PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

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

Session 8

Colfax International — *colfaxresearch.com*

February 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.

COURSE ROADMAP

- Module I. Programming
  - 01. Intel Architecture and Modern Code – Feb 13
  - 02. Xeon Phi, Coprocessors, Omni-Path – Feb 14
- Module II. Expressing Parallelism
  - 03. Automatic vectorization – Feb 15
  - 04. Multi-threading with OpenMP – Feb 16
  - 06. Distributed Computing, MPI – Feb 17
- Module III. Optimization
  - 06. Optimization Overview: N-body – Feb 20
  - 07. Scalar tuning, Vectorization – Feb 21
  - 08. Common Multi-threading Problems – Feb 22
  - 09. Multi-threading, Memory Aspect – Feb 23
  - 10. Access to Caches and Memory – Feb 24

February 2017

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				

— Webinar+remote access

HOW SERIES ONLINE

Course page:
colfaxresearch.com/how-17-02

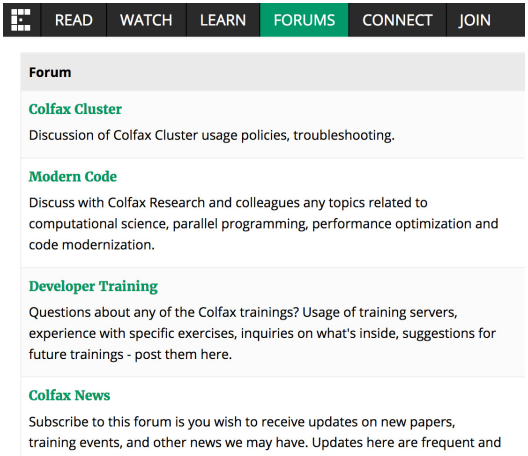
- Slides
- Code
- Video
- Chat

More workshops:
colfaxresearch.com/training

GET YOUR QUESTIONS ANSWERED: CHAT

colfaxresearch.com/how-17-02

GET YOUR QUESTIONS ANSWERED: FORUMS



colfaxresearch.com/discussion

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. REFRESH

PERFORMANCE OPTIMIZATION

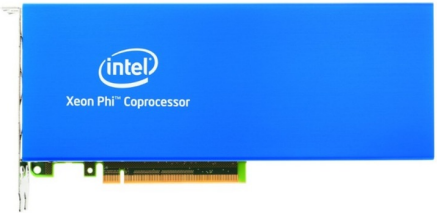
COMPUTING PLATFORMS

Intel Xeon Processor

Current: Broadwell
Upcoming: Skylake

Multi-Core Architecture

Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

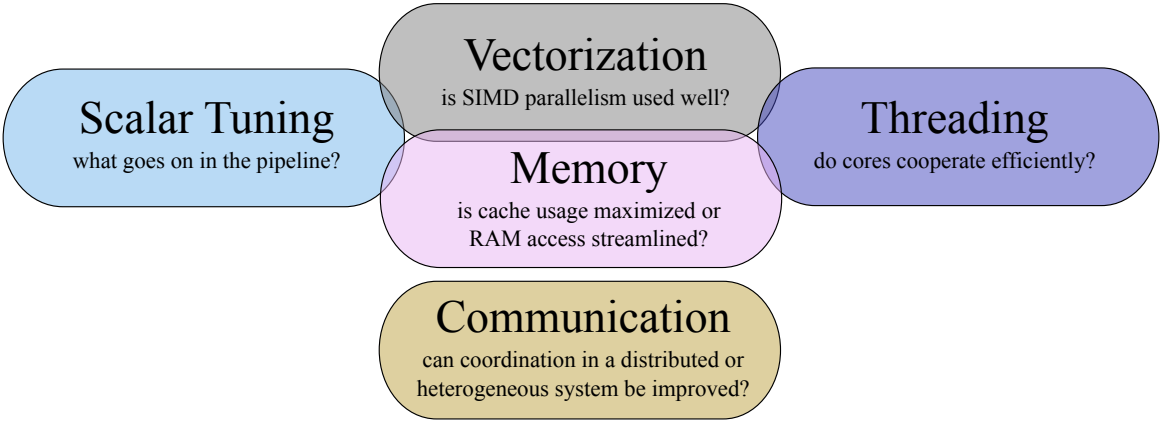
Intel Xeon Phi Processor, 2nd generation*

