

CSL 860: Modern Parallel Computation

MPI: MESSAGE PASSING INTERFACE

Message Passing Model

- Process (program counter, address space)
 - Multiple threads (pc, stacks)
- MPI is for inter-process communication
 - Process creation
 - Data communication
 - Synchronization
- Allows
 - Synchronous communication
 - Asynchronous communication
 - Shared memory like ...

MPI Overview

- MPI, by itself is a library specification
 - You need a library that implements it
- Performs message passing and more
 - High-level constructs
 - broadcast, reduce, scatter/gather message
 - Packaging, buffering etc. automatically handled
 - Also
 - Starting and ending Tasks
 - Remotely
 - Task identification
- Portable
 - Hides architecture details

Running MPI Programs

- Compile:
`mpic++ -O -o exec code.cpp`
 - Or, `mpicc ...`
 - script to compile and link
 - Automatically add flags
- Run:
 - `mpirun -host host1,host2 exec args`
 - Or, may use `hostfile`
- Exists in:
 - `~subodh/graphicsHome/bin`
 - Libraries in `~subodh/graphicsHome/lib`

Remote Execution

- Must allow remote shell command execution
 - Using ssh
 - Without password
- Set up public-private key pair
 - Store in subdirectory `.ssh` in you home directory
- Use **ssh-keygen** to create the pair
 - Leaves public key in `id_rsa.pub`
- Put in file **`.ssh/authorized_keys`**
- Test: **ssh <remotehostname> ls**
 - Should list home directory

Process Organization

- Context
 - “communication universe”
 - Messages across context have no ‘interference’
- Groups
 - collection of processes
 - Creates hierarchy
- Communicator
 - Groups of processes that share a context
 - Notion of inter-communicator
 - Default: MPI_COMM_WORLD
- Rank
 - In the group associated with a communicator

MPI Basics

- Communicator
 - Collection of processes
 - Determines scope to which messages are relative
 - identity of process (rank) is relative to communicator
 - scope of global communications (broadcast, etc.)

- Query:

```
MPI_Comm_size (MPI_COMM_WORLD, &p) ;
```

```
MPI_Comm_rank (MPI_COMM_WORLD, &id) ;
```

Starting and Ending

```
MPI_Init (&argc, &argv) ;
```

– Needed before any other MPI call

```
MPI_Finalize () ;
```

– Required

Send/Receive

```
int MPI_Send(void* buf, int count,  
             MPI_Datatype datatype, int dest,  
             int tag, MPI_Comm comm)
```

```
void MPI::Comm::Send(const void* buf,  
                    int count, const MPI::Datatype&  
                    datatype, int dest, int tag) const
```

```
int MPI_Recv(void* buf, int count,  
            MPI_Datatype datatype, int  
            source, int tag, MPI_Comm comm,  
            MPI_Status *status)
```

Blocking calls

Send

- message contents block of memory
- count number of items in message
- message type MPI_TYPE of each item
- destination rank of recipient
- tag integer “message type”
- communicator

Receive

- message contents memory buffer to store received message
- count space in buffer
overflow error if too small
- message type type of each item
- source sender's rank (can be wild card)
- tag type (can be wild card)
- communicator
- status information about message received

Example

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"      /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);           // start MPI

    MPI_Comm_size(MPI_COMM_WORLD, &numProc); // Group size
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank); // get my rank
    doProcessing(myRank, numProc);

    MPI_Finalize();                   // stop MPI
}
```

Example

```
doProcessing(int myRank, int nProcs)
{
    /* I am ID myRank of nProcs */
    int numProc; /* number of processors */
    int source; /* rank of sender */
    int dest; /* rank of destination */
    int tag = 0; /* tag to distinguish messages */
    char mesg[MAXSIZE]; /* message (other types possible) */
    int count; /* number of items in message */
    MPI_Status status; /* status of message received */
}
```

Example

