Programming and Optimization for Intel® Architecture

The Hands-On Workshop (HOW) Series

Colfax International — @colfaxintl

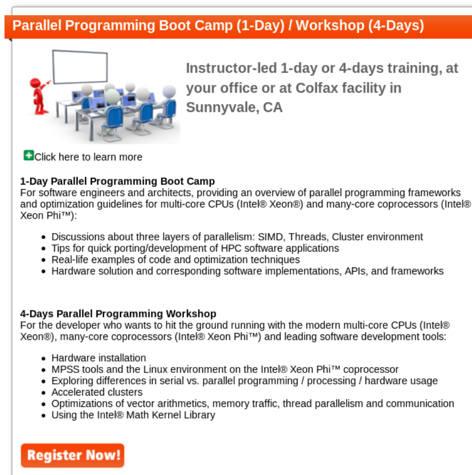
April 2016 , Rev. 02b

 © Colfax International, 2013–2016

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, 2013–2016



colfaxresearch.com/how-series

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

Course Roadmap

1. Why Intel Parallel Architectures?
   - Parallelism and specialization – April 18
   - Programming model continuity – April 18
2. Programming models for Xeon Phi coprocessors
   - Native programming – April 18
   - Offload programming – April 19
3. Expressing Parallelism
   - Introduction to vectorization – April 20
   - Crash-course on OpenMP – April 21
4. Optimization – intro on April 22
   - Vectorization tuning – April 25
   - Multi-threading – April 26, 27
   - Memory traffic – April 28
5. Distributed Computing: MPI – April 29

April 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

— Lecture+remote access

May 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				

— Self-study/remote access

© Colfax International, 2013–2016

HOW Online

Course page: colfaxresearch.com/how-16-04

- 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

© Colfax International, 2013–2016

Developer's Guide to Knights Landing

colfaxresearch.com/knl-webinar/

© Colfax International, 2013–2016

§2. Expressing Data Parallelism

 © Colfax International, 2013–2016

Handling Multiple Cores

 © Colfax International, 2013–2016

Computing Platforms

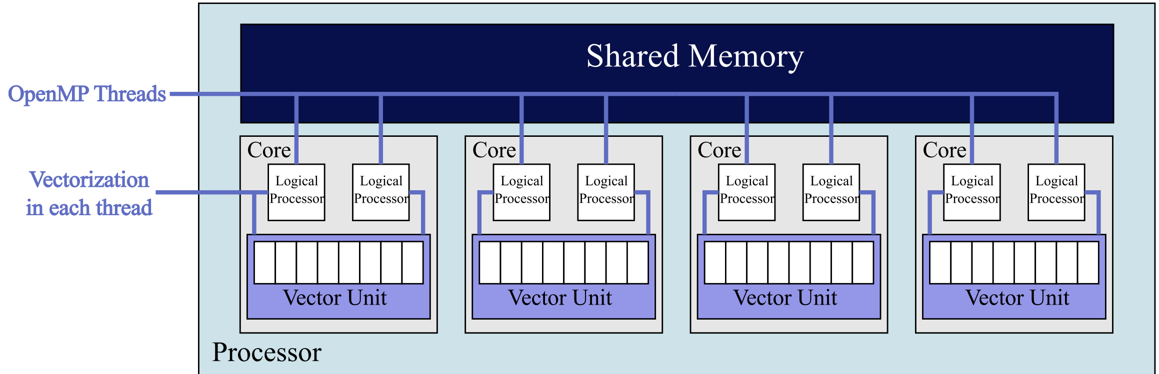
Intel Xeon Processor	Intel Xeon Phi Corocessor, 1st generation	Intel Xeon Phi Processor, 2nd generation*
Current: Broadwell Upcoming: Skylake	Current: Knights Corner (KNC)	* socket and coprocessor versions Upcoming: Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

© Colfax International, 2013–2016

Co-Existence with Vectors



 © Colfax International, 2013–2016

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]);
```

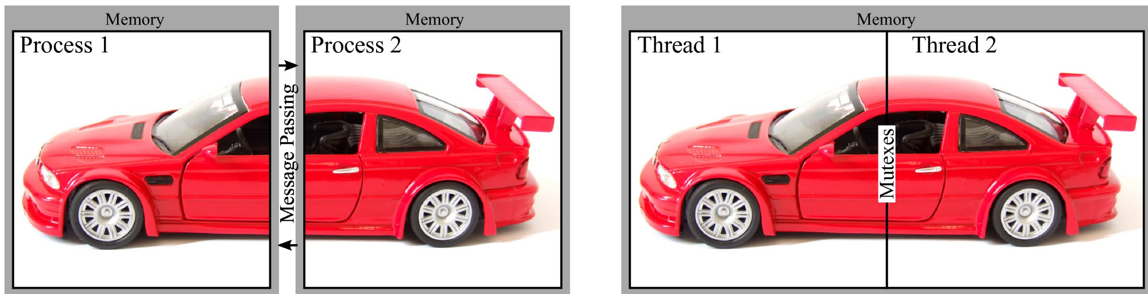
That works as well:

```
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]);
```

 © Colfax International, 2013–2016

Threads versus Processes

Option 1: Partitioning data set between threads/processes

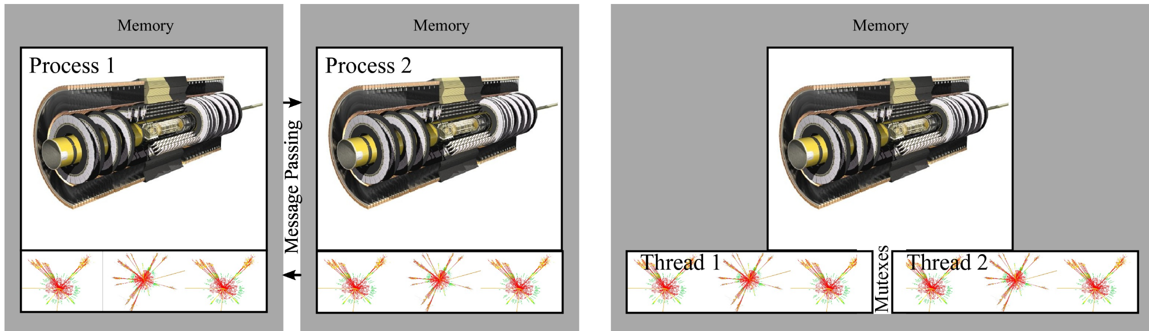


Examples: computational fluid dynamics (CFD), image processing.

© Colfax International, 2013–2016

Threads versus Processes

Option 2: Sharing data set between threads/processes



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

© Colfax International, 2013–2016

Scalability Expectations (CPU)

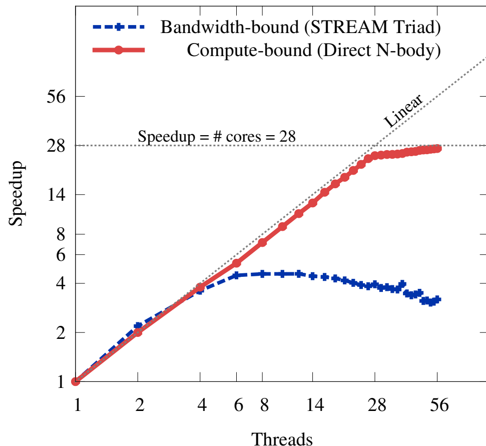
T = number of threads

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

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

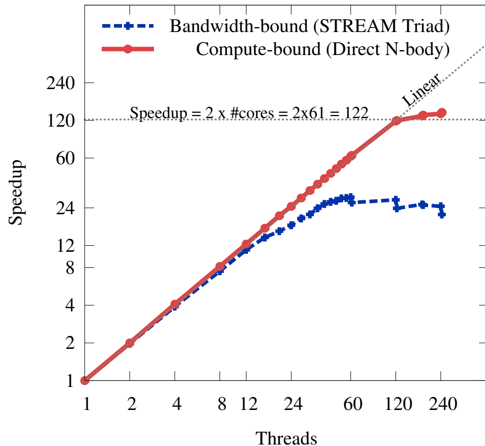
Linear scaling (ideal case, 100% parallel efficiency):

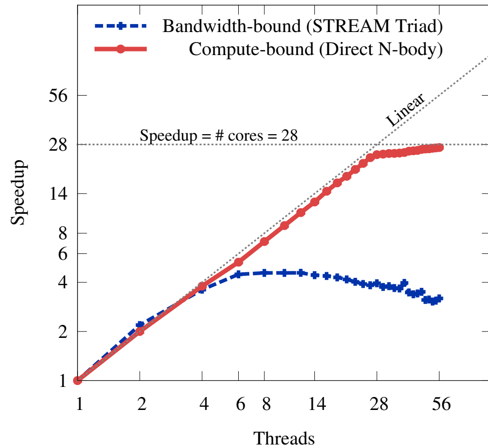
$$\text{Speedup}(T) = T$$



 © Colfax International, 2013–2016

Scalability Expectations: MIC versus CPU





 © Colfax International, 2013–2016

Threading Frameworks

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

 © Colfax International, 2013–2016

OpenMP Basics

© Colfax International, 2013–2016

“Hello World” OpenMP Programs

```
#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());
  }
}
```


© Colfax International, 2013–2016

“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)

 © Colfax International, 2013–2016

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
  }
```

