Minnesota Supercomputing Institute

UNIVERSITY OF MINNESOTA

Parallel Computation Overview

Andrew Gustafson dgustaf@umn.edu
David Porter dhp@umn.edu

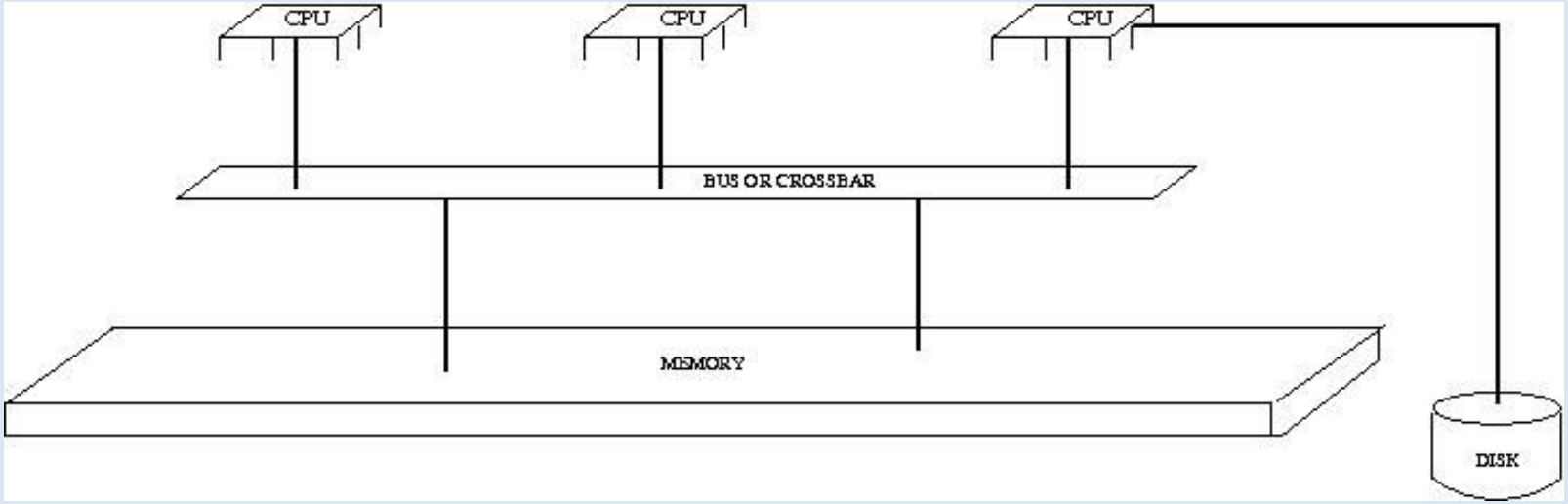
Parallel Computation

Parallel Computation means dividing up calculations into independent parts, and computing the independent parts simultaneously.

Most CPU processors are not much faster than what you can buy in a Desktop PC. To perform faster computations you must parallelize.

There are different forms of parallel computation. What form to use depends both on the type of problem, and on the type of compute system available.

