PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

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

Session 4

Colfax International — *colfaxresearch.com*

© Colfax International, 2013–2017

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, 2013–2017

Course Roadmap

- Module I. Programming Models
  - 01. Intel Architecture and Modern Code
  - 02. Xeon Phi, Coprocessors, Omni-Path
- Module II. Expressing Parallelism
  - 03. Automatic vectorization
  - 04. Multi-threading with OpenMP
  - 05. Distributed Computing, MPI
- Module III. Performance Optimization
  - 06. Optimization Overview: N-body
  - 07. Scalar tuning, Vectorization
  - 08. Common Multi-threading Problems
  - 09. Multi-threading, Memory Aspect
  - 10. Access to Caches and Memory

Course Roadmap
© Colfax International, 2013–2017

HOW SERIES ONLINE

Course page:
colfaxresearch.com/how-series

- Slides
- Code
- Video
- Chat

More workshops:
colfaxresearch.com/training

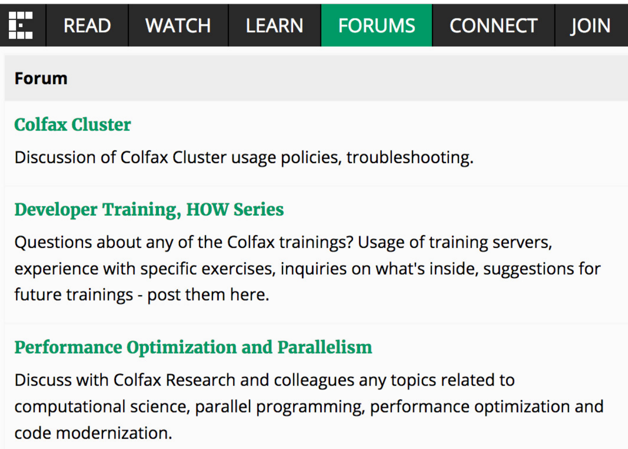
© Colfax International, 2013–2017

GET YOUR QUESTIONS ANSWERED: CHAT

colfaxresearch.com/how-series

© Colfax International, 2013–2017

GET YOUR QUESTIONS ANSWERED: FORUMS

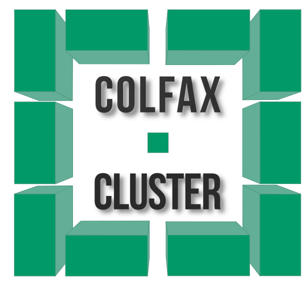


colfaxresearch.com/forum

© Colfax International, 2013–2017

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



© Colfax International, 2013–2017

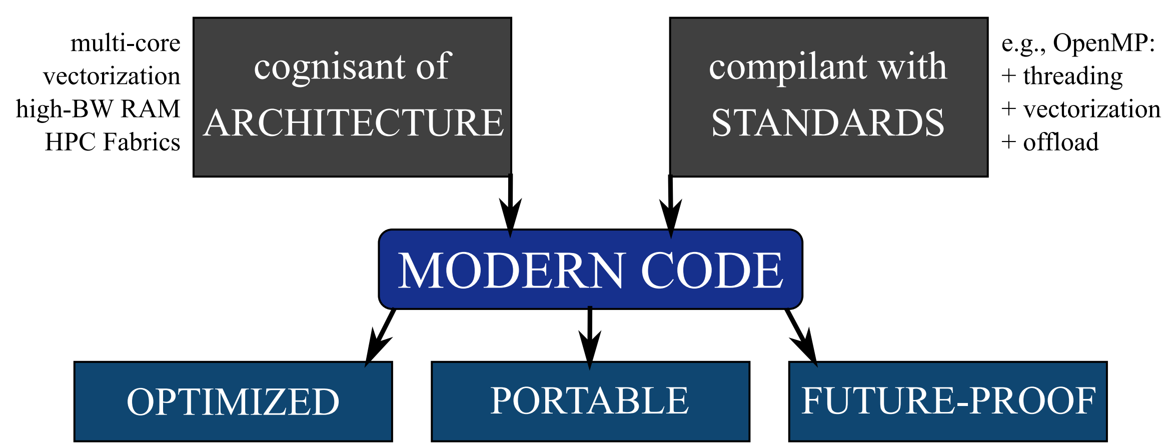
§2. EXPRESSING TASK PARALLELISM

COMPUTING PLATFORMS

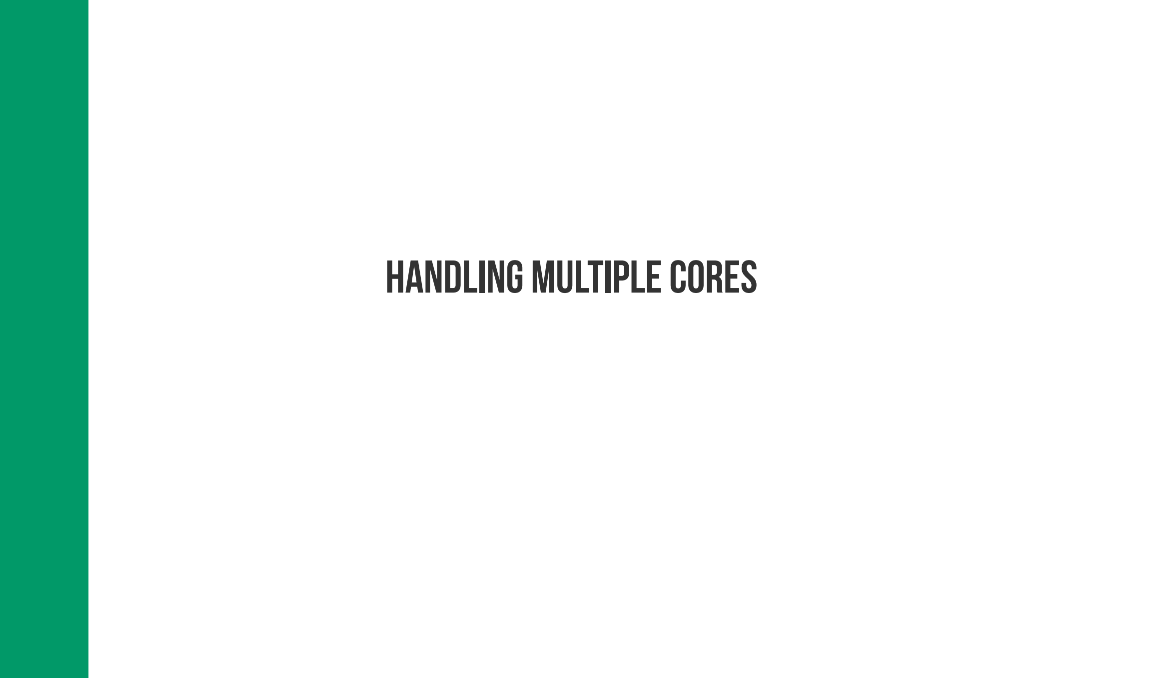
Intel Xeon Processor	Intel Xeon Phi Coprocessor, 1st generation	Intel Xeon Phi Processor, 2nd generation*
Current: Broadwell Upcoming: Skylake	Knights Corner (KNC)	* socket and coprocessor versions Knights Landing (KNL)
Multi-Core Architecture	Intel Many Integrated Core (MIC) Architecture	

