DAY 5: INTRODUCTION TO PARALLEL INTEL® ARCHITECTURES

Lecture day 5

Ryo Asai
Colfax International — *colfaxresearch.com*

April 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.

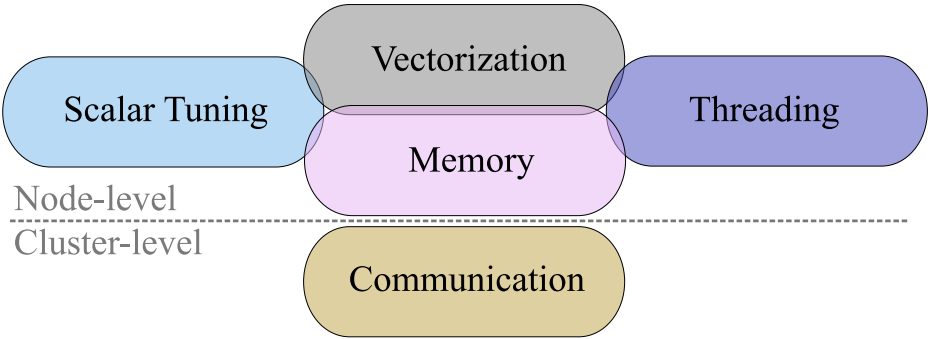
RECAP: OPENMP

Directives discussed:

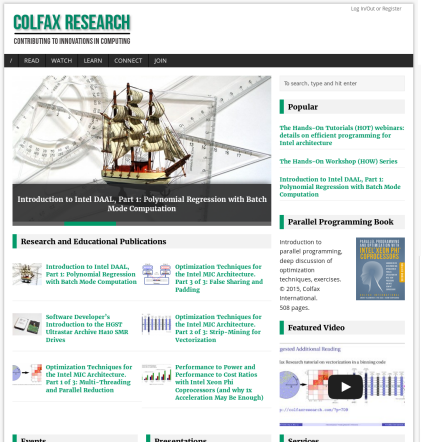
- `omp parallel`: create team of threads
- parallel `for` loop and `sections`
- `omp task`: variables are `firstprivate` by default
- `taskwait`: used for synchronizarion between threads
- `reduction < atomic < critical`
- `ordered`: execute loop in parallel; ordered block is executed sequentially following the natural loop ordering
- `taskloop`: similar to `omp for` but uses the more flexible `task` mechanism instead of worksharing `omp for`

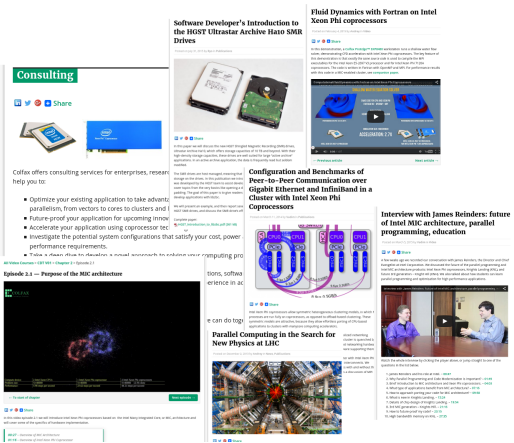
Code Modernization

Optimizing software to better utilize features available in modern computer architectures.



