An Introduction to Parallel Programming with the **M**essage **P**assing **I**nterface

Douglas Gregor, Assistant Director

Open Systems Lab @ Indiana University

Boost.MPI library co-authored with Matthias Troyer

INDIANA UNIVERSITY
BLOOMINGTON

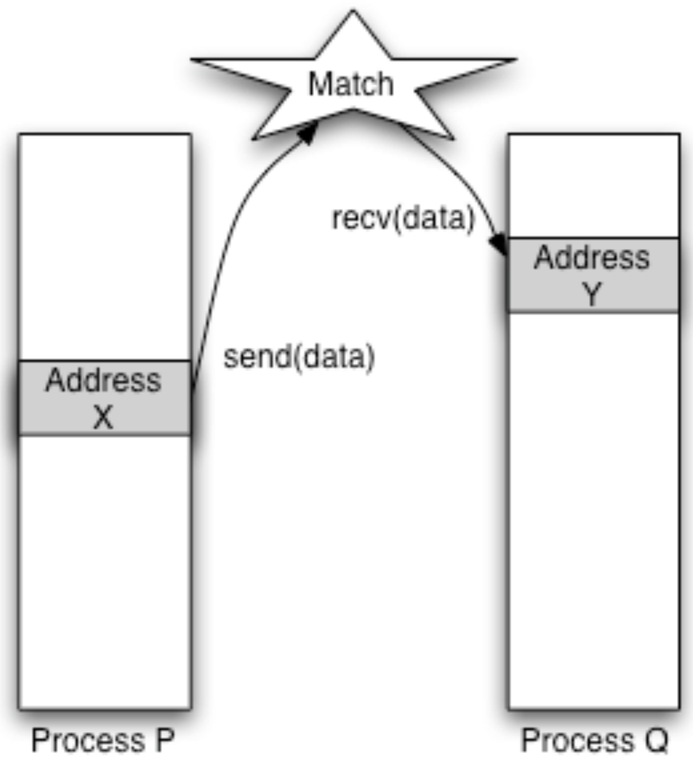
Open Systems Laboratory

Parallel Programming

- Many parallel architectures
  - Shared memory/distributed memory/NUMA
  - SMP, multi-core, massively threaded
  - Desktops, clusters, supercomputer
- Many paradigms for parallel programming
  - Task-parallel
  - Data-parallel
  - **Message passing**

Communicating: Cooperative Operations

- Message-passing is an approach that makes the exchange of data cooperative
- Data must both be explicitly sent and received
- Any change in the *receiver's* memory is made with the receiver's participation



Message Passing Interface

- **MPI** is a standard interface for writing programs using message passing
  - Library interface: C, C++, Fortran, *Boost C++*
  - Tools for executing MPI programs
- MPI is widely implemented:
  - Commercial: Sun, Microsoft, Cray, IBM, etc.
  - Open-source: Open MPI, MPICH, etc.
- Focus on high-performance computing

MPI is not Sockets

- Similarities to sockets:
  - Distributed-memory with message passing
  - Widely supported, architecture-independent
- Major differences:
  - Single messages vs. data streams
  - Collective communication
  - SPMD-focused
  - Performance-minded (to a fault)

Boost.MPI vs. Standard C MPI

- C MPI is very low-level
  - `void` pointers, explicit datatypes, PODs only
- Boost.MPI is a Boostified MPI
  - Clean, generic C++ interfaces
  - Complete support for user-defined data types
  - Plays well with the Standard Library and Boost
  - Interoperates with C MPI

When To Use MPI?

- You need a portable parallel program:
  - Data size exceeds what one system can store
  - Data processing requirements exceed what one system can handle
- You are writing a parallel library
- You care about performance

Tutorial Outline

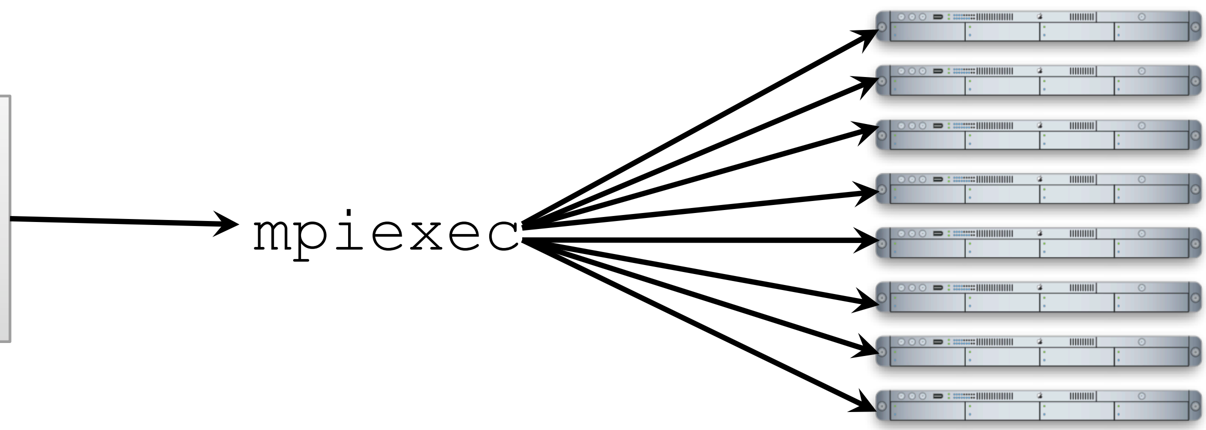
- MPI programs and communicators
  - Writing an MPI program
  - Launching an MPI program
- Collective communication I
- Point-to-point communication
- Transmitting your own data types
- Communicators and groups
- Collective communication II

Single Program, Multiple Data

- MPI programs are typically SPMD
  - Single MPI executable
  - Spawn that executable multiple times
  - All processes communicate via MPI

```cpp
#include <boost/mpi.hpp>
#include <iostream>
namespace mpi = boost::mpi;

int main(int argc, char* argv[]) {
  mpi::environment env(argc, argv);
  mpi::intracommunicator world;
  std::cout << "I am process " << world.rank()
    << " of " << world.size() << " running on "
    << mpi::environment::processor_name() << "\n";
  return 0;
}
```



Hello, World!

```
#include <boost/mpi.hpp>
#include <iostream>
namespace mpi = boost::mpi;

int main(int argc, char* argv[]) {
  mpi::environment env(argc, argv);
  mpi::intracommunicator world;
  std::cout << "I am process " << world.rank()
    << " of " << world.size() << " running on "
    << mpi::environment::processor_name()  << "\n";
  return 0;
}
```

Initializing the MPI environment

- Class `boost::mpi::environment`
  - Only one per program, allocated on the stack
  - Constructor initializes MPI communication
  - Destructor finalizes MPI

```
mpi::environment env(argc, argv);
```

MPI Communicators

- Communicators provide MPI's inter-process communication capabilities.
- `world` communicator connects all MPI processes.

```
mpi::intracommunicator world;
```

- Simple communicator queries:
  - `size()`: how many MPI processes?
  - `rank()`: which process number am I?

