MPI and comparison of models
Lecture 23, cs262a

Ion Stoica & Ali Ghodsi
UC Berkeley
April 16, 2018

MPI

MPI - Message Passing Interface

- Library standard defined by a committee of vendors, implementers, and parallel programmers
- Used to create parallel programs based on message passing

Portable: one standard, many implementations

- Available on almost all parallel machines in C and Fortran
- De facto standard platform for the HPC community

Groups, Communicators, Contexts

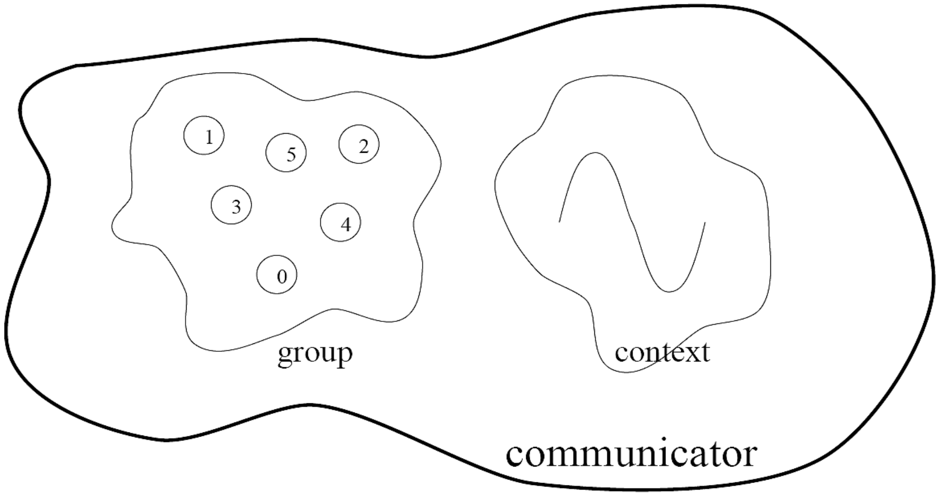
Group: a fixed ordered set of *k* processes, with **ranks**, i.e., 0, 1, …, k-1

Communicator: specify scope of communication

- Between processes in a group (intra)
- Between two disjoint groups (inter)

Context: partition of comm. space

- A message sent in one context cannot be received in another context



***This image is captured from: "Writing Message Passing Parallel Programs with MPI", Course Notes**,*
Edinburgh Parallel Computing Centre
The University of Edinburgh

Synchronous vs. Asynchronous Message Passing

A synchronous communication is not complete until the message has been received

An asynchronous communication completes before the message is received

Communication Modes

Synchronous: completes once ack is received by sender

Asynchronous: 3 modes

- Standard send: completes once the message has been sent, which may or may not imply that the message has arrived at its destination
- Buffered send: completes immediately, if receiver not ready, MPI buffers the message locally
- Ready send: completes immediately, if the receiver is ready for the message it will get it, otherwise the message is dropped silently

Blocking vs. Non-Blocking

Blocking, means the program will not continue until the communication is completed

- Synchronous communication
- Barriers: wait for every process in the group to reach a point in execution

Non-Blocking, means the program will continue, without waiting for the communication to be completed

MPI library

Huge (125 functions)

Basic (6 functions)

MPI Basic

Many parallel programs can be written using just these six functions, only two of which are non-trivial;

- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_SIZE
- MPI_COMM_RANK
- **MPI_SEND**
- **MPI_RECV**

Skeleton MPI Program (C)

```
#include <mpi.h>

main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    /* main part of the program */

   /* Use MPI function call depend on your data
     * partitioning and the parallelization architecture
    */
    MPI_Finalize();
}
```

A minimal 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;
}
```

A minimal MPI program (C)

`#include “mpi.h”` provides basic MPI definitions and types.

MPI_Init starts MPI

MPI_Finalize exits MPI

Notes:

- Non-MPI routines are local; this “printf” run on each process
- MPI functions return error codes or MPI_SUCCESS

Improved Hello (C)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
     int rank, size;
     MPI_Init(&argc, &argv);
     /* rank of this process in the communicator */
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     /* get the size of the group associates to the communicator */
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     printf("I am %d of %d\n", rank, size);
     MPI_Finalize();
     return 0;
}
```