Shared Memory System



Source: http://en.wikipedia.org/wiki/Shared_memory

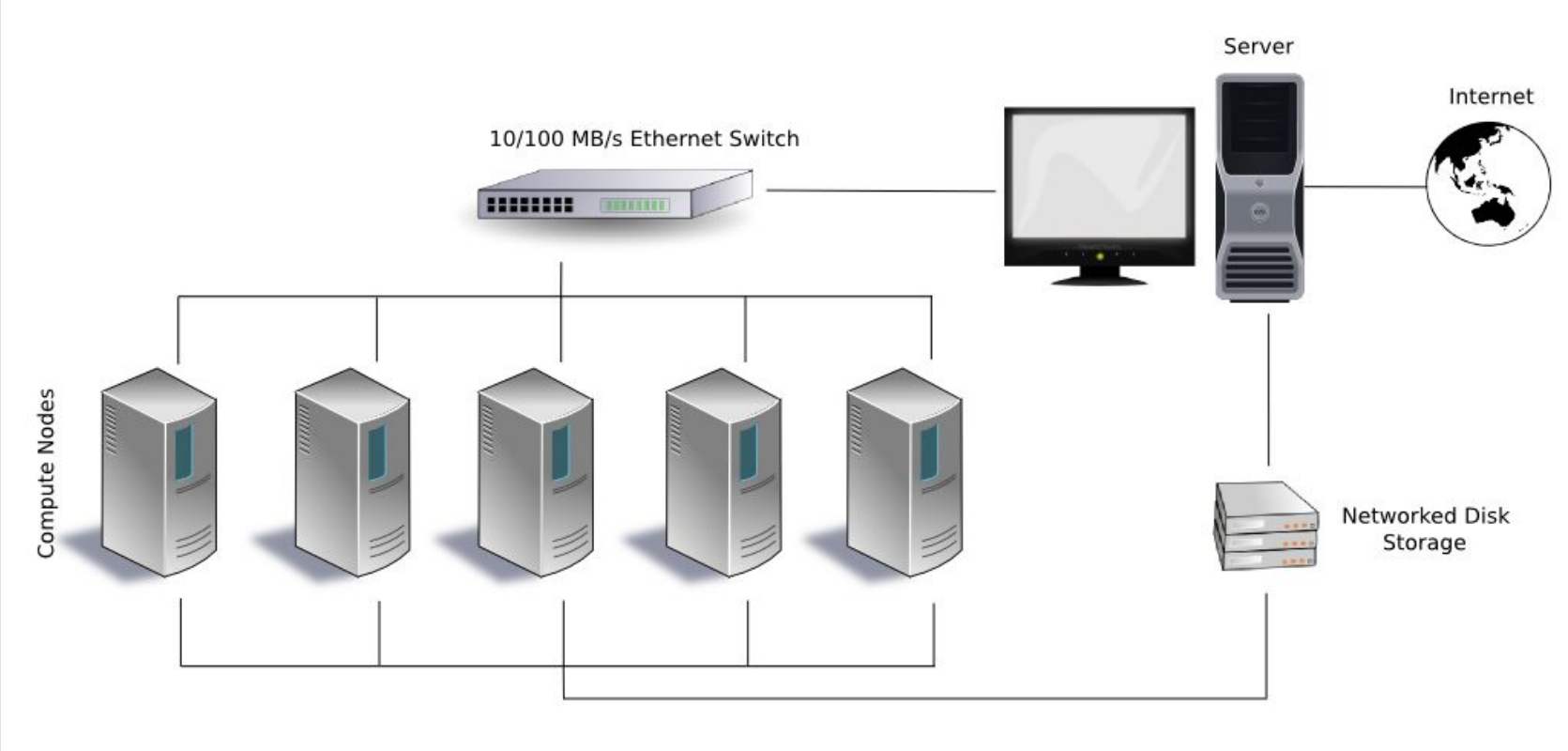
Strategies for a Shared Memory System

- Simple Parallelization: Collect Serial Calculations

- Thread parallelization
  - A single program uses multiple “threads” which can communicate using shared memory.
  - Coding for thread parallelization often means using OpenMP (which in turn is based on POSIX threads).

- Message Passing
  - Message passing frameworks such as MPI may be used, but are often not needed because thread parallel methods are sufficient.

Distributed Memory System: Cluster



Source: http://en.wikipedia.org/wiki/Cluster_%28computing%29

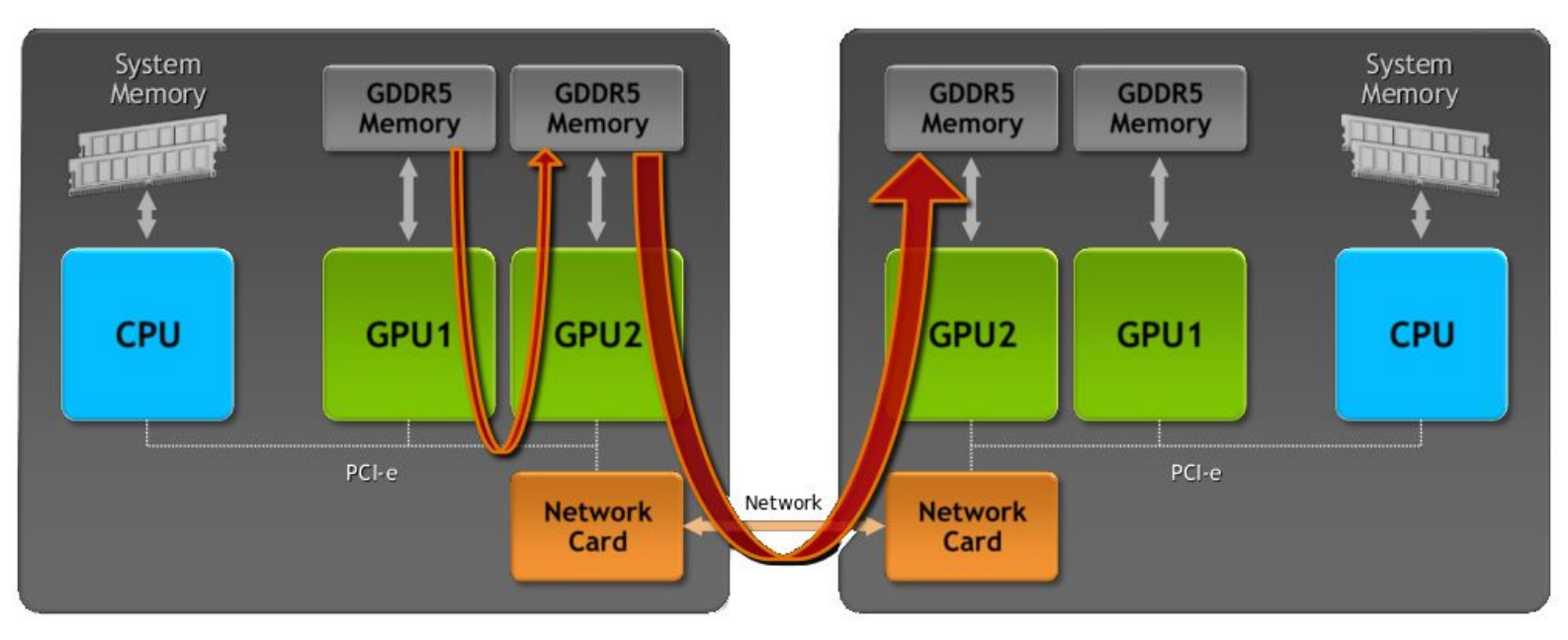
Strategies for a Distributed Memory System

- Simple Parallelization: Collect Serial Calculations
- Message Passing
  - Message passing frameworks such as MPI may be used to pass messages between nodes.
- Message Passing + Threads within node
  - Possible to combine message passing between nodes, with thread communication within a node.
  - Often involves using both MPI and OpenMP.