© Colfax International, 2013–2017

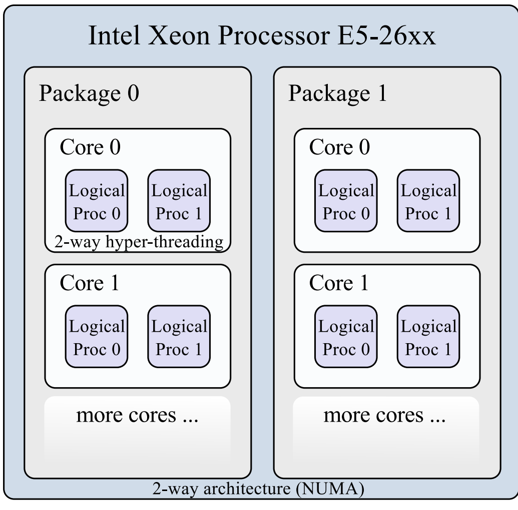
ONE CODE FOR ALL PLATFORMS



© Colfax International, 2013–2017


HANDLING MULTIPLE CORES

PROCESSOR HIERARCHY



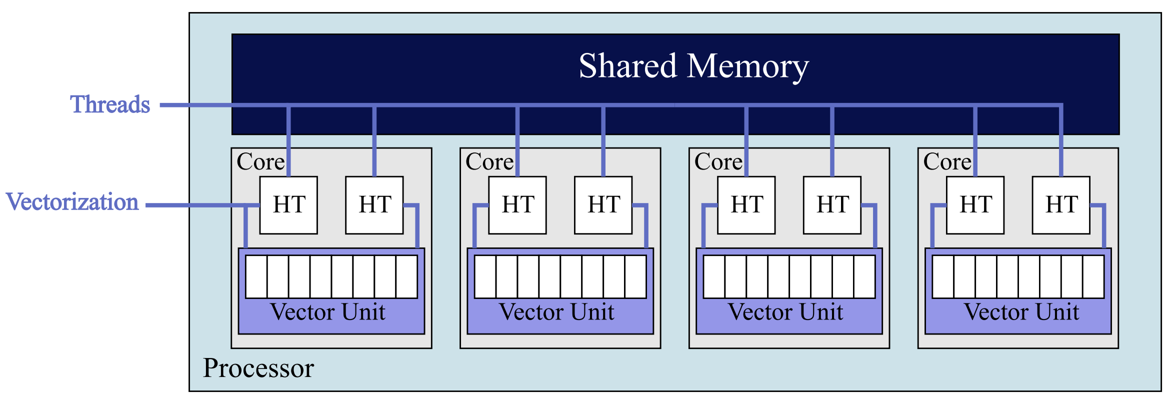
Hierarchy:
Packages ->
Cores ->
Logical processors

OS Proc = numerical ID of logical processor

Jargon:
"socket" = **package**
"logical core" =
"hyper-thread" =
"hardware thread" =
logical processor

© Colfax International, 2013–2017

CO-EXISTENCE WITH VECTORS



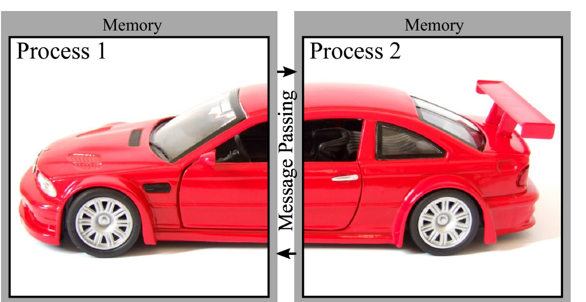
Utilize cores: run multiple threads/processes (MIMD)

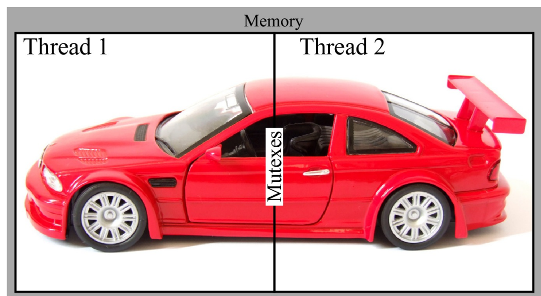
Utilize vectors: each thread (process) issues vector instructions (SIMD)

© Colfax International, 2013–2017

THREADS VERSUS PROCESSES

Option 1: Partitioning data set between threads/processes



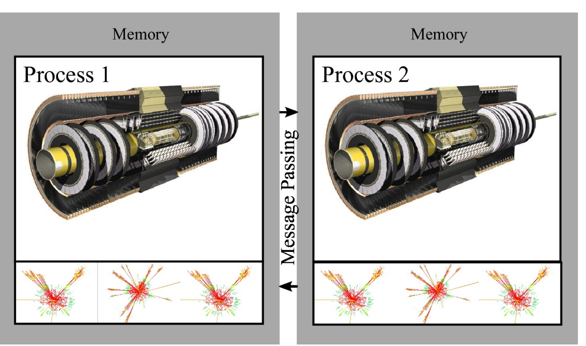


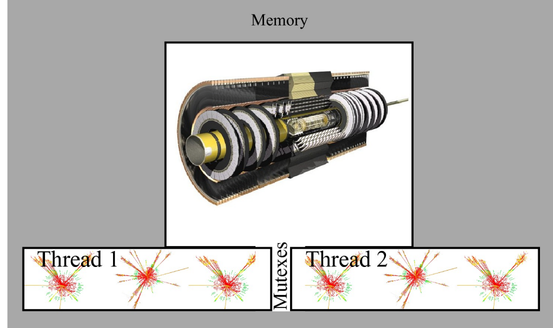
Examples: computational fluid dynamics (CFD), image processing.

