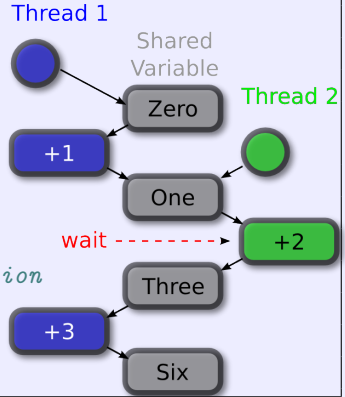
```
1 #include <omp.h>
2 #include <stdio.h>
3 int main() {
4     const int n = 1000;
5     int total = 0;
6     #pragma omp parallel for
7     for (int i = 0; i < n; i++) {
8         // Race condition
9         total = total + i;
10    }
11    printf("total=%d (must be %d)\n", total, ((n-1)*n)/2);
12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



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

Protecting Race Conditions with a Critical Section

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



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

Avoiding Races with Atomic Operations

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

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

Limitations of Atomic Operations

Read : operations in the form $v = x$

Write : operations in the form $x = v$

Update : operations in the form $x++$, $x--$, $--x$, $++x$, $x \text{ binop} = \text{expr}$
and $x = x \text{ binop expr}$

Capture : operations in the form $v = x++$, $v = x--$, $v = --x$, $v = ++x$,
 $v = x \text{ binop expr}$

- Here x and v are scalar variables
- binop is one of $+$, $*$, $-$, $- /$, $\&$, $\wedge$, $|$, $\ll$, $\gg$.
- No “trickery” is allowed for atomic operations:
 - ▶ no operator overload,
 - ▶ no non-scalar types,
 - ▶ no complex expressions.

Parallel Reduction

Reduction Clause in Parallel Region

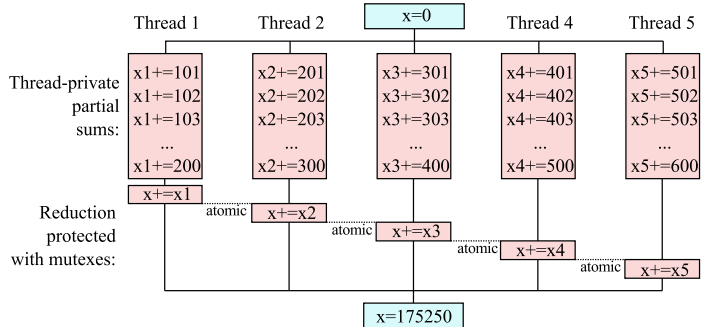
```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main() {
5      const int n = 1000;
6      int total = 0;
7      #pragma omp parallel for reduction(+: total)
8      for (int i = 0; i < n; i++) {
9          total = total + i;
10     }
11     printf("total=%d (must be %d)\n", total, ((n-1)*n)/2);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

```
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:

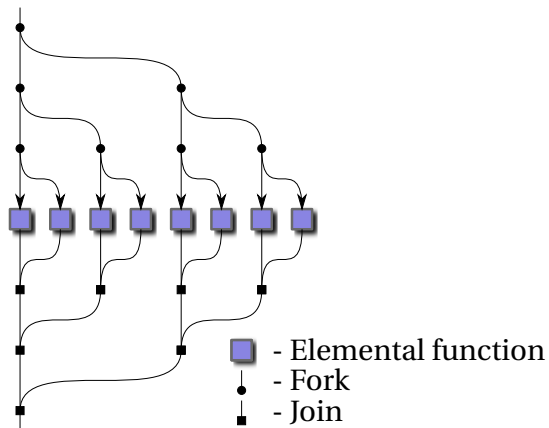
```
1  int total = 0;
2  #pragma omp parallel
3  {
4      int total_thr = 0;
5      #pragma omp for
6      for (int i=0; i<n; i++)
7          total_thr += i;
8
9      #pragma omp atomic
10     total += total_thr;
11 }
12
```



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

```
1 // Starting the first task:
2 #pragma omp parallel
3 { // Enter a parallel region
4 #pragma omp single
5     { // Start the first task
6         // from only one thread
7         RecursiveWorkload(args);
8     }
9 }
```

```
1 // Recursive task spawning:
2 void RecursiveWorkload(Arg* args) {
3     if (args->size > threshold) {
4         // Split work
5         Arg* args1=args->FirstHalf();
6         Arg* args2=args->SecondHalf();
7
8         // Parallel divide-and-conquer
9 #pragma omp task firstprivate(args1)
10    { RecursiveWorkload(args1); }
11 #pragma omp task firstprivate(args2)
12    { RecursiveWorkload(args2); }
13 } else {
14     // End of recursion
15     args->ProcessSmallestSubTask();
16 }
17 }
```

Recipes for Success

Recipe for Success: “Take a Deep Breath”

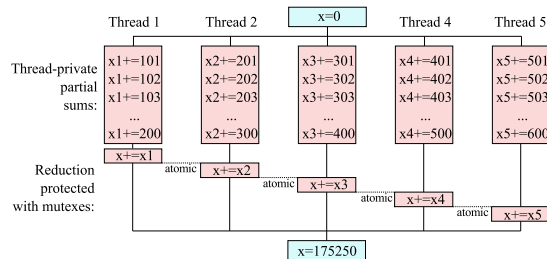
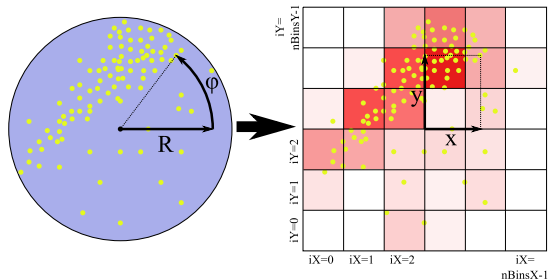
Where is parallelism in your application?



See also full interview with James Reinders at [Colfax Research](#)

Suggested Additional Reading

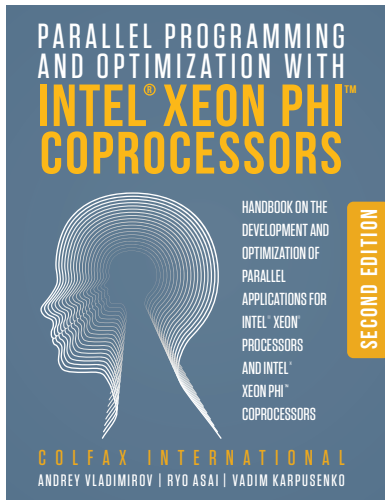
Colfax Research tutorial on multi-threading in a binning code



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

Additional Materials on OpenMP

- ① OpenMP Specifications
- ② Intel's OpenMP Video Course
- ③ LLNL tutorial: [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.