COLFAX
Customized Solutions

Primer on Computing with Intel Xeon Phi Coprocessors

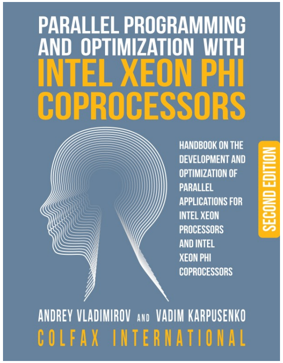
Part 1 of 2: Introduction and Programming Models

Vadim Karpusenko and Andrey Vladimirov
Colfax International

Intel HPC Developer Conference at SC14 in New Orleans, LA — November 16, 2014

Contents

intel
Xeon Phi™ Coprocessor


PARALLEL PROGRAMMING AND OPTIMIZATION WITH
INTEL XEON PHI COPROCESSORS
HANDBOOK ON THE DEVELOPMENT AND OPTIMIZATION OF PARALLEL APPLICATIONS FOR INTEL XEON PROCESSORS AND INTEL XEON PHI COPROCESSORS
SECOND EDITION
ANDREY VLADIMIROV AND VADIM KARPUSENKO
COLFAX INTERNATIONAL

§1. MIC Architecture from Developer's Perspective

Intel Xeon Phi Coprocessors and the MIC Architecture

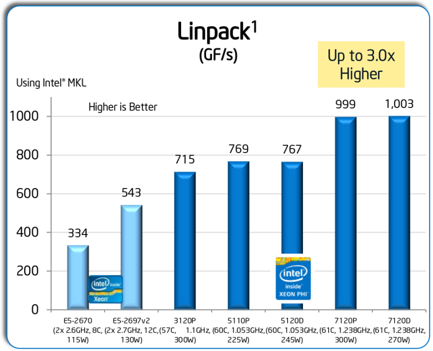
- PCIe end-point device
- High Power efficiency
- ~ 1 TFLOP/s in DP
- Heterogeneous clustering

For highly parallel applications which reach the scaling limits on Intel Xeon processors

Xeon and Xeon Phi Family Product Performance

Many-core Coprocessors (Xeon Phi) vs Multi-core Processors (Xeon) —

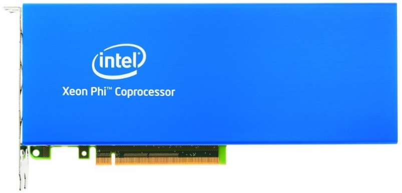
- Better performance per system & performance per watt for parallel applications
- Same programming methods, same development tools.



1. Xeon ran MP Linpack, Xeon Phi ran SMP Linpack. Expected performance difference between the two is estimated in the 3-5% range

Source: "Intel Xeon Product Family: Performance Brief"

Intel Xeon Phi Coprocessors and the MIC Architecture



- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GB of DDR3 RAM
- ≤12 cores/socket ≈3 GHz
- 2-way hyper-threading
- 256-bit AVX vectors

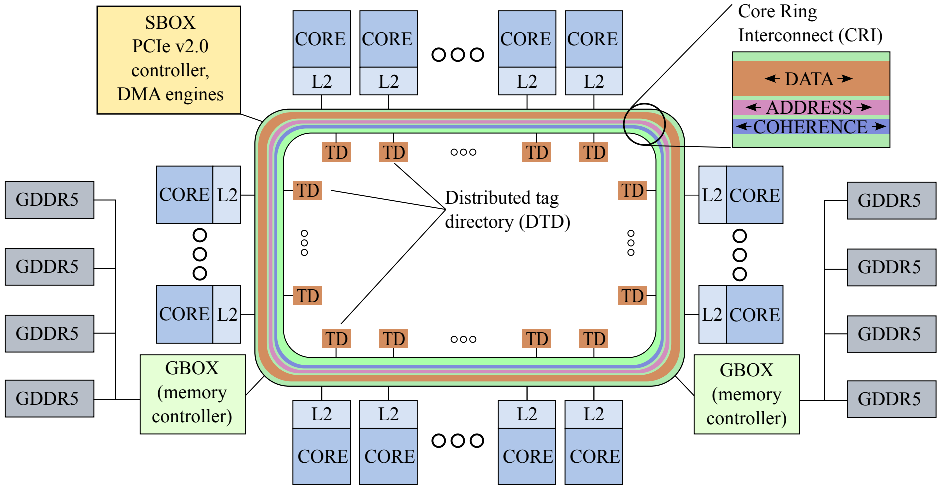
- C/C++/Fortran; OpenMP/MPI
- Special Linux μOS distribution
- 6–16 GB cached GDDR5 RAM
- 57 to 61 cores at ≈1 GHz
- 4 hardware threads per core
- 512-bit IMCI vectors

Linux μOS on Intel Xeon Phi coprocessors (part of MPSS)

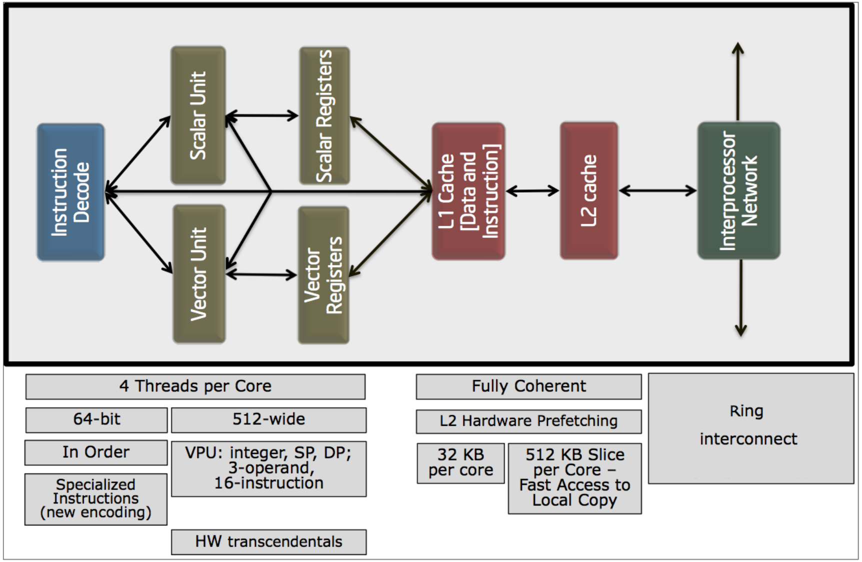
```
user@host% lspci | grep -i "co-processor"
06:00.0 Co-processor: Intel Corporation Xeon Phi coprocessor 3120 series (rev 20)
82:00.0 Co-processor: Intel Corporation Xeon Phi coprocessor 3120 series (rev 20)
user@host% sudo service mpss status
mpss is running
user@host% cat /etc/hosts | grep mic
172.31.1.1  host-mic0 mic0
172.31.2.1  host-mic1 mic1
user@host% ssh mic0
user@mic0% cat /proc/cpuinfo | grep proc | tail -n 3
processor : 237
processor : 238
processor : 239
user@mic0% ls /
amplxe   dev   home  lib64    oldroot  proc  sbin     sys  usr
bin      etc   lib   linuxrc  opt      root  sep3.10  tmp  var
```