Clusters at MSI

- Mesabi
  - About 17,700 total cores, on Intel Haswell processors.
  - 24 cores and 62 GB per node in the large primary queues.
  - Special queues with large memory (up to 1TB), and GPUs.
  - Allows node sharing: good for both small and large jobs.
  - https://www.msi.umn.edu/content/mesabi
- Itasca
  - About 9,000 total cores, on Intel Nehalem processors.
  - 8 cores and 22 GB per node in the large primary queue.
  - Special queues with larger memory and 16 cores per node.
  - https://www.msi.umn.edu/content/itasca
- Interactive (Lab) Server
  - About 500 total cores, on older hardware.
  - For interactive, or small single node jobs.
  - 16 cores and 22 GB per node in the primary queue.
  - https://www.msi.umn.edu/content/interactive-hpc

Heterogeneous Systems



Source: http://electronicdesign.com/digital-ics/gpu-architecture-improves-embedded-application-support

Coding for Heterogenous Systems

- NVIDIA GPUS
  - Coding can be done in a sub-language called CUDA, which is supported by the PGI Fortran/C compilers (module pgi/15.7).
  - Can also use OpenACC (http://www.openacc.org), which is a C/C++/Fortran standard similar to OpenMP (openmp.org), implemented on PGI and recent GNU compilers. OpenACC 2.0 is a supported feature of GCC 7.2.0 and later.
  - MPI may need to be used if multiple CPU nodes are used.

Heterogeneous Systems at MSI

Mesabi (k40 queue)

- 40 nodes with 2 NVidia k40 GPUs per node
- k40 queue on Mesabi

Programming Difficulty

The general view of programming difficulty is that programming becomes more complicated in this order:

- Simple Parallelization: Collect Serial Calculations
- OpenMP (thread parallel)
- MPI (message passing)
- MPI + OpenMP (hybrid message passing + threads)
- Accelerators (GPUs using CUDA or OpenACC, Phis)

(Note that GPUs can more easily be used via the nVidia supplied libraries, e.g. cuFFT, cuBLAS, cuSPARSE, etc. See: http://docs.nvidia.com/cuda/index.html#axzz3o0lsq3xx).

The more difficult strategies can also yield larger speed increases, but it is important to examine the calculation type.

Job Scheduling

Parallel jobs are scheduled using a queueing system so that the hardware is fairly shared.

http://sitiosatumedida.com/mysite/images/blueview.jpg

Job Scheduling

Jobs are scheduled using the Portable Batch System (PBS) queueing system

To schedule a job first make a PBS job script:

```
#!/bin/bash -l
#PBS -l walltime=8:00:00,nodes=3:ppn=8,pmem=1000mb
#PBS -m abe
#PBS -M sample_email@umn.edu

cd ~/program_directory
module load intel
module load ompi/intel
mpirun -np 24 program_name < inputfile > outputfile
```

Job Submission

To submit a job script use the command:

```
qsub -q queuename scriptname
```

A list of queues available on different systems can be found here: https://www.msi.umn.edu/queues

Submit jobs to a queue which is appropriate for the resources needed.

Resources to consider when choosing a queue:

- Walltime
- Total cores and cores per node
- Memory
- Special hardware (GPUs, etc)

Interactive Jobs

Nodes may be requested for interactive use:

```
qsub -I -l walltime=1:00:00,nodes=2:ppn=8,mem=4gb
```

The terminal will hang until the job starts, and then it will return control. You can then use the nodes interactively for the job duration.

Simple Parallelization: Backgrounding

Most easily done with single node jobs.

```
#!/bin/bash -l
#PBS -l walltime=8:00:00,nodes=1:ppn=8,pmem=1000mb
#PBS -m abe
#PBS -M sample_email@umn.edu

cd ~/job_directory
module load example/1.0
./program1.exe < input1 > output1 &
./program2.exe < input2 > output2 &
./program3.exe < input3 > output3 &
./program4.exe < input4 > output4 &
./program5.exe < input5 > output5 &
./program6.exe < input6 > output6 &
./program7.exe < input7 > output7 &
./program8.exe < input8 > output8 &
wait
```

Simple Parallelization: Job Arrays

Works best on Mesabi.

Template Job Script, template.pbs:

```
#!/bin/bash -l
#PBS -l walltime=8:00:00,nodes=1:ppn=1,mem=2gb
#PBS -m abe
#PBS -M sample_email@umn.edu

cd ~/job_directory
module load example/1.0
./program.exe < input$PBS_ARRAYID > output$PBS_ARRAYID
```

Submit an array of 10 jobs:

```
qsub -t 1-10 template.pbs
```

Simple Parallelization: GNU Parallel

A way to spawn multiple threads to perform a shell task.

