COLFAX

Customized Solutions

PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

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

Session 8

Colfax International — *colfaxresearch.com*

DISCLAIMER

COURSE ROADMAP

HOW SERIES ONLINE

Course page:
colfaxresearch.com/how-series

- ▷ Slides
- ▷ Code
- ▷ Video
- ▷ Chat

More workshops:
colfaxresearch.com/training

GET YOUR QUESTIONS ANSWERED: CHAT

leofernandesmo Hello from Recife/Brazil

gaesansi Hi, Naples, Italy

info2harish Harish f rom INDIA

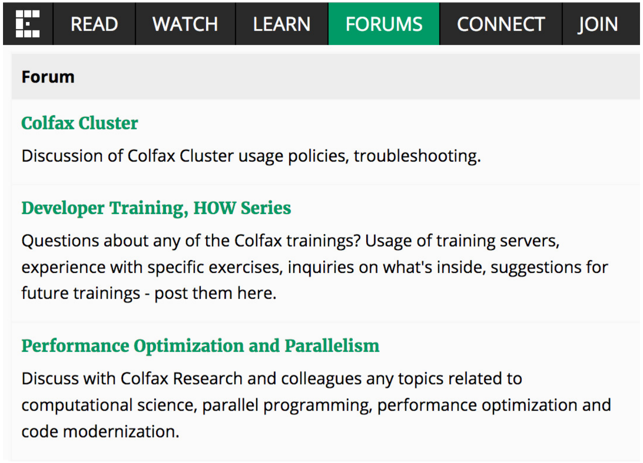
hpcfan Hello, from Texas.

radekg1000 Hi, Poznan/Poland

zanton hello, Tokyo, JP

colfaxresearch.com/how-series

GET YOUR QUESTIONS ANSWERED: FORUMS



Forum

Colfax Cluster

Discussion of Colfax Cluster usage policies, troubleshooting.

Developer Training, HOW Series

Questions about any of the Colfax trainings? Usage of training servers, experience with specific exercises, inquiries on what's inside, suggestions for future trainings - post them here.

Performance Optimization and Parallelism

Discuss with Colfax Research and colleagues any topics related to computational science, parallel programming, performance optimization and code modernization.

colfaxresearch.com/forum

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

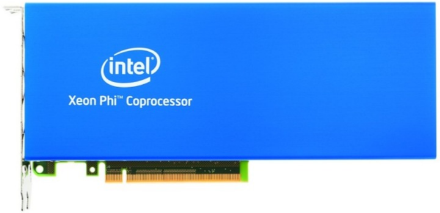
PERFORMANCE OPTIMIZATION

COMPUTING PLATFORMS

Intel Xeon Processor

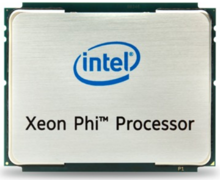
Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*



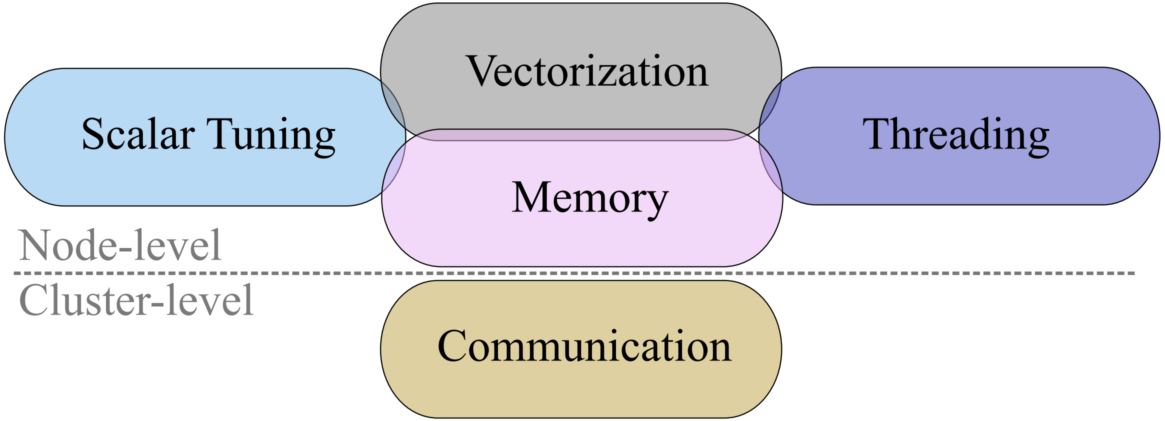
* socket and coprocessor versions

Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

OPTIMIZATION AREAS



"HELLO WORLD" OPENMP PROGRAM

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

int main(){
  // This code is executed by 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

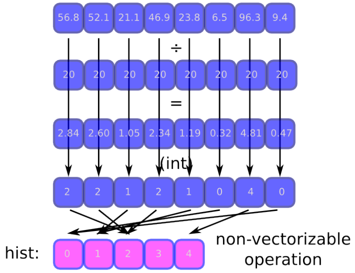
§3. MULTI-THREADING: COMMON ISSUES

TOO MUCH SYNCHRONIZATION

EXAMPLE: BINNING PROBLEM

```
void Histogram(
  // Ages, values from 0.0f to 100.0f:
  const float* age,
  // Size of array age, n=100000000:
  const int n,
  // Output: counts in groups:
  int* const hist,
  // Size of array hist, m=5:
  const int m,
  const float grpWidth) {
    for (int i = 0; i < n; i++) {
      const int j = int(age[i]/grpWidth);
      hist[j]++;
    }
}
```

