Parallel Programming and Optimization with Intel Xeon Phi Coprocessors

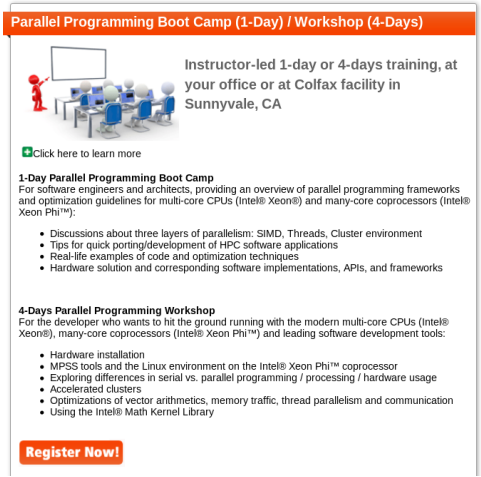
Colfax Developer Boot Camp

Vadim Karpusenko and Andrey Vladimirov
Colfax International

May 2014, Rev. 11

About This Document

This document represents the materials of a one-day training "Parallel Programming Boot Camp" developed and run by Colfax International.



http://www.colfax-intl.com/nd/xeonphi/training.aspx

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.

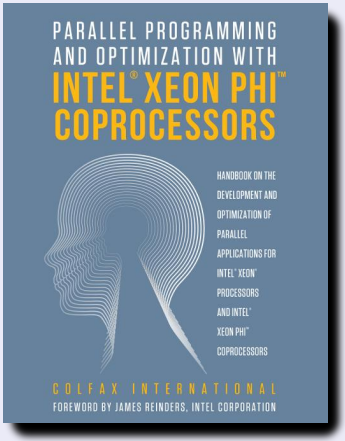
Supplementary Materials

Supplementary Materials: Textbook

ISBN: 978-0-9885234-1-8 (520 pages)

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



http://www.colfax-intl.com/nd/xeonphi/book.aspx

Additional Reading

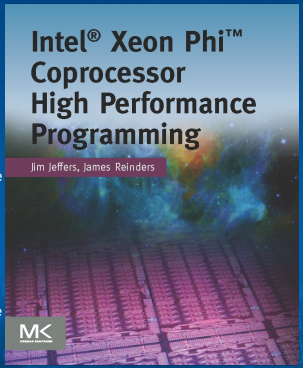
It all comes down to PARALLEL PROGRAMMING ! (applicable to processors and Intel® Xeon Phi™ coprocessors both)

Forward, Preface
Chapters:

1. Introduction
2. High Performance Closed Track Test Drive!
3. A Friendly Country Road Race
4. Driving Around Town: Optimizing A Real-World Code Example
5. Lots of Data (Vectors)
6. Lots of Tasks (not Threads)
7. Offload
8. Coprocessor Architecture
9. Coprocessor System Software
10. Linux on the Coprocessor
11. Math Library
12. MPI
13. Profiling and Timing
14. Summary

Glossary, Index

Learn more about this book:
lotsofcores.com



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

List of Topics

List of Topics

1. Introduction
   - Intel Xeon Phi Architecture from the Programmer's Perspective
   - Software Tools for Intel Xeon Phi Coprocessors
   - Will Application X benefit from the MIC architecture?

2. Programming Models for Intel Xeon Phi Applications
   - Native Applications for Coprocessors and MPI
   - Offload Programming Models
   - Using Multiple Coprocessors
   - MPI Applications and Heterogeneous Clustering

List of Topics

3. Porting Applications to the MIC Architecture
   - Future-Proofing: Reliance on Compiler and Libraries
   - Choosing the Programming Model
   - Cross-Compilation of User Applications
   - Performance Expectations

4. Parallel Scalability on Intel Architectures
   - Vectorization (Single Instruction Multiple Data, SIMD, Parallelism)
   - Multi-threading: OpenMP, Intel Cilk Plus
   - Task Parallelism in Distributed Memory, MPI

List of Topics

5. Optimization for the Intel Xeon Product Family
   - Optimization Checklist
   - Finding Bottlenecks with Intel VTune Amplifier
   - MPI Diagnostics Using Intel Trace Analyzer and Collector
   - Intel Math Kernel Library (MKL)
   - Scalar Optimization Considerations
   - Automatic Vectorization and Data Structures
   - Optimization of Thread Parallelism

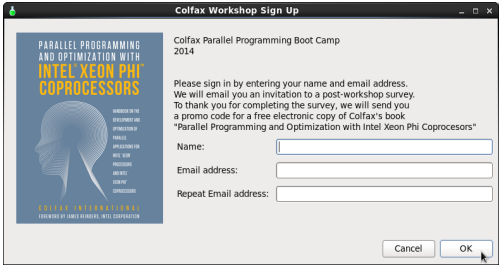
List of Topics

6. Advanced Optimization for the MIC Architecture
   - Memory Access and Cache Utilization
   - Data Persistence and PCIe Traffic
   - MPI Applications on Clusters with Coprocessors

7. Conclusion
   - Course Recap
   - Additional Resources: Reading, Guides, Support

Sign In

Please sign in during any coffee break to receive an invitation to a survey. Completing the survey earns you a free electronic copy of our book “Parallel Programming and Optimization with Intel Xeon Phi Coprocessors”.

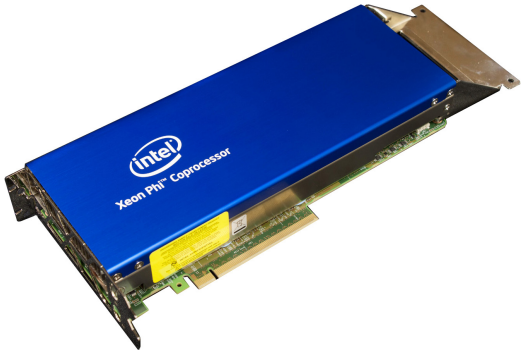


§1. Introduction to the Intel Many Integrated Core (MIC) Architecture

MIC Architecture from the Programmer's Perspective

Intel Xeon Phi Coprocessors and the MIC Architecture

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

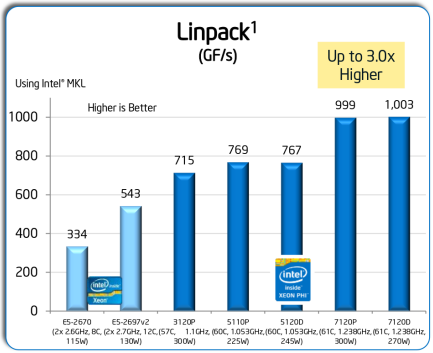


For highly parallel applications which reach the scaling limits on Intel Xeon processors

Xeon Family Product Performance

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

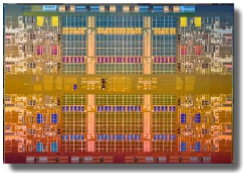
- Better performance per system & performance per watt for parallel applications
- Same programming methods, same development tools.



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

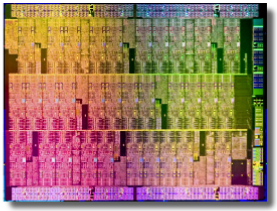
Source: "Intel Xeon Product Family: Performance Brief"

Intel Xeon Processors and the MIC Architecture



Multi-core Intel Xeon processor

- 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



Many-core Intel Xeon Phi coprocessor

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

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

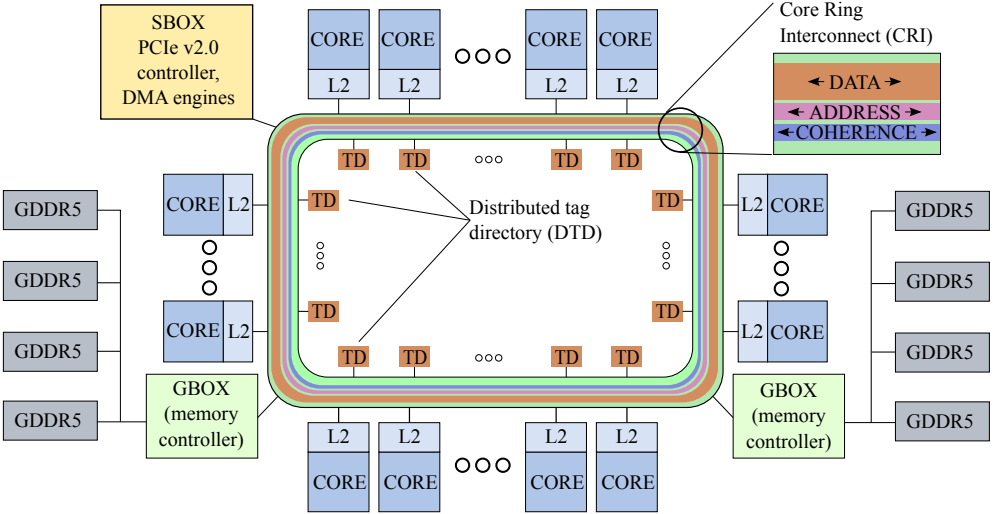
COLFAX
Customized Solutions

N-body simulation on...

Two Intel® Xeon® CPUs	One Intel® Xeon Phi™ coprocessor	Two Intel® Xeon Phi™ coprocessors
		

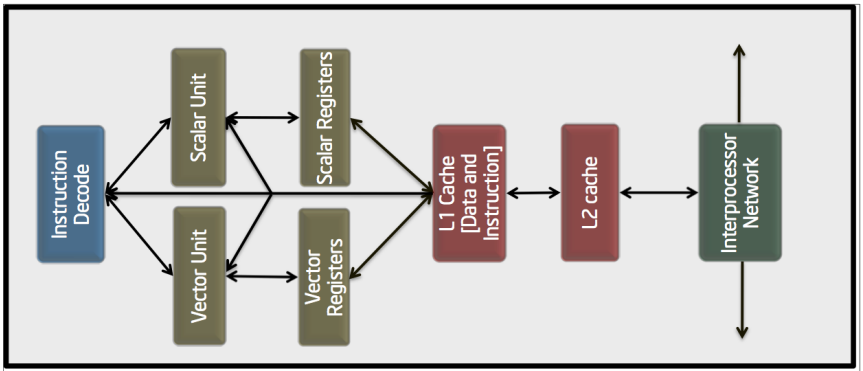
Paper: research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx
Demo: http://www.youtube.com/watch?v=KxaSEcmkGTo

Microarchitecture



Source: “Intel Xeon Phi Coprocessor - the Architecture“

Core Topology



Cache Structure

The caches are 8-way associative, fully coherent with the LRU (Least Recently Used) replacement policy

Cache line size	64B
L1 size	32KB data, 32KB code
L1 latency	1 cycle
L2 size	512KB
L2 ways	8
L2 latency	11 cycles
Memory → L2 prefetching	hardware and software
L2 → L1 prefetching	software only
Translation Lookaside Buffer(TLB) coverage options (L1, data)	64 pages of size 4KB (256KB coverage) 8 pages of size 2MB (16MB coverage)

Features of the IMCI Instruction Set

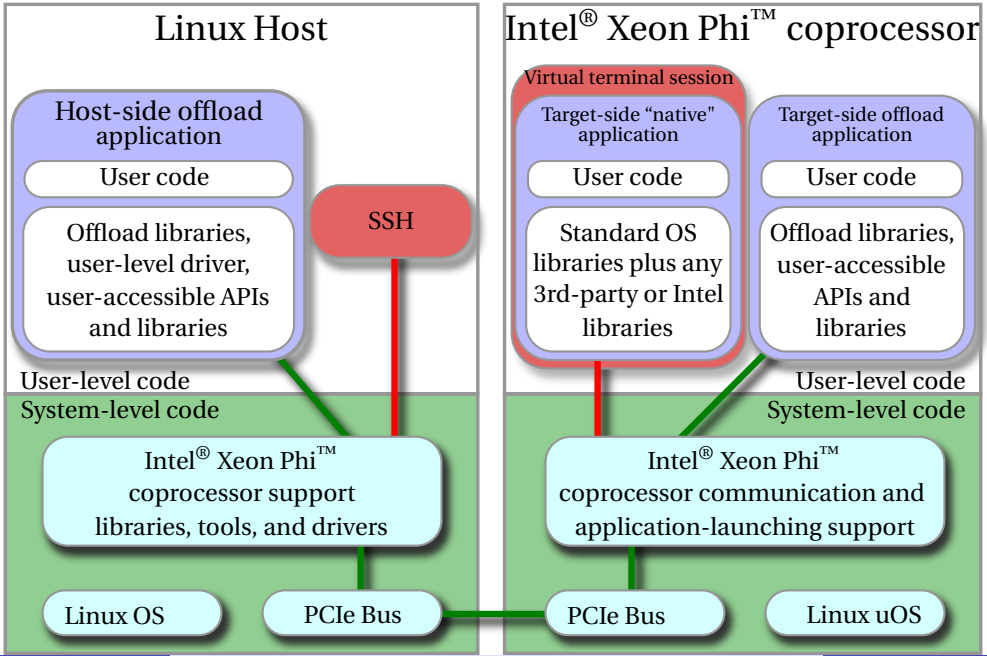
Intel IMCI is the instruction set supported by Intel Xeon Phi copr.

- 512-bit wide registers
  - can pack up to eight 64-bit elements (`long int`, `double`)
  - up to sixteen 32-bit elements (`int`, `float`)
- Arithmetic Instructions
  - Addition, subtraction and multiplication
  - Fused Multiply-Add instruction (FMA)
  - Division and reciprocal calculation;
  - Error function, inverse error function;
  - Exponential functions (natural, base 2 and base 10) and the power function.
  - Logarithms (natural, base 2 and base 10).
  - Square root, inverse square root, hypothenuse value and cubic root;
  - Trigonometric functions (`sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `asin`, `acos` ...);
  - Rounding functions

Features of the IMCI Instruction Set

- Initialization, Load and Store, Gather and Scatter
- Comparison
- Conversion and type cast
- Bitwise instructions: `NOT`, `AND`, `OR`, `XOR`, `XAND`
- Reduction and minimum/maximum instructions
- Vector mask instructions
- Scalar instructions
- Swizzle and permute

Interactions between Operating Systems



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

```
user@host% lspci | grep -i "co-processor"
06:00.0 Co-processor: Intel Corporation Device 2250 (rev 11)
82:00.0 Co-processor: Intel Corporation Device 2250 (rev 11)
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
```

Software Tools for Intel Xeon Phi Coprocessors

Execute MIC Applications (all free):

Drivers : Intel MIC Platform Software Stack (Intel MPSS) — mandatory — detect, boot and manage coprocessors

Libraries : Redistributable libraries — optional — run and distribute pre-built applications

OpenCL : Intel OpenCL SDK — optional

Monitoring MIC activity with `micsmc` (an MPSS tool)

MPSS Tools and Utilities

`micinfo` a system information query tool

`micsmc` a utility for monitoring and modifying the physical paramaters: temperature, power modes, core utilization, etc.

`micctrl` a comprehensive configuration tool for the Intel Xeon Phi coprocessor operating system

`miccheck` a set of diagnostic tests for the verification of the Intel Xeon Phi coprocessor configuration

`micrasd` a host daemon logger of hardware errors reported by Intel Xeon Phi coprocessors

`micflash` an Intel Xeon Phi flash memory agent

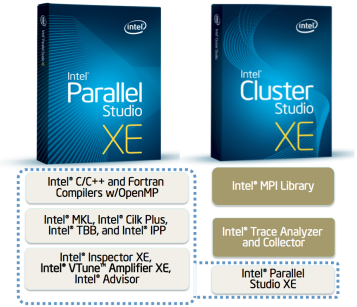
Build Xeon Phi & Xeon CPU Applications (all licensed):

Compilers : Intel C Compiler, Intel C++ Compiler, and Intel Fortran Compiler — mandatory

Optimization tools : Intel VTune Amplifier XE and Intel Trace Analyzer and Collector (ITAC) — highly recommended

Mathematics support : Intel Math Kernel Library (MKL) — highly recommended

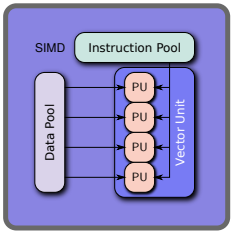
Development : Intel Inspector XE, Intel Advisor XE — optional



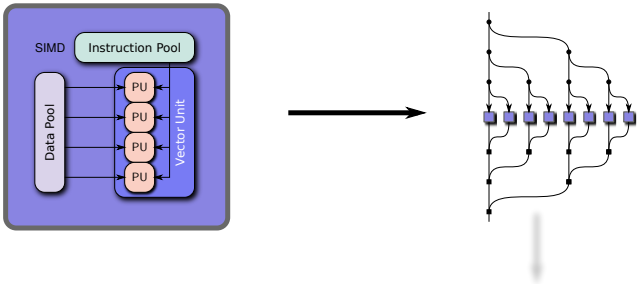
All-in-One Bundles, common for CPU and MIC

Will Application X Benefit from the MIC architecture?

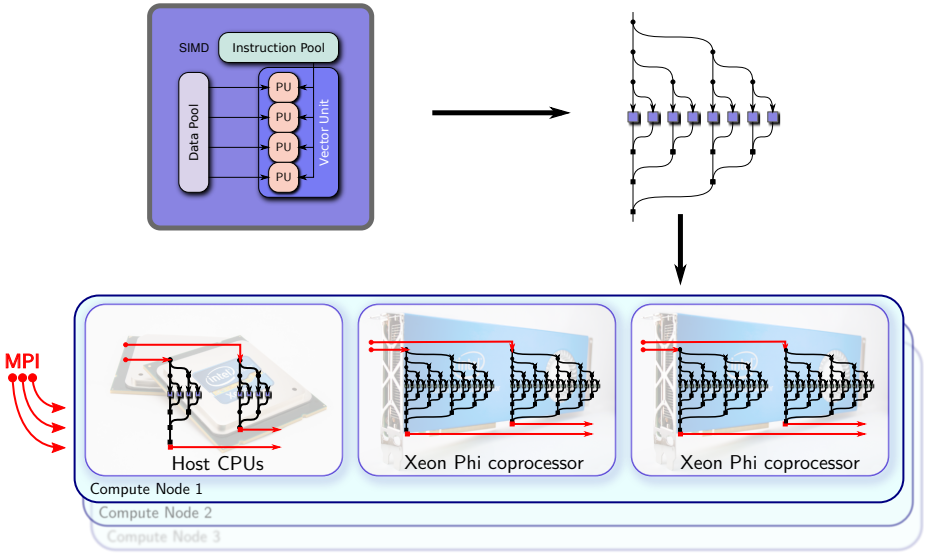
Three Layers of Parallelism



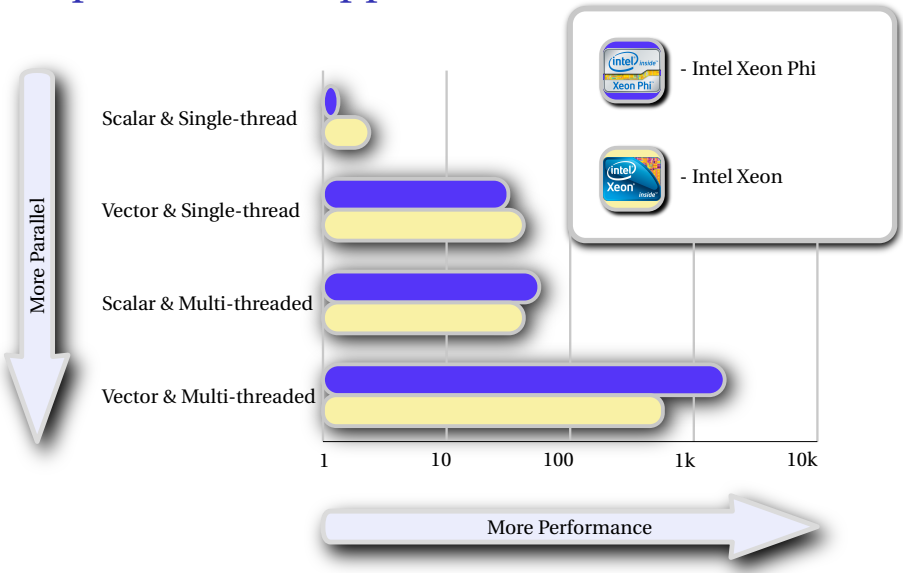
Three Layers of Parallelism



Three Layers of Parallelism



Compute-Bound Application Performance



One Size Does Not Fit All

An application must reach scalability limits on the CPU in order to benefit from the MIC architecture.