Building an MPI Program

```cpp
#include <boost/mpi.hpp>
#include <iostream>
namespace mpi = boost::mpi;

int main(int argc, char* argv[]) {
  mpi::environment env(argc, argv);
  mpi::intracommunicator world;
  std::cout << "I am process " << world.rank()
    << " of " << world.size() << " running on "
    << mpi::environment::processor_name() << "\n";
  return 0;
}
```

- Compile with `mpic++`:

```
mpic++ -I$BOOST_ROOT hello.cpp -lboost_mpi-mt_1_35
```

Executing an MPI Program

- `mpiexec` launches MPI jobs

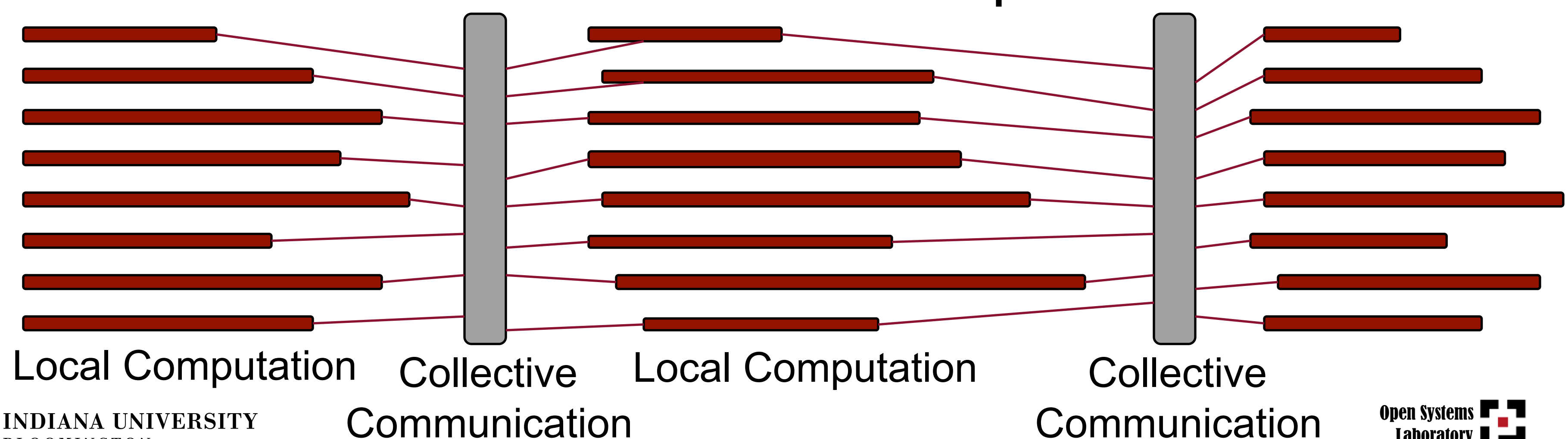
```
mpiexec -np 8 ./hello
```

- Output:

```
I am process 2 of 8 running on odin017.cs.indiana.edu
I am process 0 of 8 running on odin015.cs.indiana.edu
I am process 1 of 8 running on odin016.cs.indiana.edu
I am process 3 of 8 running on odin018.cs.indiana.edu
I am process 4 of 8 running on odin019.cs.indiana.edu
I am process 7 of 8 running on odin022.cs.indiana.edu
I am process 5 of 8 running on odin020.cs.indiana.edu
I am process 6 of 8 running on odin021.cs.indiana.edu
```

Collective Communication

- Collective communication involves all MPI processes
  - Processes each supply some data
  - Processes coordinate to compute the result
  - Results are sent back to the processes



Collectives I - Barrier

- Collective communication involves all processes in a communicator.
- `barrier()` waits for all processes to reach that barrier before exiting

```
// Everyone works on step #1
world.barrier();
// Everyone works on step #2
world.barrier();
// Everyone works on step #3
```

Collectives I – Broadcast

```
template<typename T>
void broadcast(const communicator& comm, T& value,
               int root);
```

- Broadcast values from one process to every process.
  - The root provides the value
  - Every process receives the value

Broadcast Example

- A simple command interpreter run in parallel:

```
while (true) {
  string command;
  if (world.rank() == 0)
    command = ReadUserInput();
  broadcast(world, command, 0);
  if (command == "quit")
    break;
  ExecuteCommand(command);
}
```

Collectives I – Gather

```
template<typename T>
void gather(const communicator& comm,
            const T& in_value,
            std::vector<T>& out_values, int root)
```

- Gathers data from all of the processes into a vector at a specified “root”.
  - Everyone must provide the same communicator, root, and their own value
  - Only the root needs an output vector

Gather Example

- Let's print all of the processor names in sorted order:

```
vector<string> hostnames;
gather(world, mpi::environment::processor_name(),
       hostnames, 0);
if (world.rank() == 0) {
  sort(hostnames.begin(), hostnames.end());
  copy(hostnames.begin(), hostnames.end(),
    ostream_iterator<string>(cout, "\n"));
}
```

Gather Results

- Output:

```
odin015.cs.indiana.edu
odin016.cs.indiana.edu
odin017.cs.indiana.edu
odin018.cs.indiana.edu
odin019.cs.indiana.edu
odin020.cs.indiana.edu
odin021.cs.indiana.edu
odin022.cs.indiana.edu
```

Collectives I – Scatter

```
template<typename T>
void scatter(const communicator& comm,
             const std::vector<T>& in_values,
             T& out_value, int root);
```

- Scatters values from the root to every process (the reverse of `gather`)

