Introduction to Parallel Computing

Victor Eijkhout

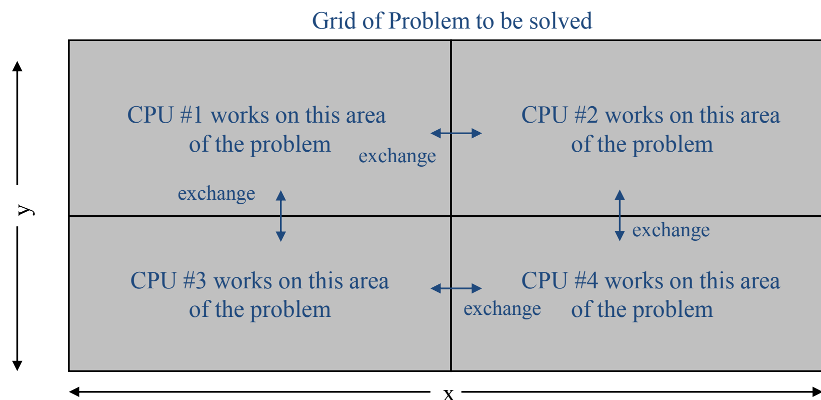
October, 2012

What is Parallel Computing?

- Parallel computing: use of multiple processors or computers working together on a common task.
  - Each processor works on part of the problem
  - Processors can exchange information

Paradigm #1: Data parallelism

- The program models a physical object, which gets partitioned and divided over the processors



Paradigm #2: Task parallelism

- There is a list of tasks (for instance runs of a small program) and processors cycle through this list until it is exhausted.
- Tasks can be partially ordered: Directed Acyclic Graph

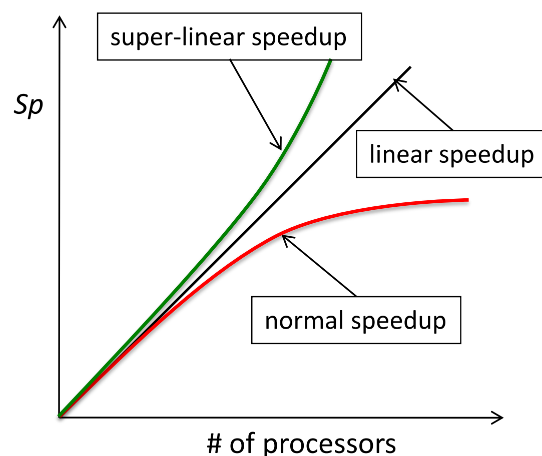
Why Do Parallel Computing?

- Limits of single CPU computing
  - performance
  - available memory
- Parallel computing allows one to:
  - solve problems that don't fit on a single CPU
  - solve problems that can't be solved in a reasonable time
- We can solve…
  - larger problems
  - faster
  - more cases

Speedup & Parallel Efficiency

- Speedup: $S_p = \frac{T_s}{T_p}$
  - *p* = # of processors
  - *Ts* = execution time of the sequential algorithm
  - *Tp* = execution time of the parallel algorithm with *p* processors
  - *Sp= P* (linear speedup: ideal)
- Parallel efficiency

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}$$



THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Limits of Parallel Computing

- Theoretical Upper Limits
  - Amdahl's Law
- Practical Limits
  - Load balancing
  - Non-computational sections
- Other Considerations
  - time to re-write code

Amdahl's Law

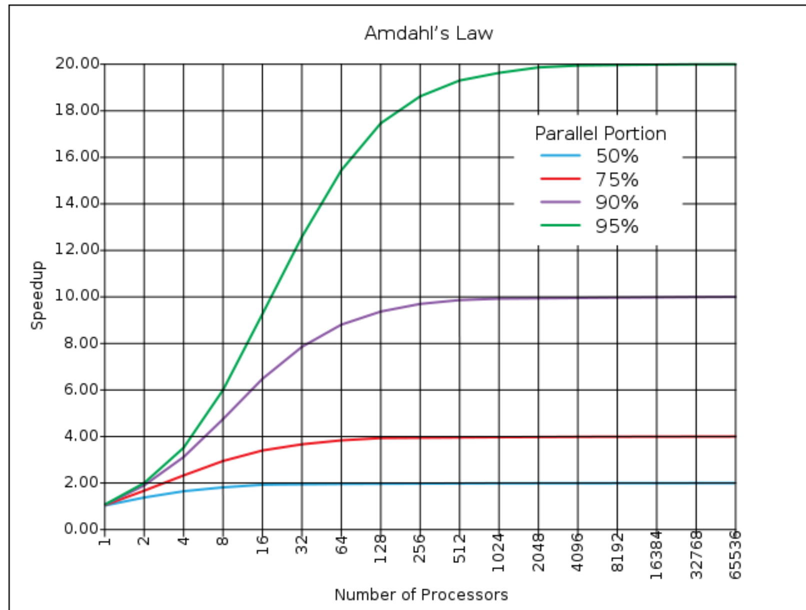
- All parallel programs contain:
  - parallel sections (we hope!)
  - serial sections (unfortunately)
- Serial sections limit the parallel effectiveness
- Amdahl's Law states this formally
  - Effect of multiple processors on speed up

$$S_P \le \frac{T_S}{T_P} = \frac{1}{f_s + \frac{f_p}{P}} \to \frac{1}{f_s}, p \to \infty$$

where

- f_s = serial fraction of code
- f_p = parallel fraction of code
- P = number of processors

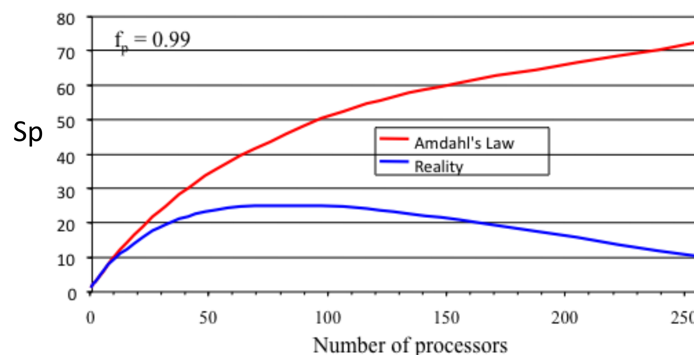
Amdahl's Law



Practical Limits: Amdahl's Law vs. Reality

- In reality, the situation is even worse than predicted by Amdahl's Law due to:
  - Load balancing (waiting)
  - Scheduling (shared processors or memory)
  - Cost of Communications
  - I/O

Sp



“Old school” hardware classification

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

SISD No parallelism in either instruction or data streams (mainframes)

SIMD Exploit data parallelism (stream processors, GPUs)

MISD doesn’t really exist

MIMD Multiple instructions operating independently on multiple data streams (most modern general purpose computers)

Hardware in parallel computing

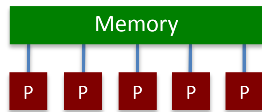
Memory access

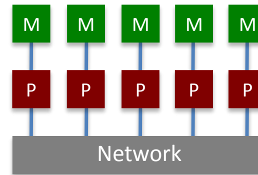
- Shared memory
  - SGI Altix
  - Cluster nodes
- Distributed memory
  - Uniprocessor clusters
- Hybrid
  - Multi-processor clusters
  - Processor with co-processor (GPU)
  - Cluster with multiple co-processors (Stampede!)

Processor type

- Single core CPU
  - Intel Xeon (Prestonia, Wallatin)
  - AMD Opteron (Sledgehammer, Venus)
  - IBM POWER (3, 4)
- Multi-core CPU (since 2005)
  - Intel Xeon (Paxville, Woodcrest, Harpertown…)
  - AMD Opteron (Barcelona, Shanghai, Istanbul,…)
  - IBM POWER (5, 6…)
- GPU based
  - Tesla systems

Shared and distributed memory

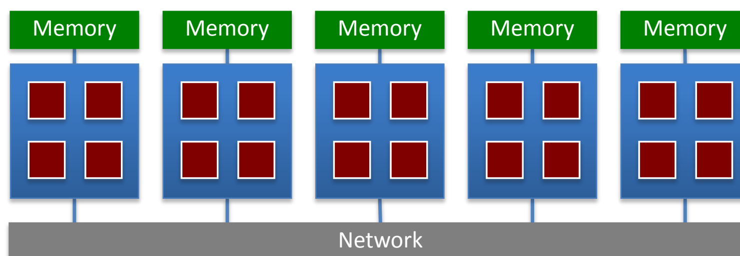




- All processors have access to a pool of shared memory
- Access times vary from CPU to CPU in NUMA systems
- Example: SGI Altix, IBM P5 nodes

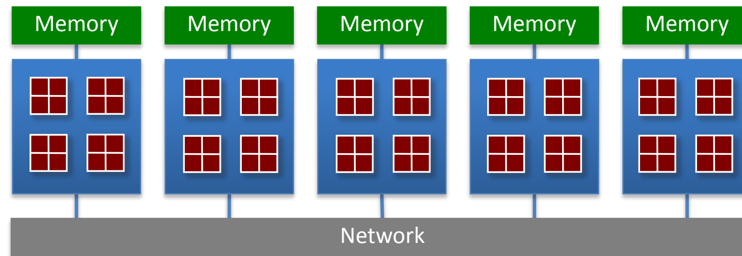
- Memory is local to each processor
- Data exchange by message passing over a network
- Example: Clusters with single-socket blades

Hybrid systems



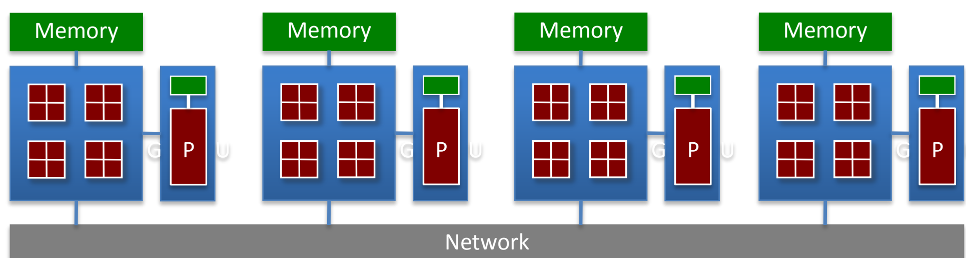
- A limited number, N, of processors have access to a common pool of shared memory
- To use more than N processors requires data exchange over a network
- Example: Cluster with multi-socket blades

Multi-core systems



- Extension of hybrid model
- Communication details increasingly complex
  - Cache access
  - Main memory access
  - Quick Path / Hyper Transport socket connections
  - Node to node connection via network

Co-processor Systems



- Calculations made in both CPUs and co-processors (GPU, MIC)
- No longer limited to single precision calculations
- Load balancing critical for performance
- Requires specific libraries and compilers (GPU: CUDA, OpenCL, MIC: OpenMP)

Argonne NATIONAL LABORATORY

Interconnects

Ken Raffenetti

Software Development Specialist

Programming Models and Runtime Systems Group

Mathematics and Computer Science Division

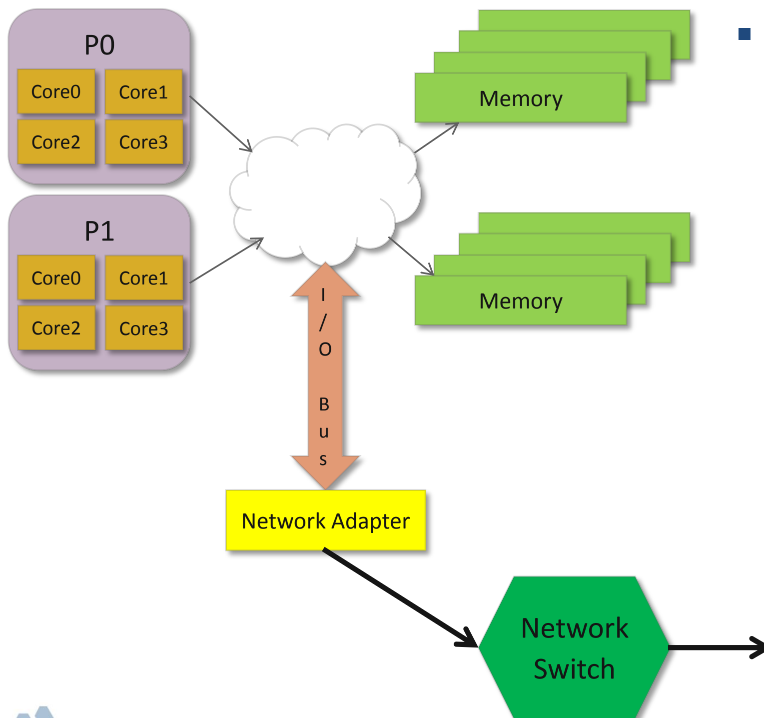
Argonne National Laboratory

U.S. DEPARTMENT OF ENERGY

End-host Network Interface Speeds

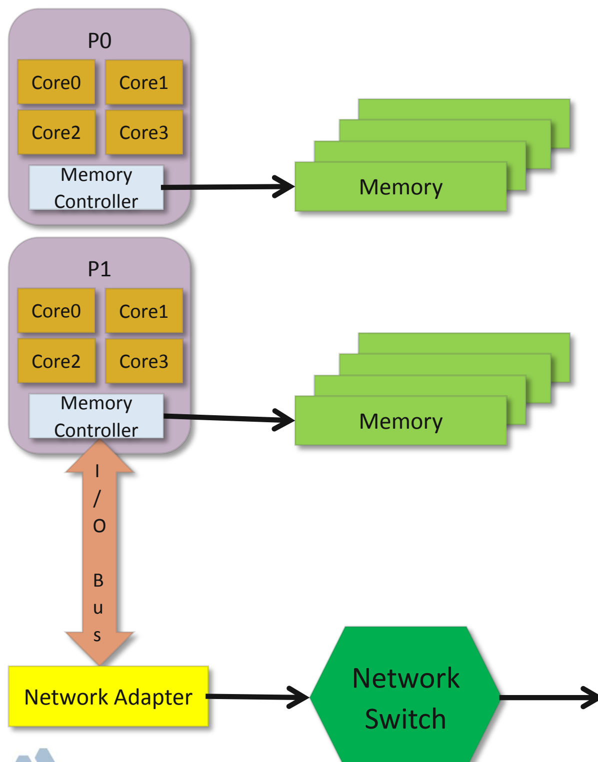
- Recent network technologies provide high bandwidth links
  - InfiniBand EDR gives 100 Gbps per network link
    - Upcoming networks expected to increase that by several fold
  - Multiple network links becoming a common place
    - ORNL Summit and LLNL Sierra machines, Japanese Post T2K machine
    - Torus style or other multi-dimensional networks
- End-host peak network bandwidth is "mostly" no longer considered a major limitation
- Network latency is still an issue
  - That's a harder problem to solve – limited by physics, not technology
    - There is some room to improve it in current technology (trimming the fat)
    - Significant effort in making systems denser so as to reduce network latency
- Other important metrics: message rate, congestion, …