Use Xeon Phi if:

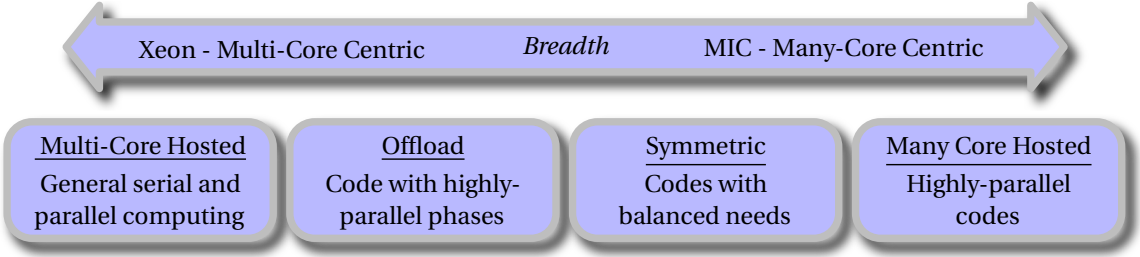
- Scales up to 100 threads
- Compute bound & vectorized, or bandwidth-bound

Use Xeon if:

- Serial or scales to $\lesssim$10 threads
- Unvectorized or latency-bound

Xeon + Xeon Phi Coprocessors = Xeon Family

Programming models allow a range of CPU+MIC coupling modes



§2. Programming Models for Intel Xeon Phi Applications

Native Execution

"Hello World" application:

```
1 #include <stdio.h>
2 #include <unistd.h>
3 int main(){
4   printf("Hello world! I have %ld logical cores.\n",
5   sysconf(_SC_NPROCESSORS_ONLN ));
6 }
```

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)

Native Applications for Coprocessors with MPI

"Hello World" in MPI:

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>
int main (int argc, char *argv[]) {
  int i, rank, size, namelen;
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name (name, &namelen);
  printf ("Hello World from rank %d running on %s!\n", rank, name);
  if (rank == 0) printf("MPI World size = %d processes\n", size);
  MPI_Finalize ();
}
```


Running MPI Applications on Host

```
user@host% source /opt/intel/impi/4.1.0/intel64/bin/mpivars.sh
user@host% export I_MPI_FABRICS=shm:tcp
user@host% mpiicpc -o HelloMPI.XEON HelloMPI.c
user@host% mpirun -host localhost -np 2 ./HelloMPI.XEON
Hello World from rank 1 running on host!
Hello World from rank 0 running on host!
MPI World size = 2 processes
```

- Set up MPI environment variables
- Use wrapper script `mpiicpc` to compile
- Use automated tool `mpirun` to launch

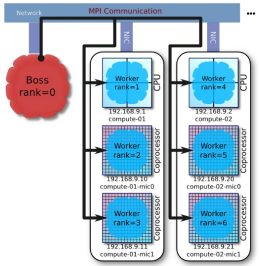
Running Native MPI Applications on Coprocessors

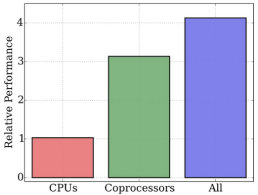
```
user@host% source /opt/intel/impi/4.1.0/intel64/bin/mpivars.sh
user@host% export I_MPI_MIC=1
user@host% export I_MPI_FABRICS=shm:tcp
user@host% mpiicpc -mmic -o HelloMPI.MIC HelloMPI.c
user@host% scp HelloMPI.MIC mic0:~/
user@host% mpirun -host mic0 -np 2 ~/HelloMPI.MIC
Hello World from rank 1 running 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

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





Paper: research.colfaxinternational.com/post/2013/10/17/Heterogeneous-Clustering.aspx
Demo: http://youtu.be/GffmChTcWf8

Offload Programming Models

Explicit Offload: Pragma-based approach

"Hello World" in the explicit offload model:

```
1 #include <stdio.h>
2 int main(int argc, char * argv[] ) {
3   printf("Hello World from host!\n");
4 #pragma offload target(mic)
5     {
6       printf("Hello World from coprocessor!\n"); fflush(0);
7     }
8   printf("Bye\n");
9 }
```

Application runs on the host, but some parts of code and date 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!
```

- No additional arguments if compiled with an Intel compiler
- Run application on host as a regular application
- 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` runs on the host system

Offloading Functions

```
1  __attribute__((target(mic))) void MyFunction() {
2    // ... implement function as usual
3  }
4
5  int main(int argc, char * argv[] ) {
6  #pragma offload target(mic)
7      {
8        MyFunction();
9      }
10 }
```

- Functions used on coprocessor must be marked with the specifier `__attribute__((target(mic)))`
- Compiler produces a host version and a coprocesor version of such functions (to enable fall-back to host)

Offloading Multiple Functions

```
1 #pragma offload_attribute(push, target(mic))
2 void MyFunctionOne() {
3   // ... implement function as usual
4 }
5
6 void MyFunctionTwo() {
7   // ... implement function as usual
8 }
9 #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

```
1 void MyFunction() {
2   const int N = 1000;
3   int data[N];
4 #pragma offload target(mic)
5     {
6       for (int i = 0; i < N; i++)
7         data[i] = 0;
8     }
```

- 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

Offloading Data: Global and Static Variables

```
1  int* __attribute__((target(mic))) data;
2
3  void MyFunction() {
4    static int __attribute__((target(mic))) N;
5    // ...
6  }
7
8  int main() {
9    // ...
10 }
```

- Global and static variables must be marked with the offload attribute
- `#pragma offload_attribute(push/pop)` may be used as well

Data Marshalling for Dynamically Allocated Data

```
1 double *p1=(double*)malloc(sizeof(double)*N);
2 double *p2=(double*)malloc(sizeof(double)*N);
3
4 #pragma offload target(mic) in(p1 : length(N)) out(p2 : length(N))
5   {
6     // ... perform operations on p1[] and p2[]
7   }
```

- `#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

Memory retention and data persistence on coprocessor

```
1 #pragma offload target(mic) in(p : length(N) alloc_if(1) free_if(0) )
2   { /* allocate memory for array p on coprocessor, do not deallocate */ }
3
4 #pragma offload target(mic) in(p : length(0) alloc_if(0) free_if(0) )
5   { /* re-use previously allocated memory on coprocessor */ }
6
7 #pragma offload target(mic) out(p : length(N) alloc_if(0) free_if(1) )
8   { /* re-use memory and deallocate at the end of offload */ }
```

- By default, memory on coprocessor is allocated before, deallocated after offload
- Specifiers `alloc_if` and `free_if` allow to avoid allocation/deallocation
- Can be combined with `length(0)` to avoid data transfer
- Why bother: data transfer across the PCIe bus is relatively slow (6 GB/s), and memory allocation on coprocessor is even slower (0.5 GB/s)

Precautions with persistent data

- Use explicit zero-based coprocessor number (e.g., `mic:0` as shown below)
- With multiple coprocessors, if target number is unspecified, any coprocessor can be used, which will result in runtime errors if persistent data cannot be found.

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

- Do not change the value of the host pointer to a persistent array: the runtime system finds the data on coprocessor using the host pointer value, not variable name.

Virtual-shared Memory Model

```
1  _Cilk_shared int arr[N]; // This is a virtual-shared array
2
3  _Cilk_shared void Compute() { // This function may be offloaded
4    // ... function uses array arr[]
5  }
6
7  int main() {
8    // arr[] can be initialized on the host
9    _Cilk_offload Compute(); // and used on coprocessor
10   // and the values are returned to the host
11 }
```

- 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

Virtual-shared Memory Model

```
1 int* _Cilk_shared data; // Pointer to a virtual-shared array
2
3 int main() {
4   // Working with pointer-based data is illustrated below:
5   data = (_Cilk_shared int*)_Offload_shared_malloc(N*sizeof(float));
6   _Offload_shared_free(data);
7 }
```

- Addresses of virtual-shared pointers identical on host and coprocessors
- Synchronized before and after offload, with page granularity

Target-Specific Code

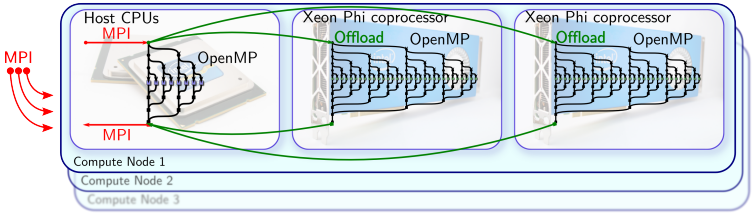
- During MIC architecture compilation, preprocessor macro `__MIC__` is defined.
- Allows to fine-tune application performance or output where necessary

```
1  void __attribute__((target(mic))) MyFunction() {
2  #ifdef __MIC__
3    printf("I am running on a coprocessor.\n");
4    const int tuningParameter = 16;
5  #else
6    printf("I am running on the host.\n");
7    const int tuningParameter = 8;
8  #endif
9    // ... Proceed, using the variable tuningParameter
10 }
```

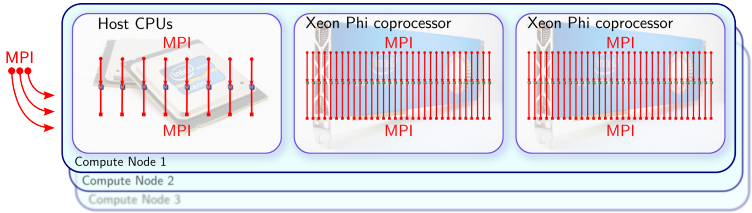

Using Multiple Coprocessors

Clusters with Intel Xeon Phi Coprocessors

- Option 1: run MPI processes on hosts and offload to coprocessors:



- Option 2: MPI as native applications on hosts and coprocessors:



Multiple Coprocessors with Explicit Offload

- Querying the number of coprocessors:

```
1 const int numDevices = _Offload_number_of_devices();
2 printf("Number of available coprocessors: %d\n" , numDevices);
```

- Specifying offload target:

```
1 #pragma offload target(mic: 0)
2   { /* ... */ }
```


Multiple Blocking Offloads Using Host Threads (Explicit Offload)

```
1 const int nDevices = _Offload_number_of_devices();
2 #pragma omp parallel for
3   for (int i = 0; i < nDevices; i++) {
4 #pragma offload target(mic: i)
5     {
6       MyFunction(/*...*/);
7     }
8   }
```

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

Blocking Explicit Offloads Using Threads: Dynamic Work Distribution Across Coprocessors

```
1  const int nDevices = _Offload_number_of_devices();
2  omp_set_num_threads(nDevices);
3  #pragma omp parallel for schedule(dynamic, 1)
4    for (int i = 0; i < nWorkItems; i++) {
5      const int iDevice = omp_get_thread_num();
6  #pragma offload target(mic: iDevice)
7      {
8        MyFunction(i);
9      }
10   }
```

- Up to 8 coprocessors, up to 32 host threads
- `nWorkItems` are dynamically scheduled on `nDevices`

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

```
1 char* offload0;
2 #pragma offload target(mic:0) signal(offload0) in(data : length(N))
3  { /* ... will not block code execution because of clause "signal" */ }
4
5 DoSomethingElse();
6
7 /* Now block until offload signalled by pointer "offload0" completes */
8 #pragma offload_wait target(mic:0) wait(offload0)
```

- Use the target number to avoid hanging

Offload diagnostics

```
user@host% export OFFLOAD_REPORT=2
user@host% ./offload-application
Transferring some data to and from coprocessor...
Done. Bye!
[Offload] [MIC 0] [File]            offload-application.cpp
[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)
user@host%
```

- 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

```
user@host% # This enables s
user@host% export MIC_ENV_PREFIX=XEONPHI
user@host%
user@host% # This sets the value of OMP_NUM_THREADS on the host:
user@host% export OMP_NUM_THREADS=32
user@host%
user@host% # This sets the value of OMP_NUM_THREADS on the coprocessor:
user@host% export XEONPHI_OMP_NUM_THREADS=236
```

Multiple Asynchronous Explicit Offloads From a Single Thread

```
1   const int nDevices = _Offload_number_of_devices();
2   char sig[nDevices];
3   for (int i = 0; i < nDevices; i++) {
4 #pragma offload target(mic: i) signal(&sig[i])
5     {
6       MyFunction(/*...*/);
7     }
8   }
9   for (int i = 0; i < nDevices; i++) {
10#pragma offload_wait target(mic: i) wait(&sig[i])
11  }
```

- Any pointer acts as a signal
- Must wait for all signals

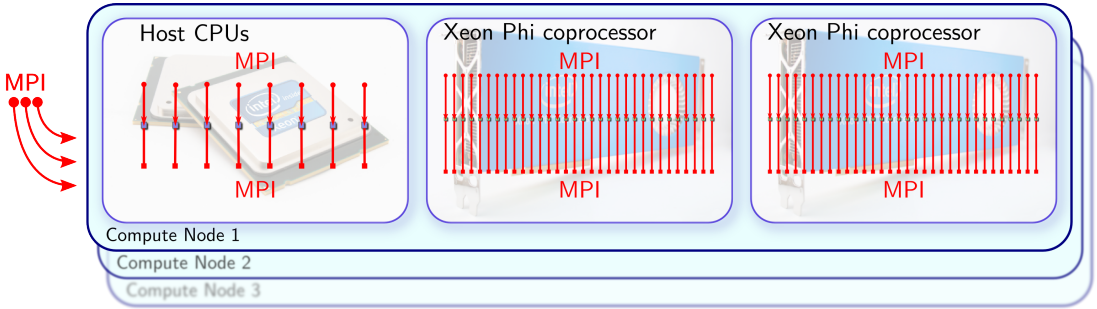
MPI Applications and Heterogeneous Clustering

Heterogeneous MPI Applications: Host + Coprocessors

```
user@host% mpirun -host mic0 -n 2 ~/Hello.MIC : -host mic1 -n 2 ~/Hello.MIC : \
% -host localhost -n 2 ~/Hello.XEON
Hello World from rank 5 running on localhost!
Hello World from rank 4 running on localhost!
Hello World from rank 2 running on mic1!
Hello World from rank 3 running on mic1!
Hello World from rank 1 running on mic0!
Hello World from rank 0 running on mic0!
MPI World size = 6 ranks
```

- Specify Xeon executable for host processes
- Specify Xeon Phi executable for coprocessor processes

Heterogeneous Distributed Computing with Xeon Phi

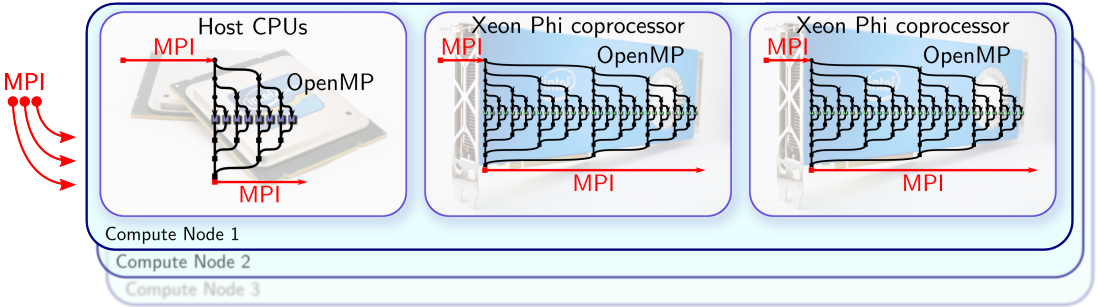


Option 1: Symmetric Pure MPI.

- MPI processes are single-threaded.
- Native MPI processes on the coprocessor.

E.g., 32 MPI processes on each CPU, 240 on each coprocessor.

Heterogeneous Distributed Computing with Xeon Phi

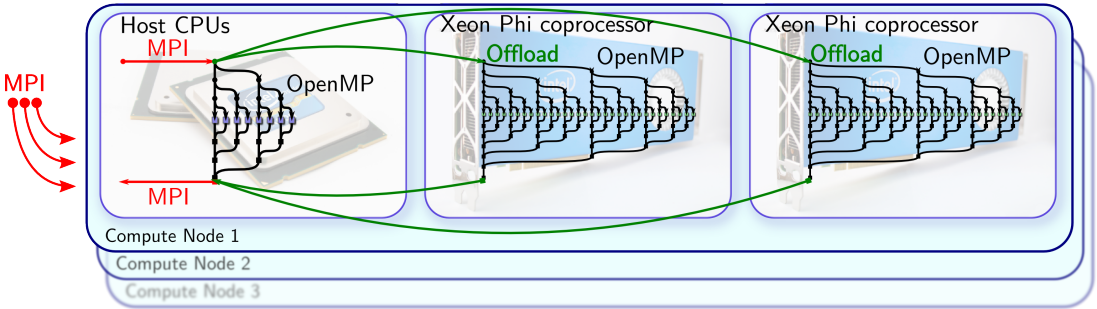


Option 2: Symmetric Hybrid MPI+OpenMP.

- MPI processes are multi-threaded with OpenMP.
- Native MPI processes on the coprocessor.

E.g., one 32-thr MPI proc on each CPU, 240-thr on each coprocessor.

Heterogeneous Distributed Computing with Xeon Phi



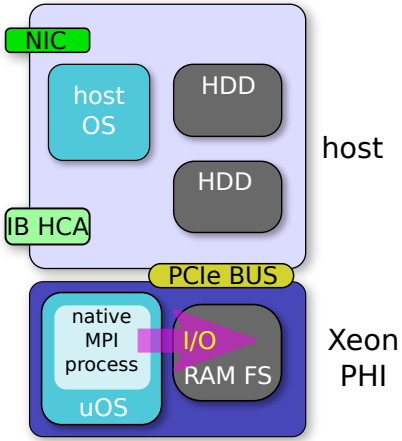
Option 3: Hybrid MPI+OpenMP with Offload.

- MPI processes are multi-threaded with OpenMP.
- MPI processes run only on CPUs.
- One or more OpenMP threads perform offload to coprocessor(s).

File I/O in MPI Applications

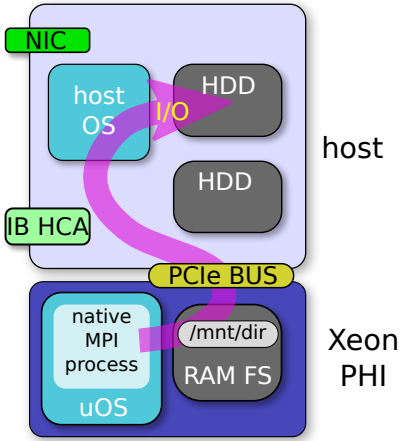
RAM Filesystem

- 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



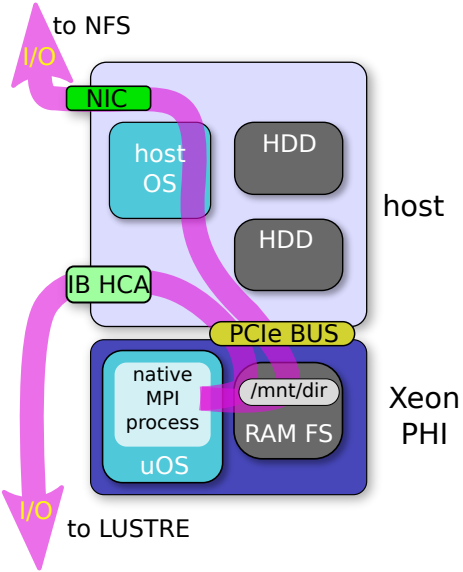
Virtio Transfer to Local Host Drives

- 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



Review: Programming Models

Programming Models

1. Native coprocessor applications
   - Compile with `-mmic`
   - Run with `micnativeloadex` or `scp+ssh`
   - The way to go for MPI applications without offload
2. Explicit offload
   - Functions, global variables require `__attribute__((target(mic)))`
   - Initiate offload, data marshalling with `#pragma offload`
   - Only bitwise-copyable data can be shared
3. Clusters and multiple coprocessors
   - `#pragma offload target(mic:i)`
   - Use threads to offload to multiple coprocessors
   - Run native MPI applications

§3. Porting Applications to the MIC Architecture

Choosing the Programming Model

To Offload or Not To Offload

For a “MIC-friendly” application,

Use offload if:

- Per-rank data set does not fit in the Xeon Phi onboard memory
- Need CPU: serial workload, intensive file I/O
- MPI bandwidth-bound or latency-bound workload
- Cannot compile some of dependencies for MIC

Use native/symmetric MPI if:

- Parallel work-items too small, so data transfer overhead is significant
- Peer-to-peer communication between workers is required
- Difficult to instrument data movement or sharing with coprocessor

PCIe Bandwidth Considerations

With data sent from host to coprocessor, communication overhead must be considered:

- PCIe bandwidth: 6 GB/s, theoretical max arithmetic performance 1 TFLOP/s, practical memory bandwidth 150-170 GB/s
- Offload if MIC performs $\gg$ 1000 operations per transferred word
- Algorithms with strong complexity scaling (e.g., $O(n^2)$) likely less impacted by communication than with weak scaling (e.g., $O(n)$, $O(n \log n)$)

Cross-Compilation of User Applications

Simple Applications, 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
```

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
...
```


Static Libraries with Offload

In offload applications, additional object files are produced:

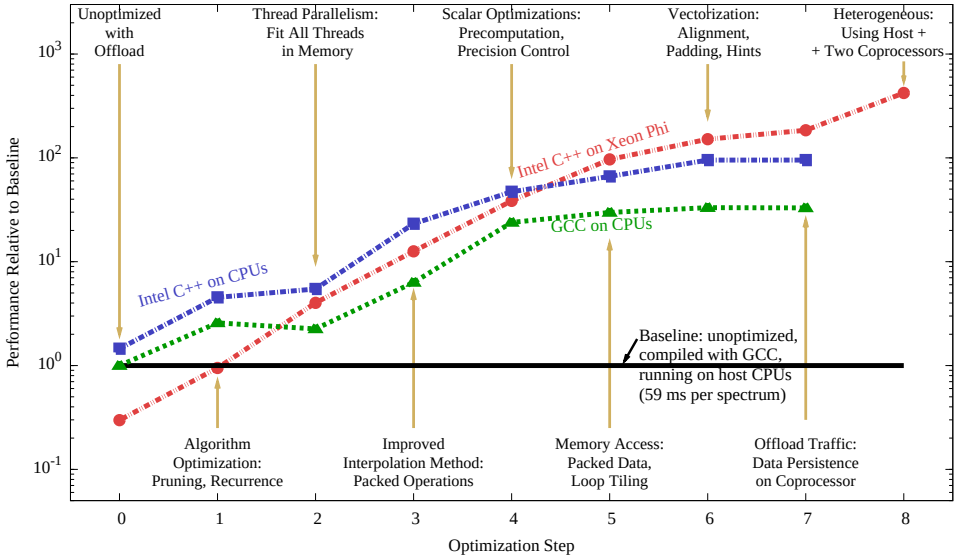
```
user@host% # Program in myobject.cc contains #pragma offload
user@host% icpc -c myobject.cc
user@host% ls
myobject.cc  myobjectMIC.o  myobject.o
```

In order to compile the `*MIC.o` files into a static library with offload, use `xiar -qoffload-build` instead of `ar`.