Details of the MIC Architecture

Die Organization Diagram



Core Topology



SIMD Operations

SIMD — Single Instruction Multiple Data

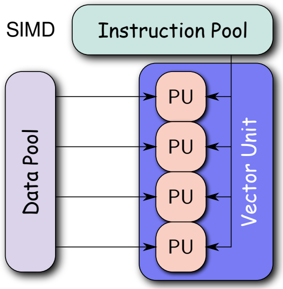
Scalar Loop

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

SIMD Loop

```
for (i = 0; i < n; i += 16)
  A[i:(i+16)] = A[i:(i+16)] + B[i:(i+16)];
```

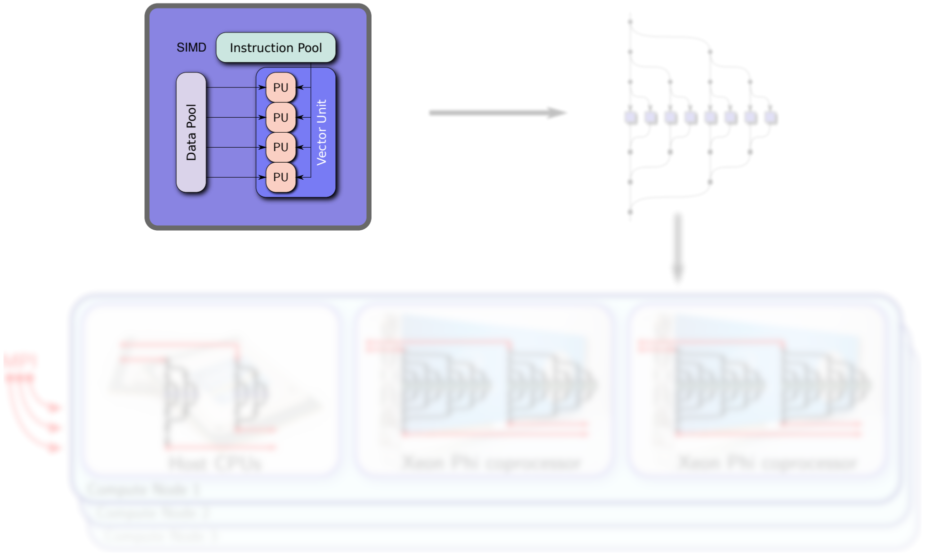
Each SIMD addition operator acts on 16 numbers at a time.



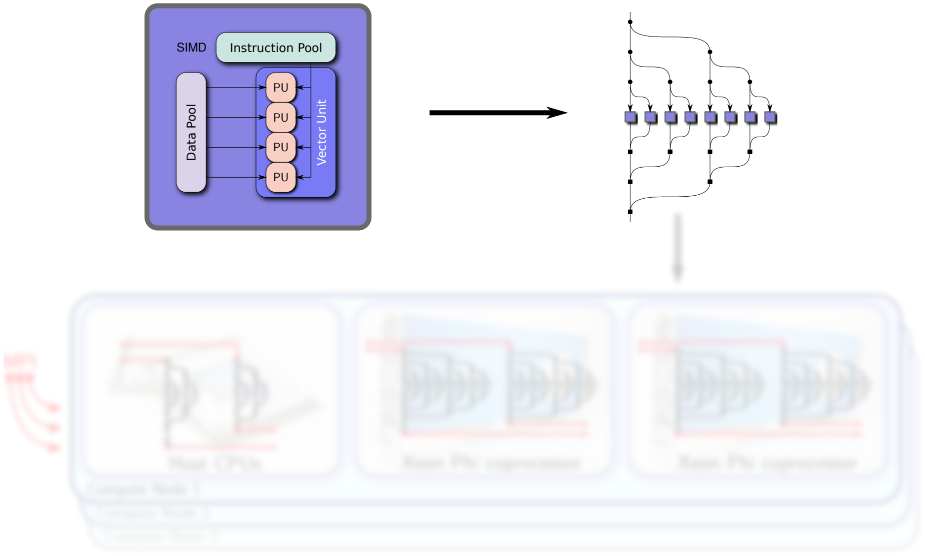
Instruction Sets in Intel Architectures

Instruction Set	Year and Intel Processor	Vector registers	Packed Data Types
MMX	1997, Pentium	64-bit	8-, 16- and 32-bit integers
SSE	1999, Pentium III	128-bit	32-bit single precision FP
SSE2	2001, Pentium 4	128-bit	8 to 64-bit integers; SP & DP FP
SSE3–SSE4.2	2004 – 2009	128-bit	(additional instructions)
AVX	2011, Sandy Bridge	256-bit	single and double precision FP
AVX2	2013, Haswell	256-bit	integers, additional instructions
IMCI	2012, Knights Corner Intel Xeon Phi coproc.	512-bit	32- and 64-bit integers; single & double precision FP
AVX-512	(future) Knights Landing	512-bit	32- and 64-bit integers; single & double precision FP

