



Programming and Optimization for Intel[®] Architecture

The Hands-On Workshop (HOW) Series

Colfax International — @colfaxintl

April 2016 , Rev. 02b

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

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!

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.

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

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/training
- Video courses: colfaxresearch.com/video-courses
- [Intel Many Integrated Core Architecture Forum](#)

HOW Series “Deep Dive” in May

Wish you had joined us earlier? Want to recommend the HOW series to a friend?
Another HOW Series run coming up in May.



THE "HOW" SERIES

DEEP DIVE


WITH CODE MODERNIZATION EXPERTS

STARTS MAY 23

*10x 2-hour sessions | 24-hour 2-weeks remote access to a system | Filling up fast, register now!

colfaxresearch.com/how-series/

HOW Series “Tools”



GOT THE TOOLS - NOW WHAT?

Learn workflows and methodology with the “HOW” tools* training and hands-on demos

* Intel MKL | Intel Advisor | Intel VTune Amplifier

MAY 16, 18, 20

Register now

<http://colfaxresearch.com/how-tools/>

Developer's Guide to Knights Landing



colfaxresearch.com/knl-webinar/

§2. Refresh

Performance Optimization

Computing Platforms

Intel Xeon Processor



Current: Broadwell
Upcoming: Skylake

Multi-Core Architecture

Intel Xeon Phi Coprocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*

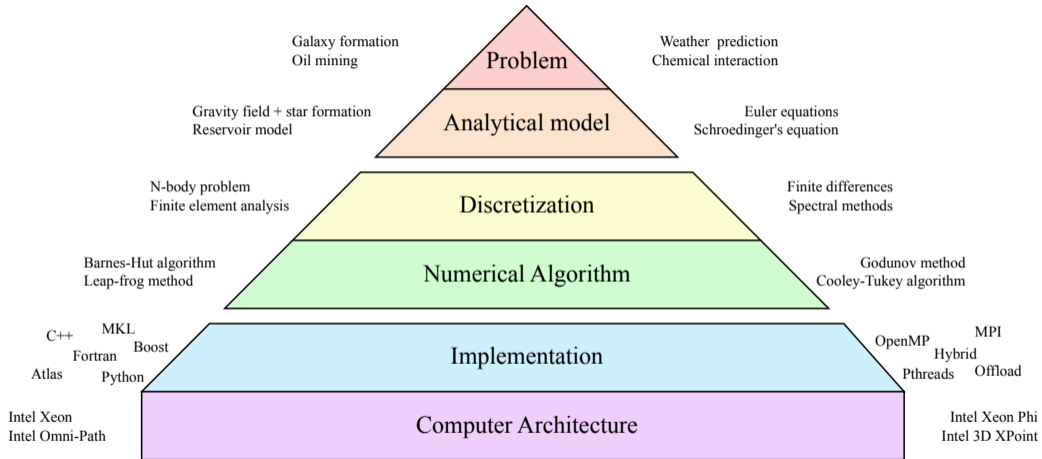


* socket and coprocessor versions

Upcoming: Knights Landing (KNL)

Intel Many Integrated Core (MIC) Architecture

Computing in Science and Engineering

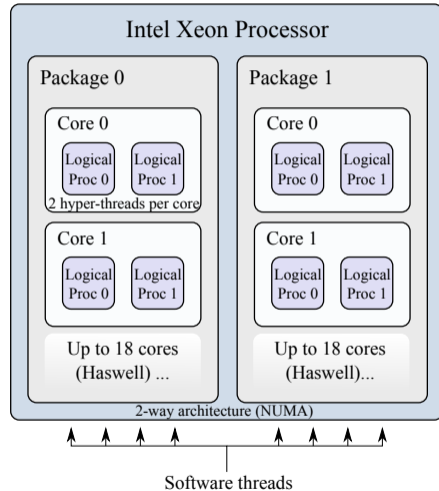
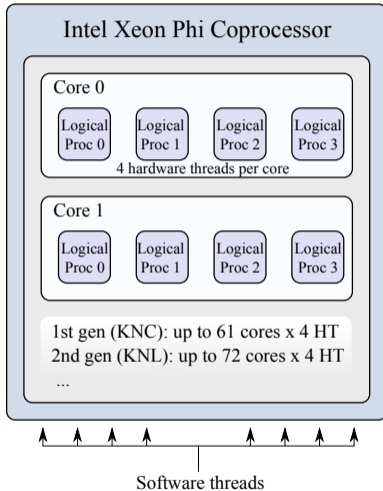


