LSU

LSU INFORMATION TECHNOLOGY SERVICES

Parallel Applications on Distributed Memory Systems

Le Yan

HPC User Services @ LSU

LSU CENTER FOR COMPUTATION & TECHNOLOGY

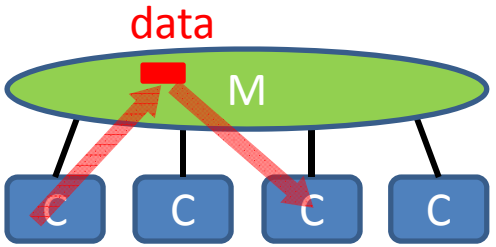
LONI

Outline

- Distributed memory systems
- Message Passing Interface (MPI)
- Parallel applications

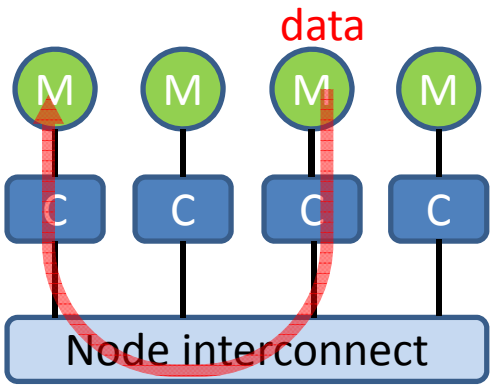
Shared memory model

- All threads can access the global address space
- Data sharing achieved via writing to/reading from the same memory location
- Example: OpenMP



Distributed memory systems (1)

- Each process has its own address space
  - Data is local to each process
- Data sharing achieved via explicit message passing (through network)
- Example: MPI (Message Passing Interface)

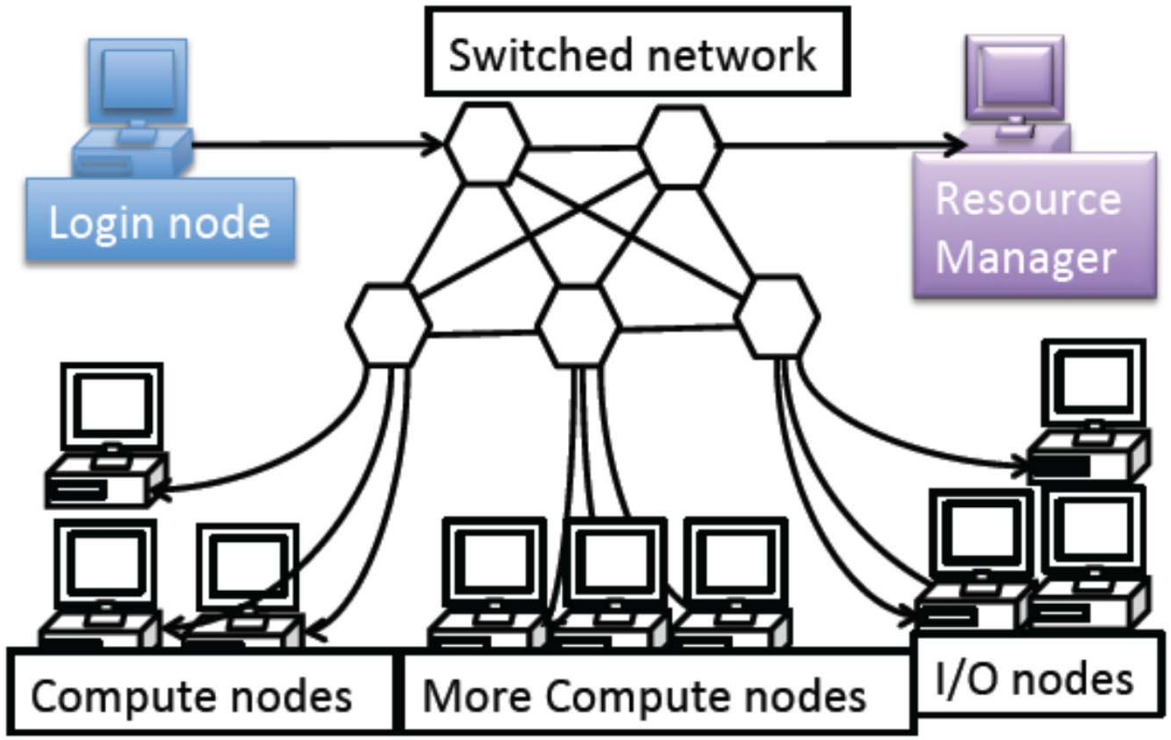


Distributed memory systems (2)

- Advantages
  - Memory is scalable with the number of processors
  - Cost effective (compared to big shared memory systems)
- Disadvantages
  - Explicit data transfer – programmers are responsible for moving data around
- The Top 500 list is dominated by distributed memory systems nowadays

Distributed Memory Systems (3)

- Login nodes
  - Users use these nodes to access the system
- Compute nodes
  - Run user jobs
  - Not accessible from outside
- I/O nodes
  - Serve files stored on disk arrays over the network
  - Not accessible from outside either



Distributed Memory System: Shelob

- LSU HPC system
- 32 nodes
  - Each node is equipped with
    - Two 8-core Intel “Sandy Bridge” processors
    - 64 GB memory
    - Two Nvidia K20 “Kepler” GPUs
- FDR Infiniband interconnect (56 Gbps)

Distributed Memory System: SuperMIC

- LSU HPC System
- 380 nodes
  - Each node is equipped with
    - Two 10-core Intel “Ivy Bridge” processors
    - 64 GB memory
    - Two Intel Xeon Phi coprocessors
- FDR Infiniband network interface (56 Gbps)