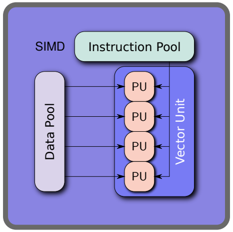
Three Layers of Parallelism

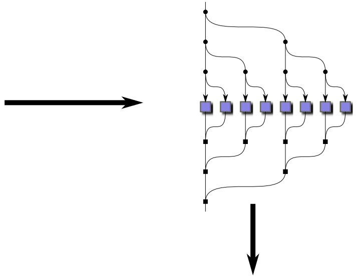


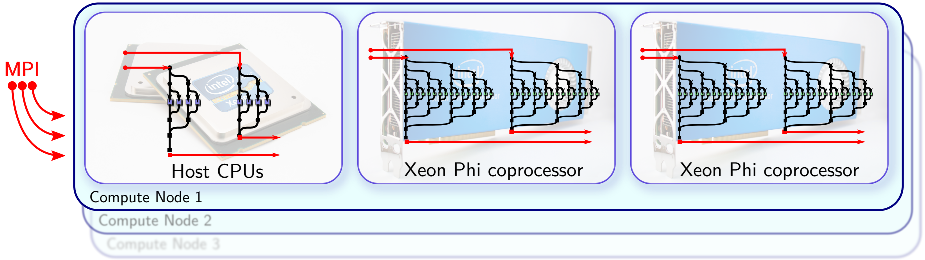
Three Layers of Parallelism



Three Layers of Parallelism







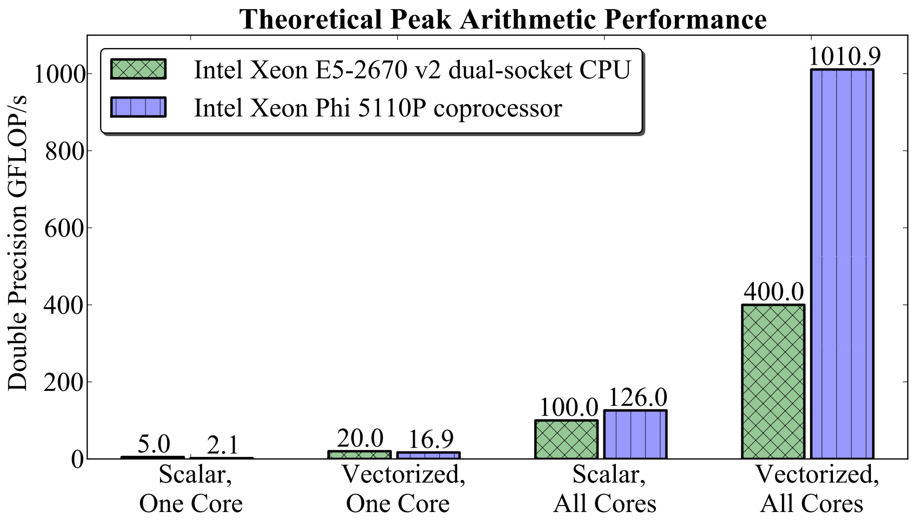
Examples of Solutions with the Intel MIC Architecture

Colfax's CXP7450 workstation with two Intel Xeon Phi coprocessors

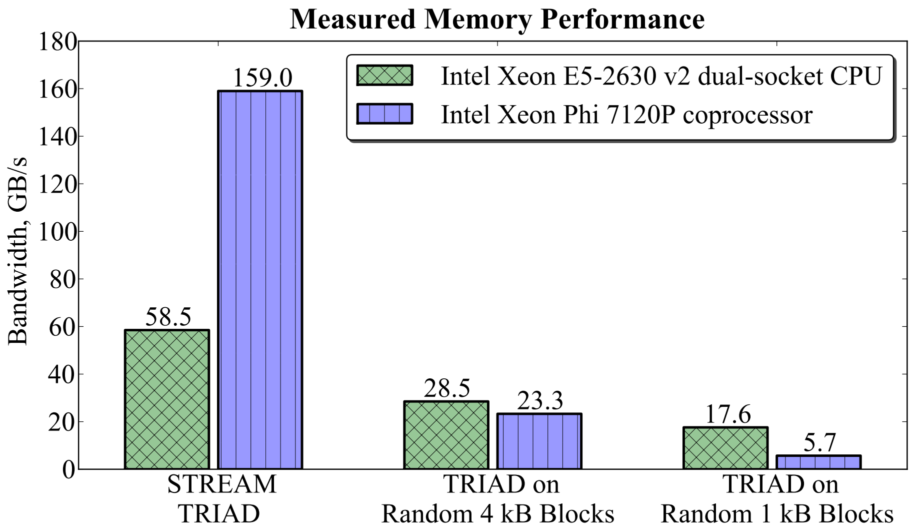
Colfax's CXP9000 server with eight Intel Xeon Phi coprocessors

`xeonphi.com`

Data and Task Parallelism (Vectors and Cores)



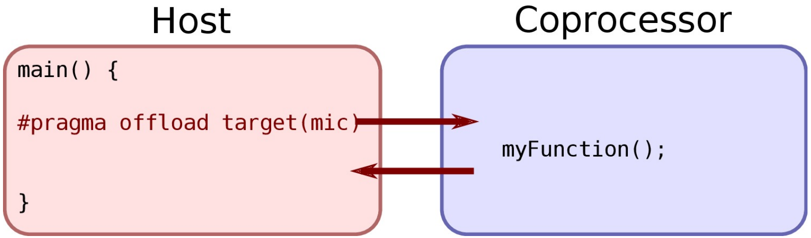
Memory Access Pattern



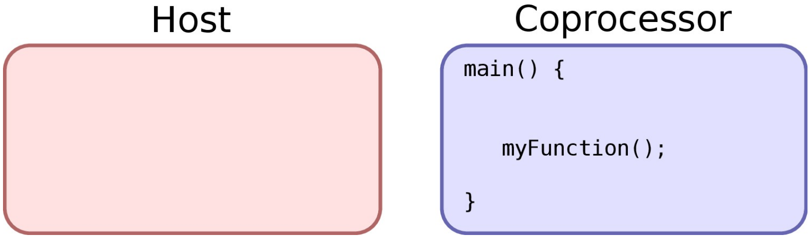
§2. Programming Models and Application Porting

