Cornell University
Center for Advanced Computing

# Preparing for Highly Parallel, Heterogeneous Coprocessing

Steve Lantz
Senior Research Associate
Cornell CAC

*Workshop: Parallel Computing on Ranger and Lonestar*
*May 17, 2012*

# What Are We Talking About Here?

- Hardware trend since around 2004: processors gain more cores (execution engines) rather than greater clock speed
  - IBM POWER4 (2001) became the first chip with 2 cores, 1.1–1.9 GHz; meanwhile, Intel's single-core Pentium 4 was a bust at >3.8 GHz
  - Top server and workstation chips in 2012 (Intel Xeon, AMD Opteron) now have 4, 8, even 16 cores, running at 1.6–3.2 GHz
- Does it mean Moore's Law is dead? No!
  - Transistor densities are still doubling every 2 years
  - Clock rates have stalled at < 4 GHz due to power consumption
  - Only way to increase flop/s/watt is through greater on-die parallelism…
- *If 1 chip holds 10s of the best cores, why not 100s of weaker ones?*
  - Around 2007–8, "Cell" chips had 1 main and 8 synergistic processors; but then something else came along…

# Highly Parallel Hardware Is in PCs Already

- High-end graphics processing units (GPUs) contain 100s of thread processors and RAM enough to rival CPUs in compute capability
- GPUs are being further tailored for HPC
- Lonestar example: NVIDIA Tesla M2070
  - 448 CUDA cores @ 1.15 GHz
  - 6GB dedicated memory
  - 1.03 Tflop/s peak SP rate
  - 238W power consumption

Tesla C2070

- Initially there were hardware obstacles to using GPUs for general calculations, but these have been overcome
  - ECC memory, double precision, IEEE-compliant arithmetic are built in
  - What about software…?

# General Purpose Computing on GPUs (GPGPU)

- Given the right software tools, developers can write code allowing the GPU to perform calculations usually handled by the CPU
  - Stream processing: GPU executes a code “kernel” on a stream of inputs
  - Exploits the GPU’s rendering pipeline, designed to transform and shade a stream of vertices: highly parallel, very energy efficient
  - Works well if kernel is multithreaded, vectorized (SIMD), pipelined
- NVIDIA CUDA (2006) is the forerunner in this area
  - SDK + API that permits programmers to use the C language to code algorithms for execution on NVIDIA GPUs (must be compiled with nvcc)
- OpenCL (2008) is a more recent, open standard originated by Apple
  - C99-based language + API that enables data-parallel computation on GPUs as well as CPUs
  - Actively supported on Intel, AMD, NVIDIA, ARM platforms

# Not All Applications Are Suitable for GPGPU

- Workload must be compute intensive, i.e., any data item fetched from main memory must take part in several GPU operations
  - Reason: to reach the GPU, data travel over a “slow” PCIe interconnect
- Workload must be decomposed into many small, independent units
  - Reason: GPU is only effective when all thread processors are kept busy
- *Nontrivial (re)coding may be needed, based on a specialized API*
  - Good performance depends on very specific tuning to the hardware (cache sizes, etc.)
  - Resulting code is far less portable due to the API and special tuning
  - May be avoided if a suitable kernel or library already exists
- Is there a better way for numerically-intensive applications to take advantage of hardware trends?

# The Intel Approach: MIC

- MIC = Many Integrated Cores = a “coprocessor” on a PCIe card that features >50 compute cores
  - Represents Intel’s response to GPGPU, especially NVIDIA’s CUDA
  - Incorporates lessons learned from the former “Larrabee” development effort (which never became a product)
  - Answers the question: if 8 modern Xeon cores fit on a die, how many Pentium III’s would fit?
- Addresses the API problem: standard x86 instructions are supported
  - Includes 64-bit addressing
  - Other recent x86 extensions may not be available
  - Special instructions are added for an extra-wide (512-bit) vector register
- MIC executables are built using familiar Intel compilers, libraries, and analysis tools

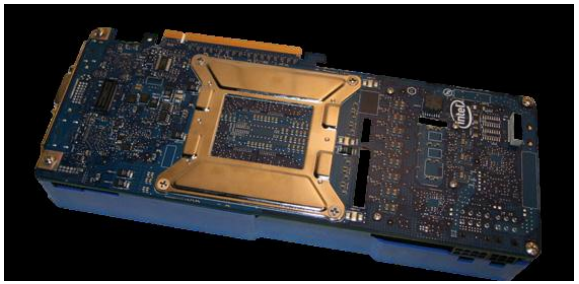
# MIC = A Teraflop/s *System* on a Chip!