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



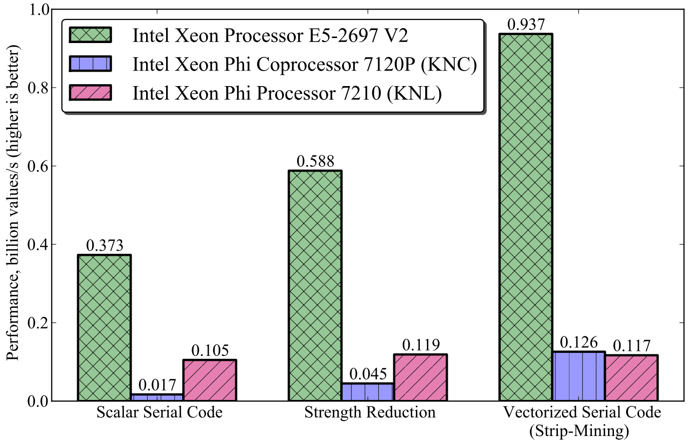
The Same Calculation, Strip-Mined, Vectorized

```
const float recGrpWidth = 1.0f/grpWidth; // precompute the reciprocal

for (int ii = 0; ii < n; ii += 16) { // strip-mining

  int index[16]; // a block of indices
  for (int i = ii; i < ii + 16; i++) // vectorizable
    index[i-ii] = (int) ( age[i] * recGrpWidth ); // unit-stride access

  for (int c = 0; c < 16; c++) // not vectorizable
    hist[index[c]]++; // indirect access
}
```

STRIP-MINING FOR VECTORIZATION

Vectorization improves performance.

More work is needed to take advantage of multiple cores.