Offload and Native modes

- Explicit offload mode:



- Native mode:



Native Programming

Native Execution

“Hello World” application:

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

Compile and run on host:

```
user@host% icc hello.c
user@host% ./a.out
Hello world! I have 32 logical cores.
user@host%
```

Native Execution

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

```
user@host% icc hello.c -mmic
user@host% scp a.out mic0:~/
a.out 100% 10KB 10.4KB/s 00:00
user@host% ssh mic0
user@mic0% pwd
/home/user
user@mic0% ls
a.out
user@mic0% ./a.out
Hello world! I have 240 logical cores.
user@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)

Porting User Applications for Native Execution

Simple CPU applications can be compiled for native execution on Xeon Phi coprocessors by supplying the flag “-mmic” to the Intel compiler:

```
user@host% icpc -c myobject1.cc -mmic
user@host% icpc -c myobject2.cc -mmic
user@host% icpc -o myapplication myobject1.o myobject2.o -mmic
```

Same for coprocessor-only MPI applications:

```
user@host% mpiicpc -c myobject1.cc -mmic
user@host% mpiicpc -c myobject2.cc -mmic
user@host% mpiicpc -o myapplication myobject1.o myobject2.o -mmic
```

Native Applications with Autotools

- Use the Intel compiler with flag `-mmic`
- Eliminate assembly and unncecessary dependencies
- Use `--host=x86_64` to avoid “program does not run” errors

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

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

Running Native MPI Applications on Coprocessors

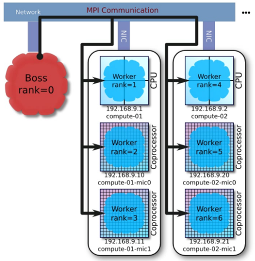
```
user@host% export I_MPI_MIC=1
user@host% mpiicpc -mmic -o HelloMPI.MIC HelloMPI.c
user@host% scp HelloMPI.MIC mic0:~/
user@host% mpirun -host mic0 -np 2 ~/HelloMPI.MIC
Hello World from rank 1 running on host-mic0!
Hello World from rank 0 running on host-mic0!
MPI World size = 2 processes
```

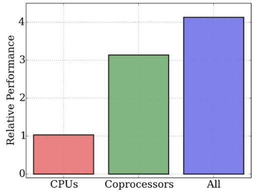
- Enable the MIC architecture in Intel MPI: `I_MPI_MIC=1`
- Copy or NFS-share MPI library & executables with coprocessor
- Use `mpiicpc` with `-mmic` to compile
- Launch as if `mic0` is a remote host

Native Heterogeneous Clustering in Action

Heterogeneous Clustering with Homogeneous Code: Asian Option Pricing

- Monte Carlo method
- MPI + OpenMP + automatic vectorization
- The same C code for clusters of
  a) CPUs
  b) Coprocessors
  c) CPUs+Coprocessors (heterogeneous)
- **No code modification** to run on the Intel MIC architecture
- No platform-specific tunning
- Bridged network configuration





More information in paper at `xeonphi.com/papers/heterogeneous`

Offload Programming

Explicit Offload Programming Model

“Hello World” in the explicit offload model:

```c
#include <stdio.h>

int main(int argc, char * argv[] ) {

    printf("Hello World from host!\n");

#pragma offload target(mic)
    {
        printf("Hello World from coprocessor!\n"); fflush(0);
    }

    printf("Bye\n");
}
```

Application launches and runs on the host, but some parts of code and data are moved (“offloaded”) the coprocessor.

Compiling and Running an Offload Application

```
user@host% icpc hello_offload.cpp -o hello_offload
user@host% ./hello_offload
Hello World from host!
Bye
Hello World from coprocessor!
```

- Code inside of `#pragma offload` is offloaded automatically
- Console output on Intel Xeon Phi coprocessor is buffered and mirrored to the host console
- If coprocessor is not installed, code inside `#pragma offload` may fall back to the host system

Offloading Functions and Data

```
int* __attribute__((target(mic))) data;

__attribute__((target(mic))) void MyFunction(int* foo) {
    // ... implement function as usual
}

int main(int argc, char * argv[] ) {
    // ...
#pragma offload target(mic) inout(data : length(N))
    {
        MyFunction(data);
    }
}
```

- Functions and data used on coprocessor must be marked with the specifier `__attribute__((target(mic)))`

Data Marshalling for Dynamically Allocated Data

```
double *p1=(double*)malloc(sizeof(double)*N);
// ...

#pragma offload target(mic)                                   \
        in    (p1 : length(N) alloc_if(1) free_if(0)) \
        out   (p2 : length(N) align(4096))               \
        inout(p3[0:N/2]   : into(p4[N/2:N/2]))
  {
    // ...
  }
```

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

Multiple Coprocessors and Fallback to Host

```
const int nDevices = _Offload_number_of_devices();
#pragma omp parallel num_threads(nDevices) if(nDevices>0)
  {
    const int iDevice = omp_get_thread_num();
#pragma offload target(mic: iDevice) if(nDevices>0)
      {
        MyFunction(/*...*/);
      }
  }
```

- Up to 8 coprocessors, up to 32 host threads
- All offloads start simultaneously and block the respective thread

Asynchronous Offload

- By default, `#pragma offload` blocks until offload completes
- Use clause "`signal`" with any pointer to avoid blocking
- Use `#pragma offload_wait` to block where needed

```
char* offload0;
#pragma offload target(mic:0) signal(offload0) in(data : length(N))
 { /* ... will not block code execution because of clause "signal" */ }

DoSomethingElse();

/* Now block until offload signalled by pointer "offload0" completes */
#pragma offload_wait target(mic:0) wait(offload0)
```

- Used to overlap communication with computation

Alternative: Virtual-shared Memory Model