Simple Network Architecture (past systems)



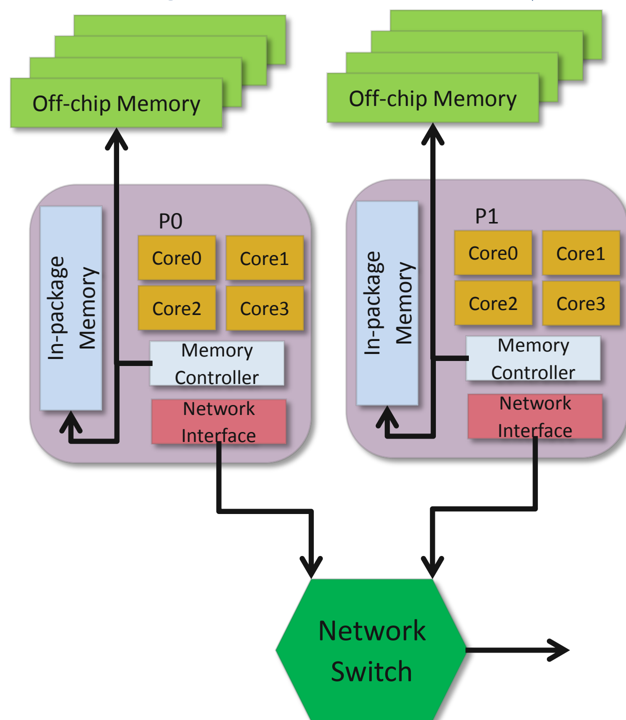
- Processor, memory, network are all decoupled

Integrated Memory Controllers (current systems)



- In the past 10 years or so, memory controllers have been integrated on to the processor
- Primary purpose was scalable memory bandwidth (NUMA)
- Also helps network communication
  - Data transfer to/from network requires coordination with caches
- Several network I/O technologies exist
  - PCIe, HTX, NVLink
  - Expected to provide higher bandwidth than what network links will have

Integrated Networks (current/future systems)



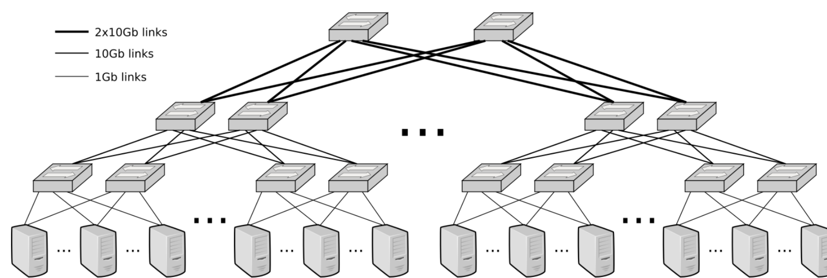
- Several vendors are considering processor-integrated network adapters
- May improve network bandwidth
  - Unclear if the I/O bus would be a bottleneck
- Improves network latencies
  - Control messages between the processor, network, and memory are now on-chip
- Improves network functionality
  - Communication is a first-class citizen and better integrated with processor features
  - E.g., network atomic operations can be atomic with respect to processor atomics

Traditional Network Topologies: Crossbar

- A network topology describes how different network adapters and switches are interconnected with each other
- The ideal network topology (for performance) is a crossbar
  - Alltoall connection between network adapters
  - Typically done on a single network ASIC
  - Current network crossbar ASICs go up to 64 ports; too expensive to scale to higher port counts
  - All communication is nonblocking

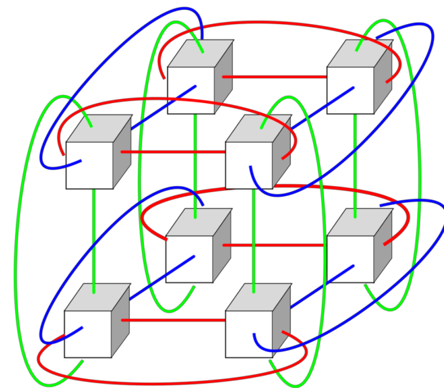
Traditional Network Topologies: Fat-tree

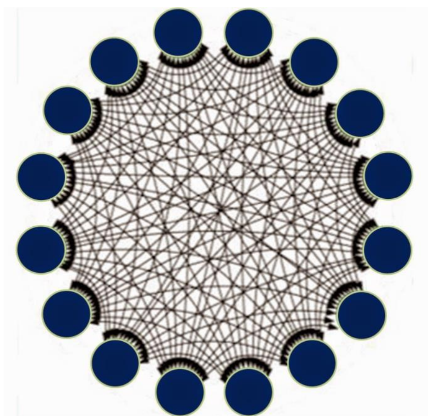
- The most common topology for small and medium scale systems is a fat-tree
  - Nonblocking fat-tree switches available in abundance
    - Allows for pseudo nonblocking communication
    - Between all pairs of processes, there exists a completely nonblocking path, but not all paths are nonblocking
  - More scalable than crossbars, but the number of network links still increases super-linearly with node count
    - Can get very expensive with scale



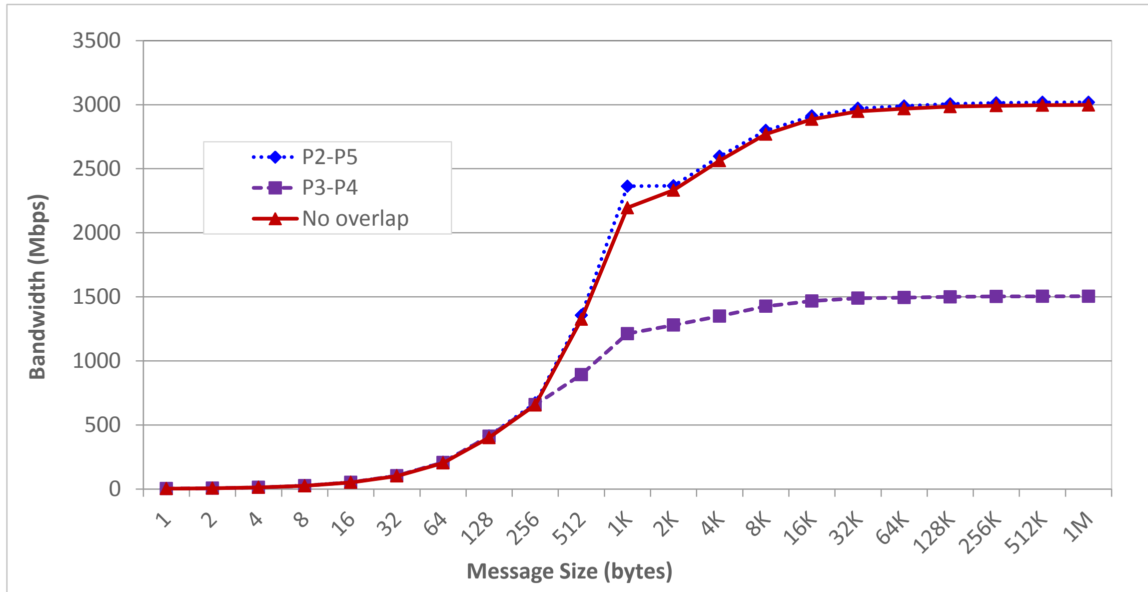
Network Topology Trends

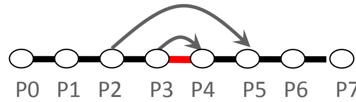
- Modern topologies are moving towards more "scalability" (with respect to cost, not performance)
- Blue Gene, Cray XE/XK, and K supercomputers use a torus-network; Cray XC uses dragonfly
  - Linear increase in the number of links/routers with system size
  - Any communication that is more than one hop away has a possibility of interference – congestion is not just possible, but common
  - Even when there is no congestion, such topologies increase the network diameter causing performance loss
- Take-away: topological locality is important and its not going to get better



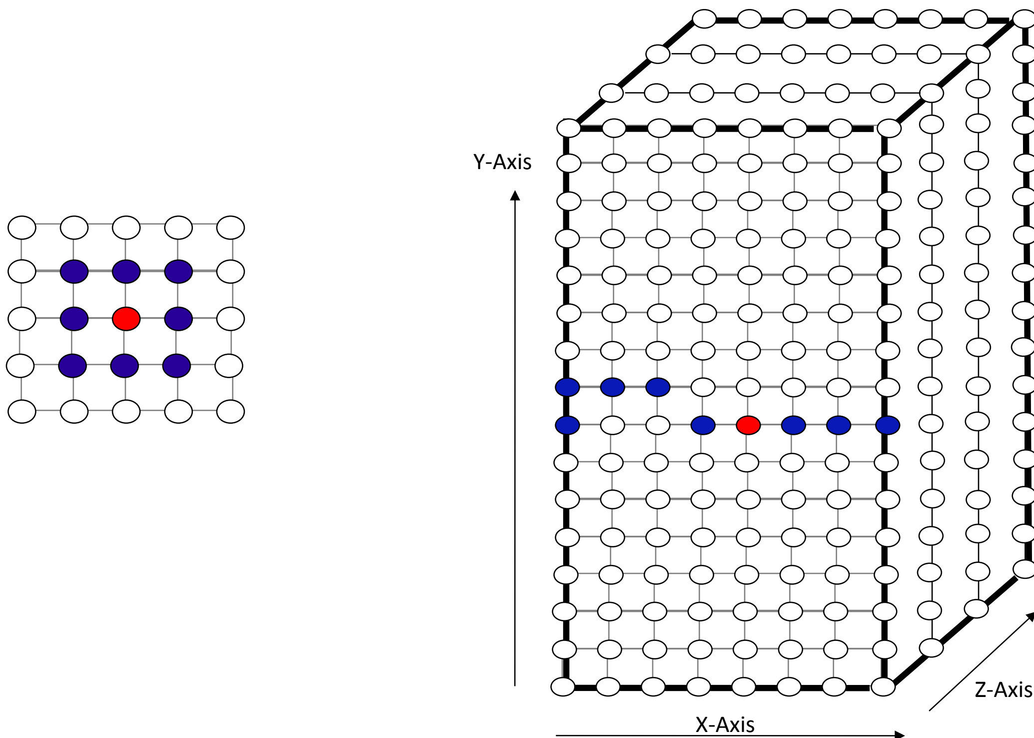


Network Congestion Behavior: IBM BG/P

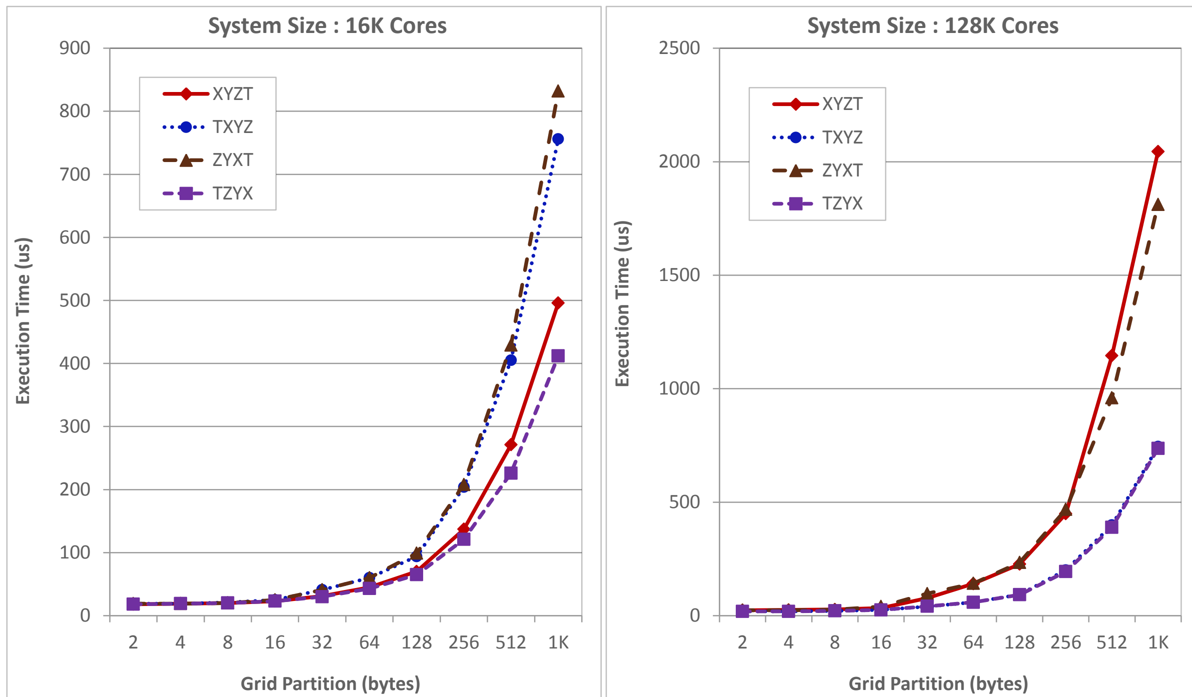




2D Nearest Neighbor: Process Mapping (XYZ)