See white paper for more details:
http://research.colfaxinternational.com/post/2013/05/03/Fast-Library-Xeon-Phi.aspx

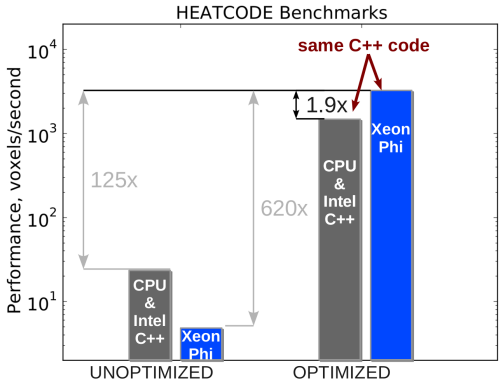
Performance Expectations

Performance on MIC is a Function of Optimization Level



Performance on MIC is a Function of Optimization Level

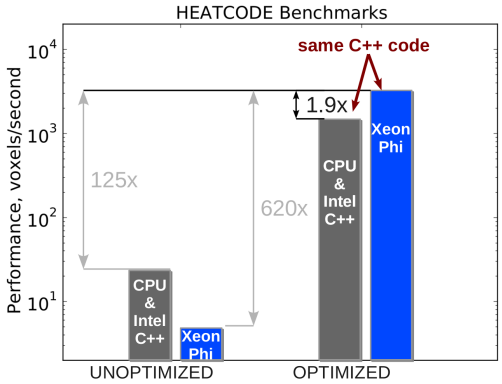
- Performance will be disappointing if code is not optimized for multi-core CPUs
- Optimized code runs better on the MIC platform *and* on the multi-core CPU
- Single code for two platforms + Ease of porting = Incremental optimization



Case study:
http://research.colfaxinternational.com/post/2013/11/25/sc13-talk.aspx

Caution on Comparative Benchmarks

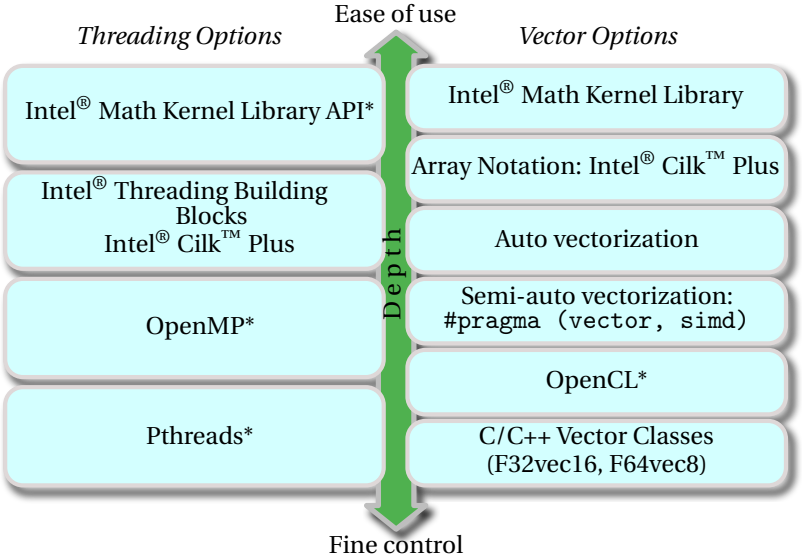
- In most of our benchmarks, "Xeon Phi" = 5110P SKU (60 cores, TDP 225 W, $2.7k), "CPU" = dual Xeon E5-2680 (16 cores, TDP 260 W, $3.4k + system cost)
- Why dual CPU vs single coprocessor? Approximately the same Thermal Design Power (TDP) and cost.



Case study:
http://research.colfaxinternational.com/post/2013/11/25/sc13-talk.aspx

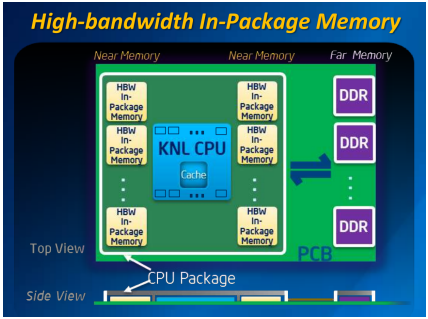
Future-Proofing: Reliance on Compiler and Libraries

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

§4. Parallel Scalability on Intel Architectures

Vectorization (Single Instruction Multiple Data, SIMD, Parallelism)

SIMD Operations

SIMD — Single Instruction Multiple Data

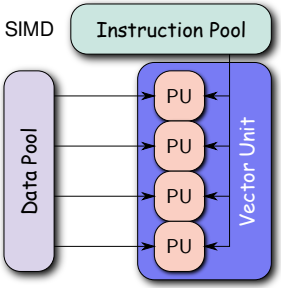
Scalar Loop

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

SIMD Loop

```
1 for (i = 0; i < n; i += 4)
2   A[i:(i+4)] = A[i:(i+4)] + B[i:(i+4)];
```

Each SIMD addition operator acts on 4 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	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

Explicit Vectorization: Compiler Intrinsics

SSE2 Intrinsics

```
1 for (int i=0; i<n; i+=4) {
2   __m128 Avec=_mm_load_ps(A+i);
3   __m128 Bvec=_mm_load_ps(B+i);
4   Avec=_mm_add_ps(Avec, Bvec);
5   _mm_store_ps(A+i, Avec);
6 }
```

IMCI Intrinsics

```
1 for (int i=0; i<n; i+=16) {
2   __m512 Avec=_mm512_load_ps(A+i);
3   __m512 Bvec=_mm512_load_ps(B+i);
4   Avec=_mm512_add_ps(Avec, Bvec);
5   _mm512_store_ps(A+i, Avec);
6 }
```

- The arrays `float A[n]` and `float B[n]` are aligned on a 16-byte (SSE2) and 64-byte (IMCI) boundary
- `n` is a multiple of 4 for SSE and a multiple of 16 for IMCI
- Variables `Avec` and `Bvec` are
  $128 = 4 \times$ `sizeof(float)` bits in size for SSE2 and
  $512 = 16 \times$ `sizeof(float)` bits for the Intel Xeon Phi architecture

Automatic Vectorization of Loops

```c
#include <stdio.h>

int main(){
  const int n=8;
  int i;
  int A[n] __attribute__((aligned(64)));
  int B[n] __attribute__((aligned(64)));

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

  // This loop will be auto-vectorized
  for (i=0; i<n; i++)
    A[i]+=B[i];

  // Output
  for (i=0; i<n; i++)
    printf("%2d %2d %2d\n", i, A[i], B[i]);
}
```

```
user@host% icpc autovec.c -vec-report3
autovec.c(10): (col. 3) remark:
    loop was not vectorized:
    vectorization possible
    but seems inefficient.
autovec.c(14): (col. 3) remark:
    LOOP WAS VECTORIZED.
autovec.c(18): (col. 3) remark:
    loop was not vectorized:
    existence of vector
    dependence.
user@host% ./a.out
 0  0  0
 1  2  1
 2  4  2
 3  6  3
 4  8  4
 5 10  5
 6 12  6
 7 14  7
```


Automatic Vectorization of Loops on MIC architecture

Compilation and runtime output of the code for Intel Xeon Phi execution

```
user@host% icpc autovec.c -vec-report3 -mmic
autotest.c(10): (col. 3) remark: LOOP WAS VECTORIZED.
autotest.c(14): (col. 3) remark: LOOP WAS VECTORIZED.
autotest.c(18): (col. 3) remark: loop was not vectorized:
                         existence of vector dependence.
user@host% micnativeloadex a.out
 0  0  0
 1  2  1
 2  4  2
 3  6  3
 4  8  4
 5 10  5
 6 12  6
 7 14  7
```

Automatic Vectorization of Loops

Limitations:

- Only `for`-loops can be auto-vectorized. Number of iterations must be known at a runtime and/or compilation time
- Memory access in the loop must have a regular pattern, ideally with unit stride
- Non-standard loops that cannot be automatically vectorized:
  - loops with irregular memory access pattern
  - calculations with vector dependence
  - `while`-loops, for-loops with undetermined number of iterations
  - outer loops (unless `#pragma simd` overrides this restriction)
  - loops with complex branches (i.e., `if`-conditions)
  - anything else that cannot be, or is very difficult to vectorize.

Multi-Threading: OpenMP

Parallelism in Shared Memory: OpenMP and Intel Cilk Plus

- Intel Cilk Plus
  - Good performance “out of the box”
  - Little freedom for fine-tuning
  - Programmer should focus on exposing the parallelism
  - Low-level optimization (thread creation, work distribution and data sharing) is performed by the Cilk Plus library
  - Novel framework
- OpenMP
  - Easy to use for simple algorithms
  - For complex parallelism, may require more tuning to perform well
  - Allows more control over synchronization, work scheduling and distribution
  - Well-established framework

Program Structure in OpenMP

```
1  main() {                          // Begin serial execution.
2  ...                               // Only the initial thread executes
3  #pragma omp parallel              // Begin a parallel construct and form
4     {                              // a team.
5  #pragma omp sections              // Begin a work-sharing construct.
6       {
7  #pragma omp section               // One unit of work.
8         {...}
9  #pragma omp section               // Another unit of work.
10        {...}
11      }                            // Wait until both units of work complete.
12     ...                           // This code is executed by each team member.
13 #pragma omp for                   // Begin a work-sharing Construct
14      for(...)
15      {                            // Each iteration chunk is unit of work.
16        ...                        // Work is distributed among the team members.
17      }                            // End of work-sharing construct.
```


Program Structure in OpenMP

```
18 #pragma omp critical          // Begin a critical section.
19     {...}                    // Only one thread executes at a time.
20 #pragma omp task              // Execute in another thread without blocking
21     {...}
22     ...                      // This code is executed by each team member.
23 #pragma omp barrier           // Wait for all team members to arrive.
24     ...                      // This code is executed by each team member.
25     }                        // End of Parallel Construct
26                              // Disband team and continue serial execution.
27 ...                          // Possibly more parallel constructs.
28 }                            // End serial execution.
```

1. Code outside `#pragma omp parallel` is serial, i.e., executed by only one thread
2. Code directly inside `#pragma omp parallel` is executed by each thread
3. Code inside work-sharing construct `#pragma omp for` is distributed across the threads in the team

“Hello World” OpenMP Programs

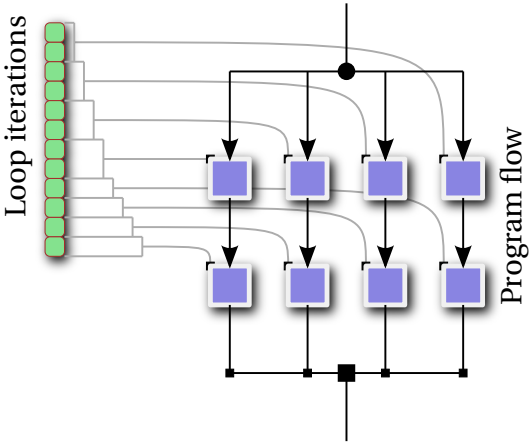
```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(){
5    const int nt=omp_get_max_threads();
6    printf("OpenMP with %d threads\n", nt);
7
8  #pragma omp parallel
9    printf("Hello World from thread %d\n", omp_get_thread_num());
10 }
```


“Hello World” OpenMP Programs

```
user@host% export OMP_NUM_THREADS=5
user@host% icpc -openmp hello_omp.cc
user@host% ./a.out
OpenMP with 5 threads
Hello World from thread 0
Hello World from thread 3
Hello World from thread 1
Hello World from thread 2
Hello World from thread 4
user@host% icpc -openmp-stubs hello_omp.cc
hello_omp.cc(8): warning #161: unrecognized #pragma
  #pragma omp parallel
          ^
user@host% ./a.out
OpenMP with 1 threads
Hello World from thread 0
```


Loop-Centric Parallelism: For-Loops in OpenMP

- Simultaneously launch multiple threads
- Scheduler assigns loop iterations to threads
- Each thread processes one iteration at a time



Parallelizing a `for`-loop.

Loop-Centric Parallelism: For-Loops in OpenMP

The OpenMP library will distribute the iterations of the loop following the `#pragma omp parallel for` across threads.

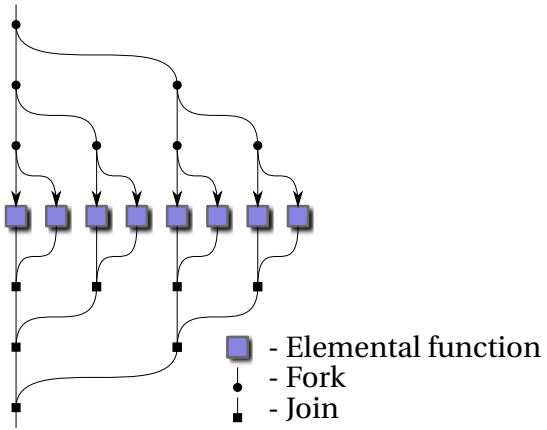
```
#pragma omp parallel for
    for (int i=0; i<n; i++) {
        printf("Iteration %d is processed by thread %d\n",
                                    i, omp_get_thread_num());
        // ... iterations will be distributed across available threads...
    }
```


Loop-Centric Parallelism: For-Loops in OpenMP

```
#pragma omp parallel
{
    // Code placed here will be executed by all threads.
    // Stack variables declared here will be private to each thread.
    int private_number=0;
#pragma omp for schedule(dynamic, 4)
    for (int i=0; i<n; i++) {
        // ... iterations will be distributed across available threads...
    }
    // ... code placed here will be executed by all threads
}
```


Fork-Join Model of Parallel Execution

- Each thread can spawn daughter threads
- Available threads pick up queued tasks
- Expresses algorithms that cannot be expressed in the loop model (e.g., parallel recursion)

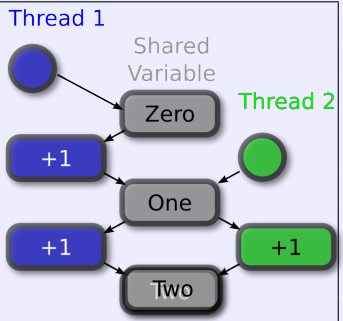


Fork-join model of parallel execution.

(see `#pragma omp task` functionality, e.g., here)

Synchronization: Avoiding Unpredictable Program Behavior

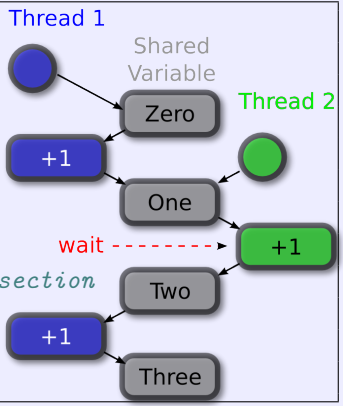
```
1  #include <omp.h>
2  #include <stdio.h>
3  int main() {
4      const int n = 1000;
5      int total = 0;
6  #pragma omp parallel for
7      for (int i = 0; i < n; i++) {
8          // Race condition
9          total = total + i;
10     }
11     printf("total=%d (must be %d)\n", total, ((n-1)*n)/2);
12 }
```




```
user@host% icpc -o omp-race omp-race.cc -openmp
user@host% ./omp-race
total=208112 (must be 499500)
```


Synchronization: Avoiding Unpredictable Program Behavior

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main() {
4      const int n = 1000;
5      int total = 0;
6  #pragma omp parallel for
7      for (int i = 0; i < n; i++) {
8  #pragma omp critical
9          { // Only one thread at a time can execute this section
10             total = total + i;
11         }
12     }
```




```
user@host% icpc -o omp-critical omp-critical.cc -openmp
user@host% ./omp-race
total=499500 (must be 499500)
```


Synchronization: Avoiding Unpredictable Program Behavior

This parallel fragment of code has predictable behavior, because the race condition was eliminated with *an atomic operation*:

```
1 #pragma omp parallel for
2    for (int i = 0; i < n; i++) {
3        // Lightweight synchronization
4 #pragma omp atomic
5        sum += i;
6    }
```


Synchronization: Avoiding Unpredictable Program Behavior

Read : operations in the form `v = x`

Write : operations in the form `x = v`

Update : operations in the form `x++`, `x--`, `--x`, `++x`, `x` *binop*`=` *expr*
and `x = x` *binop expr*

Capture : operations in the form `v = x++`, `v = x-`, `v = -x`, `v = ++x`,
`v = x` *binop expr*

- Here `x` and `v` are scalar variables
- *binop* is one of `+`, `*`, `-`, `- /`, `&`, `^ `, `|`, `«`, `»`.
- No "trickery" is allowed for atomic operations:
  - no operator overload,
  - no non-scalar types,
  - no complex expressions.

Reduction: Avoiding Synchronization

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main() {
5      const int n = 1000;
6      int sum = 0;
7  #pragma omp parallel for reduction(+: sum)
8      for (int i = 0; i < n; i++) {
9          sum = sum + i;
10     }
11     printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
12 }
```

```
user@host% icpc -o omp-reduction omp-reduction.cc -openmp
user@host% ./omp-reduction
sum=499500 (must be 499500)
```


Implementation of Reduction using Private Variables

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main() {
5      const int n = 1000;
6      int sum = 0;
7  #pragma omp parallel
8      {
9          int sum_th = 0;
10 #pragma omp for
11         for (int i = 0; i < n; i++)
12             sum_th = sum_th + i;
13 #pragma omp atomic
14         sum += sum_th;
15     }
16     printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
17 }
```

Multi-Threading: Intel Cilk Plus

Program Structure with Intel Cilk Plus

- Only three keywords in the Cilk Plus standard:
  - `_Cilk_for`
  - `_Cilk_spawn`
  - `_Cilk_sync`
- Programming for coprocessors may also require keywords
  - `_Cilk_shared`
  - `_Cilk_offload`
- Language extensions
  - array notation
  - hyperobjects
  - elemental function
  - `#pragma simd`
- Guarantees that serialized code will produce the same results as parallel code
- Integrates vectorization and thread-parallelism

“Hello World” Intel Cilk Plus Programs

```
#include <cilk/cilk.h>
#include <stdio.h>

int main(){
  const int nw=__cilkrts_get_nworkers();
  printf("Cilk Plus with %d workers.\n", nw);

  _Cilk_for (int i=0; i<nw; i++) // Light workload: gets serialized
    printf("Hello World from worker %d\n", __cilkrts_get_worker_number());

  _Cilk_for (int i=0; i<nw; i++) {
    double f=1.0;
    while (f<1.0e40) f*=2.0; // Extra workload: gets parallelized
    printf("Hello Again from worker %d (%f)\n",
           __cilkrts_get_worker_number(), f);
  }
}
```


“Hello World” Intel Cilk Plus Programs

```
user@host$ export CILK_NWORKERS=5
user@host$ icpc hello_cilk.cc
user@host$ ./a.out
Cilk Plus with 5 workers.
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello Again from worker 0 (10889035741470030830827987437816582766592.000000)
Hello Again from worker 0 (10889035741470030830827987437816582766592.000000)
Hello Again from worker 1 (10889035741470030830827987437816582766592.000000)
Hello Again from worker 3 (10889035741470030830827987437816582766592.000000)
Hello Again from worker 0 (10889035741470030830827987437816582766592.000000)
user@host$
```

“Hello World” Intel Cilk Plus Programs

```
user@host$ export CILK_NWORKERS=5
user@host$ icpc -cilk-serialize hello_cilk.cc
user@host$ ./a.out
Cilk Plus with 5 workers.
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello Again from worker 0 (108890357414700308308279874378165827665 92.000000)
Hello Again from worker 0 (108890357414700308308279874378165827665 92.000000)
Hello Again from worker 0 (108890357414700308308279874378165827665 92.000000)
Hello Again from worker 0 (108890357414700308308279874378165827665 92.000000)
Hello Again from worker 0 (108890357414700308308279874378165827665 92.000000)
user@host$
```

Loop-Centric Parallelism: For-Loops in Intel Cilk Plus

Intel Cilk Plus distributes the iterations between the available workers

```
_Cilk_for (int i=0; i<n; i++) {
    // ... iterations will be distributed across available threads...
    printf("Iteration %d is processed by worker %d\n",
                    i, __cilkrts_get_worker_number());
 }
```

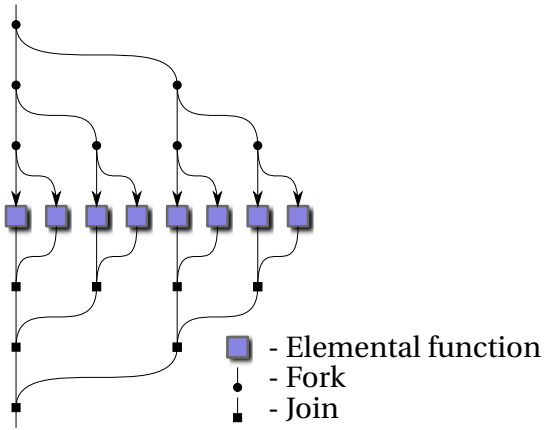
Controlling *grain size* in Intel Cilk Plus