```
_Cilk_shared int arr[N]; // This is a virtual-shared array

_Cilk_shared void Compute() { // This function may be offloaded
    // ... function uses array arr[]
}

int main() {
    // arr[] can be initialized on the host
    _Cilk_offload Compute(); // and used on coprocessor
    // and the values are returned to the host
}
```

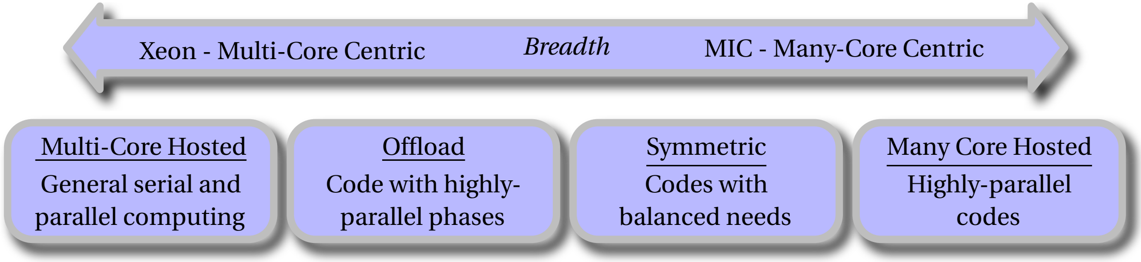
- Alternative to Explicit Offload
- Data synced from host to coprocessor before the start of offload
- Data synced from coprocessor to host at the end of offload

§3. Heterogeneous Computing with the MIC Architecture

Inter-Operation of Offload, MPI and OpenMP

Teaming Xeon and Xeon Phi Coprocessors

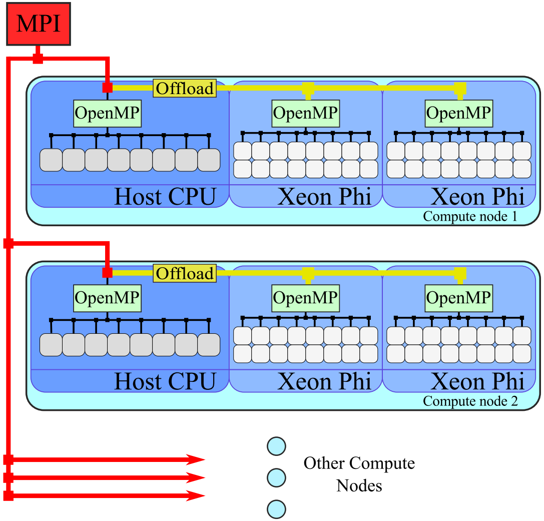
Programming models allow a range of CPU+MIC coupling modes


Xeon - Multi-Core Centric
Breadth
MIC - Many-Core Centric
Multi-Core Hosted
General serial and parallel computing
Offload
Code with highly-parallel phases
Symmetric
Codes with balanced needs
Many Core Hosted
Highly-parallel codes

Heterogeneous Distributed Computing with Xeon Phi

Option 1: MPI+OpenMP with Offload.

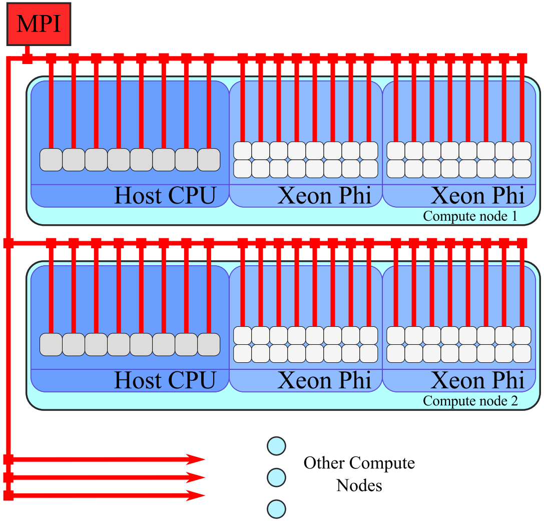
- MPI processes are multi-threaded with OpenMP.
- MPI runs only on CPUs.
- MPI processes offload to coprocessor(s).
- OpenMP in offload regions.



Heterogeneous Distributed Computing with Xeon Phi

Option 2: Symmetric pure MPI (native mode).

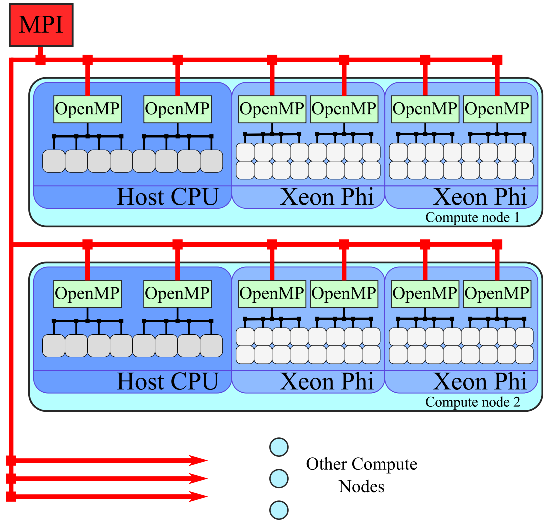
- MPI processes on hosts
- Native MPI processes on the coprocessor.
- No OpenMP.
- `xeonphi.com/papers/p2p`



Heterogeneous Distributed Computing with Xeon Phi

Option 3: Symmetric hybrid MPI+OpenMP.

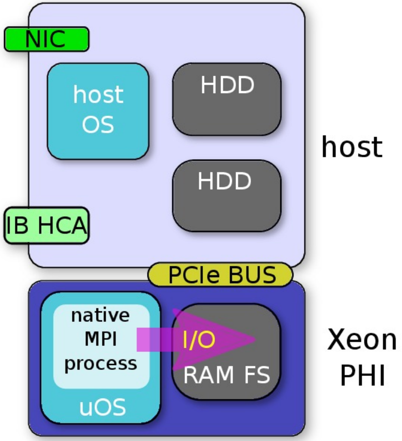
- MPI processes on hosts
- Native MPI processes on the coprocessor.
- Multi-threading with OpenMP.
- `xeonphi.com/papers/p2p`



File I/O in MPI Applications on Coprocessors

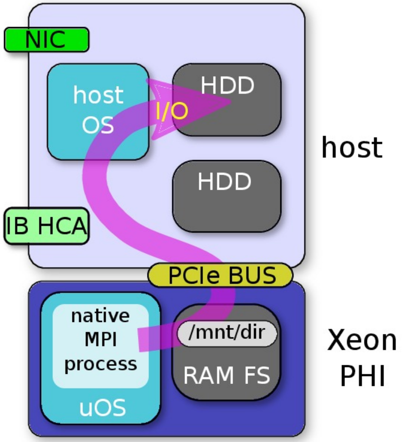
RAM Filesystem

- Files are stored in the coprocessor RAM
- Does not survive MPSS restart or host reboot
- Fastest method
- Good for local pre-staged input or runtime scratch data



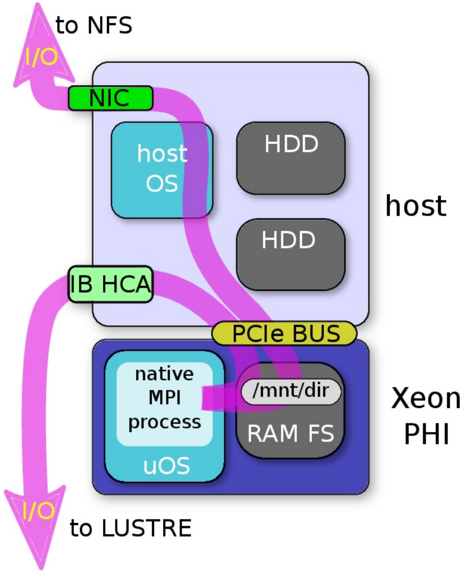
Virtio Transfer to Local Host Drives

- Files are stored on a physical or virtual drive on the host
- Written data persistent across reboots
- Fast method
- Cannot share a drive between coprocessors
- Good for distributed checkpointing



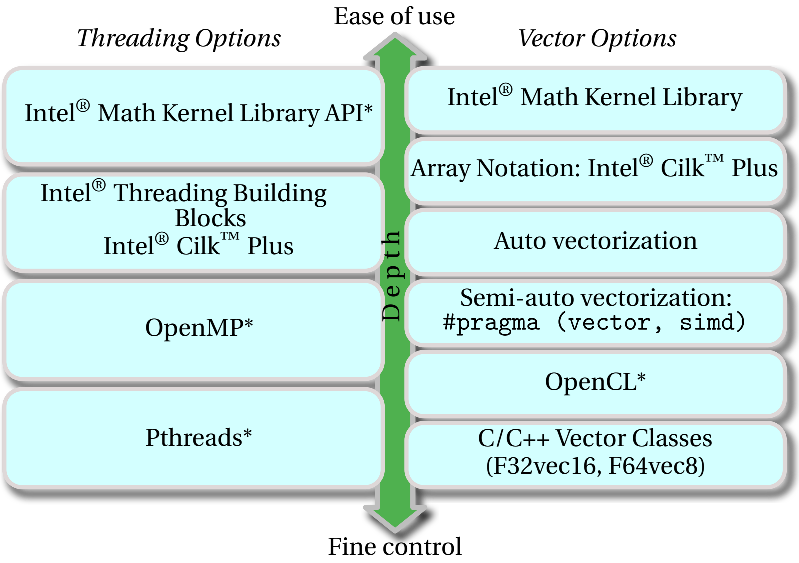
Network Storage

- Files are stored on a remote file server
- Can share a mount point across the cluster
- Lustre has scalable performance
- NFS is slow but easy to set up
- More information: `xeonphi.com/papers/io`



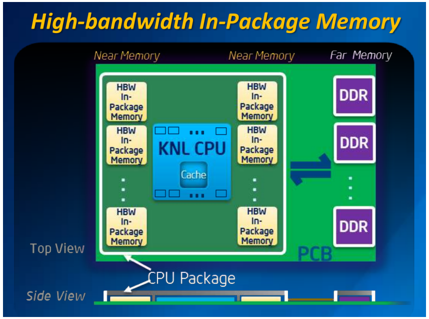
§4. Future-Proofing and Knights Landing

Future-Proofing: Reliance on Compiler and Libraries



Next Generation MIC: Knights Landing (KNL)

- 2nd generation MIC product: code name Knights Landing (KNL)