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

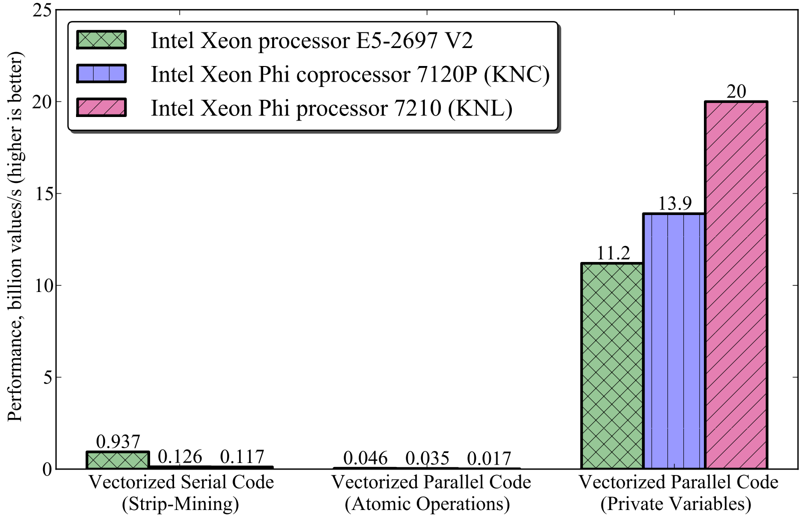
Correct, but inefficient solution:

```
#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



FALSE SHARING

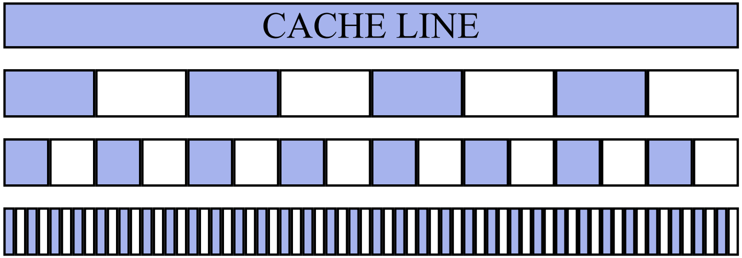
CACHE LINES

- Minimal block of data transferred between memory and cache
- 64 bytes long in Intel Architecture
- Aligned on 64-byte boundaries in memory

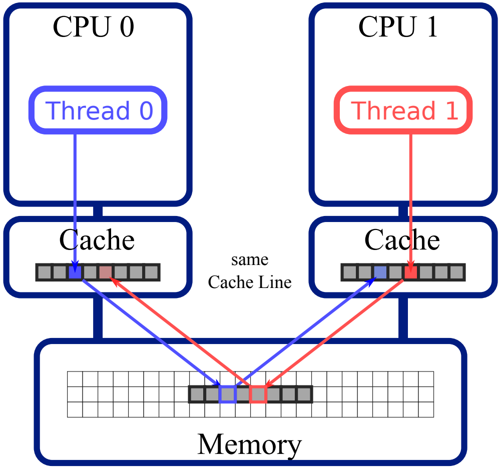
8 double precision values

16 single precision values

64 bytes



FALSE SHARING. DATA PADDING AND PRIVATE VARIABLES



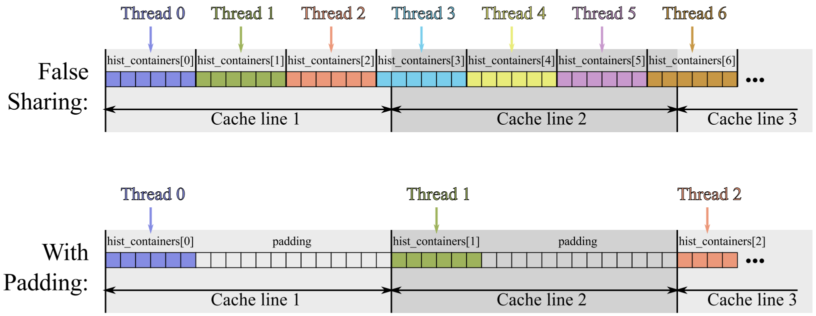
- Occurs when 2 or more threads acess the same cache line, and at least one of the accesses is for writing
- Caused by *coherent caches*
- Cache line is 64-byte wide (in modern Intel architectures)

FALSE SHARING. DATA PADDING AND PRIVATE VARIABLES

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

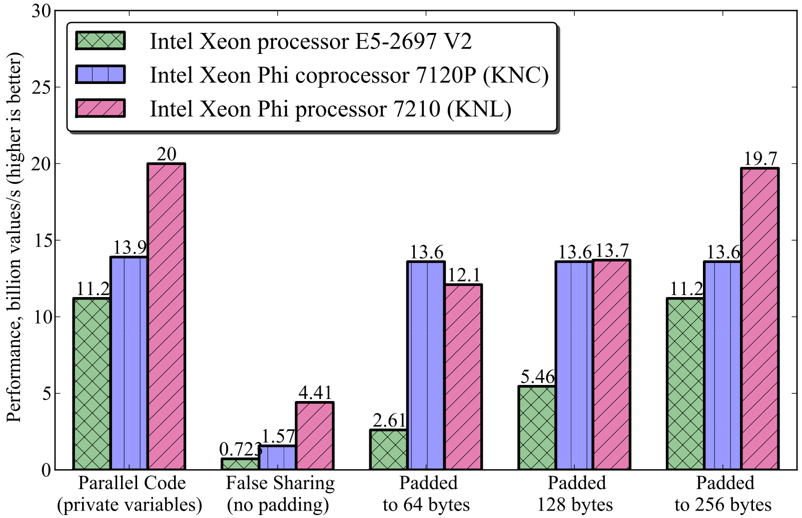
- ▷ The value of `m=5` is small
- ▷ Array elements `hist_thr[0][:]` are within `m*sizeof(int)=20` bytes of array elements `hist_thr[1][:]`

PADDING TO AVOID FALSE SHARING



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

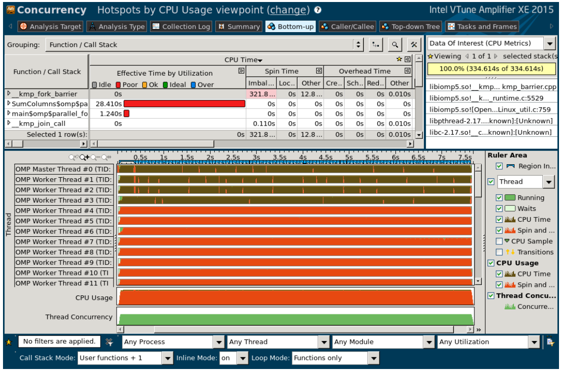
PADDING TO AVOID FALSE SHARING



INSUFFICIENT PARALLELISM

INSUFFICIENT PARALLELISM

Analysis in Intel VTune Amplifier XE


Concurrency Hotspots by CPU Usage viewpoint (change) Intel VTune Amplifier XE 2015

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (🟫 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Spin Time: Imbal...	Spin Time: Loc..	Spin Time: Other	Overhead Time: Cre..	Overhead Time: Sch..	Overhead Time: Red..	Overhead Time: Other
__kmp_fork_barrier	0s	321.8...	0s	12.8...	0s	0s	0s	0.010s
SumColumns$omp$pa	28.410s	0s	0s	0s	0s	0s	0s	0s
main$omp$parallel_fo	1.240s	0s	0s	0s	0s	0s	0s	0s
__kmp_join_call	0s	0.110s	0s	0s	0s	0s	0s	0.010s
Selected 1 row(s):	0s	321.8...	0s	12.8...	0s	0s	0s	0.010s

Data Of Interest (CPU Metrics)

Viewing 1 of 1 selected stack(s)

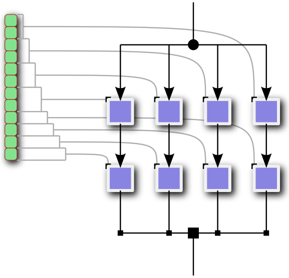
100.0% (334.614s of 334.614s)

libiomp5.so!__kmp... kmp_barrier.cpp
libiomp5.so!__k..._runtime.c:5529
libiomp5.so![Open...Linux_util.c:759
libpthread-2.17....known]:[Unknown]
libc-2.17.so!__c...known]:[Unknown]

0.5s 1s 1.5s 2s 2.5s 3s 3.5s 4s 4.5s 5s 5.5s 6s 6.5s 7s 7.5s
Thread
OMP Master Thread #0 (TID:
OMP Worker Thread #1 (TID:
OMP Worker Thread #2 (TID:
OMP Worker Thread #3 (TID:
OMP Worker Thread #4 (TID:
OMP Worker Thread #5 (TID:
OMP Worker Thread #6 (TID:
OMP Worker Thread #7 (TID:
OMP Worker Thread #8 (TID:
OMP Worker Thread #9 (TID:
OMP Worker Thread #10 (TI
OMP Worker Thread #11 (TI
CPU Usage
Thread Concurrency
Ruler Area
Region In...
Thread
Running
Waits
CPU Time
Spin and ...
CPU Sample
Transitions
CPU Usage
CPU Time
Spin and ...
Thread Concu...
Concurre...

No filters are applied. | Any Process | Any Thread | Any Module | Any Utilization

Call Stack Mode: User functions + 1 | Inline Mode: on | Loop Mode: Functions only



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

EXAMPLE: DEALING WITH INSUFFICIENT PARALLELISM

$$S_i = \sum_{j=0}^{n} M_{ij}, \; i = 0 \ldots m. \qquad (1)$$

- 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

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

DOES NOT WORK: PARALLELIZING INNER LOOP

Inner loop has more iterations, parallelize there?

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

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:

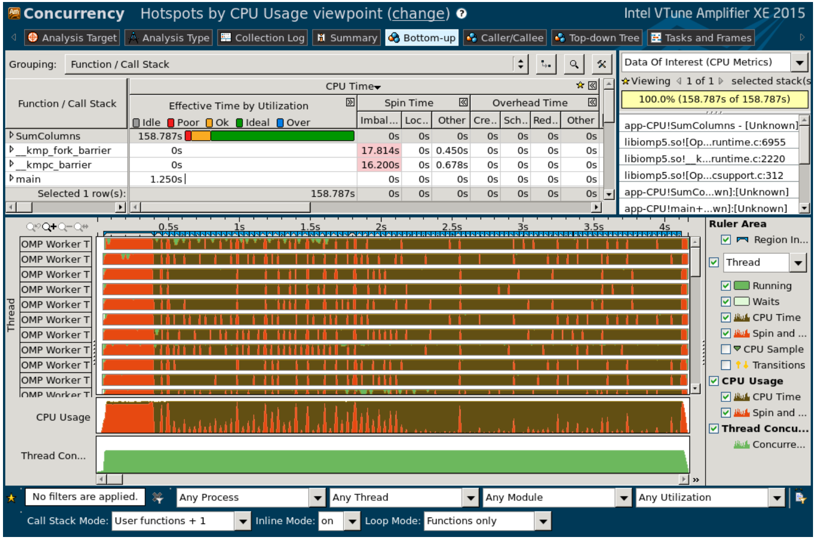
```
void SumCollapse(const int m, const int n, long* M, long* s){
  s[:]=0;
#pragma omp parallel
  { // Each thread will have a private container
    long total[m];  total[:] = 0;
#pragma omp for collapse(2)
    for (int i = 0; i < m; i++) // m=4
      for (int j = 0; j < n; j++) // n=100000000
    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