```
#pragma cilk grainsize = 4
_Cilk_for (int i = 0; i < N; i++) {
 // ...
}
```

Intel Cilk Plus uses the “work stealing” scheduling mode.

Fork-Join Model of Parallel Execution

- Each task can spawn daughter tasks
- Available workers pick up queued tasks
- Expresses algorithms that cannot be expressed in the loop model (e.g., parallel recursion)



Fork-join model of parallel execution.

(see `_Cilk_spawn` functionality, e.g., here or at www.cilkplus.org)

Cooperation Between Threads in Intel Cilk Plus

Limited vocabulary of Cilk is conducive to good parallel programming:

- No synchronization facilities (locks). Programmer must use reduction instead.
  - Reducers (C++ templates) available for common operations on objects
  - Custom reducers can be written
- No private variables. Use holders (C++ template classes) instead.
- Little user control over the number of workers and scheduling.

More information, e.g., www.cilkplus.org

C++ Language Extensions of Intel Cilk Plus

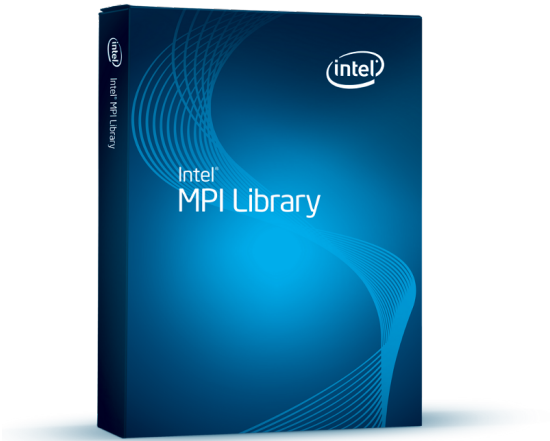
- Elemental functions: compiled for usage in thread- and data-parallel loops (see, e.g., white paper)
- Array notation: Intel C++ compiler understands expressions like `A[0:M][0:N]=B[0:M][0:N]`
- Guided automatic vectorization: `#pragma simd`.

Task Parallelism in Distributed Memory, MPI

Task Parallelism in Distributed Memory, MPI

The most commonly used framework for distributed memory HPC calculations is the Message Passing Interface (MPI).

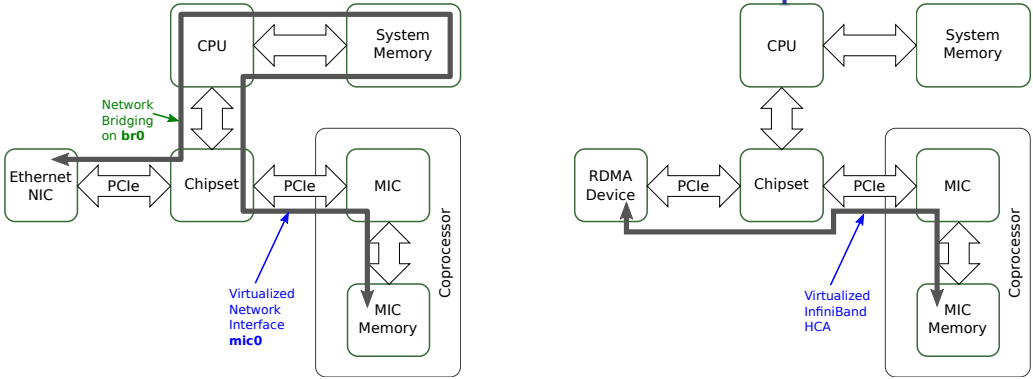
Intel MPI library implements MPI for the x86 and for the MIC architectures.



Compiling and Running MPI applications

1. Compile and link with the MPI wrapper of the compiler:
   - `mpiicc` for C,
   - `mpiicpc` for C++,
   - `mpiifort` for Fortran 77 and Fortran 95.
2. Set up MPI environment variables *and* `I_MPI_MIC=1`
3. NFS-share or copy the MPI library and the application executable to the coprocessors
4. Launch with the tool `mpirun`
   - Colon-separated list of executables and hosts (argument `-host` *hostname*),
   - Alternatively, use the machine file to list hosts
   - Coprocessors have hostnames defined in `/etc/hosts`

Peer-to-Peer Communication between Coprocessors



- Left: Gigabit Ethernet bridging on host allows to place coprocessors on the same subnet as hosts(`I_MPI_FABRICS=tcp`)
- Right: Coprocessor Communication Link (CCL) – virtualization of an InfiniBand device on each coprocessor (`I_MPI_FABRICS=dapl`)

Structure of MPI Applications

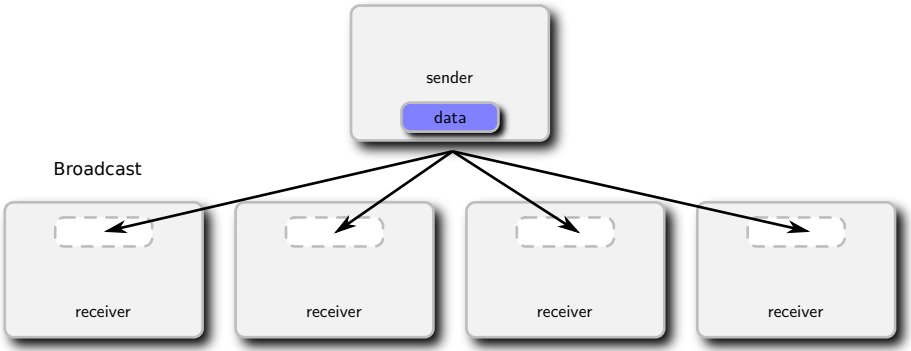
```
#include "mpi.h"
int main(int argc, char** argv) {
  int ret = MPI_Init(&argc,&argv);   // Set up MPI environment
  if (ret != MPI_SUCCESS) {
          MyErrorLogger("...");
          MPI_Abort(MPI_COMM_WORLD, ret);
  }
  int worldSize, myRank, myNameLength;
  char myName[MPI_MAX_PROCESSOR_NAME];
  MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Get_processor_name(myName, &myNameLength);
  // ... Perform work, exchange messages with MPI_Send, MPI_Recv, etc. ...
  // Terminate MPI environment
  MPI_Finalize();
}
```


Point to Point Communication

```
1  if (rank == receiver) {
2
3    char incomingMsg[messageLength];
4    MPI_Recv (&incomingMsg, messageLength, MPI_CHAR, sender,
5                      tag, MPI_COMM_WORLD, &stat);
6    printf ("Received message with tag %d: '%s'\n", tag, incomingMsg);
7
8  } else if (rank == sender) {
9
10   char outgoingMsg[messageLength];
11   strcpy(outgoingMsg, "/Jenny");
12   MPI_Send(&outgoingMsg, messageLength, MPI_CHAR, receiver, tag, MPI_COMM_WORLD);
13
14 }
```

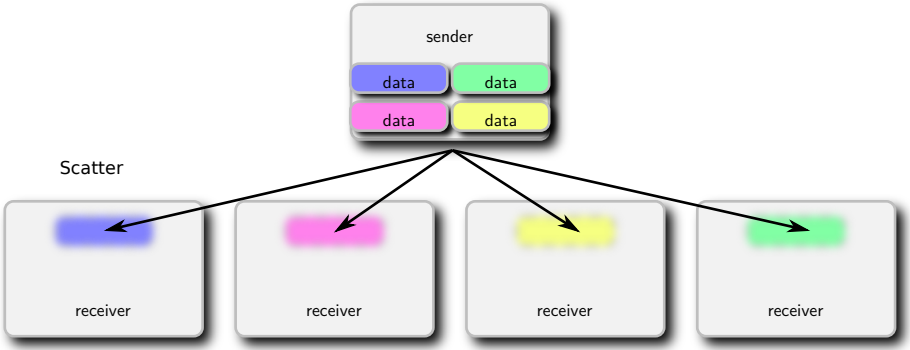

Collective Communication: Broadcast

```
1 int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype,
2                int root, MPI_Comm comm );
```



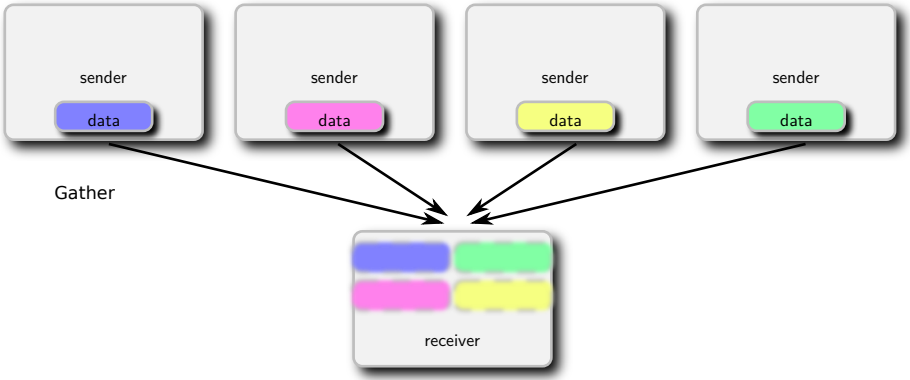
Collective Communication: Scatter

```
1 int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
2                 int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
```



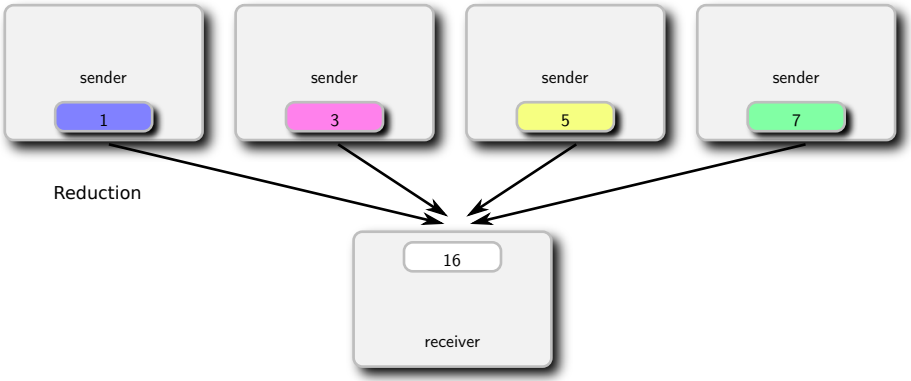
Collective Communication: Gather

```
1 int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
2                void *recvbuf, int recvcnt, MPI_Datatype recvtype,
3                int root, MPI_Comm comm);
```



Collective Communication: Reduction

```
1 int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
2                MPI_Op op, int root, MPI_Comm comm);
```



Available reducers: max/min, minloc/maxloc, sum, product, AND, OR, XOR (logical or bitwise).

Review: Parallel Scalability

Expressing Parallelism

1. Data parallelism (vectorization)
   - Automatic vectorization by the compiler: portable and convenient
   - For-loops and array notation can be vectorized
   - Compiler hints (`#pragma simd`, `#pragma ivdep`, etc.) to assist the compiler
2. Shared-memory parallelism with OpenMP and Intel Cilk Plus
   - Parallel threads access common memory for reading and writing
   - Parallel loops: `#pragma omp parallel for` and `_Cilk_for` — automatic work distribution
   - In OpenMP: private and shared variables; synchronization, reduction.
3. Distributed-memory parallelism with MPI
   - MPI processes do not share memory, but can send information to each other
   - All MPI processes execute the same code; role is determined by its rank
   - Point-to-point and collective communication patterns

§5. Optimization for the Intel Xeon Product Family

Optimization Roadmap

Performance Expectations

vs.

One Intel Xeon Phi coprocessor

Two Intel Xeon Sandy Bridge CPUs

- Up to 2x-3x for linear algebraic workloads
- Up to 2x-4x for bandwidth-bound and transcendental arithmetics
- Why compare 1 coprocessor agains 2 processors?
  Same thermal design power (TDP).

See also "Intel Xeon Product Family: Performance Brief"

Optimization Checklist

1. Scalar optimization
2. Vectorization
3. Scale above 100 threads
4. Arithmetically intensive or bandwidth-limited
5. Efficient cooperation between the host and the coprocessor(s)

Optimization for Xeon Family: Two Birds with One Stone

Our most flattering criticism:

"...the labs should have demonstrated problems that were geared more towards showing off the Phi's capabilities rather than mostly dealing with general Intel compiler optimizations that worked on both the Xeon & Phi. (Obviously the optimizations for one are good on the other, but this class could have just been billed as an Intel Compiler Optimization course with the same material)."

Customer of Colfax's 4-day Parallel Programming Workshop

From the creators of MIC:

"Really, everything that we talked about in the book was about, fundamentally, parallel programming... The one thing we've seen over and over again is that if you parallelize either for Xeon, or for Xeon Phi, they both benefit."

Jim Jeffers, author of "Intel Xeon Phi Coprocessor High Performance Programming", in an interview to insideHPC.

Finding Bottlenecks with Intel VTune Amplifier

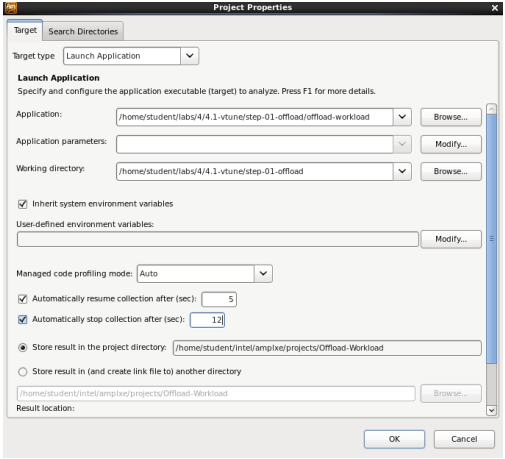
Intel VTune Parallel Amplifier XE

Hardware event-based profiler for parallel applications on Xeon CPUs and Xeon Phi coprocessors.

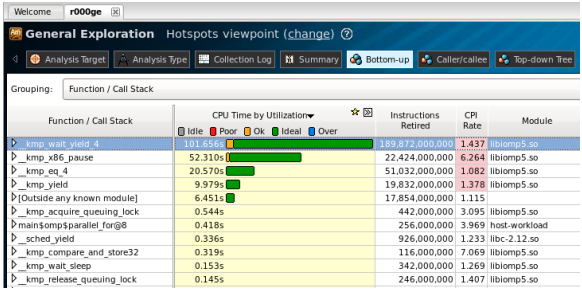
Bottleneck detection down to a single line of code, hardware event collection, minimal impact on performance.

Using VTune

Setting up a VTune project:

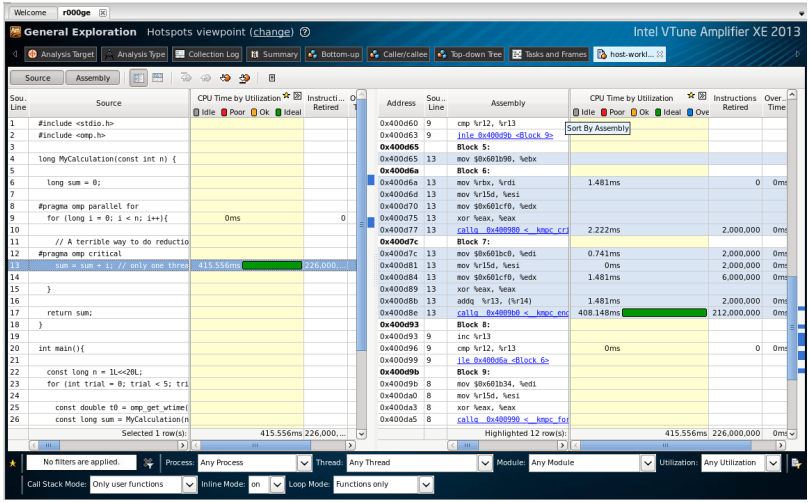


Results of profiling, bottom-up view:



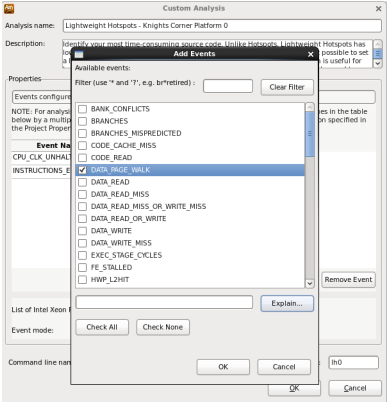
Using VTune

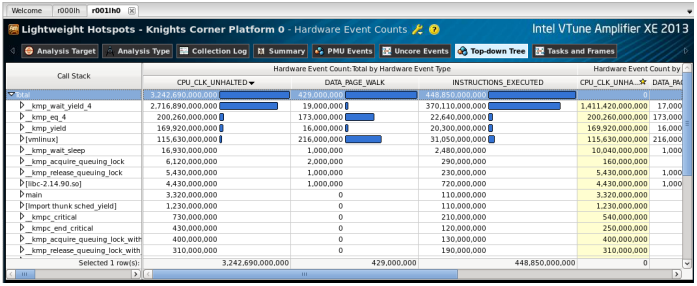
Locating hotspots down to a single line of code:



Using VTune

Analyzing custom events



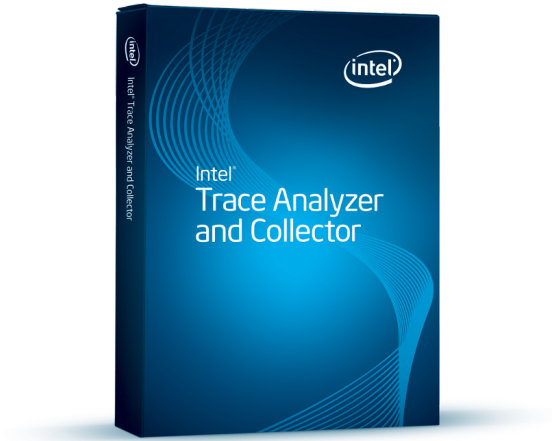


MPI Diagnostics Using Intel Trace Analyzer and Collector

Intel Trace Analyzer and Collector

Profiler for MPI Applications on Xeon and Xeon Phi architectures.

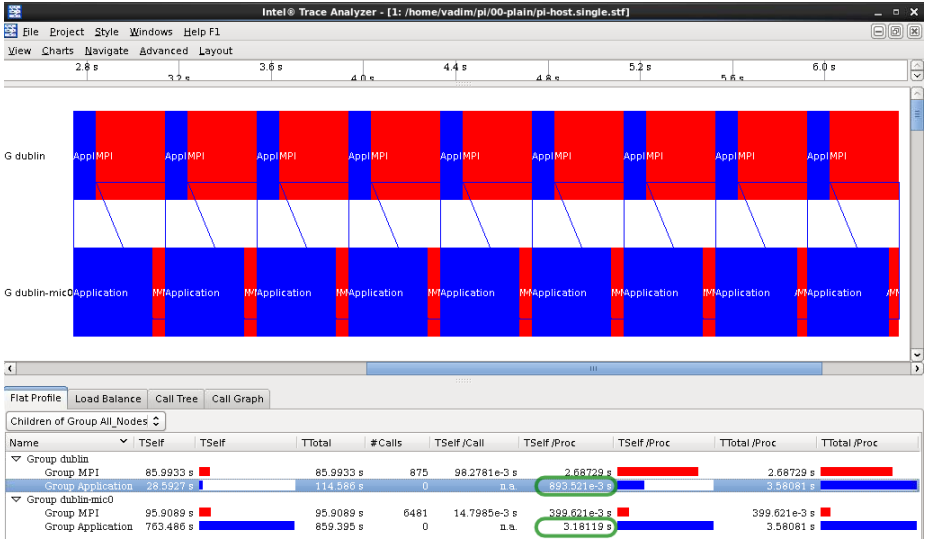
Graphical user interface, visualization of computation and communication.



Using Intel Trace Analyzer and Collector

```
user@host% source /opt/intel/itac/8.1.0.024/bin/itacvars.sh
user@host% source /opt/intel/itac/8.1.0.024/mic/bin/itacvars.sh
user@host% mpiicpc -mkl -o pi_mpi pi_mpi.c
user@host% mpiicpc -mmic -mkl -o pi_mpi.mic pi_mpi.c
user@host% scp pi_mpi.mic mic0:~/
pi_mpi.mic                                    100%  433KB 432.5KB/s   00:00
user@host% export VT_LOGFILE_FORMAT=stfsingle
user@host% mpirun -trace -n 32 -host localhost ./pi_mpi : \
% -n 240 -host mic0 ~/pi_mpi.mic
Time, s:    0.36
[0] Intel(R) Trace Collector INFO: Writing tracefile pi_mpi.single.stf
in /home/user/pi
user@host% traceanalyzer pi_mpi.single.stf
```

Using Intel Trace Analyzer and Collector

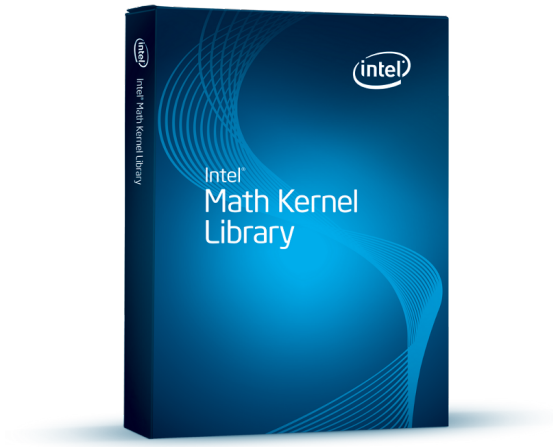


Intel Math Kernel Library (MKL)

Intel Math Kernel Library (MKL)

Linear algebra, fast Fourier transforms, vector math, parallel random numbers, statistics, data fitting, sparse solvers.

Intel MKL functions are optimized for Xeon Processors as well as for Xeon Phi coprocessors.



Using Intel MKL

Three modes of usage:

- Automatic Offload
  - No code change required to offload calculations to a Xeon Phi coprocessor
  - Automatically uses both the CPU and the coprocessor
  - The library takes care of data transfer and execution management
- Compiler-Assisted Offload
  - Programmer maintains explicit control of data transfer and remote execution
  - Requires using compiler offload pragmas and directives
- Native Execution
  - Uses an Intel Xeon Phi coprocessor as an independent compute node.
  - Data initialized & processed on the coprocessor, or communicated via MPI

Using MKL in Automatic Offload Mode

Calling an MKL function from host code:

```
1 sgemm(&transa, &transb, &SIZE, &SIZE, &SIZE, &alpha,
2         A, &newLda, B, &newLda, &beta, C, &SIZE);
```

Compiling and running the code. Calculation will be offloaded to a Xeon Phi coprocessor, if one is available at runtime.