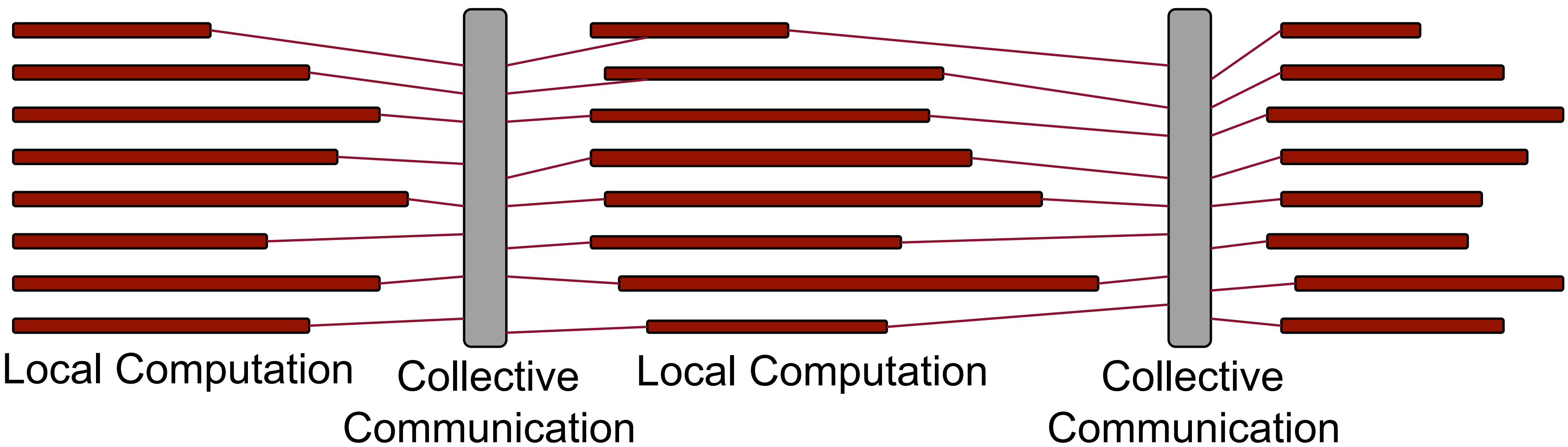
Scatter Example

- Master process distributes tasks to each of the processes:

```
class Task { ... };

std::vector<Task> tasks;
if (world.rank() == 0)
  GenerateTasks(tasks); //tasks.size()==world.size()
Task myTask;
scatter(world,tasks, myTask, 0);
myTask.Execute();
```

Process Skew



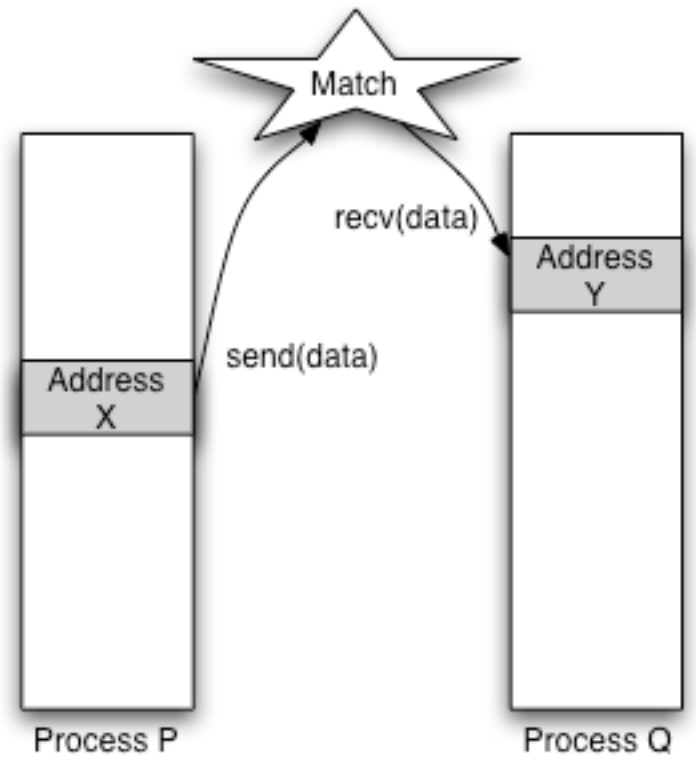
- Uneven computation time leads to wasted cycles, poor scalability.

Why Use Collective Comm?

- For correctness:
  - Easy to globally exchange data correctly.
- For performance:
  - The right collective is often better implemented than the equivalent point-to-point operation.
  - Beware process skew!

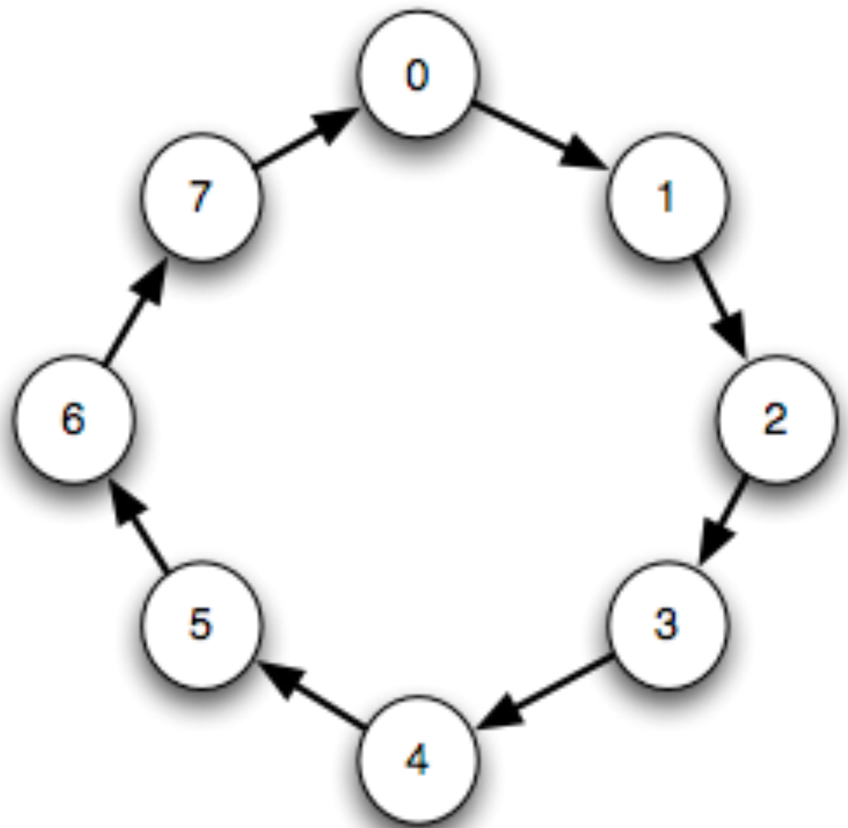
Point-to-Point Communication

- Each process can send a single message to another process
- Message contains:
  - A *communicator*
  - A *destination*
  - A *tag* that identifies the message
  - Data (of any type)

