PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

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

Session 2

Colfax International — *colfaxresearch.com*

WELCOME

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

- Module I. Programming Models
  - 01. Intel Architecture and Modern Code
  - 02. Xeon Phi, Coprocessors, Omni-Path
- Module II. Expressing Parallelism
  - 03. Automatic vectorization
  - 04. Multi-threading with OpenMP
  - 05. Distributed Computing, MPI
- Module III. Performance Optimization
  - 06. Optimization Overview: N-body
  - 07. Scalar tuning, Vectorization
  - 08. Common Multi-threading Problems
  - 09. Multi-threading, Memory Aspect
  - 10. Access to Caches and Memory

HOW SERIES ONLINE

Course page:
colfaxresearch.com/how-series

- Slides
- Code
- Video
- Chat

More workshops:
colfaxresearch.com/training

GET YOUR QUESTIONS ANSWERED: CHAT

colfaxresearch.com/how-series

GET YOUR QUESTIONS ANSWERED: FORUMS

READ | WATCH | LEARN | FORUMS | CONNECT | JOIN

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

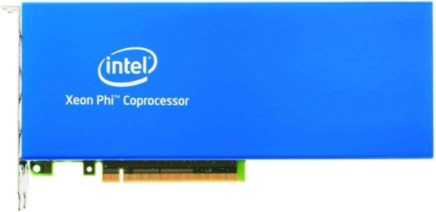
HANDS-ON EXERCISES AND REMOTE ACCESS

- 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

COLFAX CLUSTER

§2. ROADMAP OF INTEL ARCHITECTURE

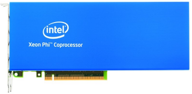
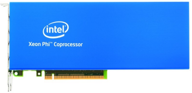
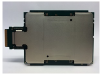
COMPUTING PLATFORMS

Intel Xeon Processor	Intel Xeon Phi Coprocessor, 1st generation	Intel Xeon Phi Processor, 2nd generation*
		
Current: Broadwell Upcoming: Skylake	Knights Corner (KNC)	* socket and coprocessor versions Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

INTEL XEON PHI PROCESSORS

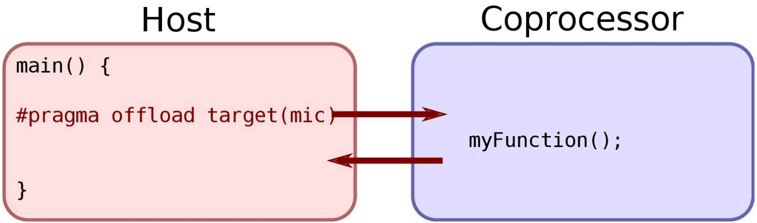
	Knights Corner	Knights Landing	Knights Mill	Knights Hill
Lith	22 nm	14 nm	14 nm	10 nm
Models	71xx P/A	72xx, 72xx F	?	?
Form-factors	coprocessor 	processor  coprocessor  processor with fabric 	?	?

§3. PROGRAMMING COPROCESSORS

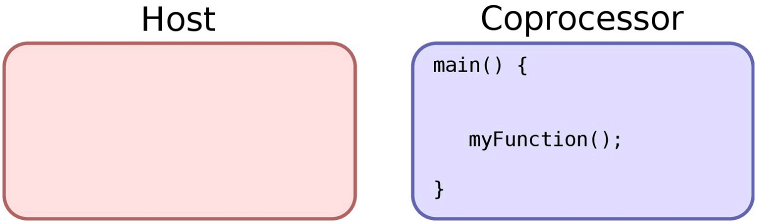
OFFLOAD AND NATIVE MODELS

OFFLOAD AND NATIVE MODELS

- Offload model (explicit/virtual-shared memory/OpenMP 4.0):



- Native model (standalone application/MPI process):



OFFLOAD OR NATIVE: HOW TO DECIDE

Native	**Offload**
$\leq$ 16 GiB	$>$ 16 GiB
All parallel	Parallel + serial phases
Complex data structures	Bitwise-copyable data
Any arithmetic intensity	(FLOPs/transfer) $\gg$ 1000