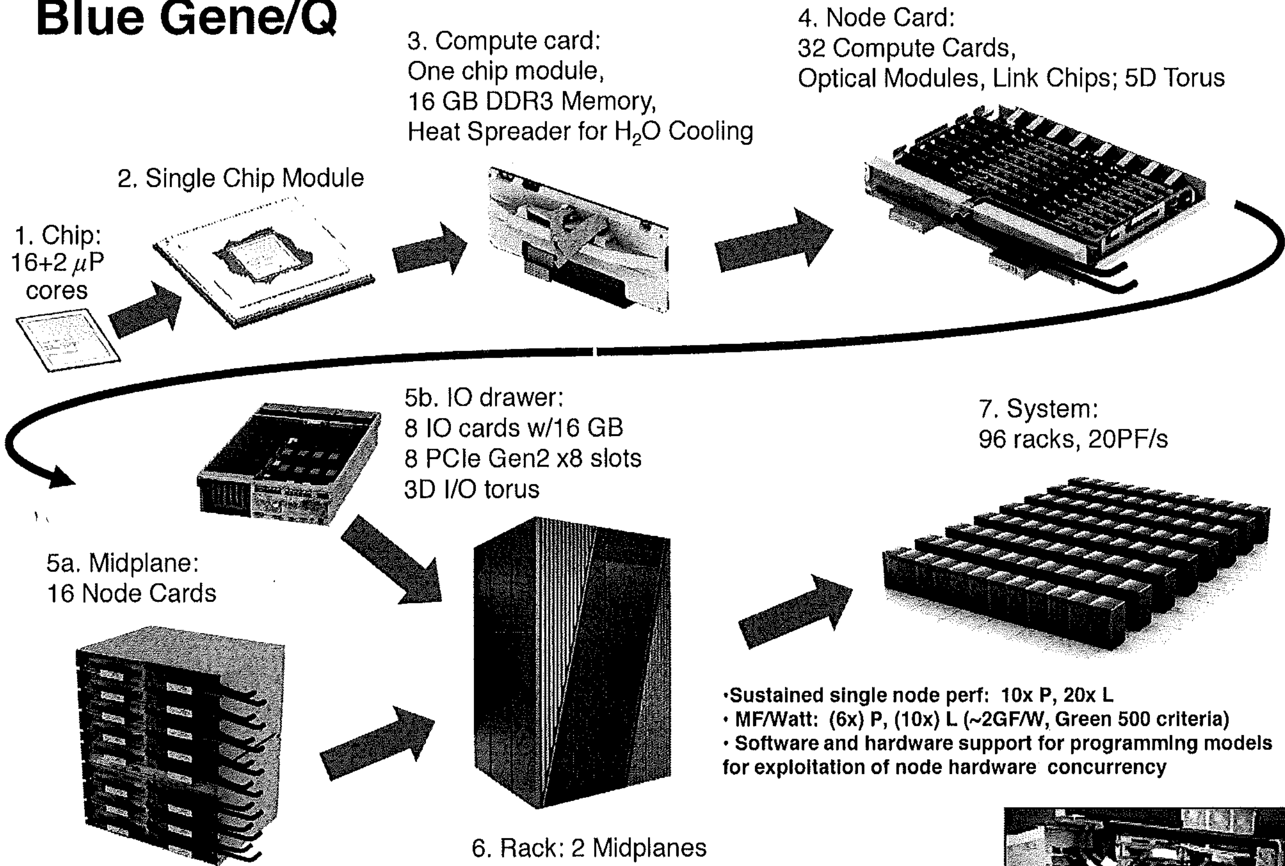
Nearest Neighbor Performance: IBM BG/P



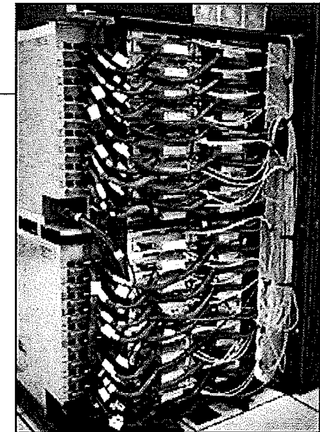
2D Halo Exchange

IBM

Blue Gene/Q

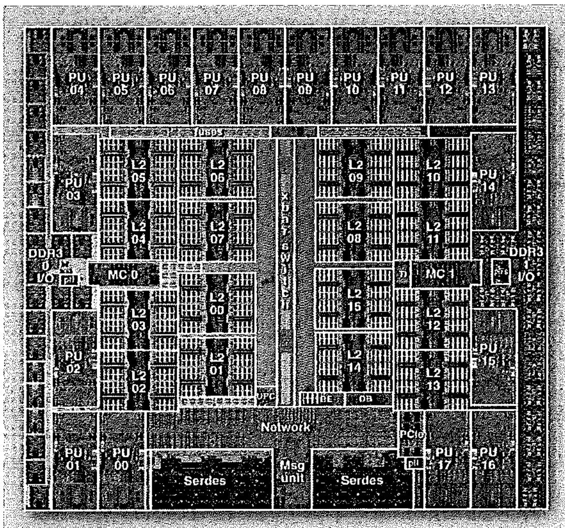


- **Sustained single node perf: 10x P, 20x L**
- **MF/Watt: (6x) P, (10x) L (~2GF/W, Green 500 criteria)**
- **Software and hardware support for programming models for exploitation of node hardware concurrency**



BlueGene/Q Compute chip

System-on-a-Chip design : integrates processors, memory and networking logic into a single chip

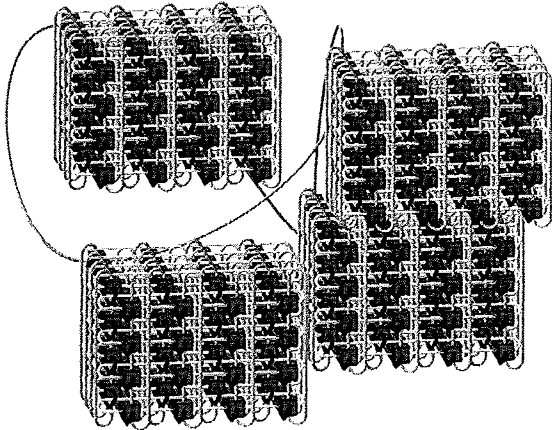


- 360 mm² Cu-45 technology (SOI)
- 16 user + 1 service PPC processors
  - plus 1 redundant processor
  - all processors are symmetric
  - 11 metal layer
  - each 4-way multi-threaded
  - 64 bits
  - 1.6 GHz
  - L1 I/D cache = 16kB/16kB
  - L1 prefetch engines
  - each processor has Quad FPU
  - (4-wide double precision, SIMD)
  - peak performance 204.8 GFLOPS @ 55 W
- Central shared L2 cache: 32 MB
  - eDRAM
  - multiversioned cache -- supports transactional memory, speculative execution.
  - supports scalable atomic operations
- Dual memory controller
  - 16 GB external DDR3 memory
  - 42.6 GB/s DDR3 bandwidth (1.333 GHz DDR3) (2 channels each with chip kill protection)
- Chip-to-chip networking
  - 5D Torus topology + external link
  - → 5 x 2 + 1 high speed serial links
  - each 2 GB/s send + 2 GB/s receive
  - DMA, remote put/get, collective operations
- External (file) IO -- when used as IO chip.
  - PCIe Gen2 x8 interface (4 GB/s Tx + 4 GB/s Rx)
  - re-uses 2 serial links
  - interface to Ethernet or Infiniband cards

© 2011 IBM Corporation

Sequoia	LLNL	96 racks	20.1 petaflops
Mira	ANL	24 racks	10.1 petaflops
****	RPI	2 racks	0.4 petaflops

Inter-Processor Communication

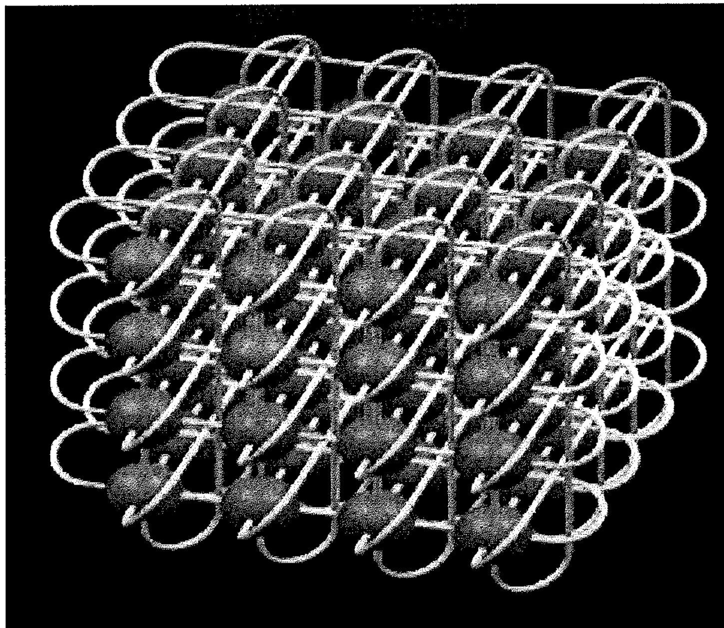


- **Integrated 5D torus**
  - Virtual Cut-Through routing
  - Hardware assists for collective & barrier functions
  - FP addition support in network
  - RDMA
    - Integrated on-chip Message Unit
- **2 GB/s raw bandwidth on all 10 links**
  - each direction -- i.e. 4 GB/s bidi
  - 1.8 GB/s user bandwidth
    - protocol overhead
- **5D nearest neighbor exchange measured at 1.76 GB/s per link (98% efficiency)**
- **Hardware latency**
  - Nearest: 80ns
  - Farthest: 3us
    (96-rack 20PF system, 31 hops)
- **Additional 11th link for communication to IO nodes**
  - BQC chips in separate enclosure
  - IO nodes run Linux, mount file system
  - IO nodes drive PCIe Gen2 x8 (4+4 GB/s)
    ↔ IB/10G Ethernet ↔ file system & world

Network Performance

- **All-to-all: 97% of peak**
- **Bisection: > 93% of peak**
- **Nearest-neighbor: 98% of peak**
- **Collective: FP reductions at 94.6% of peak**

© 2011 IBM Corporation



Summit at the Oak Ridge Leadership Computing Facility

Judy Hill
Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory

July 30, 2018

Argonne Training Program on Extreme-Scale Computing

ORNL is managed by UT-Battelle
for the US Department of Energy

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

What is the Leadership Computing Facility (LCF)?

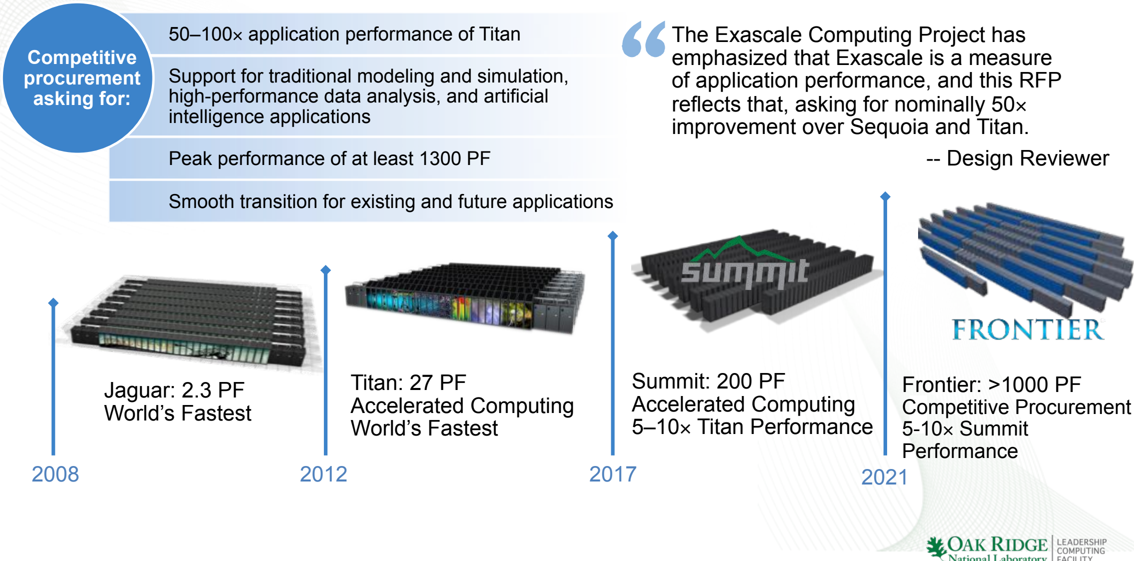
- Collaborative DOE Office of Science program at ORNL and ANL
- Mission: Provide the computational and data resources required to solve the most challenging problems.
- 2-centers/2-architectures to address diverse and growing computational needs of the scientific community
- Highly competitive user allocation programs (INCITE, ALCC).
- Projects receive 10x to 100x more resource than at other generally available centers.
- LCF centers partner with users to enable science & engineering breakthroughs (Liaisons, Catalysts).

OAK RIDGE National Laboratory

Argonne NATIONAL LABORATORY

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

OLCF Path to Exascale



Coming Soon: Summit is replacing Titan as the OLCF's leadership supercomputer

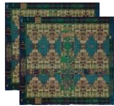
- Many fewer nodes
- Much more powerful nodes
- Much more memory per node and total system memory
- Faster interconnect
- Much higher bandwidth between CPUs and GPUs
- Much larger and faster file system

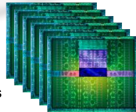
Feature	Titan	Summit
Application Performance	Baseline	5-10x Titan
Number of Nodes	18,688	4,608
Node performance	1.4 TF	42 TF
Memory per Node	32 GB DDR3 + 6 GB GDDR5	512 GB DDR4 + 96 GB HBM2
NV memory per Node	0	1600 GB
Total System Memory	710 TB	>10 PB DDR4 + HBM2 + Non-volatile
System Interconnect	Gemini (6.4 GB/s)	Dual Rail EDR-IB (25 GB/s)
Interconnect Topology	3D Torus	Non-blocking Fat Tree
Bi-Section Bandwidth	112 TB/s	115.2 TB/s
Processors	1 AMD Opteron™ 1 NVIDIA Kepler™	2 IBM POWER9™ 6 NVIDIA Volta™
File System	32 PB, 1 TB/s, Lustre®	250 PB, 2.5 TB/s, GPFS™
Power Consumption	9 MW	13 MW

Summit Overview

Components

IBM POWER9
- 22 Cores
- 4 Threads/core
- NVLink





NVIDIA GV100
- 7 TF
- 16 GB @ 0.9 TB/s
- NVLink

Compute Node

2 x POWER9
6 x NVIDIA GV100
NVMe-compatible PCIe 1600 GB SSD

25 GB/s EDR IB- (2 ports)
512 GB DRAM- (DDR4)
96 GB HBM- (3D Stacked)
Coherent Shared Memory

Compute Rack

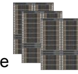
18 Compute Servers
Warm water (70°F direct-cooled components)
RDHX for air-cooled components

39.7 TB Memory/rack
55 KW max power/rack

Compute System

10.2 PB Total Memory
256 compute racks
4,608 compute nodes
Mellanox EDR IB fabric
200 PFLOPS
~13 MW

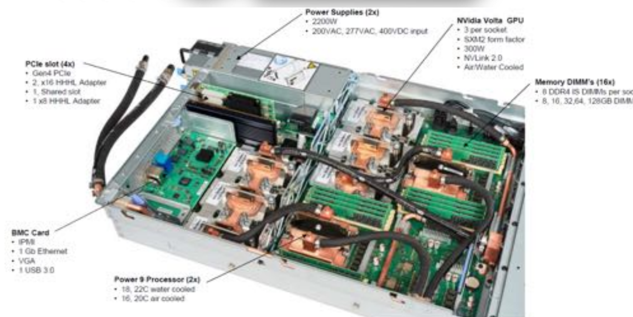
GPFS File System

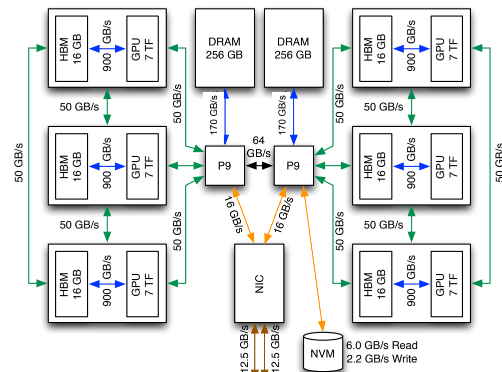


250 PB storage
2.5 TB/s read, 2.5 TB/s write
(**2.5 TB/s sequential and 2.2 TB/s random I/O)

Oak Ridge National Laboratory | Leadership Computing Facility

Summit Node Overview





TF	42 TF (6x7 TF)
HBM	96 GB (6x16 GB)
DRAM	512 GB (2x16x16 GB)
NET	25 GB/s (2x12.5 GB/s)
MMsg/s	83

- HBM/DRAM Bus (aggregate B/W)
- NVLINK
- X-Bus (SMP)
- PCIe Gen4
- EDR IB

HBM & DRAM speeds are aggregate (Read+Write).
All other speeds (X-Bus, NVLink, PCIe, IB) are bi-directional.