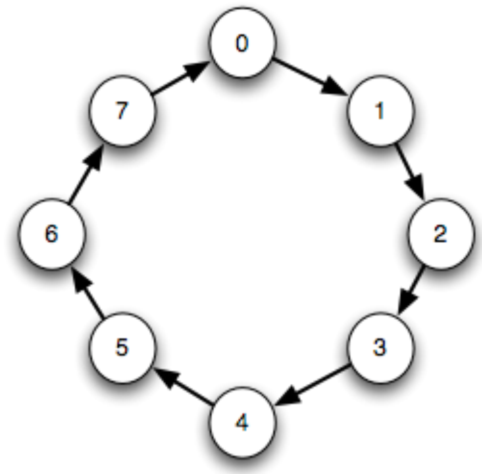


Example: Ring Communication

- Process 0 initiates a message
- Process 1 receives that message, forwards to process 2.
- Continue propagation until the message reaches 0.

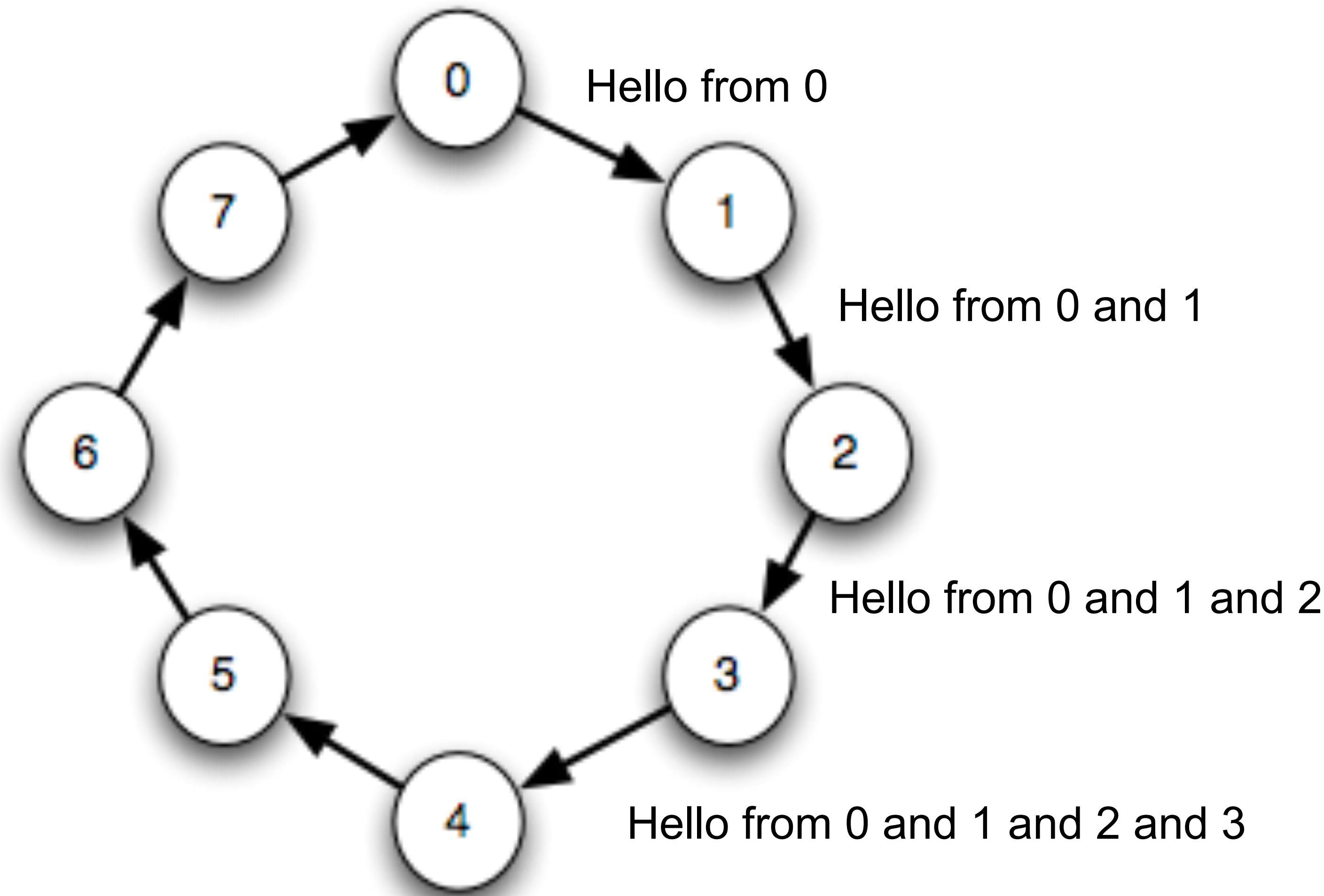


Example: Ring Communication




```
std::string msg;
if (world.rank() == 0) {
  msg = “Hello from 0”;
  world.send(1, 0, msg);
  world.recv(world.size()-1, 0, msg);
  std::cout << msg << ‘\n’;
} else {
  world.recv(world.rank()-1, 0, msg);
  std::cout << msg << ‘\n’;
  msg += “ and “
      + lexical_cast<std::string>(world.rank());
  world.send((world.rank()+1) % world.size(), 0, msg);
}
```

Ring Communication Output



Wildcard Receives

- Receive operation matches a message based on source process and tag
- Wildcards match messages from any source process (`mpi::any_source`) with any tag (`mpi::any_tag`).
- Example:

```
std::string msg;
world.recv(boost::mpi::any_source, 0, msg);
```

