Parallelism at Scale: MPI

cs378h
Outline for Today

2PC review
Rust Wrapup
Scale
MPI

Acknowledgements:

Portions of the lectures slides were adopted from:
Argonne National Laboratory, MPI tutorials.
Lawrence Livermore National Laboratory, MPI tutorials
See online tutorial links in course webpage


Scale Out vs Scale Up
Scale Out vs Scale Up

Vertical Scaling $\uparrow$
Make boxes bigger

Horizontal Scaling $\rightarrow$
Make more boxes
Scale Out vs Scale Up

<table>
<thead>
<tr>
<th>Vertical Scaling</th>
<th>Horizontal Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher Capital Investment</td>
<td>On Demand Investment</td>
</tr>
<tr>
<td>Utilization concerns</td>
<td>Utilization can be optimized</td>
</tr>
<tr>
<td>Relatively Quicker and works with the current design</td>
<td>Relatively more time consuming and needs redesigning</td>
</tr>
<tr>
<td>Limiting Scale</td>
<td>Internet Scale</td>
</tr>
</tbody>
</table>
Parallel Systems Architects Wanted
Parallel Systems Architects Wanted

Hot Startup Idea: www.purchase-a-pooch.biz
Parallel Systems Architects Wanted
Parallel Systems Architects Wanted

1. User Browses Potential Pets
Parallel Systems Architects Wanted

1. User Browses Potential Pets
2. Clicks “Purchase Pooch”
Parallel Systems Architects Wanted

1. User Browses Potential Pets
2. Clicks “Purchase Pooch”
3. Web Server, CGI/EJB + Database complete request
Parallel Systems Architects Wanted

1. User Browses Potential Pets
2. Clicks “Purchase Pooch”
3. Web Server, CGI/EJB + Database complete request
4. Pooch delivered (not shown)
Parallel Systems Architects Wanted

1. User Browses Potential Pets
2. Clicks “Purchase Pooch”
3. Web Server, CGI/EJB + Database complete request
4. Pooch delivered (not shown)

How to handle lots and lots of dogs?
3 Tier architecture
3 Tier architecture

Web Servers (Presentation Tier) and App servers (Business Tier) scale horizontally
3 Tier architecture

Web Servers (Presentation Tier) and App servers (Business Tier) scale horizontally
Database Server → scales vertically

*Horizontal Scale → “Shared Nothing”*
3 Tier architecture

Web Servers (Presentation Tier) and App servers (Business Tier) scale horizontally.

Database Server scales vertically.

Horizontal Scale \(\rightarrow\) “Shared Nothing”

Why is this a good arrangement?
3 Tier architecture

Web Servers (Presentation Tier) and App servers (Business Tier) scale horizontally

Database Server \( \rightarrow \) scales vertically

*Horizontal Scale* \( \rightarrow \) “Shared Nothing”

Why is this a good arrangement?
3 Tier architecture

Web Servers (Presentation Tier) and App servers (Business Tier) scale **horizontally**

Database Server ➞ scales **vertically**

*Horizontal Scale ➞ “Shared Nothing”*

Why is this a good arrangement?
3 Tier architecture

Web Servers (Presentation Tier) and App servers (Business Tier) scale **horizontally**

Database Server → scales **vertically**

*Horizontal Scale ➔ “Shared Nothing”*

Why is this a good arrangement?
3 Tier architecture

Web Servers (Presentation Tier) and App servers (Business Tier) scale horizontally

Database Server → scales vertically

*Horizontal Scale* → “Shared Nothing”

Why is this a good arrangement?
3 Tier architecture

Web Servers (Presentation Tier) and App servers (Business Tier) scale horizontally
Database Server scales vertically

Horizontal Scale \(\rightarrow\) “Shared Nothing”

Why is this a good arrangement?
3 Tier architecture

Web Servers (Presentation Tier) and App Servers (Business Tier) scale horizontally.

Vertical scale gets you a long way, but there is always a bigger problem size.

Why is this a good arrangement?
Horizontal Scale: Goal
Design Space

Internet

Private data center

Latency

Throughput

Shared nothing

Shared something
Design Space

- Internet
- Private data center
- Latency
- Throughput
- Throughput
- Latency
- Shared nothing
- Shared something
- Transaction
Design Space

Throughput

Latency

Internet

Private data center

Shared nothing

Shared something

Transaction

Grid
Design Space

Throughput vs. Latency

- Internet
- Private data center
- Shared nothing
- Shared something

- Grid

Transaction
Search
Design Space

- Throughput
- Latency

- Private data center
- Internet

- Shared nothing
- Shared something

- Transaction
- Search
- Grid
- MapReduce
- Dryad
- Spark
Design Space

- Throughput
- Latency

- Internet
- Private data center

- Shared nothing
- Shared something

- Transaction
- Search
- HPC MPI
- MapReduce
- Spark
- Dryad
- Grid
Parallel Architectures and MPI
Parallel Architectures and MPI
Parallel Architectures and MPI