Optimization Areas

- 1 **Scalar optimization** (compiler-friendly practices)
- 2 **Vectorization** (must use 16- or 8-wide vectors)
- 3 **Multi-threading** (must scale to 100+ threads)
- 4 **Memory access** (streaming access or tiling)
- 5 **Communication** (offload, MPI traffic control)

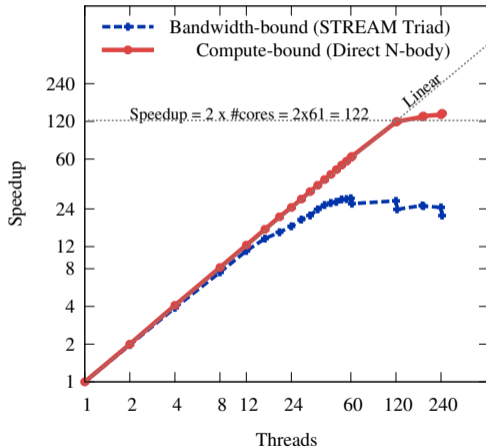
Cores, Threads and OpenMP

Processor Hierarchy

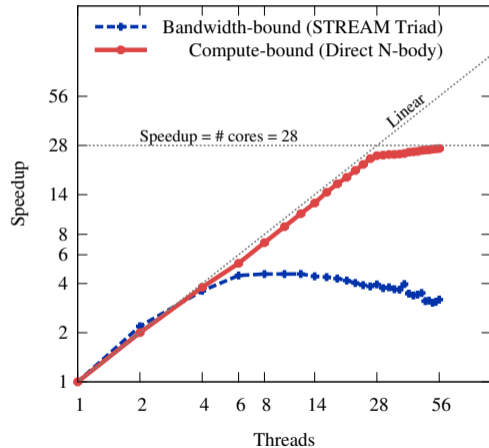


Scalability Expectations: MIC versus CPU

Performance on the MIC architecture



Performance on the CPU architecture



“Hello World” OpenMP Programs

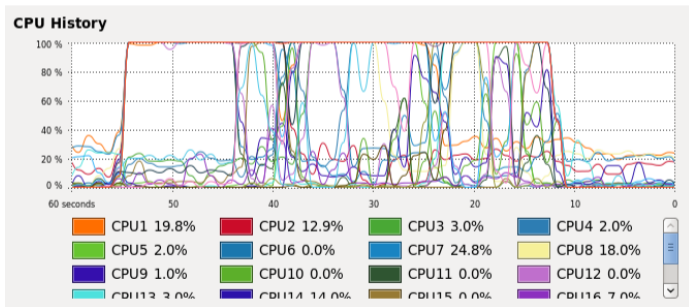
```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(){
5      const int nt=omp_get_max_threads();
6      printf("OpenMP with %d threads\n", nt);
7
8      #pragma omp parallel
9      {
10         printf("Hello World from thread %d\n", omp_get_thread_num());
11     }
12 }
```

§3. Optimization of Multi-Threading II

Thread Affinity

What is Thread Affinity

- OpenMP threads may migrate between cores according to OS decisions.
- Forbid migration — improve locality — increase the performance.



The KMP_AFFINITY Environment Variable

```
KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]
```

modifier:

- verbose/nonverbose
- respect/norespect
- warnings/nowarnings
- granularity=core or thread
- type=compact, scatter or balanced
- type=explicit, proclist=[<proc_list>]
- type=disabled or none.