Does not work: automatic vectorization fails.

EXPOSING PARALLELISM: STRIP-MINING AND LOOP COLLAPSE

```
void sum_stripmine(const int m, const int n, long* M, long* s){
  const int STRIP=1024;
  assert(n%STRIP==0);
  s[0:m]=0;
#pragma omp parallel
  {
    long total[m];   total[0:m]=0;
#pragma omp for collapse(2) schedule(guided)
    for (int i=0; i<m; i++)
      for (int jj=0; jj<n; jj+=STRIP)
#pragma vector aligned
        for (int j=jj; j<jj+STRIP; j++)
          total[i]+=M[i*n+j];
    for (int i=0; i<m; i++)          // Reduction
#pragma omp atomic
      s[i]+=total[i];
}   }
```

EXPOSING PARALLELISM: STRIP-MINING AND LOOP COLLAPSE



Concurrency Hotspots by CPU Usage viewpoint (change) Intel VTune Amplifier XE 2015

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time: Imbal...	Spin Time: Loc..	Spin Time: Other	Overhead Time: Cre..	Overhead Time: Sch..	Overhead Time: Red..	Overhead Time: Other
SumColumns	158.787s	0s	0s	0s	0s	0s	0s	0s
__kmp_fork_barrier	0s	17.814s	0s	0.450s	0s	0s	0s	0s
__kmpc_barrier	0s	16.200s	0s	0.678s	0s	0s	0s	0s
main	1.250s	0s	0s	0s	0s	0s	0s	0s
Selected 1 row(s):	158.787s	0s	0s	0s	0s	0s	0s	0s

Data Of Interest (CPU Metrics)

Viewing 1 of 1 selected stack(s)

100.0% (158.787s of 158.787s)