© Colfax International, 2013–2017

THREADS VERSUS PROCESSES

Option 2: Sharing data set between threads/processes





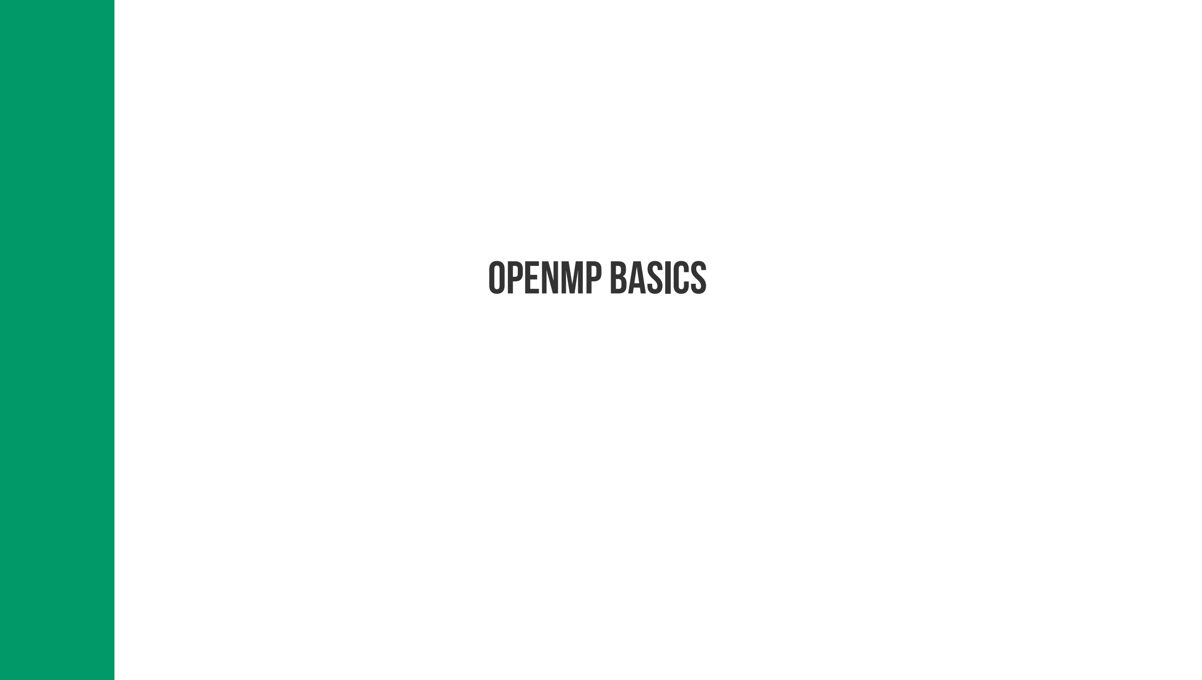
Examples: particle transport simulation, machine learning (inference).

© Colfax International, 2013–2017

THREADING FRAMEWORKS

Framework	Functionality
C++11 Threads	Asynchronous functions; only C++
POSIX Threads	Fork/join; C/C++/Fortran; Linux
Cilk Plus	Async tasks, loops, reducers, load balance; C/C++
TBB	Trees of tasks, complex patterns; only C++
OpenMP	Tasks, loops, reduction, load balancing, affinity, nesting, C/C++/Fortran (+SIMD, offload)

© Colfax International, 2013–2017


OPENMP BASICS

"HELLO WORLD" OPENMP PROGRAM

```
1  #include <omp.h>
2  #include <cstdio>
3
4  int main(){
5    // This code is executed by 1 thread
6    const int nt=omp_get_max_threads();
7    printf("OpenMP with %d threads\n", nt);
8
9  #pragma omp parallel
10   { // This code is executed in parallel
11     // by multiple threads
12     printf("Hello World from thread %d\n",
13                     omp_get_thread_num());
14   }
15 }
```

- 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, 2013–2017

COMPILING THE "HELLO WORLD" OPENMP PROGRAM

```
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, 2013–2017

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   }
```