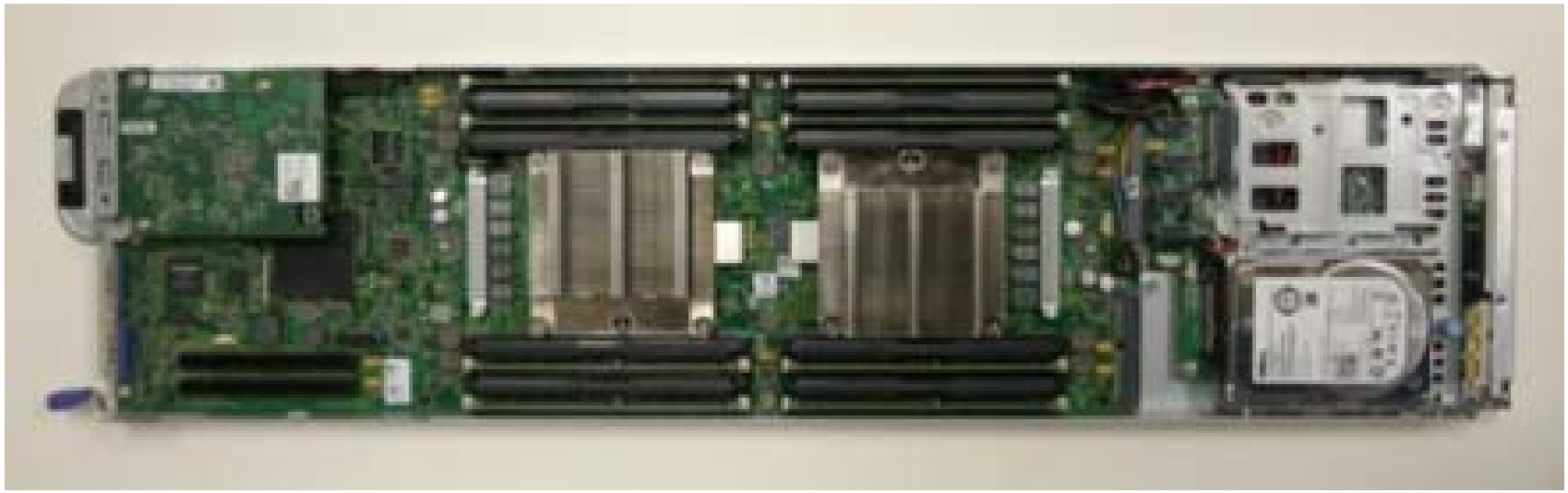
Distributed Memory System: Stampede

- Texas Advanced Computing Center system
- 6400 nodes
  - Intel “Sandy Bridge” processors and Xeon Phi coprocessors
- Infiniband interconnect
  - 75 miles of network cable

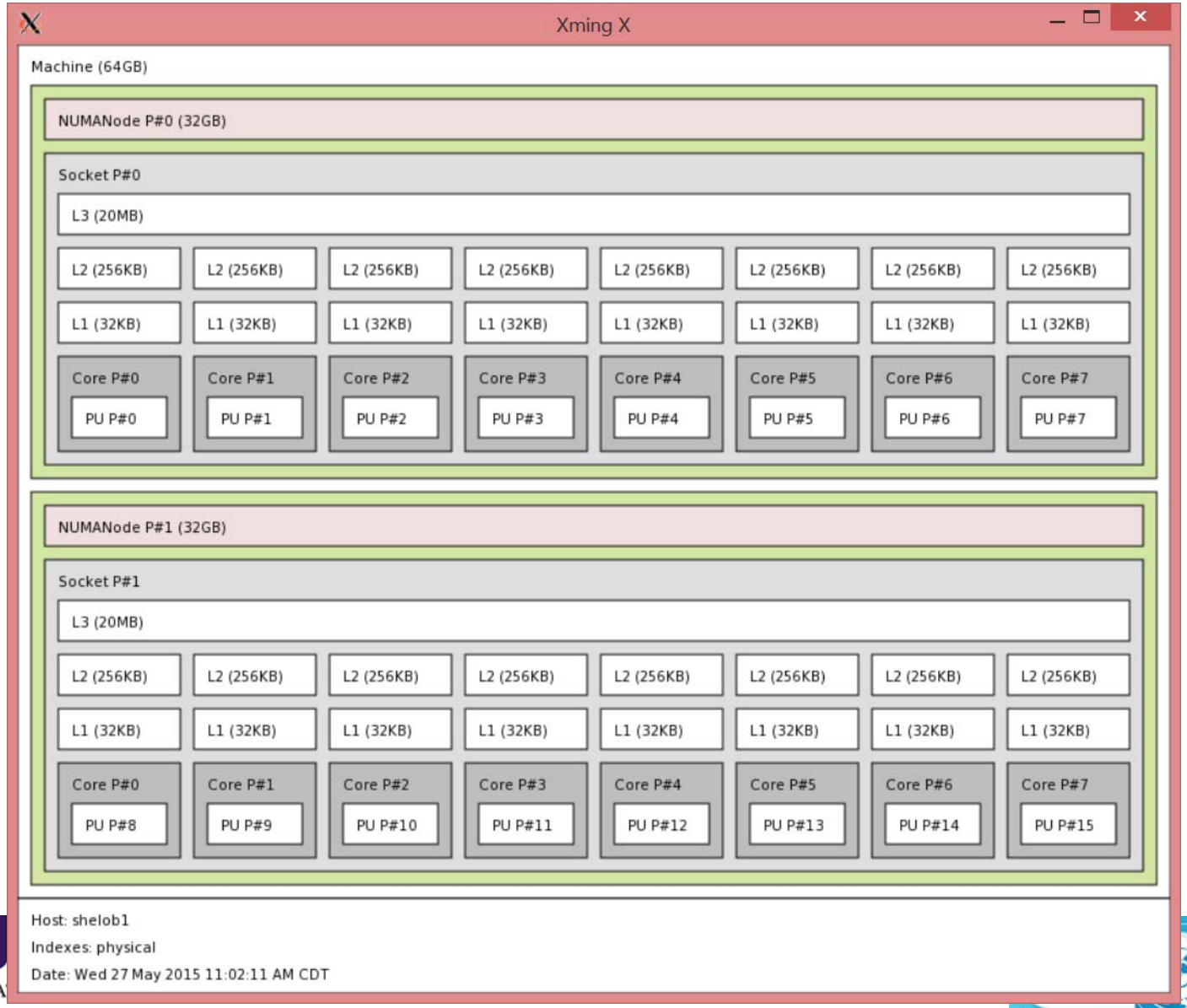
Source: https://www.tacc.utexas.edu/stampede/