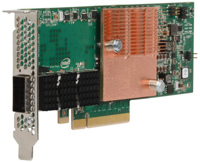
Native = same code on CPU and MIC

Offload = must insert directives in code

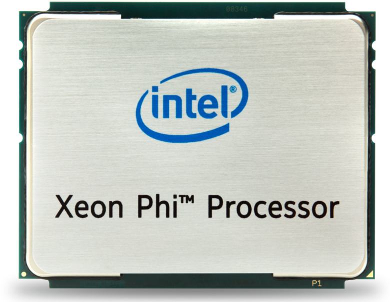
OFFLOAD OVER FABRIC

Heterogeneous computing is possible even with bootable KNL

+



+



Tutorial: Offload over Fabric to Intel Xeon Phi Processor

NATIVE PROGRAMMING (KNC)

INTEL COMPILERS + INTEL XEON PROCESSOR

"Hello World" application:

```
#include <cstdio>
#include <unistd.h>
int main(){
    printf("Hello world! I have %ld logical processors.\n",
    sysconf(_SC_NPROCESSORS_ONLN ));
}
```

Compile and run on host CPU:

```
vega@lyra% icpc hello.cc -xhost
vega@lyra% ./a.out
Hello world! I have 48 logical processors.
vega@lyra%
```

NATIVE EXECUTION ON AN INTEL XEON PHI COPROCESSOR (KNC)

Compile and run the same code on the coprocessor in the native mode:

```
vega@lyra% icpc hello.cc -mmic # Cross-compile
vega@lyra% scp a.out mic0:~/ # Put executable on coprocessor
a.out 100% 10KB 10.4KB/s 00:00
vega@lyra% ssh mic0 # Log in to coprocessor
vega@mic0% pwd
/home/lyra
vega@mic0% ls
a.out
vega@mic0% ./a.out # Launch application
Hello world! I have 244 logical processors.
vega@mic0%
```

- Use `-mmic` to produce executable for MIC architecture
- Must transfer executable to coprocessor (or NFS-share) and run from shell
- Native MPI applications work the same way (need Intel MPI library)

NATIVE APPLICATIONS WITH AUTOTOOLS

- Use the Intel compiler with flag `-mmic`
- Knights Landing: `-xMIC-AVX512`
- Eliminate assembly and unncecessary dependencies
- Use `--host=x86_64` to avoid "program does not run" errors

Example, the GNU Multiple Precision Arithmetic Library (GMP):

```
vega@lyra% wget https://ftp.gnu.org/gnu/gmp/gmp-5.1.3.tar.bz2
vega@lyra% tar -xf gmp-5.1.3.tar.bz2
vega@lyra% cd gmp-5.1.3
vega@lyra% ./configure CC=icc CFLAGS="-mmic" --host=x86_64 --disable-assembly
...
vega@lyra% make
...
```

EXPLICIT OFFLOAD (LEO)

Explicit Offload: Pragma-Based Approach

"Hello World" in the explicit offload model:

```
#include <cstdio>
int main() {
    printf("Hello World from host!\n");
#pragma offload target(mic)
    {
        printf("Hello World from coprocessor!\n"); fflush(stdout);
    }
    printf("Bye\n");
}
```

Application runs on the host, but some parts of code and date are moved ("offloaded") the coprocessor.

Detailed syntax in the Intel C++ Compiler Reference.