Oak Ridge National Laboratory | Leadership Computing Facility

Summit Node Overview: System Balance Ratios

	Summit	Titan
Memory subsystem to Intra-node connectivity ratios		
HBM BW : DDR BW	15.8	4.9
HBM BW : CPU-GPU BW	18	39
Per HBM BW : GPU-GPU BW	18	--
DDR BW : CPU-GPU BW	1.13	8
HBM capacity : GPU-GPU BW	0.32	--
Memory subsystem to FLOPS ratios		
Memory capacity : GFLOPS	0.01	0.03
Interconnect subsystem to FLOPS ratios		
Injection BW : GFLOPS	0.0006	0.004
Other ratios		
Filesystem : Memory capacity	89	42
FLOPS : Power (MW)	15.4	3

Reference: Vazhkudai, et. al. The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems. SC18 Proceedings. To appear.

IBM Power9 Processor

- Up to 24 cores
  - CORAL has 22 cores for yield optimization on first processors
- PCI-Express 4.0
  - Twice as fast as PCIe 3.0
- NVLink 2.0
  - Coherent, high-bandwidth links to GPUs
- 14nm FinFET SOI technology
  - 8 billion transistors
- Cache
  - L1I: 32 KiB per core, 8-way set associative
  - L1D: 32KiB per core, 8-way
  - L2: 258 KiB per core
  - L3: 120 MiB eDRAM, 20-way

Stream benchmark: Summit vs Titan

- A simple synthetic benchmark program that measures achievable memory bandwidth (in GB/s) under OpenMP threading.

System Cores	Peak (Summit) 44	Titan 16
Copy	274.6	34.9
Scale	271.4	35.3
Add	270.6	33.6
Triad	275.3	33.7
Peak (theoretical)	340	51.2
Fraction of Peak	82%	67%

DRAM Bandwidth

System	Peak (Summit)	Titan
Copy	789	181
Scale	788	181
Add	831	180
Triad	831	180
Peak (theoretical)	900	250
Fraction of Peak	92%	72%

GDDR Bandwidth

Slide courtesy of Wayne Joubert, ORNL

For Peak (Summit):
- GCC compiler
- Best result in 1000 tests
- Runtime variability up to 9%

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

NVIDIA Volta Details

	Tesla V100 for NVLink	Tesla V100 for PCIe
PERFORMANCE with NVIDIA GPU Boost™	DOUBLE-PRECISION 7.8 TeraFLOPS	DOUBLE-PRECISION 7 TeraFLOPS
	SINGLE-PRECISION 15.7 TeraFLOPS	SINGLE-PRECISION 14 TeraFLOPS
	DEEP LEARNING 125 TeraFLOPS	DEEP LEARNING 112 TeraFLOPS
INTERCONNECT BANDWIDTH Bi-Directional	NVLINK 300 GB/s	PCIE 32 GB/s
MEMORY CoWoS Stacked HBM2	CAPACITY 16 GB HBM2	
	BANDWIDTH 900 GB/s	

TensorCores™
Mixed Precision
(16b Matrix-Multiply-Add and 32b Accumulate)

Note: The performance numbers are peak and not representative of Summit's Volta

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

NVLink Bandwidth

- Measured from core 0 the achieved CPU-GPU NVLink rates with a modified bandwidthTest from NVIDIA CUDA Samples

GPU	0	1	2	3	4	5	peak
Host to Device	45.93	45.92	45.92	40.63	40.59	40.64	50
Device to Host	45.95	45.95	45.95	36.60	36.52	35.00	50
Bi-Directional	86.27	85.83	77.36	66.14	65.84	64.76	100

Single Node Single GPU NVLink Rates (GB/s)

- Not necessarily a use case that most applications will employ

Slide courtesy of Wayne Joubert, ORNL

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

NVLink Bandwidth

- Measured the achieved CPU-GPU NVLink rates with a modified bandwidthTest from NVIDIA CUDA Samples using multiple MPI process evenly spread between the sockets.

MPI Process Count	1	2	3	4	5	6	Peak (6)
Host to Device	45.93	91.85	137.69	183.54	229.18	274.82	300
Device to Host	45.95	91.90	137.85	183.80	225.64	268.05	300
Bi-Directional	85.60	172.59	223.54	276.34	277.39	278.07	600

NVLink Rates with MPI Processes (GB/s)

- Ultimately limited by the CPU memory bandwith
- 6 ranks driving 6 GPUs is an expected use case for many applications

Slide courtesy of Wayne Joubert, ORNL

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

NVLink Bandwidth

- Measured the achived NVLink transfer rates between GPUs, both within a socket and across them, using p2pBandwidthLatencyTest from NVIDIA CUDA Samples. (Peer-to-Peer communication turned on).

Socket	0	1	Cross	Peak
Uni-Directional	46.33	46.55	25.89	50
Bi-Directional	93.02	93.11	21.63	100

NVLink Rates for GPU-GPU Transfers (GB/s)

- Cross-socket bandwidth is much lower than that between GPUs attached to the same CPU socket

Slide courtesy of Wayne Joubert, ORNL

Summit Programming Environment

- Compilers supporting OpenMP and OpenACC
  - IBM XL, PGI, LLVM, GNU, NVIDIA
- Libraries
  - IBM Engineering and Scientific Subroutine Library (ESSL)
  - FFTW, ScaLAPACK, PETSc, Trilinos, BLAS-1,-2,-3, NVBLAS
  - cuFFT, cuSPARSE, cuRAND, NPP, Thrust
- Debugging
  - Allinea DDT, IBM Parallel Environment Runtime Edition (pdb)
  - Cuda-gdb, Cuda-memcheck, valgrind, memcheck, helgrind, stacktrace
- Profiling
  - IBM Parallel Environment Developer Edition (HPC Toolkit)
  - VAMPIR, Tau, Open|Speedshop, nvprof, gprof, Rice HPCToolkit

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Summit vs Titan PE comparison

Compiler	Titan	Summit
PGI	Yes	Yes
GCC	Yes	Yes
XL	No	Yes
LLVM	No	Yes
Cray	Yes	No
Intel	Yes	No

Debugger	Titan	Summit
DDT	Yes	Yes
cuda-gdb, -memcheck	Yes	Yes
Valgrind, memcheck, helgrind	Yes	Yes
Stack trace analysis tool	Yes	Yes
pdb	No	Yes

Performance Tools	Titan	Summit
Open|SpeedShop	Yes	Yes
TAU	Yes	Yes
CrayPAT	Yes	No
Reveal	Yes	No
HPCToolkit (IBM)	No	Yes
HPCToolkit (Rice)	Yes	Yes
VAMPIR	Yes	Yes
nvprof	Yes	Yes
gprof	Yes	Yes

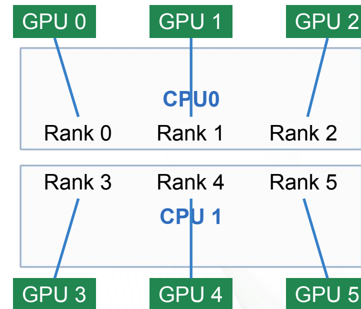
The majority of tools available on Titan are also available on Summit. A few transitions may be necessary.

Programming Multiple GPUs

- Multiple paths, with different levels of flexibility and sophistication
  - Simple model looks like Titan
  - Additional models expose the node-level parallelism mode directly
  - Low-level approaches are available, but not what we would recommend to users unless there is a particular reason
- Exposing more (node-level) parallelism is key to scalable applications from petascale up

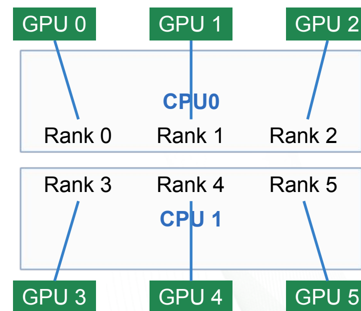
One GPU Per MPI Rank

- Deploy one MPI rank per GPU (6 per node)
  - Bind each rank to a specific GPU
- This model looks like Titan
- MPI ranks can use OpenMP (or pthreads) to utilize more of the CPU cores
  - CPU is only a small percentage of the total FLOPS



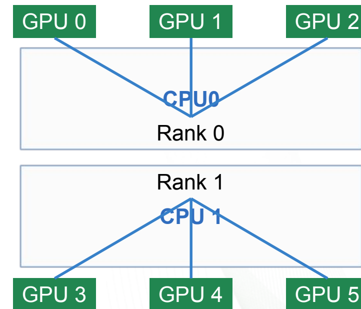
One GPU Per MPI Rank

- Expect this to be the most commonly used approach.
- Pros:
  - ✓ Straightforward extension for those already using Titan
- Cons:
  - Assumes similar amount of work to be done by all ranks
  - Potentially leaves a core on the Power9 unoccupied (or available to do something else)



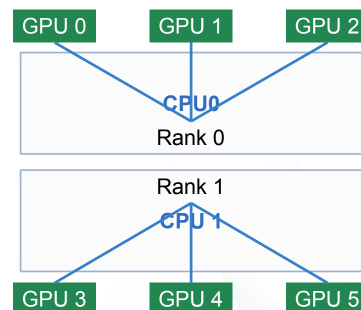
Multiple GPUs Per MPI Rank

- Deploy one MPI rank per 2-6 GPUs
  - Likely configurations:
    - 3 ranks/node (1:2)
    - 2 ranks/node (1:3)
    - 1 rank/node (1:6)
- Use threads and/or language constructs to offload to specific devices
- Multiple approaches possible, depending on language



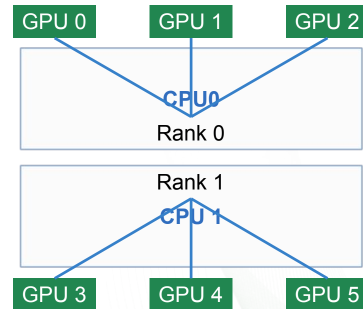
Multiple GPUs Per MPI Rank, Explicit Control

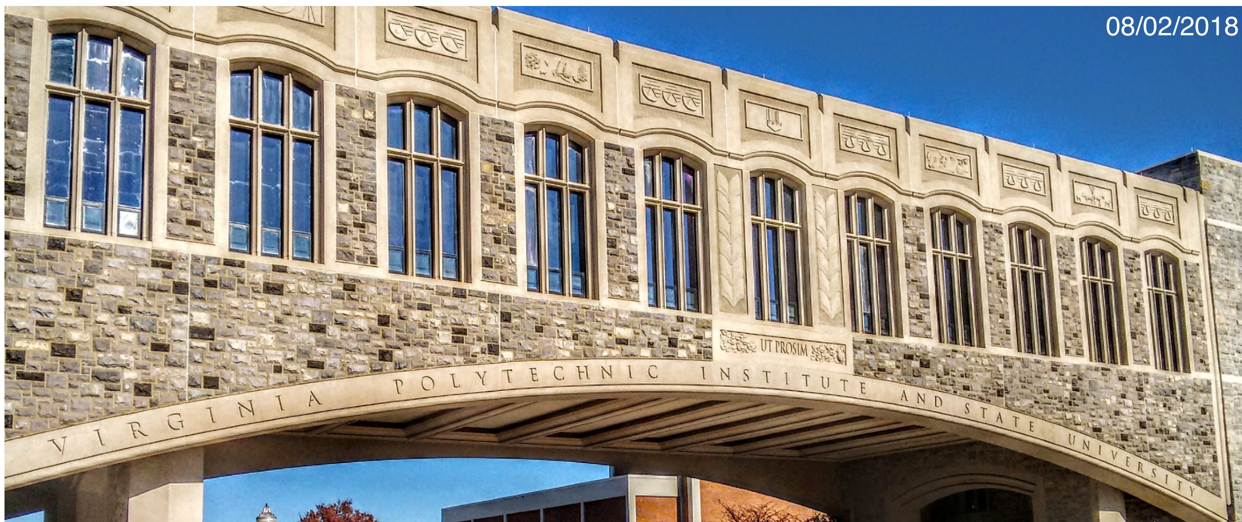
- OpenMP+OpenACC
  - Launch one OpenMP thread per GPU
  - Within each thread make OpenACC calls using `acc_set_device_num()`
- OpenMP 4 (accelerator target)
  - `device_num()` clause
- OpenACC
  - `acc_set_device_num()`
  - (Need to add similar clause for directives)
  - Eventually: compiler+runtime could break up large offload tasks across multiple GPUs automatically
- CUDA
  - `cudaSetDevice()` method



Multiple GPUs Per MPI Rank, Implicit Control

- OpenMP and OpenACC
  - Eventually: compiler+runtime could break up large offload tasks across multiple GPUs automatically
- Task-based execution models are available for CUDA, OpenMP and under development for OpenACC
  - Provide more flexibility to distribute work to multiple GPUs
- Multi-GPU aware libraries
  - CUBLAS
  - CUFFT



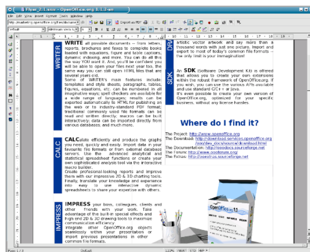
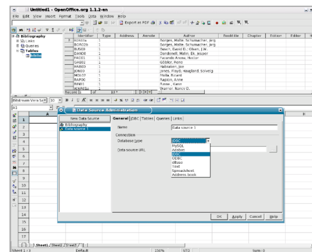
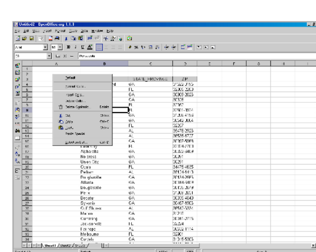


An Introduction to Graphics Processing Unit Architecture and Programming Models

Argonne Training Program on Exascale Computing

Tim Warburton
John K. Costain Faculty Chair in the College of Science
Professor Of Mathematics and Affiliate Faculty in CMDA
Virginia Tech

CPU: architecture follows purpose

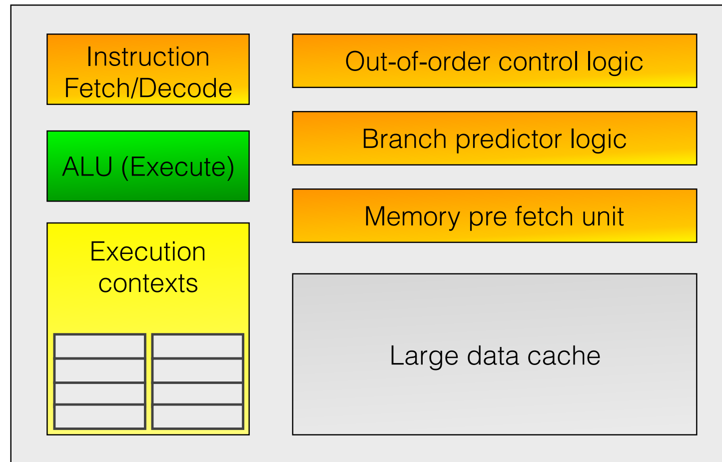




Original design goals for CPUs:

- Make single threads very fast.
  - Reduce latency through large caches.
  - Predict, speculate.