The most important argument is type:

- compact: place threads as *close to each* other as possible
- scatter: place threads as *far from each* other as possible

Thread Affinity: Scatter Pattern

Generally beneficial for bandwidth-bound applications.

`KMP_AFFINITY=scatter,granularity=fine`

Threads:

0

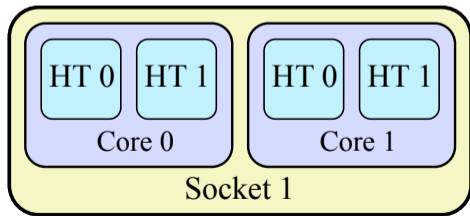
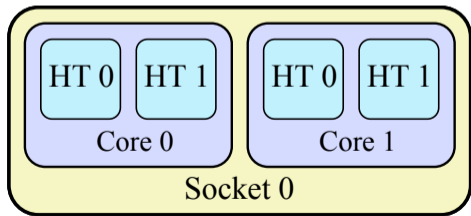
2

1

3



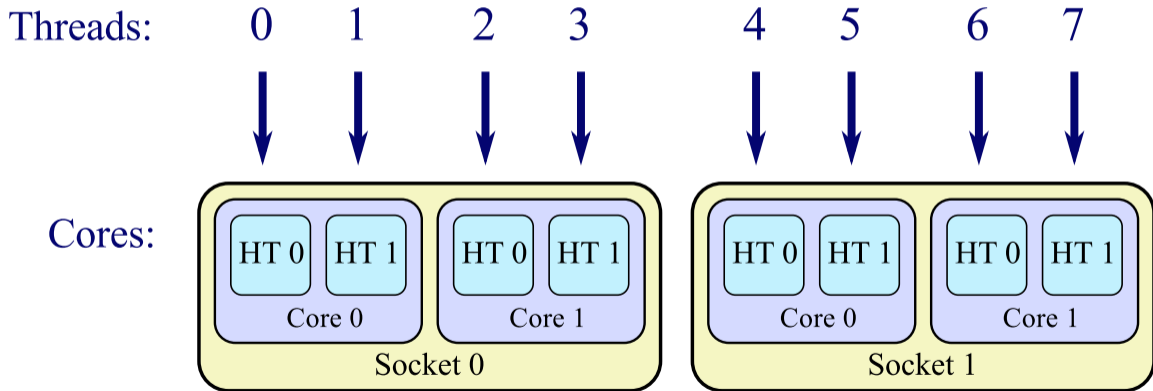
Cores:



Thread Affinity: Compact Pattern

Generally beneficial for compute-bound applications.

`KMP_AFFINITY=compact,granularity=fine`



Thread Affinity: Compact Pattern with Permutation

Same effect as “compact” without hyper-threading.

`KMP_AFFINITY=compact,granularity=fine,1`

Threads:

0

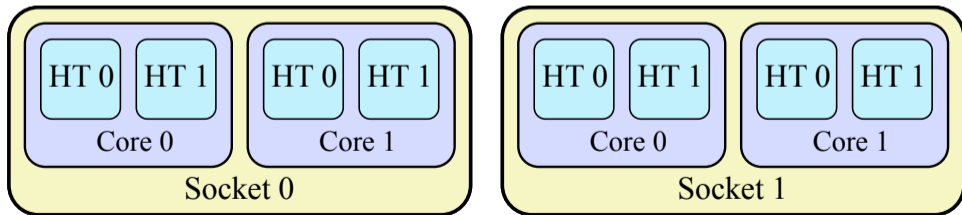
1

2

3



Cores:



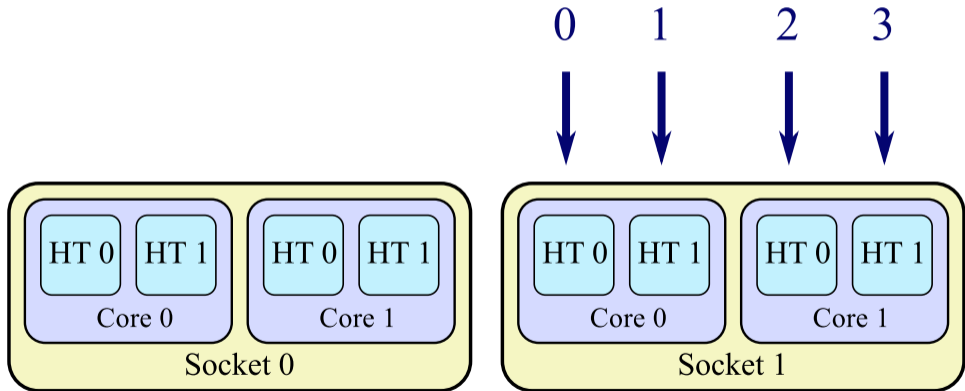
Thread Affinity: Compact Pattern with an Offset

Useful for partitioning a system between multiple processes.

`KMP_AFFINITY=compact,granularity=fine,0,4`

Threads:

Cores:



Bandwidth-bound, KMP_AFFINITY=scatter

```
vega@lyra% export OMP_NUM_THREADS=32
vega@lyra% export KMP_AFFINITY=none
vega@lyra% for i in {1..4} ; do ./rowsum_stripmine | tail -1; done
Problem size: 2.980 GB, outer dimension: 4, threads: 32
Strip-mine and collapse: 0.061 +/- 0.002 seconds (52.89 +/- 1.31 GB/s)
Strip-mine and collapse: 0.059 +/- 0.002 seconds (54.11 +/- 1.56 GB/s)
Strip-mine and collapse: 0.077 +/- 0.001 seconds (41.71 +/- 0.69 GB/s)
Strip-mine and collapse: 0.070 +/- 0.005 seconds (45.59 +/- 3.14 GB/s)
vega@lyra% export OMP_NUM_THREADS=16
vega@lyra% export KMP_AFFINITY=scatter
vega@lyra% for i in {1..4}; do ./rowsum_stripmine | tail -1 ; done
Problem size: 2.980 GB, outer dimension: 4, threads: 16
Strip-mine and collapse: 0.059 +/- 0.004 seconds (54.47 +/- 3.25 GB/s)
Strip-mine and collapse: 0.061 +/- 0.004 seconds (52.30 +/- 3.30 GB/s)
Strip-mine and collapse: 0.062 +/- 0.005 seconds (51.37 +/- 4.29 GB/s)
Strip-mine and collapse: 0.058 +/- 0.001 seconds (55.48 +/- 1.27 GB/s)
```

Compute-Bound, KMP_AFFINITY=compact/balanced

```
1 double* A = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);
2 double* B = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);
3 double* C = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);
4
5 for(int k = 0; k < nIter; k++) {
6
7     dgemm(&tr, &tr, &N, &N, &N, &v, A, &Nld, B, &Nld, &v, C, &N);
8
9     double flopsNow = (2.0*N*N*N+1.0*N*N)*1e-9/(t2-t1);
10    printf("Iteration %d: %.1f GFLOP/s\n", k+1, flopsNow);
11 }
12 _mm_free(A); _mm_free(B); _mm_free(C);
```

Compute-Bound, KMP_AFFINITY=compact/balanced

```
vega@lyra% icpc -o bench-dgemm -mkl -mmic bench-dgemm.cc
```

```
vega@lyra% micnativeloadex ./bench-dgemm
```

```
Iteration 1: 312.7 GFLOP/s
```

```
Iteration 2: 346.5 GFLOP/s
```

```
Iteration 3: 348.5 GFLOP/s
```

```
Iteration 4: 347.2 GFLOP/s
```

```
Iteration 5: 348.3 GFLOP/s
```

```
vega@lyra% micnativeloadex ./bench-dgemm -e "KMP_AFFINITY=compact"
```

```
Iteration 1: 626.8 GFLOP/s
```

```
Iteration 2: 769.1 GFLOP/s
```

```
Iteration 3: 769.4 GFLOP/s
```

```
Iteration 4: 769.3 GFLOP/s
```

```
Iteration 5: 769.4 GFLOP/s
```