COLFAX RESEARCH

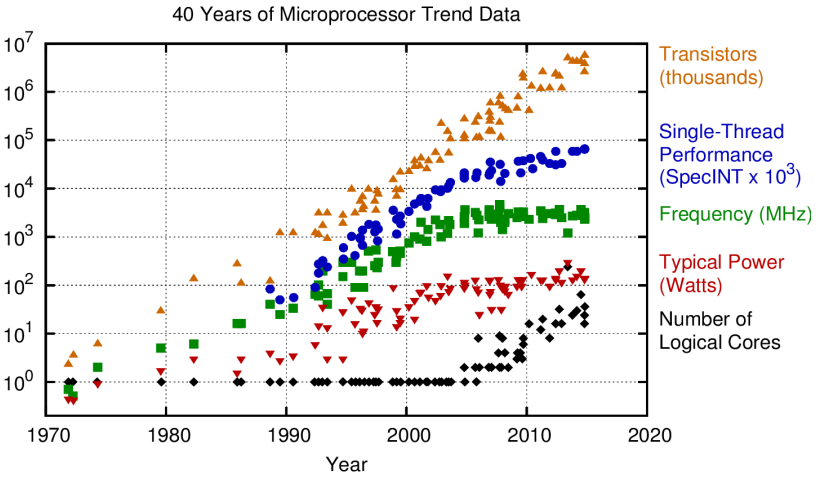




http://colfaxresearch.com/

§1. INTRODUCTION

40-YEAR MICROPROCESSOR TREND

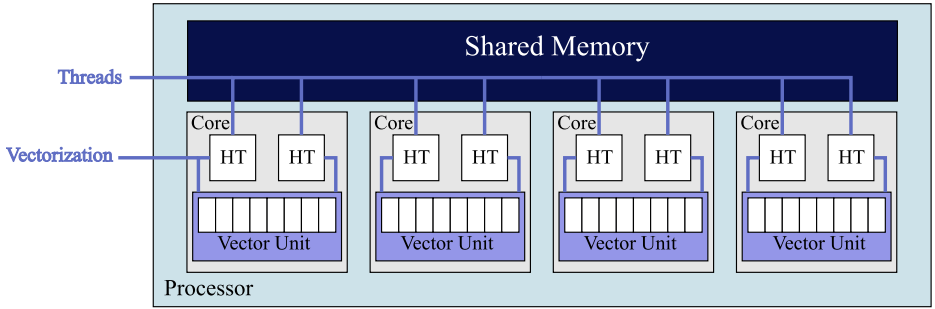


Source: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

PARALLELISM

Task Parallelism – multiple instructions multiple data elements (MIMD)

Data Parallelism – single instruction multiple data elements (SIMD)

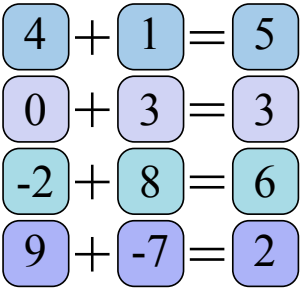


Unbounded growth opportunity, but **not automatic**

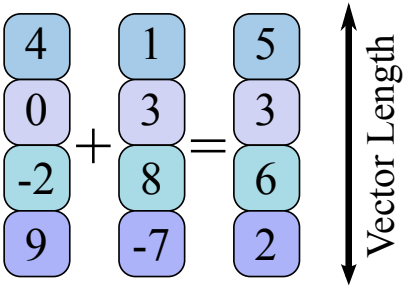
SHORT VECTOR SUPPORT

Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.

Scalar Instructions



Vector Instructions



INTEL ARCHITECTURES

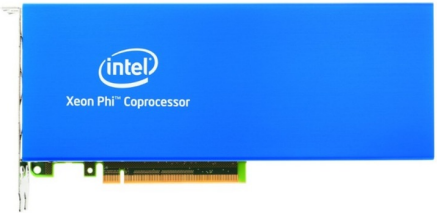
INTEL ARCHITECTURE

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*

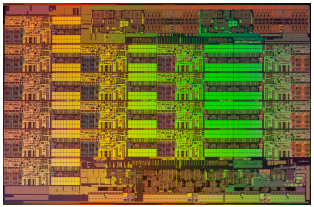
* socket and coprocessor versions

Knights Landing (KNL)

Intel Many Integrated Core (MIC) Architecture

INTEL XEON PROCESSORS

- 1-, 2-, 4-way
- General-purpose
- Highly parallel (44 cores*)
- Resource-rich
- Forgiving performance
- Theor. ~ 1.0 TFLOP/s in DP*
- Meas. ~ 154 GB/s bandwidth*

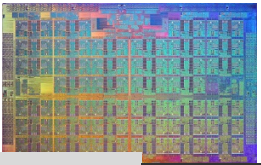


* 2-way Intel Xeon processor, Broadwell architecture (2016), top-of-the-line (e.g., E5-2699 V4)

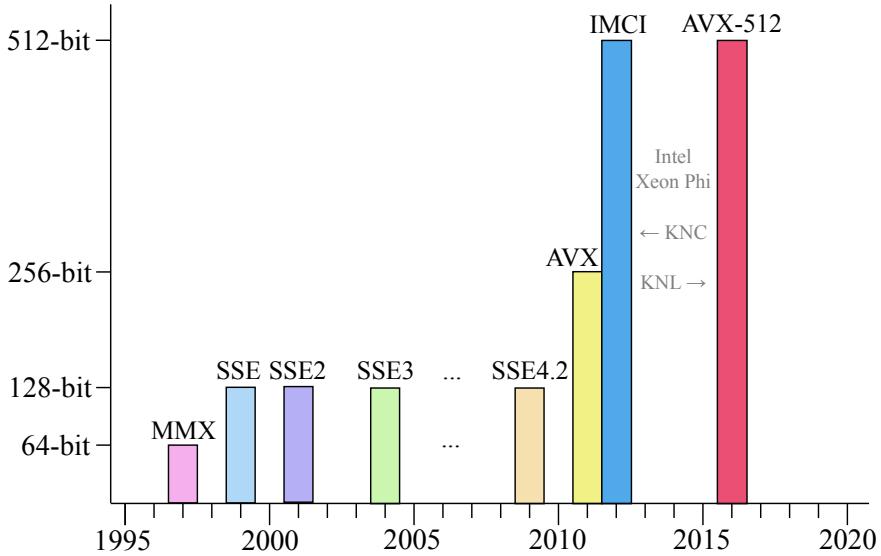
Intel Xeon Phi Processors (2nd Gen)

2nd Generation of Intel Many Integrated Core (MIC) Architecture.
Specialized platform for demanding computing applications.