```
if (myRank != 0) { // all others send to 0
    // create message
    sprintf(message, "Hello from %d", myRank);
    dest = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR,
             dest, tag, MPI_COMM_WORLD);
}
else{ // P0 receives from everyone else in order
    for(source = 1; source < numProc; source++){
        if(MPI_Recv(msg, MAXSIZE, MPI_CHAR, source, tag,
                   MPICOMM_WORLD, &status) == MPI_SUCCESS)
            printf("Received from %d: %s\n", source, mess);
        else
            printf("Receive from %d failed\n", source);
    }
}
}
```

Send, Receive = “Synchronization”

- Fully Synchronized (Rendezvous)
 - Send and Receive complete simultaneously
 - whichever code reaches the Send/Receive first waits
 - provides synchronization point (up to network delays)
- Asynchronous
 - Sending process may proceed immediately
 - does not need to wait until message is copied to buffer
 - must check for completion before using message memory
 - Receiving process may proceed immediately
 - will not have message to use until it is received
 - must check for completion before using message

MPI Send and Receive

- **MPI_Send/MPI_Recv** is blocking
 - **MPI_Recv** blocks until message is received
 - **MPI_Send** may be synchronous or buffered
- Standard mode:
 - implementation dependent
 - Buffering improves performance, but requires sufficient resources
- Buffered mode
 - If no receive posted, system must buffer
 - User specified buffer size
- Synchronous mode
 - Will complete only if receive operation has accepted
 - send can be started whether or not a matching receive was posted.
- Ready mode
 - Send may start only if receive has been posted
 - Buffer may be re-used
 - Like standard, but helps performance

Function Names for Different Modes

- MPI_Send
- MPI_Bsend
- MPI_Ssend
- MPI_Rsend
- Only one MPI_Recv

Message Semantics

- In order
 - Multi-threaded applications need to be careful about order
- Progress
 - For a matching send/Recv pair, at least one of these two operations will complete
- Fairness not guaranteed
 - A Send or a Recv may starve because all matches are satisfied by others
- Resource limitations
 - Can lead to deadlocks
- Synchronous sends rely the least on resources
 - May be used as a debugging tool

Asynchronous Send and Receive

- **MPI_Isend()** / **MPI_Irecv()**
 - Non-blocking.: Control returns after setup
 - Blocking and non-blocking Send/Recv match
 - Still lower Send overhead if Recv has been posted
- All four modes are applicable
 - Limited impact for buffered and ready modes
- Syntax is the similar to Send and Recv
 - MPI_Request* parameter is added to Isend and replaces the MPI_Status* for receive.

No blocking Send/Receive

```
int MPI_Isend(void* buf, int count,  
             MPI_Datatype datatype, int dest,  
             int tag, MPI_Comm comm,  
             MPI_Request *request)
```

```
int MPI_Irecv(void* buf, int count,  
             MPI_Datatype datatype, int  
             source, int tag, MPI_Comm comm,  
             MPI_Request *request)
```

Non-blocking calls

Detecting Completion

- **MPI_Wait(&request, &status)**
 - **status** returns status similar to **Recv**
 - Blocks for send until safe to reuse buffer
 - Means message was copied out, or Recv was started
 - Blocks for receive until message is in the buffer
 - Call to Send may not have returned yet
 - Request is de-allocated
- **MPI_Test(&request, &flag, &status)**
 - does not block
 - flag indicates whether operation is complete
 - Poll
- **MPI_Request_get_status(&request, &flag, &status)**
 - This variant does not de-allocate request
- **MPI_Request_free(&request)**
 - Free the request

Non-blocking Batch Communication

- Ordering is by the initiating call
- There is provision for
**MPI_Waitany(count, requestsarray,
&whichReady, &status)**
 - If no active request:
 - whichReady = MPI_UNDEFINED, and empty status returned
- Also:
 - **MPI_Waitall, MPI_Testall**
 - **MPI_Waitsome, MPI_Testsome**

Receiver Message Peek

- **MPI_Probe(source, tag, comm, &flag, &status)**
- **MPI_Iprobe(source, tag, comm, &flag, &status)**
 - Check information about incoming messages without actually receiving them
 - Eg., useful to know message size
 - Next (matching) Recv will receive it
- **MPI_Cancel(&request)**
 - Request cancellation of a non-blocking request (no de-allocation)
 - Itself non-blocking: marks for cancellation and returns
 - Must still complete communication (or deallocate request) with **MPI_Wait/MPI_Test/MPI_Request_free**
- The operation that ‘completes’ the request returns status
 - One can test with **MPI_Test_Cancelled(&status, &flag)**

Persistent Send/Recv