The KMP_PLACE_THREADS Environment Variable

Only for Xeon Phi, control the # of cores and # of threads per core:

```
KMP_PLACE_THREADS=[<cores>c,]<threads-per-core>t
```

Complements KMP_AFFINITY:

```
vega@lyra-mic0% export KMP_PLACE_THREADS=61c,3t # 3 threads per core
vega@lyra-mic0% export KMP_AFFINITY=balanced
vega@lyra-mic0% ./my-native-app
```

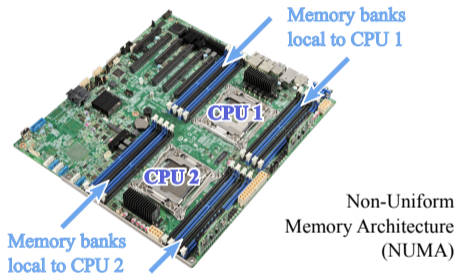
or

```
vega@lyra% export MIC_ENV_PREFIX=XEONPHI
vega@lyra% export XEONPHI_KMP_PLACE_THREADS=60c,3t # 3 threads per core
vega@lyra% export XEONPHI_KMP_AFFINITY=balanced
vega@lyra% ./my-offload-app
```

NUMA Locality

NUMA Architectures

NUMA = Non-Uniform Memory Access. Cores have fast access to local memory, slow access to remote memory.



Examples:

- Multi-socket Intel Xeon processors
- Second generation Intel Xeon Phi

Allocation on First Touch

- Memory allocation occurs not during `_mm_malloc()`, but upon the first write to the buffer (“first touch”)
- Default NUMA allocation policy is “on first touch”
- For better performance in NUMA systems, initialize data with the same parallel pattern as during data usage

```
1 float* A = (float*)_mm_malloc(n*m*sizeof(float), 64);
2
3 // Initializing from parallel region for better performance
4 #pragma omp parallel for
5 for (int i = 0; i < n; i++)
6     for (int j = 0; j < m; j++)
7         A[i*m + j] = 0.0f;
```

Binding to NUMA Nodes with numactl

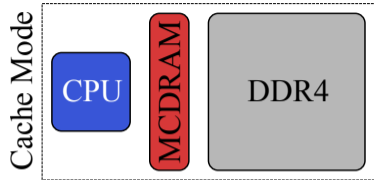
- numactl – a Linux tool for controlling NUMA policy for processes

```
vega@lyra% numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 12 13 14 15 16 17
node 0 size: 65457 MB
node 0 free: 24426 MB
node 1 cpus: 6 7 8 9 10 11 18 19 20 21 22 23
node 1 size: 65536 MB
node 1 free: 53725 MB
node distances:
node  0  1
  0:  10  21
  1:  21  10
vega@lyra% numactl --membind=<nodes> --cpunodebind=<nodes> ./myApplication
```

Using High-Bandwidth Memory (MCDRAM) in KNL

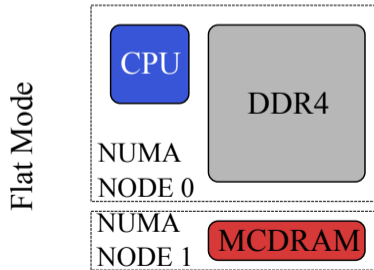
Option 1 : cache/hybrid mode

- Treat it as LLC
- Data locality techniques
- Miss latency 2x the direct DDR4 access