Improved Hello (C)

```
/* Find out rank, size */
int world_rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size                LD            );
int number;
if (world_rank == 0)
  number = -1;
  MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
  MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("Process 1 received number %            ess 0\n", num
}
```

Number of elements

Rank of destination

Tag to identify message

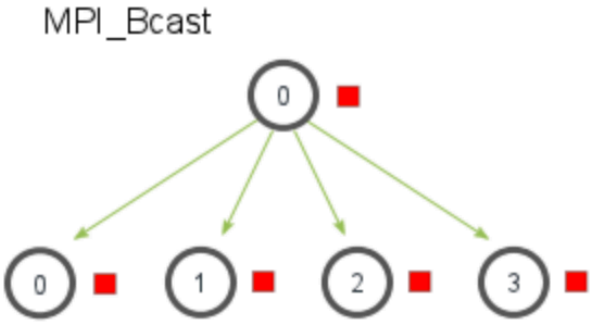
Default communicator

Rank of source

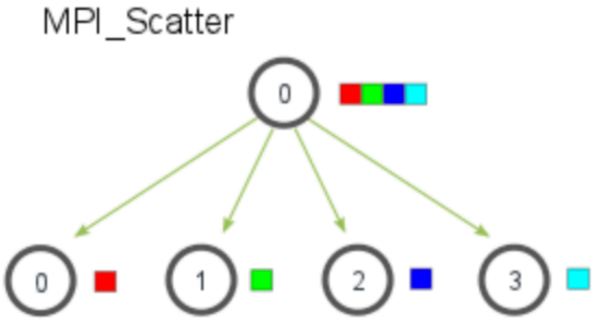
Status

Many other functions…

MPI_Bcast: send same piece of data to all processes in the group



MPI_Scatter: send different pieces of an array to different processes (i.e., partition an array across processes)

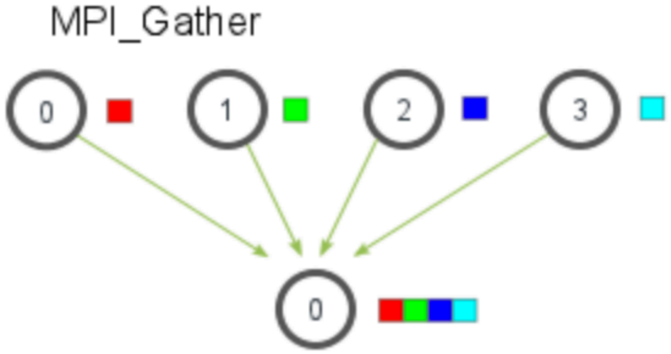


From: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Many other functions…

MPI_Gather: take elements from many processes and gathers them to one single process