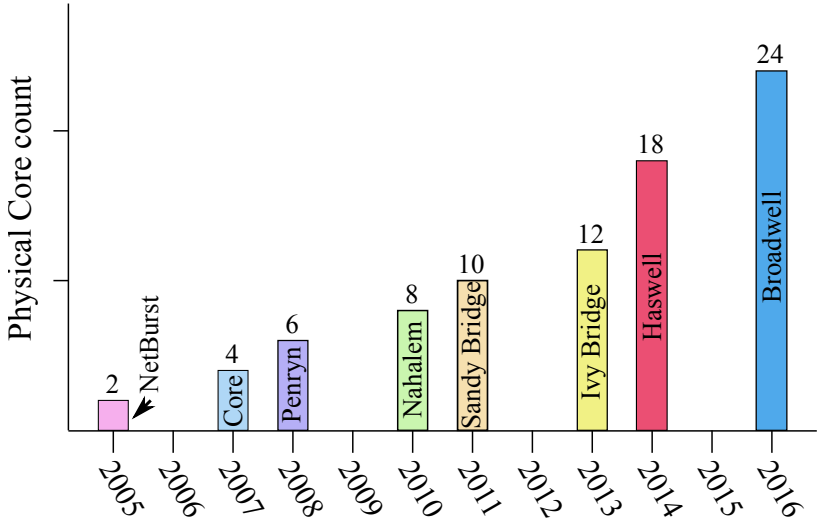
- Bootable host proccesor or coprocessor
- 3+ TFLOP/s DP
- 6+ TFLOP/s SP
- Up to 16 GiB MCDRAM
- MCDRAM bandwidth ≈5x DDR4
- Binary compatible with Intel Xeon
- More information



INSTRUCTION SETS IN INTEL ARCHITECTURE



INCREASING CORE COUNT (INTEL XEON)



§2. VECTORIZATION

SHORT VECTOR SUPPORT

Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.

Scalar Instructions

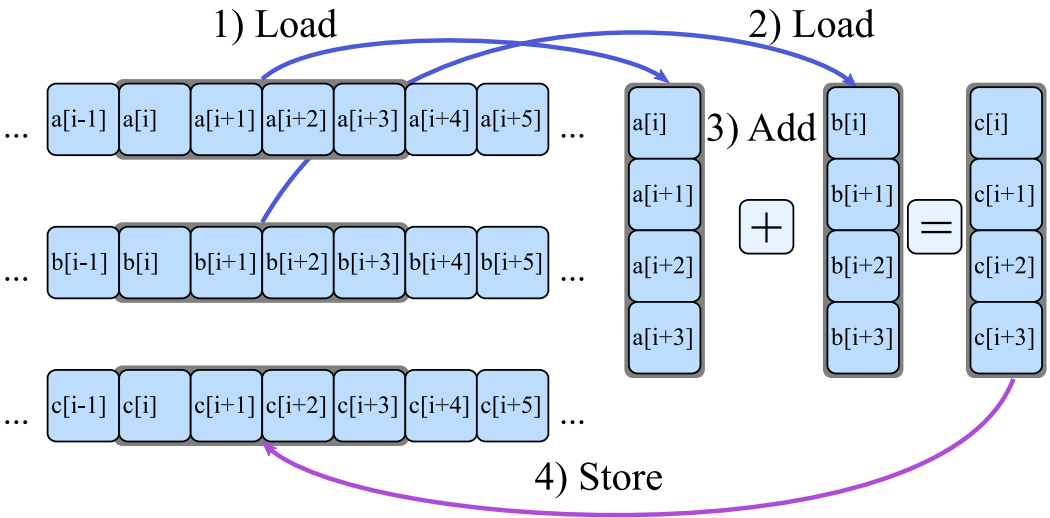
$$4 + 1 = 5$$
$$0 + 3 = 3$$
$$-2 + 8 = 6$$
$$9 + -7 = 2$$

Vector Instructions

$$\begin{matrix} 4 \\ 0 \\ -2 \\ 9 \end{matrix} + \begin{matrix} 1 \\ 3 \\ 8 \\ -7 \end{matrix} = \begin{matrix} 5 \\ 3 \\ 6 \\ 2 \end{matrix}$$