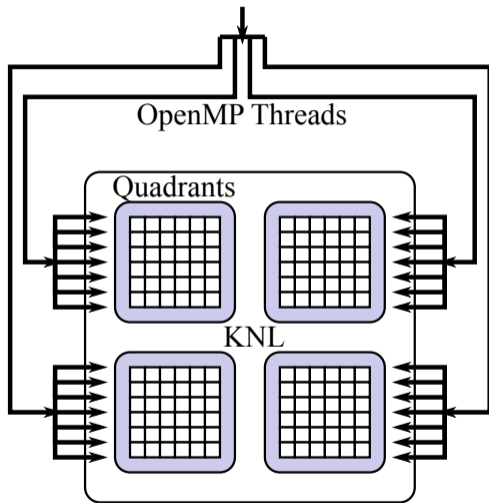
Option 2 : flat mode

- Application fits in 16 GiB? `numactl`
- More than 16 GiB data? Use special allocators (e.g., `memkind`)



Nested Parallelism

Nested Parallelism with OpenMP



```
1  #pragma omp parallel
2  {
3  #pragma omp parallel
4    {
5      // ...
6    }
7  }
```

OpenMP Hot Teams

Xeon

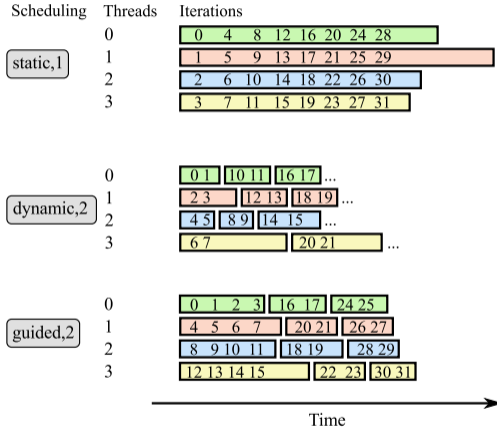
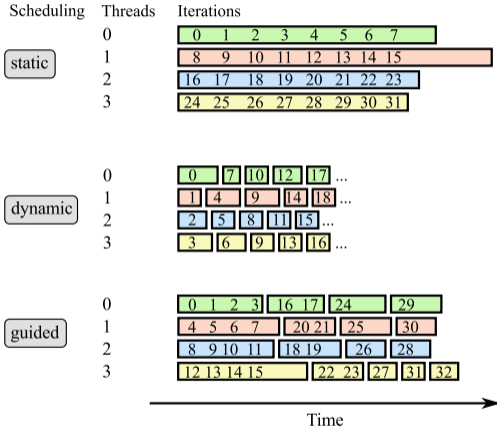
- `OMP_NUM_THREADS=2,14`
- `OMP_NESTED=1`
`OMP_PROC_BIND=spread,close`
`OMP_PLACES=cores`
- `KMP_HOT_TEAMS_MODE=1`
`KMP_HOT_TEAMS_MAX_LEVEL=2`
`OMP_MAX_ACTIVE_LEVELS=2`

Xeon Phi

- `OMP_NUM_THREADS=60,4`
 - `OMP_NESTED=1`
`OMP_PROC_BIND=spread,close`
`OMP_PLACES=threads`
 - `KMP_HOT_TEAMS_MODE=1`
`KMP_HOT_TEAMS_MAX_LEVEL=2`
`OMP_MAX_ACTIVE_LEVELS=2`
-
-

Loop Shedding

Loop Scheduling Modes in OpenMP



Control of Scheduling Modes

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

```
1 #pragma omp parallel for schedule(dynamic,4)  
2 // ...
```

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

```
1 #pragma omp parallel for schedule(runtime)  
2 // ...
```

