DAY 5: INTRODUCTION TO PARALLEL INTEL® ARCHITECTURES

Lecture day 5

Ryo Asai
Colfax International — colfaxresearch.com

April 2017
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 synchronization 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.

- Scalar Tuning
- Vectorization
- Memory
- Threading
- Communication

Node-level
Cluster-level
http://colfaxresearch.com/
§1. INTRODUCTION
INTRODUCTION

Source: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/
Task Parallelism – multiple instructions multiple data elements (MIMD)

Data Parallelism – single instruction multiple data elements (SIMD)

Unbounded growth opportunity, but not automatic
Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.

<table>
<thead>
<tr>
<th>Scalar Instructions</th>
<th>Vector Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 + 1 = 5</td>
<td>4 + 1 = 5</td>
</tr>
<tr>
<td>0 + 3 = 3</td>
<td>0 + 3 = 3</td>
</tr>
<tr>
<td>-2 + 8 = 6</td>
<td>-2 + 8 = 6</td>
</tr>
<tr>
<td>9 + -7 = 2</td>
<td>9 + -7 = 2</td>
</tr>
</tbody>
</table>
INTEL ARCHITECTURES
Intel Xeon Processor
Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi Coprocessor, 1st generation
Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*
Knights Landing (KNL)

* socket and coprocessor versions

Multi-Core Architecture

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

- Bootable host processor 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

© Colfax International, 2013–2017
INCREASING CORE COUNT (INTEL XEON)

Physical Core count

- 2005: NetBurst
- 2006: Core
- 2007: Penryn
- 2008: Sandy Bridge
- 2009: Nehalem
- 2010: Ivy Bridge
- 2011: Haswell
- 2012: Broadwell
- 2013: 12
- 2014: 18
- 2015: 24
- 2016: 24

INTEL ARCHITECTURES

© Colfax International, 2013–2017
§2. VECTORIZATION
Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.
WORKFLOW OF VECTOR COMPUTATION

1) Load

... a[i-1] a[i] a[i+1] a[i+2] a[i+3] a[i+4] a[i+5] ...

... b[i-1] b[i] b[i+1] b[i+2] b[i+3] b[i+4] b[i+5] ...

... c[i-1] c[i] c[i+1] c[i+2] c[i+3] c[i+4] c[i+5] ...

2) Load

... a[i] a[i+1] a[i+2] a[i+3] a[i+4] a[i+5] ...

... b[i] b[i+1] b[i+2] b[i+3] ...

... c[i] c[i+1] c[i+2] ...

3) Add

+ =

... c[i+1] ...

... b[i+1] ...

... a[i+3] ...

4) Store
EXPLICIT VECTORIZATION
The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

**Synopsis**

```
_mm_search

_mm_add_epi16 (__m1281 a, __m1281 b)
_mm_add_epi32 (__m1281 a, __m1281 b)
_mm_add_epi64 (__m1281 a, __m1281 b)
_mm_add_epi8 (__m1281 a, __m1281 b)
_mm_add_pd (__m128d a, __m128d b)
```

**Description**

Add packed double-precision (64-bit) floating-point elements in `a` and `b`, and store the results in `dst`.

**Operation**

```
FOR j := 0 to 1,
  i := j*64
ENDFOR
```

**Performance**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haswell</td>
<td>3</td>
<td>0.8</td>
</tr>
<tr>
<td>Ivy Bridge</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
#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);
}
In the OS:

```bash
[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):

```c
// Intel compiler
// preprocessor macros:

#ifdef __SSE__
// ...SSE code path
#endif

#ifdef __SSE4_2__
// ...SSE code path
#endif

#ifdef __AVX__
// ...AVX code path
#endif
```
AUTOMATIC VECTORIZATION
Automatic Vectorization (Intel Compiler)

Intel Compilers have auto vectorization enabled by default:

```bash
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.