Example:

```
cat command_list.txt | parallel -j 24
```

This will take a list of command in command_list.txt, and then have GNU Parallel execute them simultaneously on one node using up to 24 concurrent threads.

Example:

```
find . -name '*.txt' | parallel -j 48 -sshloginfile $PBS_NODEFILE wc {}
```

This will find files ending in .txt, and then will use 48 threads to word count (wc) each of the files. Specifying sshloginfile makes it aware of all nodes being used.

Simple Parallelization: pdsh

A way to run multiple independent processes on multiple hosts.

Example:

```
pdsh -R ssh -w node0123,node0123,node0124 "./program.exe"
```

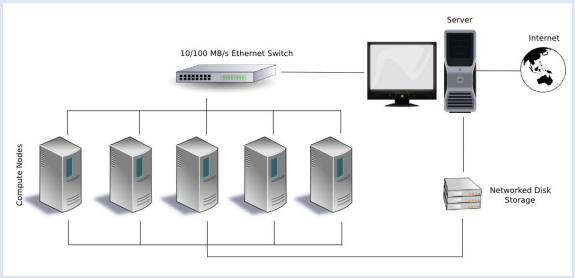
This would start two copies of program.exe on node0123, and one copy of program.exe on node0124, using ssh to connect.

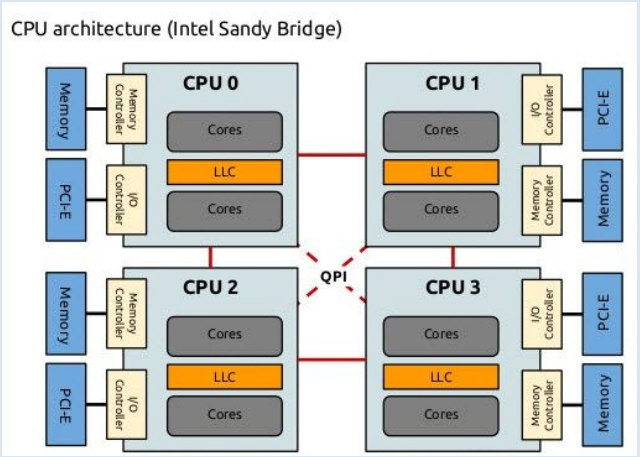
Example:

```
pdsh -R ssh -w^"$PBS_NODEFILE" "./program.exe"
```

This run one copy of program.exe on each of the cores assigned to the job.

MSI Systems and Hardware





All the MSI compute systems are “clusters”.
Memory is shared within single nodes.

Mesabi Nodes have:
24 Cores
64 GB of memory (2.67 GB per core).

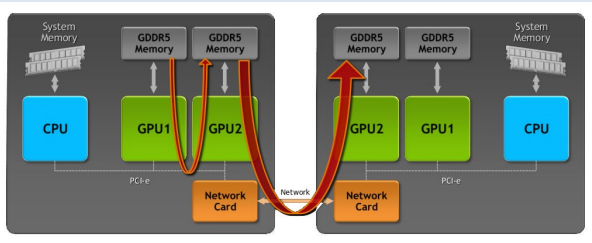
Mangi Nodes have:
128 Cores (!)
256 GB of memory (2 GB per core).

OpenMP can be used to write parallel programs for hardware with shared memory (single nodes).

MPI can be used to write parallel programs for hardware that can pass messages (multiple nodes within a cluster, or single nodes also).

Image Sources:
http://en.wikipedia.org/wiki/Cluster_%28computing%29
https://blog.sqlauthority.com/2015/12/12/sql-server-discussion-on-understanding-numa/

MSI Systems and Hardware



Mesabi and Mangi both have special nodes with GPU accelerators attached.

Mesabi has:
40 nodes, each with 2 NVidia K40 GPUs.
A K40 achieves ~ 1.4 TFlops double precision.

Mangi has:
12 nodes, each with 2 NVidia V100 GPUs.
4 nodes, each with 4 NVidia V100 GPUs.
1 node with 8 NVidia V100 GPU.
A V100 achieves ~ 7 TFlops double precision.

CUDA or OpenACC can be used to write programs that can run on GPUs. There are some other additional options.

Image Sources:
http://electronicdesign.com/digital-ics/gpu-architecture-improves-embedded-application-support

MPI

Message Passing Interface

MPI: Message Passing Interface

MPI is a specification

Library: subroutines, functions, constants & data types
Commands: starting apps across a cluster
Framework: control & communication

MPI routines

Called from source code (C,C++,Fortran)
Compiled & linked with MPI library
Work with framework for inter-process communication.