CPU: abstract modern architecture

Modern "CPU-Style" core design emphasizes
individual thread performance.



Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

Execution context: memory and hardware associated to a specific stream of instructions, e.g. registers.

GPU: massively parallel processing

http://developer.nvidia.com/object/gpu-gems-3.html

GPU: massively parallel compute

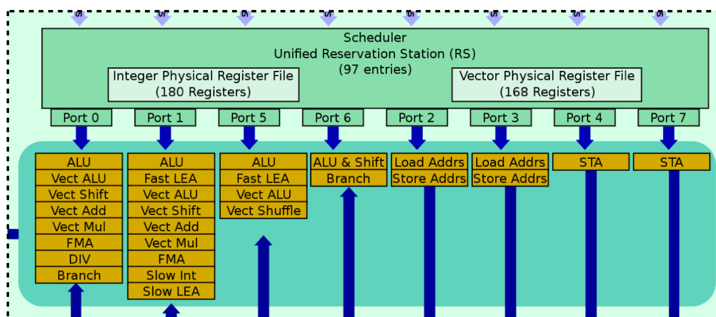
Design goals for GPUs:

- Throughput matters and single threads do not.
- Hide memory latency through parallelism.
- Let programmer deal with “raw” storage hierarchy.
- Avoid high frequency clock speed:
  - Desirable for portable devices, consoles, laptops…

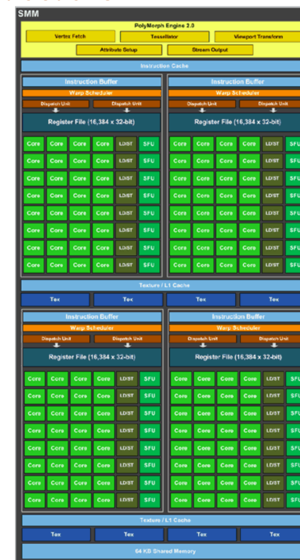
http://developer.nvidia.com/object/gpu-gems-3.html

CPU v GPU: fundamental difference #1

Each CPU core executes scalar or vector operations.
Each GPU core only executes vector instructions.



CPU: Single Instruction Multiple Data (SIMD) parallelism through ILP & vector execution units.



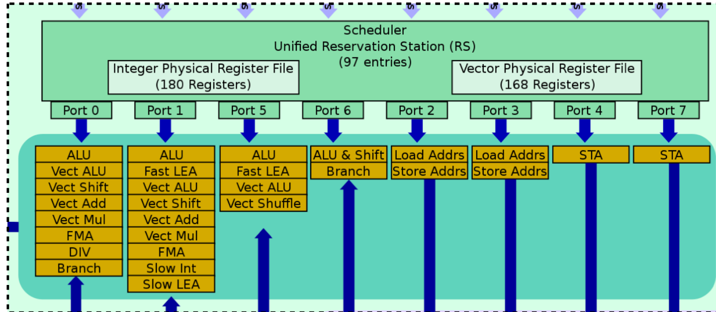
GPU: SIMD parallel execution of **all** operations

http://en.wikichip.org/wiki/intel/microarchitectures/skylake

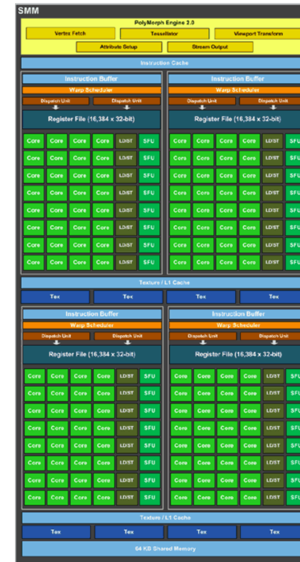
Compilers may need to be coaxed into generating vector instructions for CPU.
Recall: “Performance, SIMD, Vectorization and Performance Tuning” talk by James Reindeer.

CPU v GPU: fundamental difference #2

GPU cores are engineered to switch quickly
between threads to recover stalls



Skylake core: 180 Integer registers and
168 floating point registers



Maxwell core: 16K registers

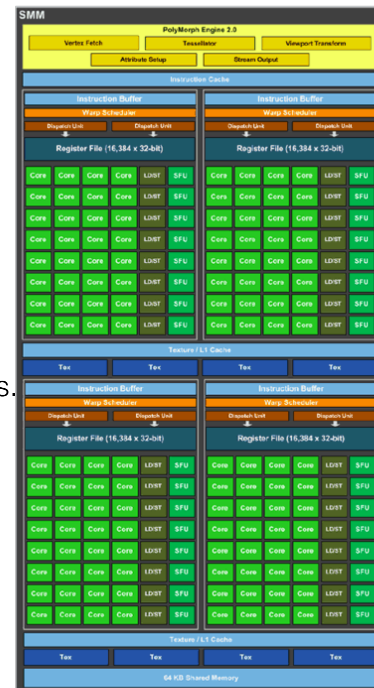
http://en.wikichip.org/wiki/intel/microarchitectures/skylake

Compilers may need to be coaxed into generating vector instructions for CPU.

GPU: summary of architecture

Summary of multi-level GPU parallel architecture

- A GPU has multiple cores and each core:
  - Has one (or more) wide SIMD vector units.
  - Wide SIMD vector units execute one instruction stream.
  - Has a pool of shared memory.
  - Shares a register file shared privately among all the ALUs.
  - Fast switches thread blocks to hide memory latency.
- Branching code (“ifs”) involves partial serialization.
- Nice summary:
  http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html



** SIMD width here is the number of ALUs in one of the core’s vector unit.*
The actual specifics vary but this is a good abstract viewpoint.

GPU: natural thread model

The GPU architecture admits a natural parallel threading model

- Programmer partitions a compute task into kernel code:
  - Programmer assigns kernel code to independent work-blocks:
    - Work-block assigned to a core with sufficient resources to process it:
      - Each core processes work-block kernel code with a work-group of “threads”
        - The work-group is batch processed in sub-groups of SIMD* work-items.
        - Each work-item processed by a “thread” passing through a SIMD lane.
        - A stalling SIMD group of “threads” is idled until it can continue.
        - “Threads” in a work-group can collaborate through shared memory.
        - The work-block stays resident until completed by core (using resources).
  - Main assumption: same instructions for independent work-groups.

** SIMD here is the number of ALUs in one of the core’s vector unit.*

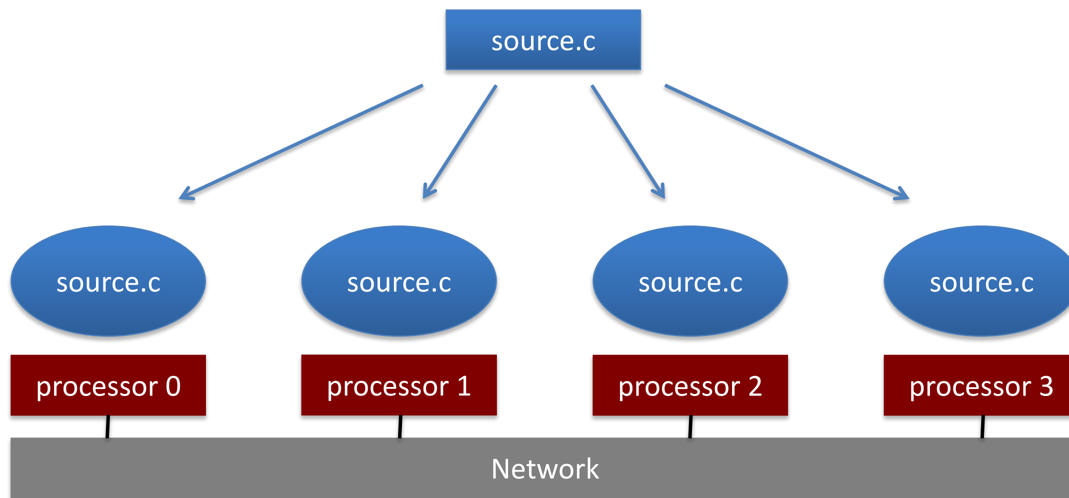
Parallel programming models

- Data Parallelism
  - Each processor performs the same task on different data
- Task Parallelism
  - Each processor performs a different task on different data
- Most applications fall between these two

Single Program Multiple Data

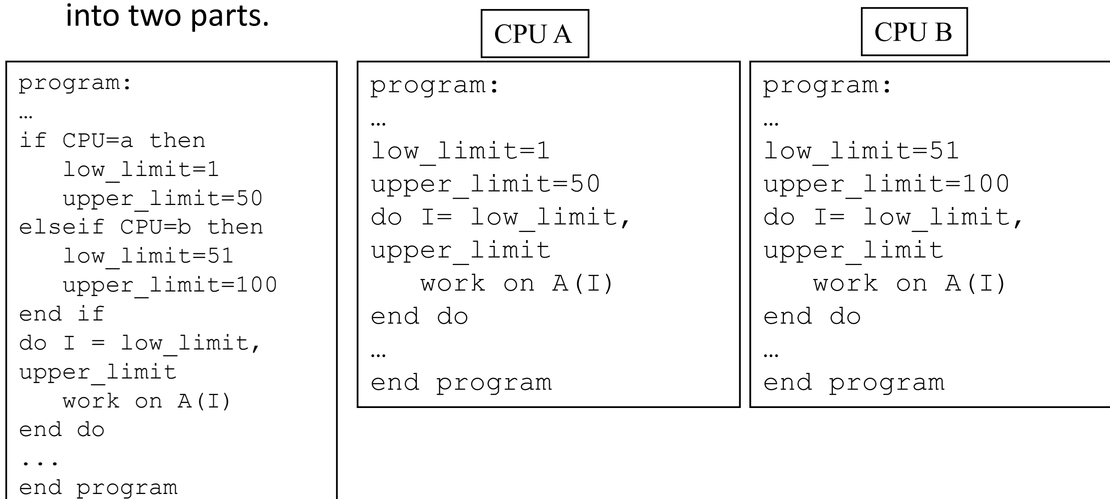
- SPMD: dominant programming model for shared and distributed memory machines.
  - One source code is written
  - Code can have conditional execution based on which processor is executing the copy
  - All copies of code start simultaneously and communicate and sync with each other periodically
- MPMD: more general, and possible in hardware, but no system/programming software enables it

SPMD Model



Data Parallel Programming Example

- One code will run on 2 CPUs
- Program has array of data to be operated on by 2 CPUs so array is split into two parts.



```
program:
…
if CPU=a then
   low_limit=1
   upper_limit=50
elseif CPU=b then
   low_limit=51
   upper_limit=100
end if
do I = low_limit,
upper_limit
   work on A(I)
end do
...
end program
```

CPU A

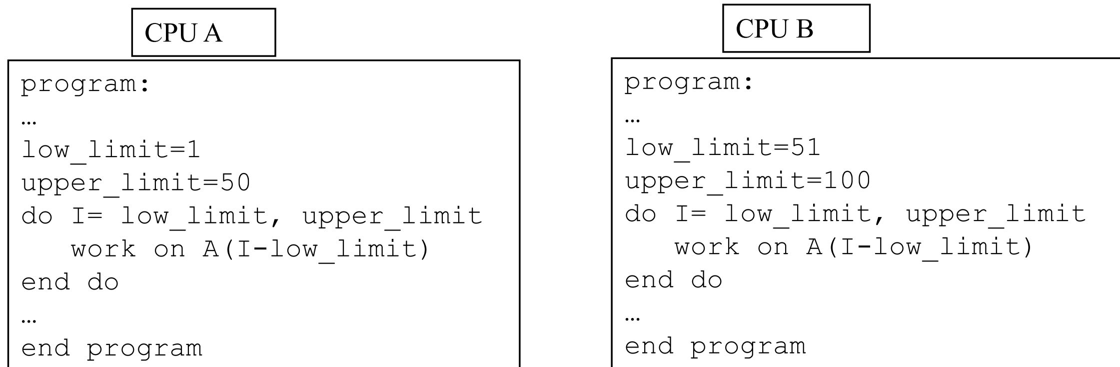
```
program:
…
low_limit=1
upper_limit=50
do I= low_limit,
upper_limit
   work on A(I)
end do
…
end program
```

CPU B

```
program:
…
low_limit=51
upper_limit=100
do I= low_limit,
upper_limit
   work on A(I)
end do
…
end program
```

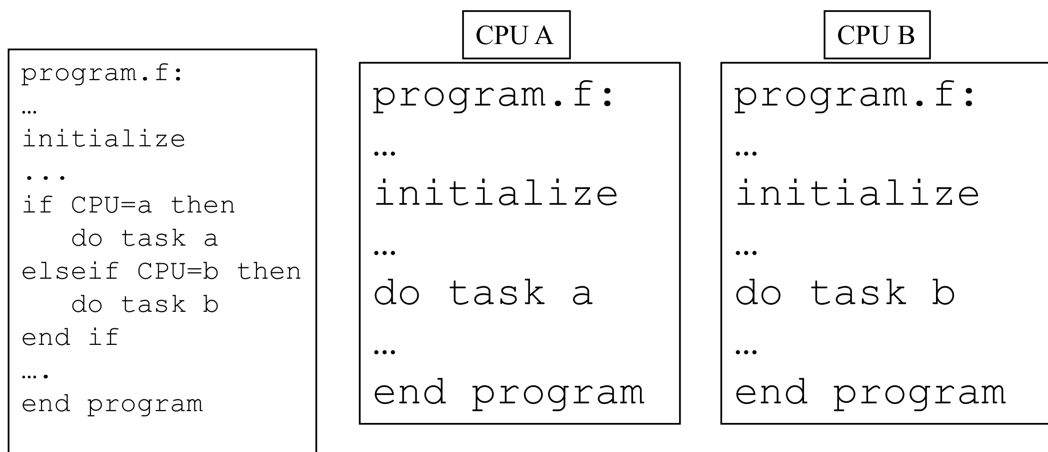
Distributed Data Parallel Programming

- Since each CPU has local address space: local indexing only



Task Parallel Programming Example

- One code will run on 2 CPUs
- Program has 2 tasks (a and b) to be done by 2 CPUs



Introduction to OpenMP

- Introduction
- OpenMP basics
- OpenMP directives, clauses, and library routines

Motivation

- Pthread is too tedious: explicit thread management is often unnecessary
  - Consider the matrix multiply example
    - We have a sequential code, we know which loop can be executed in parallel; the program conversion is quite mechanic: we should just say that the loop is to be executed in parallel and let the compiler do the rest.
    - OpenMP does exactly that!!!