Vector Length

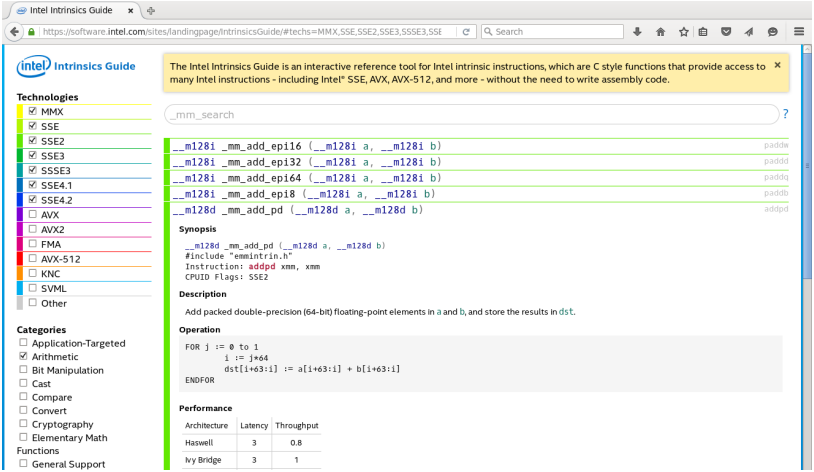
WORKFLOW OF VECTOR COMPUTATION



EXPLICIT VECTORIZATION

INTEL INTRINSICS GUIDE

https://software.intel.com/sites/landingpage/IntrinsicsGuide



EXAMPLE EXPLICIT VECTORIZATION (AVX512)

```
#include <immintrin.h>
// ... //
double *A = (double *) malloc(sizeof(double)*n);
double *B = (double *) malloc(sizeof(double)*n);
// ... //
for(int i = 0; i < n; i+=16) {
  // A[i] += B[i];
  __m512d Avec = _mm512_loadu_pd(&A[i]);
  __m512d Bvec = _mm512_loadu_pd(&B[i]);
  Avec = _mm512_add_pd(Avec, Bvec);
  _mm512_storeu_pd(&A[i], Avec);
}
```

```
student@cdt% icpc -xMIC_AVX512 explicit.cc
student@cdt% g++ -mavx512f -mavx512pf -mavx512cd -mavx512er explicit.cc
```

DETECTING AVAILABLE INSTRUCTIONS

In the OS:

```
[student@cdt ~]% cat /proc/cpuinfo
...
fpu_exception   : yes
cpuid level     : 11
wp              : yes
flags           : fpu vme de pse tsc msr pae mce
cx8 apic mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ss ht syscall nx lm constant_tsc
unfair_spinlock pni ssse3 cx16 sse4_1 sse4_2
x2apic popcnt aes hypervisor lahf_lm fsgsbase
bogomips        : 5985.17
clflush size    : 64
cache_alignment: 64
address sizes   : 46 bits physical, 48 bits virtual
...
```

In code (see also):

```
1  // Intel compiler
2  // preprocessor macros:
3
4  #ifdef __SSE__
5  // ...SSE code path
6  #endif
7
8  #ifdef __SSE4_2__
9  // ...SSE code path
10 #endif
11
12 #ifdef __AVX__
13 // ...AVX code path
14 #endif
```

AUTOMATIC VECTORIZATION

AUTOMATIC VECTORIZATION (INTEL COMPILER)

Intel Compilers have auto vectorization enabled by default:

```
student@cdt% icpc -xMIC_AVX512 automatic.cc
student@cdt% icpc -S -xMIC_AVX512 automatic.cc  # produce assembly
student@cdt% cat automatic.s                    # Default name. Change with -o
// ..... //
  vmovups   8(%r14,%rsi,8), %zmm0                        #17.5 c1
  vaddpd    8(%rax,%rsi,8), %zmm0, %zmm2                 #17.5 c13 stall 2
  vmovupd   %zmm2, 8(%r14,%rsi,8)                        #17.5 c19 stall 2
// ..... //
student@cdt% icpc -xMIC_AVX512 automatic.cc -qopt-report=5 # produce report
student@cdt% cat automatic.optrpt
// ..... //
LOOP BEGIN at automatic.cc(16,3)
   // ..... //
   remark #15300: LOOP WAS VECTORIZED
// ..... //
```

AUTOMATIC VECTORIZATION (GCC)