Distributed Memory
Multiprocessor
Messaging between nodes

processor
memory
interconnection network
processor
memory

...
Parallel Architectures and MPI

Distributed Memory Multiprocessor
  Messaging between nodes

Massively Parallel Processor (MPP)
  Many, many processors
Parallel Architectures and MPI

Distributed Memory Multiprocessor
Messaging between nodes

Massively Parallel Processor (MPP)
Many, many processors
Parallel Architectures and MPI

Distributed Memory Multiprocessor
- Messaging between nodes

Massively Parallel Processor (MPP)
- Many, many processors

Cluster of SMPs
- Shared memory in SMP node
- Messaging ↔ SMP nodes

- also regarded as MPP if processor # is large
Parallel Architectures and MPI

Distributed Memory Multiprocessor
Messaging between nodes

- Massively Parallel Processor (MPP)
  - Many, many processors

Cluster of SMPs
- Shared memory in SMP node
- Messaging $\leftrightarrow$ SMP nodes
  - also regarded as MPP if processor # is large

Multicore SMP+GPU Cluster
- Shared mem in SMP node
- Messaging between nodes
  - GPU accelerators attached
Parallel Architectures and MPI

Distributed Memory Multiprocessor
- Messaging between nodes
  - Massively Parallel Processor (MPP)
  - Many, many processors

Cluster of SMPs
- Shared memory in SMP node
- Messaging $\leftrightarrow$ SMP nodes
- Interconnection network

Massively Parallel Processor (MPP)
- Also regarded as MPP if processor # is large

Multicore SMP+GPU Cluster
- Shared mem in SMP node
- Messaging between nodes

What have we left out?
Parallel Architectures and MPI

Distributed Memory Multiprocessor
Messaging between nodes

Massively Parallel Processor (MPP)
Many, many processors

Cluster of SMPs
• Shared memory in SMP node
• Messaging ↔ SMP nodes

• also regarded as MPP if processor # is large

Multicore SMP+GPU Cluster
• Shared mem in SMP node
• Messaging between nodes

What have we left out?
• DSMs
• CMPs
• Non-GPU Accelerators
What requires extreme scale?
What requires extreme scale?

Simulations—why?
What requires extreme scale?

Simulations—why?
  Simulations are sometimes more cost effective than experiments
What requires extreme scale?

Simulations—why?
  Simulations are sometimes more cost effective than experiments

Why extreme scale?
  More compute cycles, more memory, etc, lead for faster and/or more accurate simulations
What requires extreme scale?

Simulations—why?
Simulations are sometimes more cost effective than experiments

Why extreme scale?
More compute cycles, more memory, etc, lead for faster and/or more accurate simulations

**Image credit:** Prabhat, LBNL
How big is “extreme” scale?

Measured in FLOPs

Floating point Operations Per second

1 GigaFLOP = 1 billion FLOPs
1 TeraFLOP = 1000 GigaFLOPs
1 PetaFLOP = 1000 TeraFLOPs

Most current super computers
1 ExaFLOP = 1000 PetaFLOPs
Arriving in 2018 (supposedly)
How big is “extreme” scale?

Measured in FLOPs

Floating point Operations Per Second

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Cores</th>
<th>Rmax [TFlop/s]</th>
<th>Rpeak [TFlop/s]</th>
<th>Power [kW]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
<tr>
<td>2</td>
<td>Tianhe-2 [MilkyWay-2] - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P, NUDT National Super Computer Center in Guangzhou China</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>3</td>
<td>Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100, Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland</td>
<td>361,760</td>
<td>19,590.0</td>
<td>25,326.3</td>
<td>2,272</td>
</tr>
<tr>
<td>4</td>
<td>Gyousou - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz, ExaScaler Japan Agency for Marine-Earth Science and Technology Japan</td>
<td>19,860,000</td>
<td>19,135.8</td>
<td>28,192.0</td>
<td>1,350</td>
</tr>
<tr>
<td>5</td>
<td>Titan - Cray XK7, Opteron 6774 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x, Cray Inc. DOE/SC/Oak Ridge National Laboratory United States</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>6</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom, IBM DOE/NNSA/LANL</td>
<td>1,572,864</td>
<td>17,173.2</td>
<td>20,132.7</td>
<td>7,890</td>
</tr>
</tbody>
</table>
How big is “extreme” scale?