HPC Jargon Soup

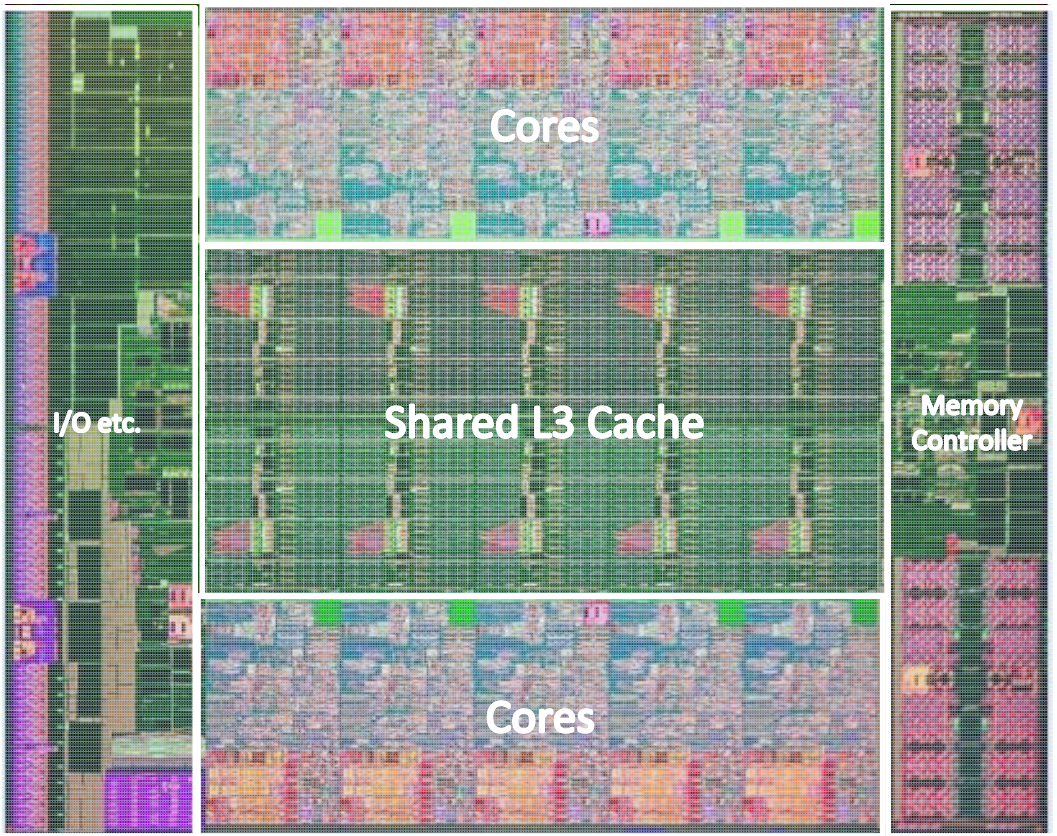
- Nodes, sockets, processors, cores, memory, cache, register … what?



Source: https://portal.tacc.utexas.edu/user-guides/stampede

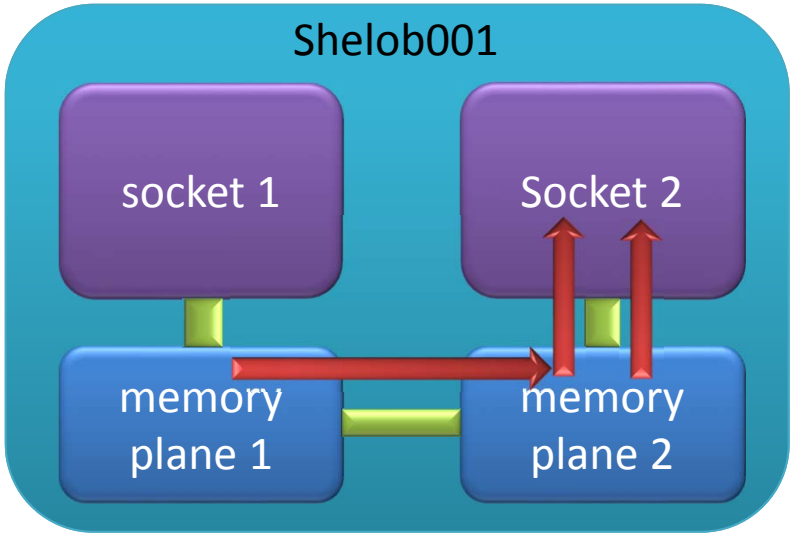

Xming X
Machine (64GB)
NUMANode P#0 (32GB)
Socket P#0
L3 (20MB)
L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)
L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB)
Core P#0 PU P#0
Core P#1 PU P#1
Core P#2 PU P#2
Core P#3 PU P#3
Core P#4 PU P#4
Core P#5 PU P#5
Core P#6 PU P#6
Core P#7 PU P#7
NUMANode P#1 (32GB)
Socket P#1
L3 (20MB)
L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)
L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB)
Core P#0 PU P#8
Core P#1 PU P#9
Core P#2 PU P#10
Core P#3 PU P#11
Core P#4 PU P#12
Core P#5 PU P#13
Core P#6 PU P#14
Core P#7 PU P#15
Host: shelob1
Indexes: physical
Date: Wed 27 May 2015 11:02:11 AM CDT

Inside of the Intel "Ivy Bridge" Processor



Source: http://www.theregister.co.uk/2013/09/10/intel_ivy_bridge_xeon_e5_2600_v2_launch/