Easiest to enable with -O3 flag.

```
student@cdt% g++ -O3 -mavx512f -mavx512pf -mavx512cd -mavx512er -ffast-math \
% automatic.cc
student@cdt% g++ -O3 -S -mavx512f -mavx512pf -mavx512cd -mavx512er -ffast-math \
% -g -fverbose-asm automatic.cc                    # produce verbose assembly
student@cdt% cat automatic.s
// ..... //
  .loc 1 17 0 discriminator 2
  // ... //
  vaddpd  (%rsi,%rdx), %zmm0, %zmm0       # MEM[base: vectp_A.28_89, ....
student@cdt% g++ -O3 -mavx512f -mavx512pf -mavx512cd -mavx512er -ffast-math -g \
%  -fopt-info-vec -fopt-info-vec-missed automatic.cc   # produce verbose report
// ... //
automatic.cc:16:23: note: loop vectorized
automatic.cc:16:23: note: loop peeled for vectorization to enhance alignment
student@cdt% g++ -O3 -mavx512f -mavx512pf -mavx512cd -mavx512er -g \
%  -fopt-info-vec=v.rpt -fopt-info-vec-missed=v.rpt automatic.cc   # report file
```

OPENMP SIMD

OpenMP 4.0 introduced SIMD construct. Compiler will try to vectorize this loop.

```
#pragma omp simd
for(int i = 0; i < n; i++)
  A[i] += B[i];
```

With `parallel`. May need to define chunksize that is a multiple of vector length.

```
#pragma omp parallel for simd schedule(static,16)
for(int i = 0; i < n; i++)
  A[i] += B[i];
```

Nested `parallel` and `simd` construct.

```
#pragma omp parallel for
for(int i = 0; i < n; i++)
#pragma omp simd
  for(int j = 0; j < n; j++)
    A[i*n+j] += B[i*n+j];
```

LIMITATIONS OF AUTO-VECTORIZATION

LIMITATIONS OF AUTO-VECTORIZATION

There are certain limitations on automatic vectorization.

- Only for loops are supported. No while loops.
- Iteration count must be known at the beginning of the for loop.
- Loop can't contain non-vectorizable operations. (e.g. I/O)
- All functions are in-lined or declared simd.
- No vector dependence.

Any of these could prevent vectorization, but you may be able to find "hints" on what is preventing vectorization in the vectorization reports

SIMD-ENABLED FUNCTIONS

Define function in one file (e.g., library), use in another

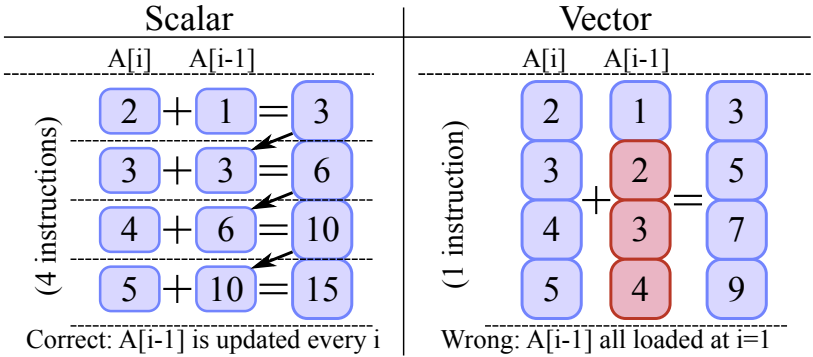
```
// Compiler will produce 3 versions:
#pragma omp declare simd
float my_simple_add(float x1, float x2){
  return x1 + x2;
}
```

```
// May be in a separate file
#pragma omp simd
for (int i = 0; i < N, ++i) {
  output[i] = my_simple_add(inputa[i], inputb[i]);
}
```

VECTOR DEPENDENCE

It is unsafe to vectorize a loop with vector dependence.

```
1 // A = {1,2,3,4,5}
2 for(int i = 1; i < 5; i++)
3   A[i] += A[i-1];
```



ASSUMED VECTOR DEPENDENCE

- True vector dependence – vectorization impossible:

```
1 float *a, *b;
2 for (int i = 1; i < n; i++)
3   a[i] += b[i]*a[i-1]; // dependence on the previous element
```

- *Assumed vector dependence* – compiler suspects dependence

```
1 void mycopy(int n,
2     float* a, float* b) {
3   for (int i=0; i<n; i++)
4     a[i] = b[i];
5 }
```

```
vega@lyra% icpc -c vdep.cc -qopt-report
vega@lyra% cat vdep.optrpt
...
remark #15304: loop was not
vectorized: non-vectorizable loop
instance from multiversioning
...
```

RESOLVING ASSUMED DEPENDENCY

- Restrict: Keyword indicating that there is no pointer aliasing (C++11)

```
1 void mycopy(int n,
2     float* restrict a,
3     float* restrict b) {
4   for (int i=0; i<n; i++)
5     a[i] = b[i];
6 }
```

```
vega@lyra% icpc -c vdep.cc -qopt-report \
%  -restrict
vega@lyra% cat vdep.optrpt
...
remark #15304: LOOP WAS VECTORIZED
...
```

- `#pragma ivdep`: ignores assumed dependency for a loop (Intel Compiler)

```
1 void mycopy(int n, float* a, float* b) {
2 #pragma ivdep
3   for (int i=0; i<n; i++)
4     a[i] = b[i];
5 }
```

§3. SNEAK PEAK

NOW WHAT?