Compiling and Running an Offload Application

```
vega@lyra% icpc hello_offload.cc -o hello_offload
vega@lyra% ./hello_offload
Hello World from host!
Bye
Hello World from coprocessor!
```

- No additional arguments (for Intel compiler)
- Launch on host as a regular application
- Code inside of `#pragma offload` is offloaded automatically
- Console output on coprocessor buffered, mirrored to the host
- If no coprocessor available, default behavior is error; may be overridden to fall back to host

OFFLOADING FUNCTIONS AND DATA

OFFLOADING MULTIPLE FUNCTIONS

```
#pragma offload_attribute(push, target(mic))
void MyFunctionOne() {
// ... implement function as usual
}

void MyFunctionTwo() {
// ... implement function as usual
}
#pragma offload_attribute(pop)
```

- To mark a long block of code with the offload attribute, use `#pragma offload_attribute(push/pop)`

OFFLOADING DATA: LOCAL SCALARS AND ARRAYS

```
void MyFunction() {
    const int N = 1000;
    int data[N];
#pragma offload target(mic)
    {
        for (int i = 0; i < N; i++)
          data[i] = 0;
    }
```

- Scope-local scalars and known-size arrays offloaded automatically
- Data is copied from host to coprocessor at the start of offload
- Data is copied back from coprocessor to host at the end of offload
- Bitwise-copyable data only (arrays of basic types and scalars)
  C++ classes, etc. should use virtual-shared memory model

DATA MARSHALLING FOR DYNAMICALLY ALLOCATED DATA

```
double *p1=(double*)malloc(sizeof(double)*N);
double *p2=(double*)malloc(sizeof(double)*N);

#pragma offload target(mic) in(p1 : length(N)) out(p2 : length(N))
{
    // ... perform operations on p1[] and p2[]
}
```

- `#pragma offload` recognizes clauses `in`, `out`, `inout` and `nocopy`
- Data size (length), alignment, redirection, and other properties may be specified
- Marshalling is required for pointer-based data

OPTIONAL OFFLOAD, FALL-BACK TO HOST

```
#pragma offload target(mic) optional
{
  printf("Hello World! I have %d logical processors.\n",
  sysconf(_SC_NPROCESSORS_ONLN )); fflush(stdout);
}
```

```
vega@lyra% icpc Offload-Fallback.cc -o Offload-Fallback
vega@lyra% ./Offload-Fallback
Hello World! I have 244 logical processors.
vega@lyra% sudo systemctl stop mpss # Disabling coprocessors
vega@lyra% ./Offload-Fallback
Hello World! I have 48 logical processors.
```

MEMORY ALLOCATION CONTROL

MEMORY RETENTION AND DATA PERSISTENCE ON COPROCESSOR

- By default, memory on coprocessor is allocated before, deallocated after offload
- Specifiers `alloc_if` and `free_if` allow to avoid allocation/deallocation
- Data transfer across the PCIe bus rate is ≈ 7 GB/s
- To allocate memory on the coprocessor – 0.5-2.0 GB/s

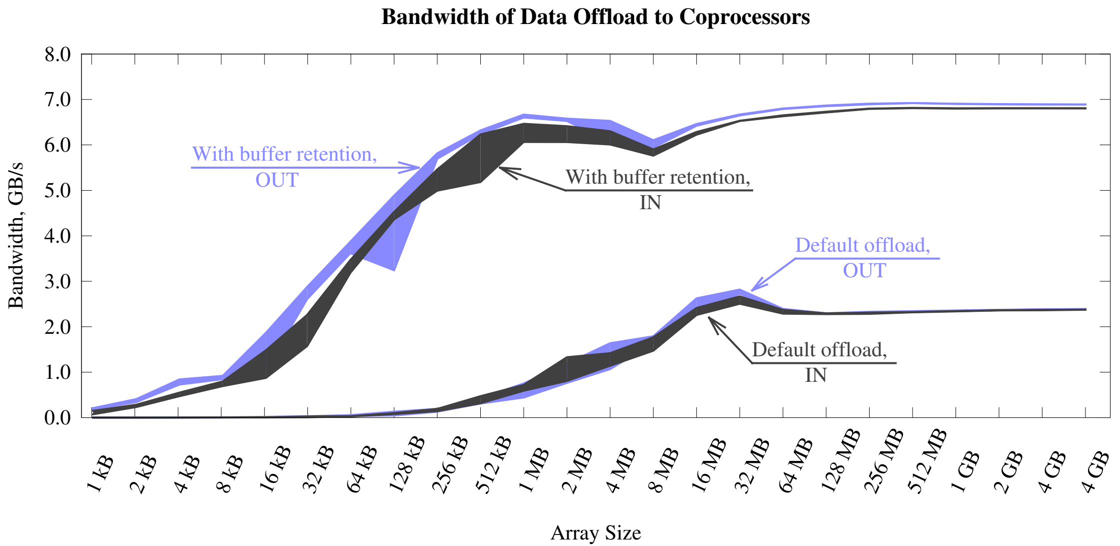
```
#pragma offload target(mic:0) in(p : length(N) alloc_if(1) free_if(0) )
{ /* allocate memory for array p on coprocessor, do not deallocate */ }

#pragma offload target(mic:0) in(p : length(N) alloc_if(0) free_if(0) )
{ /* re-use previously allocated memory buffer on coprocessor */ }

#pragma offload target(mic:0) in(p : length(0) alloc_if(0) free_if(0) )
{ /* re-use previously transferred data on coprocessor */ }

#pragma offload target(mic:0) out(p : length(N) alloc_if(0) free_if(1) )
{ /* re-use memory and deallocate at the end of offload */ }
```