What is OpenMP?

- What does OpenMP stands for?
  - Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia.
- OpenMP is an Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory parallelism.***
    - API components: Compiler Directives, Runtime Library Routines. Environment Variables
- OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors

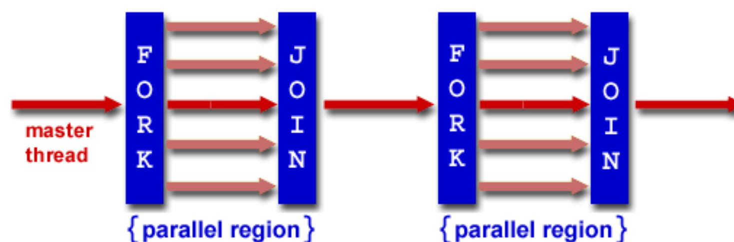
What is OpenMP?

- OpenMP API is specified for C/C++ and Fortran.
- OpenMP is not intrusive to the original serial code: instructions appear in comment statements for fortran and pragmas for C/C++.
- OpenMP website: http://www.openmp.org
  - Materials in this lecture are taken from various OpenMP tutorials in the website and other places.

Why OpenMP?

- OpenMP is portable: supported by HP, IBM, Intel, SGI, SUN, and others
  - It is the de facto standard for writing shared memory programs.
  - To become an ANSI standard?
- OpenMP can be implemented incrementally, one function or even one loop at a time.
  - A nice way to get a parallel program from a sequential program.

OpenMP execution model

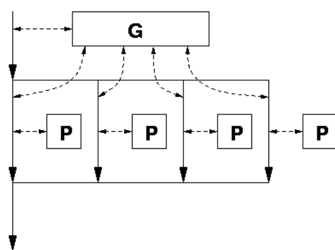


- OpenMP uses the fork-join model of parallel execution.
  - All OpenMP programs begin with a single master thread.
  - The master thread executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK).
  - When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread that executes sequentially (JOIN).

OpenMP general code structure

```
#include <omp.h>
main () {
  int var1, var2, var3;
  Serial code
  . . .
 /* Beginning of parallel section. Fork a team of threads. Specify variable scoping*/
  #pragma omp parallel private(var1, var2) shared(var3)
  {
    /* Parallel section executed by all threads */
    . . .
    /* All threads join master thread and disband*/
  }
  Resume serial code
  . . .
}
```

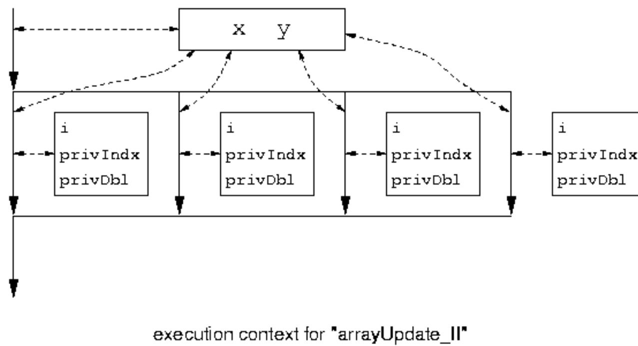
Data model



- Private and shared variables
  - Variables in the global data space are accessed by all parallel threads (shared variables).
  - Variables in a thread's private space can only be accessed by the thread (private variables)
    - several variations, depending on the initial values and whether the results are copied outside the region.

```
#pragma omp parallel for private( privIndx, privDbl )
  for ( i = 0; i < arraySize; i++ ) {
     for ( privIndx = 0; privIndx < 16; privIndx++ )
     { privDbl = ( (double) privIndx ) / 16;
      y[i] = sin( exp( cos( - exp( sin(x[i]) ) ) ) ) +
     cos( privDbl );
   }
}
```

Parallel for loop index is Private by default.



execution context for "arrayUpdate_II"

Sequential Matrix Multiply

```
For (I=0; I<n; I++)
   for (j=0; j<n; j++)
      c[I][j] = 0;
      for (k=0; k<n; k++)
         c[I][j] = c[I][j] + a[I][k] * b[k][j];
```

OpenMP Matrix Multiply

```
#pragma omp parallel for private(j, k)
For (I=0; I<n; I++)
    for (j=0; j<n; j++)
        c[I][j] = 0;
        for (k=0; k<n; k++)
            c[I][j] = c[I][j] + a[I][k] * b[k][j];
```

An Introduction to MPI

Parallel Programming with the Message Passing Interface

William Gropp
Ewing Lusk
Argonne National Laboratory

The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's.

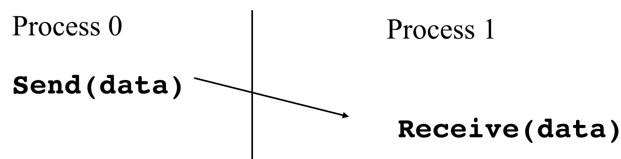
Types of Parallel Computing Models

- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
- Task Parallel - different instructions on different data (MIMD)
- SPMD (single program, multiple data) not synchronized at individual operation level
- SPMD is equivalent to MIMD since each MIMD program can be made SPMD (similarly for SIMD, but not in practical sense.)

Message passing (and MPI) is for MIMD/SPMD parallelism. HPF is an example of an SIMD interface.

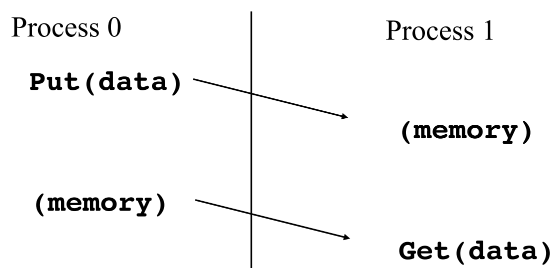
Cooperative Operations for Communication

- The message-passing approach makes the exchange of data *cooperative*.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiving process' s memory is made with the receiver' s explicit participation.
- Communication and synchronization are combined.



One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled
- One-sided operations are part of MPI-2.

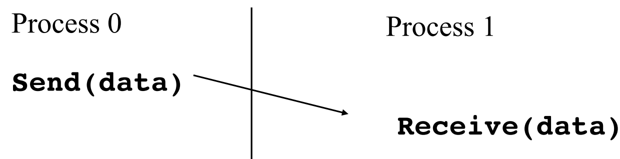


What is MPI?

- A *message-passing library specification*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

MPI Basic Send/Receive

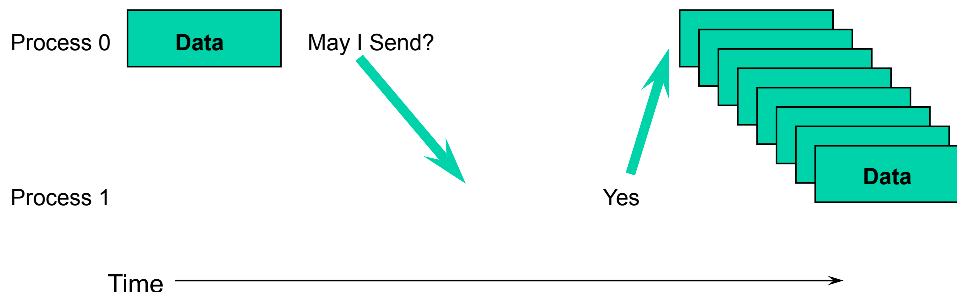
- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

What is message passing?

- Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

MPI Datatypes

- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  – predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  – a contiguous array of MPI datatypes
  – a strided block of datatypes
  – an indexed array of blocks of datatypes
  – an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive.
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes.

MPI Basic (Blocking) Send

MPI_SEND (start, count, datatype, dest, tag, comm)

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (on **`source`** and **`tag`**) message is received from the system, and the buffer can be used.
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**.
- **`status`** contains further information
- Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECV`**
- Point-to-point (send/recv) isn't the only way...

When to use MPI

- Portability and Performance
- Irregular Data Structures
- Building Tools for Others
  - Libraries
- Need to Manage memory on a per processor basis

Argonne NATIONAL LABORATORY

MPI for Scalable Computing

William Gropp[1] Rajeev Thakur[2] Pavan Balaji[2]

[1]University of Illinois

[2]Argonne National Laboratory

U.S. DEPARTMENT OF ENERGY

Timeline of the MPI Standard

- MPI-1 (1994), presented at SC'93
  - Basic point-to-point communication, collectives, datatypes, etc
- MPI-2 (1997)
  - Added parallel I/O, Remote Memory Access (one-sided operations), dynamic processes, thread support, C++ bindings, …
- ---- Unchanged for 10 years ----
- MPI-2.1 (2008)
  - Minor clarifications and bug fixes to MPI-2
- MPI-2.2 (2009)
  - Small updates and additions to MPI 2.1
- MPI-3.0 (2012)
  - Major new features and additions to MPI (nonblocking collectives, neighborhood collectives, improved RMA, tools interface, Fortran 2008 bindings, etc.)
- MPI-3.1 (2015)
  - Small updates to MPI 3.0

Understanding MPI Performance on Modern Processors

- MPI was developed when a single processor required multiple chips and most processors and nodes had a single core.
- Building effective, scalable applications requires having a model of how the system executes, how it performs, and what operations it can perform
  - This is (roughly) the *execution model* for the system, along with a *performance model*
- For decades, a simple model worked for designing and understanding MPI programs
  - Programs communicate either with point-to-point communication (send/recv), with a performance model of $T = s + r n$, where s is latency (startup) and r is inverse bandwidth (rate), or collective communication
- But today, processors are multi-core and many nodes are multi-chip.
  - How does that change how we think about performance and MPI?

Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in the communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECV`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

MPI Collective Communication

- Communication and computation is coordinated among a group of processes in a communicator
- Tags are not used; different communicators deliver similar functionality
- Non-blocking collective operations in MPI-3
- Three classes of operations: synchronization, data movement, collective computation

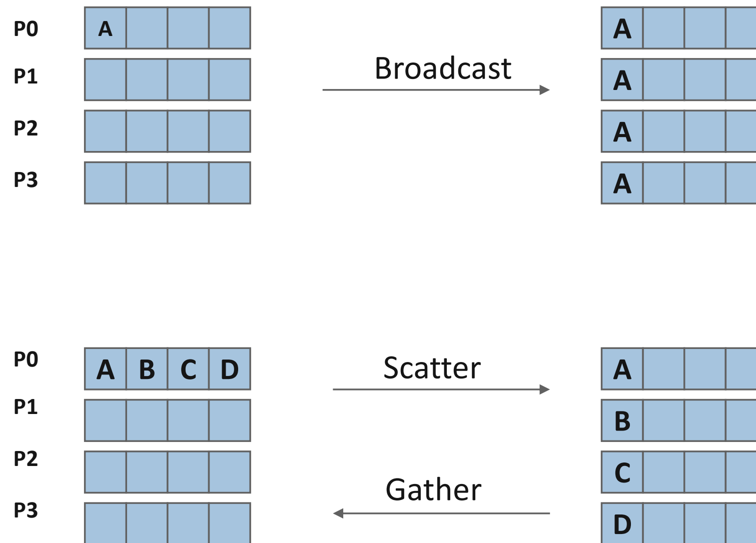
Synchronization

- `MPI_BARRIER(comm)`
  - Blocks until all processes in the group of communicator `comm` call it
  - A process cannot get out of the barrier until all other processes have reached barrier

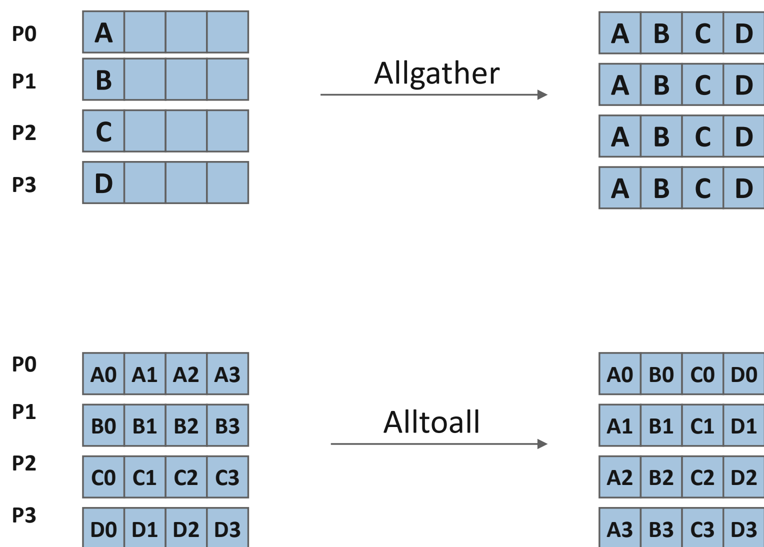
- Note that a barrier is rarely, if ever, necessary in an MPI program
- Adding barriers "just to be sure" is a bad practice and causes unnecessary synchronization. Remove unnecessary barriers from your code.

- One legitimate use of a barrier is before the first call to MPI_Wtime to start a timing measurement. This causes each process to start at *approximately* the same time.
- Avoid using barriers other than for this.