- 1996: ASCI Red, first system to achieve 1 Tflop/s sustained
- 72 cabinets


- 2011: Knights Corner, first Tflop/s *system* on a chip
- 1 PCIe slot

# The Knights Ferry (KNF) Coprocessor

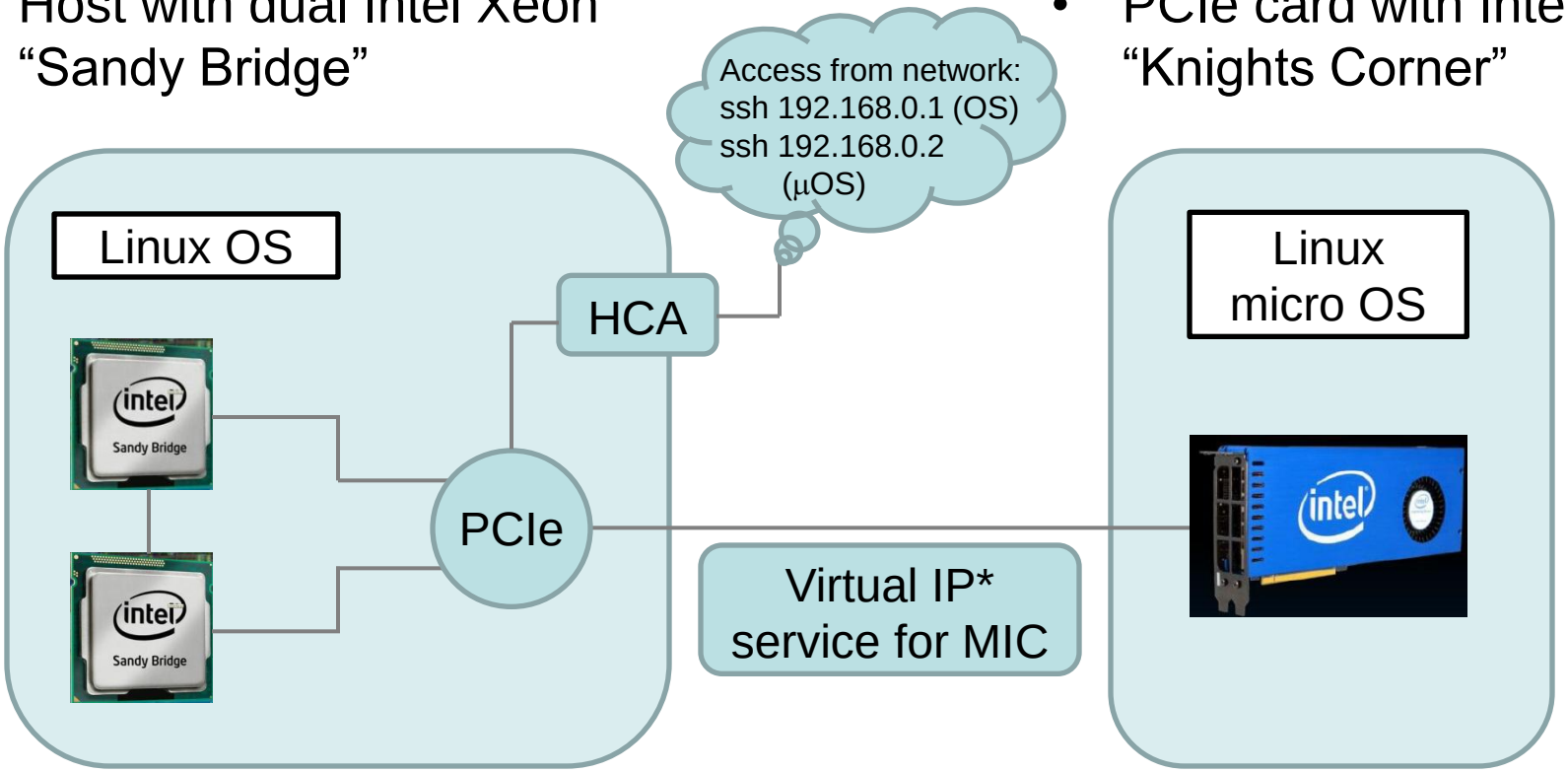
- KNF is the early development platform for the Intel MIC architecture
  - Chip + memory on a PCI Express card
  - Up to 8MB coherent shared L2 cache
  - Up to 32 cores, 4 threads/core, < 1.2 GHz
  - SIMD vector unit (8-DP floats wide)
  - In-order instruction pipeline



