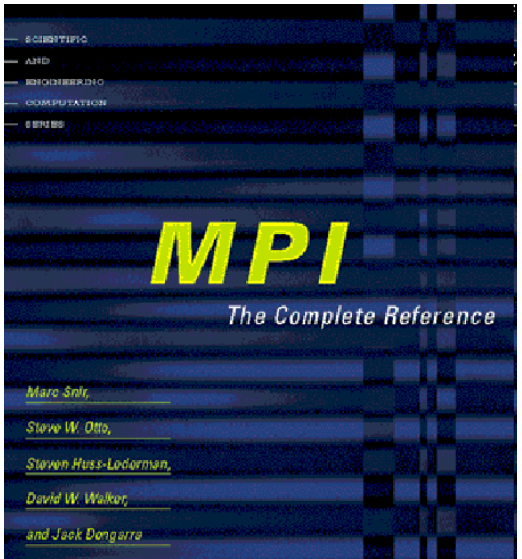
A Message Passing Standard for MPP and Workstations

Communications of the ACM, July 1996

J.J. Dongarra, S.W. Otto, M. Snir, and D.W. Walker



SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

MPI
The Complete Reference

Marc Snir,
Steve W. Otto,
Steven Huss-Lederman,
David W. Walker,
and Jack Dongarra

Message Passing Interface (MPI)

Message passing library

Can be added to sequential languages (C, Fortran)

Designed by consortium from industry, academics, government

Goal is a message passing standard

MPI Programming Model

Multiple Program Multiple Data (MPMD)

- Processors may execute different programs (unlike SPMD)
- Number of processes is fixed (one per processor)
- No support for multi-threading

Point-to-point and collective communication

MPI Basics

MPI_INIT	initialize MPI
MPI_FINALIZE	terminate computation
MPI_COMM SIZE	number of processes
MPI_COMM RANK	my process identifier
MPI_SEND	send a message
MPI_RECV	receive a message

Language Bindings

Describes for a given base language

- concrete syntax
- error-handling conventions
- parameter modes

Popular base languages:

- C
- Fortran

Point-to-point message passing

Messages sent from one processor to another are FIFO ordered
Messages sent by different processors arrive non-deterministically

Receiver may specify source
- source = sender's identity => symmetric naming
- source = MPI_ANY_SOURCE => asymmetric naming
- example: specify sender of next pivot row in ASP

Receiver may also specify tag
- Distinguishes different kinds of messages
- Similar to operation name in SR

Examples (1/2)

```
int x, status;
float buf[10];

MPI_SEND (buf, 10, MPI_FLOAT, 3, 0, MPI_COMM_WORLD);
    /* send 10 floats to process 3; MPI_COMM_WORLD = all processes */

MPI_RECV (&x, 1, MPI_INT, 15, 0, MPI_COMM_WORLD, &status);
    /* receive 1 integer from process 15 */

MPI_RECV (&x, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    /* receive 1 integer from any process */
```

Examples (2/2)

```
int x, status;
#define NEW_MINIMUM 1

MPI_SEND (&x, 1, MPI_INT, 3, NEW_MINIMUM, MPI_COMM_WORLD);
    /* send message with tag NEW_MINIMUM */.

MPI_RECV (&x, 1, MPI_INT, 15, NEW_MINIMUM, MPI_COMM_WORLD, &status);
   /* receive 1 integer with tag NEW_MINIMUM */

MPI_RECV (&x, 1, MPI_INT, MPI_ANY_SOURCE, NEW_MINIMUM,
   MPI_COMM_WORLD, &status);
   /* receive tagged message from any source */
```

Forms of message passing

MPI has a very wide and complex variety of primitives

Programmer can decide whether to:

- minimize copying overhead
  -> synchronous and ready-mode sends
- minimize idle time, overlap communication & computation
  -> buffered sends and nonblocking sends

Communication modes control the buffering

(Non)blocking determines when sends complete

Communication modes

- **Standard:**
  - Programmer cannot make any assumptions whether the message is buffered, this is up to the system
- **Buffered:**
  - Programmer provides (bounded) buffer space
  - SEND completes when message is copied into buffer (local)
  - Erroneous if buffer space is insufficient
- **Synchronous:**
  - Send waits for matching receive
  - No buffering -> easy to get deadlocks
- **Ready:**
  - Programmer asserts that receive has already been posted
  - Erroneous if there is no matching receive yet

Unsafe programs

- MPI does not guarantee any system buffering
- Programs that assume it are unsafe and may deadlock