MPI versions available at MSI

Intel MPI	module load impi	
OpenMPI	module load ompi	
Platform MPI	module load pmpi	(Itasca only)

MPI: Motivation & Examples

Hierarchies

- Hardware
- Software

Parallel Applications & Message Passing

Source Code

- A short list of MPI routines is all you need to remember
- A simple example

MPI: Pros & Cons

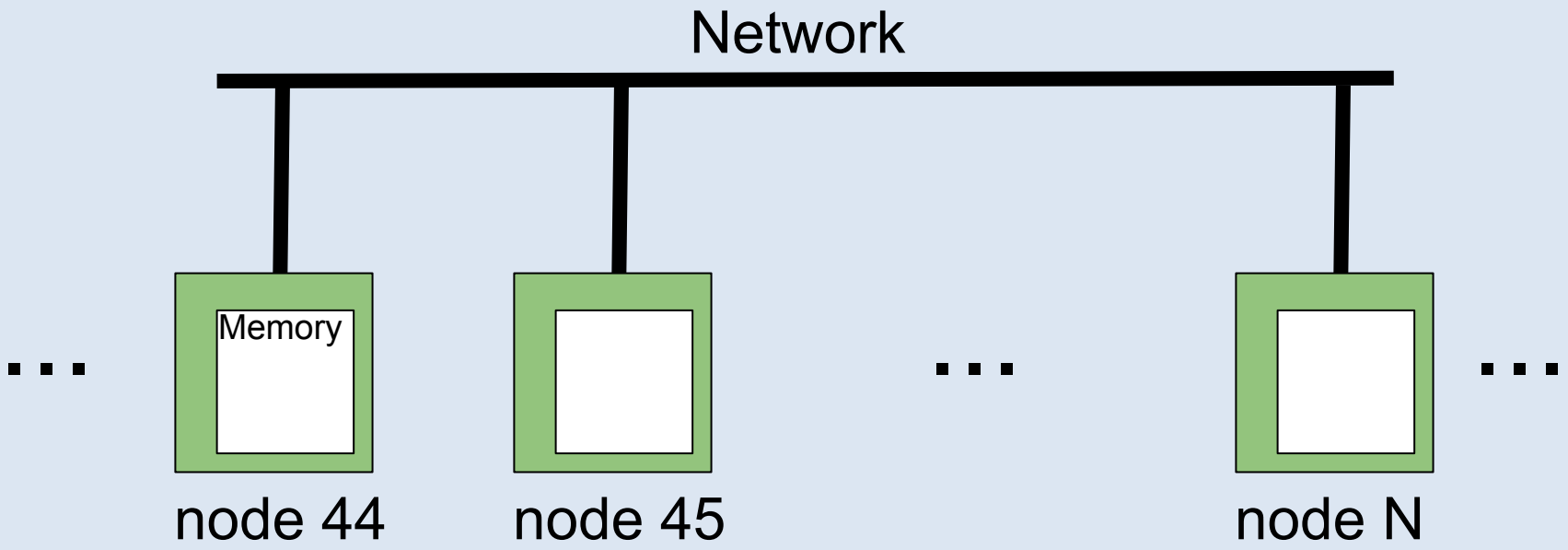
Mesabi: Hardware Hierarchy

core	2.5 GHz clock; up to 16 merged (* , +) per clock
Processor	12 cores; 30 MB Cache memory
Node	2 processors & 64+ GB shared memory
Level 1 Switch: Leaf	24 nodes (or fewer) 1x EDR to each node
Level 2 Switch: Island	8 Leafs 6x FDR to each leaf
Level 3 Switch: Cluster	4 Islands 12x FDR to each island

MPI: Software Hierarchy

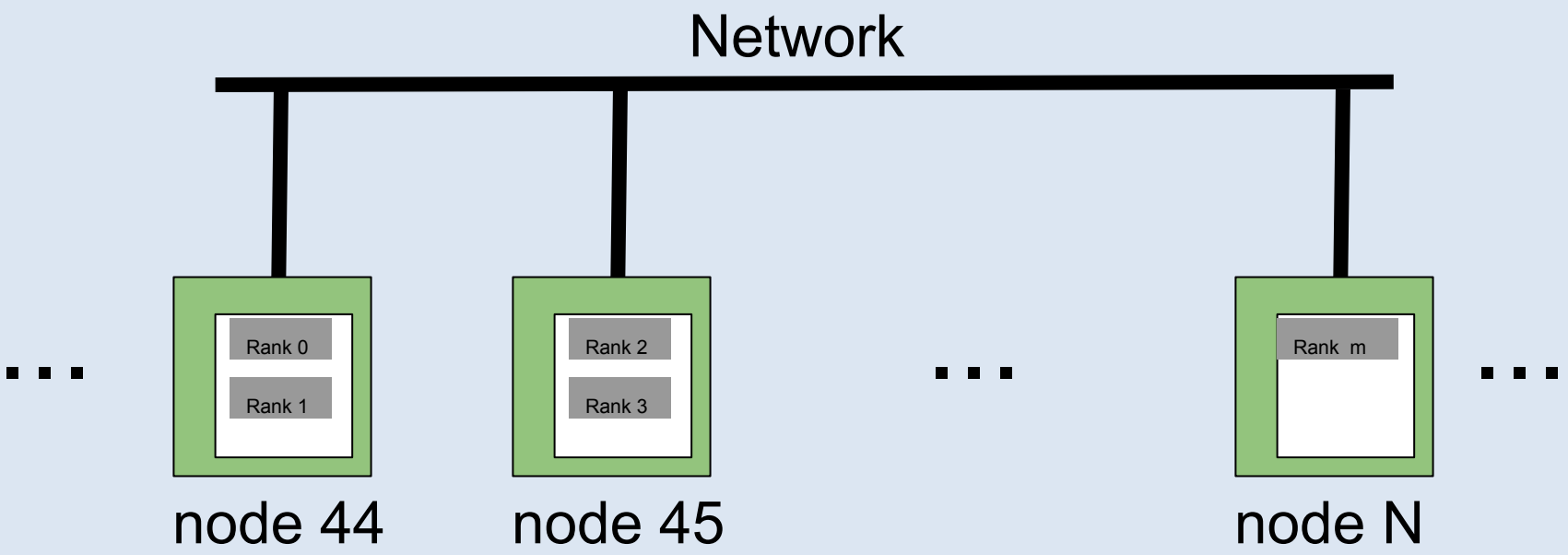
Thread	Scheduled work in time slices
Process	1 MPI rank: variables, arrays, IO streams, one or more threads
Application	1 or more Ranks (processes) MPI communicators
Workflow	1 or more applications, scripts, ...