* socket and coprocessor versions

Knights Landing (KNL)

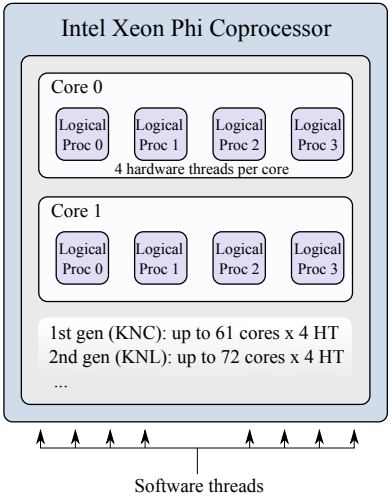
Intel Many Integrated Core (MIC) Architecture

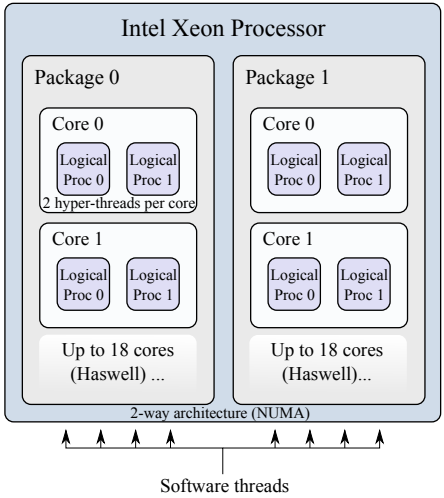
OPTIMIZATION AREAS



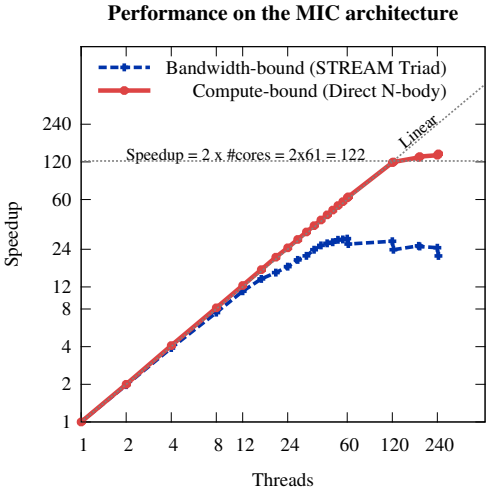
CORES, THREADS AND OPENMP

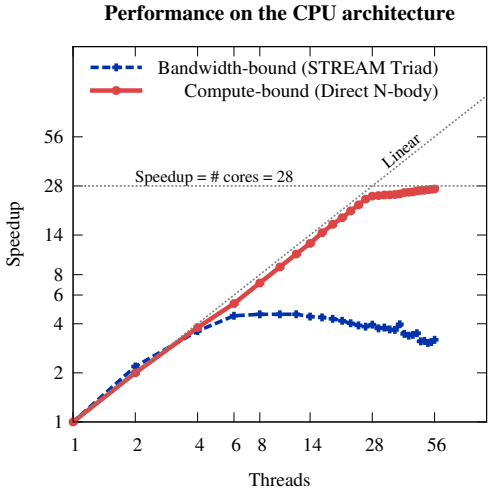
PROCESSOR HIERARCHY





SCALABILITY EXPECTATIONS: MIC VERSUS CPU





"HELLO WORLD" OPENMP PROGRAM

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

int main(){
  // This code is executed by only 1 thread
  const int nt=omp_get_max_threads();
  printf("OpenMP with %d threads\n", nt);

#pragma omp parallel
  {
    // This code is executed in parallel
    // by multiple threads
    printf("Hello World from thread %d\n",
                        omp_get_thread_num());
  }
}
```

- 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

§3. Multi-threading: Common Issues

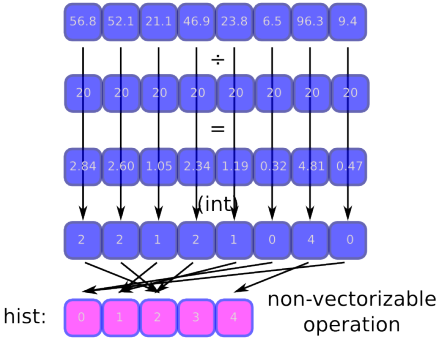
TOO MUCH SYNCHRONIZATION

EXAMPLE: BINNING PROBLEM

Computing a histogram ($m << n$):