- Linux Micro OS (μOS) runs on MIC
  - Small OS memory footprint
  - Basic functionality (I/O, standards Unix commands, etc.)
  - Users can telnet to MIC
- RHEL 6.0, 6.1, 6.2 or SUSE 11 SP1 runs on host
- Details for *Knights Corner* (KNC) have yet not been disclosed

# Typical Configuration of a Future Stampede Node

- Host with dual Intel Xeon "Sandy Bridge"
- PCIe card with Intel "Knights Corner"




* can't do this with a Lonestar GPU node, e.g., which is otherwise similar

# First Large-Scale MIC System: TACC Stampede

**System specs from news release**

- 10PF+ peak performance in initial system (1Q 2013)
  - 2PF conventional cluster (Sandy Bridge)
  - 8PF complementary coprocessors (KNC)
- 15PF+ after upgrade
- 14PB+ disk, 200TB+ RAM
- 56Gb/s FDR InfiniBand, fat-tree interconnect, ~75 miles of cables
  - Compare Lonestar: 32Gb/s QDR (effective)
- Nearly 200 racks of compute hardware
- Integrated shared memory and remote visualization subsystems
- Total concurrency approaching 500,000 cores

# Construction Is Already Under Way at TACC


Left: water chiller plant; right: addition to main facility

# Programming Models for Stampede

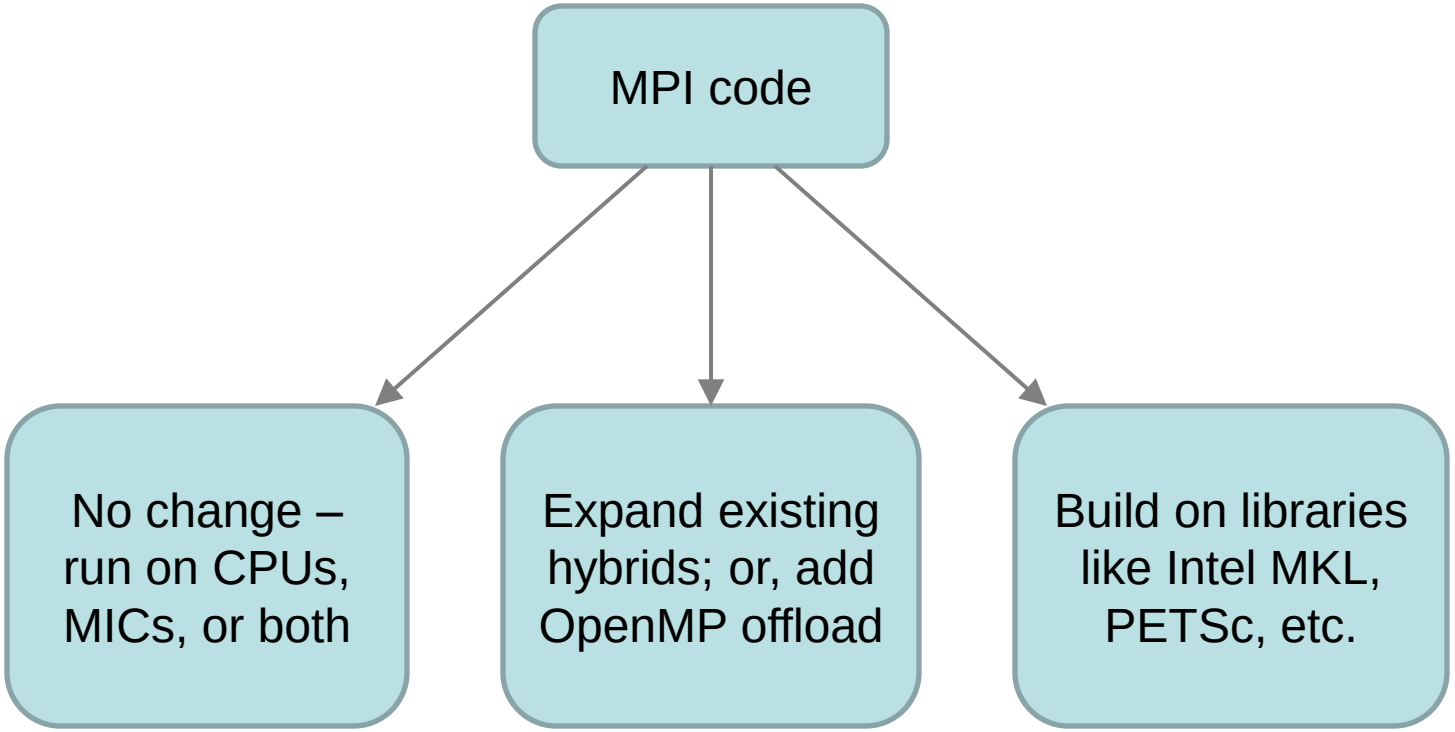
**Offload Execution**

- Directives indicate data and functions to send from CPU to MIC for execution
- Unified source code
- Code modifications required
- Compile once with offload flags
  - Single executable includes instructions for MIC and CPU
- Run in parallel using MPI and/or scripting, if desired

**"Symmetric" Execution**

- Message passing (MPI) on CPUs and MICs alike
- Unified source code
- Code modifications optional
  - Assign different work to CPUs vs. MICs
  - Multithread with OpenMP for CPUs, MICs, or both
- Compile twice, 2 executables
  - One for MIC, one for host
- Run in parallel using MPI

# Strategies for HPC Codes



# Pros and Cons of MIC Programming Models

- Offload engine: accelerator for host
  - *Pros:* distinct hardware gets distinct role; programmable via simple calls to a library such as MKL, or via directives (we'll go into depth on this)
  - *Cons:* most work travels over PCIe; difficult to retain data on card
- "Symmetric" #1: heterogeneous MPI cores
  - *Pros:* MPI works for all cores (though 1 MIC core < 1 server core)