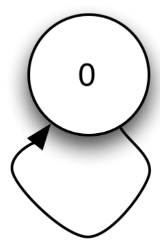
Probing Unreceived Messages

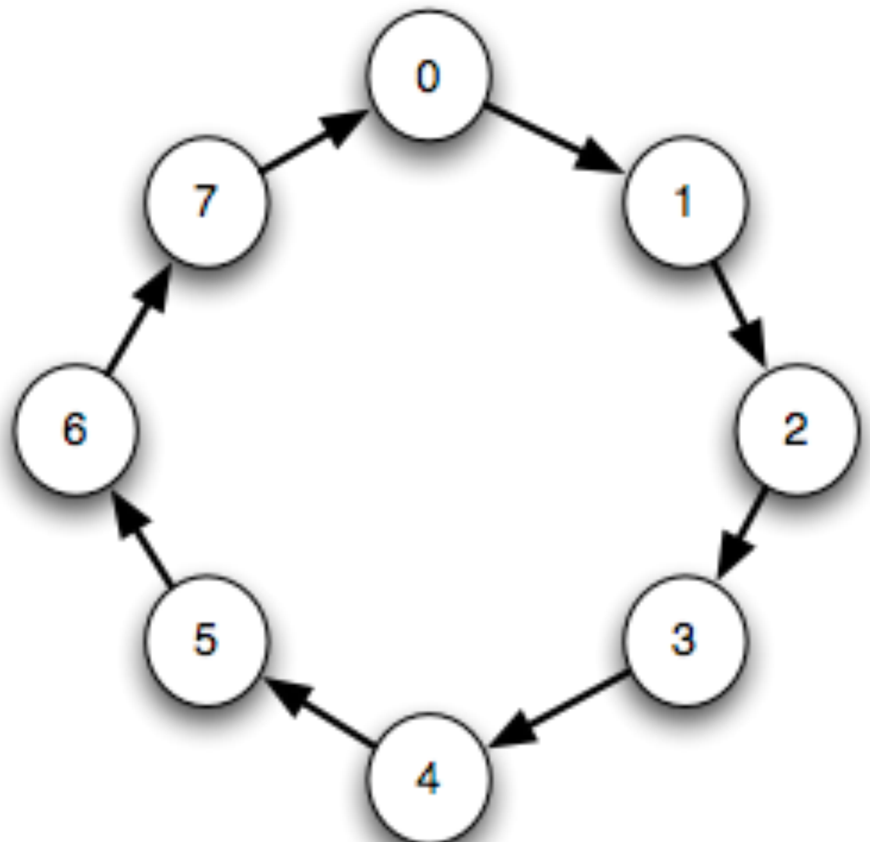
```
status probe(int source = any_source,
             int tag = any_tag) const;
optional<status> iprobe(int source = any_source,
                        int tag = any_tag) const;
```

- Look for a message that can be received.
  - Match message on source/tag
  - `status` structure has:
    - `source()` and `tag()` of message
    - `count<T>()` with the number of `T`'s in the message

Revisiting the Ring

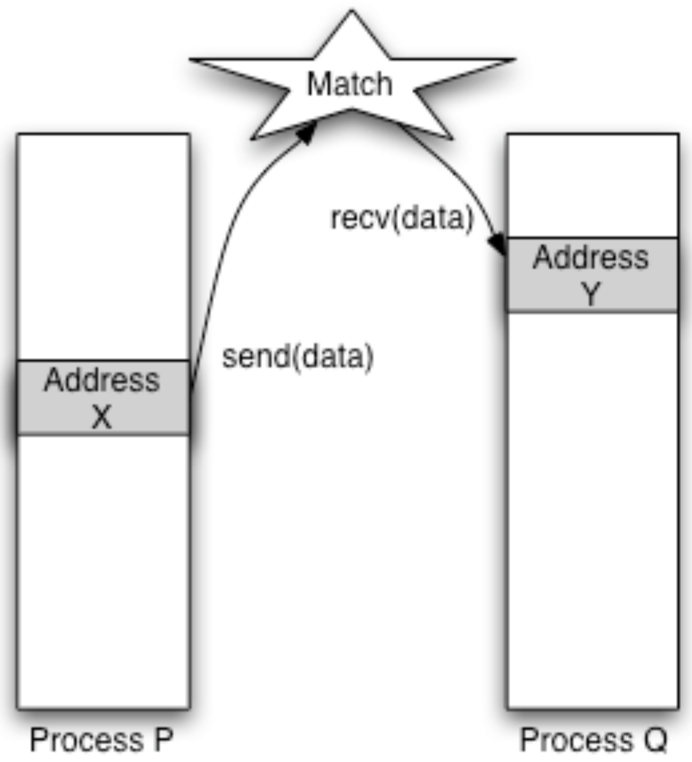
- Our ring program works with > 1 processes.
- What happens with just one process?



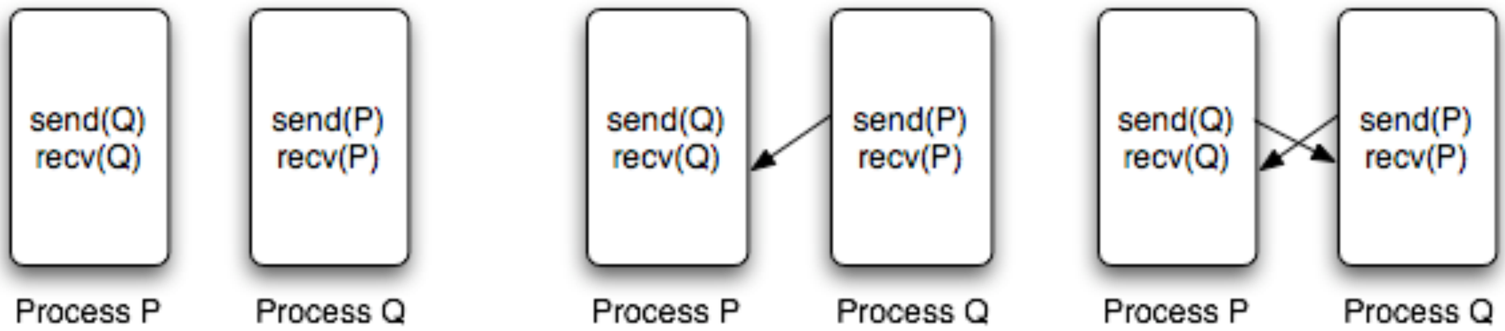


Deadlock!

```
std::string msg;
if (world.rank() == 0) {
  msg = "Hello from 0";
  world.send(0, 0, msg);
  world.recv(0, 0, msg);
}
```



Deadlock Situations



- Sends *can* block until matching receive is found:
  - Buffering in MPI might hide these problems.
  - Many MPI implementations detect them.