```
MPI_Send_init(buf, count, datatype,  
dest, tag, comm, &request);
```

```
MPI_Start(&request);
```

- MPI_Start is non-blocking
 - blocking versions do not exist
- There is also **MPI_Start_all**
 - And **MPI_Recv_init**
 - And **MPI_Bsend_init** etc.
- Reduces Process interaction with the Communication system

Send and Recv

```
MPI_Sendrecv(sendbuf, sendcount,  
sendDataType, dest, sendtag,  
recvbuf, recvcount, recvtype,  
source, recvtag, comm, &status)
```

- Does both
- Semantics:
 - Fork, Send and Recv, Join
- Non-blocking
 - Blocking variant: **MPI_Sendrecv_replace**

Review Basics

- Send - Recv is point-to-point
 - Can Recv from any source using `MPI_ANY_SOURCE`
- Buffer in Recv must contain enough space for message
 - Count parameter is the capacity of buffer
 - Can query the actual count received

```
int cnt;
MPI_Get_count(&status, MPI_CHAR, &cnt);
```
- Count parameter in Send determines number
- type parameter determines exact number of bytes
- Integer tag to distinguish message streams
 - Can Recv any stream using `MPI_ANY_TAG`

MPI Data types

- MPI_CHAR signed char
- MPI_SHORT signed short int
- MPI_INT signed int
- MPI_LONG signed long int
- MPI_LONG_LONG_INT signed long long int
- MPI_LONG_LONG signed long long int
- MPI_SIGNED_CHAR signed char
- MPI_UNSIGNED_CHAR unsigned char
- MPI_UNSIGNED_SHORT unsigned short int
- MPI_UNSIGNED unsigned int
- MPI_UNSIGNED_LONG unsigned long int
- MPI_UNSIGNED_LONG_LONG unsigned long long int
- MPI_FLOAT float
- MPI_DOUBLE double
- MPI_LONG_DOUBLE long double
- MPI_WCHAR wchar_t
- MPI_BYTE
- MPI_PACKED

Derived Datatypes

- MPI does not understand *struct* etc.
 - Too system architecture dependent

```
MPI_Datatype newtype;  
MPI_Type_contiguous(count,  
                    knowntype, &newtype)
```

- Typemap:
 - $(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})$
 - i^{th} entry is of type $type_i$ and starts at byte base + $disp_i$

Blocks

```
MPI_Type_vector(blockcount,  
blocklength, blockstride,  
knowntype, &newtype);
```

- replication
- of a datatype into locations that consist of equally spaced blocks. Each block is
- obtained by concatenating the same number of copies of the old datatype.

```
MPI_Type_create_hvector(blk_count,  
blk_length, bytestride,  
knowntype, &newtype);
```

Generalized Blocks

```
MPI_Type_indexed(count,  
array_of_blocklengths,  
array_of_strides, knowntype,  
&newtype);
```

- Replication into a sequence of blocks
- Blocks can contain different number of copies
- And may have different strides

Struct