```
user@host% icpc -c mycode.cc -mkl -o mycode
user@host% export MKL_MIC_ENABLE=1
user@host% ./mycode
```


Using MKL in Compiler-Assisted Offload Mode

Calling an MKL function from offloaded section:

```
#pragma offload target(mic) \
  in(transa, transb, N, alpha, beta) \
  in(A:length(matrix_elements)) \
  in(B:length(matrix_elements)) \
  out(C:length(matrix_elements) alloc_if(0))
  {
    sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N, &beta, C, &N);
  }
```

Compiling and running the code. If no coprocessor at runtime, MKL will fall back to CPU calculation.

```
user@host% icpc -c mycode.cc -mkl -o mycode
user@host% ./mycode
```


Using MKL Native Execution Mode

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
  const size_t N = 1<<29L;
  const size_t F = sizeof(float);
  float* A = (float*)malloc(N*F);
  srand(0); // Initialize RNG
  for (int i = 0; i < N; i++) {
   A[i]=(float)rand() /
        (float)RAND_MAX;
  }
  printf("Generated %ld random \
numbers\nA[0]=%e\n", N, A[0]);
  free(A);
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <mkl_vsl.h>
int main() {
  const size_t N = 1<<29L;
  const size_t F = sizeof(float);
  float* A = (float*)malloc(N*F);
  VSLStreamStatePtr rnStream;
  vslNewStream( &rnStream, //Init RNG
        VSL_BRNG_MT19937, 1 );
  vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD,
        rnStream, N, A, 0.0f, 1.0f);
  printf("Generated %ld random \
numbers\nA[0]=%e\n", N, A[0]);
  free(A);
}
```


Using MKL in Native Execution Mode

```
user@host% icpc -mmic -o rand \
% rand.cc
user@host% # Run on coprocessor
user@host% # and benchmark
user@host% time micnativeloadex \
% rand
Generated 536870912 random numbers
A[0]=8.401877e-01


real 0m56.591s
user 0m0.002s
sys 0m0.011s
```

```
user@host% icpc -mkl -mmic -o \
% rand-mkl rand-mkl.cc
user@host% export SINK_LD_LIBRARY_PATH=\
% /opt/intel/composerxe/mkl/lib/mic:\
% /opt/intel/composerxe/lib/mic
user@host% time micnativeloadex rand-mkl
Generated 536870912 random numbers
A[0]=1.343642e-01


real 0m7.951s
user 0m0.053s
sys 0m0.168s
```

On Intel Xeon Phi coprocessor, *random number generation* with Intel MKL is 7x faster than with the C standard Library.

Scalar Optimization Considerations

Optimization Level

```
user@host% icc -o mycode -O3 source.c
```

The default optimization level -O2

- optimization for speed
- automatic vectorization
- inlining
- constant propagation
- dead-code elimination
- loop unrolling

```
1 #pragma intel optimization_level 3
2 void my_function() {
3   //...
4 }
```

Optimization level -O3

- enables more aggressive optimization
- loop fusion
- block-unroll-and-jam
- `if`-statement collapse

Using the const Qualifier

```
1  #include <stdio.h>
2  int main() {
3      const int N=1<<28;
4      double w = 0.5;
5      double T = (double)N;
6      double s = 0.0;
7      for (int i = 0; i < N; i++)
8          s += w*(double)i/T;
9      printf("%e\n", s);
10 }
```

```
user@host% icpc noconst.cc
user@host% time ./a.out
6.710886e+07
real 0m0.461s
user 0m0.460s
sys 0m0.001s
```

```
1  #include <stdio.h>
2  int main() {
3      const int N=1<<28;
4      const double w = 0.5;
5      const double T = (double)N;
6      double s = 0.0;
7      for (int i = 0; i < N; i++)
8          s += w*(double)i/T;
9      printf("%e\n", s);
10 }
```

```
user@host% icpc const.cc
user@host% time ./a.out
6.710886e+07
real 0m0.097s
user 0m0.094s
sys 0m0.003s
```

Array Reference by Index instead of Pointer Arithmetics

```
1 for (int i = 0; i < N; i++)
2   for (int j = 0; j < N; j++) {
3     float* cp = c + i*N + j;
4     for (int k = 0; k < N; k++)
5       *cp += a[i*N+k]*b[k*N+j];
6   }
```

```
1 for (int i = 0; i < N; i++)
2   for (int j = 0; j < N; j++) {
3
4     for (int k = 0; k < N; k++)
5       c[i*N+j] += a[i*N+k]*b[k*N+j];
6   }
```

```
user@host% icc array_pointer.cc
user@host% time ./a.out
real 0m1.110s
user 0m1.104s
sys 0m0.005s
```

```
user@host% icpc array_index.cc
user@host% time ./a.out
real 0m0.228s
user 0m0.225s
sys 0m0.002s
```

- With *Pointer arithmetics*, the code is 5x slower than with reference to array elements *by index*.

Common Subexpression Elimination

```
1 for (int i = 0; i < n; i++)
2 {
3   for (int j = 0; j < m; j++) {
4     const double r =
5         sin(A[i])*cos(B[j]);
6     // ...
7   }
8 }
```

```
1 for (int i = 0; i < n; i++) {
2   const double sin_A = sin(A[i]);
3   for (int j = 0; j < m; j++) {
4     const double cos_B = cos(B[j]);
5     const double r = sin_A*cos_B;
6     // ...
7   }
8 }
```

- The value of `sin_A` can be calculated once and re-used `m` times in the `j`-loop
- In some cases, at `-O2` compiler eliminates common subexpressions automatically

Ternary if-operator Trap

- Ternary if operator `( ? : )` is a short-hand for `if ...else`
- Example: the `min()` function as a pre-processor expression

```
1 #define min(a, b) ( (a) < (b) ? (a) : (b) )
2 const float c = min(my_function(x), my_function(y));
```

- Problem: line 2 calls `my_function()` 3 times
- Optimization:

```
1 #define min(a, b) ( (a) < (b) ? (a) : (b) )
2 const float result_a = my_function(x);
3 const float result_b = my_function(y);
4 const float c = min(result_a, result_b);
```

Strength Reduction

Replace expensive operations with a combination of fast operations.

Example 1: replacing division with multiplication by the precomputed reciprocal:

```
1 for (int i = 0; i < n; i++)  {
2   A[i] /= n;
3 }
```

```
1 const float rn = 1.0f/(float)n;
2 for (int i = 0; i < n; i++)
3   A[i] *= rn;
```

Example 2: algebraic transformations to replace two divisions with one

```
1 for (int i = 0; i < n; i++)  {
2   A[i] = (B[i]/C[i])/D[i];
3   E[i] = A[i]/B[i] + C[i]/D[i];
4
5 }
```

```
1 for (int i = 0; i < n; i++) {
2   A[i] = B[i]/(C[i]*D[i]);
3   E[i] = (A[i]*D[i] + B[i]*C[i])/
4                          (B[i]*D[i]);
5 }
```

Consistency of Precision: Constants

1. Operations on type `float` is faster than operations on type `double`. Avoid type conversions and define single-precision literal constants with suffix `-f`.

```
1 const double twoPi = 6.283185307179586;
2 const float phase = 0.3f; // single precision
```

2. Use 32-bit `int` values including 64-bit `long` where possible, including array indices. Avoid type conversions and define 64-bit literal constants with suffix `-L` or `UL`

```
1 const long N2 = 1000000*1000000; // Overflow error
2 const long N3 = 1000000L*1000000L; // Correct
```


Consistency of Precision: Functions

1. `math.h` contains fast single precision versions of arithmetic functions ending with suffix `-f`

```
1 double sin(double x);
2 float sinf(float x);
```

2. `math.h` contains fast base 2 exponential and logarithmic functions:

```
1 double exp(double x); // Double precision, natural base
2 float expf(float x); // Single precision, natural base
3 double exp2(double x); // Double precision, base 2
4 float exp2f(float x); // Single precision, base 2
```


Floating-Point Semantics

The Intel C++ Compiler may represent floating-point expressions in executable code differently, depending on the *floating-point semantics*.

`-fp-model strict`	Only value-safe optimizations
`-fp-model precise`	calculations are reproducible from run to run exceptions controlled using `-fp-model except`
`-fp-model fast=1`	(default) Value-unsafe optimizations are allowed
`-fp-model fast=2`	better performance at the cost of lower accuracy
`-fp-model source`	Intermediate arithmetic results are rounded to the precision defined in the source code.
`-fp-model double`	Intermediate arithmetic results are rounded to 53-bit (double) precision.
`-fp-model extended`	Intermediate arithmetic results are rounded to 64-bit (extended) precision.
`-fp-model [no-]except`	controls floating-point exception semantics.

Precision Control for Transcendental Functions

`-fimf-precision=value[:funclist]` Defines the precision for math functions. `value` is one of: `high`, `medium` or `low`

`-fimf-max-error=ulps[:funclist]` The maximum allowable error expressed in ulps (*units in last place*)

`-fimf-accuracy-bits=n[:funclist]` The number of correct bits required for mathematical function accuracy.

`-fimf-domain-exclusion=n[:funclist]` Defines a list of special-value numbers that do not need to be handled.
`int n` derived by the bitwise OR of types:
extremes: 1, NaNs: 2, infinites: 4, denormals[1]: 8, zeroes: 16.

[1] by default, on Intel Xeon Phi, denormals are flushed to zero in hardware, but supported in SVML

Precision Control for Transcendental Functions

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main() {
5      const int N = 1000000;
6      const int P = 10;
7      double A[N];
8      const double startValue = 1.0;
9      A[:] = startValue;
10     for (int i = 0; i < P; i++)
11 #pragma simd
12         for (int r = 0; r < N; r++)
13             A[r] = exp(-A[r]);
14
15     printf("Result=%.17e\n", A[0]);
16 }
```


Precision Control for Transcendental Functions

```
user@host% icpc -o precision-1 -mmic \
% -fimf-precision=low precision.cc
user@host% scp precision-1 mic0:~/
% precision-1  100%  11KB 11.3KB/s
user@host% ssh mic0 time ./precision-1
Result=5.68428695201873779e-01
real 0m 0.08s
user 0m 0.06s
sys 0m 0.02s
user@host%
```

```
user@host% icpc -o precision-2 -mmic \
% -fimf-precision=high precision.cc
user@host% scp precision-2 mic0:~/
% precision-2  100%  19KB 19.4KB/s
user@host% ssh mic0 time ./precision-2
Result=5.68428725029060722e-01
real 0m 0.14s
user 0m 0.12s
sys 0m 0.02s
user@host%
```

Automatic Vectorization: Making it Happen and Tuning

Challenges with Optimizing Vectorization on Xeon Phi

- **Must utilize 512-bit vector registers (16 `float` or 8 `double`)**
- **Must convince compiler that vectorization is possible**
- Preferably unit-stride access to data
- Preferably align data on 64-byte boundary
- Avoid branches in vector loops
- Guide compiler regarding expected iteration count, memory alignment, outer loop vectorization, etc.

This section:

Ensuring that automatic vectorization succeeds where it must exist.

Diagnosing the Utilization of Vector Instructions

When porting and optimizing an application:

- Find performance-critical parts
- Use `-vec-report3` to get information about automatic vectorization
- Use Intel VTune Amplifier XE to diagnose the executable
- Benchmark regular compilation *vs.* `-no-vec -no-simd` case
- Provide additional information to the compiler about loops in form of `#pragmas`

Assumed Vector Dependence. The `restrict` Keyword.

- True vector dependence makes 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*: when compiler cannot determine wheter vector dependence exists, auto-vectorization fails:

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

```
user@host% icpc -vec-report3 \
                -c vdep.cc
vdep.cc(2): (col. 3) remark:
    loop skipped: multiversioned.
vdep.cc(2): (col. 3) remark:
    loop was not vectorized:
    not inner loop.
```

Ignoring Assumed Vector Dependence

To ignore assumed vector dependence

```
#pragma ivdep
```

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

```
user@host% icpc -vec-report3 \
                -c vdep.cc
vdep.cc(3): (col. 3) remark:
    LOOP WAS VECTORIZED.
vdep.cc(3): (col. 3) remark:
    loop was not vectorized:
    not inner loop.
```

Pointer Disambiguation (alternative to `#pragma ivdep`)

- `restrict` keyword applies to each pointer variable qualified with it
- The object accessed by the pointer is **only accessed by that pointer** in the given scope
- The compiler argument `-restrict` must be used.

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

```
user@host% icpc -vec-report3 -restrict -c vdep.cc
vdep.cc(2): (col. 3) remark: LOOP WAS VECTORIZED.
vdep.cc(2): (col. 3) remark: loop was not vectorized: not inner loop.
```

Automatic Vectorization: Data Structures

Challenges with Optimizing Vectorization on Xeon Phi

- Must utilize 512-bit vector registers (16 `float` or 8 `double`)
- Must convince compiler that vectorization is possible
- **Preferably unit-stride access to data**
- Preferably align data on 64-byte boundary
- Avoid branches in vector loops
- Guide compiler regarding expected iteration count, memory alignment, outer loop vectorization, etc.

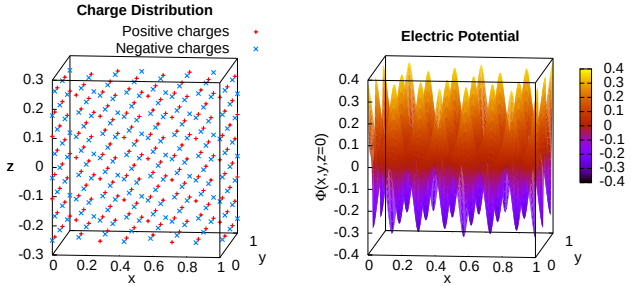
The rule of thumb for achieving unit-stride access

Use structures of arrays (SoA) instead of arrays of structures (AoS)

Example: Unit-Stride Access in Coulomb's Law Application

$$\Phi\left(\vec{R}_j\right) = -\sum_{i=1}^{m} \frac{q_i}{\left|\vec{r}_i - \vec{R}_j\right|}, \tag{1}$$

$$\left|\vec{r}_i - \vec{R}\right| = \sqrt{\left(r_{i,x} - R_x\right)^2 + \left(r_{i,y} - R_y\right)^2 + \left(r_{i,z} - R_z\right)^2}. \tag{2}$$



White paper: research.colfaxinternational.com/post/2012/03/12/AVX.aspx

Elegant, but Inefficient Solution: Array of Structures

```
struct Charge {                    // Elegant, but ineffective data layout
    float x, y, z, q;
    } chgs[m]; // Coordinates and value of this charge
```

```
for (int i=0; i<m; i++)  { // This loop will be auto-vectorized
    // Non-unit stride: (&chg[i+1].x - &chg[i].x) != sizeof(float)
    const float dx=chg[i].x - Rx;
    const float dy=chg[i].y - Ry;
    const float dz=chg[i].z - Rz;
    phi -= chg[i].q / sqrtf(dx*dx+dy*dy+dz*dz); // Coulomb's law
}
```

Arrays of Structures versus Structures of Arrays

Array of Structures (AoS)

```
struct Charge { // Elegant, but ineffective data layout
    float x, y, z, q; // Coordinates and value of this charge
};
// The following line declares a set of m point charges:
Charge chg[m];
```

Structure of Arrays (SoA)

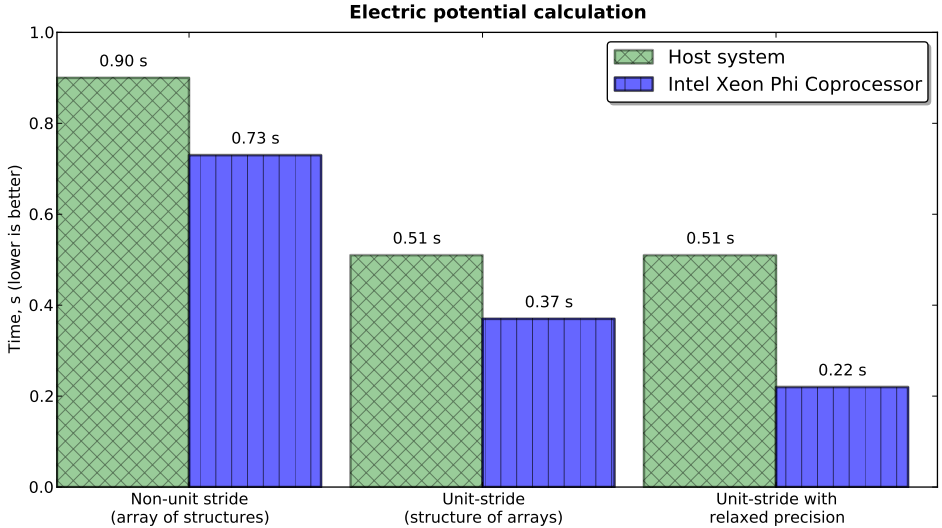
```
struct Charge_Distribution {
  // Data layout permits effective vectorization of Coulomb's law application
  const int m; // Number of charges
  float * x; // Array of x-coordinates of charges
  float * y; // ...y-coordinates...
  float * z; // ...etc.
  float * q; // These arrays are allocated in the constructor
};
```

Optimized Solution: Structure of Arrays, Unit-Stride Access

```
struct Charge_Distribution {
  // Data layout permits effective vectorization of Coulomb's law application
  const int m; // Number of charges
  float *x, *y, *z, *q; // Arrays of x-, y- and z-coordinates of charges
};
```

```
// This version vectorizes better thanks to unit-stride data access
for (int i=0; i<chg.m; i++)  {
  // Unit stride: (&chg.x[i+1] - &chg.x[i]) == sizeof(float)
  const float dx=chg.x[i] - Rx;
  const float dy=chg.y[i] - Ry;
  const float dz=chg.z[i] - Rz;
  phi -= chg.q[i] / sqrtf(dx*dx+dy*dy+dz*dz);
}
```

Electric Potential Calculation with Coulomb's Law



Automatic Vectorization: Data Alignment

Challenges with Optimizing Vectorization on Xeon Phi

- Must utilize 512-bit vector registers (16 `float` or 8 `double`)
- Must convince compiler that vectorization is possible
- Preferably unit-stride access to data
- **Preferably align data on 64-byte boundary**
- Avoid branches in vector loops
- **Guide compiler regarding expected iteration count, memory alignment, outer loop vectorization, etc.**

This section:

Data alignment and compiler hints.

Data Alignment

- `char* p` points to an address aligned on an `n`-byte boundary if `((size_t)p%n==0)`.
- 128-bit SSE load and store instructions require 16-byte alignment,
- 256-bit AVX load and store instructions do not require alignment,
- 512-bit IMCI load and store instructions require 64-byte alignment.

Data Alignment

- Data alignment on the stack

```
1 float A[n] __attribute__((aligned(64))); // 64-byte alignment applied
```

  - The address of `A[0]` is a multiple of 64, *i.e.*, aligned on a 64-byte boundary.
  - Setting a very high alignment value may lead to wasted virtual memory.

- Alignment of memory blocks on the heap

```
1 #include <malloc.h>
2 // ...
3 float *A = (float*)_mm_malloc(n*sizeof(float), 16);
4 // ...
5 _mm_free(A);
```

  - `_mm_malloc` and `_mm_free` are aligned version of `malloc` and `free`:
  - the header file `malloc.h` must be included

Data Alignment Hints

Programmer may promise to the compiler (under penalty of segmentation fault) than alignment has been taken care of:

```
1  float* packedData = _mm_malloc(sizeof(float)*nData, 64);
2  float* inVector = _mm_malloc(sizeof(float)*nRows, 64);
3  // ... Pragma vector aligned promises to the compiler that elements of array
4  // used in the first iteration are 64-byte boundary aligned.
5  #pragma vector aligned
6    for (int c = 0; c < blockLen[idx]; c++) // blockLen[idx] are multiples of 64
7      sum += packedData[offs+c]*inVector[j0+c];
8    outVector[i] += sum;
9  // ...
10 _mm_free(packedData); _mm_free(inVector);
```

This can lead to significant speedups, because compiler will not implement runtime checks for alignment situation.

Data Alignment and Padding

Note: when relying on `#pragma vector aligned`, may need to pad the inner dimension on data structures to a multiple of 16 (in single precision) or 8 (double precision).