```
1  void Histogram(
2    // Ages, values from 0.0f to 100.0f:
3    const float* age,
4    // Size of array age, n=100000000:
5    const int n,
6    // Output: counts in groups:
7    int* const hist,
8    // Size of array hist, m=5:
9    const int m,
10   const float group_width) {
11     for (int i = 0; i < n; i++) {
12       const int j = int(age[i]/group_width);
13       hist[j]++;
14     }
15 }
```

- Vector dependence in `hist[j]++`
- Strip-mine or use conflict detection

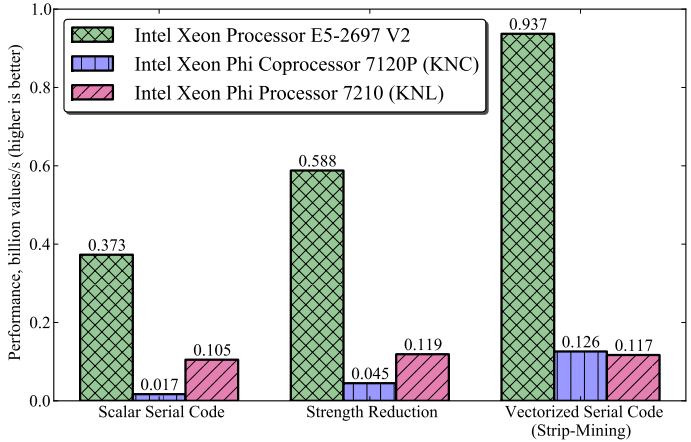


THE SAME CALCULATION, STRIP-MINED, VECTORIZED

```
1  void Histogram(const float* age, int* const hist, const int n,
2  const float group_width, const int m) {
3    const int vecLen = 16; // Length of vectorized loop
4    const float invGroupWidth = 1.0f/group_width; // Pre-compute the reciprocal
5    // Strip-mining the loop in order to vectorize the inner short loop
6    // Note: this algorithm assumes n%vecLen == 0.
7    for (int ii = 0; ii < n; ii += vecLen) { //Temporary store vecLen indices
8      int index[vecLen] __attribute__((aligned(64)));
9      // Vectorize the multiplication and rounding
10 #pragma vector aligned
11     for (int i = ii; i < ii + vecLen; i++)
12       index[i-ii] = (int) ( age[i] * invGroupWidth );
13     // Scattered memory access, does not get vectorized
14     for (int c = 0; c < vecLen; c++)
15       hist[index[c]]++;
16   }
17 }
```

STRIP-MINING FOR VECTORIZATION

Vectorization improves performance on both platforms. However, more work is needed to take advantage of the MIC architecture. See materials on multi-threading.



TOO MUCH SYNCHRONIZATION

HISTOGRAM CALCULATION EXAMPLE: ADDING THREAD PARALLELISM

Incorrect solution: unprotected data races

```
#pragma omp parallel for schedule(guided)
for (int ii = 0; ii < n; ii += vecLen) {
  int index[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
  for (int i = ii; i < ii + vecLen; i++)
    index[i-ii] = (int) ( age[i] * invGroupWidth );
  for (int c = 0; c < vecLen; c++)
    // Multiple threads will write into a single shared container
    // These data races lead to incorrect results!
    hist[index[c]]++;
}
```

HISTOGRAM CALCULATION EXAMPLE: ADDING THREAD PARALLELISM

Correct, but inefficient solution:

```
#pragma omp parallel for schedule(guided)
for (int ii = 0; ii < n; ii += vecLen) {
  int index[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
  for (int i = ii; i < ii + vecLen; i++)
    index[i-ii] = (int) ( age[i] * invGroupWidth );
  for (int c = 0; c < vecLen; c++)
    // Protect the ++ operation with the atomic mutex (inefficient!)
#pragma omp critical
    { hist[index[c]]++; }
}
```

HISTOGRAM CALCULATION EXAMPLE: ADDING THREAD PARALLELISM

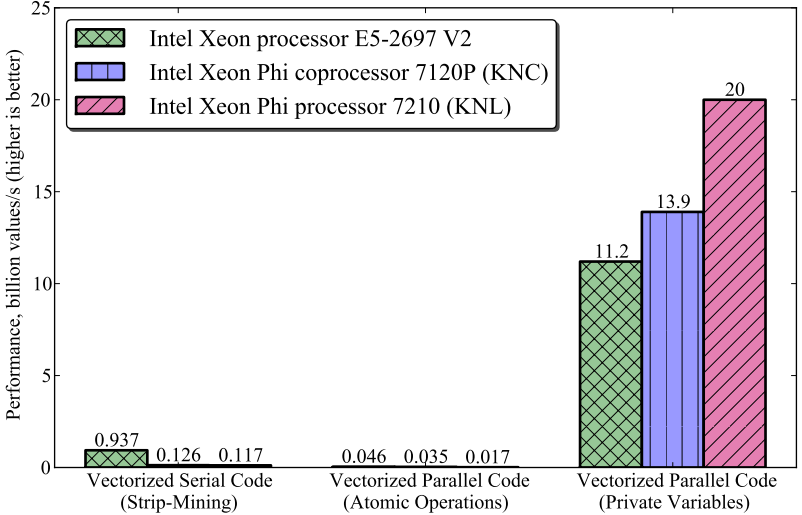
Correct, but inefficient solution:

```
#pragma omp parallel for schedule(guided)
for (int ii = 0; ii < n; ii += vecLen) {
  int index[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
  for (int i = ii; i < ii + vecLen; i++)
    index[i-ii] = (int) ( age[i] * invGroupWidth );
  for (int c = 0; c < vecLen; c++)
    // Protect the ++ operation with the atomic mutex (inefficient!)
#pragma omp atomic
    hist[index[c]]++;
}
```

CORRECT AND EFFICIENT SOLUTION WITH REDUCTION

```
#pragma omp parallel
{
  int hist_priv[m]; // Better idea: thread-private storage
  hist_priv[:] = 0;
  int index[vecLen] __attribute__((aligned(64)));
#pragma omp for schedule(guided)
  for (int ii = 0; ii < n; ii += vecLen) {
#pragma vector aligned
    for (int i = ii; i < ii + vecLen; i++)
      index[i-ii] = (int) ( age[i] * invGroupWidth );
    for (int c = 0; c < vecLen; c++)
      hist_priv[index[c]]++;
  }
  for (int c = 0; c < m; c++) {
#pragma omp atomic
    hist[c] += hist_priv[c];
} }
```

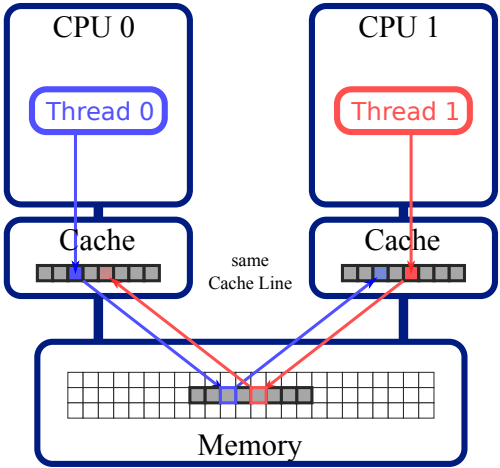
USING REDUCTION INSTEAD OF SYNCHRONIZATION



TOO MUCH SYNCHRONIZATION

FALSE SHARING

FALSE SHARING. DATA PADDING AND PRIVATE VARIABLES



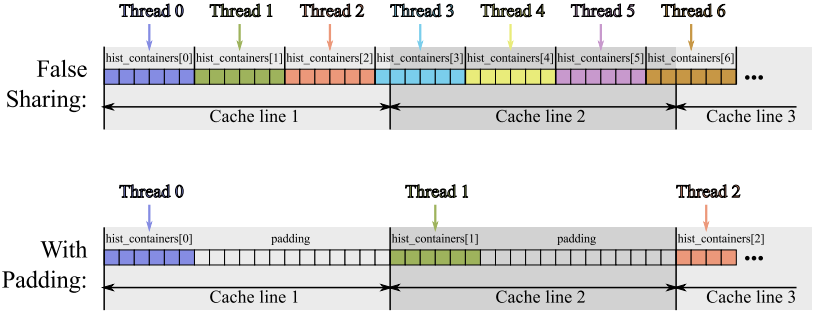
- Occurs when 2 or more threads acess the same cache line, and at least one of the accesses is for writing
- Caused by *coherent caches*
- Cache line is 64-byte wide (in modern Intel architectures)

FALSE SHARING. DATA PADDING AND PRIVATE VARIABLES