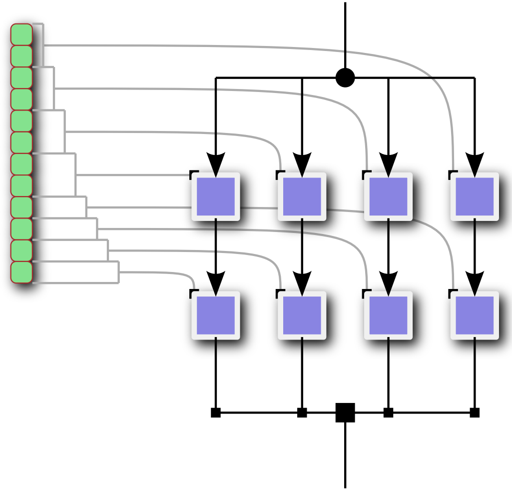
Method 2: using scoping (only C and C++):

```
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   }
```


© Colfax International, 2013–2017

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, 2013–2017

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, 2013–2017

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, 2013–2017

LOOP SCHEDULING MODES IN OPENMP

Scheduling	Threads	Iterations
static	0	0 1 2 3 4 5 6 7
	1	8 9 10 11 12 13 14 15
	2	16 17 18 19 20 21 22 23
	3	24 25 26 27 28 29 30 31
dynamic	0	0 7 10 12 17 ...
	1	1 4 9 14 18 ...
	2	2 5 8 11 15 ...
	3	3 6 9 13 16 ...
guided	0	0 1 2 3 16 17 24 29
	1	4 5 6 7 20 21 25 30
	2	8 9 10 11 18 19 26 28
	3	12 13 14 15 22 23 27 31 32

Time

Scheduling	Threads	Iterations
static,1	0	0 4 8 12 16 20 24 28
	1	1 5 9 13 17 21 25 29
	2	2 6 10 14 18 22 26 30
	3	3 7 11 15 19 23 27 31
dynamic,2	0	0 1 10 11 16 17 ...
	1	2 3 12 13 18 19 ...
	2	4 5 8 9 14 15 ...
	3	6 7 20 21 ...
guided,2	0	0 1 2 3 16 17 24 25
	1	4 5 6 7 20 21 26 27
	2	8 9 10 11 18 19 28 29
	3	12 13 14 15 22 23 30 31

Time

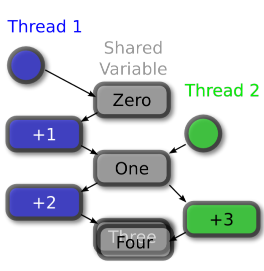
© Colfax International, 2013–2017

THREAD SYNCHRONIZATION

RACE CONDITIONS AND UNPREDICTABLE PROGRAM BEHAVIOR

```
1  #include <omp.h>
2  #include <cstdio>
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      total = total + i; // Race condition
9    }
10   printf("total=%d (must be %d)\n", total,
11                                ((n-1)*n)/2);
12 }
```

```
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, 2013–2017

PROTECTING RACE CONDITIONS WITH A CRITICAL SECTION

```
1  #include <omp.h>
2  #include <cstdio>
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 } }
```

Thread 1
Shared Variable
Thread 2
Zero
+1
One
wait
+2
Three
+3
Six

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


© Colfax International, 2013–2017

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 }
```