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

```c
#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.

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

Nested parallel and simd construct.

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

```c
// 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]);
}
```
It is unsafe to vectorize a loop with vector dependence.

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

<table>
<thead>
<tr>
<th>Scalar</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i]</td>
<td>A[i]</td>
</tr>
<tr>
<td>A[i-1]</td>
<td>A[i-1]</td>
</tr>
<tr>
<td>2 + 1</td>
<td>3</td>
</tr>
<tr>
<td>3 + 3</td>
<td>6</td>
</tr>
<tr>
<td>4 + 6</td>
<td>10</td>
</tr>
<tr>
<td>5 + 10</td>
<td>15</td>
</tr>
</tbody>
</table>

Correct: A[i-1] is updated every i
Wrong: A[i-1] all loaded at i=1
True vector dependence – vectorization impossible:

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

Assumed vector dependence – compiler suspects dependence

```c
void mycopy(int n,
            float* a, float* b) {
    for (int i=0; i<n; i++)
a[i] = b[i];
}
```

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

▷ Restricted: Keyword indicating that there is no pointer aliasing (C++11)

```cpp
class MyClass {
  private:
    int x;
  public:
    MyClass() : x(0) {}
    void doSomething() {
      // Assumed dependent code...
    }
};
```

```cpp
void mycopy(int n, float* restrict a, float* restrict b) {
  for (int i=0; i<n; i++)
    a[i] = b[i];
}
```

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)

```cpp
void mycopy(int n, float* a, float* b) {
  #pragma ivdep
  for (int i=0; i<n; i++)
    a[i] = b[i];
}
```
§3. SNEAK PEAK
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.

Examples:
- Multi-socket Intel Xeon processors
- Second generation Intel Xeon Phi in sub-NUMA clustering mode
Hierarchical cache structure

Two-way processors have NUMA architecture
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**

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

**Modifier:**
- verbose/nonverbose
- respect/norespect
- warnings/nowarnings
- granularity=core or thread

**Type:**
- compact, scatter or balanced
- explicit, proclist= [<proc_list>]
- 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)>
```
Generally beneficial for bandwidth-bound applications.

OMP_NUM_THREADS={1 thread/core} or KMP_HW_SUBSET=1t
KMP_AFFINITY=scatter,granularity=fine
Generally beneficial for compute-bound applications.

OMP_NUM_THREADS={2(4) threads/core on Xeon (Xeon Phi)}

KMP_AFFINITY=compact,granularity=fine
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

STREAM benchmark: SCALE, 40 threads

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

Plot from this paper
FIRST-TOUCH LOCALITY
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

```c
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;
```
 Poor First-Touch Allocation

array A[i]

for (i=0; i<n; i++)
    A[i] = 0.0;

Serial execution

CPU 0

Thread 0

for (i=0; i<n/4; i++)
    A[i] = 0.0;

Thread 2

for (i=n/2; i<3*n/4; i++)
    A[i] = 0.0;

CPU 1

Thread 1

for (i=n/4; i<n/2; i++)
    A[i] = 0.0;

Thread 3

for (i=3*n/4; i<n; i++)
    A[i] = 0.0;

Good First-Touch Allocation

array A[i]

for (i=0; i<n; i++)
    A[i] = 0.0;

Serial execution

CPU 0

Thread 0

for (i=0; i<n/4; i++)
    A[i] = 0.0;

CPU 1

Thread 1

for (i=n/4; i<n/2; i++)
    A[i] = 0.0;

CPU 0

Thread 2

for (i=n/2; i<3*n/4; i++)
    A[i] = 0.0;

CPU 1

Thread 3

for (i=3*n/4; i<n; i++)
    A[i] = 0.0;

© Colfax International, 2013–2017
Impact of First-Touch Allocation

Vectorized Parallel Code (Private Variables) vs. Parallel Initialization (First-Touch Allocation)

Performance, billion values/s (higher is better)

- Intel Xeon processor E5-2697 V2
- Intel Xeon Phi coprocessor 7120P (KNC)
- Intel Xeon Phi processor 7210 (KNL)

First-Touch Locality

© Colfax International, 2013–2017
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

```bash
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, 2013–2017