Distributed Memory Systems



Each node can only directly access it's own memory
Nodes communicate through the network

MPI & Distributed Memory



Each rank is an instance of your code

- Each rank can only see its own variables
- All the ranks working together are an MPI application.

MPI: Hardware vs. Software

A process is NOT a processor

A processor has physical cores & cache memory.
A process has time slices and process address space.

1 MPI Rank = 1 process

An MPI rank *may* have many cores or share a core.
An MPI rank usually is confined to a shared memory node.
An MPI rank ALWAYS has a process address space.
MPI ranks are visible in the process table.
Different MPI ranks have different address spaces.

#PBS -l ppn=... Really means: Cores Per Node

MPI: How You Use It

Start "ranks" (copies of your program) on a list of nodes

mpirun -np 8 -hostfile $PBS_NODEFILE ***program***

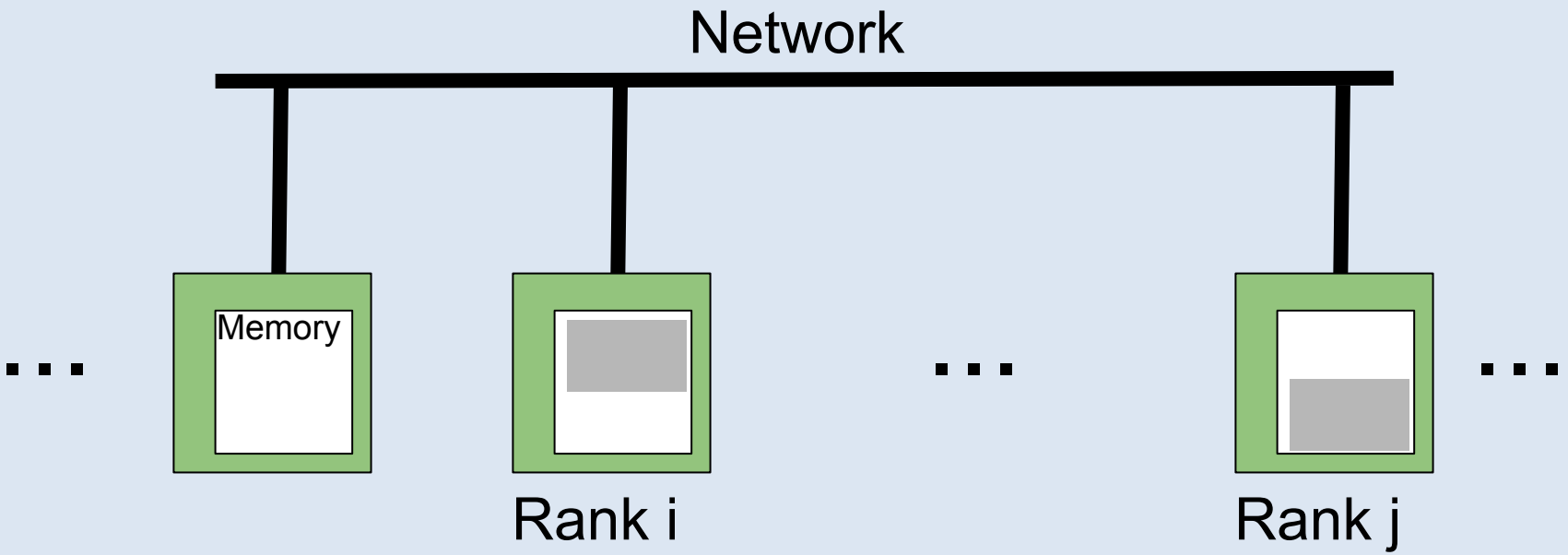
Coordinate the operation of all these ranks

MPI_Init	Initialize MPI within each rank
MPI_Comm_size	Get the total number of ranks
MPI_Comm_rank	Get the local rank
MPI_Finalize	Shut down MPI framework