```
1  const int m = 5;
2  int hist_thr[nThreads][m];
3  #pragma omp parallel for
4  for (int ii = 0; ii < n; ii += vecLen) {
5    // ...
6    // False sharing occurs here
7    for (int c = 0; c < vecLen; c++)
8      hist_thr[iThread][index[c]]++;
9  }
10 // Reducing results from all threads to the common histogram hist
11 for (int iThread = 0; iThread < nThreads; iThread++)
12   hist[0:m] += hist_thr[iThread][0:m];
```

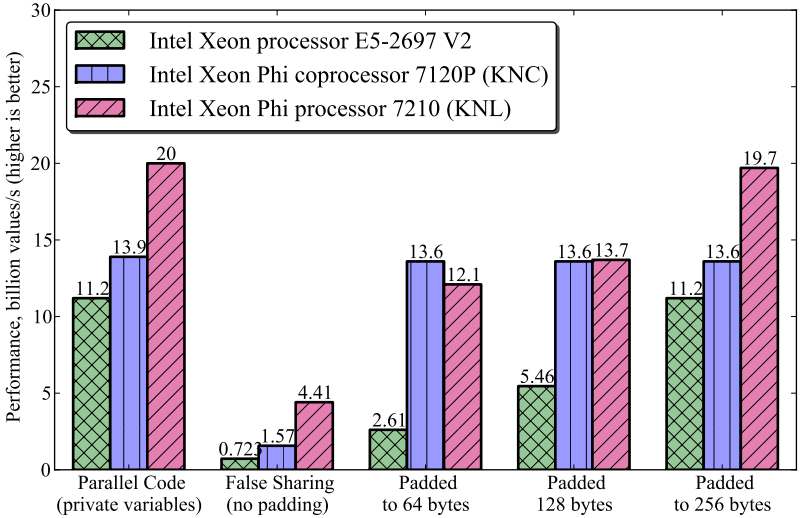
- The value of `m=5` is small
- Array elements `hist_thr[0][:]` are within `m*sizeof(int)=20` bytes of array elements `hist_thr[1][:]`

PADDING TO AVOID FALSE SHARING




```
1 // Padding to avoid sharing a cache line between threads
2 const int paddingBytes = 64;
3 const int paddingElements = paddingBytes / sizeof(int);
4 const int mPadded = m + (paddingElements-m%paddingElements);
5 int hist_containers[nThreads][mPadded]; // New container
```

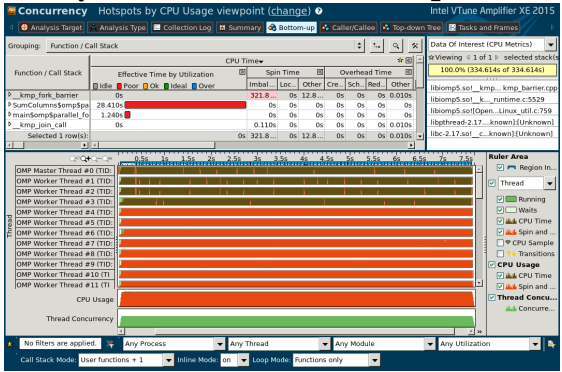
PADDING TO AVOID FALSE SHARING

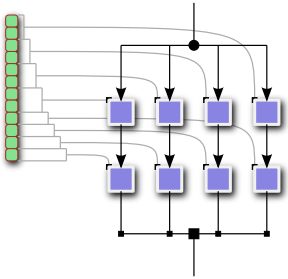


INSUFFICIENT PARALLELISM

INSUFFICIENT PARALLELISM

Analysis in Intel VTune Amplifier XE





- Occurs when there are not enough iterations or parallel work-items exposed to the parallel loop in OpenMP.

EXAMPLE: DEALING WITH INSUFFICIENT PARALLELISM

$$S_i = \sum_{j=0}^{n} M_{ij},\ i = 0 \ldots m. \tag{1}$$

- `m`=4 is small, smaller than the number of threads in the system
- `n` $\approx 10^8$ is large enough so that matrix does not fit into cache

```
1 void sum_unoptimized(const int m, const int n, long* M, long* s){
2 #pragma omp parallel for
3   for (int i=0; i<m; i++) { // m=4
4     long total=0;
5 #pragma vector aligned
6     for (int j=0; j<n; j++) // n=100000000
7       total+=M[i*n+j];
8   s[i]=total; }}
```

DOES NOT WORK: PARALLELIZING INNER LOOP

Inner loop has more iterations, parallelize there?