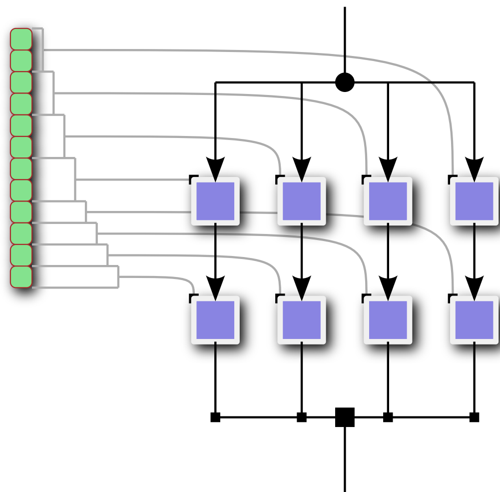
Method 2: using scoping (only C and C++):

```
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
  }
```


© Colfax International, 2013–2016

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.

© Colfax International, 2013–2016

Loop-Centric Parallelism: For-Loops in OpenMP

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

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

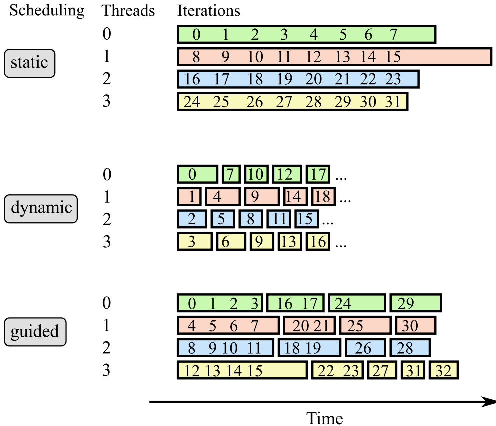
 © Colfax International, 2013–2016

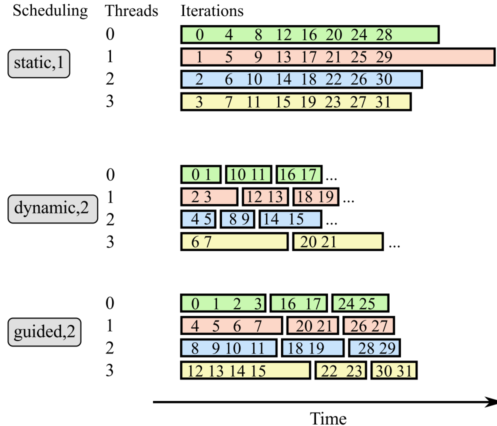
Loop-Centric Parallelism: For-Loops in OpenMP

```
#pragma omp parallel
{
  // Code placed here will be executed by all threads.

  // Alternative way to specify private variables:
  // declare them in the scope of pragma omp parallel
  int private_number=0;

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

 © Colfax International, 2013–2016

Loop Scheduling Modes in OpenMP





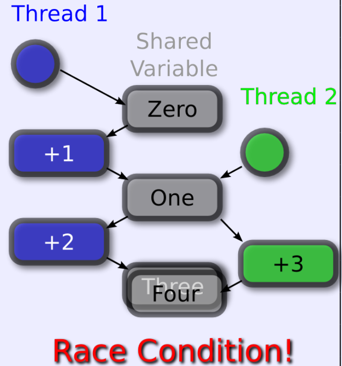
 © Colfax International, 2013–2016

Thread Synchronization

 © Colfax International, 2013–2016

Race Conditions and Unpredictable Program Behavior

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

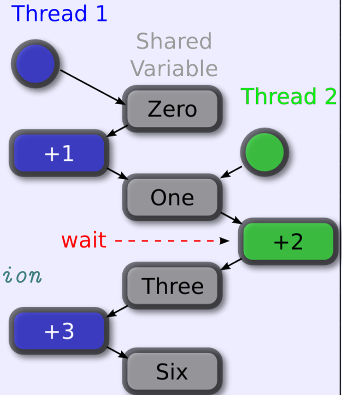



```
vega@lyra% icpc -o omp-race omp-race.cc -qopenmp
vega@lyra% ./omp-race
total=208112 (must be 499500)
```

 © Colfax International, 2013–2016

Protecting Race Conditions with a Critical Section

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




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


© Colfax International, 2013–2016

Avoiding Races with Atomic Operations

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

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

 © Colfax International, 2013–2016

Limitations of Atomic Operations

Read : operations in the form `v = x`

Write : operations in the form `x = v`

Update : operations in the form `x++`, `x--`, `--x`, `++x`, `x` *binop*= *expr* and `x = x` *binop expr*

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

- Here `x` and `v` are scalar variables
- *binop* is one of `+, *, -, - /, &, ^ , |, «, »`.
- No “trickery” is allowed for atomic operations:
  - no operator overload,
  - no non-scalar types,
  - no complex expressions.

© Colfax International, 2013–2016

Parallel Reduction

 © Colfax International, 2013–2016

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 }
```