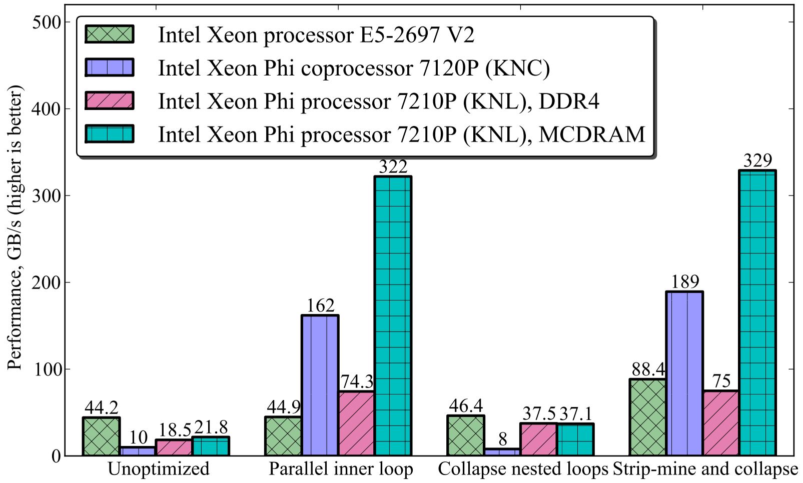
app-CPU!SumColumns - [Unknown]
libiomp5.so![Op...runtime.c:6955
libiomp5.so!__k...runtime.c:2220
libiomp5.so![Op...csupport.c:312
app-CPU!SumCo...wn]:[Unknown]
app-CPU!main+...wn]:[Unknown]

No filters are applied. Any Process | Any Thread | Any Module | Any Utilization

Call Stack Mode: User functions + 1 Inline Mode: on Loop Mode: Functions only

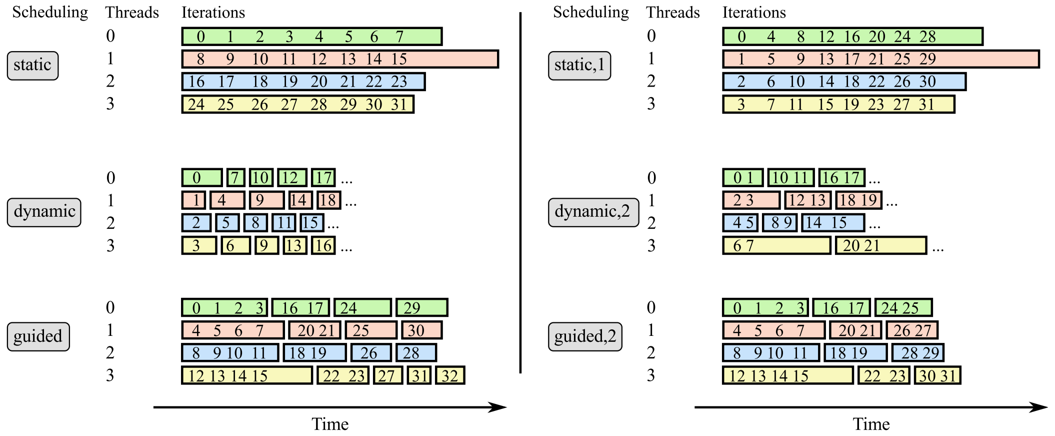
INSUFFICIENT PARALLELISM

DEALING WITH INSUFFICIENT PARALLELISM



LOAD IMBALANCE

LOOP SCHEDULING MODES IN OPENMP



CONTROL OF SCHEDULING MODES

To set scheduling for a particular loop in code (example):

```
#pragma omp parallel for schedule(dynamic,4)
  // ...
```

To set scheduling for the entire application at run time (example):

```
#pragma omp parallel for schedule(runtime)
  // ...
```

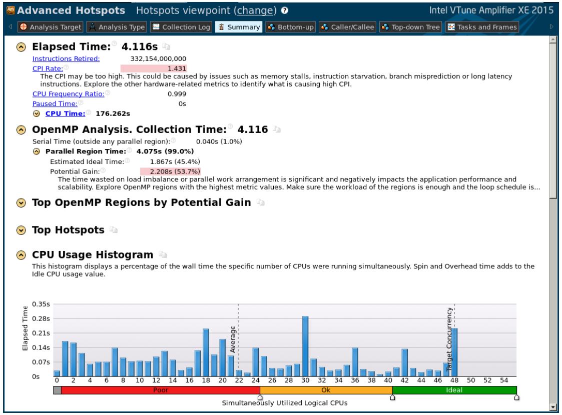
```
vega@lyra% export OMP_SCHEDULE=dynamic,4
vega@lyra% ./run-my-app
```

ITERATIVE JACOBI SOLVER

```
int IterativeSolver(int n, double* M, double* b, double* x, double minAccuracy){
  double accuracy; int iters=0; double bTrial[n] __attribute__((aligned(64)));
  x[0:n] = 0.0; // Initial guess
  do { iters++; // The Jacobi method - iterate until convergence
       for (int i = 0; i < n; i++) {
         double c = 0.0;
#pragma vector aligned
         for (int j = 0; j < n; j++) c += M[i*n+j]*x[j]; // Iterate
         x[i] = x[i] + (b[i] - c)/M[i*n+i]; }
       bTrial[:] = 0.0; // Verification
       for (int i = 0; i < n; i++)
#pragma vector aligned
         for (int j = 0; j < n; j++) bTrial[i] += M[i*n+j]*x[j];
       accuracy = RelativeNormOfDifference(n, b, bTrial); // Check convergence
     } while (accuracy > minAccuracy); // Must achieve the requested accuracy
  return iters; }
```