Measured in FLOPs

Floating point Operations Per second

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Cores</th>
<th>Rmax [TFlop/s]</th>
<th>Rpeak [TFlop/s]</th>
<th>Power [kW]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China</td>
<td>914.6</td>
<td>125,435.9</td>
<td>15,371</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Tianhe-2 [MilkyWay-2] - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P, NUDT National Super Computer Center in Guangzhou China</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>3</td>
<td>Plz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100, Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland</td>
<td>361,760</td>
<td>19,590.0</td>
<td>25,326.3</td>
<td>2,272</td>
</tr>
<tr>
<td>4</td>
<td>Gyoukou - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700MHz, ExaScaler Japan Agency for Marine-Earth Science and Technology Japan</td>
<td>19,860,000</td>
<td>19,135.8</td>
<td>28,192.0</td>
<td>1,350</td>
</tr>
<tr>
<td>5</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x, Cray Inc. DOE/SC/Oak Ridge National Laboratory United States</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>6</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom, IBM DOE/NNSA/LANL</td>
<td>1,572,864</td>
<td>17,173.2</td>
<td>20,132.7</td>
<td>7,890</td>
</tr>
</tbody>
</table>

1 GigaFLOP = 1 billion FLOPs
1 TeraFLOP = 1000 GigaFLOPs
1 PetaFLOP = 1000 TeraFLOPs
1 ExaFLOP = 1000 PetaFLOPs

Most current super computers

Arriving in 2018 (supposedly)
Distributed Memory Multiprocessors
Distributed Memory Multiprocessors
Distributed Memory Multiprocessors

- Nodes: complete computer
  - Including I/O
- Nodes communicate via network
  - Standard networks (IP)
  - Specialized networks (RDMA, fiber)
Distributed Memory Multiprocessors

Each processor has a local memory
Physically separated address space

- Nodes: complete computer
  - Including I/O
- Nodes communicate via network
  - Standard networks (IP)
  - Specialized networks (RDMA, fiber)
Distributed Memory Multiprocessors

Each processor has a local memory
Physically separated address space

Processors communicate to access non-local data
Message communication
Message passing architecture
Processor interconnection network

- Nodes: complete computer
  - Including I/O
- Nodes communicate via network
  - Standard networks (IP)
  - Specialized networks (RDMA, fiber)
Distributed Memory Multiprocessors

Each processor has a local memory
Physically separated address space

Processors communicate to access non-local data
Message communication
Message passing architecture
Processor interconnection network

Parallel applications partitioned across
Processors: execution units
Memory: data partitioning

• Nodes: complete computer
  • Including I/O
• Nodes communicate via network
  • Standard networks (IP)
  • Specialized networks (RDMA, fiber)
Distributed Memory Multiprocessors

Each processor has a local memory
Physically separated address space

Processors communicate to access non-local data
Message communication
Message passing architecture
Processor interconnection network

Parallel applications partitioned across
Processors: execution units
Memory: data partitioning

Scalable architecture
Incremental cost to add hardware (cost of node)

• Nodes: complete computer
  • Including I/O

• Nodes communicate via network
  • Standard networks (IP)
  • Specialized networks (RDMA, fiber)
Performance: Latency and Bandwidth
Performance: Latency and Bandwidth

Bandwidth

- Need high bandwidth in communication
- Match limits in network, memory, and processor
- Network interface speed vs. network bisection bandwidth
Performance: Latency and Bandwidth

Bandwidth
- Need high bandwidth in communication
- Match limits in network, memory, and processor
- Network interface speed vs. network bisection bandwidth

Wait...bisection bandwidth?
Performance: Latency and Bandwidth

Bandwidth
- Need high bandwidth in communication
- Match limits in network, memory, and processor
- Network interface speed vs. network bisection bandwidth

Wait...bisection bandwidth?
if network is bisection, bisection bandwidth == bandwidth
Performance: Latency and Bandwidth

Bandwidth
- Need high bandwidth in communication
- Match limits in network, memory, and processor
- Network interface speed vs. network bisection bandwidth

Wait... bisection bandwidth?

if network is bisected, bisection bandwidth == bandwidth between the two partitions
Performance: Latency and Bandwidth

Bandwidth
- Need high bandwidth in communication
- Match limits in network, memory, and processor
- Network interface speed vs. network bisection bandwidth

Wait... bisection bandwidth?

if network is bisected, bisection bandwidth == bandwidth between the two partitions
Performance: Latency and Bandwidth

Bandwidth
- Need high bandwidth in communication
- Match limits in network, memory, and processor
- Network interface speed vs. network bisection bandwidth
Performance: Latency and Bandwidth

Bandwidth
- Need high bandwidth in communication
- Match limits in network, memory, and processor
- Network interface speed vs. network bisection bandwidth

Latency
- Performance affected: processor may have to wait
- Hard to overlap communication and computation
- Overhead to communicate: a problem in many machines
Performance: Latency and Bandwidth

Bandwidth
- Need high bandwidth in communication
- Match limits in network, memory, and processor
- Network interface speed vs. network bisection bandwidth

