Programming and Optimization for Intel® Architecture

The Hands-On Workshop (HOW) Series

Andrey Vladimirov, PhD, and Ryo Asai
Colfax International — @colfaxintl

March 2016 , Rev. 02a

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, 2013-2015

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 – March 7
   - Programming model continuity – March 7
2. Programming models for Xeon Phi coprocessors
   - Native programming – March 7
   - Offload programming – March 8
3. Expressing Parallelism
   - Introduction to vectorization – March 9
   - Crash-course on OpenMP – March 10
4. Optimization – intro on March 11
   - Vectorization tuning – March 14
   - Multi-threading – March 15, 16
   - Memory traffic – March 17
5. ~~Tools: MKL and VTune~~ MPI – March 18

March 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		

■ — Lecture+remote access (March 7–11, 14–18)

■ — Self-study/remote access (March 21–25)

 © Colfax International, 2013–2016

HOW Online

Course page: colfaxresearch.com/how-16-03

- 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

HOT Series

THE "HOT" (HANDS ON TUTORIAL) SERIES

FREE ONLINE WEBINAR

EFFICIENT PROGRAMMING FOR INTEL® ARCHITECTURE

MARCH 21, 23 & 25

3 webinar series | Filling up fast, register now!

colfaxresearch.com/hot-16-03/

 © Colfax International, 2013–2016

HOW Series, April 2016

colfaxresearch.com/how-16-04/

§2. Refresh

 © Colfax International, 2013–2016

Performance Optimization

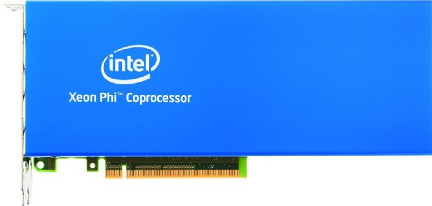
 © Colfax International, 2013–2016

Computing Platforms

Intel Xeon
Processor

Current: Haswell
Upcoming: Broadwell

Intel Xeon Phi
Corocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi
Processor, 2nd generation*

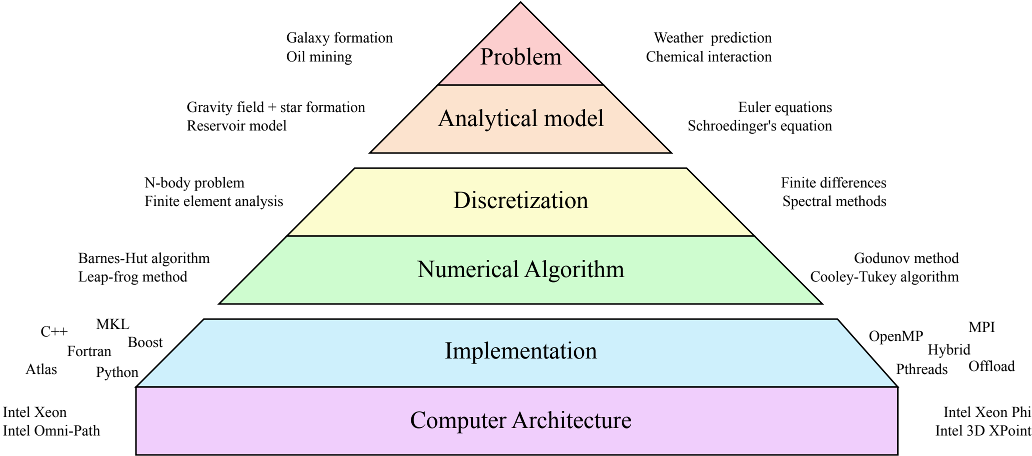
* socket and coprocessor versions

Upcoming: Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

Computing in Science and Engineering



 © Colfax International, 2013–2016

Optimization Areas

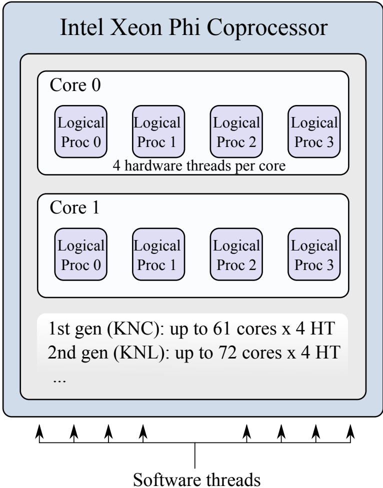
1. **Scalar optimization** (compiler-friendly practices)
2. **Vectorization** (must use 16- or 8-wide vectors)
3. **Multi-threading** (must scale to 100+ threads)
4. **Memory access** (streaming access or tiling)
5. **Communication** (offload, MPI traffic control)