- Intel's 14 nm manufacturing process
- A processor (running the OS) or a coprocessor (PCIe device)
- On-package high-bandwidth memory w/flexible memory models: flat, cache, & hybrid
- Intel Advanced Vector Extensions AVX-512 (public)

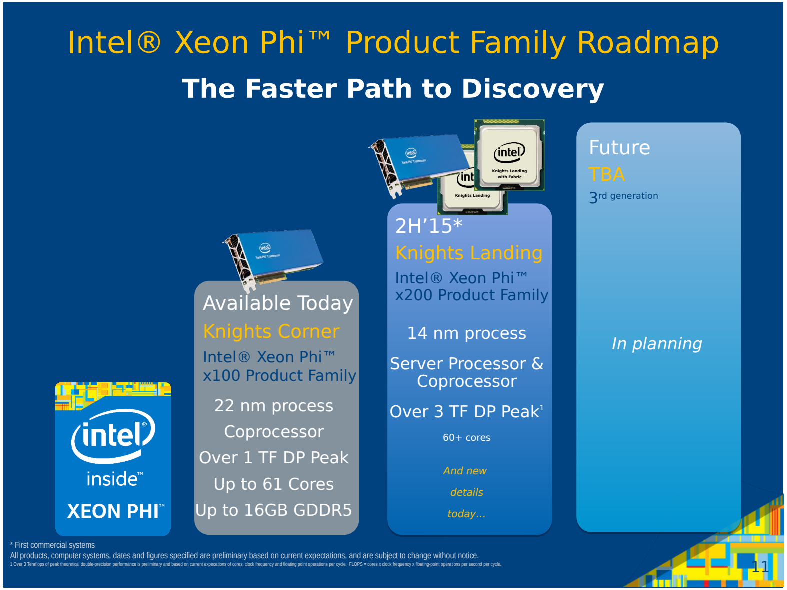


Source: *Intel Newsroom*

Getting Ready for the Future

- Porting HPC applications to today's MIC architecture makes them ready for future architectures, such as KNL
- Xeon, KNC and KNL are not binary compatible, therefore assembly-level tuning will not scale forward.
- Reliance on compiler optimization and using optimized libraries (such as Intel MKL) ensures future-readiness.

Source: Intel Newsroom



Intel® Xeon Phi™ Product Family Roadmap

The Faster Path to Discovery

Available Today
Knights Corner
Intel® Xeon Phi™
x100 Product Family
22 nm process
Coprocessor
Over 1 TF DP Peak
Up to 61 Cores
Up to 16GB GDDR5

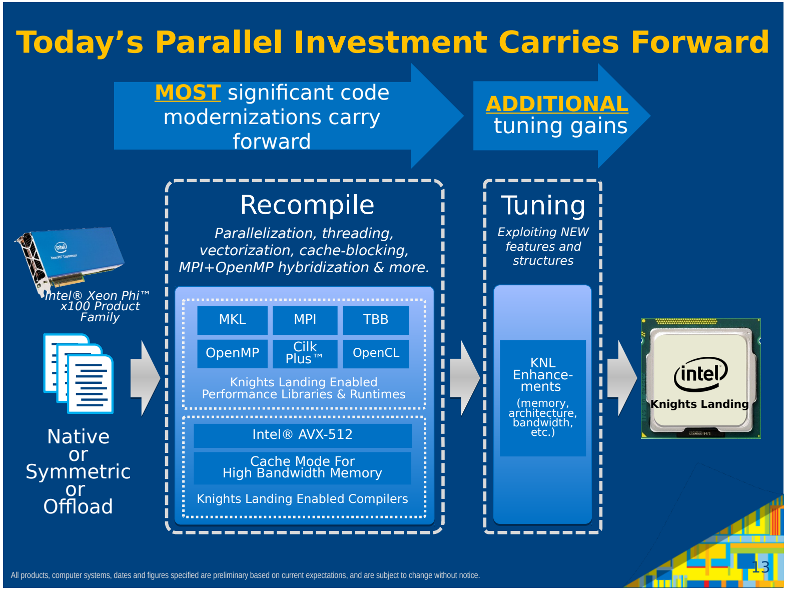
2H'15*
Knights Landing
Intel® Xeon Phi™
x200 Product Family
14 nm process
Server Processor & Coprocessor
Over 3 TF DP Peak[1]
60+ cores
And new details today…

Future
TBA
3rd generation
In planning

* First commercial systems
All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.
1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expecations of cores, clock frequency and floating point operations per cycle. FLOPS = cores x clock frequency x floating-point operations per second per cycle.

11

Source: https://www.brighttalk.com/webcast/10773/116329



Today's Parallel Investment Carries Forward

MOST significant code modernizations carry forward

ADDITIONAL tuning gains

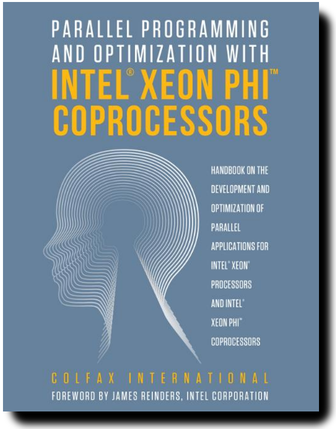
Intel® Xeon Phi™ x100 Product Family
Native or Symmetric or Offload
Recompile
Parallelization, threading, vectorization, cache-blocking, MPI+OpenMP hybridization & more.
MKL
MPI
TBB
OpenMP
Cilk Plus™
OpenCL
Knights Landing Enabled Performance Libraries & Runtimes
Intel® AVX-512
Cache Mode For High Bandwidth Memory
Knights Landing Enabled Compilers
Tuning
Exploiting NEW features and structures
KNL Enhance-ments
(memory, architecture, bandwidth, etc.)
Knights Landing
13

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Source: https://www.brighttalk.com/webcast/10773/116329

Colfax Developer Training

Colfax runs one-day and four-day trainings for organizations and individuals on parallel programming with Intel Xeon Phi coprocessors.


PARALLEL PROGRAMMING AND OPTIMIZATION WITH
INTEL® XEON PHI™ COPROCESSORS
HANDBOOK ON THE DEVELOPMENT AND OPTIMIZATION OF PARALLEL APPLICATIONS FOR INTEL® XEON® PROCESSORS AND INTEL® XEON PHI™ COPROCESSORS
COLFAX INTERNATIONAL
FOREWORD BY JAMES REINDERS, INTEL CORPORATION

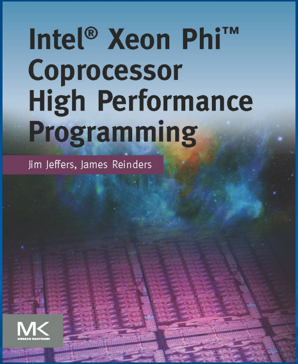
Information: xeonphi.com/training

Additional Reading

It all comes down to PARALLEL PROGRAMMING ! (applicable to processors and Intel® Xeon Phi™ coprocessors both)