```
1  void SumParallelInnerLoop(const int m, const int n, long* M, long* s){
2    for (int i = 0; i < m; i++) { // m=4
3      long total = 0;
4  #pragma omp parallel for reduction(+: total)
5      for (int j = 0; j < n; j++) { // n=100000000
6        total += M[i*n + j];
7      }
8      s[i] = total;
9    }
10 }
```

Does not work well: code must spawn and stop threads many times; OpenMP does not see the entire parallel region.

LOOP COLLAPSE: PRINCIPLE

Idea: combine iterations spaces of the inner loop and the outer loop.

```
#pragma omp parallel for collapse(2)
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) {
      // ...
      // ...
    }
```

```
#pragma omp parallel for
  for (int c = 0; c < m*n; c++)  {
    i = c / n;
    j = c % n;
    // ...
  }
```

DOES NOT WORK, BUT CORRECT DIRECTION: LOOP COLLAPSE

Loop collapse applied to the wide short matrix example:

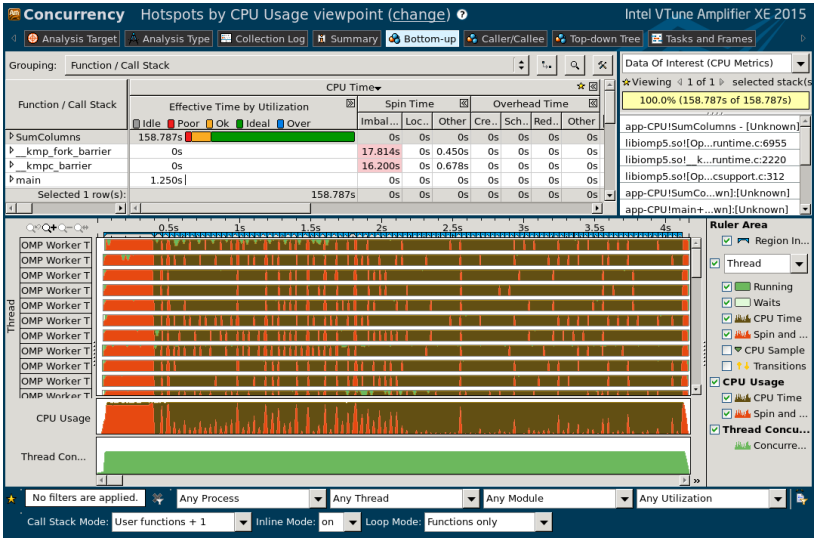
```
1  void SumCollapse(const int m, const int n, long* M, long* s){
2    s[:]=0;
3  #pragma omp parallel
4    { // Each thread will have a private container
5      long total[m];  total[:] = 0;
6  #pragma omp for collapse(2)
7      for (int i = 0; i < m; i++) // m=4
8        for (int j = 0; j < n; j++) // n=100000000
9          total[i] += M[i*n + j];
10     for (int i = 0; i < m; i++)
11 #pragma omp atomic
12       s[i]=total[i];
13 } }
```

Does not work: automatic vectorization fails.