AN ITERATIVE JACOBI SOLVER

```
#pragma omp parallel for
    for (int c = 0; c < nBVectors; c++)
        IterativeSolver(n, M, &b[c*n], &x[c*n], accuracy[c]);
```



Advanced Hotspots Hotspots viewpoint (change) Intel VTune Amplifier XE 2015

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Elapsed Time: 4.116s

Instructions Retired: 332,154,000,000
CPI Rate: 1.431
The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.
CPU Frequency Ratio: 0.999
Paused Time: 0s
CPU Time: 176.262s

OpenMP Analysis. Collection Time: 4.116

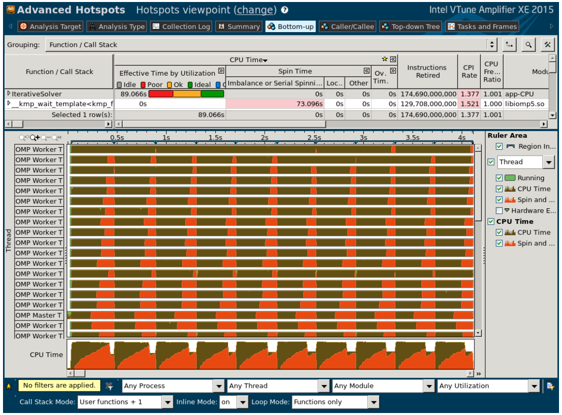
Serial Time (outside any parallel region): 0.040s (1.0%)
Parallel Region Time: 4.075s (99.0%)
Estimated Ideal Time: 1.867s (45.4%)
Potential Gain: 2.208s (53.7%)
The time wasted on load imbalance or parallel work arrangement is significant and negatively impacts the application performance and scalability. Explore OpenMP regions with the highest metric values. Make sure the workload of the regions is enough and the loop schedule is...

Top OpenMP Regions by Potential Gain

Top Hotspots

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



Advanced Hotspots Hotspots viewpoint (change) Intel VTune Amplifier XE 2015

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: Function / Call Stack

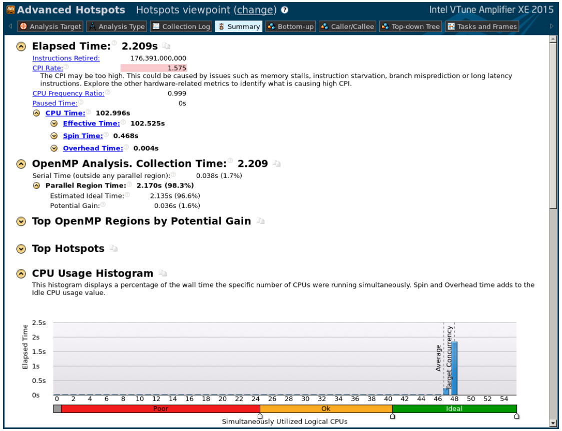
Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal)	CPU Time: Spin Time: Imbalance or Serial Spinni...	Spin Time: Loc..	Spin Time: Other	Ov. Tim.	Instructions Retired	CPI Rate	CPU Fre... Ratio	Modu
IterativeSolver	89.066s	0s	0s	0s	0s	174,690,000,000	1.377	1.001	app-CPU
__kmp_wait_template<kmp_f	0s	73.096s	0s	0s	0s	129,708,000,000	1.521	1.000	libiomp5.so
Selected 1 row(s):	89.066s	0s	0s	0s	0s	174,690,000,000	1.377	1.001	

No filters are applied. Any Process | Any Thread | Any Module | Any Utilization
Call Stack Mode: User functions + 1 | Inline Mode: on | Loop Mode: Functions only

LOAD IMBALANCE

AN ITERATIVE JACOBI SOLVER WITH DYNAMIC SCHEDULING

```
#pragma omp parallel for schedule(dynamic,4)
    for (int c = 0; c < nBVectors; c++)
        IterativeSolver(n, M, &b[c*n], &x[c*n], accuracy[c]);
```



Elapsed Time: 2.209s

Instructions Retired:	176,391,000,000
CPI Rate:	1.575

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

CPU Frequency Ratio:	0.999
Paused Time:	0s
CPU Time:	102.996s
Effective Time:	102.525s
Spin Time:	0.468s
Overhead Time:	0.004s

OpenMP Analysis. Collection Time: 2.209

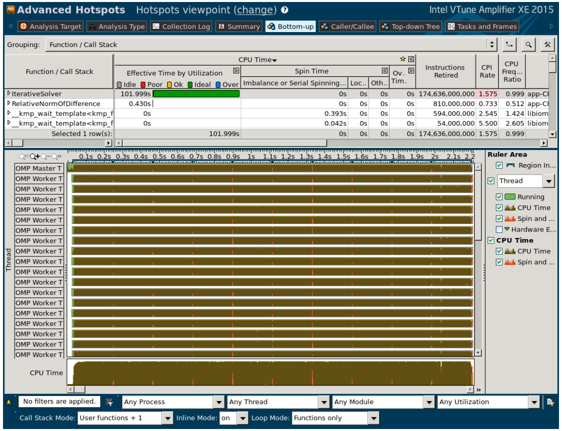
Serial Time (outside any parallel region):	0.038s (1.7%)
Parallel Region Time:	2.170s (98.3%)
Estimated Ideal Time:	2.135s (96.6%)
Potential Gain:	0.036s (1.6%)