```
MPI_Type_create_struct (count,  
array_of_blocklengths,  
array_of_bytedisplacements,  
array_of_knowntypes, &newtype)
```

- Example:
 - Supposr Type0 = {(double, 0), (char, 8)},
 - int B[] = {2, 1, 3}, D[] = {0, 16, 26};
 - MPI_Datatype T[] = {MPI_FLOAT, Type0, MPI_CHAR}.
- MPI_Type_create_struct(3, B, D, T, &newtype) returns
 - (float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27),
(char, 28)
- Other functions for structs distributed across processors

Data Type Functions

MPI_Type_size(datatype, &size)

- Total size in bytes

MPI_Type_commit(&datatype)

- A datatype object must be committed before communication

```
MPI_Datatype type0;
```

```
MPI_Type_contiguous(1, MPI_CHAR, &type0);
```

```
int size;
```

```
MPI_Type_size(type0, &size);
```

```
MPI_Type_commit(&type0);
```

```
MPI_Send(buf, nItems, type0, dest, tag, MPI_COMM_WORLD);
```

Derived Datatype Example Usage

```
struct Partstruct
{
    int class; /* particle class */
    double d[6]; /* particle coordinates */
    char b[7]; /* some additional information */
};
```

```
void useStruct() {

    struct Partstruct particle[1000];
    int i, dest, rank;
    MPI_Comm comm;
```

```
/* build datatype describing structure */
MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int blocklen[3] = {1, 6, 7};
MPI_Aint disp[3];
MPI_Aint base;

/* compute displacements of structure components */
MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i <3; i++) disp[i] -= base;

MPI_Type_struct( 3, blocklen, disp, type, &Particletype);
MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);
}
```

Data packaging

- User calls pack/unpack into a byte array
- Backward compatibility
- Needed to combine irregular, non-contiguous data into single message
- Also easier to break message into parts
- Supports development of newer API layer

MPI_Pack()

```
MPI_Pack(dataPtr, count, type,  
         packedbuffer, buffersize,  
         &bufferposition, communicator);
```

dataPtr	pointer to data that needs to be packed
count	number of items to pack
type	type of items to pack
buffer	buffer to pack into
size	size of buffer (in bytes) – must be large enough
position	starting position in buffer (in bytes), updated to the end of the packed area

MPI_Unpack()

- Can first check packed output size:

```
MPI_Pack_size(incount, datatype,  
comm, &size)
```

```
MPI_Unpack(packedbuffer, size,  
&position, dataPtr, count, type,  
communicator);
```

count actual number of items to unpack
 must have enough space

Packing Example