```
1  void GaussEl(const int n, const int m, const int start, float* const matrix) {
2    for (int i = start+1; i < n; i++) {
3      const float factor = matrix[(i-1)*m]/matrix[i*m];
4  #pragma vector aligned
5      for (int j = 0; j < m; j++)
6        matrix[i*m + j] += factor*matrix[(i-1)*m + j];
7    }
8    // ... Padding inner dimension and allocating matrix
9    if (m % 16 != 0) m += (16 - m%16);
10   matrix = (float*)_mm_malloc(n*m*sizeof(float), 64);
11   //...
12   GaussEl(n, m, 0, matrix);
```


Vectorization Pragmas, Keywords and Compiler Arguments

- `#pragma simd`
- `#pragma vector always`
- `#pragma vector aligned | unaligned`
- `#pragma vector nontemporal | temporal`
- `#pragma novector`
- `#pragma ivdep`
- `restrict` qualifier and `-restrict` command-line argument
- `#pragma loop count`
- `__assume_aligned` keyword
- `-vec-report[n]`
- `-O[n]`
- `-x[code]`

Vectorization Pragmas, Keywords and Compiler Arguments

- `#pragma simd`
- `#pragma vector always`
- `#pragma vector aligned | unaligned`
- `#pragma vector nontemporal | temporal`
- `#pragma novector`
- `#pragma ivdep`
- `restrict` qualifier and `-restrict` command-line argument
- `#pragma loop count`
- `__assume_aligned` keyword
- `-vec-report[n]`
- `-O[n]`
- `-x[code]`

Thread Parallelism: Reducing Synchronization

Challenges with Thread Parallelism on Xeon Phi

- Multi-core CPU: 4–48 threads, Xeon Phi: 228–244 threads.
- Must have enough parallelism to keep all cores busy
- Must have less synchronization than on CPU
- Must have lower per-thread memory overhead
- Must access core-local data whenever possible
- Must co-exist with vectorization in each core

Example: Dealing with Excessive Synchronization

Computing a histogram ($m << n$) with a serial code:

```
1 void Histogram(const float* age, int* const hist, const int n,
2                const float group_width, const int m) {
3     for (int i = 0; i < n; i++) {
4         const int j = (int) ( age[i] / group_width );
5         hist[j]++;
6     }
7 }
```

- Code cannot be automatically vectorized
- True vector dependence

The Same Calculation, Strip-Mined, Vectorized

```
1  void Histogram(const float* age, int* const hist, const int n,
2                 const float group_width, const int m) {
3      const int vecLen = 16; // Length of vectorized loop
4      const float invGroupWidth = 1.0f/group_width; // Pre-compute the reciprocal
5      // Strip-mining the loop in order to vectorize the inner short loop
6      // Note: this algorithm assumes n%vecLen == 0.
7      for (int ii = 0; ii < n; ii += vecLen) { //Temporary store vecLen indices
8          int histIdx[vecLen] __attribute__((aligned(64)));
9          // Vectorize the multiplication and rounding
10 #pragma vector aligned
11         for (int i = ii; i < ii + vecLen; i++)
12             histIdx[i-ii] = (int) ( age[i] * invGroupWidth );
13         // Scattered memory access, does not get vectorized
14         for (int c = 0; c < vecLen; c++)
15             hist[histIdx[c]]++;
16   }
17 }
```

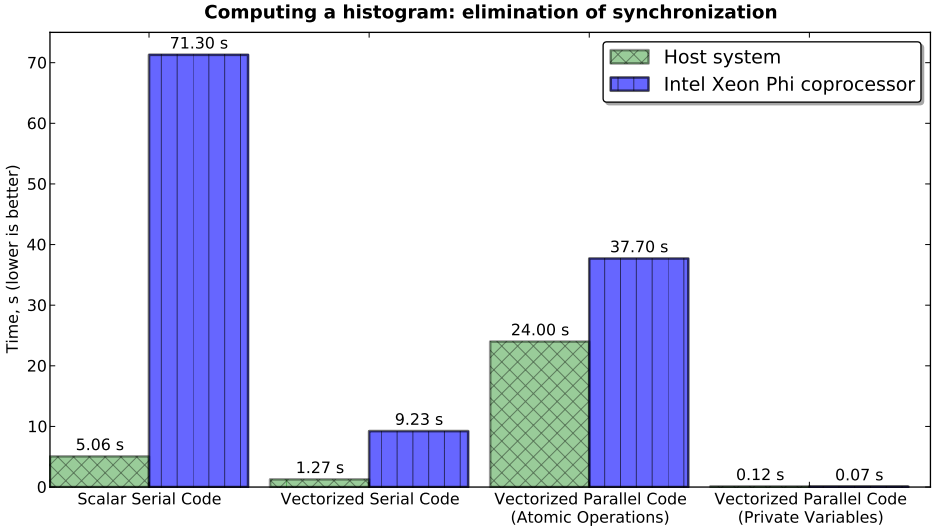
Adding Thread Parallelism

```
#pragma omp parallel for schedule(guided)
    for (int ii = 0; ii < n; ii += vecLen) {
        int histIdx[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
        for (int i = ii; i < ii + vecLen; i++)
            histIdx[i-ii] = (int) ( age[i] * invGroupWidth );
        for (int c = 0; c < vecLen; c++)
            // Protect the ++ operation with the atomic mutex (inefficient!)
#pragma omp atomic
            hist[histIdx[c]]++;
  }
}
```


Improving Thread Parallelism

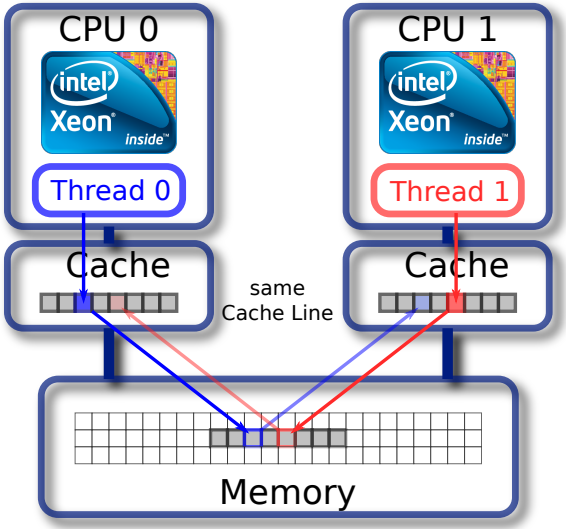
```
#pragma omp parallel
    {
    int hist_priv[m]; // Better idea: thread-private storage
    hist_priv[:] = 0;
    int histIdx[vecLen] __attribute__((aligned(64)));
#pragma omp for schedule(guided)
    for (int ii = 0; ii < n; ii += vecLen) {
#pragma vector aligned
        for (int i = ii; i < ii + vecLen; i++)
            histIdx[i-ii] = (int) ( age[i] * invGroupWidth );
        for (int c = 0; c < vecLen; c++)
            hist_priv[histIdx[c]]++;
    }
    for (int c = 0; c < m; c++) {
#pragma omp atomic
        hist[c] += hist_priv[c];
}   }   }
```


Dealing with Excessive Synchronization



Thread Parallelism: False Sharing

False Sharing. Data Padding and Private Variables



- *False sharing* is similar to *race condition*
- Threads accessing the same *cache line*
- Caused by *coherent caches*
- Cache line is 64-byte wide (in modern Intel architectures)

False Sharing. Data Padding and Private Variables

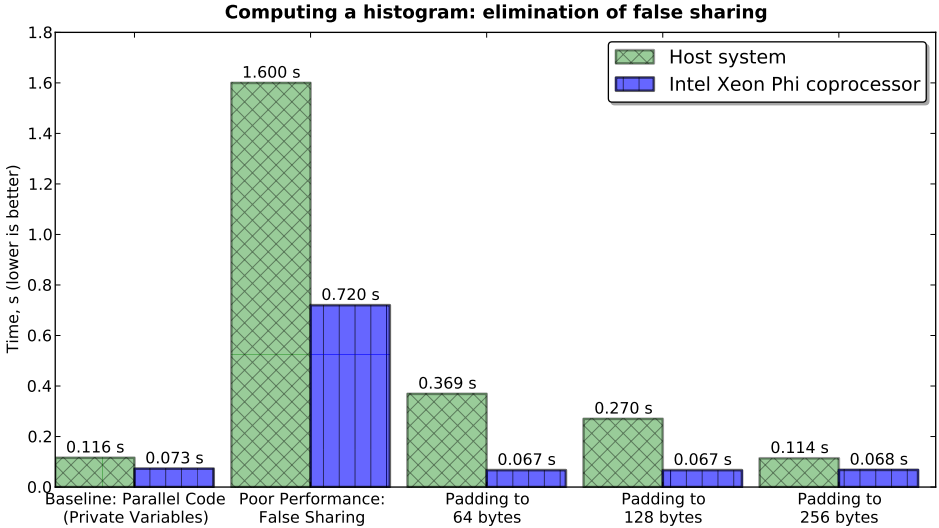
```
1     const int m = 5;
2     int hist_thr[nThreads][m];
3  #pragma omp parallel for
4     for (int ii = 0; ii < n; ii += vecLen) {
5         // False sharing occurs here
6         for (int c = 0; c < vecLen; c++)
7             hist_thr[iThread][histIdx[c]]++;
8     }
9    // Reducing results from all threads to the common histogram hist
10    for (int iThread = 0; iThread < nThreads; iThread++)
11        hist[0:m] += hist_thr[iThread][0:m];
```

- The value of `m=5` is small
- Array elements `hist_thr[0][:]` are within `m*sizeof(int)=20` bytes of array elements `hist_thr[1][:]`

Padding to Avoid False Sharing

```
1    // Padding for hist_thr[][] in order to avoid a situation
2    // where two (or more) rows share a cache line.
3    const int paddingBytes = 64;
4    const int paddingElements = paddingBytes / sizeof(int);
5    const int mPadded = m + (paddingElements-m%paddingElements);
6    // Shared histogram with a private section for each thread
7    int hist_thr[nThreads][mPadded];
8    hist_thr[:][:] = 0;
```


Padding to Avoid False Sharing



Thread Parallelism: Expanding Iteration Space

Example: Dealing with Insufficient Parallelism

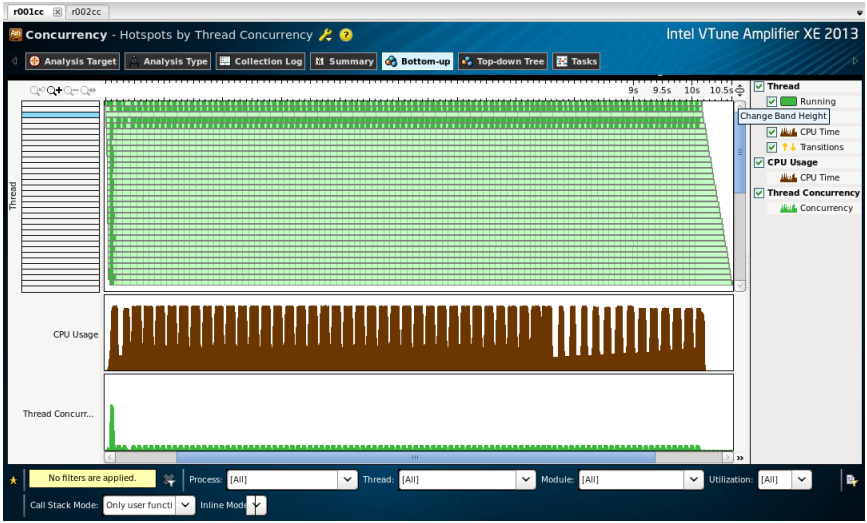
$$S_i = \sum_{j=0}^{n} M_{ij},\ i = 0 \ldots m. \tag{3}$$

- `m` is small, smaller than the number of threads in the system
- `n` is large, large enough so that the matrix does not fit into cache

```
1 void sum_unoptimized(const int m, const int n, long* M, long* s){
2 #pragma omp parallel for
3     for (int i=0; i<m; i++) {
4         long sum=0;
5 #pragma simd
6 #pragma vector aligned
7         for (int j=0; j<n; j++)
8             sum+=M[i*n+j];
9         s[i]=sum; }}
```


Dealing with Insufficient Parallelism

VTune Analysis: Row-Wise Reduction of a Short, Wide Matrix



Strip-Mining: Simultaneous Thread and Data Parallelism

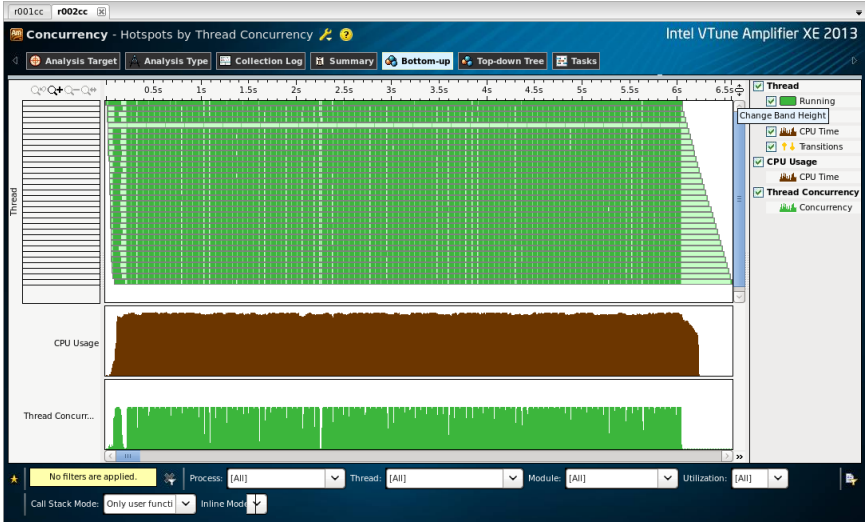
```
// Compiler may be able to simultaneously parallelize and auto-vectorize it
#pragma omp parallel for
#pragma simd
for (int i = 0; i < n; i++) {
    // ... do work
}
```

```
// The strip-mining technique separates parallelization from vectorization
const int STRIP=1024;
#pragma omp parallel for
for (int ii = 0; ii < n; ii += STRIP)
#pragma simd
    for (int i = ii; i < ii + STRIP; i++) {
        // ... do work
    }
```


Exposing Parallelism: Strip-Mining and Loop Collapse

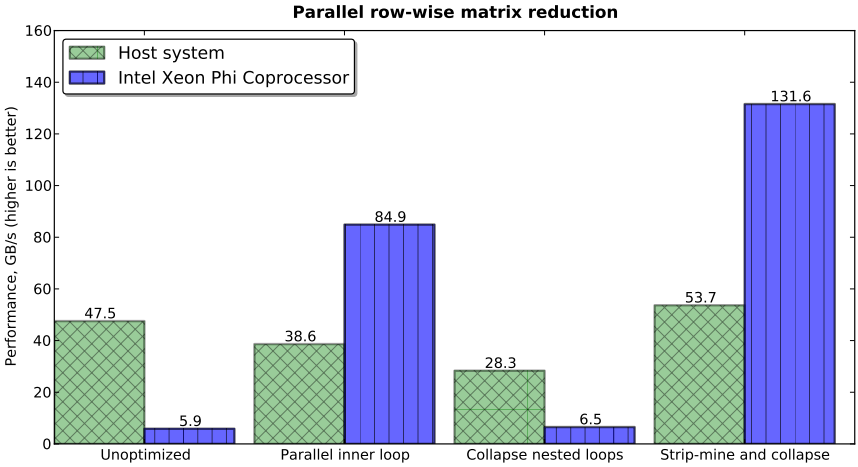
```
1  void sum_stripmine(const int m, const int n, long* M, long* s){
2      const int STRIP=1024;
3      assert(n%STRIP==0);
4      s[0:m]=0;
5  #pragma omp parallel
6      {
7          long sum[m];    sum[0:m]=0;
8  #pragma omp for collapse(2) schedule(guided)
9          for (int i=0; i<m; i++)
10             for (int jj=0; jj<n; jj+=STRIP)
11 #pragma simd
12 #pragma vector aligned
13                 for (int j=jj; j<jj+STRIP; j++)
14                     sum[i]+=M[i*n+j];
15         for (int i=0; i<m; i++)           // Reduction
16 #pragma omp atomic
17             s[i]+=sum[i];
18 }   }
```


Exposing Parallelism: Strip-Mining and Loop Collapse



Dealing with Insufficient Parallelism

Row-Wise Reduction of a Short, Wide Matrix



Thread Parallelism: Affinity

Setting Thread Affinity

- OpenMP threads may migrate from one core to another according to OS decisions.
- Forbid migration — increase the performance.
- Control: environment variable `KMP_AFFINITY`

Uses of Thread Affinity

- Bandwidth-bound applications: 1 thread per core + prevent migration. Optimizes utilization of memory controllers.
- Compute-bound applications: 2 (Xeon) or 4 (Xeon Phi) threads per core + prevent migration. Ensures that threads consistently access local L1 cache data (+L2 for Xeon Phi).
- Offload applications : physical core 0 on Xeon Phi is used by μOS for offload tasks. Prevent placing compute threads on that core.
- Aplications in multi-socket NUMA (Non-Uniform Memory Access) systems: partition the system for two independent tasks, pin tasks to respective CPUs.

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=[<pr
- type=disabled or none.

```
user@host% export MIC_ENV_PREFIX=MIC
user@host% export KMP_AFFINITY=compact,granularity=fine
user@host% export MIC_KMP_AFFINITY=balanced,granularity=fine
```


Bandwidth-bound, KMP_AFFINITY=scatter

```
user@host% export OMP_NUM_THREADS=32
user@host% export KMP_AFFINITY=none
user@host% 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)
user@host% export OMP_NUM_THREADS=16
user@host% export KMP_AFFINITY=scatter
user@host% 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)
```


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

```
user@host% icpc -o bench-dgemm -mkl -mmic bench-dgemm.cc
user@host% 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

user@host% 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
```

Other Optimization Topics for Thread Parallelism

Examples found in our 4-day training and in the book:

- Avoiding excessive synchronization with reduction
- Load balancing across threads
- Using thread affinity to partition a multi-socket NUMA system

§6. Advanced Optimization for the MIC Architecture

Memory Access and Cache Utilization

Challenges with Memory Access on Xeon Phi

- More threads than CPU, same amount of Level-2 cache (~30 MB)
- No hardware prefetching from Level-2 to Level-1
- High penalty for data page walks
- Dynamic memory allocation is serial → greater penalty than CPU per Amdahl's law

"Rule of Thumb" for memory optimization: locality of data access in space and in time.

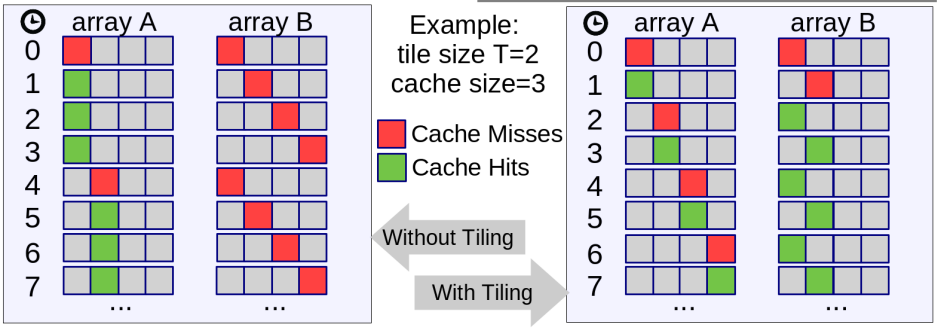
Spatial locality = data structures (packing, reordering).

Temporal locality = order of operations (e.g., loop tiling).

Loop Tiling (Blocking)

```
/* Nested loops without tiling.
Array B[] does not fit into cache */
for (int i = 0; i < iMax; ++i)
 for (int j = 0; j < jMax; ++j)
  PerformWork(A[i], B[j]);
```

```
/* Tiled nested loops */
for (int jj = 0; jj < jMax; jj+= T)
 for (int i = 0; i < iMax; ++i)
  for (int j = jj; j < jj+T; ++j)
   PerformWork(A[i], B[j]);
```



Cache Hit Rate = 6/16

SLOWER

Cache Hit Rate = 10/16

FASTER

Loop Tiling (Blocking)

```
// Plain nested loops
for (int i = 0; i < m; i++)
  for (int j = 0; j < n; j++)
    compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
// Tiled nested loops
for (int ii = 0; ii < m; ii += TILE)
  for (int j = 0; j < n; j++)
    for (int i = ii; i < ii + TILE; i++) //Re-use data for each j with several i
      compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
// Doubly tiled nested loops
for (int ii = 0; ii < m; ii += TILE)
  for (int jj = 0; jj < n; jj += TILE)
    for (int i = ii; i < ii + TILE; i++) //Re-use data for each j with several i
      for (int j = jj; j < jj + TILE; j++)
        compute(a[i], b[j]); // Memory access is unit-stride in j
```

Optimization Example: In-Place Square Matrix Transposition

```
1 #pragma omp parallel for
2   for (int i = 0; i < n; i++) { // Distribute across threads
3     for (int j = 0; j < i; j++) { // Employ vector load/stores
4       const double c = A[i*n + j]; // Swap elements
5       A[i*n + j] = A[j*n + i];
6       A[j*n + i] = c;
7     }
8   }
```

Unoptimized code:

- Large-stride memory accesses
- Inefficient cache use
- Does not reach memory bandwidth limit

Tiling a Parallel For-Loop (Matrix Transposition)

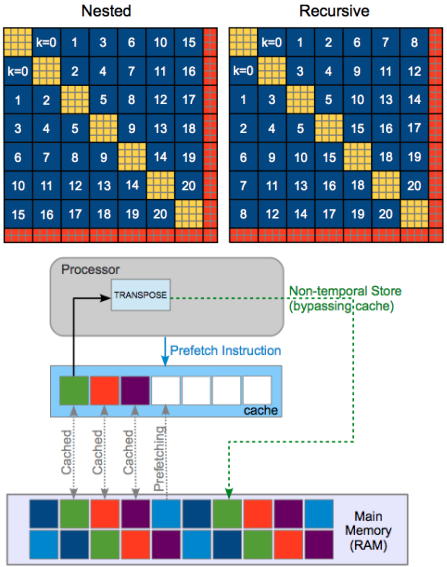
```
#pragma omp parallel for
  for (int ii = 0; ii < n; ii += TILE) { // Distribute across threads
    const int iMax = (n < ii+TILE ? n : ii+TILE); // Adapt to matrix shape
    for (int jj = 0; jj <= ii; jj += TILE)  { // Tile the work
      for (int i = ii; i < iMax; i++) { // Universal microkernel
        const int jMax = (i < jj+TILE ? i : jj+TILE); // for whole matrix
#pragma loop count avg(TILE) // Vectorization tuning
#pragma simd // Vectorization hint
        for (int j = jj; j<jMax; j++) { // Variable loop count (bad)
          const double c = A[i*n + j]; // Swap elements
          A[i*n + j] = A[j*n + i];
          A[j*n + i] = c;
    } } } }
```

Better (but not optimal) solution:

- Loop tiling to improve locality of data access
- Not enough outer loop iterations to keep 240 threads busy

Further Optimization of Matrix Transposition

- Multi-versioned inner loop for diagonal, edges and body
- Tuning pragma to enforce non-temporal stores
- Expand parallel iteration space occupy all threads
- Control data alignment
- OpenMP thread affinity for bandwidth optimization



Further Optimization: Code Snippet