NUMA Architecture

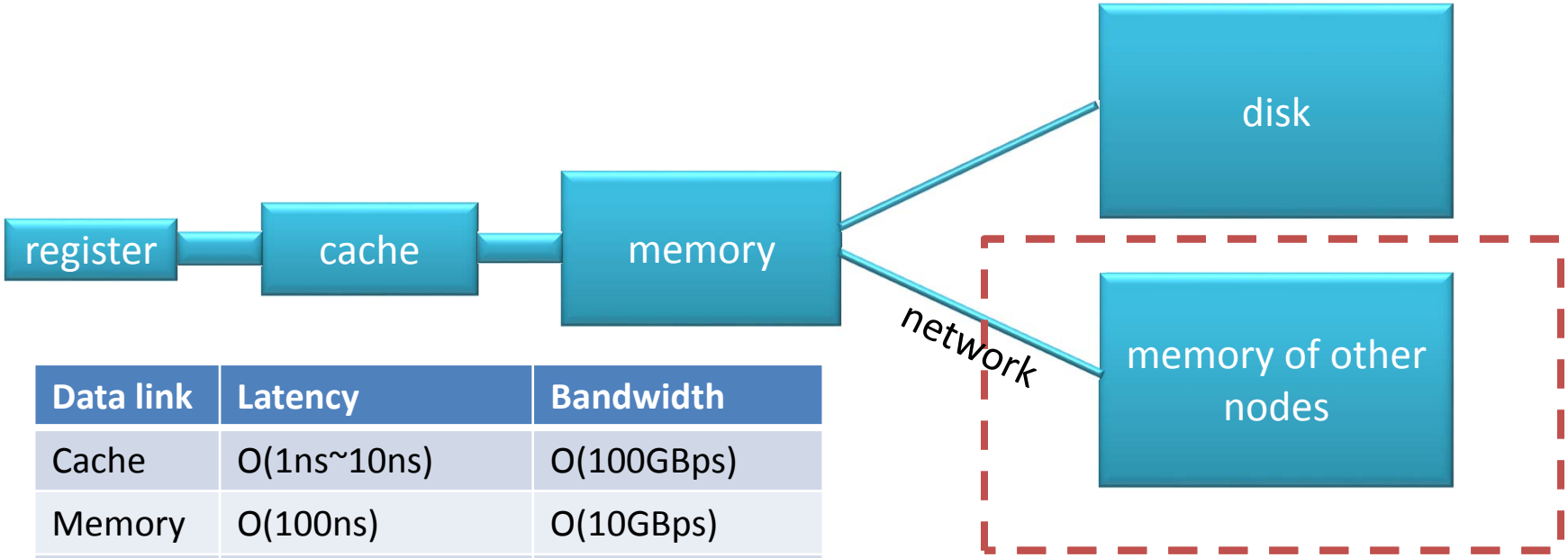



```
[lyan1@shelob1 ~]$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 32739 MB
node 0 free: 29770 MB
node 1 cpus: 8 9 10 11 12 13 14 15
node 1 size: 32768 MB
node 1 free: 26576 MB
node distances:
node   0   1
  0:  10  11
  1:  11  10
```

- Non-Uniform Memory Access
  - Most likely ccNUMA (cache coherent NUMA) these days
  - Not all cores can access any memory location with same speed

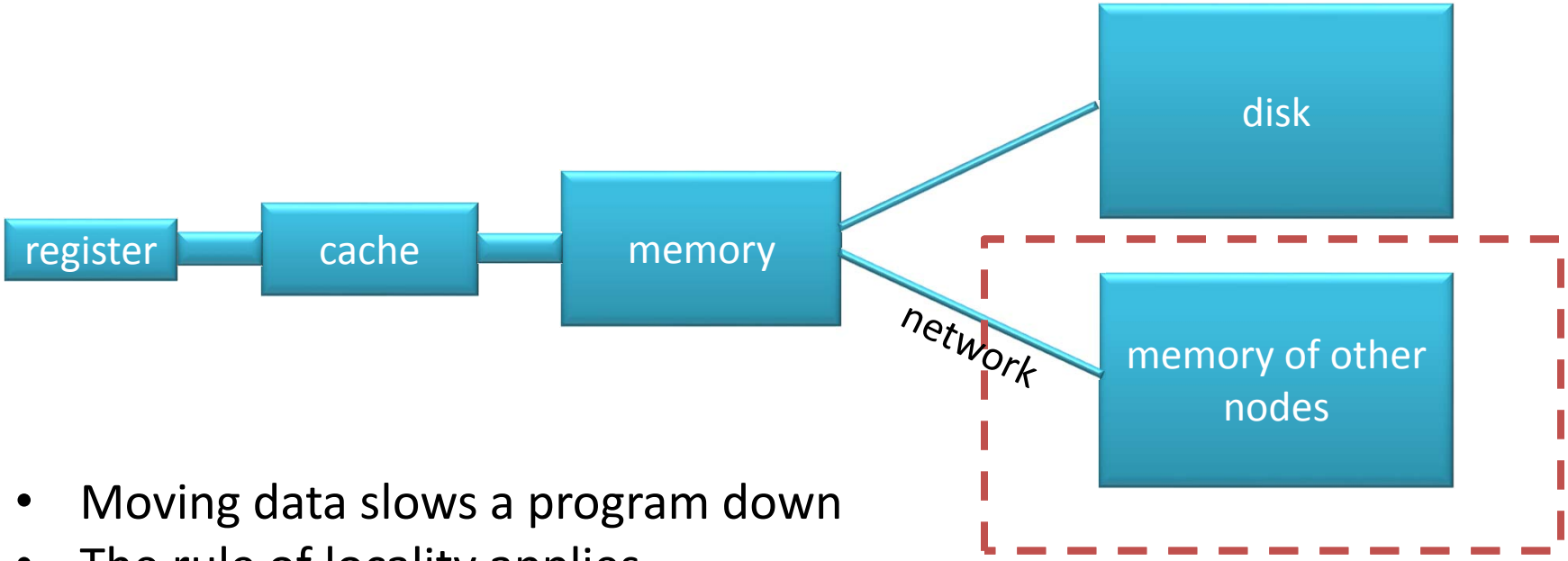
LSU CENTER FOR COMPUTATION & TECHNOLOGY

Memory Hierarchy



Data link	Latency	Bandwidth
Cache	O(1ns~10ns)	O(100GBps)
Memory	O(100ns)	O(10GBps)
Disk I/O	O(1millisecond)	O(100MBps)
Network	O(1microsecond)	O(1GBps)

Memory Hierarchy



- Moving data slows a program down
- The rule of locality applies
  - Temporal: keep data where it is as long as possible
  - Spatial: move adjacent data (avoid random access if possible)

Distributed Memory Systems Are Hierarchical

- A cluster with
  - N nodes, each with
    - M sockets (processors), each with
      - L cores
    - K accelerators (GPU or Xeon Phi)
- Many different programming models/types of parallel applications
  - No silver bullet for all problems

Potential Performance Bottlenecks on Distributed Memory Systems

- Intra-node
  - Memory bandwidth
  - Link between sockets (CPU to CPU)
  - Intra-CPU communication (core to core)
  - Link between CPU and accelerators
- Inter-node
  - Communication over network