Non-Blocking Communication

- Separates the initiation of a communication operation from its completion
  - "i" prefix indicates a non-blocking operation
  - Non-blocking operations return a `request` object
  - `request` object used to complete the communication

Safe Send-to-Self Operation

```
std::string msg;
if (world.rank() == 0) {
  std::string out_msg = "Hello from 0";
  std::vector<mpi::request> requests;
  requests.push_back(world.isend(0, 0, out_msg));
  requests.push_back(world.irecv(0, 0, msg));
  mpi::wait_all(requests.begin(), requests.end());
}
```

Non-Blocking Requests

- One request per communication:

```
class request {
public:
  status wait();
  optional<status> test();
  void cancel();
};
```

Completing Multiple Requests

```
template<typename ForwardIterator,
         typename OutputIterator>
OutputIterator
wait_all(ForwardIterator first,
         ForwardIterator last, OutputIterator out);
```

- Wait until all requests have completed, output `status` objects.

Completing Multiple Requests

```
template<typename ForwardIterator>
std::pair<status, ForwardIterator>
wait_any(ForwardIterator first,
         ForwardIterator last);
```

- Wait until any request has completed, and return it.

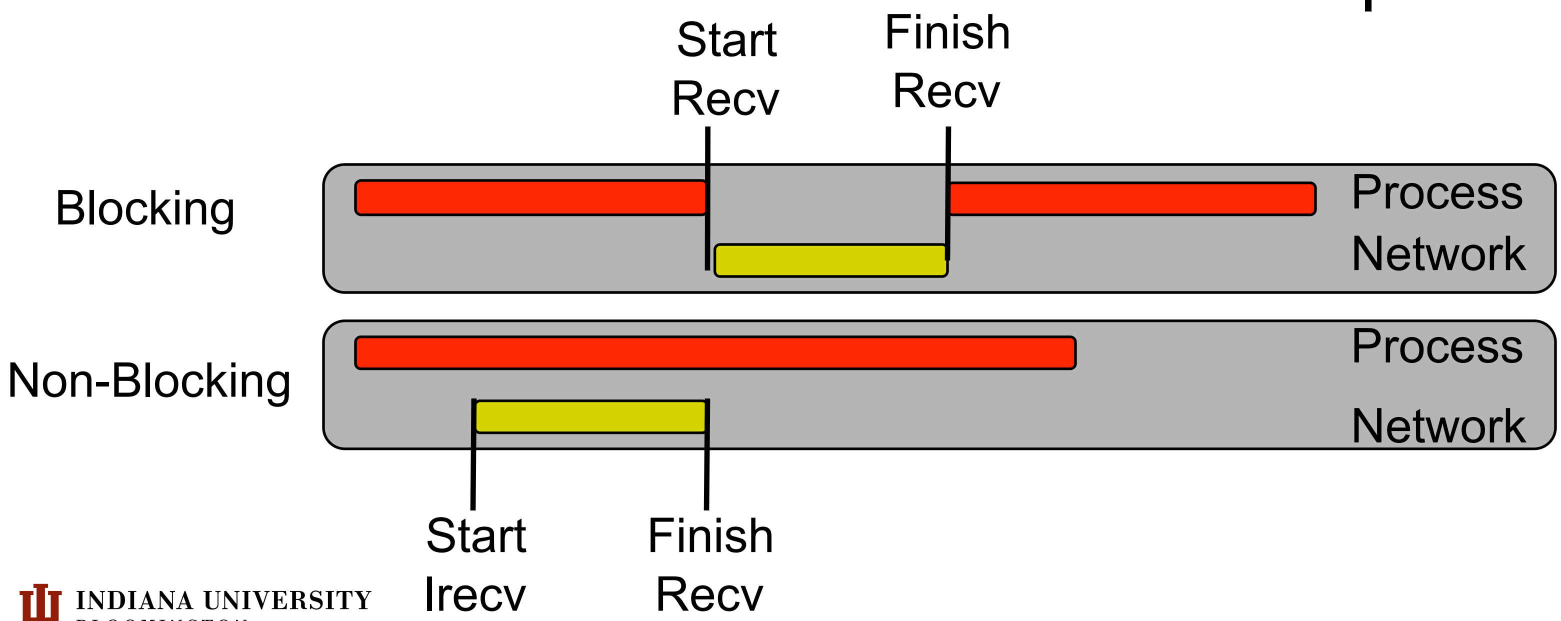
Testing Multiple Requests

```
template<typename ForwardIterator>
optional<std::pair<status, ForwardIterator> >
test_any(ForwardIterator first,
         ForwardIterator last);
```

- Determine whether any request has completd; if so, return it.

Non-Blocking Performance

- Try to maximize communication-computation overlap:
  - Start communication as early as possible
  - Finish communication as late as possible



Why Use Non-Blocking Comm?

- For correctness:
  - Good for deadlock avoidance.
- For performance:
  - Communication-computation overlap.
  - Remove synchronization between processes.

Working with Types

- Boost.MPI can transmit any serializable type

```
struct Point {
  float x, y, z;
  template<typename Archiver>
    void serialize(Archiver& ar, unsigned version) {
      ar & x & y & z;
    }
};
Point p = {3.14f, 2.71f, 0.0f};
comm.send(1, 0, p);
```

A Peek Under the Hood

- Boost.MPI uses Boost.Serialization with custom archivers
  - `mpi::packed_oarchive` serializes data in MPI's transmission format (`MPI_Pack`)
  - `mpi::packed_iarchive` de-serializes data (`MPI_Unpack`)
- Allows interoperability with MPI programs written without Boost.MPI

Some Types Are More Equal Than Others

- Serialization is the lowest common denominator
  - Unnecessary serialization → poor performance
- Boost.MPI can bypass serialization for some types:
  - Primitive types will be transmitted *fast*
  - POD classes can bypass serialization

Transmitting `Points`, Fast

```
struct Point {
  float x, y, z;
  template<typename Archiver>
    void serialize(Archiver& ar, unsigned version) {
      ar & x & y & z;
    }
};
```

- Enabling the optimization:

```
namespace boost { namespace mpi {
  template<>
    struct is_mpi_datatype<Point> : mpl::true_ { };
} }
```

Back Under the Hood

- MPI *derived datatypes* describe C structs for direct transmission
- Boost.MPI builds derived datatypes via Boost.Serialization
  - `mpi_datatype_oarchive` records offsets and types of struct members for MPI

Communicators and Groups

- Each communicator is a separate communication space
  - Distinct tags/messages from other communicators
  - Provides communication for an arbitrary subset of the MPI processes
  - Great for parallel libraries!