Forward, Preface
Chapters:

Learn more about this book:
lotsofcores.com


Intel® Xeon Phi™ Coprocessor High Performance Programming
Jim Jeffers, James Reinders
MK

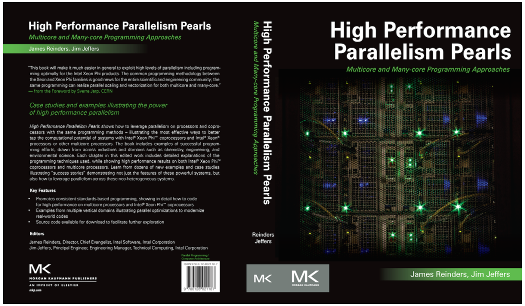
Available since February 2013.

This book belongs on the bookshelf of every HPC professional. Not only does it successfully and accessibly teach us how to use and obtain high performance on the Intel MIC architecture, it is about much more than that. It takes us back to the universal fundamentals of high-performance computing including how to think and reason about the performance of algorithms mapped to modern architectures, and it puts into your hands powerful tools that will be useful for years to come.
—Robert J. Harrison
Institute for Advanced Computational Science, Stony Brook University

Intel® Xeon Phi™ Coprocessor High Performance Programming, Jim Jeffers, James Reinders, (c) 2013, publisher: Morgan Kaufmann

Additional Reading

Available November 17, 2014



High Performance Parallelism Pearls

Multicore and Many-core Programming Approaches

James Reinders, Jim Jeffers

"This book will make it much easier in general to exploit high levels of parallelism including programming optimally for the Intel Xeon Phi products. The common programming methodology between the Xeon and Xeon Phi families is good news for the entire scientific and engineering community; the same programming can realize parallel scaling and vectorization for both multicore and many-core."
— from the Foreword by Sverre Jarp, CERN

Case studies and examples illustrating the power of high performance parallelism

High Performance Parallelism Pearls shows how to leverage parallelism on processors and coprocessors with the same programming methods – illustrating the most effective ways to better tap the computational potential of systems with Intel® Xeon Phi™ coprocessors and Intel® Xeon® processors or other multicore processors. The book includes examples of successful programming efforts, drawn from across industries and domains such as chemistry, engineering, and environmental science. Each chapter in this edited work includes detailed explanations of the programming techniques used, while showing high performance results on both Intel® Xeon Phi™ coprocessors and multicore processors. Learn from dozens of new examples and case studies illustrating "success stories" demonstrating not just the features of these powerful systems, but also how to leverage parallelism across these neo-heterogeneous systems.

Key Features

- Promotes consistent standards-based programming, showing in detail how to code for high performance on multicore processors and Intel® Xeon Phi™ coprocessors
- Examples from multiple vertical domains illustrating parallel optimizations to modernize real-world codes
- Source code available for download to facilitate further exploration