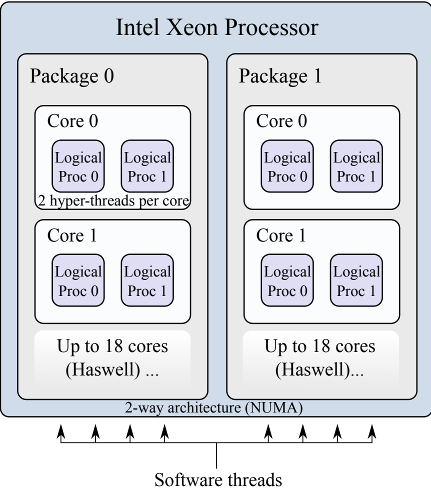
© Colfax International, 2013–2016

Cores, Threads and OpenMP

© Colfax International, 2013–2016

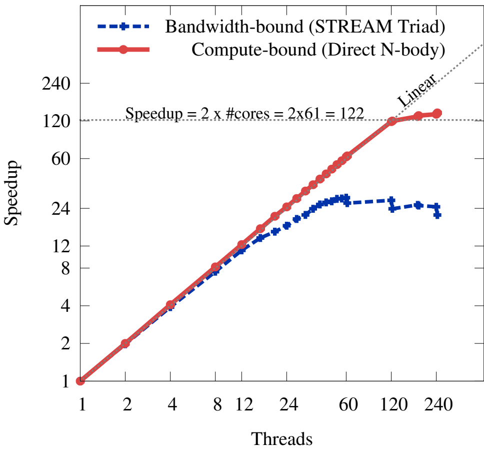
Processor Hierarchy

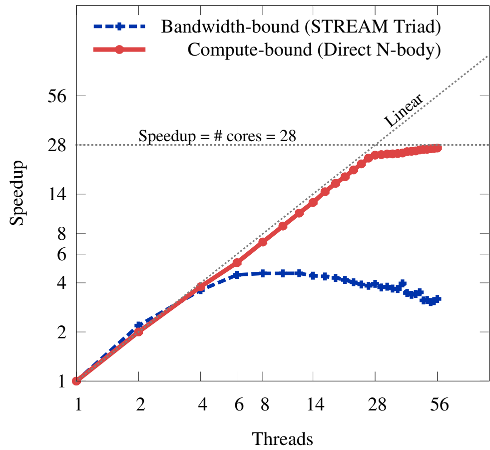




 © Colfax International, 2013–2016

Scalability Expectations: MIC versus CPU





“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());
  }
}
```

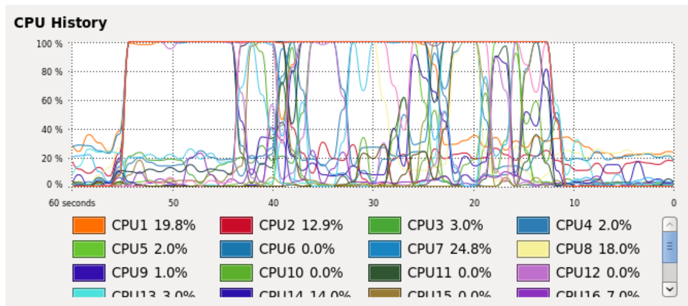
§3. Optimization of Multi-Threading II

© Colfax International, 2013–2016

Thread Affinity

What is Thread Affinity

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



 © Colfax International, 2013–2016

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

© Colfax International, 2013–2016

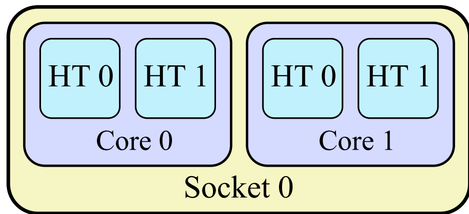
Thread Affinity: Scatter Pattern

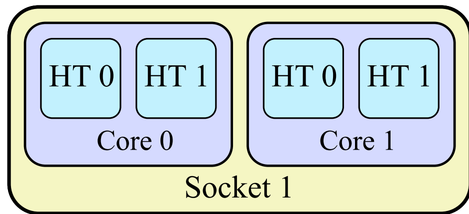
Generally beneficial for bandwidth-bound applications.

`KMP_AFFINITY=scatter,granularity=fine`

Threads: 0 2 1 3

Cores:





© Colfax International, 2013–2016

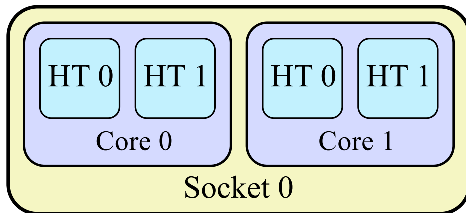
Thread Affinity: Compact Pattern

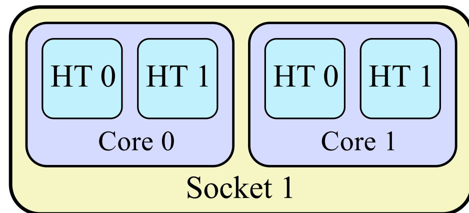
Generally beneficial for compute-bound applications.

`KMP_AFFINITY=compact,granularity=fine`

Threads: 0 1 2 3 4 5 6 7

Cores:





 © Colfax International, 2013–2016

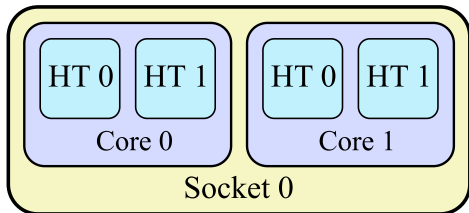
Thread Affinity: Compact Pattern with Permutation

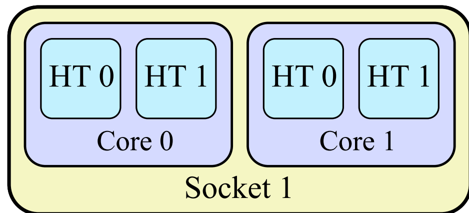
Same effect as "compact" without hyper-threading.