OFFLOAD LATENCY WITH AND WITHOUT MEMORY/DATA RETENTION



ADDITIONAL OFFLOAD CONTROLS

TARGET-SPECIFIC CODE

- During MIC architecture compilation, preprocessor macro `__MIC__` is defined.
- Allows to fine-tune application performance or output where necessary

```
__attribute__((target(mic))) void MyFunction() {
#ifdef __MIC__
    printf("I am running on a coprocessor.\n");
    const int tuningParameter = 16;
#else
    printf("I am running on the host.\n");
    const int tuningParameter = 8;
#endif
    // ... Proceed, using the variable tuningParameter
}
```

OFFLOAD DIAGNOSTICS

```
vega@lyra% export OFFLOAD_REPORT=2
vega@lyra% ./offload-application
Transferring some data to and from coprocessor...
Done. Bye!
[Offload] [MIC 0] [File]            offload-application.cc
[Offload] [MIC 0] [Line]            6
[Offload] [MIC 0] [CPU Time]        0.505982 (seconds)
[Offload] [MIC 0] [CPU->MIC Data]   1024 (bytes)
[Offload] [MIC 0] [MIC Time]        0.000409 (seconds)
[Offload] [MIC 0] [MIC->CPU Data]   1024 (bytes)
vega@lyra%
```

- Set environment variable `OFFLOAD_REPORT` to 1 or 2 for automatic collection and output of offload information.
- Unset or set `OFFLOAD_REPORT=0` to disable offload diagnostics

ENVIRONMENT VARIABLE FORWARDING WITH OFFLOAD

- By default, all host environment variables on the host will be copied to the coprocessor when offload starts.
- In order to have different values for an environment variable on host and coprocessor, set `MIC_ENV_PREFIX`
- The prefix is dropped when variables are copied to coprocessor

```
vega@lyra% # This sets the value of OMP_NUM_THREADS on the host:
vega@lyra% export OMP_NUM_THREADS=48
vega@lyra%
vega@lyra% # This enables special rules for variable copying:
vega@lyra% export MIC_ENV_PREFIX=XEONPHI
vega@lyra%
vega@lyra% # This sets the value of OMP_NUM_THREADS on the coprocessor:
vega@lyra% export XEONPHI_OMP_NUM_THREADS=240
```

OFFLOAD IN OPENMP 4.0

OpenMP 4.0 Target Offload

- Another API for offload: `#pragma omp target`
- Part of the OpenMP 4.0 standard
- Designed as portable solution (coprocessors, GPGPUs)
- On Xeon Phi, uses the same back-end as `#pragma offload`