```
#pragma omp parallel
  {
#pragma omp for schedule(guided)
    for (int k = 0; k < nTilesParallel; k++) { // Bulk of calculations here
      const int ii = plan[HEADER_OFFSET + 2*k + 0]*TILE; // Planned order
      const int jj = plan[HEADER_OFFSET + 2*k + 1]*TILE; // of operations
      for (int j = jj; j < jj+TILE; j++) { // Simplified main microkernel
#pragma simd // Vectorization hint
#pragma vector nontemporal // Cache traffic hint
        for (int i = ii; i < ii+TILE; i++) { // Constant loop count (good)
          const double c = A[i*n + j]; // Swap elements
          A[i*n + j] = A[j*n + i];
          A[j*n + i] = c;
        } } }
    // Transposing the tiles along the main diagonal and edges...
    // ...
```

- Longer code but still in the C language; works for CPU and MIC

Arithmetic Intensity and Roofline Model

Theoretical estimates, Intel Xeon Phi coprocessor

$$\text{Arithmetic Performance} = 60 \times 1.0 \times (512/64) \times 2 = 960\,\text{GFLOP/s}.$$

$$\text{Memory Bandwidth} = \eta \times 6.0 \times 8 \times 2 \times 4 = \eta \times 384\,\text{GB/s},$$

Peak performance for:

- 60-core Intel Xeon Phi
- clocked at 1.0 GHz
- 512-bit SIMD registers
- 64-bit floating-point numbers
- fused multiply-add

The peak memory bandwidth:

- $\eta \approx 0.5$ – practical efficiency
- 6.0 GT/s (Transfers)
- 8 memory controllers
- 2 channels in each
- 4 bytes per channel

Arithmetic Intensity and Roofline Model

Theoretical estimates, Intel Xeon Phi coprocessor

$$\text{Arithmetic Performance} = 60 \times 1.0 \times (512/64) \times 2 = 960\,\text{GFLOP/s}.$$

$$\text{Memory Bandwidth} = \eta \times 6.0 \times 8 \times 2 \times 4 = \eta \times 384\,\text{GB/s},$$

To saturate Arithmetic and Logic Units (ALUs):

384/8 = 48 billion floating-point numbers per second should be delivered from memory to the cores (double precision)

960/48 = 20 floating-point operations (multiplication/addition) must be performed on every number fetched from the main memory

Arithmetic Intensity and Roofline Model

Theoretical estimates, 2x 8-core Intel Xeon E5 processors at 3.0 GHz

$$\text{Arithmetic Performace} = 2 \text{ sockets} \times 8 \times 3.0 \times (256/64) \times 2 = 384 \text{ GFLOP/s},$$

$$\text{Memory Bandwidth} = 2 \text{ sockets} \times \eta \times 6.4 \times 8 = \eta \times 102 \text{ GB/s},$$

Peak performance for:

- 16 Intel Xeon cores
- clocked at 3.0 GHz
- 256-bit SIMD registers
- 64-bit floating-point numbers
- 2 ALUs

The peak memory bandwidth:

- $\eta \approx 0.5$ – practical efficiency
- 6.4 GT/s (Transfers)
- 8 bytes per transfer

Arithmetic Intensity and Roofline Model

Theoretical estimates, 2x 8-core Intel Xeon E5 processors at 3.0 GHz

$$\text{Arithmetic Performace} = 2 \text{ sockets} \times 8 \times 3.0 \times (256/64) \times 2 = 384 \text{ GFLOP/s},$$

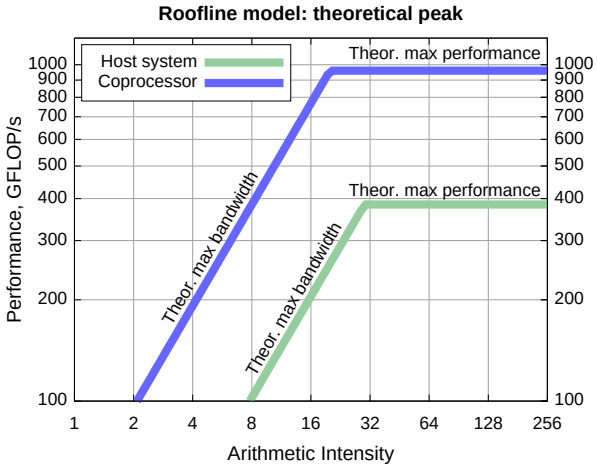
$$\text{Memory Bandwidth} = 2 \text{ sockets} \times \eta \times 6.4 \times 8 = \eta \times 102 \text{ GB/s},$$

To saturate Arithmetic and Logic Units (ALUs):

$102/8 \approx 13$ billion floating-point numbers per second should be delivered from memory to the cores (double precision)

$384/13 = 30$ floating-point operations (multiplication/addition) must be performed on every number fetched from the main memory

Arithmetic Intensity and Roofline Model



More on roofline model: Williams et al.

Other Topics on Memory Traffic Optimization

Discussions found in our 4-day training and in the book:

- Recursive cache-oblivious algorithms
- Cross-procedural loop fusion
- Software prefetching

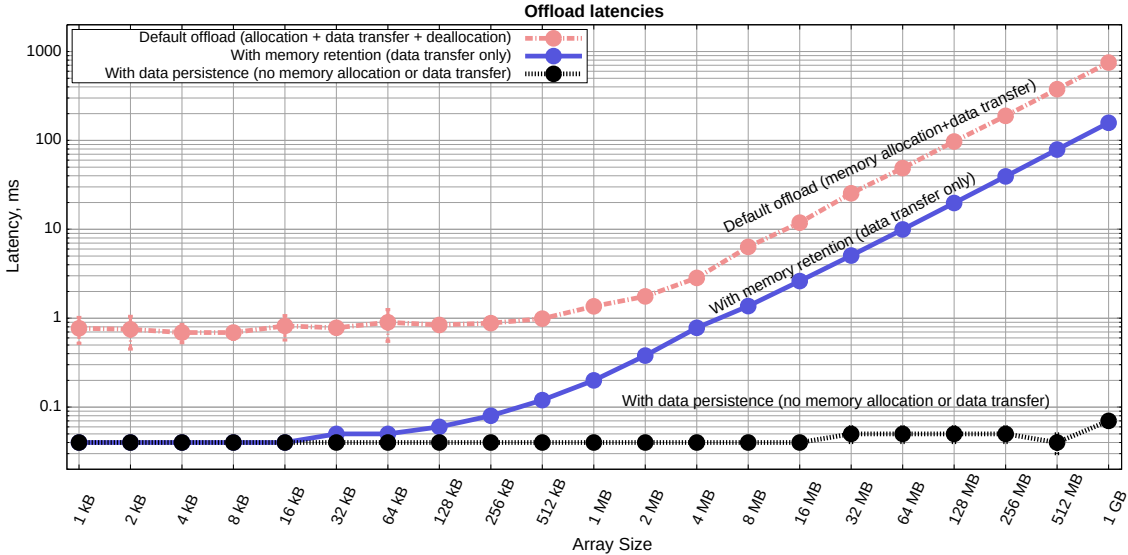
Data Persistence and PCIe Traffic

Memory Retention Between Offloads

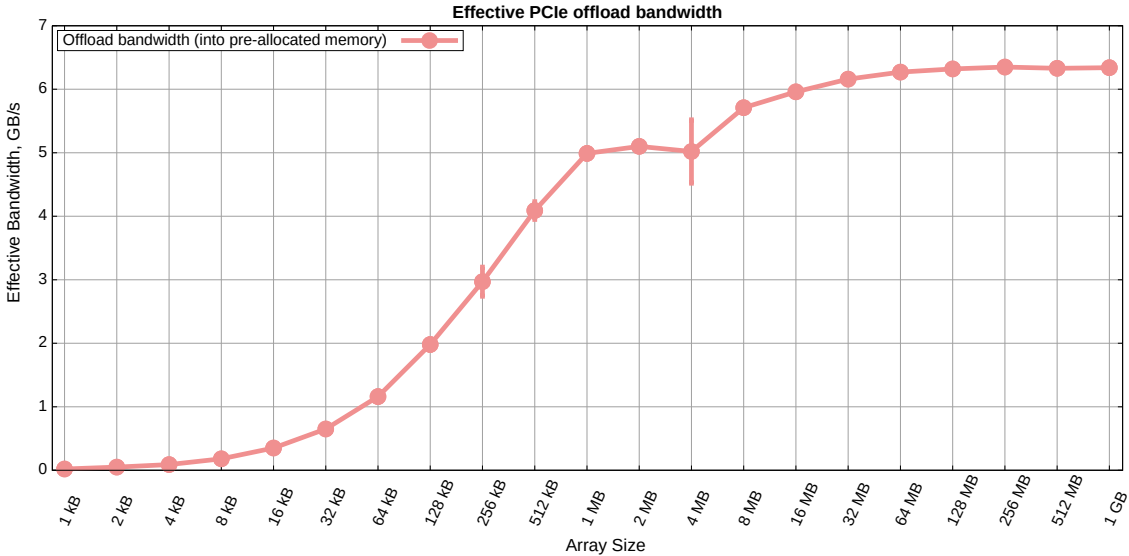
```
1   // Allocate arrays on coprocessor during the first iteration;
2   // retain allocated memory for subsequent iterations
3 #pragma offload target(mic:0) \
4   in(data: length(size) alloc_if(k==0) free_if(k==nTrials-1) align(64))
5   {
6       // offloaded code here...
7   }
```

- Data transfer across the PCIe bus rate is 6 GB/s
- To allocate memory on the coprocessor – 0.5 GB/s
- The memory allocation operation is serial and therefore slow
- Memory retention reduces the latency by a factor of 10x
- For smaller arrays, the effect is even more dramatic

Offload Latency With and Without Memory/Data Retention



Memory Alignment and TLB Page Size Control

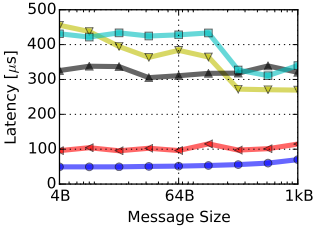


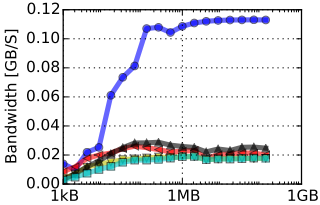
MPI Applications on Clusters with Coprocessors

MPI: Fabrics

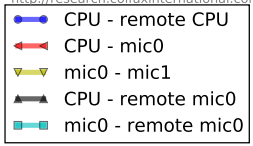
MPI Fabric Selection: Ethernet and InfiniBand

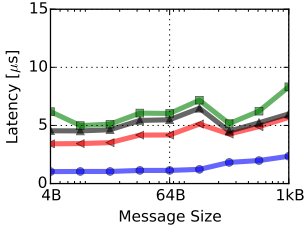
- Ethernet+TCP between coprocessors slower than the hardware limit
- InfiniBand approaches the hardware limit from CPU to coprocessors

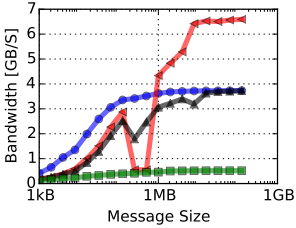




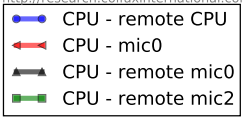
http://research.colfaxinternational.com/





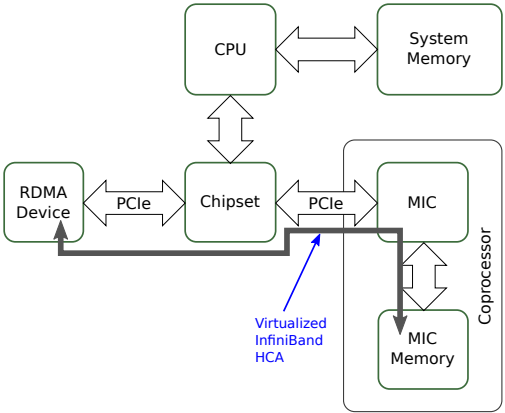


http://research.colfaxinternational.com/



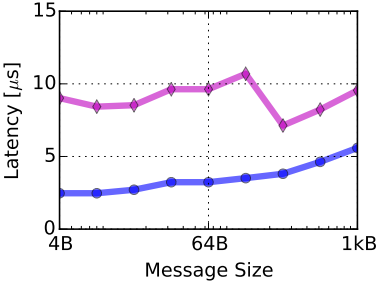
MPI Fabric Selection: Ethernet and InfiniBand

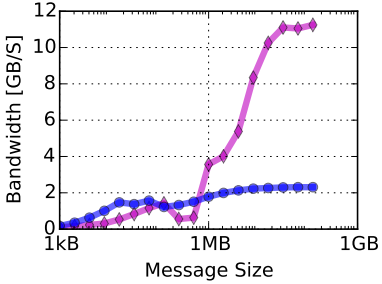
- InfiniBand requires additional software on top of MPSS
- Environment variable `I_MPI_FABRICS`
- More information in white paper



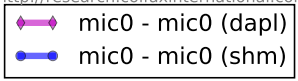
MPI Fabric Selection: Intra-Device Fabric

- Part of CCL: virtual interface `ibscif` for communication between coprocessors within a system
- Default Combination: `I_MPI_FABRICS=shm:dapl`
- `shm` provides better latency, `dapl` – greater bandwidth



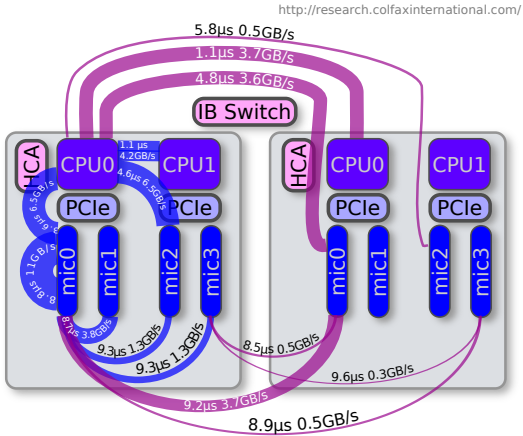


http://research.colfaxinternational.com/



Communication Efficiency with Symmetric Clustering

- MPI communication between CPU and coprocessors as efficient as offload
- Peer-to-peer communication not uniform, but better than with Gigabit Ethernet



White paper with details:
http://research.colfaxinternational.com/post/2014/03/11/InfiniBand-for-MIC.aspx

Process Parallelism: MPI Optimization Strategies

- Dynamic scheduling
- Load balancing
- Communication-efficient algorithms
- OpenMP/MPI hybrid

The Monte Carlo Method of Computing the Number π

$$A_{\text{quarter circle}} = \frac{1}{4}\pi R^2$$

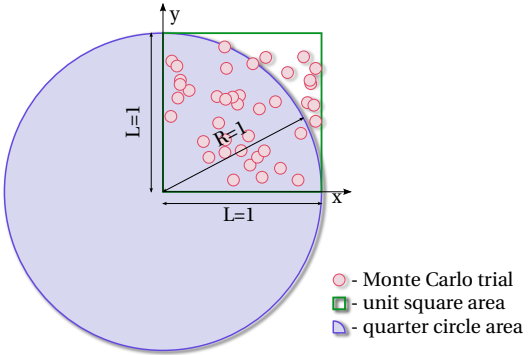
$$A_{\text{square}} = L^2.$$

$$\langle N_{\text{quarter circle}} \rangle = \frac{A_{\text{quarter circle}}}{A_{\text{square}}} N.$$

$$4\frac{\langle N_{\text{quarter circle}} \rangle}{N} = 4\frac{\pi R^2}{4L^2} = \pi.$$

$$\pi \approx 4\frac{N_{\text{quarter circle}}}{N}.$$

$\pi = 3.141592653589793\ldots$



The Monte Carlo Method of Computing the Number π

```
#include <mkl_vsl.h>
const long BLOCK_SIZE=4096;

// Random number generator from MKL
VSLStreamStatePtr stream;
vslNewStream( &stream, VSL_BRNG_MT19937,  seed );

for (long j = 0; j < nBlocks; j++) {
    vsRngUniform( 0, stream, BLOCK_SIZE*2, r, 0.0f, 1.0f );
    for (i = 0; i < BLOCK_SIZE; i++) {
        const float x = r[i];
        const float y = r[i+BLOCK_SIZE];
        if (x*x + y*y < 1.0f) dUnderCurve++;
    }
}
const double pi = (double)dUnderCurve / (double)iter * 4.0
```


The Monte Carlo Method of Computing the Number π

```
1     int rank, nRanks, trial;
2     MPI_Init(&argc, &argv);
3     MPI_Comm_size(MPI_COMM_WORLD, &nRanks);
4     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5
6     const double blocksPerProc = (double)nBlocks / (double)nRanks;
7     const long myFirstBlock = (long)(blocksPerProc*rank);
8     const long myLastBlock  = (long)(blocksPerProc*(rank+1));
9
10    RunMonteCarlo(myFirstBlock, myLastBlock, stream, dUC);
11    // Compute pi
12    MPI_Reduce(&dUC, &UnderCurveSum, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
13    if (rank==0)
14        const double pi = (double)UnderCurveSum / (double) iter * 4.0 ;
15
16    MPI_Barrier(MPI_COMM_WORLD);
17    MPI_Finalize();
```

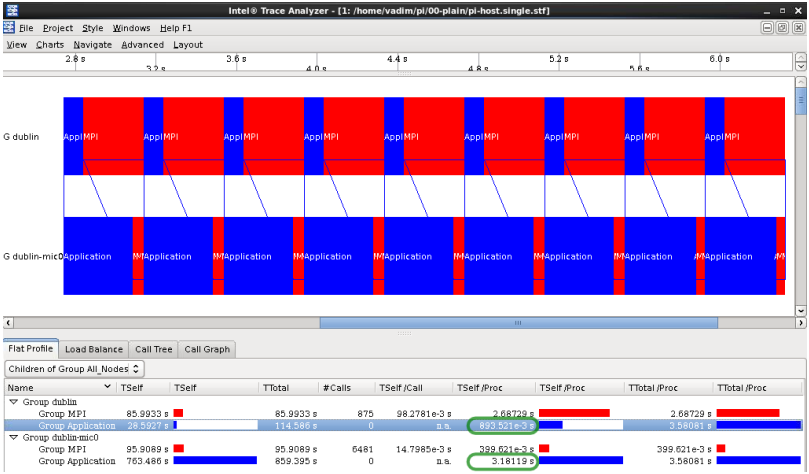

The Monte Carlo Method of Computing the Number π

Host, coprocessor, heterogeneous