Top OpenMP Regions by Potential Gain

Top Hotspots

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



Grouping: Function / Call Stack

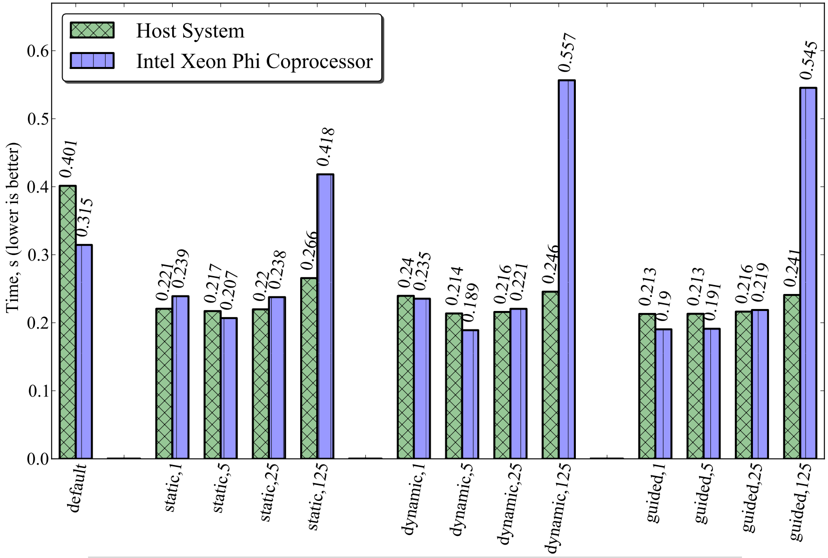
Function / Call Stack	CPU Time: Effective Time by Utilization (🟥 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Spin Time: Imbalance or Serial Spinning...	Loc..	Oth..	Ov. Tim.	Instructions Retired	CPI Rate	CPU Freq... Ratio	
IterativeSolver	101.999s	0s	0s	0s	0s	174,636,000,000	1.575	0.999	app-CI
RelativeNormOfDifference	0.430s	0s	0s	0s	0s	810,000,000	0.733	0.512	app-CI
__kmp_wait_template<kmp_f	0s	0.393s	0s	0s	0s	594,000,000	2.545	1.424	libiom
__kmp_wait_template<kmp_f	0s	0.042s	0s	0s	0s	54,000,000	5.500	2.605	libiom
Selected 1 row(s):	101.999s	0s	0s	0s	0s	174,636,000,000	1.575	0.999	

No filters are applied. Any Process · Any Thread · Any Module · Any Utilization

Call Stack Mode: User functions + 1 · Inline Mode: on · Loop Mode: Functions only

LOAD IMBALANCE

PERFORMANCE OF ITERATIVE JACOBI SOLVER



LOAD IMBALANCE

§4. REVIEW AND WHAT'S NEXT

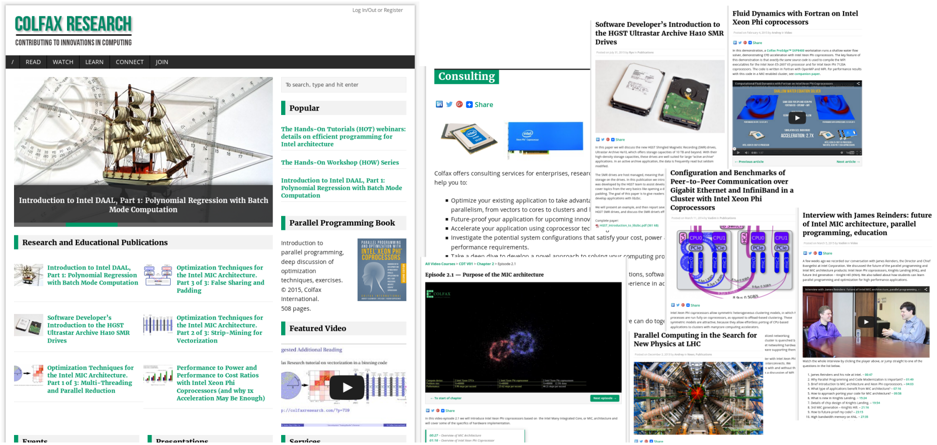
SUMMARY

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
6. Trade-off between load balance and low scheduling overhead

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

COLFAX RESEARCH



Log In/Out or Register

COLFAX RESEARCH

CONTRIBUTING TO INNOVATIONS IN COMPUTING

/ READ WATCH LEARN CONNECT JOIN

Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation

To search, type and hit enter

Popular

The Hands-On Tutorials (HOT) webinars: details on efficient programming for Intel architecture

The Hands-On Workshop (HOW) Series

Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation

Parallel Programming Book

Introduction to parallel programming, deep discussion of optimization techniques, exercises. © 2015, Colfax International. 508 pages.