Latency
- Performance affected: processor may have to wait
- Hard to overlap communication and computation
- Overhead to communicate: a problem in many machines

Latency hiding
- Increases programming system burden
- E.g.: communication/computation overlap, prefetch
Performance: Latency and Bandwidth

Bandwidth

- Need high bandwidth in communication
- Match limits in network, memory, and processor
- Network interface speed vs. network bisection bandwidth

Latency

- Performance affected: processor may have to wait
- Hard to overlap communication and computation
- Overhead to communicate: a problem in many machines

Latency hiding

- Increases programming system burden
- E.g.: communication/computation overlap, prefetch

Is this different from metrics we’ve cared about so far?
Ostensible Advantages of Distributed Memory Architectures
Ostensible Advantages of Distributed Memory Architectures

Hardware simpler (especially versus NUMA), more scalable
Ostensible Advantages of Distributed Memory Architectures

Hardware simpler (especially versus NUMA), more scalable
Communication explicit, simpler to understand
Ostensible Advantages of Distributed Memory Architectures

- Hardware simpler (especially versus NUMA), more scalable
- Communication explicit, simpler to understand

Explicit communication →
    focus attention on costly aspect of parallel computation
Ostensible Advantages of Distributed Memory Architectures

Hardware simpler (especially versus NUMA), more scalable
Communication explicit, simpler to understand

Explicit communication →
  focus attention on costly aspect of parallel computation

Synchronization →
  naturally associated with sending messages
  reduces possibility for errors from incorrect synchronization
Ostensible Advantages of Distributed Memory Architectures

Hardware simpler (especially versus NUMA), more scalable
Communication explicit, simpler to understand

Explicit communication ➔
  focus attention on costly aspect of parallel computation

Synchronization ➔
  naturally associated with sending messages
  reduces possibility for errors from incorrect synchronization

Easier to use sender-initiated communication ➔
  some advantages in performance
Ostensible Advantages of Distributed Memory Architectures

- Hardware simpler (especially versus NUMA), more scalable
- Communication explicit, simpler to understand
  - Explicit communication $\Rightarrow$ focus attention on costly aspect of parallel computation
  - Synchronization $\Rightarrow$ naturally associated with sending messages, reduces possibility for errors from incorrect synchronization
  - Easier to use sender-initiated communication $\Rightarrow$ some advantages in performance

Can you think of any disadvantages?
Running on Supercomputers
Running on Supercomputers

• Programmer plans a **job**; job ==
  • parallel binary program
  • “input deck” (specifies input data)

• Submit job to a **queue**

• Scheduler allocates resources when
  • resources are available,
  • (or) the job is deemed “high priority”
Running on Supercomputers

• Programmer plans a \textit{job}; job ==
  • parallel binary program
  • “input deck” (specifies input data)

• Submit job to a \textit{queue}

• Scheduler allocates resources when
  • resources are available,
  • (or) the job is deemed “high priority”
Running on Supercomputers

- Programmer plans a job; job ==
  - parallel binary program
  - “input deck” (specifies input data)
- Submit job to a queue
- Scheduler allocates resources when
  - resources are available,
  - (or) the job is deemed “high priority”
Running on Supercomputers

- Programmer plans a **job**; job ==
  - parallel binary program
  - “input deck” (specifies input data)
- Submit job to a **queue**
- Scheduler allocates resources when
  - resources are available,
  - (or) the job is deemed “high priority”

Sometimes 1 job takes whole machine
These are called “hero runs”...
Sometimes many smaller jobs
Running on Supercomputers

- Programmer plans a job; job ==
  - parallel binary program
  - “input deck” (specifies input data)
- Submit job to a queue
- Scheduler allocates resources when
  - resources are available,
  - (or) the job is deemed “high priority”

Sometimes 1 job takes whole machine
These are called “hero runs”...
Sometimes many smaller jobs
Supercomputers used continuously
Processors: “scarce resource”
jobs are “plentiful”
Running on Supercomputers

- Programmer plans a **job**; job ==
  - parallel binary program
  - “input deck” (specifies input data)
- Submit job to a **queue**
- Scheduler allocates resources when
  - resources are available,
  - (or) the job is deemed “high priority”

Sometimes 1 job takes whole machine
  These are called “hero runs”...
Sometimes many smaller jobs
Supercomputers used continuously
  Processors: “scarce resource”
  jobs are “plentiful”
Running on Supercomputers

- Programmer plans a **job**; job ==
  - parallel binary program
  - “input deck” (specifies input data)
- Submit job to a **queue**
- Scheduler allocates resources when
  - resources are available,
  - (or) the job is deemed “high priority”