- Example of such a deadlock:

```
Machine 0:
MPI_SEND (&x1, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
MPI_RECV (&x2, 10, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);

Machine 1:
MPI_SEND (&y1, 10, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_RECV (&y2, 10, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```

(Non)blocking sends

A blocking send returns when it’s safe to modify its arguments

A non-blocking ISEND returns immediately (dangerous)

```
int buf[10];
request = MPI_ISEND (buf, 10, MPI_FLOAT, 3, 0, MPI_COMM_WORLD);
compute();  /* these computations can be overlapped with the transmission */
buf[2]++;  /* dangerous: may or may not affect the transmitted buf */
MPI_WAIT(&request, & status);  /* waits until the ISEND completes */
```

Don’t confuse this with synchronous vs asynchronous !

synchronous = wait for (remote) receiver

nonblocking = wait till arguments have been saved (locally)

Non-blocking receive

MPI_IPROBE	check for pending message
MPI_PROBE	wait for pending message
MPI_GET_COUNT	number of data elements in message

```
MPI_PROBE (source, tag, comm, &status)       =>  status
MPI_GET_COUNT (status, datatype, &count)   =>  message size
status.MPI_SOURCE   =>  identity of sender
status.MPI_TAG          =>  tag of message
```

Example: Check for Pending Messages

```
int buf[1], flag, source, minimum;
while ( ...) {
    MPI_IPROBE(MPI_ANY_SOURCE, NEW_MINIMUM, comm, &flag, &status);
    while (flag) {
          /* handle new minimum */
          source = status.MPI_SOURCE;
          MPI_RECV (buf, 1, MPI_INT, source, NEW_MINIMUM, comm, &status);
          minimum = buf[0];
          /* check for another update */
          MPI_IPROBE(MPI_ANY_SOURCE, NEW_MINIMUM, comm, &flag, &status);
    }
    ...   /* compute */
}
```

Example: Receiving Message with Unknown Size