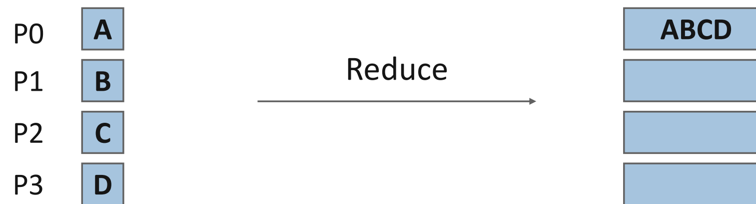
Collective Data Movement

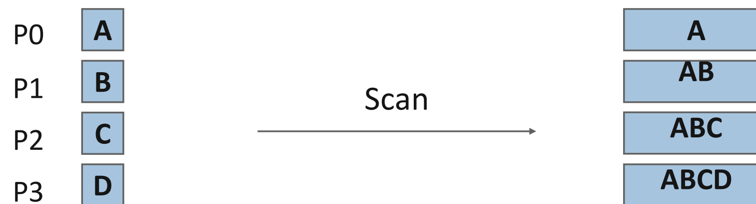


More Collective Data Movement



Collective Computation





MPI Collective Routines

- Many Routines, including: **MPI_ALLGATHER**, **MPI_ALLGATHERV**, **MPI_ALLREDUCE**, **MPI_ALLTOALL**, **MPI_ALLTOALLV**, **MPI_BCAST**, **MPI_EXSCAN**, **MPI_GATHER**, **MPI_GATHERV**, **MPI_REDUCE**, **MPI_REDUCE_SCATTER**, **MPI_SCAN**, **MPI_SCATTER**, **MPI_SCATTERV**
- "**All**" versions deliver results to all participating processes
- "**V**" versions (stands for vector) allow the chunks to have different sizes
- "**W**" versions for ALLTOALL allow the chunks to have different sizes in bytes, rather than units of datatypes
- **MPI_ALLREDUCE**, **MPI_REDUCE**, **MPI_REDUCE_SCATTER**, M**PI_REDUCE_SCATTER_BLOCK, MPI_EXSCAN**, and **MPI_SCAN** take both built-in and user-defined combiner functions

MPI Built-in Collective Computation Operations

`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_PROD`	Product
`MPI_SUM`	Sum
`MPI_LAND`	Logical and
`MPI_LOR`	Logical or
`MPI_LXOR`	Logical exclusive or
`MPI_BAND`	Bitwise and
`MPI_BOR`	Bitwise or
`MPI_BXOR`	Bitwise exclusive or
`MPI_MAXLOC`	Maximum and location
`MPI_MINLOC`	Minimum and location
`MPI_REPLACE`, `MPI_NO_OP`	Replace and no operation (RMA)

Defining your own Collective Operations

- Create your own collective computations with:

```
MPI_OP_CREATE(user_fn, commutes, &op);
MPI_OP_FREE(&op);

user_fn(invec, inoutvec, len, datatype);
```

- The user function should perform:

```
inoutvec[i]  =  invec[i]  op  inoutvec[i];
```

for i from 0 to len-1

- The user function can be non-commutative, but must be associative

Nonblocking Collectives

Nonblocking Collective Communication

- Nonblocking communication
  - Deadlock avoidance
  - Overlapping communication/computation
- Collective communication
  - Collection of pre-defined optimized routines
- Nonblocking collective communication
  - Combines both advantages
  - System noise/imbalance resiliency
  - Semantic advantages

Nonblocking Communication

- Semantics are simple:
  - Function returns no matter what
  - No progress guarantee!
- E.g., MPI_Isend(<send-args>, MPI_Request *req);
- Nonblocking tests:
  - Test, Testany, Testall, Testsome
- Blocking wait:
  - Wait, Waitany, Waitall, Waitsome

Nonblocking Collective Communication

- Nonblocking variants of all collectives
  - MPI_Ibcast(<bcast args>, MPI_Request *req);
- Semantics:
  - Function returns no matter what
  - No guaranteed progress (quality of implementation)
  - Usual completion calls (wait, test) + mixing
  - Out-of order completion
- Restrictions:
  - No tags, in-order matching
  - Send and vector buffers may not be touched during operation
  - MPI_Cancel not supported
  - No matching with blocking collectives

Hoefler et al.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI

Nonblocking Collective Communication

- Semantic advantages:
  - Enable asynchronous progression (and manual)
    - Software pipelining
  - Decouple data transfer and synchronization
    - Noise resiliency!
  - Allow overlapping communicators
    - See also neighborhood collectives
  - Multiple outstanding operations at any time
    - Enables pipelining window

Hoefler et al.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI

A Non-Blocking Barrier?

- What can that be good for? Well, quite a bit!
- Semantics:
  - MPI_Ibarrier() – calling process entered the barrier, **no** synchronization happens
  - Synchronization **may** happen asynchronously
  - MPI_Test/Wait() – synchronization happens **if** necessary
- Uses:
  - Overlap barrier latency (small benefit)
  - Use the split semantics! Processes **notify** non-collectively but **synchronize** collectively!

Nonblocking And Collective Summary

- Nonblocking communication
  - Overlap and relax synchronization
- Collective communication
  - Specialized pre-optimized routines
  - Performance portability
  - Hopefully transparent performance
- They can be composed
  - E.g., software pipelining

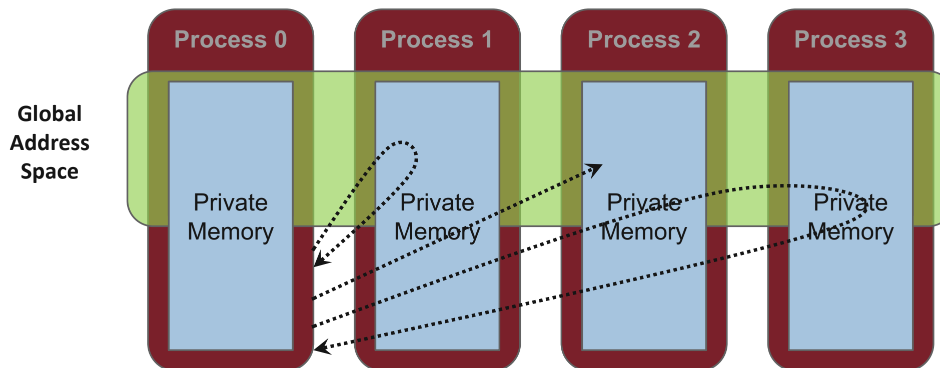
Argonne NATIONAL LABORATORY

Advanced Topics: One-sided Communication

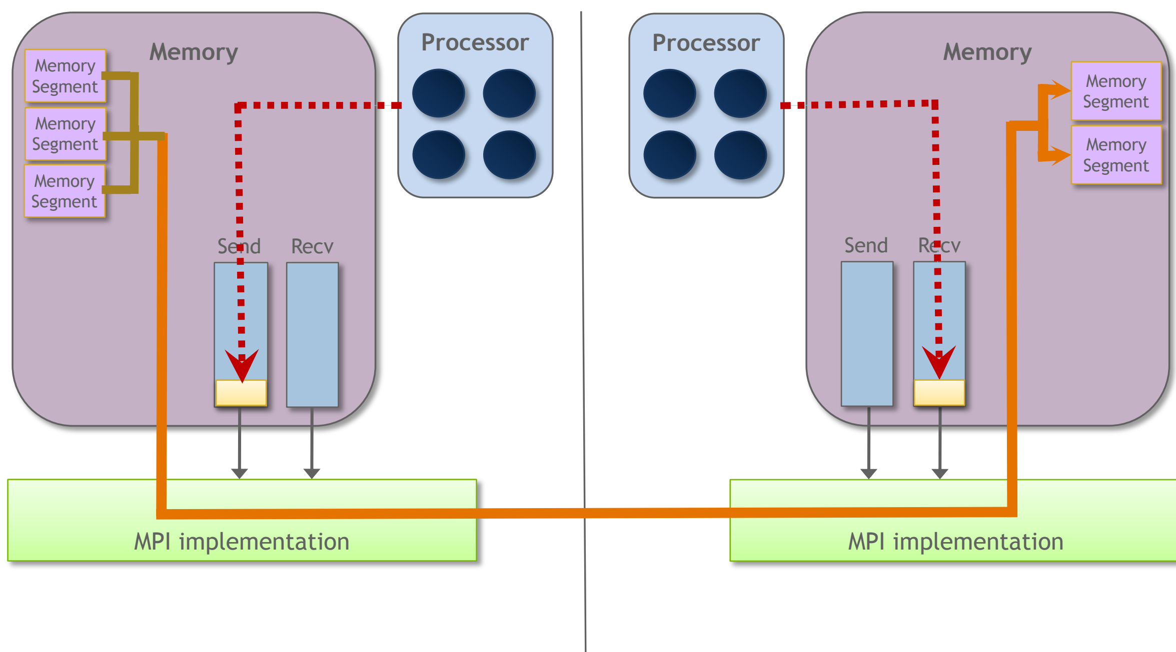
U.S. DEPARTMENT OF ENERGY

One-sided Communication

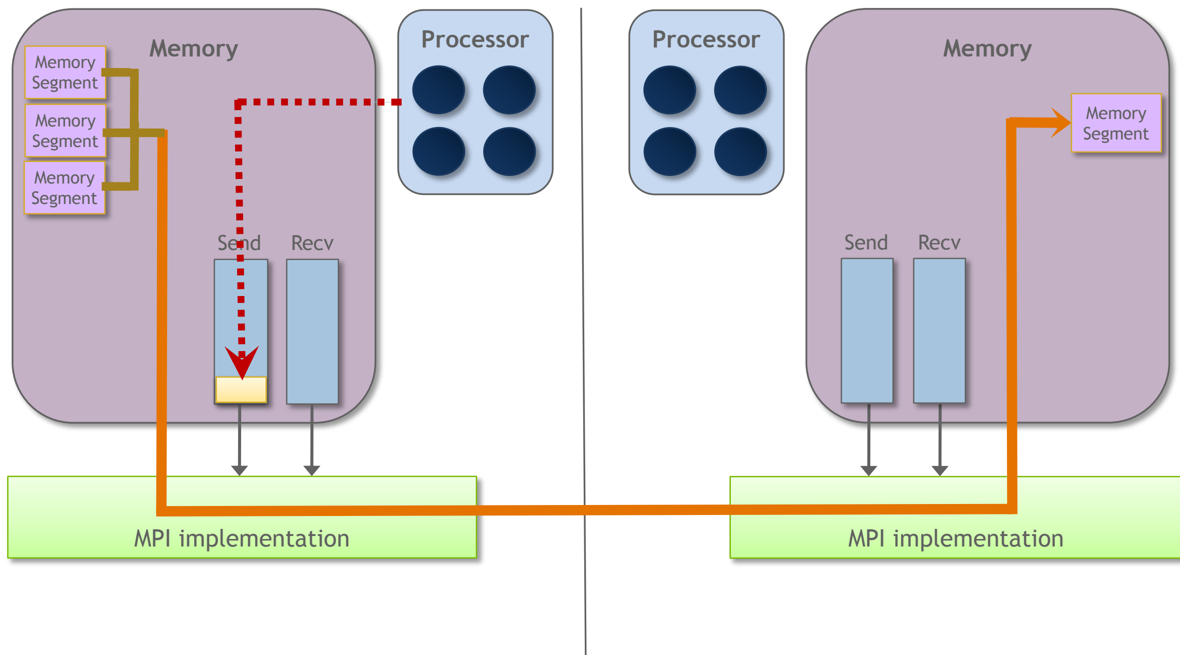
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory



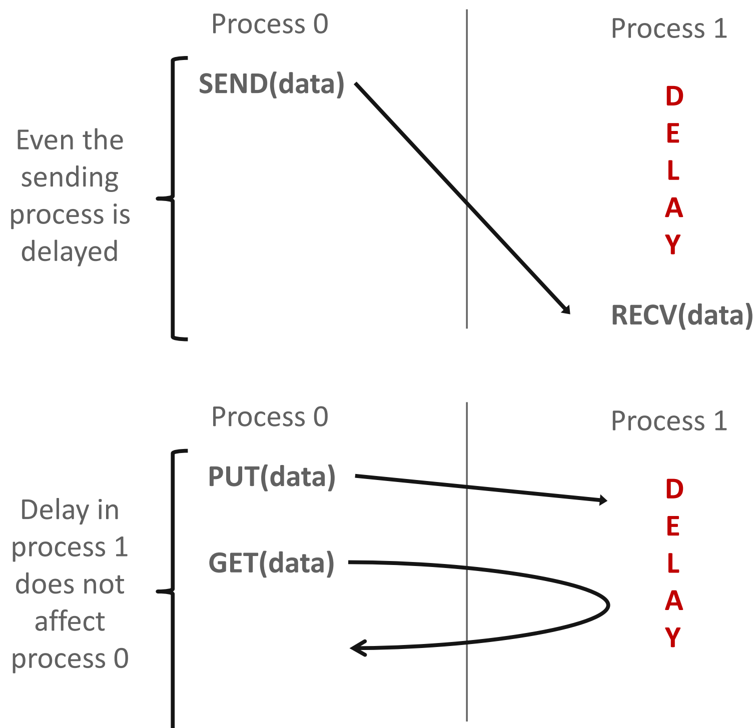
Two-sided Communication Example



One-sided Communication Example

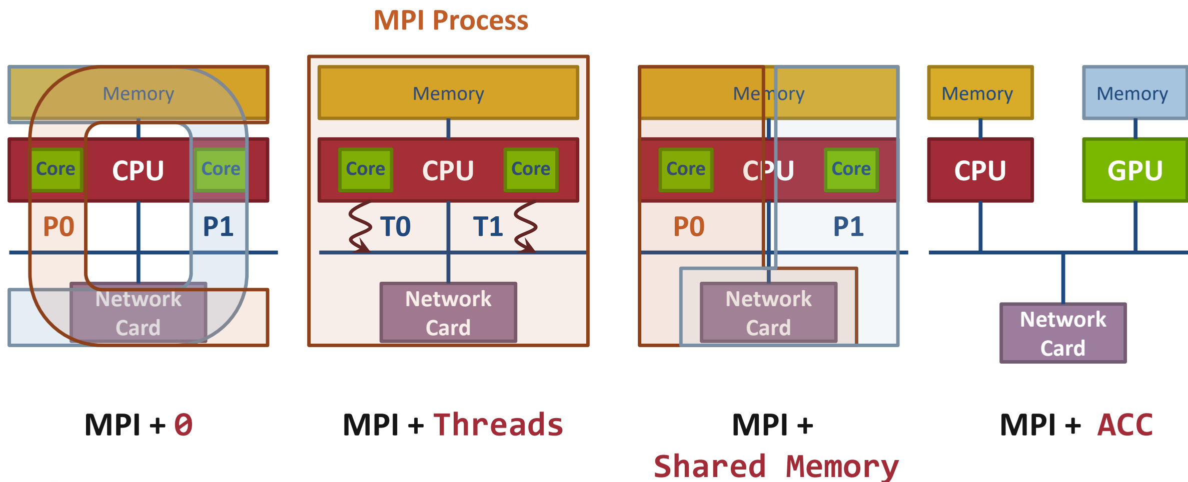


Comparing One-sided and Two-sided Programming



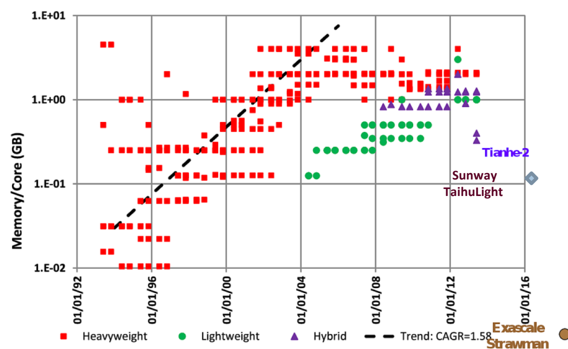
Hybrid MPI + X : Most Popular Forms

MPI + X



Why Hybrid MPI+X? Towards Strong Scaling (1/3)

- Strong scaling applications is increasing in importance
  - Hardware limitations: not all resources scale at the same rate as cores (e.g., memory capacity, network resources)
  - Desire to solve the same problem faster on a bigger machine
    - Nek5000, HACC, LAMMPS



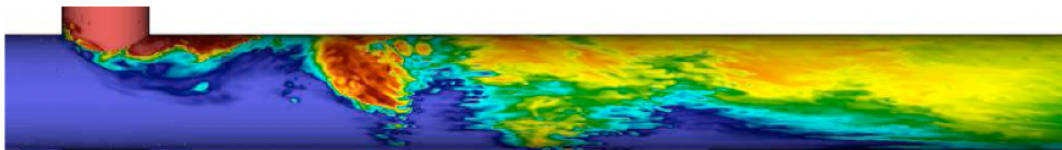
Evolution of the memory capacity per core in the Top500 list (Peter Kogge. PIM & memory: The need for a revolution in architecture.)