Research and Educational Publications

Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation

Optimization Techniques for the Intel MIC Architecture. Part 3 of 3: False Sharing and Padding

Software Developer's Introduction to the HGST Ultrastar Archive Ha10 SMR Drives

Optimization Techniques for the Intel MIC Architecture. Part 2 of 3: Strip-Mining for Vectorization

Optimization Techniques for the Intel MIC Architecture. Part 1 of 3: Multi-Threading and Parallel Reduction

Performance to Power and Performance to Cost Ratios with Intel Xeon Phi Coprocessors (and why 1x Acceleration May Be Enough)

Featured Video

Events Presentations Services

Consulting

Share

Colfax offers consulting services for enterprises, researc help you to:

- Optimize your existing application to take advanta parallelism, from vectors to cores to clusters and
- Future-proof your application for upcoming innov
- Accelerate your application using coprocessor tec
- Investigate the potential system configurations that satisfy your cost, power performance requirements.
- Take a deep dive to develop a novel approach to solving your computing pro

tions, softwa erience in ac

e can do toge

Software Developer's Introduction to the HGST Ultrastar Archive Ha10 SMR Drives

Posted on July 31, 2015 by Ryo in Publications

Share

In this paper we will discuss the new HGST Shingled Magnetic Recording (SMR) drives, Ultrastar Archive Ha10, which offers storage capacities of 10 TB and beyond. With their high-density storage capacities, these drives are well suited for large "active archive" applications. In an active archive application, the data is frequently read but seldom modified.

The SMR drives are host managed, meaning that storage on the drives. In this publication we intro was developed by the HGST team to assist develo cover topics from the very basics like opening a d padding. The goal of this paper is to give readers develop applications with libzbc.

We will present an example, and then report seve HGST SMR drives, and discuss the SMR drive's eff

Complete paper: HGST_Introduction_to_libzbc.pdf (361 kB)

Fluid Dynamics with Fortran on Intel Xeon Phi coprocessors

Posted on February 4, 2015 by Andrey in Video

Share

In this demonstration, a Colfax ProEdge™ SXP8400 workstation runs a shallow water flow solver, demonstrating CFD acceleration with Intel Xeon Phi coprocessors. The key feature of this demonstration is that *exactly the same source code* is used to compile the MPI executables for the Intel Xeon E5-2697 V3 processor and for Intel Xeon Phi 7120A coprocessors. The code is written in Fortran with OpenMP and MPI. For performance results with this code in a MIC-enabled cluster, see companion paper.

Computational Fluid Dynamics with Fortran on Intel Xeon Phi Coprocessors

← Previous article Next article →

Configuration and Benchmarks of Peer-to-Peer Communication over Gigabit Ethernet and InfiniBand in a Cluster with Intel Xeon Phi Coprocessors

Posted on March 11, 2014 by Vadim in Publications

Share

Intel Xeon Phi coprocessors allow symmetric heterogeneous clustering models, in which processes are run fully on coprocessors, as opposed to offload-based clustering. These symmetric models are attractive, because they allow effortless porting of CPU-based applications to clusters with manycore computing accelerators.

alized networking cluster is quenched b et networking hardwa ware supporting them ter with Intel Xeon Phi interconnects. We s with and without th s a discussion of MPI

All Video Courses > CDT V01 > Chapter 2 > Episode 2.1

Episode 2.1 — Purpose of the MIC architecture

← To start of chapter Next episode →

Share

In this video episode 2.1 we will introduce Intel Xeon Phi coprocessors based on the Intel Many Integrated Core, or MIC, architecture and will cover some of the specifics of hardware implementation.

00:27 – Overview of MIC Architecture
01:16 – Overview of Intel Xeon Phi Coprocessor

Parallel Computing in the Search for New Physics at LHC

Posted on December 2, 2013 by Andrey in News, Publications

Share

Interview with James Reinders: future of Intel MIC architecture, parallel programming, education

Posted on March 5, 2015 by Vadim in Video

Share

A few weeks ago we recorded our conversation with James Reinders, the Director and Chief Evangelist at Intel Corporation. We discussed the future of the parallel programming and Intel MIC architecture products: Intel Xeon Phi coprocessors, Knights Landing (KNL), and future 3rd generation – Knight Hill (KNH). We also talked about how students can learn parallel programming and optimization for high performance applications.

Interview with James Reinders: future of Intel MIC architecture, parallel programming, ed

Watch the whole interview by clicking the player above, or jump straight to one of the questions in the list below.

1. James Reinders and his role at Intel. – 00:47
2. Why Parallel Programming and Code Modernization is important? – 01:49
3. Brief introduction to MIC architecture and Xeon Phi coprocessors. – 04:03
4. What type of applications benefit from MIC architecture? – 07:16
5. How to approach porting your code for MIC architecture? – 09:58
6. What is new in Knights Landing. – 15:24
7. Details of chip design of Knights Landing. – 19:54
8. 3rd MIC generation – Knights Hill. – 21:16
9. How to future-proof my code? – 23:15
10. High bandwidth memory on KNL. – 27:35

https://colfaxresearch.com/