Sometimes 1 job takes whole machine
These are called “hero runs”...
Sometimes many smaller jobs
Supercomputers used continuously
Processors: “scarce resource”
jobs are “plentiful”

- Scheduler runs scripts that initialize the environment
  - Typically done with environment variables
Running on Supercomputers

• Programmer plans a **job**; job ==
  • parallel binary program
  • “input deck” (specifies input data)

• Submit job to a **queue**

• Scheduler allocates resources when
  • resources are available,
  • (or) the job is deemed “high priority”

Sometimes 1 job takes whole machine
These are called “hero runs”...
Sometimes many smaller jobs
Supercomputers used continuously
Processors: “scarce resource”
jobs are “plentiful”

• Scheduler runs scripts that initialize the environment
  • Typically done with environment variables
• At the end of initialization, it is possible to infer:
  • What the desired job configuration is (i.e., how many tasks per node)
  • What other nodes are involved
  • How your node’s tasks relates to the overall program
Running on Supercomputers

- Programmer plans a job; job ==
  - parallel binary program
  - “input deck” (specifies input data)
- Submit job to a queue
- Scheduler allocates resources when
  - resources are available,
  - (or) the job is deemed “high priority”

Sometimes 1 job takes whole machine
These are called “hero runs”...
Sometimes many smaller jobs
Supercomputers used continuously
Processors: “scarce resource”
jobs are “plentiful”

- Scheduler runs scripts that initialize the environment
  - Typically done with environment variables
- At the end of initialization, it is possible to infer:
  - What the desired job configuration is (i.e., how many tasks per node)
  - What other nodes are involved
  - How your node’s tasks relates to the overall program
- MPI library interprets this information, hides the details
The Message-Passing Model

Process: a program counter and address space
Processes: multiple threads sharing a single address space

MPI is for communication among processes
Not threads

Inter-process communication consists of
Synchronization
Data movement
The Message-Passing Model

Process: a program counter and address space
Processes: multiple threads sharing a single address space

MPI is for communication among *processes*
Not threads

Inter-process communication consists of
- Synchronization
- Data movement

How does this compare with CSP?
The Message-Passing Model

Process: a program counter and address space
Processes: multiple threads sharing a single address space

MPI is for communication among processes, not threads.

Inter-process communication consists of synchronization and data movement.

How does this compare with CSP?
The Message-Passing Model

Process: a program counter and address space

Process: multiple threads sharing a single address space

MPI is for communication among processes, not threads.

Interprocess communication consists of:
- Synchronization
- Data movement

How does this compare with CSP?

- MPI == Message-Passing Interface specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
The Message-Passing Model

Process: a program counter and address space
Process: multiple threads sharing a single address space

MPI is for communication among processes, not threads.
Inter-process communication consists of synchronization and data movement.

- MPI == Message-Passing Interface specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- Specified in C, C++, Fortran 77, F90
The Message-Passing Model

Process: a program counter and address space

Processes: multiple threads sharing a single address space

MPI is for communication among processes

Not threads

Inter-process communication consists of

Synchronization

Data movement

---

MPI == **Message-Passing Interface specification**

- Extended message-passing model
- Not a language or compiler specification
- Not a specific implementation or product

- Specified in C, C++, Fortran 77, F90
- Message Passing Interface (MPI) Forum
  - [http://www.mpi-forum.org/docs/docs.html](http://www.mpi-forum.org/docs/docs.html)
The Message-Passing Model

Process: a program counter and address space

Processes: multiple threads sharing a single address space

MPI is for communication among processes

Inter-process communication consists of:
- Synchronization
- Data movement

- MPI == Message-Passing Interface specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
  - Specified in C, C++, Fortran 77, F90
  - Message Passing Interface (MPI) Forum
    - http://www.mpi-forum.org/
    - http://www.mpi-forum.org/docs/docs.html

- Two flavors for communication
  - Cooperative operations
  - One-sided operations
Cooperative Operations
Cooperative Operations

Process 0

Send(data)

Process 1

Receive(data)

time
Cooperative Operations

Data is cooperatively exchanged in message-passing
Cooperative Operations

Data is cooperatively exchanged in message-passing
Explicitly sent by one process and received by another

Process 0

Send(data)

Receive(data)

Process 1

time
Cooperative Operations

Data is cooperatively exchanged in message-passing
Explicitly sent by one process and received by another
Advantage of local control of memory
  Change in the receiving process’s memory made with receiver’s explicit participation
Cooperative Operations

Data is cooperatively exchanged in message-passing
Explicitly sent by one process and received by another
Advantage of local control of memory
  Change in the receiving process’s memory made with receiver’s explicit participation
Communication and synchronization are combined

![Diagram showing message passing between Process 0 and Process 1. Process 0 sends (data) and Process 1 receives (data).]
Cooperative Operations