- Strong scaling pure MPI applications is getting harder
  - On-node communication is costly compared to load/stores
  - O(Px) communication patterns (e.g., All-to-all) costly

Why Hybrid MPI+X? Towards Strong Scaling (2/3)

- MPI+X benefits (X= {threads,MPI shared-memory, etc.})
  - Less memory hungry (MPI runtime consumption, O(P) data structures, etc.)
  - Load/stores to access memory instead of message passing
  - P is reduced by constant C (#cores/process) for O(Px) communication patterns
- Example 1: the Nek5000 team is working at the strong scaling limit

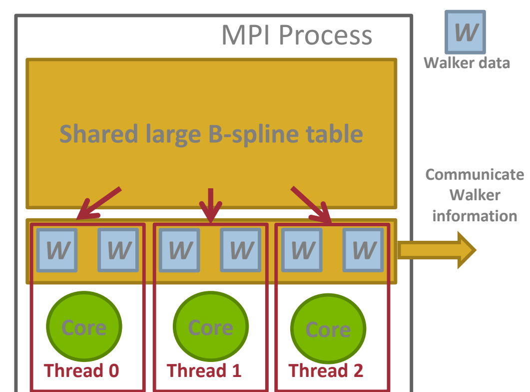
Nek5000



Why Hybrid MPI+X? Towards Strong Scaling (3/3)

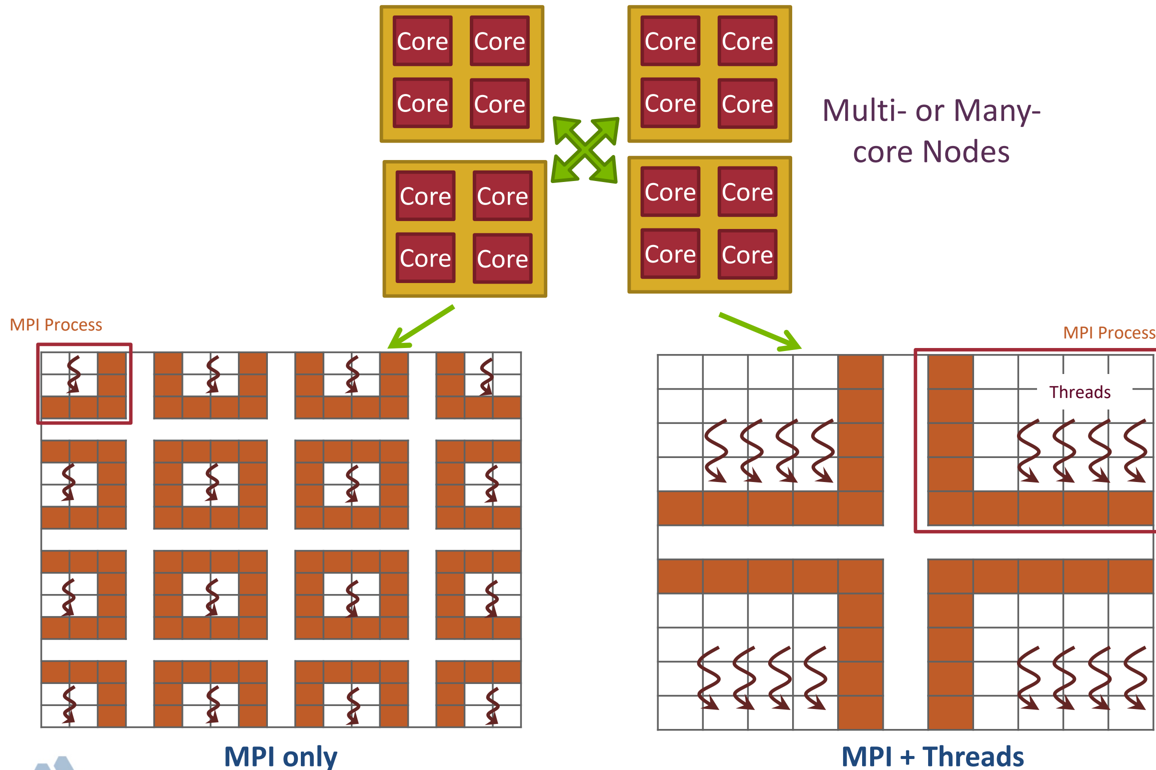
- Example 2: Quantum Monte Carlo Simulation (QCMPACK)
  - Size of the physical system to simulate is bound by memory capacity [1]
  - Memory space dominated by large interpolation tables (typically several GB of storage)
  - Threads are used to share those tables
  - Memory for communication buffers must be kept low to be allow simulation of larger and highly detailed simulations.

QMCPACK



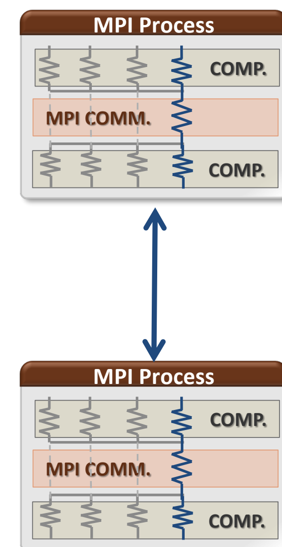
[1] Kim, Jeongnim, et al. "Hybrid algorithms in quantum Monte Carlo." Journal of Physics, 2012.

MPI + Threads: How To? (1/3)

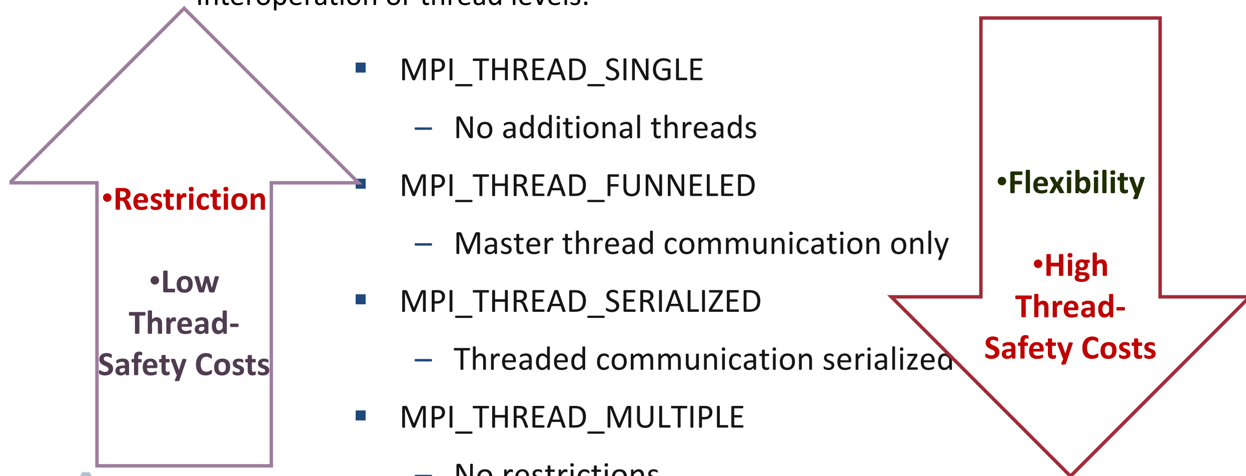


MPI + Threads: How To? (2/3)

- MPI describes parallelism between *processes* (with separate address spaces)
- *Thread* parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.



MPI + Threads: How To? (3/3)

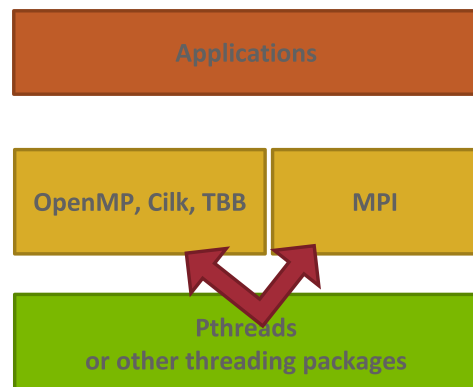


Interoperation or thread levels:

- MPI_THREAD_SINGLE
  - No additional threads
- MPI_THREAD_FUNNELED
  - Master thread communication only
- MPI_THREAD_SERIALIZED
  - Threaded communication serialized
- MPI_THREAD_MULTIPLE
  - No restrictions

MPI+OpenMP correctness semantics

- For OpenMP threads, the MPI+OpenMP correctness semantics are similar to that of MPI+threads
  - Caution: OpenMP iterations need to be carefully mapped to which thread executes them (some schedules in OpenMP make this harder)
- For OpenMP tasks, the general model to use is that an OpenMP thread can execute one or more OpenMP tasks
  - An MPI blocking call should be assumed to block the entire OpenMP thread, so other tasks might not get executed

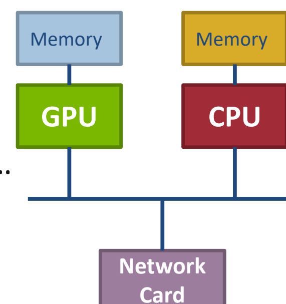


Hybrid Programming with Shared Memory

- MPI-3 allows different processes to allocate shared memory through MPI
  - MPI_Win_allocate_shared
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads
  - Because memory locality is clear (needed for performance) and data sharing is explicit

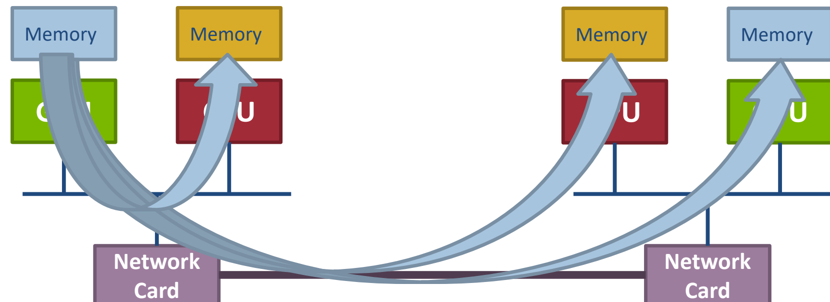
Accelerators in Parallel Computing

- General purpose, highly parallel processors
  - High FLOPs/Watt
  - Unit of execution *Kernel*
  - Separate physical memory subsystems
  - Programming Models: OpenAcc, CUDA, OpenCL, …
- Clusters with accelerators are becoming common
- New programmability and performance challenges for programming models and runtime systems



MPI + Accelerator Programming Examples

How to move data between GPUs with MPI?



Real answer: It depends on what GPU library, what hardware and what MPI implementation you are using

Simple answer: For modern GPUs, “just like you would with a non-GPU machine”

Section Summary

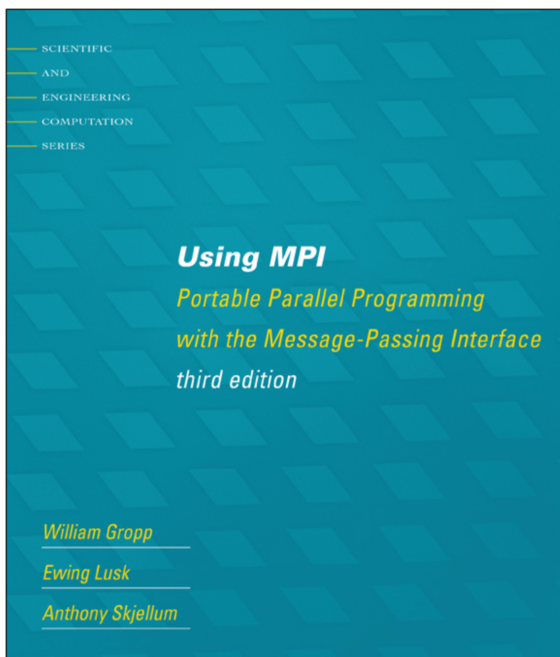
- Programming with accelerators is becoming increasingly important
- MPI is playing its role in enabling the usage of accelerators across distributed memory nodes
- The situation with MPI + GPU support is improving in both MPI implementations and in GPU hardware/software capabilities

Web Pointers

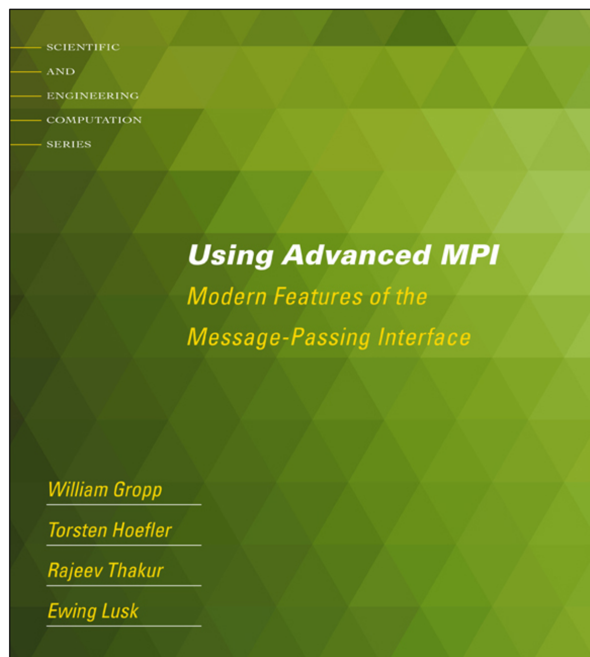
- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/

- MPI implementations:
  - MPICH : http://www.mpich.org
  - MVAPICH : http://mvapich.cse.ohio-state.edu/
  - Intel MPI: http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI: www.microsoft.com/en-us/download/details.aspx?id=39961
  - Open MPI : http://www.open-mpi.org/
  - IBM MPI, Cray MPI, HP MPI, TH MPI, NEC MPI, Fujitsu MPI, …
- Several MPI tutorials can be found on the web

Tutorial Books on MPI



Basic MPI



Advanced MPI, including MPI-3