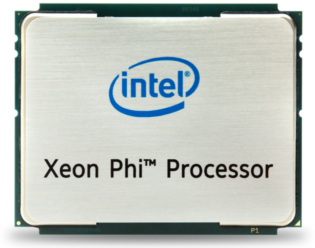
```
#pragma omp target
  {
#pragma omp parallel for
    for(int i=0; i<size; i++)
      data[i] = 0;
  }
```

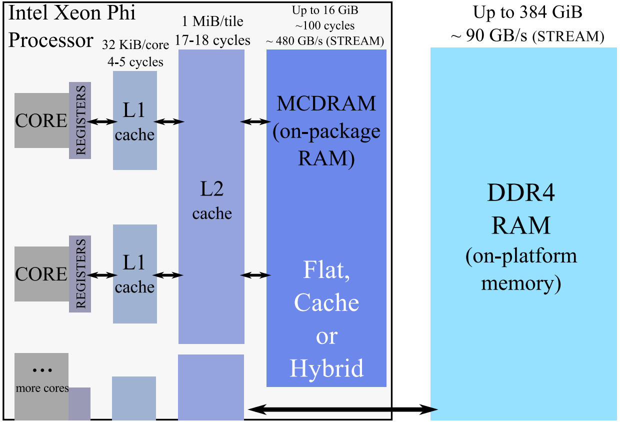
Application runs on the host, but some parts of code and data are moved ("offloaded") the coprocessor. Scope-local scalars and stack arrays offloaded automatically.

§4. HIGH-BANDWIDTH MEMORY

KNL MEMORY ORGANIZATION (BOOTABLE)

- On-package high-bandwidth memory (HBM) – MCDRAM
- Optimized for arithmetic performance and bandwidth (not latency)

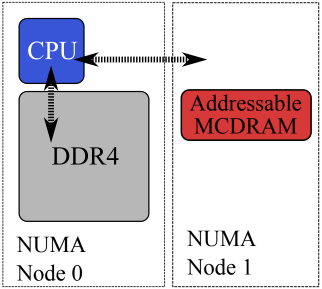




HIGH-BANDWIDTH MEMORY MODES

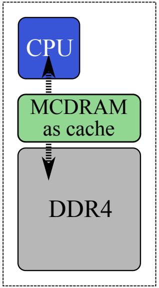
Flat Mode

- MCDRAM treated as a NUMA node
- Users control what goes to MCDRAM



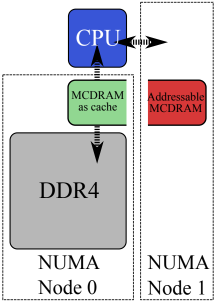
Cache Mode

- MCDRAM treated as a Last Level Cache (LLC)
- MCDRAM is used automatically



Hybrid Mode

- Combination of Flat and Cache
- Ratio can be chosen in the BIOS



RUNNING APPLICATIONS IN HBM WITH NUMACTL

- Finding information about the NUMA nodes in the system.

```
user@knl% # In Flat mode of MCDRAM
user@knl% numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ... 254 255
node 0 size: 98207 MB
node 1 cpus:
node 1 size: 16384 MB
```

- Binding the application to HBM (Flat/Hybrid)

```
user@knl% icc myapp.c -o runme -xMIC_AVX512
user@knl% numactl --membind 1 ./runme
// ... Application running in HBM ... //
```

ALLOCATION IN HBM WITH MEMKIND LIBRARY

```
#include <hbwmalloc.h>
const int n = 1<<10;
// Allocation to MCDRAM
double* A = (double*) hbw_malloc(sizeof(double)*n);
// No replacement for _mm_malloc. Use posix_memalign
double* B;
int ret = hbw_posix_memalign((void**) &B, 64, sizeof(double)*n);
.....
// Free with hbw_free
hbw_free(A); hbw_free(B);
```