```
int position, i, j, a[2];
char buf[1000];
....
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
    /* SENDER CODE */
    position = 0;
    MPI_Pack(&i, 1, MPI_INT, buf, 1000, &position, MPI_COMM_WORLD);
    MPI_Pack(&j, 1, MPI_INT, buf, 1000, &position, MPI_COMM_WORLD);
    MPI_Send( buf, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else /* RECEIVER CODE */
    MPI_Recv(a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD)
```

Collective Communication

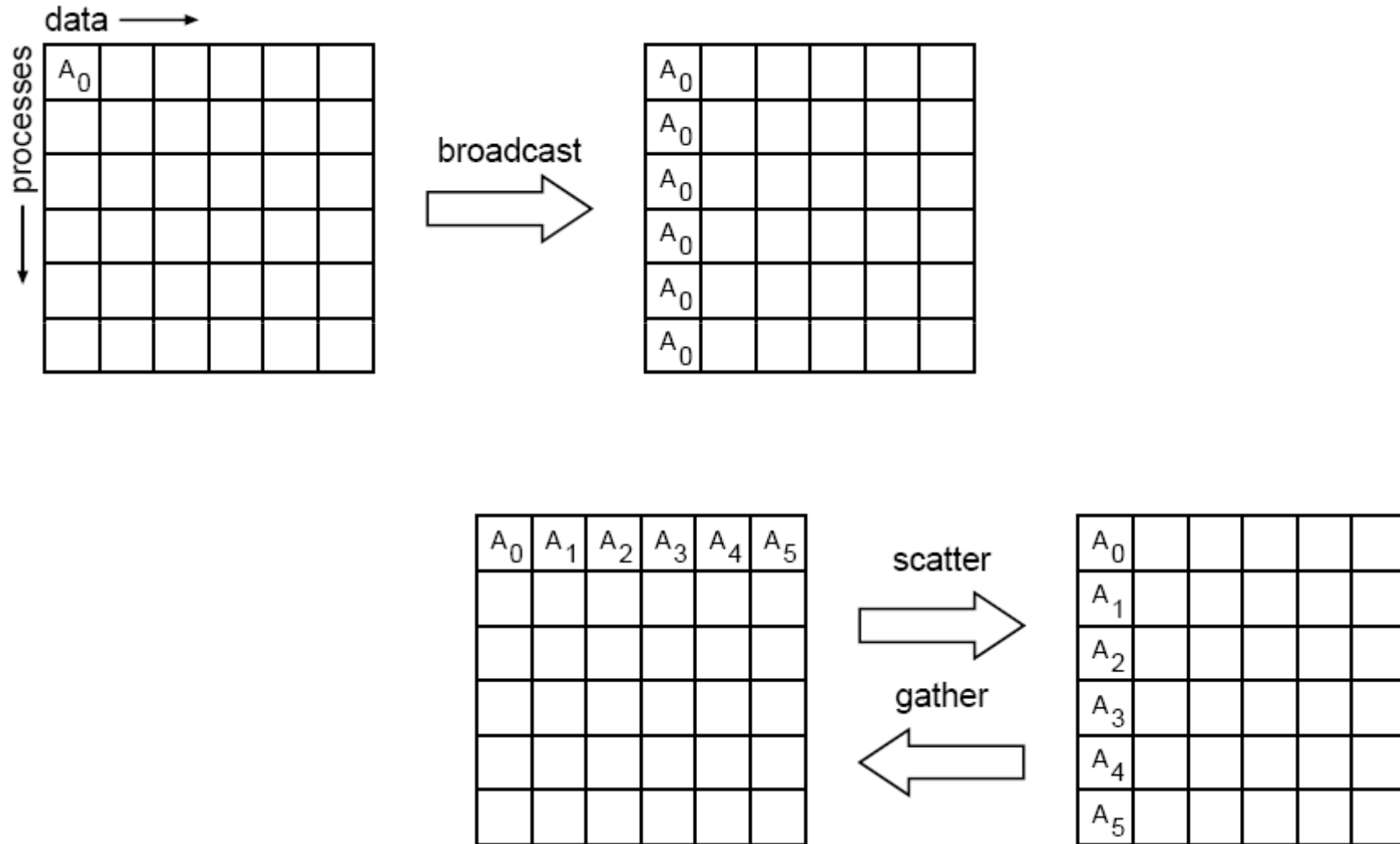
- MPI_Barrier
 - Barrier synchronization across all members of a group (Section 5.3).
- MPI_Bcast
 - Broadcast from one member to all members of a group (Section 5.4).
- MPI_Scatter, MPI_Gather, MPI_Allgather
 - Gather data from all members of a group to one
- MPI_Alltoall
 - complete exchange or all-to-all
- MPI_Allreduce, MPI_Reduce
 - Reduction operations
- MPI_Reduce_Scatter
 - Combined reduction and scatter operation
- MPI_Scan, MPI_Exscan
 - Prefix

Barrier

- Synchronization of the calling processes
 - the call blocks until all of the processes have placed the call

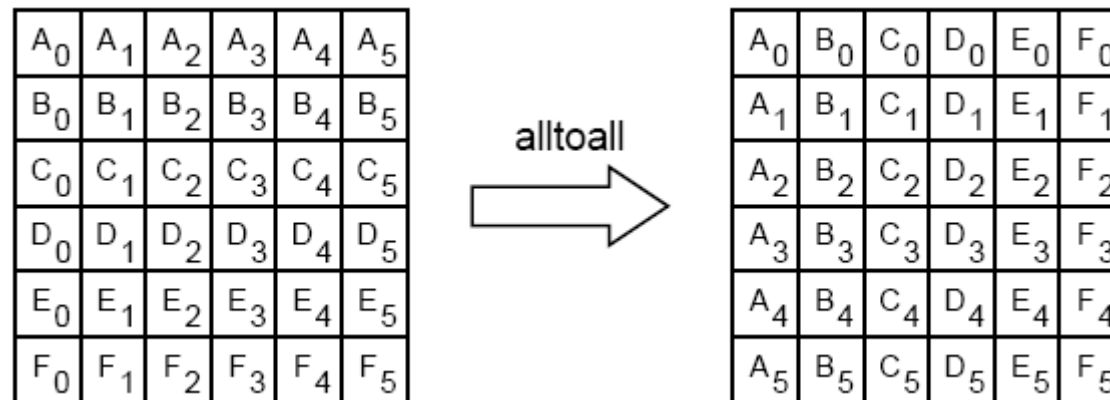
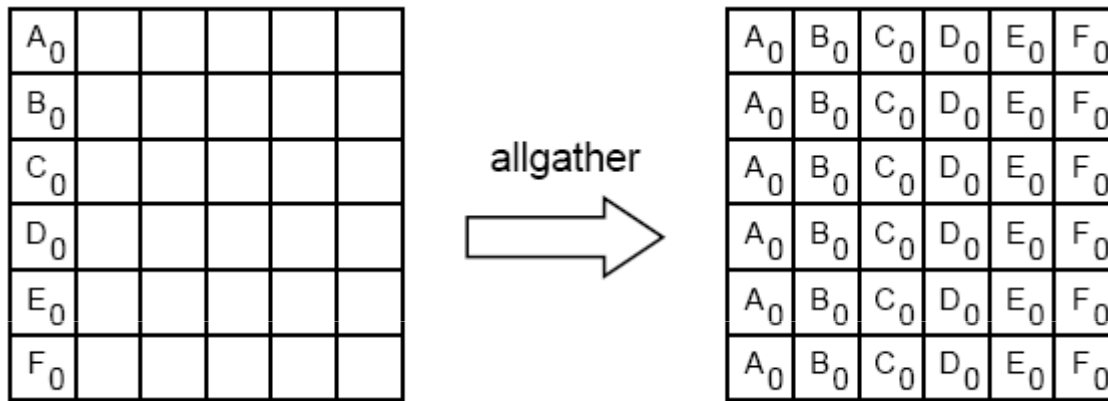
```
MPI_Barrier(comm) ;
```

Collective Communication



Pictures from MPI-2 document

Collective Communication