Enable these ranks to communicate

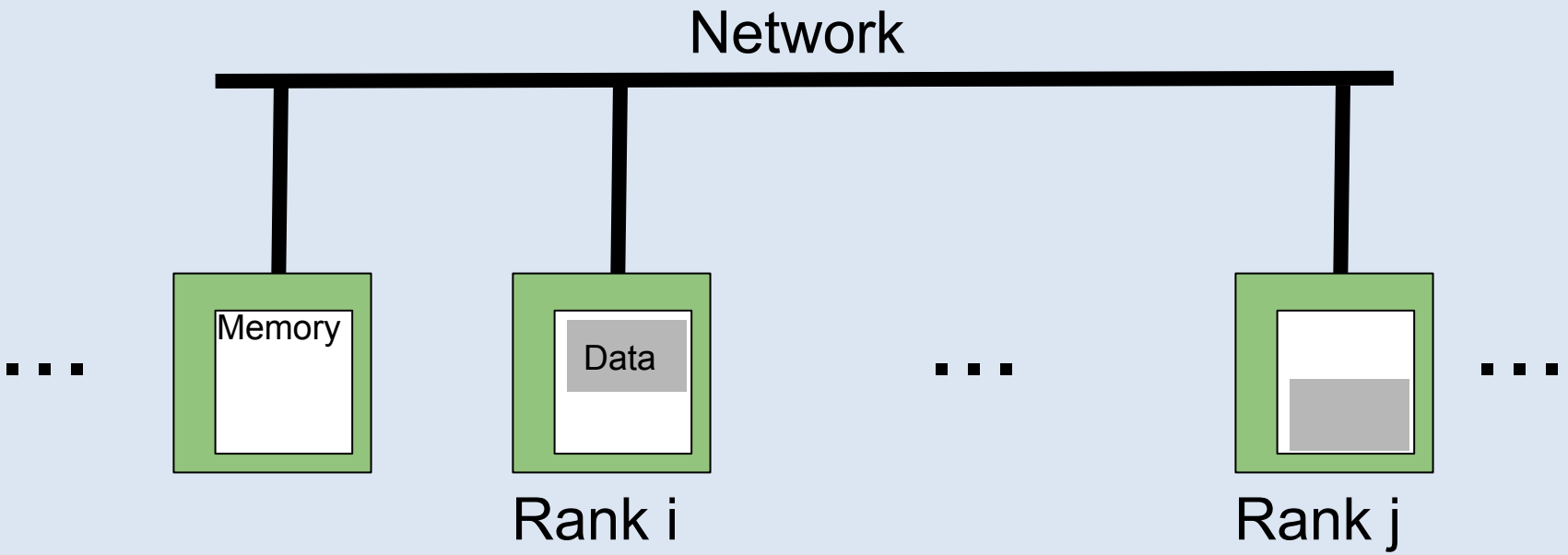
	MPI_Send(buffer, …)	Rank i sends a message
⇒	**MPI_Recv**(buffer, …)	Rank j receives the message

MPI Starts Copies of Your App



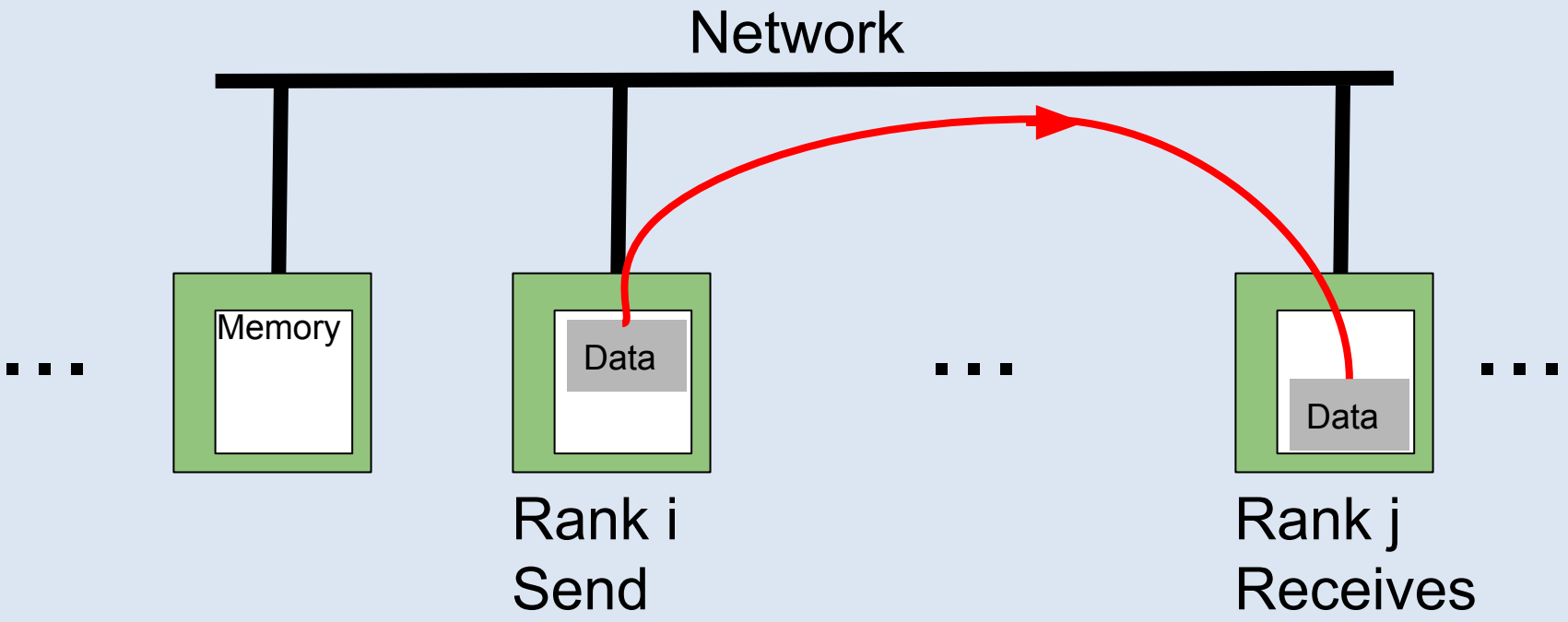
mpirun starts ranks (instances of your app) on nodes