  - *Cons:* memory is insufficient to give each core a μOS plus lots of data; fails to take good advantage of shared memory; PCIe is a bottleneck
- "Symmetric" #2: heterogeneous SMPs (symmetric multiprocessors)
  - *Pros:* MPI/OpenMP works for both host and MIC; efficient use of limited PCIe bandwidth and MIC memory due to single message source/sink
  - *Cons:* hybrid programming is already tough on homogeneous SMPs; not clear whether existing OpenMP-based hybrids scale to 50+ cores

# Using Compiler Directives to Offload Work

- OpenMP's directives provide a natural model
  - 2010: OpenMP working group starts to consider accelerator extensions
  - Related efforts are launched to target specific types of accelerators…
- LEO, Language Extensions for Offload
  - Intel moves forward to support processors and co-processors, initially
- OpenACC
  - PGI moves forward to support GPUs, initially
- Will OpenMP 4.0 produce a compromise among all the above?
  - Clearly desirable, but it's difficult
  - Other devices exist: network controllers, antenna A/D, cameras…
  - Exactly what falls in the "accelerator" class? How diverse is it?

# OpenMP Offload Constructs: Base Program

```
#include <omp.h>
#define N 10000

void foo(double *, double *, double *, int );
int main(){
    int i; double a[N], b[N], c[N];
    for(i=0;i<N;i++){ a[i]=i; b[i]=N-1-i;}

    ...

    foo(a,b,c,N);
}

void foo(double *a, double *b, double *c, int n){
    int i;

    for(i=0;i<n;i++) { c[i]=a[i]*2.0e0 + b[i]; } }
```

- Objective: offload foo to a device
- Use OpenMP to do the offload

# OpenMP Offload Constructs: Requirements

```
#include <omp.h>
#define N 10000
#pragma omp <offload_function_spec>
void foo(double *, double *, double *, int );
int main(){
    int i; double a[N], b[N], c[N];
    for(i=0;i<N;i++){ a[i]=i; b[i]=N-1-i;}

    ...
    #pragma omp <offload_this>
    foo(a,b,c,N);
}
#pragma omp <offload_function_spec>
void foo(double *a, double *b, double *c, int n){
    int i;
    #pragma omp parallel for
    for(i=0;i<n;i++) { c[i]=a[i]*2.0e0 + b[i]; } }
```

- Direct compiler to offload function or block
- “Decorate” function and prototype
- Usual OpenMP directives work on device