- E.g., parallel sorting, searching

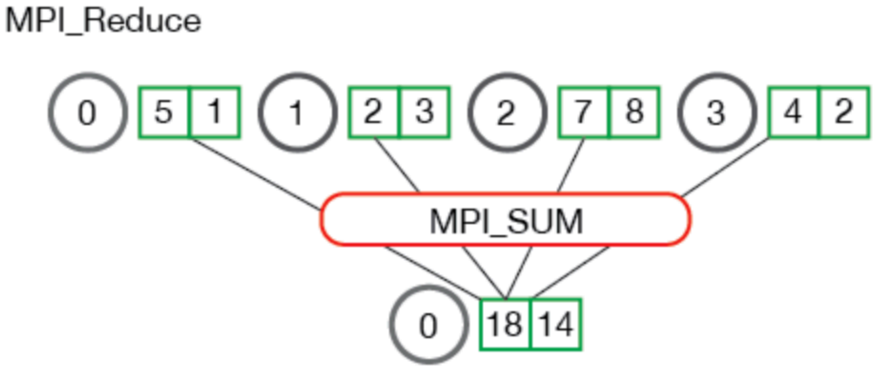


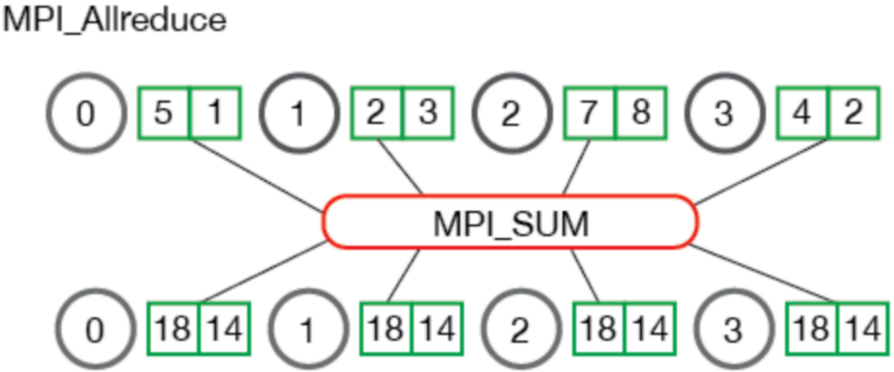
From: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Many other functions…

MPI_Reduce: takes an array of input elements on each process and returns an array of output elements to the root process given a specified operation

MPI_Allreduce: Like MPI_Reduce but distribute results to all processes





From: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

MPI Discussion

Gives full control to programmer

- Exposes number of processes
- Communication is explicit, driven by the program

Assume

- Long running processes
- Homogeneous (same performance) processors

Little support for failures (checkpointing), no straggler mitigation

Summary: achieve high performance by hand-optimizing jobs but requires experts to do so, and little support for fault tolerance

Today's Paper

A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard,
William Gropp, Ewing Lusk, Nathan Doss, Anthony Skjellum, Journal of Parallel Computing, Vol 22, Issue 6, Sep 1996
https://ucbrise.github.io/cs262a-spring2018/notes/MPI.pdf

MPI Chameleon

- Many MPI implementations existed. MPICH's goal was to create an implementation that was both
  - Portable (hence the name CHameleon)
  - Performant

Portability

MPICH is portable and leverages:

- High performance switches
  - Supercomputers where different node communicate over switches (Paragon, SP2, CM-5)
- Shared memory architectures
  - Implement efficient message passing on these machines (SGI Onyx)
- Networks of workstations
  - Ethernet connected distributed systems communicating using TCP/IP

Performance

- MPI standard already allowed to optimizations where usability wasn’t restricted.
- MPICH comes with performance test suite (mpptest), it works both on MPICH but also on top of other MPI implementations!

Performance & Portability tradeoff

- Why is there a tradeoff?
    - Custom implementation for each hardware (+performance)
    - Shared re-usable code across all hardware (+quick portability)
- Keep in mind that this was the era of super computers
    - IBM SP2, Meiko CS-2, CM-5, NCube-2 (fast switching)
    - Cray T3D, SGI Onyx, Challenge, Power Challenge, IBM SMP (shared memory)
    - How do you use all the advanced hardware features ASAP?
- This paper shows you how to have your cake and eat it too!

How to eat your cake and have it too?

- Small narrow **Abstract Device Interface (ADI)**
    - Implemented on lots of different hardwares
    - Highly tuned and performant
    - Uses an even smaller (5 function) **Channel Interface**.

- Implement all of MPI on top of ADI and the Channel interface
    - Porting to a new hardware requires porting ADI/Channel implementations.
    - All of the rest of the code is re-used (+portability)
    - Super fast message passing for various hardwares (+performance)