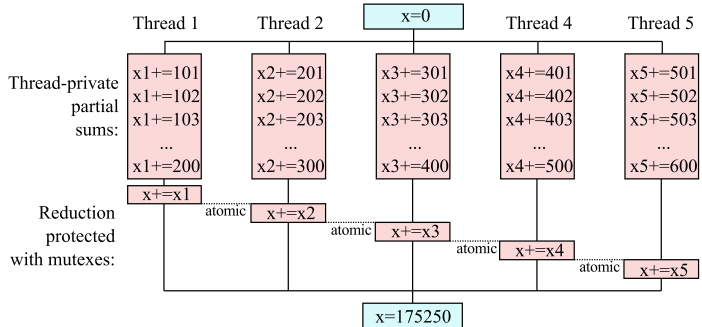
```
vega@lyra% icpc -o omp-reduction omp-reduction.cc -qopenmp
vega@lyra% ./omp-reduction
total=499500 (must be 499500)
```

 © Colfax International, 2013–2016

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 }
```



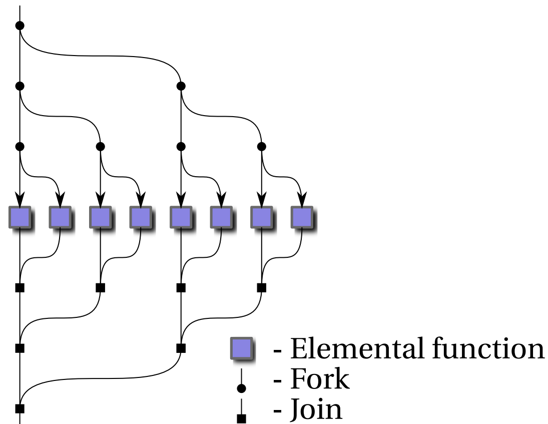
 © Colfax International, 2013–2016

Tasks in OpenMP

 © Colfax International, 2013–2016

Fork-Join Model of Parallel Execution

- Each thread can spawn daughter threads
- Available threads pick up queued tasks
- Expresses algorithms that cannot be expressed in the loop model (e.g., parallel recursion)



Fork-join model of parallel execution.

(`#pragma omp task` functionality)

 © Colfax International, 2013–2016

Tasks in OpenMP: Example

```
// Starting the first task:
#pragma omp parallel
{ // Enter a parallel region
#pragma omp single
  { // Start the first task
    // from only one thread
    RecursiveWorkload(args);
  }
}
```

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

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

 © Colfax International, 2013–2016

Recipes for Success

© Colfax International, 2013–2016

Recipe for Success: “Take a Deep Breath”

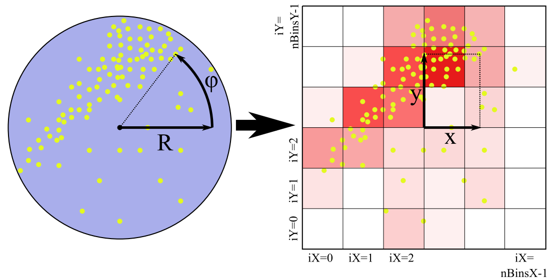
Where is parallelism in your application?

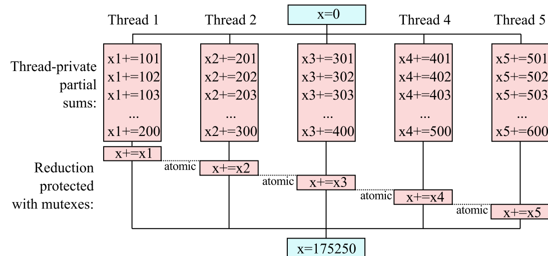
See also full interview with James Reinders at Colfax Research

© Colfax International, 2013–2016

Suggested Additional Reading

Colfax Research tutorial on multi-threading in a binning code



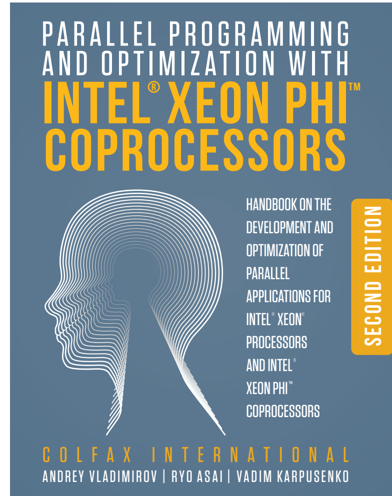


http://colfaxresearch.com/?p=6

© Colfax International, 2013–2016

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.



 © Colfax International, 2013–2016

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.

 © Colfax International, 2013–2016