EXPOSING PARALLELISM: STRIP-MINING AND LOOP COLLAPSE

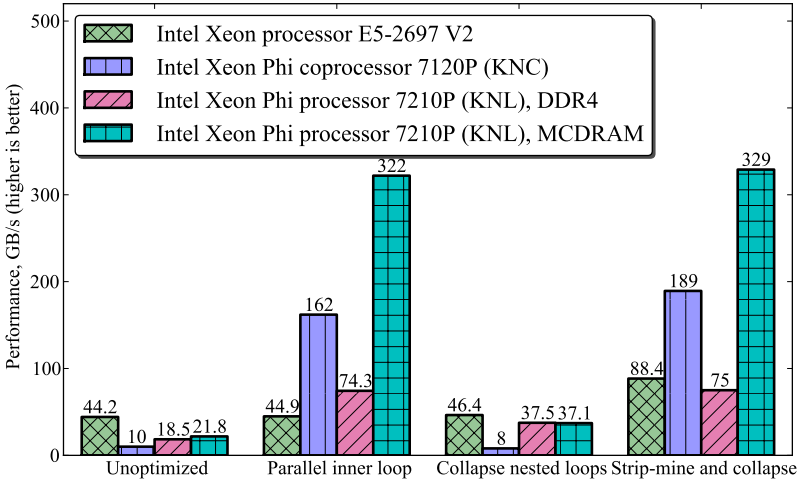
```
1  void sum_stripmine(const int m, const int n, long* M, long* s){
2    const int STRIP=1024;
3    assert(n%STRIP==0);
4    s[0:m]=0;
5  #pragma omp parallel
6    {
7      long total[m];   total[0:m]=0;
8  #pragma omp for collapse(2) schedule(guided)
9      for (int i=0; i<m; i++)
10       for (int jj=0; jj<n; jj+=STRIP)
11 #pragma vector aligned
12         for (int j=jj; j<jj+STRIP; j++)
13           total[i]+=M[i*n+j];
14     for (int i=0; i<m; i++)          // Reduction
15 #pragma omp atomic
16       s[i]+=total[i];
17 }   }
```

EXPOSING PARALLELISM: STRIP-MINING AND LOOP COLLAPSE



INSUFFICIENT PARALLELISM

DEALING WITH INSUFFICIENT PARALLELISM



§4. REVIEW AND WHAT'S NEXT

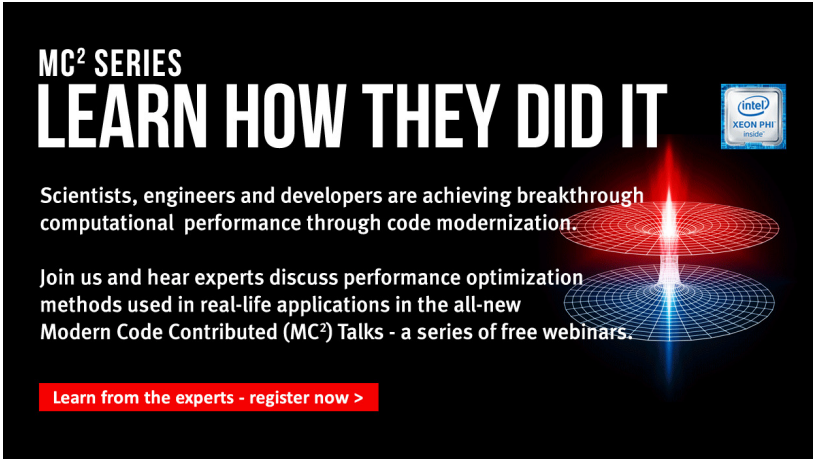
SUMMARY

This session:

1. Synchronization is necessary to resolve data races
2. Mutexes must be moved out of innermost loops
3. False sharing can be resolved with padding
4. Loop collapse can help to expose parallelism
5. Strip-mining to make vectorization co-exist with threading

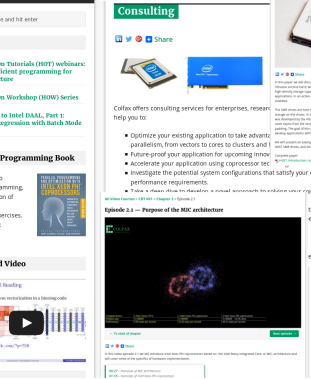
Next session: optimization of thread affinity, NUMA locality, nested parallelism and loop scheduling.

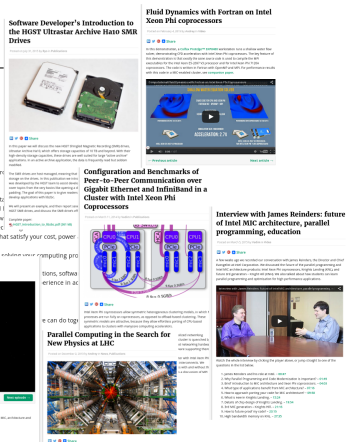
NEW PROJECT: MC^2 SERIES



mc2series.com

COLFAX RESEARCH





http://colfaxresearch.com/