```
int count, *buf, source;
MPI_PROBE(MPI_ANY_SOURCE, 0, comm, &status);
source = status.MPI_SOURCE;
MPI_GET_COUNT (status, MPI_INT, &count);
buf = malloc (count * sizeof (int));
MPI_RECV (buf, count, MPI_INT, source, 0, comm, &status);
```

Global Operations - Collective Communication

Coordinated communication involving all processes

Functions:

MPI_BARRIER	synchronize all processes
MPI_BCAST	send data to all processes
MPI_GATHER	gather data from all processes
MPI_SCATTER	scatter data to all processes
MPI_REDUCE	reduction operation
MPI_REDUCE ALL	reduction, all processes get result

Barrier

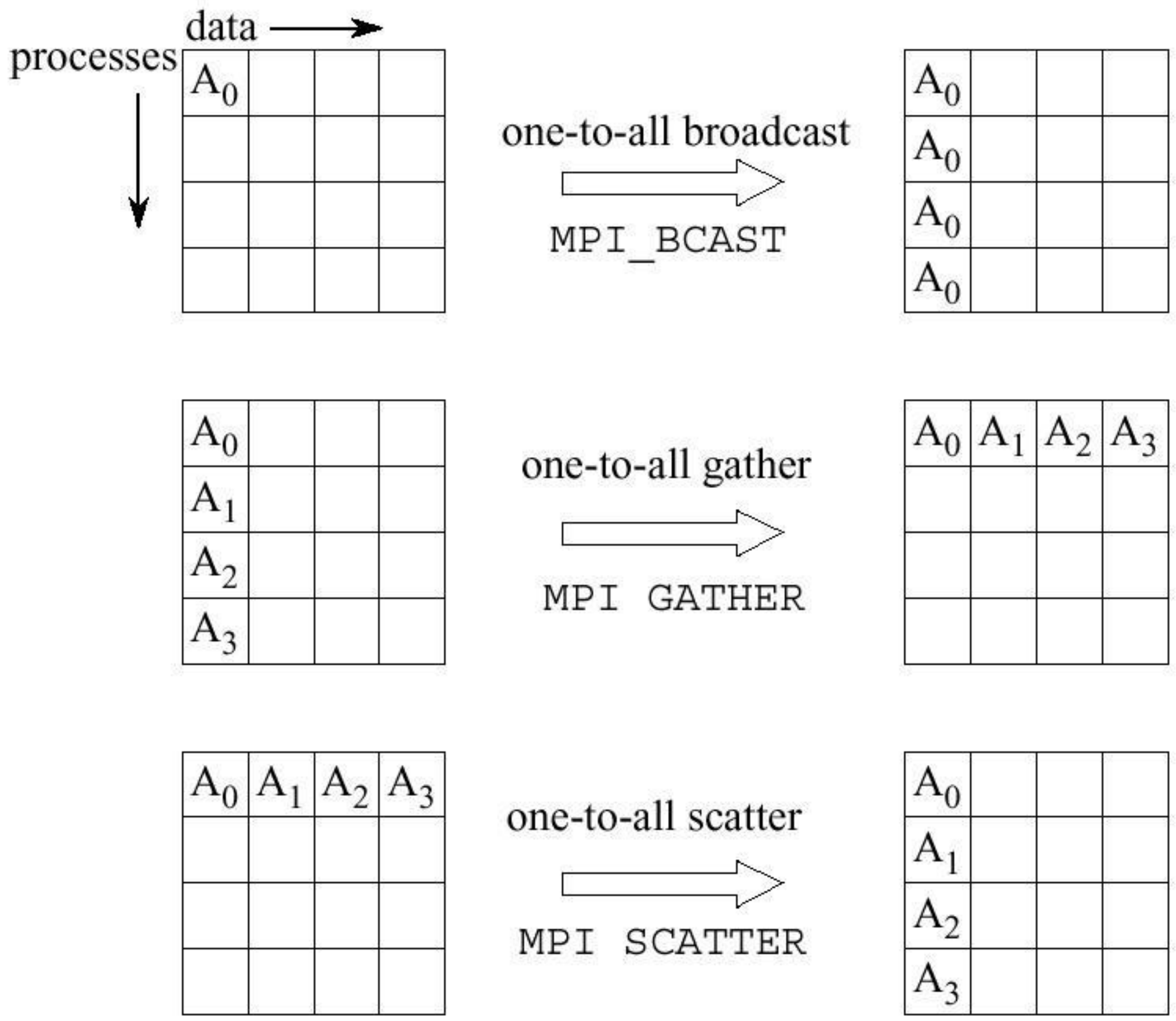
MPI_BARRIER (comm)

Synchronizes group of processes

All processes block until all have reached the barrier

Often invoked at end of loop in iterative algorithms

Figure 8.3 from Foster's book



Reduction

Combine values provided by different processes

Result sent to one processor (MPI REDUCE) or all processors (MPI REDUCE ALL)

Used with commutative and associative operators:

MAX, MIN, +, x , AND, OR

Example 1

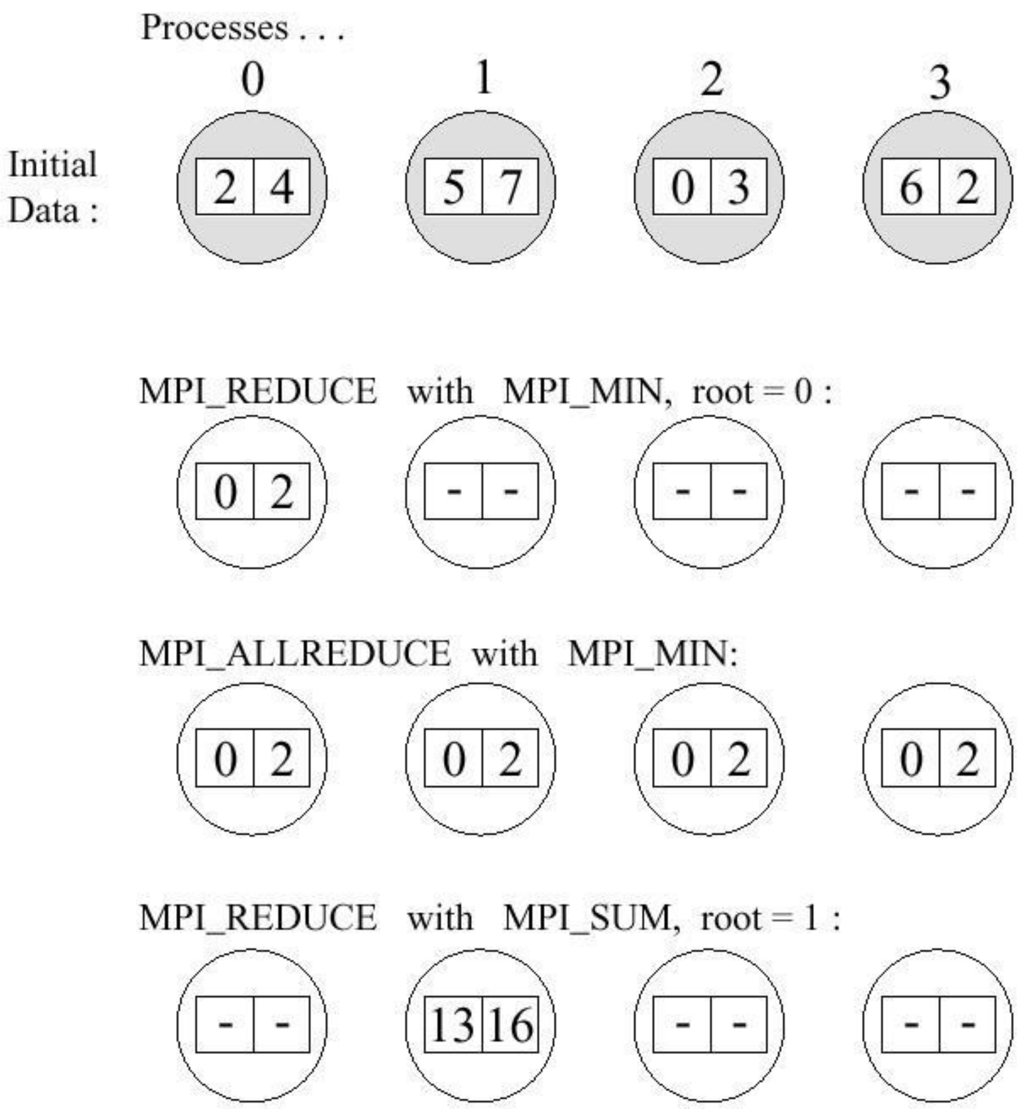
Global minimum operation