```
KMP_AFFINITY=compact,granularity=fine,1
```

Threads: 0 1 2 3

Cores:





 © Colfax International, 2013–2016

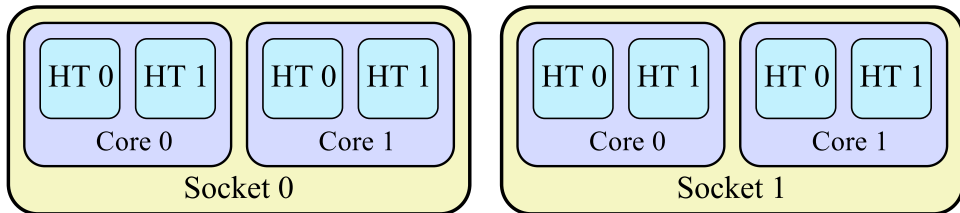
Thread Affinity: Compact Pattern with an Offset

Useful for partitioning a system between multiple processes.

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

Threads:

Cores:



 © Colfax International, 2013–2016

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)
```

 © Colfax International, 2013–2016

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
```

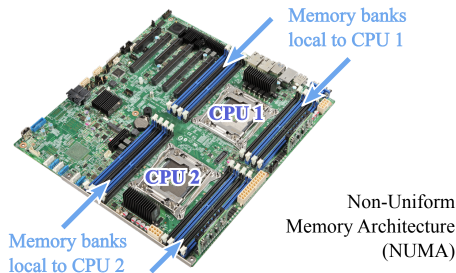

© Colfax International, 2013–2016

NUMA Locality

 © Colfax International, 2013–2016

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

 © Colfax International, 2013–2016

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

```
float* A = (float*)_mm_malloc(n*m*sizeof(float), 64);

// Initializing from parallel region for better performance
#pragma omp parallel for
for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    A[i*m + j] = 0.0f;
```

 © Colfax International, 2013–2016

Binding to NUMA Nodes with numactl

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

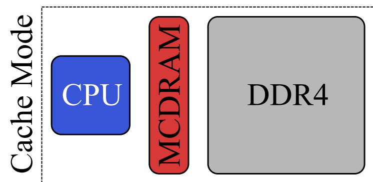
```
vega@lyra% numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 12 13 14 15 16 17
node 0 size: 65457 MB
node 0 free: 24426 MB
node 1 cpus: 6 7 8 9 10 11 18 19 20 21 22 23
node 1 size: 65536 MB
node 1 free: 53725 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
vega@lyra%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

 © Colfax International, 2013–2016

Using High-Bandwidth Memory (MCDRAM) in KNL

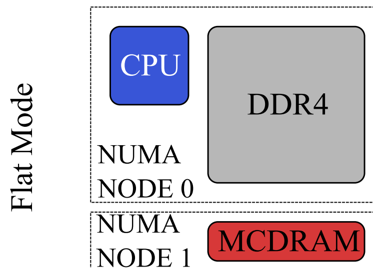
Option 1 : cache/hybrid mode

- 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`)