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

Iterative Jacobi Solver

```

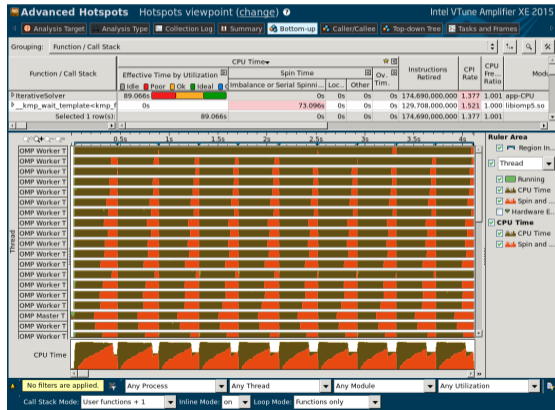
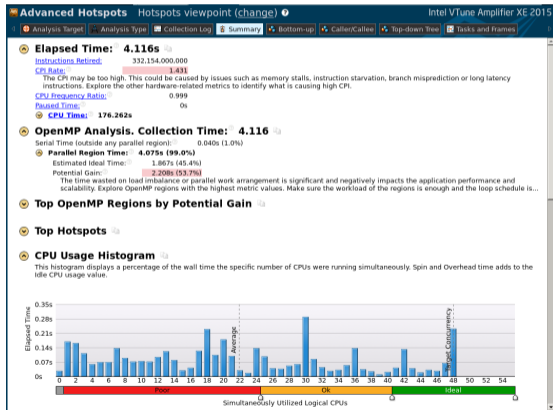
1 int IterativeSolver(int n, double* M, double* b, double* x, double minAccuracy){
2     double accuracy; int iters=0; double bTrial[n] __attribute__((aligned(64)));
3     x[0:n] = 0.0; // Initial guess
4     do { iters++; // The Jacobi method - iterate until convergence
5         for (int i = 0; i < n; i++) {
6             double c = 0.0;
7             #pragma vector aligned
8                 for (int j = 0; j < n; j++) c += M[i*n+j]*x[j]; // Iterate
9                 x[i] = x[i] + (b[i] - c)/M[i*n+i]; }
10            bTrial[:] = 0.0; // Verification
11            for (int i = 0; i < n; i++)
12                #pragma vector aligned
13                    for (int j = 0; j < n; j++) bTrial[i] += M[i*n+j]*x[j];
14            accuracy = RelativeNormOfDifference(n, b, bTrial); // Check convergence
15        } while (accuracy > minAccuracy); // Must achieve the requested accuracy
16    return iters; }

```

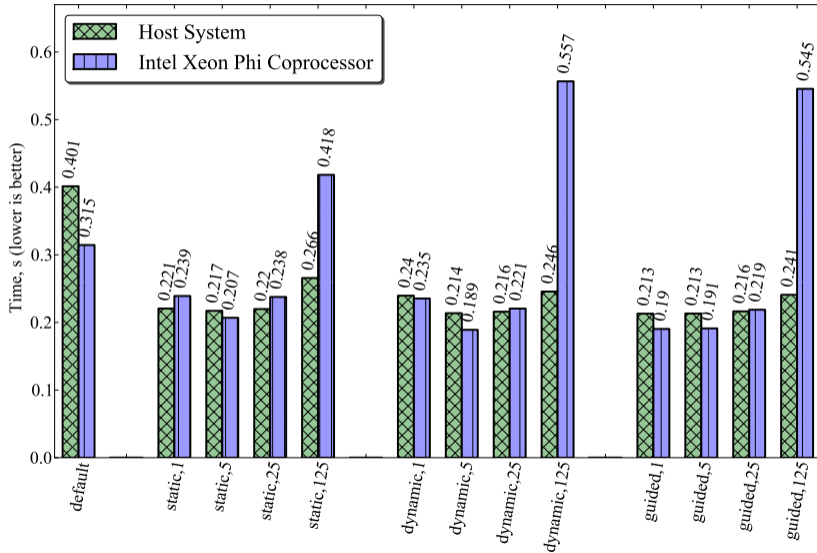
An Iterative Jacobi Solver

```

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



Performance of Iterative Jacobi Solver



§4. Review and What's Next

Summary

This session:

- 1 Setting affinity prevents thread migration
- 2 Affinity pattern “scatter” for bandwidth-bound, “compact” for compute-bound
- 3 Scheduling control allows to find the optimal tradeoff between load balancing and scheduling overhead

Next session: optimization of memory traffic.