Outline

- Distributed memory systems
- Message Passing Interface (MPI)
- Parallel applications

Message Passing

- Context: distributed memory systems
  - Each processor has its own memory space and cannot access the memory of other processors
  - Any data to be shared must be explicitly transferred from one to another as “messages”, hence the name “message passing”

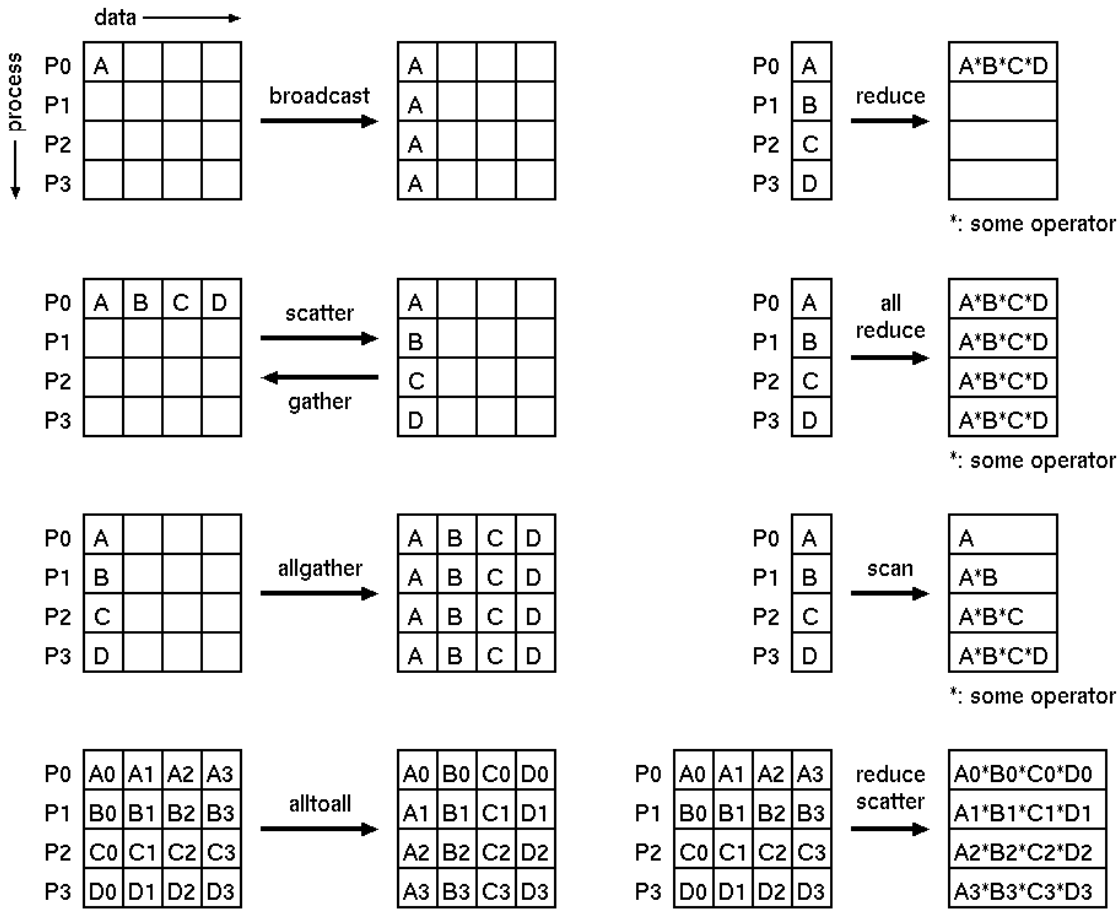
Message Passing Interface

- MPI defines a standard API for message passing
  - The standard includes
    - What functions are available
    - The syntax and semantics of those functions
    - What the expected outcome is when those functions are called
  - The standard does NOT include
    - Implementation details (e.g. how the data transfer occurs)
    - Runtime details (e.g. how many processes the code with run with etc.)
- MPI provides C/C++ and Fortran bindings

MPI Functions

- Point-to-point communication functions
  - Message transfer from one process to another
- Collective communication functions
  - Message transfer involving all processes in a communicator
- Environment management functions
  - Initialization and termination
  - Process group and topology

Example of MPI Functions



Source: Practical MPI Programming, IBM Redbook

Why MPI?

- Standardized
  - With efforts to keep it evolving (MPI 3.0 draft came out in 2010)
- Portability
  - MPI implementations are available on almost all platforms
- Scalability
  - In the sense that it is not limited by the number of processors that can access the same memory space
- Popularity
  - De Facto programming model for distributed memory systems

Implementations of MPI (1)

- There are many different implementations
  - Only a few are being actively developed
- MPICH and derivatives
  - MPICH
  - MVAPICH2
  - Intel MPI (not open source)
- OpenMPI (Not OpenMP!!!)
  - We will focus on OpenMPI today, which is the default implementation on Shelob
- They all comply to the MPI standards, but differ in implement details

Implementations of MPI (2)

- MPICH
  - Developed as an example implementation by a group of people who are involved in making the MPI standard
- MVAPICH2
  - Based on MPICH
  - Mainly targets HPC systems using Infiniband interconnect
- OpenMPI
  - Grew out of four other implementations

Outline

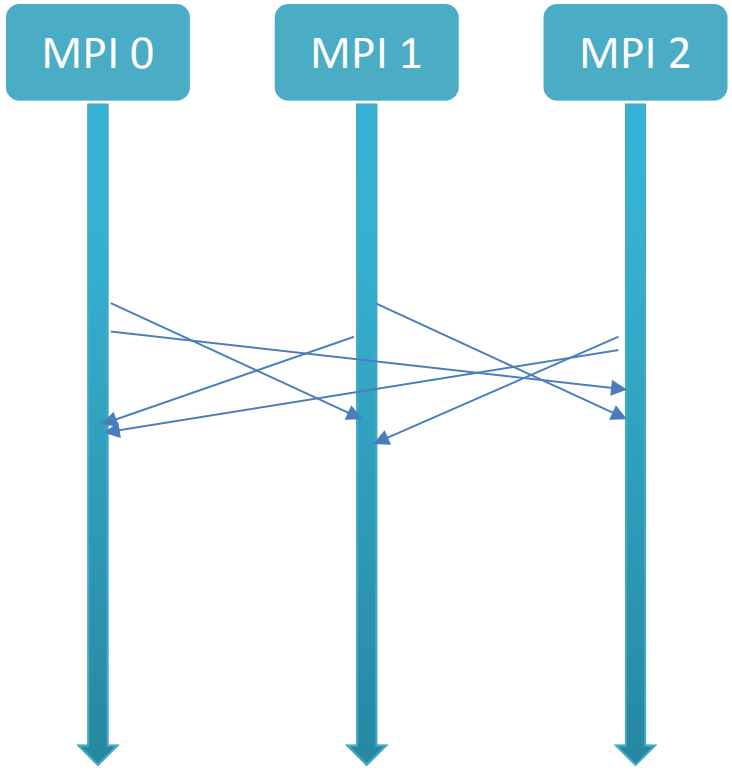
- Distributed memory systems
- Message Passing Interface (MPI)
- Parallel applications