```
MPI_REDUCE (inbuf, outbuf, 2, MPI_INT, MPI_MIN,
                                  0, MPI_COMM_WORLD)
```

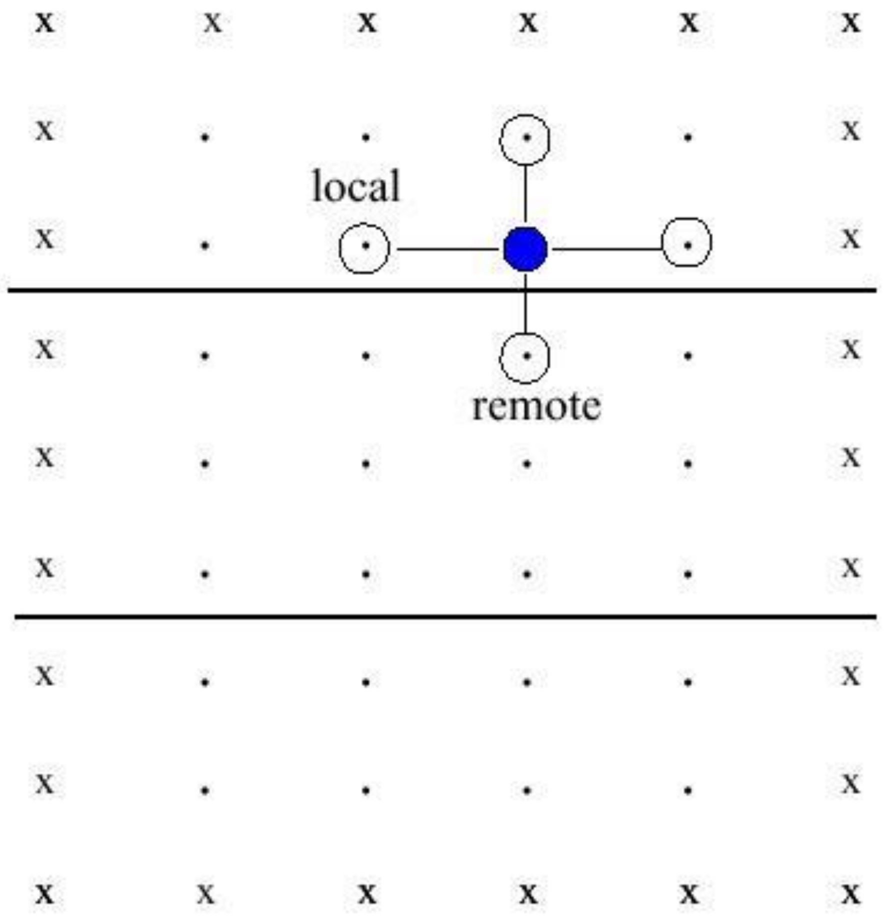
outbuf[0] = minimum over inbuf[0]'s

outbuf[1] = minimum over inbuf[1]'s

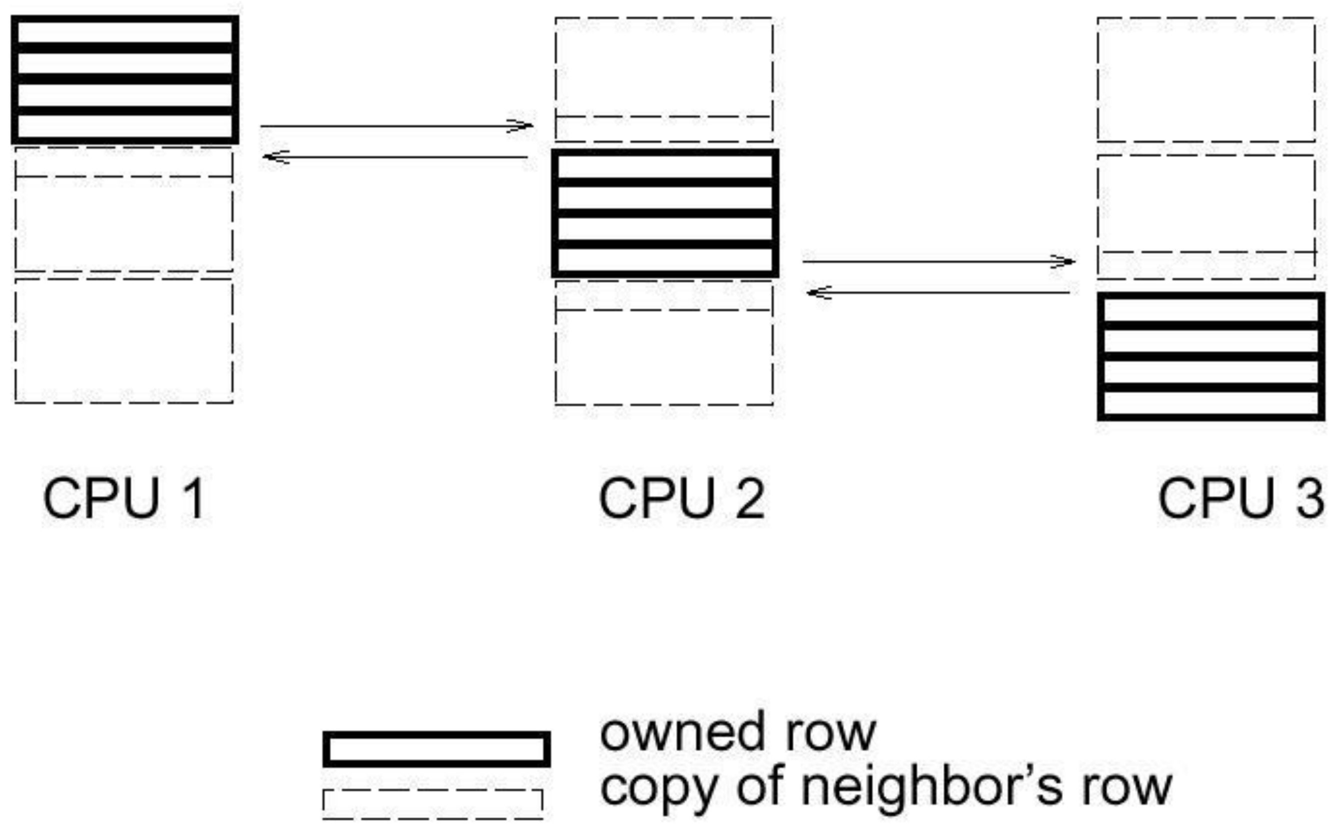
Figure 8.4 from Foster's book


Processes . . .
0
1
2
3
Initial Data :
2 4
5 7
0 3
6 2
MPI_REDUCE with MPI_MIN, root = 0 :
0 2
- -
- -
- -
MPI_ALLREDUCE with MPI_MIN:
0 2
0 2
0 2
0 2
MPI_REDUCE with MPI_SUM, root = 1 :
- -
13 16
- -
- -

Example 2: SOR in MPI



SOR communication scheme



Each CPU communicates with left & right neighbor (if existing)

Also need to determine convergence criteria

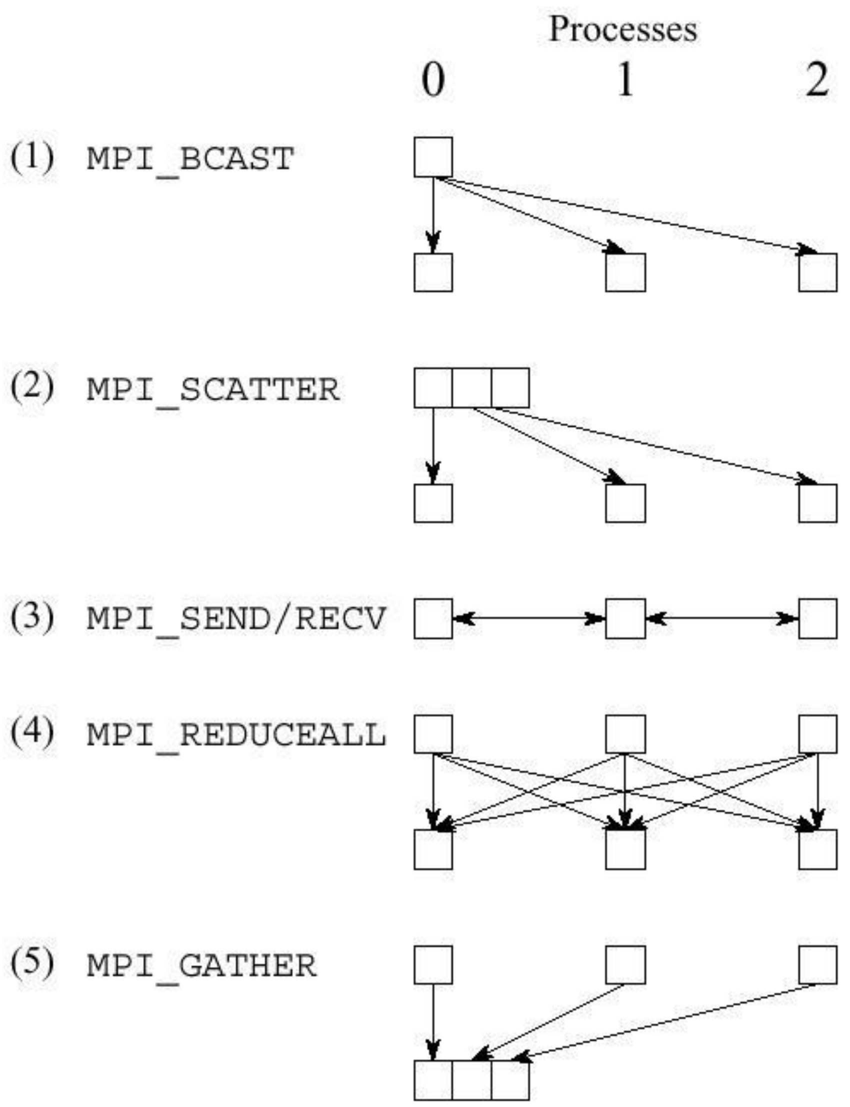
Expressing SOR in MPI

Use a ring topology

Each processor exchanges rows with left/right neighbor

Use REDUCE_ALL to determine if grid has changed less than epsilon during last iteration

Figure 8.5 from Foster's book



Semantics of collective operations

- Blocking operations:
  - It’s safe to reuse buffers after they return
- Standard mode only:
  - Completion of a call does not guarantee that other processes have completed the operation
- A collective operation may or may not have the effect of synchronizing all processes

Modularity

MPI programs use libraries

Library routines may send messages

These messages should not interfere with application messages

Tags do not solve this problem

Communicators

Communicator denotes group of processes (context)

MPI_SEND and MPI_RECV specify a communicator

MPI_RECV can only receive messages sent to same communicator

Library routines should use separate communicators, passed as parameter

Discussion

Library-based:

- No language modifications
- No compiler

Syntax is awkward

Message receipt based on identity of sender and operation tag, but not on contents of message

Needs separate mechanism for organizing name space

No type checking of messages

Syntax

SR:

```
call slave.coordinates(2.4, 5.67);
in coordinates (x, y);
```

MPI:

```
#define COORDINATES_TAG 1
#define SLAVE_ID 15
float buf[2];
buf[0] = 2.4; buf[1] = 5.67;
MPI_SEND (buf, 2, MPI_FLOAT, SLAVE_ID, COORDINATES_TAG,
                    MPI_COMM_WORLD);
MPI_RECV (buf, 2, MPI_FLOAT, MPI_ANY_SOURCE, COORDINATES_TAG,
MPI_COMM_WORLD, &status);
```