Data is cooperatively exchanged in message-passing
Explicitly sent by one process and received by another
Advantage of local control of memory
  Change in the receiving process’s memory made with receiver’s explicit participation
Communication and synchronization are combined

Process 0

Send(data)

Process 1

Receive(data)

time

Familiar argument?
One-Sided Operations
One-Sided Operations

Process 0
- Put(data)
- (memory)

Process 1
- (memory)
- Get(data)

Time
One-Sided Operations

One-sided operations between processes
Include remote memory reads and writes
One-Sided Operations

One-sided operations between processes
Include remote memory reads and writes
Only one process needs to explicitly participate
There is still agreement implicit in the SPMD program

Process 0
Put(data)
(memory)

Process 1
Get(data)
(memory)
time
One-Sided Operations

One-sided operations between processes
  Include remote memory reads and writes
Only one process needs to explicitly participate
  There is still agreement implicit in the SPMD program
Implication:
  Communication and synchronization are decoupled
One-Sided Operations

One-sided operations between processes
Include remote memory reads and writes

Only one process needs to explicitly participate
There is still agreement implicit in the SPMD program

Implication:
Communication and synchronization are decoupled

Are 1-sided operations better for performance?
A Simple MPI Program

```c
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```
MPI_Init
MPI_Init

Hardware resources allocated
  MPI-managed ones anyway...
Hardware resources allocated
  MPI-managed ones anyway...
Start processes on different nodes
  Where does their executable program come from?
MPI_Init

Hardware resources allocated
  MPI-managed ones anyway...
Start processes on different nodes
  Where does their executable program come from?
Give processes what they need to know
  Wait...what do they need to know?
MPI_Init

Hardware resources allocated
  MPI-managed ones anyway...

Start processes on different nodes
  Where does their executable program come from?

Give processes what they need to know
  Wait...what do they need to know?

Configure OS-level resources
Hardware resources allocated
  MPI-managed ones anyway...
Start processes on different nodes
  Where does their executable program come from?
Give processes what they need to know
  Wait...what do they need to know?
Configure OS-level resources
Configure tools that are running with MPI
Hardware resources allocated
    MPI-managed ones anyway…
Start processes on different nodes
    Where does their executable program come from?
Give processes what they need to know
    Wait...what do they need to know?
Configure OS-level resources
Configure tools that are running with MPI
...
MPI_Finalize
MPI_Finalize

Why do we need to finalize MPI?
Why do we need to finalize MPI?
What is necessary for a “graceful” MPI exit?
   Can bad things happen otherwise?
   Suppose one process exits...
MPI_Finalize

Why do we need to finalize MPI?
What is necessary for a “graceful” MPI exit?
  Can bad things happen otherwise?
  Suppose one process exits...
How do resources get de-allocated?
MPI_Finalize

Why do we need to finalize MPI?
What is necessary for a “graceful” MPI exit?
  Can bad things happen otherwise?
  Suppose one process exits...
How do resources get de-allocated?
How to shut down communication?
MPI_Finalize

Why do we need to finalize MPI?
What is necessary for a “graceful” MPI exit?
  Can bad things happen otherwise?
  Suppose one process exits...
How do resources get de-allocated?
How to shut down communication?
What type of exit protocol might be used?
MPI_Finalize

Why do we need to finalize MPI?
What is necessary for a “graceful” MPI exit?
   Can bad things happen otherwise?
   Suppose one process exits...
How do resources get de-allocated?
How to shut down communication?
What type of exit protocol might be used?

Executive Summary
• Undo all of init
• Be able to do it on success or failure exit
MPI_Finalize

Why do we need to finalize MPI?
What is necessary for a “graceful” MPI exit?
   Can bad things happen otherwise?
   Suppose one process exits...
How do resources get de-allocated?
How to shut down communication?
What type of exit protocol might be used?

• By default, an error causes all processes to abort
MPI_Finalize

Why do we need to finalize MPI?
What is necessary for a “graceful” MPI exit?
   Can bad things happen otherwise?
   Suppose one process exits...
How do resources get de-allocated?
How to shut down communication?
What type of exit protocol might be used?

Executive Summary
   • Undo all of init
   • Be able to do it on success or failure exit

   • By default, an error causes all processes to abort
   • The user can cause routines to return (with an error code)
      • In C++, exceptions are thrown (MPI-2)
MPI_Finalize

Why do we need to finalize MPI?
What is necessary for a “graceful” MPI exit?
   Can bad things happen otherwise?
   Suppose one process exits...
How do resources get de-allocated?