Compilation with Memkind Library and HBWMALLOC

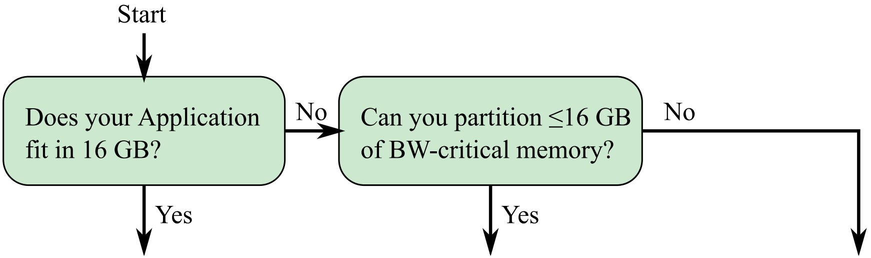
To compile C/C++ applications:

```
user@knl%  icpc -lmemkind foo.cc -o runme
user@knl%  g++ -lmemkind foo.cc -o runme
```

Open source distribution of Memkind library can be found at:
memkind.github.io/memkind

Learn more:
colfaxresearch.com/knl-mcdram

FLOW CHART FOR BANDWIDTH-BOUND APPLICATIONS

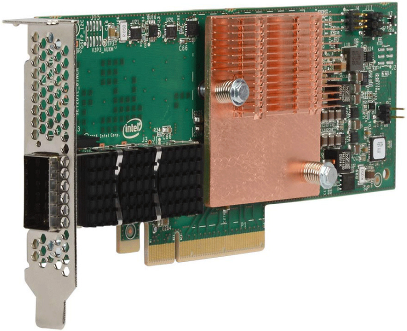


numactl	**Memkind**	**Cache mode**
▷ Simply run the whole program in MCDRAM ▷ No code modification required	▷ Manually allocate BW-critical memory to MCDRAM ▷ Memkind calls need to be added.	▷ Allow the chip to figure out how to use MCDRAM ▷ No code modification required

§5. INTEL OMNI-PATH ARCHITECTURE

INTEL'S HPC COMMUNICATION FABRIC

Intel Omni-Path Architecture - low-latency, high-bandwidth, scalable communication fabric for HPC applications.



Discrete

Integrated

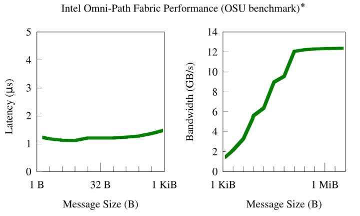
INTEL OMNI-PATH FABRIC 100 WITH INTEL XEON PROCESSORS

First generation: 100 Gbps bandwidth, ≈ 1 microsecond latency

Intel® Omni-Path Fabric

```
File Edit View Search Terminal Help
[ryo@c010-n003 ~]$ cat /sys/module/hfi1/parameters/desct_intr
64
[ryo@c010-n003 ~]$ cat /sys/module/hfi1/parameters/sdma_descq_cnt
2048
[ryo@c010-n003 ~]$ mpirun -np 2 -ppn 1 -host c010-n003,c010-n004 \
                  -PSM -genv I_MPI_PIN_PROCESSOR_LIST=0 ~/osu_bw

# OSU MPI Bandwidth Test v5.0
# Size        Bandwidth (MB/s)
1                        1.77
2                        3.54
4                        7.22
8                       14.74
16                      26.96
32                      57.22
64                     114.45
128                    232.93
256                    405.45
512                    753.79
1024                  1386.73
2048                  2264.14
4096                  3389.89
8192                  5612.19
16384                 6375.83
32768                 8975.61
65536                 9558.37
131072               12061.21
262144               12222.53
524288               12306.10
1048576              12338.90
2097152              12354.31
4194304              12379.14
[ryo@c010-n003 ~]$
```