I have a vectorized and multi-threaded code!

Some people stop here. But even if your application is multi-threaded and vectorized, it may not be optimal. Optimization could unlock more performance for your application.

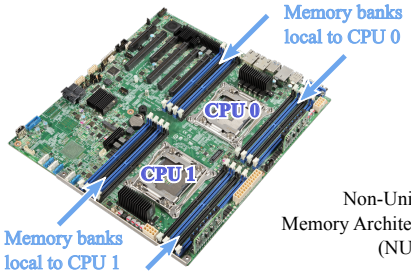
Example areas for consideration:

- Multi-threading
  - Do my threads have enough work?
  - Are my threads independent?
  - Is work distributed properly?
- Vectorization
  - Is my data organized well for vectorization?
  - Do I have regular loop patterns?

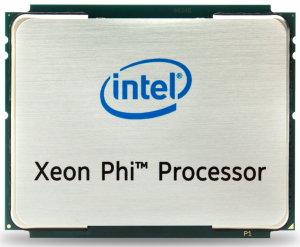
§4. ADDITIONAL TOPIC: WORKING WITH NUMA

NUMA ARCHITECTURES

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



Non-Uniform
Memory Architecture
(NUMA)

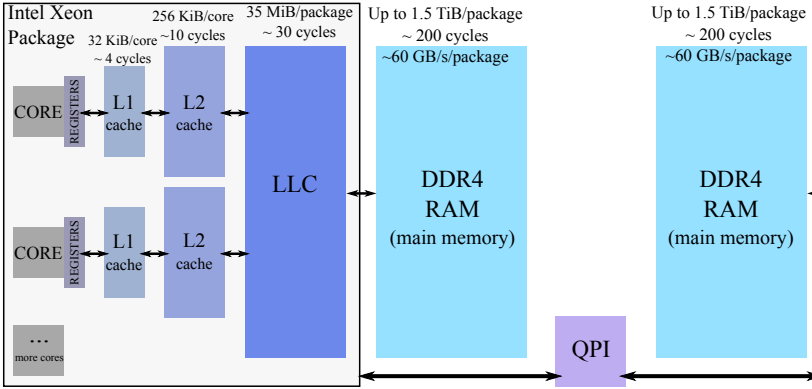


Examples:

- Multi-socket Intel Xeon processors
- Second generation Intel Xeon Phi in sub-NUMA clustering mode

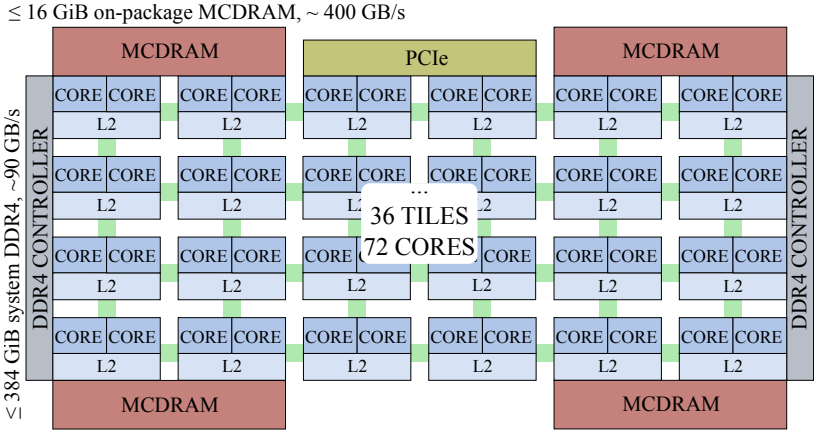
INTEL XEON CPU: MEMORY ORGANIZATION

- Hierarchical cache structure
- Two-way processors have NUMA architecture



KNL DIE ORGANIZATION: TILES

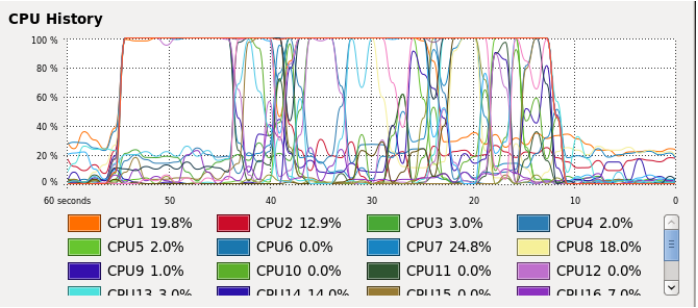
- Up to 36 tiles, each with 2 physical cores (72 total).
- Distributed L2 cache across a mesh interconnect.