Cloning Communicators

- Build a new communicator with the same processes as the old communicator:

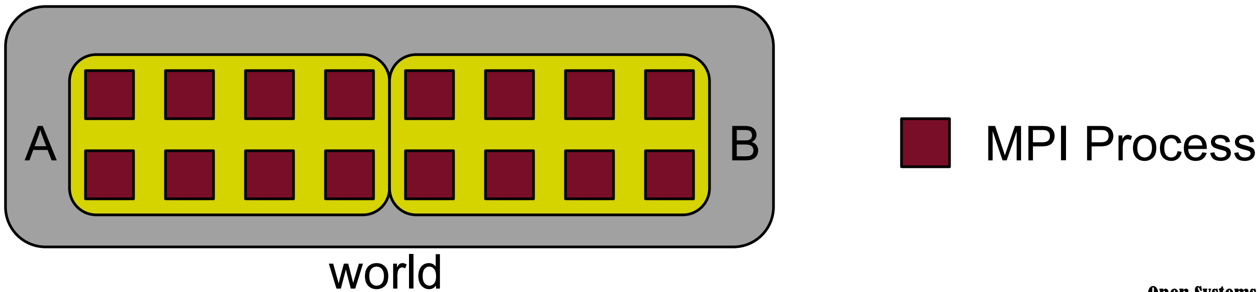
```
communicator comm(world, comm.group());
```

- Typically used by libraries to separate their communication from users.

Splitting Into Subgroups

- Split the world into two subgroups, each with its own communicator:
  - Group A: First half of processes
  - Group B: Second half of processes

```
int half = world.size() / 2;
communicator subcomm
  = world.split(world.rank() < half? 0 : 1);
```



Rank Translation

- Each communicator `C` has ranks numbered 0..`C.size()`-1.
  - Due to subgroups, a process may have different ranks in different communicators
- Rank translation:

```
std::vector<int> subRanks, worldRanks;
for (int p = 0; p < subcomm.size(); ++p)
  subRanks.push_back(p);
subcomm.group().translate_ranks(subRanks.begin(),
   subRanks.end(), world.group(),
   std::back_inserter(worldRanks));
```

Advanced Group Manipulation

- `communicator::group` provides more advanced manipulation of groups
- Group operations:
  - `operator&`: intersection
  - `operator|`: union
  - `operator-`: difference
  - include/exclude specific ranks

When To Use (Sub)groups?

- In a library, clone the communicator to get your own communication space.
- If you are dividing your tasks into pieces that each cover several processes, use a communicator for each subgroup.

Collectives II – Reduce

```
template<typename T, typename Op>
void reduce(const communicator& comm,
            const T& in_value, T& out_value, Op op,
            int root);
```

- Reduce combines the values provided by each processor into a single value
  - `out = op(in`$_0$`, op(in`$_1$`, op(in`$_2$`, ...)...)`
  - `op` must be an *associative* binary operation

Reduction Example

- Compute the global sum from local sums:

```
int localSum = ..., globalSum;
reduce(world, localSum, globalSum, std::plus<int>(),
       0);
if (world.rank() == 0) std::cout << globalSum;
```

- We could have written:

```
std::vector<int> sums;
gather(world, localSum, sums, 0);
if (world.rank() == 0) {
  globalSum = accumulate(sums.begin(), sums.end());
}
```

Collectives II – Allreduce

```
template<typename T, typename Op>
T all_reduce(const communicator& comm,
             const T& in_value, Op op);
```

- Allreduce is a reduce that broadcasts the result to all processes.

```
int localMin = ...;
int globalMin = all_reduce(world, localMin,
                           mpi::minimum<int>());
```

Global Common Prefix

- Task: compute the longest prefix common to the strings stored in each process
- Example:
  - Process 0 has: “foot”
  - Process 1 has “foolish”
  - Process 2 has “foobar”
  - Result: “foo”

Example:Global Common Prefix

```
string
common_prefix(const string& s1, const string& s2) {
  if (s1.size() <= s2.size())
    return string(s1.begin(),
                  mismatch(s1.begin(), s1.end(),
                           s2.begin()).first);
  else return common_prefix(s2, s1);
}

string global_common_prefix
  = all_reduce(comm, my_string, &common_prefix);
```

Collectives II – Scan

```
template<typename T, typename Op>
T scan(const communicator& comm, const T& in_value,
       Op op);
```

- Scan is a parallel prefix operation.
  - Process `p` receives the result of reducing values $\mathtt{in}_0 \mathtt{..in}_i$
  - Typically used to compute partial sums (like the `partial_sum` algorithm)

Implementation of Collectives

- Boost.MPI re-implements all of the MPI collectives for serialized data types
- *Pro*: Provides greatly improved functionality over existing MPI collectives
- *Con*: MPI implementers have spent *years* optimizing their collectives.
  - We'll never be that fast.