* Pre-production A0 hardware and Alpha-level software

www.colfax-intl.com

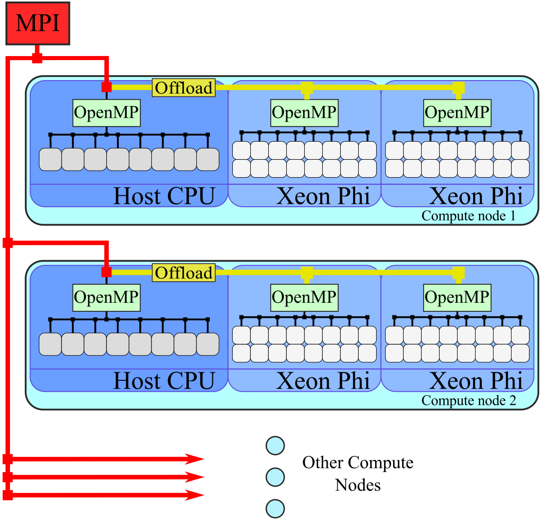
COLFAX Customized Solutions

- Rely on MPI for platform-independent communication
- Intel MPI: set `I_MPI_FABRICS=tmi`.

HETEROGENEOUS DISTRIBUTED COMPUTING WITH XEON PHI

Option 1: MPI+OpenMP with Offload.

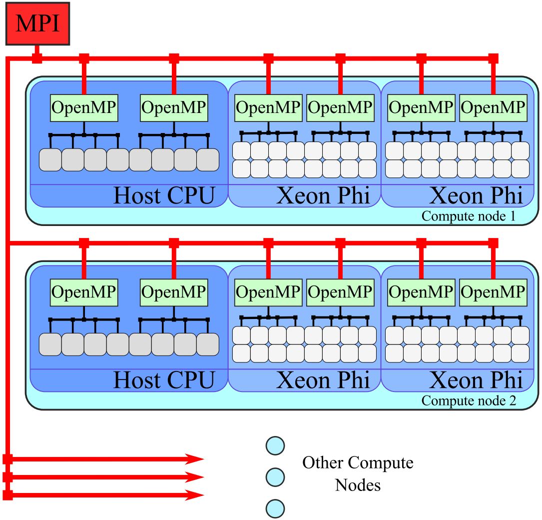
- MPI processes are multi-threaded with OpenMP.
- MPI runs only on CPUs.
- MPI processes offload to coprocessor(s).
- OpenMP in offload regions.



HETEROGENEOUS DISTRIBUTED COMPUTING WITH XEON PHI

Option 2: Symmetric hybrid MPI+OpenMP.

- MPI processes on hosts
- Native MPI processes on the coprocessor.
- Multi-threading with OpenMP.

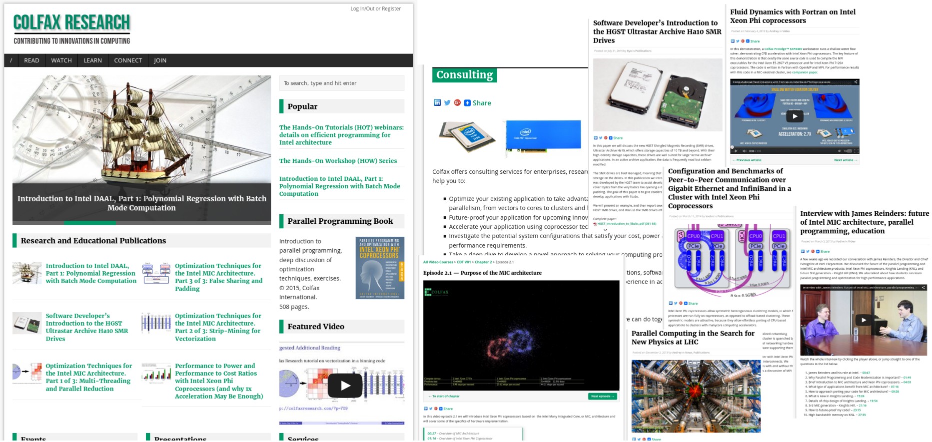


REVIEW AND WHAT'S NEXT

- Coprocessor programming: native and offload models
- High-bandwidth memory: cache mode or flat mode
- Intel OPA: use MPI for transparent, portable programming

Next session: expressing data parallelism, vectorization.

COLFAX RESEARCH



https://colfaxresearch.com/