How to shut down communication?
What type of exit protocol might be used?

Executive Summary
• Undo all of init
• Be able to do it on success or failure exit

• By default, an error causes all processes to abort
• The user can cause routines to return (with an error code)
  • In C++, exceptions are thrown (MPI-2)
• A user can also write and install custom error handlers
MPI_Finalize

Why do we need to finalize MPI?
What is necessary for a “graceful” MPI exit?
   Can bad things happen otherwise?
   Suppose one process exits...
How do resources get de-allocated?
How to shut down communication?
What type of exit protocol might be used?

• By default, an error causes all processes to abort
• The user can cause routines to return (with an error code)
   • In C++, exceptions are thrown (MPI-2)
• A user can also write and install custom error handlers
• Libraries may handle errors differently from applications
Running MPI Programs
Running MPI Programs

MPI-1 does not specify how to run an MPI program
Running MPI Programs

MPI-1 does not specify how to run an MPI program

Starting an MPI program is dependent on implementation

Scripts, program arguments, and/or environment variables
Running MPI Programs

MPI-1 does not specify how to run an MPI program
Starting an MPI program is dependent on implementation
    Scripts, program arguments, and/or environment variables
% mpirun -np <procs> a.out
    For MPICH under Linux
Running MPI Programs

MPI-1 does not specify how to run an MPI program
Starting an MPI program is dependent on implementation
   Scripts, program arguments, and/or environment variables

% mpirun -np <procs> a.out
   For MPICH under Linux

mpiexec <args>
   Recommended part of MPI-2, as a recommendation
   mpiexec for MPICH (distribution from ANL)
   mpirun for SGI’s MPI
Finding Out About the Environment
Finding Out About the Environment

Two important questions that arise in message passing
  How many processes are being use in computation?
  Which one am I?
Finding Out About the Environment

Two important questions that arise in message passing:
How many processes are being used in computation?
Which one am I?

MPI provides functions to answer these questions:
- `MPI_Comm_size` reports the number of processes.
- `MPI_Comm_rank` reports the rank number between 0 and size-1 which identifies the calling process.
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
Hello World Revisited

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}

What does this program do?
```
Hello World Revisited

```c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

What does this program do?
Basic Concepts

Processes can be collected into *groups*

Each message is sent in a *context*
  - Must be received in the same context!

A group and context together form a *communicator*

A process is identified by its *rank*
  - With respect to the group associated with a communicator

There is a default communicator **MPI_COMM_WORLD**
  - Contains all initial processes
MPI Datatypes
MPI Datatypes

Message data (sent or received) is described by a triple address, count, datatype
MPI Datatypes

Message data (sent or received) is described by a triple address, count, datatype

An MPI datatype is recursively defined as:
   Predefined data type from the language
   A contiguous array of MPI datatypes
   A strided block of datatypes
   An indexed array of blocks of datatypes
   An arbitrary structure of datatypes
Message data (sent or received) is described by a triple address, count, datatype

An MPI *datatype* is recursively defined as:
- Predefined data type from the language
- 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
- Array of (int, float) pairs
- Row of a matrix stored columnwise
MPI Datatypes

Message data (sent or received) is described by a triple address, count, datatype

An MPI *datatype* is recursively defined as:

- Predefined data type from the language
- A contiguous array of MPI datatypes
- A strided block of datatypes
- An indexed array of blocks of datatypes
- An arbitrary structure of datatypes
MPI Datatypes

Message data (sent or received) is described by a triple address, count, datatype
An MPI datatype is recursively defined as:
  - Predefined data type from the language
  - A contiguous array of MPI datatypes
  - A strided block of datatypes
  - An indexed array of blocks of datatypes
  - An arbitrary structure of datatypes

- Enables heterogeneous communication
  - Support communication between processes on machines with different memory representations and lengths of elementary datatypes
  - MPI provides the representation translation if necessary
MPI Datatypes

Message data (sent or received) is described by a triple address, count, datatype

An MPI datatype is recursively defined as:

- Predefined data type from the language
- A contiguous array of MPI datatypes
- A strided block of datatypes
- An indexed array of blocks of datatypes
- An arbitrary structure of datatypes

- Enables heterogeneous communication