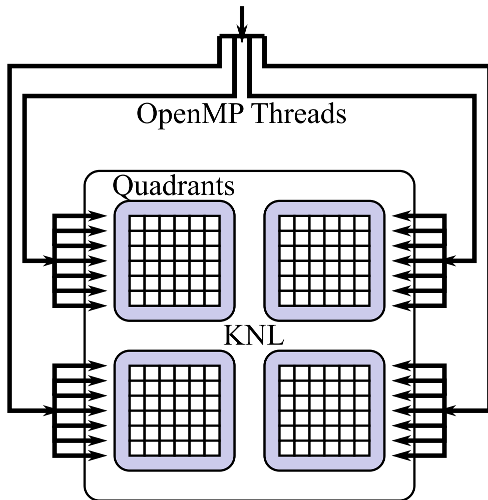


 © Colfax International, 2013–2016

Nested Parallelism

© Colfax International, 2013–2016

Nested Parallelism with OpenMP




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

OpenMP Hot Teams

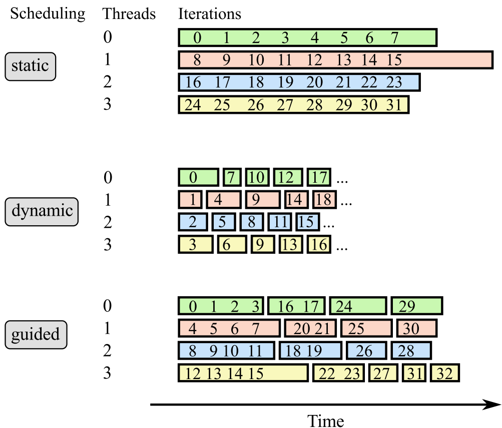
Xeon	Xeon Phi
• OMP_NUM_THREADS=2,14	• OMP_NUM_THREADS=60,4
• OMP_NESTED=1 OMP_PROC_BIND=spread,close OMP_PLACES=cores	• OMP_NESTED=1 OMP_PROC_BIND=spread,close OMP_PLACES=threads
• KMP_HOT_TEAMS_MODE=1 KMP_HOT_TEAMS_MAX_LEVEL=2 OMP_MAX_ACTIVE_LEVELS=2	• KMP_HOT_TEAMS_MODE=1 KMP_HOT_TEAMS_MAX_LEVEL=2 OMP_MAX_ACTIVE_LEVELS=2

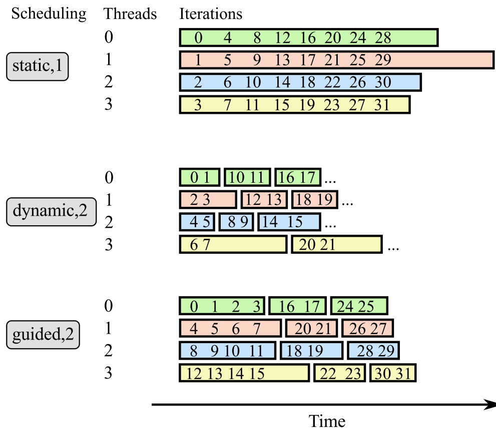
© Colfax International, 2013–2016

Loop Sheduling

© Colfax International, 2013–2016

Loop Scheduling Modes in OpenMP





 | © Colfax International, 2013–2016

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
```

 © Colfax International, 2013–2016

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; }
```

 © Colfax International, 2013–2016

An Iterative Jacobi Solver

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

An Iterative Jacobi Solver with Dynamic Scheduling

```
1  #pragma omp parallel for schedule(dynamic,4)
2      for (int c = 0; c < nBVectors; c++)
3          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: 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.

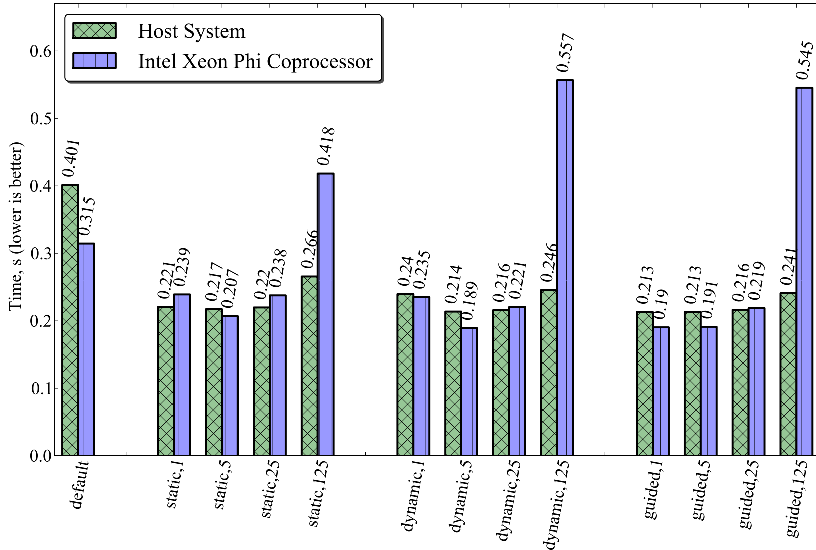
Advanced Hotspots Hotspots viewpoint (change) Intel VTune Amplifier XE 2015

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, 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	

Performance of Iterative Jacobi Solver



 © Colfax International, 2013–2016

§4. Review and What's Next

 © Colfax International, 2013–2016

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.

© Colfax International, 2013–2016