Your Application Generates Data



Rank i generates some *Data*
This Data is needed on Rank j

Message Passing



Rank i sends a buffer of data to rank j
Ranks MAY be on different nodes
Ranks WILL be different processes ⇒ Martial Bits

MPI Syntax

```
MPI_Init(&argc,&argv)
MPI_Comm_size(comm, &nranks)
MI_Comm_rank(comm, &myrank)
MPI_Finalize()

MPI_Send(data, length, type, destination,tag, comm)
MPI_Recv(data, length, type, origin, tag,  comm, status)

MPI_Barrier(comm)
MPI_Bcast(data, length, type, origin,  comm)
MPI_Reduce(data_in, data_out, length, type, operation,
                destination, comm)
```

MPI Routine Names & Calling Convention

C / C++:

int ierror = MPI_Xxxx(....)

- Case sensitive
- All MPI calls are functions
- Program must include mpi.h
- Most parameters passed by reference

FORTRAN:

Call MPI_XXXX(...,ierror)

- Case insensitive
- All MPI calls are subroutines
- ierror is always the last parameter
- Program must include mpif.h

Structure of an MPI Program

Program start

Uncoordinated parallel execution

- Declarations & Prototypes
- Usually identical setup
- ***MPI Include Statements*** — ***#include "mpi.h"***

Initialize MPI Environment — ***MPI_Init***

Coordinated Parallel execution

- **Divide work among ranks** — ***MPI_Comm_size*** ***MPI_Comm_rank***
- **Communicate between ranks** — ***MPI_Send*** ***MPI_Recv***
- **Wait on other ranks as needed** — ***MPI_Barrier***
- Do Work
- **Collect results** — ***MPI_Gather*** ***MPI_Reduce***

Terminate MPI Environment — ***MPI_Finalize***

Uncoordinated parallel execution

Program End

Example with Send and Receive

```
#include "mpi.h"
#include <cstdlib>
#include <iostream>
int main(int argc,char** argv){
        using namespace std;
        int iError,   myrank;
        MPI_Status status;

        iError = MPI_Init(&argc,  &argv);
        iError = MPI_Comm_rank(MPI_COMM_WORLD,   &myrank);
        if(myrank == 0){
                int value_to_send = 5;
                iError = MPI_Send(&value_to_send,  1,  MPI_INT,  1,  123,  MPI_COMM_WORLD);
                cout << "Process " << myrank  << " sent value " << value_to_send << endl;
        }
        else if(myrank == 1){
                int value_received;
                iError = MPI_Recv(&value_received,  1,  MPI_INT,  0,  123,  MPI_COMM_WORLD,  &status);
                cout << "Process " << myrank << " received value " << value_received << endl;
        }
        iError = MPI_Finalize();
        return(0);
}
```

MPI: Arguments For & Against

Pros:

Performance - Strong Scaling & Cache Coherency
scalability - more cores AND more memory
flexibility - hardware topology & heterogeneity
portability - MPI is a standard not an implementation

Cons:

Need to restructure your code
However: possibly not by much
May need to restructure IO & data formats
However: may lead to much faster (parallel) IO
Can fall back to Rank 0 doing all IO

Current Architecture Trends

- Multi-socket nodes with rapidly increasing core counts.
- Memory per core decreasing.
- Memory bandwidth per core decreasing.
- Network bandwidth per core decreasing.

Need a hybrid programming model with 4 levels of parallelism

– Workflow: (GNU parallel) independent cases.
– Distributed Memory: (MPI) between nodes or sockets.
– Shared memory: (OpenMP) on the nodes/sockets.
– vectorization: (SIMD) for lower level loops.

Hybrid MPI/OpenMP Applications

https://www.nersc.gov/assets/pubs_presos/hybridMPIOpenMP20150323.pdf

Optimization Strategies:

1. Serial Optimization: Compiler options, profile code, etc.
2. Increase vectorization (SIMD) for lower level loops.
3. Implement shared memory threading (using OpenMP or pthreads) on a node/socket.
4. Implement MPI between nodes or sockets.
5. Implement workflow: driven by scripts or database

Minnesota Supercomputing Institute

Web: www.msi.umn.edu

Email: help@msi.umn.edu

Telephone: (612) 626-0802

UNIVERSITY OF MINNESOTA

Minnesota Supercomputing Institute