MPICH Architecture

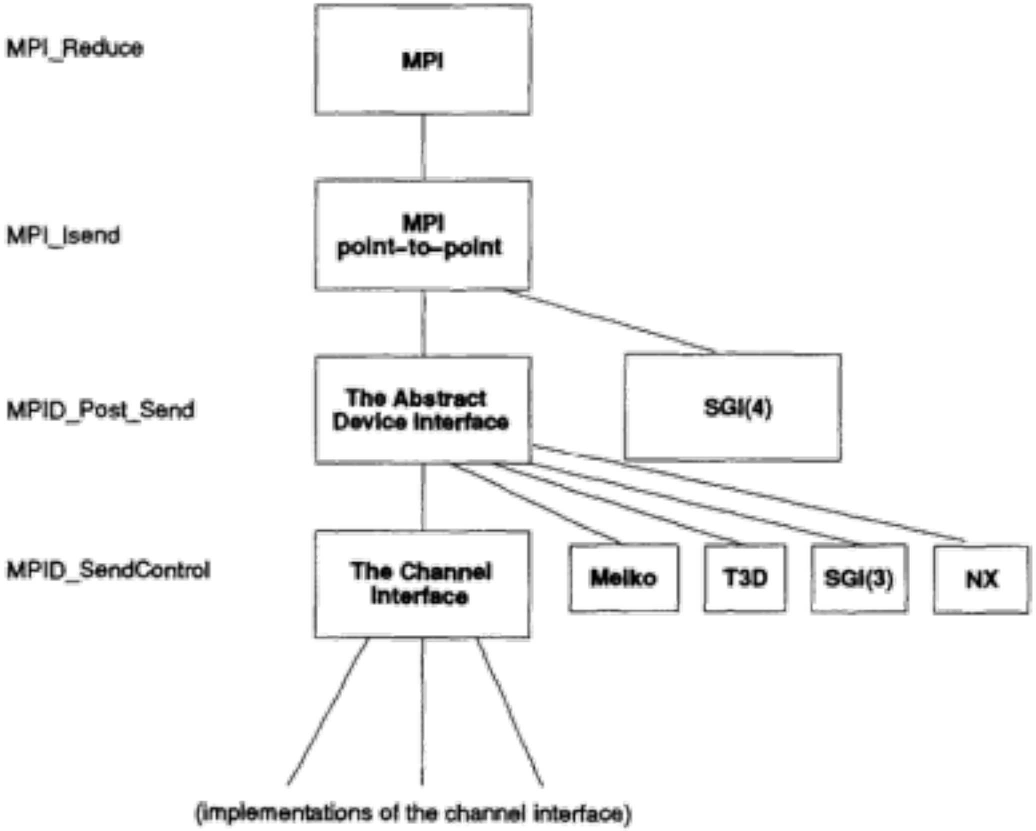


Fig. 7. Upper layers of MPICH.

ADI functions

1. Message abstraction
2. Moving messages from MPICH to actual hardware
3. Managing mailboxes (messages received/sent)
4. Providing information about the environment

If some hardware doesn't support the above, then emulate it.

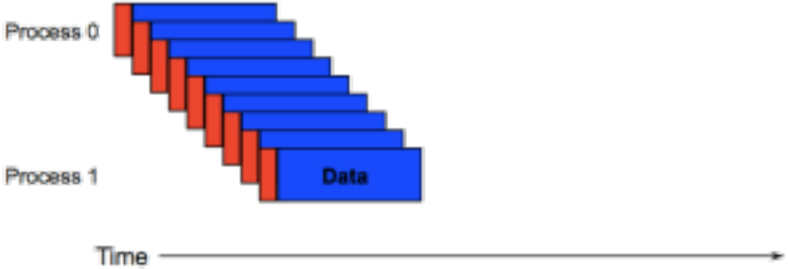
Channel Interface

Implements transferring data or envelope (e.g. communicator, length, tag) from one process to another

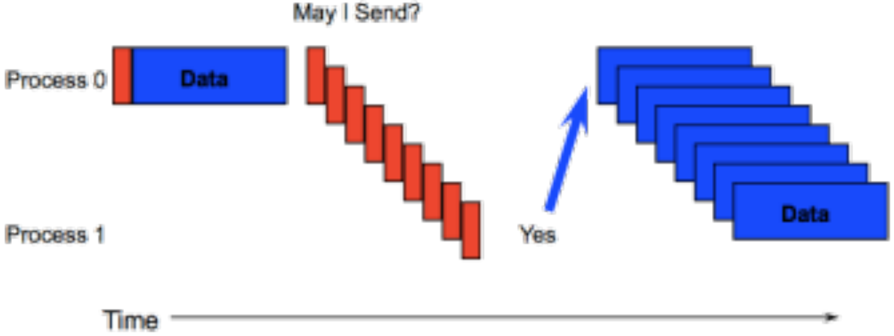
1. `MPID_SendChannel` to send a message
2. `MPID_RecvFromChannel` to receive a message
3. `MPID_SendControl` to send control (envelope) information
4. `MPID_ControlMsgAvail` checks if new ctrl msgs available
5. `MPID_RecvAnyControl` to receive any control message

Assuming that the hardware implements buffering. Tradeoff!