Specialized Collectives

1. C MPI collective with predefined `MPI_Op`
2. C MPI collective with new `MPI_Op`, created implicitly with `MPI_Op_create`.
3. Boost.MPI implementation (commutative)
4. Boost.MPI implementation (associative only)

Data Type Kind

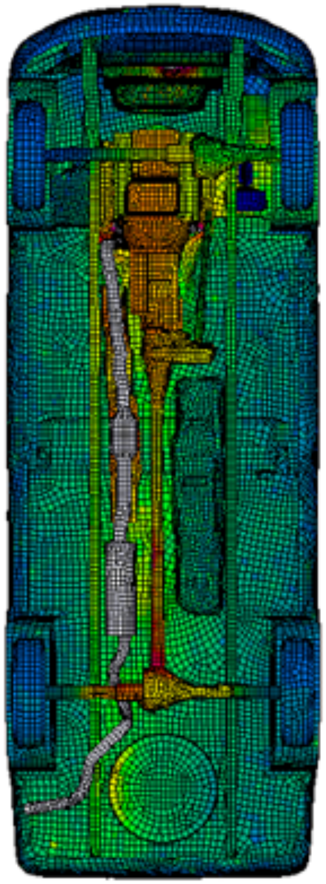
Function Object		Built-In	Datatype	Serialized
	Predefined	1, 2	2	3, 4
	Commutative	2	2	3
	Associative	2	2	4

Point-to-Point vs. Collectives

- Use collectives when:
  - A collective matches the needs of your application
  - Your tasks are relatively well-balanced
- Use point-to-point messages when:
  - No collective matches your communication
  - Non-blocking communication can improve performance

Static Structure, Dynamic Data

- Many problems iteratively compute values on a static data structure
  - Example: Finite Element Analysis
- Characteristics:
  - Distributed data structure is static
  - Boundary values in the data structure exchanged in each iteration
- Values are non-contiguous, but serialization is too inefficient.



thermoanalytics.com

Separating the Meat from the Bones

- Boost.MPI provides abstractions for these problems:
  - `skeleton`: transmit "shape" of data once
  - `content`: transmit contents of data many times, efficiently

Gaspar Becerra, ca. 1520-1570
Royal Academy of Arts Collection

Skeleton/Content Example

- Computation:

```
mesh2d<cell> m;
// load the mesh with data
comm.send(1,0,skeleton(m));

while(!done ) {
  // update local cells...
  // broadcast changes
  comm.send(1,1,
    get_content(m));
}
```

- Rendering:

```
mesh2d<cell> m;
// receive the structure
// of the mesh
comm.recv(0,0,skeleton(m));

while (!done) {
  comm.recv(0,1,
    get_content(m));
  // display updated m
}
```

Summary & Recap

- MPI is designed for high-performance parallel applications
  - Desktop → Cluster → Supercomputer
- Major themes:
  - Communicators for cooperating among processes on the same task
  - Collectives to summarize/move data
  - Point-to-point for fine-grained communication

A Simple Task

- Transmit a list of integer lists via MPI:

```
std::list<std::list<int> > ls;

int num_lists = ls.size();
MPI_Send(&num_lists, 1, MPI_INT, dest, tag, comm);
std::list<std::list<int> >::iterator out_iter
  = ls.begin();
while (out_iter != ls.end()) {
  std::vector<int> buffer(out_iter->begin(),
                          out_iter->end());
  MPI_Send((void*)&buffer.front(),out_iter->size(),
           MPI_INT, dest, tag, comm);
  ++out_iter;
}
```

Transmits Multiple Messages

- Transmit a list of integer lists via MPI:

```
std::list<std::list<int> > ls;

int num_lists = ls.size();
MPI_Send(&num_lists, 1, MPI_INT, dest, tag, comm);
std::list<std::list<int> >::iterator out_iter
  = ls.begin();
while (out_iter != ls.end()) {
  std::vector<int> buffer(out_iter->begin(),
                          out_iter->end());
  MPI_Send((void*)&buffer.front(),out_iter->size(),
           MPI_INT, dest, tag, comm);
  ++out_iter;
}
```

Needs Explicit Serialization

- Transmit a list of integer lists via MPI:

```
std::list<std::list<int> > ls;

int num_lists = ls.size();
MPI_Send(&num_lists, 1, MPI_INT, dest, tag, comm);
std::list<std::list<int> >::iterator out_iter
  = ls.begin();
while (out_iter != ls.end()) {
  std::vector<int> buffer(out_iter->begin(),
                          out_iter->end());
  MPI_Send((void*)&buffer.front(),out_iter->size(),
           MPI_INT, dest, tag, comm);
  ++out_iter;
}
```

Uses Unsafe Type Casts

- Transmit a list of integer lists via MPI:

```
std::list<std::list<int> > ls;

int num_lists = ls.size();
MPI_Send(&num_lists, 1, MPI_INT, dest, tag, comm);
std::list<std::list<int> >::iterator out_iter
  = ls.begin();
while (out_iter != ls.end()) {
  std::vector<int> buffer(out_iter->begin(),
                          out_iter->end());
  MPI_Send((void*)&buffer.front(),out_iter->size(),
           MPI_INT, dest, tag, comm);
  ++out_iter;
}
```

Redundant Information

- Transmit a list of integer lists via MPI:

```
std::list<std::list<int> > ls;

int num_lists = ls.size();
MPI_Send(&num_lists, 1, MPI_INT, dest, tag, comm);
std::list<std::list<int> >::iterator out_iter
  = ls.begin();
while (out_iter != ls.end()) {
  std::vector<int> buffer(out_iter->begin(),
                          out_iter->end());
  MPI_Send((void*)&buffer.front(),out_iter->size(),
           MPI_INT, dest, tag, comm);
  ++out_iter;
}
```

Point-to-Point in Boost.MPI

- Transmit a list of integer lists via Boost.MPI:

```
std::list<std::list<int> > ls;

comm.send(dest, tag, ls);
```

- Simple, obvious interface:
  - Treats user-defined, library-defined and built-in types uniformly
  - Eliminates need for redundancies and type casting
  - Retain same names, semantics as C MPI

MPI Features

- Point-to-point communication
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal (blocking and non-blocking), synchronous, ready, buffered
- Collective
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology

MPI Features (cont'd)

- Dynamic process control
  - Allows creation and cooperative termination of processes after an MPI application has started.
  - Mechanism to establish communication between existing MPI applications.
- One-sided operations
  - Remote Memory Access (RMA) communication mechanisms
  - Communication: Put (remote write), Get (remote read) and Accumulate (remote update)
  - Support for both active and passive target synchronization.

MPI Features (cont'd)

- Parallel I/O
  - Portable interface for optimized parallel file access
  - Support for synchronous and asynchronous I/O
  - Allows for close coupling with parallel filesystems
  - Data partitioning expressed using derived datatypes.

MPI Features (cont'd)

- Application-oriented process topologies
  - Built-in support for grids and graphs (based on groups)
- Profiling
  - Hooks allow users to intercept MPI calls to install their own tools
- Environmental
  - Inquiry
  - Error control