THREAD AFFINITY

WHAT IS THREAD AFFINITY

- OpenMP threads may migrate between cores
- Forbid migration — improve locality — increase performance
- Affinity patterns “scatter” and “compact” may improve cache sharing, relieve thread contention



THE KMP_HW_SUBSET ENVIRONMENT VARIABLE

Control the # of cores and # of threads per core:

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

```
vega@lyra-mic0% export KMP_HW_SUBSET=3t # 3 threads per core
vega@lyra-mic0% ./my-native-app
```

or

```
vega@lyra% export MIC_ENV_PREFIX=XEONPHI
vega@lyra% export KMP_HW_SUBSET=1t # 1 thread per core on host
vega@lyra% export XEONPHI_KMP_HW_SUBSET=2t # 2 threads per core on Xeon Phi
vega@lyra% ./my-offload-app
```

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

OMP_PROC_BIND AND OMP_PLACES VARIABLES

Control the binding pattern, including nested parallelism:

```
OMP_PROC_BIND=type[,type[,...]]
```

Here `type=true`, `false`, `spread`, `close` or `master`.
Comma separates settings for different levels of nesting (`OMP_NESTED` must be enabled).

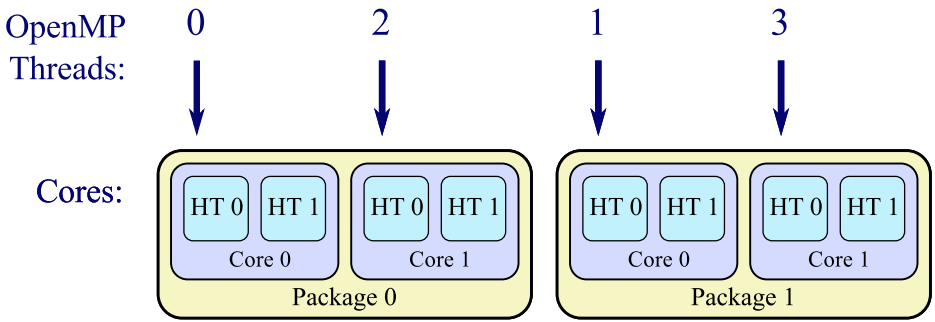
Control the granularity of binding:

```
OMP_PLACES=<threads|cores|sockets|(explicit)>
```

THREAD AFFINITY: SCATTER PATTERN

Generally beneficial for bandwidth-bound applications.

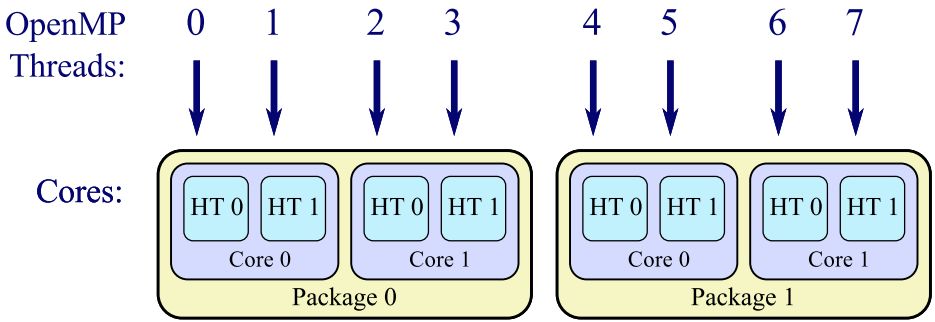
`OMP_NUM_THREADS={1 thread/core}` or `KMP_HW_SUBSET=1t`
`KMP_AFFINITY=scatter,granularity=fine`



THREAD AFFINITY: COMPACT PATTERN

Generally beneficial for compute-bound applications.

```
OMP_NUM_THREADS={2(4) threads/core on Xeon (Xeon Phi)}
KMP_AFFINITY=compact,granularity=fine
```



PARALLELISM AND AFFINITY INTERFACES

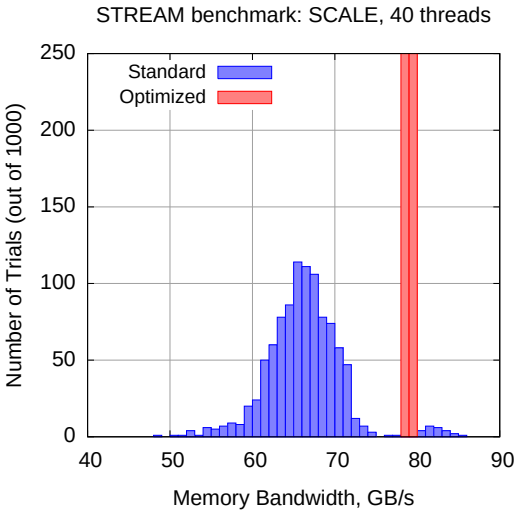
Intel-specific (in order of priority):