Editors

James Reinders, Director, Chief Evangelist, Intel Software, Intel Corporation
Jim Jeffers, Principal Engineer, Engineering Manager, Technical Computing, Intel Corporation

MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER
mkp.com

Parallel Programming/
Computer Architecture

ISBN 978-0-12-802118-7

High Performance Parallelism Pearls
Multicore and Many-core Programming Approaches
Reinders
Jeffers

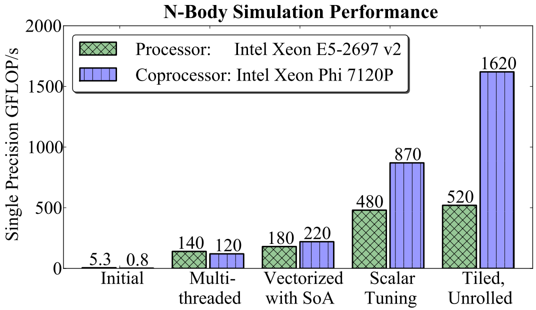
High Performance Parallelism Pearls
Multicore and Many-core Programming Approaches
James Reinders, Jim Jeffers

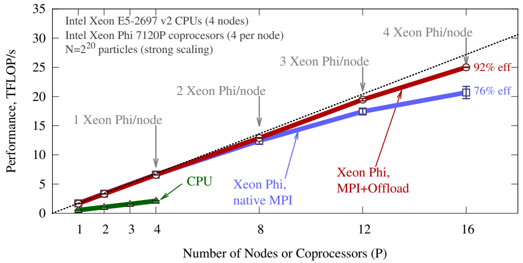
www.lotsofcores.com

- 28 chapters
- 69 expert contributors
- Numerous "Real World" Code "Recipes" and examples using OpenMP, MPI, TBB, C, C++, OpenCL, Fortran.
- Successful techniques, tips for vectorization, scalable parallel coding, load balancing, data structure and memory tuning, applicable to both processors and coprocessors!
- All figures, diagrams and code freely downloadable. (Nov'14)

Next Session Starting in 15 Minutes

- Example: N-body simulation
- Optimization essentials
- Node-level tuning
- Scalability in a cluster





Thank you for tuning in,
and
have a wonderful journey
to the Parallel World!

`http://research.colfaxinternational.com/`

P.S.: We are hiring! `xeonphi.com/jobs`

§5. Resources/Backup Slides

Reference Guides

- Intel C++ Compiler 14.0 User and Reference Guide
- Intel VTune Amplifier XE User's Guide
- Intel Trace Analyzer and Collector Reference Gude
- Intel MPI Library for Linux* OS Reference Manual
- Intel Math Kernel Library Reference Manual
- Intel Software Documentation Library
- MPI Routines on the ANL Web Site
- OpenMP Specifications

Intel's Top 10 List

1. Download programming books: "Intel Xeon Phi Coprocessor High Performance Programming" by Jeffers & Reinders, and "Parallel Programming and Optimization with Intel Xeon Phi Coprocessors" by Colfax.
2. Watch the parallel programming webinar
3. Bookmark and browse the mic-developer website
4. Bookmark and browse the two developer support forums: "Intel MIC Architecture" and "Threading on Intel Parallel Architectures".
5. Consult the "Quick Start" guide to prepare your system for first use, learn about tools, and get C/C++ and Fortran-based programs up and running

Intel's Top 10 resources list: `http://xeonphi.com/top10`

Intel's Top 10 List (continued)

6. Try your hand at the beginning lab exercises
7. Try your hand at the beginner/intermediate real world app exercises
8. Browse the case studies webpage to view examples from many segments
9. Begin optimizing your application(s); consult your programming books, the ISA reference manual, and the support forums for assistance.
10. Hone your skills by watching more advanced video workshops

Intel's Top 10 resources list: `http://xeonphi.com/top10`