# OpenMP Offload Constructs: Data Requirements

```
#include <omp.h>
#define N 10000
#pragma omp <offload_function_spec>
void foo(double *, double *, double *, int );
int main(){
    int i; double a[N], b[N], c[N];
    for(i=0;i<N;i++){ a[i]=i; b[i]=N-1-i;}

    ...
    #pragma omp <offload_this> <data_clause>
    foo(a,b,c,N);
}
#pragma omp <offload_function_spec>
void foo(double *a, double *b, double *c, int n){
    int i;
    #pragma omp parallel for
    for(i=0;i<n;i++) { c[i]=a[i]*2.0e0 + b[i]; } }
```

- Data must be moved to and from the device
- Synchronous model (move data to device at dispatch of execution, move back afterward)
- *Control of data locality* is new to OpenMP (and OpenACC)

# OpenMP Offload Constructs: Asynchronicity

```
#include <omp.h>
#define N 10000
#pragma omp <offload_function_spec>
void foo(double *, double *, double *, int );
int main(){
    int i; double a[N], b[N], c[N];
    for(i=0;i<N;i++){ a[i]=i; b[i]=N-1-i;}
    #pragma omp <offload_data> <async>
    …
    #pragma omp <offload_this> <async> <data>
    foo(a,b,c,N);
}
#pragma omp <offload_function_spec>
void foo(double *a, double *b, double *c, int n){
    int i;
    #pragma omp parallel for
    for(i=0;i<n;i++) { c[i]=a[i]*2.0e0 + b[i]; } }
```

- Offloaded “region” can be done asynchronously
- Moving data asynchronously is another important option

# Two Types of MIC Parallelism (Both Are Needed)

- OpenMP (offloading to MIC)
  - Work Level Parallelization (threads)
  - Requires management of asynchronous “processes”
  - It’s all about sharing work and scheduling
- Vectorization
  - “Lock step” Instruction Level Parallelization (SIMD)
  - Requires management of synchronized instruction execution
  - It’s all about finding simultaneous operations
- To fully utiltize MIC, both types of parallelism should be identified and exploited

# Vectorization Matters Too

- Vectorization, or SIMD processing, enables simultaneous, independent operation on multiple data operands with a single instruction. (Large arrays should provide a constant stream of data.)
- Vector reawakening
  - Pre-Sandy Bridge – DP* vector units are only 2 wide (SSE)
  - Sandy Bridge – DP* vector units are 4 wide (AVX)
  - MIC – DP* vector units are 8 wide! (VEX)
- Unvectorized loops lose 4x performance on Sandy Bridge and 8x performance on MIC!
- Evaluate performance with vectorization compiler options turned on and off ("-no-vec") to assess overall vectorization.

*DP = double precision (8 bytes, 64 bits)

# Working with a Vectorizing Compiler

- Compilers are good at vectorizing inner loops, but they need help
  - Make sure each iteration is independent
  - Align data to match register sizes and cache line boundaries
- Compilers will look for vectorization opportunities starting at -O2
  - To apply the latest relevant vector instructions for the given architecture: -x<*simd_instr_set*>
  - To examine assembly code, *myprog.s*: -S
  - To confirm with a vector report: -vec-report=<n>, n="verboseness"

```
% ifort -xHOST -vec-report=4 prog.f90 -c
prog.f90(31): (col. 11) remark:
        loop was not vectorized: existence of vector dependence
```

# Dreams for Future Software Convergence

What would application programmers find most desirable?

- Stay with standard languages, language extensions, and compilers
  - Move away from CUDA and special compilers
- Operate at a high level
  - Avoid “intrinsics” that resemble assembly language
- Express programs in terms of generic parallel tasks
  - De-emphasize APIs based on specific realizations, like threads

Maybe we’re getting close…

- Compilers support vectorization and OpenMP (CL? ACC?)
- There’s plenty of room for improvement in the emerging standards

# What Does All This Mean for Me?

- Next-generation algorithms and programs will need to run well on architectures like those just described
  - Power efficiency will inevitably drive hardware designs in this direction
  - Mobile processors are just as power-constrained as HPC behemoths; the design goals align
  - It seems likely that even workstations and laptops will soon come equipped with many-core coprocessors of some type
- Therefore, finding and expressing parallelism in your computational workload will become increasingly important
  - Codes must be flexible enough to deal with heterogeneous resources
  - Asynchronous, adaptable methods may eventually be favored

# Reference

- Much of the information in this talk was gathered from presentations at the TACC–Intel Highly Parallel Computing Symposium, Austin, Texas, April 10–11, 2012: http://www.tacc.utexas.edu/ti-hpcs12.