  - Support communication between processes on machines with different memory representations and lengths of elementary datatypes
  - MPI provides the representation translation if necessary
- Allows application-oriented layout of data in memory
  - Reduces memory-to-memory copies in implementation
  - Allows use of special hardware (scatter/gather)
MPI Tags
MPI Tags

Messages are sent with an accompanying user-defined integer *tag*

Assist the receiving process in identifying the message
MPI Tags

Messages are sent with an accompanying user-defined integer \textit{tag}.

Assist the receiving process in identifying the message.

Messages can be screened at receiving end by specifying specific tag \texttt{MPI\_ANY\_TAG} matches any tag in a receive.
MPI Tags

Messages are sent with an accompanying user-defined integer \textit{tag}
Assist the receiving process in identifying the message
Messages can be screened at receiving end by specifying specific tag
\textbf{MPI\_ANY\_TAG} matches any tag in a receive
Tags are sometimes called “message types”
MPI calls them “tags” to avoid confusion with datatypes
MPI Basic (Blocking) Send

\texttt{MPI\_SEND (start, count, datatype, dest, tag, comm)}

The message buffer is described by:
\begin{itemize}
  \item \texttt{start, count, datatype}
\end{itemize}

The target process is specified by \texttt{dest}
\begin{itemize}
  \item Rank of the target process in the communicator specified by \texttt{comm}
\end{itemize}

Process blocks until:
\begin{itemize}
  \item Data has been delivered to the system
  \item Buffer can then be reused
\end{itemize}

Message may not have been received by target process!
MPI with Only Six Functions
Many parallel programs can be written using:

- `MPI_INIT()`
- `MPI_FINALIZE()`
- `MPI_COMM_SIZE()`
- `MPI_COMM_RANK()`
- `MPI_SEND()`
- `MPI_RECV()`
MPI with Only Six Functions

Many parallel programs can be written using:

- `MPI_INIT()`
- `MPI_FINALIZE()`
- `MPI_COMM_SIZE()`
- `MPI_COMM_RANK()`
- `MPI_SEND()`
- `MPI_RECV()`

Why have any other APIs (e.g. broadcast, reduce, etc.)?
MPI with Only Six Functions

Many parallel programs can be written using:

- MPI_INIT()
- MPI_FINALIZE()
- MPI_COMM_SIZE()
- MPI_COMM_RANK()
- MPI_SEND()
- MPI_RECV()

Why have any other APIs (e.g. broadcast, reduce, etc.)?
Point-to-point (send/recv) isn’t always the most efficient...

Add more support for communication
int ctr=nLocalOriginal;
int offset=nLocalOriginal-nLocal;
for(i=0;i<worldSize;i++){
  if(i==rank){
    MPI_Bcast(s_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,i,MPI_COMM_WORLD);
  } else {
    MPI_Bcast(l_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,i,MPI_COMM_WORLD);
    for(k=0;k<l_particles[0];k++, ctr++){
      if(l_particles[MASS(k)]<0){
        offset++;
        _nparticles--;
      } else {
        s_particles[FX(ctr)]=l_particles[FX(k)];
        s_particles[FY(ctr)]=l_particles[FY(k)];
        s_particles[FZ(ctr)]=l_particles[FZ(k)];
        s_particles[MASS(ctr)]=l_particles[MASS(k)];
        indexes[ctr-offset]=ctr;
      }
    }
  }
}
```c
int ctr=nLocalOriginal;
int offset=nLocalOriginal-nLocal;
for(i=0;i<worldSize;i++)
    if(i==rank){
        MPI_Bcast(s_particles,N_POS ELEMS*nLocalMax+1,MPI_DOUBLE,i,MPI_COMM_WORLD);
    } else {
        MPI_Bcast(l_particles,N_POS ELEMS*nLocalMax+1,MPI_DOUBLE,i,MPI_COMM_WORLD);
        for(k=0;k<l_particles[0];k++, ctr++)
            if(l_particles[MASS(k)]<0){
                offset++;
                _nparticles--;
            } else {
                s_particles[FX(ctr)]=l_particles[FX(k)];
                s_particles[FY(ctr)]=l_particles[FY(k)];
                s_particles[FZ(ctr)]=l_particles[FZ(k)];
                s_particles[MASS(ctr)]=l_particles[MASS(k)];
                indexes[ctr-offset]=ctr;
            }
    }
```
int ctr=nLocalOriginal;
int offset=nLocalOriginal-nLocal;
for(i=0;i<worldSize;i++)
if(i==rank){
    MPI_Bcast(s_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,
   ) else {
        MPI_Bcast(l_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,
        for(k=0;k<_particles[0];k++, ctr++)
        if(_particles[MASS(k)]>0){
            offset++;
            _nparticles--;
        } else {
            s_particles[FX(ctr)]=l_particles[FX(k)];
            s_particles[FY(ctr)]=l_particles[FY(k)];
            s_particles[FZ(ctr)]=l_particles[FZ(k)];
            s_particles[MASS(ctr)]=l_particles[MASS(k)];
            indexes[ctr-offset]=ctr;
        }
    }
}
To use or not use MPI?
To use or not use MPI?

• USE
  • You need a portable parallel program
  • You are writing a parallel library
  • You have irregular or dynamic data relationships
  • You care about performance
To use or not use MPI?

• **USE**
  • You need a portable parallel program
  • You are writing a parallel library
  • You have irregular or dynamic data relationships
  • You care about performance

• **NOT USE**
  • You don’t need parallelism at all
  • You can use libraries (which may be written in MPI) or other tools
  • You can use multi-threading in a concurrent environment
    • You don’t need extreme scale