Eager vs Rendezvous



- Eager mode immediately sends data to receiver
  - Deliver envelope and data immediately without checking with recv
- Rendezvous
  - Deliver envelope, but check that receiver is ready to recv before sending data
- Pros/Cons? Why needed?
  - Buffer overflow, asynchrony!
  - Speed vs robustness



How to use Channel Interface?

Shared memory

- Complete channel implementation with malloc, locks, mutex:es

Specialized

- Bypass shard memory portability, use hardware directly available in SGI and HPI shared memory systems

Scalable Coherent Interface (SCI)

- Special implementation that uses the SCI standard

Competitive performance vs vendor specific solutions

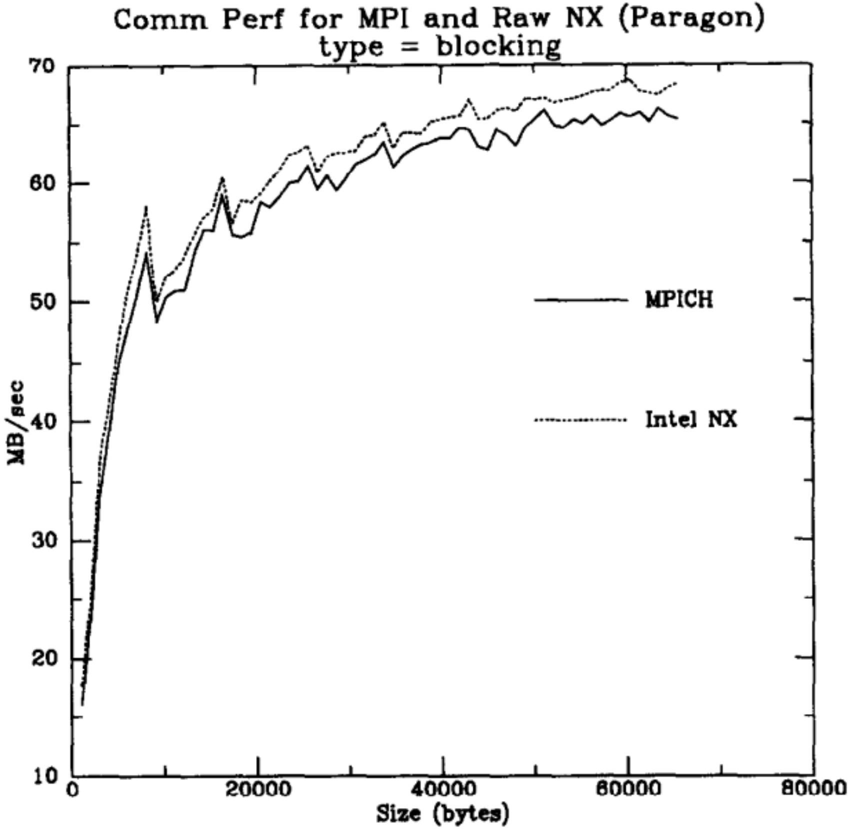


Fig. 1. MPICH vs. NX on the Paragon.

Summary

- Many super computer hardware implementations
- Difficult to port MPI to all of them and use all specialized hardware
- Portability vs Performance tradeoff
    - MPICH achieved both by carefully designing a kernel API (ADI)
    - Many ADI implementations that leverage hardware (performance)
    - All of MPICH build on ADI (portability)

Discussion

	MPI	OpenMP	Ray	Spark / MapReduce
Environment, Assumptions	Supercomputers Sophisticated programmers High performance Hard to scale hardware	Single node, multiple core, shared memory	Commodity clusters Python programmers	Commodity clusters Java programmers Programmer productivity Easier, faster to scale up cluster
Computation Model	Message passing Lowest level	Shared memory Low level	Data flow / task parallel High level	Data flow / BSP Highest level
Strengths	Fastest asynchronous code	Simplifies parallel programming on multi-cores	Very high performance Very flexible	Very easy to use for parallel data processing (seq. control) Maximum fault tolerance
Weaknesses	Fault tolerance Easy to end up with non-deterministic code (if not using barriers), more code	Pretty complex, need to be careful about race conditions	Need to understand program structure for best performance; Inefficient fault-tolerance for actors	Harder to implement irregular computations (e.g., nested parallelism, ignore stragglers); Lower performance