Parallel Applications on Distributed Systems

- Pure MPI applications
  - CPU only
  - CPU + Xeon Phi
- Hybrid applications (MPI + X)
  - CPU only: MPI + OpenMP
  - CPU + GPU: MPI + GPU / MPI + OpenACC
  - CPU + Xeon Phi: MPI + OpenMP
- Others
  - Applications that do not use MPI to launch and manage processes on multiple hosts

Pure MPI Applications

- The launcher starts one copy of the program on each available processing core
- The runtime daemon monitors all MPI processes



MPI Runtime

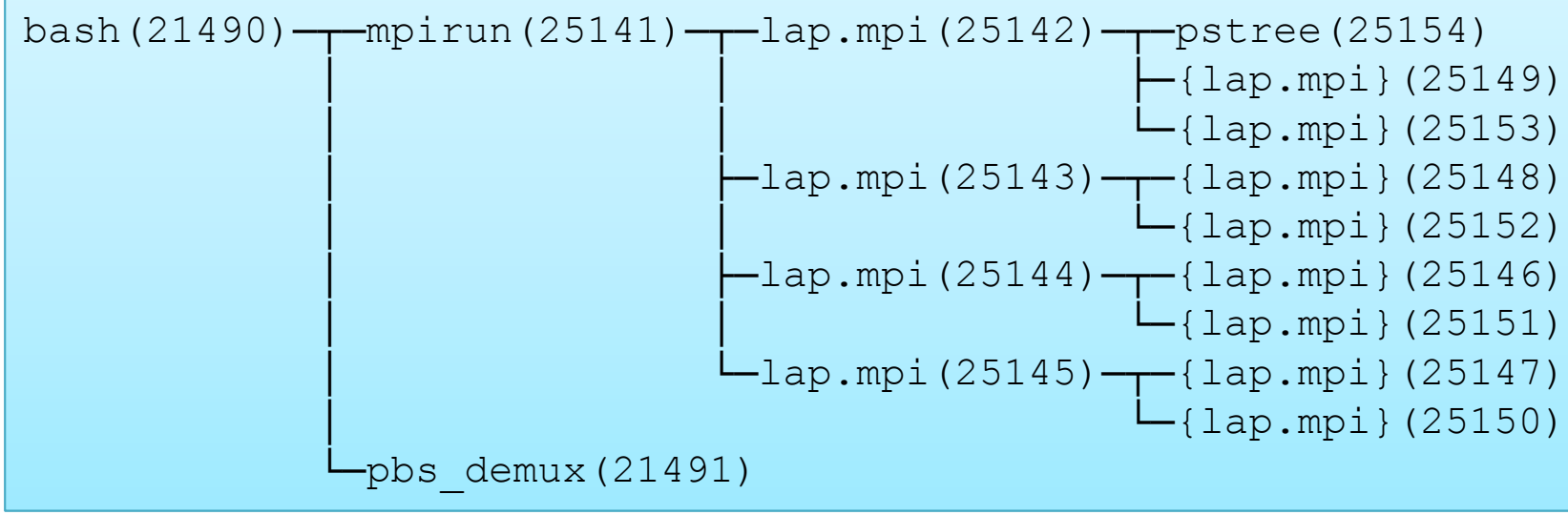
- Start multiple copies of the program
- Map MPI processes to physical cores
- Coordinate communication between MPI processes
- Send/receive signals to MPI processes (e.g. Ctrl-C is pressed by the user)
- Trap errors

Running MPI Programs

- Use `mpirun` or `mpiexec` to launch MPI programs on multiple hosts
- Need to provide a list of hosts so that the launcher knows where to start the program
  - With the `-host <host1>,<host2>…` flag
  - List all hosts in a host file and point to the file using the `-hostfile` flag

Example: MPI Laplace Solver

```
[lyan1@shelob001 par2015]$ mpicc -o lap.mpi laplace_solver_mpi_v3.c
[lyan1@shelob001 par2015]$ mpirun -np 4 -host
shelob001,shelob001,shelob001,shelob001 ./lap.mpi 1000 1000 2 100000
10000 0.00001
```



Host File

- MPI launcher spawns processes on remote hosts according to the host file
  - One host name on each line
  - If N processes are desired on a certain host, the host name needs to be repeated N times (N lines)
  - The host file can be modified to control the number of MPI processes started on each node

```
[lyan1@shelob001 par2015]$ cat hostlist
shelob001
shelob001
shelob002
shelob002
[lyan1@shelob001 par2015]$ mpirun -np 4 -hostfile ./hostlist ./lap.mpi
1000 1000 2 100000 10000 0.00001
bash(21490)─┬─mpirun(28736)─┬─lap.mpi(28737)─┬─pstree(28743)
            │               │                ├─{lap.mpi}(28740)
            │               │                └─{lap.mpi}(28742)
            │               └─lap.mpi(28738)─┬─{lap.mpi}(28739)
            │                                └─{lap.mpi}(28741)
            └─pbs_demux(21491)
```