- Functions (e.g., `kmp_set_affinity()`)
- Compiler arguments (e.g., `-par-affinity`)
- Environment variables (e.g., `KMP_AFFINITY`)

Defined by the OpenMP standard (in order of priority):

- Clauses in pragmas (e.g., `proc_bind`)
- Functions (e.g., `omp_set_num_threads()`)
- Environment variables (e.g., `OMP_PROC_BIND`)

IMPACT OF AFFINITY ON BANDWIDTH



- Without affinity: "fortunate" and "unfortunate" runs
- With affinity "scatter": consistently good performance

Plot from this paper

FIRST-TOUCH LOCALITY

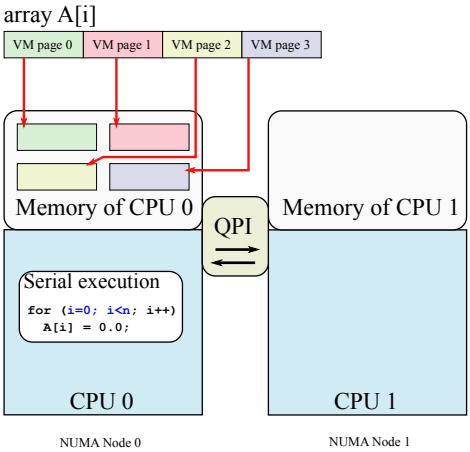
ALLOCATION ON FIRST TOUCH

- 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

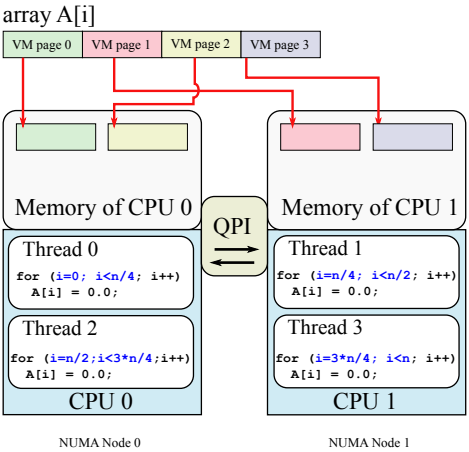
```
1  float* A = (float*)_mm_malloc(n*m*sizeof(float), 64);
2
3  // Initializing from parallel region for better performance
4  #pragma omp parallel for
5  for (int i = 0; i < n; i++)
6    for (int j = 0; j < m; j++)
7      A[i*m + j] = 0.0f;
```

FIRST-TOUCH ALLOCATION POLICY

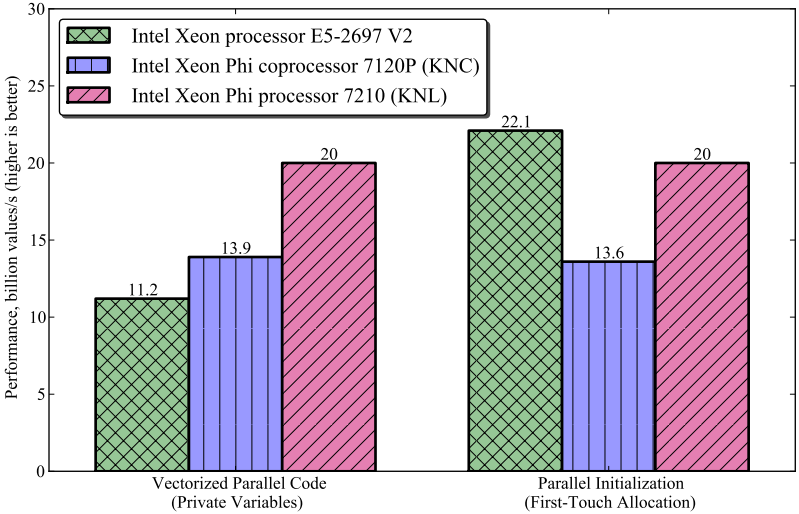
Poor First-Touch Allocation



Good First-Touch Allocation



IMPACT OF FIRST-TOUCH ALLOCATION



BINDING TO NUMA NODES WITH `numactl`

- `libnuma` – a Linux library for fine-grained control over NUMA policy
- `numactl` – a tool for global NUMA policy control

```
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
```