```
user@host% mpirun -np 32 -host localhost ./pi_mpi
Time, s:    0.84
user@host% mpirun -np 240 -host mic0 ~/pi_mpi
Time, s:    0.44
user@host% mpirun -np 32 -host localhost ./pi_mpi : -np 240 -host mic0 ~/pi_mpi
Time, s:    0.36
```

- Coprocessor is 1.9x faster than the host system
- $T_{\text{host}} \approx 0.84$ seconds, $T_{\text{Phi}} \approx 0.44$ seconds
- Expect $T_{\text{both}} = 1/(1/0.84 + 1/0.44) \approx 0.29$ seconds
- $T_{\text{measured}} \approx 0.36$ seconds, which is 25% worse than expected. Why?

Using Intel Trace Analyzer and Collector



CPU finishes its share of work faster than coporocessors.

Load Balancing with Static Scheduling

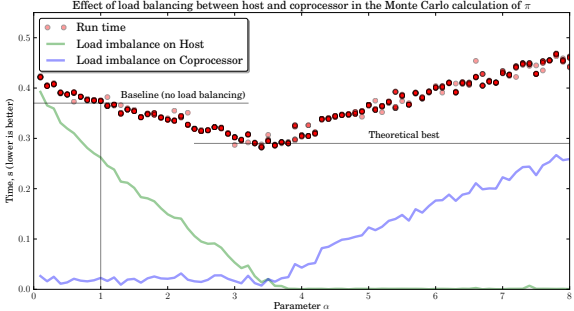
Solution: assign more work to CPU ranks.

$$\alpha = \frac{b_{\text{host}}}{b_{\text{MIC}}},$$

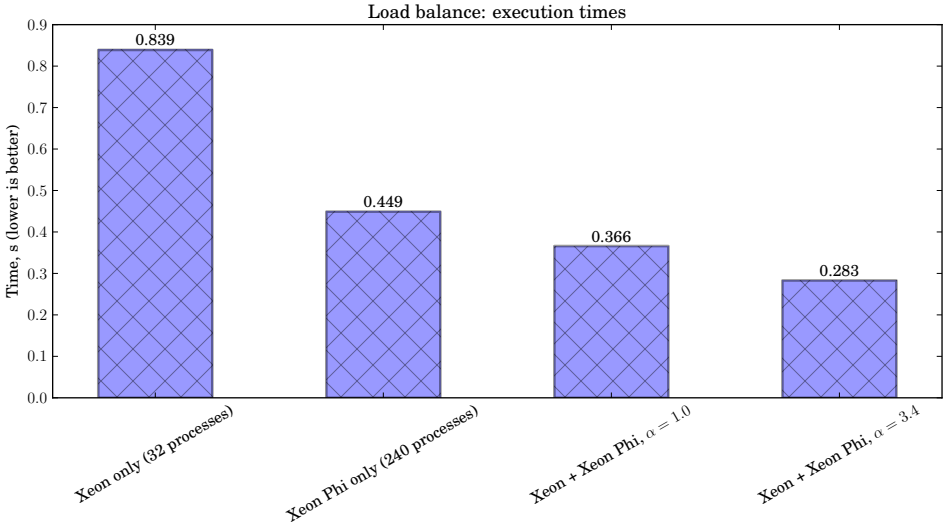
$$B_{\text{total}} = b_{\text{host}} P_{\text{host}} + b_{\text{MIC}} P_{\text{MIC}},$$

$$b_{\text{host}} = \frac{B_{\text{total}}}{\alpha P_{\text{host}} + P_{\text{MIC}}},$$

$$b_{\text{MIC}} = \frac{\alpha B_{\text{total}}}{\alpha P_{\text{host}} + P_{\text{MIC}}}.$$



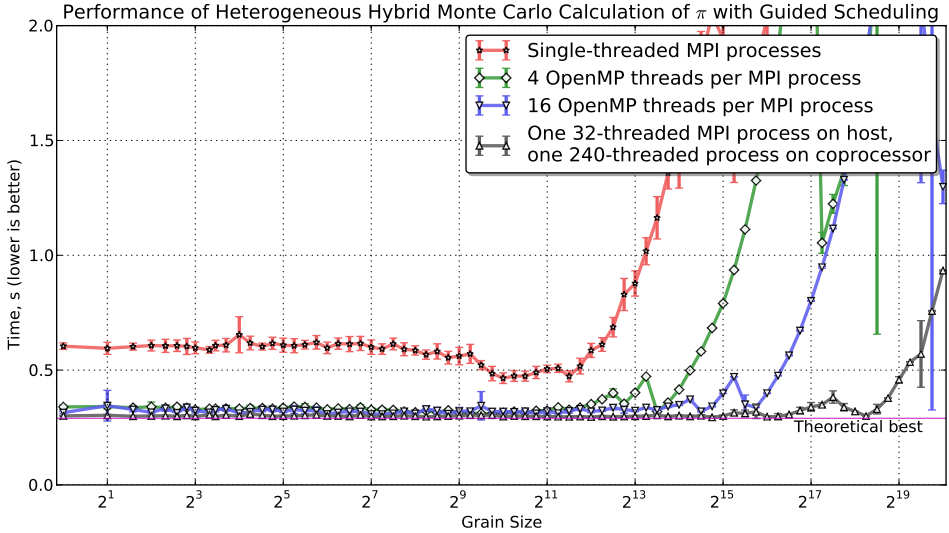
Load Balancing with Static Scheduling



Dynamic and Guided Scheduling

- An alternative, self-tuning way to balance the load on a heterogeneous system is “boss-worker” scheduling
- One MPI process is designated as “boss”, others as “workers”
- Boss assigns tasks to workers as they complete previous tasks
- Intel MPI uses pinning by default. The boss worker may have to be unpinned.
- Dynamic scheduling may have a significant communication overhead
- Reduce overhead with guided scheduling (gradually decreasing task size) and hybrid OpenMP+MPI solutions to have fewer ranks

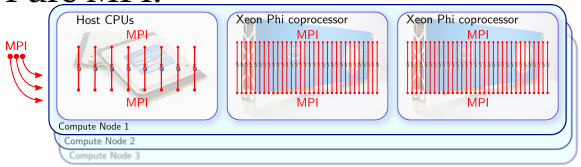
Guided Scheduling



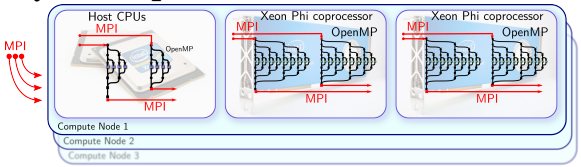
Hybrid MPI+OpenMP

Using OpenMP inside of MPI processes

Pure MPI:



Hybrid OpenMP+MPI:



- Reduces the memory footprint
- Decreases the number of MPI ranks, which reduces communication
- May incur thread synchronization overhead
- Optimal number of threads in MPI processes must be established empirically

Hybrid MPI+OpenMP

- Common usage model: each MPI process is multi-threaded
- Use `I_MPI_PIN_DOMAIN=openmp`
- Set `OMP_NUM_THREADS` for each MPI process. The MPI runtime will pin processes accordingly.
- For MPI calls from multiple MPI threads, extra caution; use `-mt_mpi`
- More information in the MPI Reference Manual

MPI Fabric Selection

- Environment variable `I_MPI_FABRICS`
- For inter-node communication: `tcp`, `dapl`
- For intra-node communication: `tcp`, `dapl`, `shm`
- Combination: `I_MPI_FABRICS=shm:dapl`
- TCP is configured in MPSS by default. DAPL requires installing OFED for MIC.
- Virtual infiniband interface `ibscif` for communication between coprocessors within a system.
- More information in the MPI Reference Manual

§7. Conclusion

Course Recap

Programming Models for Xeon Phi Coprocessors

1. Native coprocessor applications
   - Compile with `-mmic`
   - Run with `micnativeloadex` or `scp+ssh`
   - The way to go for MPI applications without offload
2. Explicit offload
   - Functions, global variables require `__attribute__((target(mic)))`
   - Initiate offload, data marshalling with `#pragma offload`
   - Only bitwise-copyable data can be shared
3. Clusters and multiple coprocessors
   - `#pragma offload target(mic:i)`
   - Use threads to offload to multiple coprocessors
   - Run native MPI applications

Optimization Checklist

1. Scalar optimization
2. Vectorization
3. Scale above 100 threads
4. Arithmetically intensive or bandwidth-limited
5. Efficient cooperation between the host and the coprocessor(s)

Double Rewards of MIC Programming

From the book "Parallel Programming and Optimization with Intel Xeon Phi Coprocessors" by Colfax:

It is not trivial to achieve good performance with Intel Xeon Phi coprocessors, especially when one compares it to the performance of modern Intel Xeon processors with the Sandy Bridge architecture. The new truth that HPC programmers must learn is: if a parallel code does not perform fast on Intel Xeon Phi coprocessors, it probably is not doing very well on Intel Xeon processors, either. The flip side of this truth is that **when developers invest time and effort into optimizing for the many-core architecture, they also reap performance benefits on multi-core processors**.
Book page: http://www.colfax-intl.com/nd/xeonphi/book.aspx

Double Rewards of MIC Programming

From the article "An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors" by James Reinders

The single most important lesson from working with Intel Xeon Phi coprocessors is this: the best way to prepare for Intel Xeon Phi coprocessors is to fully exploit the performance that an application can get on Intel Xeon processors first.

...

The experiences of users of Intel Xeon Phi coprocessors ... point out one challenge: the temptation to stop tuning before the best performance is reached. ... There ain't no such thing as a free lunch! The hidden bonus is the "transforming- and-tuning" double advantage of programming investments for Intel Xeon Phi coprocessors that generally applies directly to any general-purpose processor as well. This greatly enhances the preservation of any investment to tune working code by applying to other processors and offering more forward scaling to future systems.

Full article here

Additional Resources: Reading, Guides, Support

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

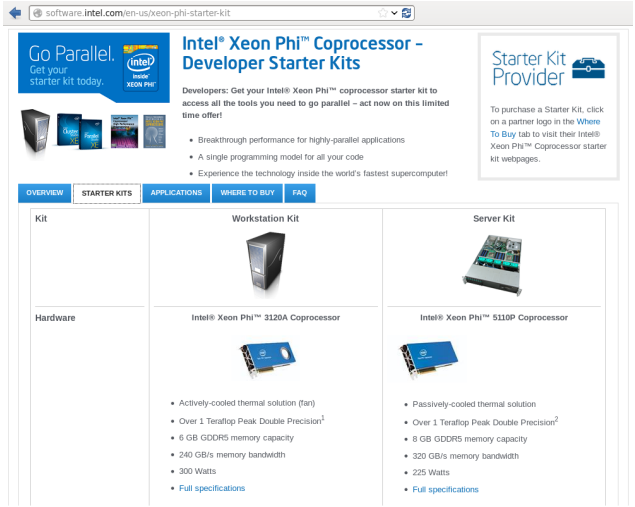
Link to TOP10 List for Starter Kit Developers

Intel's Top 10 List (continued)

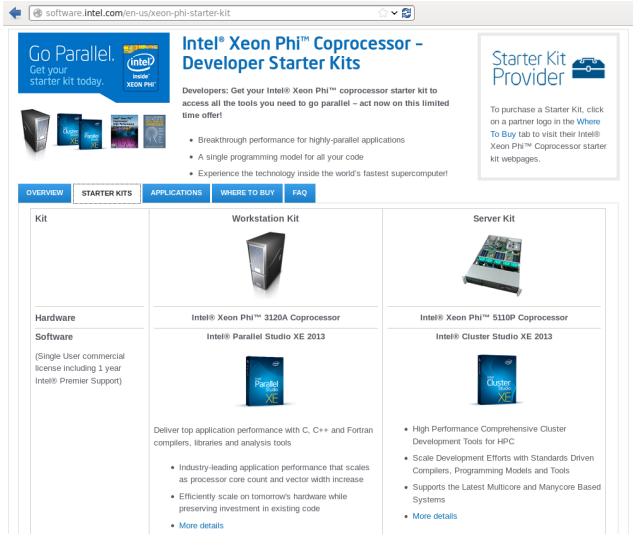
6. Try your hand at the beginning lab exercises
7. Try your hand at the beginner/intermediate real world app exercises
8. Browse the case studies webpage to view examples from many segments
9. Begin optimizing your application(s); consult your programming books, the ISA reference manual, and the support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

Link to TOP10 List for Starter Kit Developers

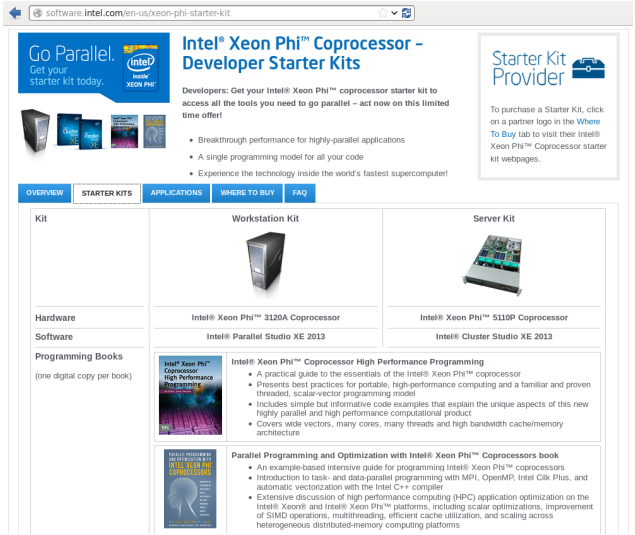
Intel Xeon Phi Starter Kit



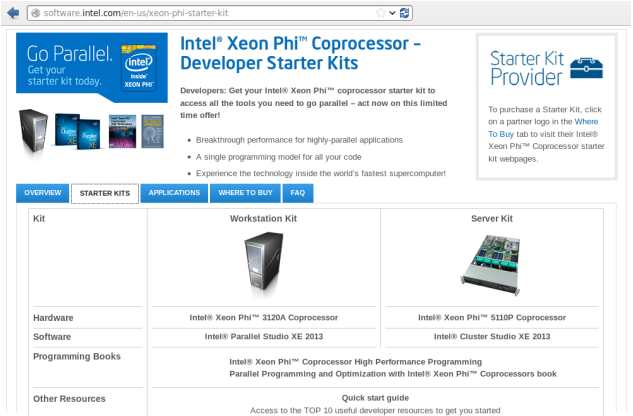
Intel Xeon Phi Starter Kit



Intel Xeon Phi Starter Kit



Intel Xeon Phi Starter Kit



Workstations with Intel Xeon Phi Coprocessors (Jan 2014)

	Model	Max Coprocessors	Graphics	CPU	Max Memory	Max Drives	Form Factor
	SXP8600p	4 5110P with Passive Heatsink	On-board	Dual Intel® Xeon® E5-2600 V2 Series	Up to 512GB	8 3.5" SAS/SATA	Tower / 4U Rackmountable
	SXP8600	4 3120A with Active Heatsink	On-board OR Discrete graphics card	Dual Intel® Xeon® E5-2600 V2 Series	Up to 512GB	8 3.5" SAS/SATA	Tower / 4U Rackmountable
	SXP7450	2 3120A with Active Heatsink	Discrete graphics card	Dual Intel® Xeon® E5-2600 V2 Series	Up to 512GB	8 2.5" HDDs/SSDs	Tower / 4U Rackmountable
	SXP2300	1 3120A with Active Heatsink	Discrete graphics card	Dual Intel® Xeon® E5-2600 V2 Series	Up to 256GB	4 3.5" SATA	Tower
	SXP1300	1 3120A with Active Heatsink	Discrete graphics card	Single Intel® Xeon® E5-2600 V2 or E5-1600 Series	Up to 64GB	4 3.5" SATA	Tower

http://www.colfax-intl.com/nd/xeonphi/workstations.aspx

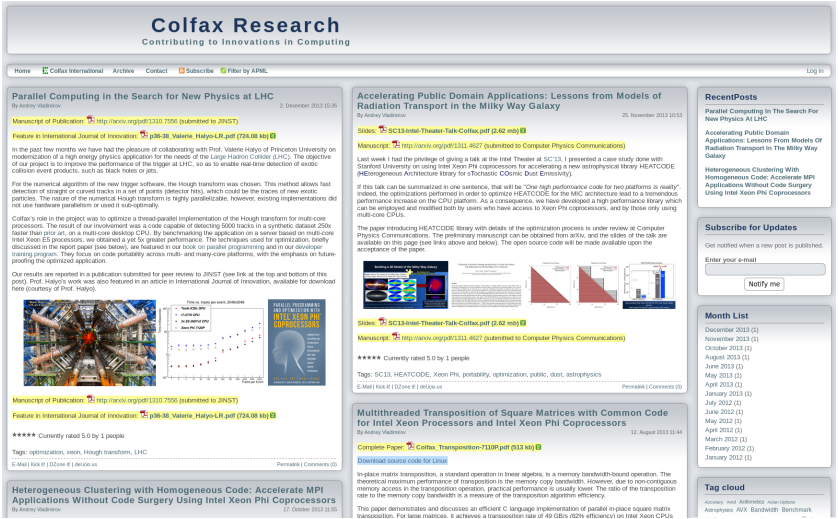
Servers with Intel Xeon Phi Coprocessors (Jan 2014)

Model	Form Factor	Sockets	CPU/GPU	Max. Memory	Max. Hdds	Features
CX1160i-XP5 	1U	1 CPU Socket	Intel Xeon E5-1600 Intel Xeon Phi Coprocessor	Max 256GB	Max 4 HDDs	• Single Intel® Xeon Phi Coprocessor • 1x PCIe 3.0 x16 Low-Profile Slot • 4x 1 Gigabit Ethernet Ports • Optional Intel Remote Management Module • Single 750W Platinum PS
CX1120s-XP5 	1U	1 CPU Socket Max 2 GPU/Coprocessor Cards	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 256GB	Max 8 HDDs	• 2x Intel Xeon Phi Coprocessor • 1x Low-Profile PCI-E 3.0 x8 Slot • Dedicated LAN for System Management • Single 1400W Platinum PS
CX1220i- XP5 	1U	1 CPU Socket Max 2 GPU/Coprocessor Cards	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 256GB	Max 4 HDDs	• 2x Intel Xeon Phi Coprocessor • 1x Low-Profile PCI-E 3.0 x8 Slot • Dedicated LAN for System Management • Single 1400W Platinum PS
CX1160i-XP5 	1U	1 CPU Socket Max 1 GPU/Coprocessor Card	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 256GB	Max 4 HDDs	• Single Intel® Xeon Phi Coprocessor • 1x PCIe 3.0 x16 Low-Profile Slot • 4x 1 Gigabit Ethernet Ports • Optional Intel Remote Management Module • Single 750W Platinum PS
CX1350s- XP5 	1U	2 CPU Sockets Max 3 GPU/Coprocessor Cards	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 256GB	Max 4 HDDs	• 3x Intel® Xeon Phi™ Coprocessors • 1x Low-Profile PCIe 3.0 x8 Slot • 2x RJ45 10GBase-T Ports Option • Dedicated LAN for System Management • Redundant 1800W Platinum PS
CX1250s-XP5 	1U	2 CPU Sockets Max 2 GPU/Coprocessor Cards	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 512GB	Max 4 HDDs	• 2x Intel Xeon Phi Coprocessor • 1x Low-Profile PCI-E 3.0 x8 Slot • Dedicated LAN for System Management • Redundant 1800W Platinum PS
CX2265i- XP5 	2U	2 CPU Sockets Max 2 GPU/Coprocessor Cards	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 768GB	Max 8 HDDs	• 2x Intel® Xeon Phi™ Coprocessors • 8x 2.5" or 8x 3.5" HDDs • 2x 2.5" Internal Fixed Mount SSDs • 4x 1 Gigabit Ethernet Ports • Intel Remote Management Module • Redundant 750W Platinum PS

Model	Form Factor	Sockets	CPU/GPU	Max. Memory	Max. Hdds	Features
CX2465s-XP5 	2U	2 CPU Sockets Max 4 GPU/Coprocessor Cards	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 512GB	Max 4 HDDs	• 4x Intel® Xeon Phi™ Coprocessors • SATA Disk-on-Module (DOM) support • Integrated Single Port QDR or FDR InfiniBand • Redundant 1600W Platinum PS
CX2475s- XP5 	2U	2 CPU Sockets Max 4 GPU/Coprocessor Cards	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 512GB	Max 8 HDDs	• 4x Intel® Xeon Phi™ Coprocessors • 1x Low-Profile PCI-E 3.0 x16 Slot • Integrated Single Port FDR InfiniBand • Dedicated LAN for System Management • Redundant 1620W Platinum PS
CX2445i-XP5 	2U	4 CPU Sockets Max 2 GPU/Coprocessor Cards	Intel Xeon E5-4600 Intel Xeon Phi Coprocessor	1024GB+	Max 8 HDDs	• 2x Intel® Xeon Phi™ Coprocessors • 2x Internal Fixed SSDs • Integrated 6Gb/s LSI SAS2308 ROC Controller • 2x 10 Gigabit Ethernet Ports • Intel Remote Management Module • Redundant 1600W Platinum PS
CX4460c-XPFT5 	4U Fat Twin	2 CPU Sockets Max 3 GPU/Coprocessor Cards	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 512GB	Max 2 HDDs	• 4U, 4 Hot-Swap Nodes • 2 CPUs/node • 16x DIMMs/node • 2 PCI-E 3.0 x8/Node • 2x 3.5" Drives/Node • Integrated 10GbE Option/Node • Dedicated LAN for System Management/Node • Redundant 1620W Platinum PS
CXP9100 	4U+	2 CPU Sockets Max 8 GPU/Coprocessor Cards	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 768GB	Max 6 HDDs	• 8x Intel® Xeon Phi™ Coprocessors • Internal 6x 2.5" or 4x 3.5" HDDs • 2x 10 Gigabit Ethernet Ports • Optional Single / Dual QDR or FDR InfiniBand Module • Redundant 2+1 2400W Platinum PS
CXP9000 	4U+	2 CPU Sockets Max 8 GPU/Coprocessor Cards	Intel Xeon E5-2600 V2 Intel Xeon Phi Coprocessor	Max 768GB	Max 6 HDDs	• 8x Intel® Xeon Phi™ Coprocessors • Internal 6x 2.5" or 4x 3.5" HDDs • 2x 1 Gigabit Ethernet Ports • Optional Single / Dual QDR or FDR InfiniBand Module • Redundant 2+1 2400W Platinum PS

http://www.colfax-intl.com/nd/xeonphi/servers.aspx

Research and Consulting



http://research.colfaxinternational.com/

Research and Consulting

Colfax offers consulting services for Enterprises, Research Labs, and Universities. We can help you to:

- Optimize your existing application to take advantage of all levels of hardware parallelism
- Future-proof for upcoming innovations in computing solutions.
- Accelerate your application using coprocessor technologies.
- Investigate the potential system configurations that satisfy your cost, power and performance requirements.
- Take a deep dive to develop a novel approach.

For more details, contact us at phi@colfax-intl.com to discuss what we can do together

Intel® Xeon Phi™ Coprocessor
Remote Access and System Loaner Programs

Remote Access Systems

Intel-supported options for Academia:

- Manycore Testing Lab through SSG (more info)
- Intel Science & Technology Center (ISTC) and Intel Collaborative Research Institutes (ICRI) programs through Intel Labs (more info)
- Texas Advanced Computing Center (TACC) and National Institute for Computational Sciences (NICS) both offer allocations through the NSF XSEDE program (more info)

Colfax Code Treadmill:

- Seven-day, 24/7 remote access to a personal HPC server at Colfax with training materials, Intel® Xeon® processors, Intel® Xeon Phi™ coprocessors and software development tools
- More Information: HERE

Loaner Programs

Intel Demo Depot:

- Contact your local Intel sales representative for requesting an Intel® Xeon Phi™ coprocessor-based system

Colfax Loaner Program:

- 30-day access to a loaner system, complete with Colfax hardware and software programming support
- More information please send email to : phi@colfax-intl.com

Please contact your Intel BDM or local OEM representative for more remote access and system loaner options

Intel® Xeon Phi™ Coprocessor Starter Kits

Go parallel today with a fully-configured system starting below $5K*

OR

software.intel.com/xeon-phi-starter-kit

Other brands and names are the property of their respective owners.
**Pricing and starter kit configurations will vary. See software.intel.com/xeon-phi-starter-kit and provider websites for full details and disclaimers. Stated currency is US Dollars.*

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