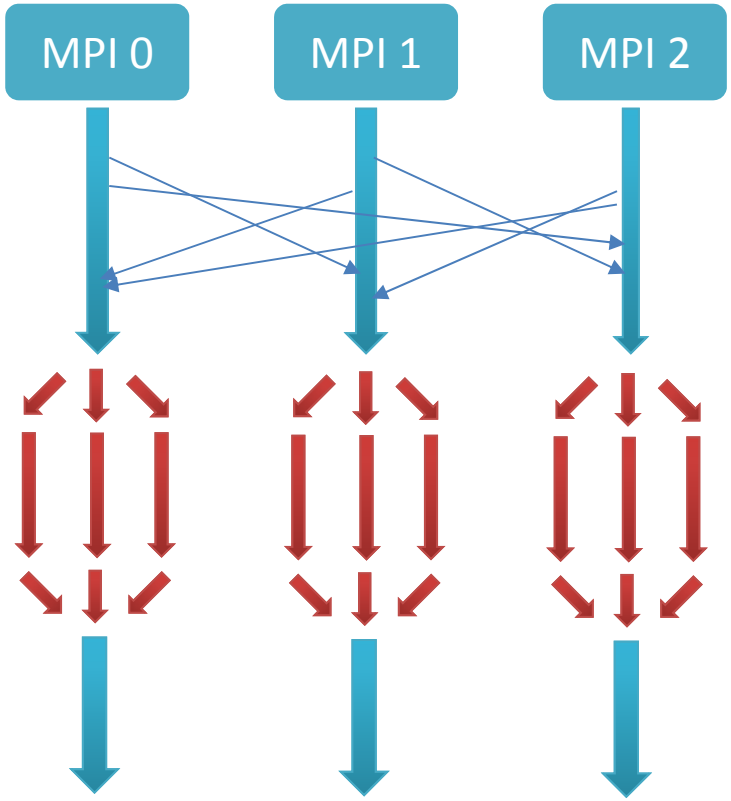
```
[lyan1@shelob002 ~]$ pstree -p -u lyan1
orted(21757)─┬─lap.mpi(21758)─┬─{lap.mpi}(21761)
             │                └─{lap.mpi}(21762)
             └─lap.mpi(21759)─┬─{lap.mpi}(21760)
                              └─{lap.mpi}(21763)
```

Why Do We Need Hybrid Applications?

- Adding OpenMP threading
  - May reduce memory footprint of MPI processes
    - MPI processes on a node replicate memory
  - May reduce MPI communication between processes
  - May reduce programming effort for certain sections of the code
  - May achieve better load balancing
- Allows programmers to take advantage of the accelerators

MPI + OpenMP

- MPI programs with OpenMP parallel regions
- MPI communication can occur within an OpenMP parallel region
  - Needs support from the MPI library



Running MPI+OpenMP Hybrid Applications

- Use `mpirun` or `mpiexec` to start the application
- Set `OMP_NUM_THREADS` to indicate how many threads per MPI process
- Indicate how many MPI processes per node
  - Modify the host file, or
  - Use the "`-perhost`" (MPICH based) or "`-npernode`" (OpenMPI) flag

Example: NPB BT-MZ Benchmark

- BT-MZ
  - Part of NPB, which is a parallel benchmark suite developed by NASA
  - Finite difference solver of the Navier-Stokes equation
  - There are a few difference sizes
    - We will run the C class (relatively small)

Running BT-MZ Hybrid Benchmark

```
[lyan1@shelob012 run]$ OMP_NUM_THREADS=4 mpirun -np 16 -npernode 4 -host
shelob012,shelob013,shelob014,shelob015 ../bin/bt-mz.C.16

……
 Use the default load factors with threads
 Total number of threads:      64   (  4.0 threads/process)
 Calculated speedup =      63.89

……
 BT-MZ Benchmark Completed.
 Class           =                        C
 Size            =            480x  320x 28
 Iterations      =                     200
 Time in seconds =                   14.51
 Total processes =                      16
```

Running BT-MZ Hybrid Benchmark

These two commands are equivalent with the one on last slide:

```
[lyan1@shelob012 run]$ OMP_NUM_THREADS=4 mpirun -np 16 -npernode 4 \
-hostfile $PBS_NODEFILE ../bin/bt-mz.C.16

[lyan1@shelob012 run]$ OMP_NUM_THREADS=4 mpirun -np 16 -hostfile
modified_hostfile ../bin/bt-mz.C.16
```

For MPICH based implementations, use:

```
[lyan1@shelob012 run]$ mpirun –env OMP_NUM_THREADS 4 -np 16 -perhost 4
-host shelob012,shelob013,shelob014,shelob015 ../bin/bt-mz.C.16
```

How Many Processes? How Many Threads?

- In our example we have 4 nodes with 16 cores each at our disposal, should we run
  - 4 MPI processes, each with 16 threads, or
  - 64 MPI processes, each with 1 thread, or
  - Anywhere in between?

How Many Processes? How Many Threads?

- In our example we have 4 nodes with 16 cores each at our disposal, should we run
  - 4 MPI processes, each with 16 threads, or
  - 64 MPI processes, each with 1 thread, or
  - Anywhere in between?
- It depends…
  - In many cases pure MPI applications are faster than their naïvely written/run hybrid counterparts

How Many Processes? How Many Threads?

```
[lyan1@shelob012 run]$ OMP_NUM_THREADS=1 mpirun -np 64 -npernode 16 -host
shelob012,shelob013,shelob014,shelob015 ../bin/bt-mz.C.64

[lyan1@shelob012 run]$ OMP_NUM_THREADS=2 mpirun -np 32 -npernode 8 -host
shelob012,shelob013,shelob014,shelob015 ../bin/bt-mz.C.32

[lyan1@shelob012 run]$ OMP_NUM_THREADS=4 mpirun -np 16 -npernode 4 -host
shelob012,shelob013,shelob014,shelob015 ../bin/bt-mz.C.16

[lyan1@shelob012 run]$ OMP_NUM_THREADS=8 mpirun -np 8 -npernode 2 -host
shelob012,shelob013,shelob014,shelob015 ../bin/bt-mz.C.8

[lyan1@shelob012 run]$ OMP_NUM_THREADS=16 mpirun -np 4 -npernode 1 -host
shelob012,shelob013,shelob014,shelob015 ../bin/bt-mz.C.4
```