© Colfax International, 2013–2017

LIMITATIONS OF ATOMIC OPERATIONS

Read : operations in the form `v = x`

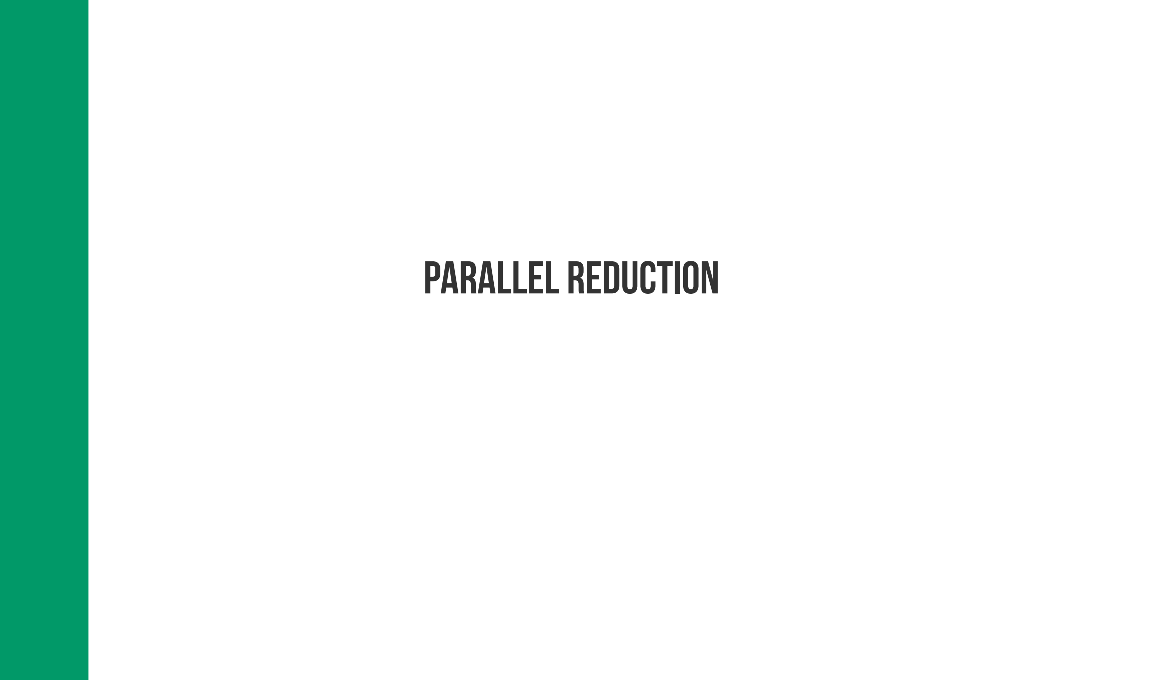
Write : operations in the form `x = v`

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, 2013–2017


PARALLEL REDUCTION

REDUCTION CLAUSE IN PARALLEL REGION

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

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)
```


© Colfax International, 2013–2017

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;

}
```

Thread 0
Thread 1
total=0
Thread 3
Thread 4

Thread-private partial sums:

x0+= 0	x1+=200	x2+=400	x3+=600	x4+=800
x0+= 1	x1+=201	x2+=401	x3+=601	x4+=800
x0+= 2	x1+=202	x2+=402	x3+=602	x4+=800
...	...	...	...	...
x0+=199	x1+=399	x2+=599	x3+=799	x4+=999

Reduction protected with mutexes:

total+=x0
atomic total+=x1
atomic total+=x2
atomic total+=x3
atomic total+=x4

total=499500

© Colfax International, 2013–2017

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]);
```


© Colfax International, 2013–2017

SIMULTANEOUS THREADING AND VECTORIZATION

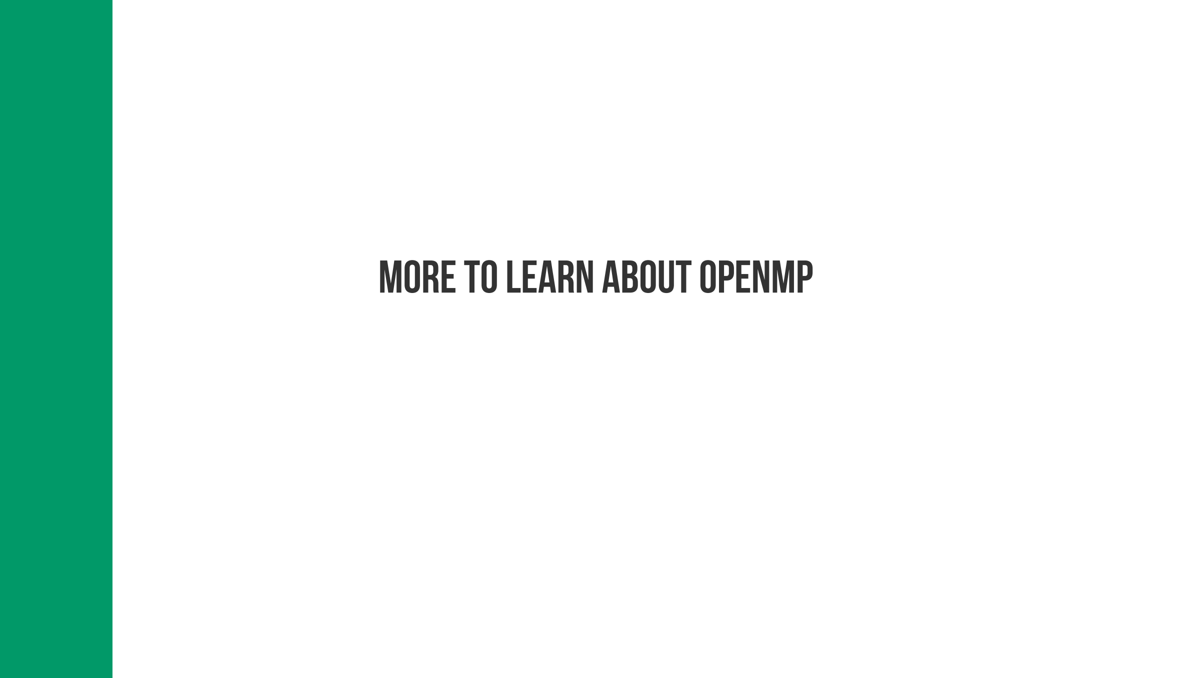
Sometimes the compiler may need a little help:

```
const int STRIP_SIZE = 128; // A multiple of vector length
const int nTrunc = n - n%STRIP_SIZE; // A multiple of vector length

#pragma omp parallel for
for (int ii = 0; ii < nTrunc; ii += STRIP_SIZE) // Thread parallelism in outer
#pragma simd
  for (int i = ii; i < ii + STRIP_SIZE; i++) // Vectorization in inner loop
    DoSomeWork(A[i]);

// Remainder loop:
for (int i = nTrunc; i < n; i++)
  DoSomeWork(A[i]);
```


© Colfax International, 2013–2017


MORE TO LEARN ABOUT OPENMP

OPENMP CONCEPTS AND CONSTRUCTS

`#pragma omp parallel` – create threads

`#pragma omp for` – process loop with threads

`#pragma omp task/taskyield` – asynchronous tasks

`#pragma omp critical/atomic` – mutexes

`#pragma omp barrier/taskwait` – synchronization points

`#pragma omp sections/single` – blocks of code for individual threads

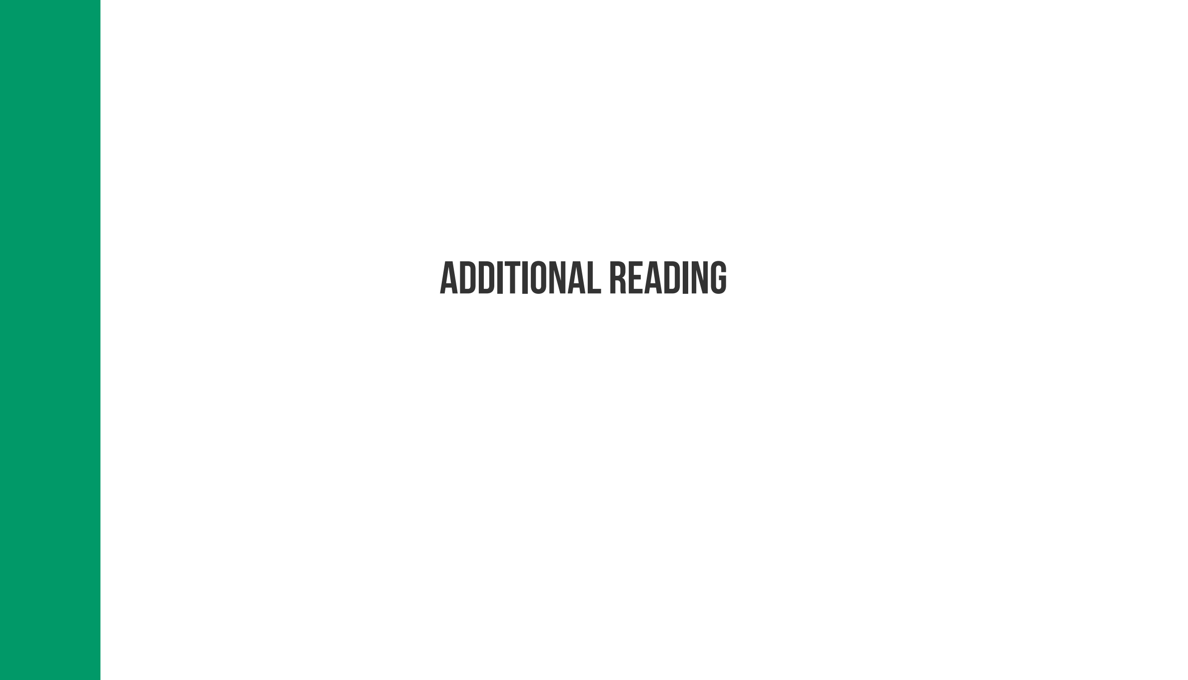
`#pragma omp flush` – enforce memory consistency

`#pragma omp ordered` – partial loop serialization

`OMP_*` – environment variables, omp_*() – functions

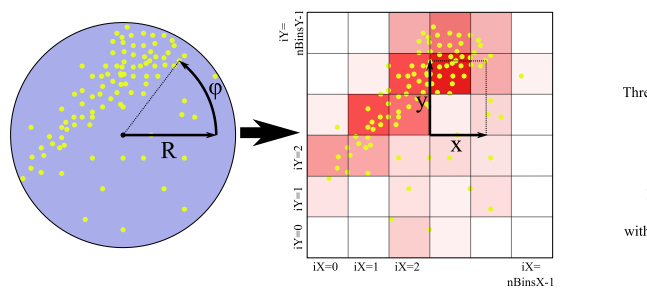
Click construct names for links to the OpenMP reference from the LLNL

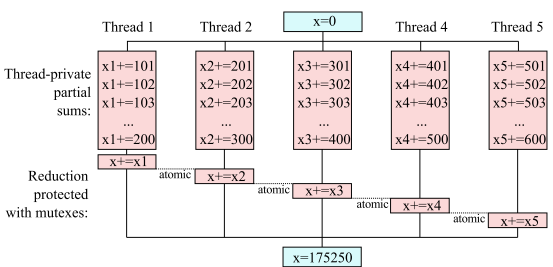
© Colfax International, 2013–2017


ADDITIONAL READING

SUGGESTED ADDITIONAL READING

Colfax Research tutorial on multi-threading in a binning code



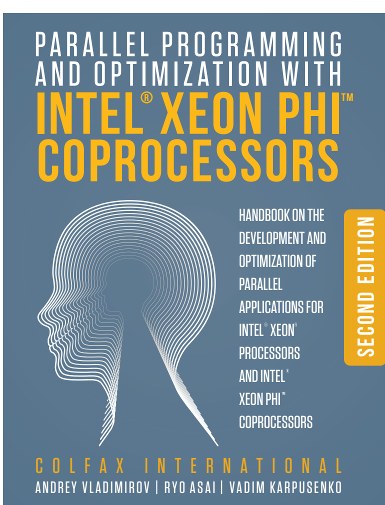


http://colfaxresearch.com/?p=6

© Colfax International, 2013–2017

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.



© Colfax International, 2013–2017

SUMMARY

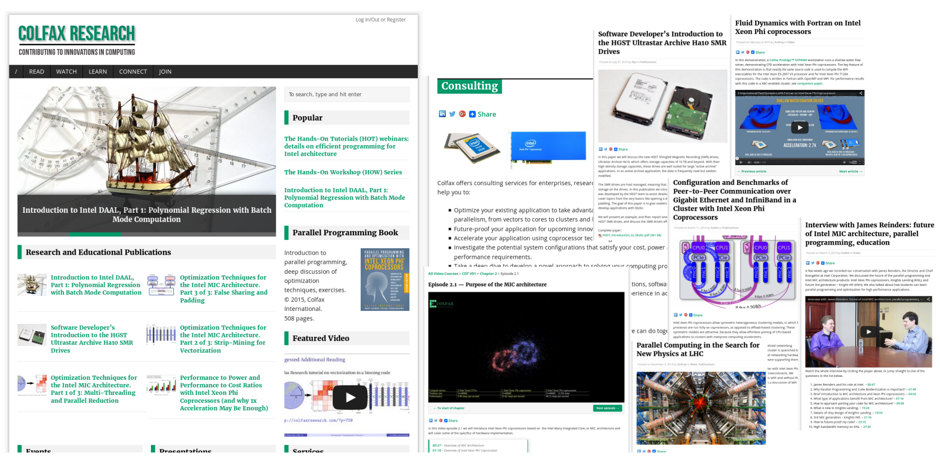
Discussed today:

- Cores can run independent programs
- Use threads to scale across cores
- OpenMP – well-established parallel framework for HPC
- Data races lead to incorrect, unpredictable results
- Mutexes control data races at cost of performance
- Co-exist with have vectorization in each thread

Next session: distributed-memory computing with MPI.

© Colfax International, 2013–2017

COLFAX RESEARCH



https://colfaxresearch.com/