How Many Processes? How Many Threads?

```
[lyan1@shelob012 run]$ OMP_NUM_THREADS=1 mpirun -np 64 -npernode 16 -host
shelob012,shelob013,shelob014,shelob015 ../bin/bt-mz.C.64

[lyan1@shelob012 run]$ OMP_NUM_THREADS=2 mpirun -np 32 -npernode 8 -host
shelob012,sh

[lyan1@shelc                                                    ode 4 -host
shelob012,sh

[lyan1@shelc                                                    le 2 -host
shelob012,sh

[lyan1@shelc                                                    ode 1 -host
shelob012,shelob013,shelob014,shelob015 ../bin/bt-mz.C.4
```

# of processes	# of threads per proc	Time (seconds)
4	16	18.61
8	8	16.11
16	4	14.51
32	2	13.05
64	1	13.28

Affinity Matters (1)

- MPI process/OpenMP thread placement will affect performance because sub-optimal placement may cause
  - Unnecessary inter-node communication
    - Communication between MPI processes could cause network contention
  - Unnecessary across-socket memory access
    - ccNUMA issue

Affinity Matters (2)

```
[lyan1@shelob012 run]$ OMP_NUM_THREADS=1 mpirun -np 64 -npernode 16 -host
shelob012,shelob013,shelob014,shelob015 –bynode –bind-to-socket
-report-bindings ../bin/bt-mz.C.64

[shelob012:04434] MCW rank 0 bound to socket 0[core 0-7]: [B B B B B B B
B][. . . . . . . .]
[shelob012:04434] MCW rank 4 bound to socket 0[core 0-7]: [B B B B B B B
B][. . . . . . . .]
[shelob012:04434] MCW rank 8 bound to socket 0[core 0-7]: [B B B B B B B
B][. . . . . . . .]
[shelob012:04434] MCW rank 12 bound to socket 0[core 0-7]: [B B B B B B B
B][. . . . . . . .]
[shelob014:25593] MCW rank 2 bound to socket 0[core 0-7]: [B B B B B B B
B][. . . . . . . .]
…

Time in seconds =                    57.81
```

4X slower!

Affinity Matters (3)

- Process placement
  - How to map MPI processes to processing elements
    - OpenMPI: `-by[slot|socket|node|…]` option
    - MPICH/MVAPICH2: `-map-by [socket|core|…]`
- Process binding
  - How to bind MPI processes to physical cores
    - OpenMPI: `-bind-to-[slot|socket|node]` option
    - MPICH/MVAPICH2: `-bind-to [socket|core|…]`
- Threading binding
  - compiler-specific runtime flags
    - Intel: `KMP_AFFINITY`
    - GNU: `GOMP_CPU_AFFINITY`

Affinity Matters (4)

- Pure MPI Applications can be affected by affinity as well

BT-MZ Benchmark C		
By	**Bind**	**Time (seconds)**
Core	No binding	13.28
Node	No binding	13.13
Node	Core	12.87
Core	Core	13.20
Socket	Socket	13.04

In this case, the code is not very demanding on memory and network, but there is still ~3% difference.

MPI + CUDA

- Use MPI launcher to start the program
  - Each MPI process can launch CUDA kernels
- Example: NAMD
  - A very popular molecular dynamics code
  - Based on Charm++, which is a parallel programming system which can function with or without MPI
  - On Shelob Charm++ is built on top of MVAPICH2

Running NAMD with MPI + CUDA

```
[lyan1@shelob001 stmv] mpirun -np 32 -hostfile $PBS_NODEFILE \
`which namd2` stmv.namd

Charm++> Running on MPI version: 2.1
…
Charm++> Running on 2 unique compute nodes (16-way SMP).
…
Pe 1 physical rank 1 will use CUDA device of pe 4
Pe 14 physical rank 14 will use CUDA device of pe 8
Pe 0 physical rank 0 will use CUDA device of pe 4
Pe 3 physical rank 3 will use CUDA device of pe 4
Pe 12 physical rank 12 will use CUDA device of pe 8
Pe 4 physical rank 4 binding to CUDA device 0 on
shelob001: 'Tesla K20Xm'  Mem: 5759MB  Rev: 3.5
…
<simulation starts>
```

MPI + Xeon Phi

- Use MPI launcher to start the program
  - Each MPI process can offload computation to Xeon Phi device(s)
- Example: LAMMPS
  - Stands for “Large-scale Atomic/Molecular Massively Parallel Simulator”
  - Another popular molecular dynamics code
  - On SuperMIC, it is built with Intel MPI

Running LAMMPS with Xeon Phi

```
[lyan1@smic043 TEST]$ mpiexec.hydra -env OMP_NUM_THREADS=1 -n 80 -perhost
20 -f $PBS_NODEFILE `which lmp_intel_phi` -in in.intel.rhodo -log none -v
b -1 -v s intel

LAMMPS (21 Jan 2015)
  using 1 OpenMP thread(s) per MPI task
Intel Package: Affinitizing MPI Tasks to 1 Cores Each
……
Using Intel Coprocessor with 4 threads per core, 24 threads per task
Precision: mixed
……
Setting up run ...
Memory usage per processor = 86.7087 Mbytes
……
<simulation starts>
```

Non-MPI Parallel Applications

- Example: NAMD
  - On SuperMIC Charm++ is built using IB verbs, the Infiniband communication library
  - No MPI is involved
  - It automatically detects and utilizes the Xeon Phi devices

```
[lyan1@smic031 stmv]for node in `cat $PBS_NODEFILE | uniq`; \
do echo host $node; done > hostfile

[lyan1@smic031 stmv]$ cat hostfile
host smic031
host smic037
host smic039
host smic040

[lyan1@smic031 stmv]charmrun ++p 80 ++nodelist ./hostfile \
++remote-shell ssh `which namd2` stmv.namd

Charmrun> IBVERBS version of charmrun
…
Pe 1 physical rank 0 binding to MIC device 0 on smic277:
240 procs 240 threads
…
Pe 8 physical rank 2 will use MIC device of pe 32
Pe 25 physical rank 6 will use MIC device of pe 1
Pe 0 physical rank 0 will use MIC device of pe 32
…
<simulation starts>
```

Wrap-up

- Distributed memory systems are dominant in the HPC world
  - MPI is de facto programming model on those systems
  - There are a few different implementations around
- That said, node-level architecture awareness is also important
  - If you are a developer: keep in mind all the bottlenecks that could hurt the performance of your code
  - If you are not a developer: at least try a few different combinations

Questions?