Pictures from MPI-2 document

Broadcast

- Broadcast: one sender, many receivers
- Includes all processes in communicator
 - all processes must make a call to `MPI_Bcast`
 - Must agree on sender
- Broadcast does not ensure global synchronization
 - Some implementations may incur synchronization
 - Call may return before other have received, e.g.
 - Different from `MPI_Barrier(communicator)`

Broadcast

```
MPI_Bcast (mesg, count, MPI_INT,  
            root, comm);
```

mesg pointer to message buffer

count number of items sent

MPI_INT type of item sent

root sending processor

- **Again:** All participants must call
- **count** and **type** should be the same on all members
- Can broadcast on inter-communicators also

MPI_Gather

```
MPI_Gather(sendbuf, sendcount,  
sendtype, recvbuf, recvcount,  
recvtype, root, comm);
```

- Same as non-roots doing:
 - MPI_Send(sendbuf, sendcount, sendtype, root, ...),
- and the root had n iterations of:
 - MPI_Recv(recvbuf + i · recvcount .extent(recvtype),
recvcount, recvtype, i, ...),
- MPI_Gatherv allows different size data to be gathered

Many to Many Communications

- **MPI_Allreduce**
 - Syntax like reduce, except no root parameter
 - All nodes get result
- **MPI_Allgather**
 - Syntax like gather, except no root parameter
 - All nodes get resulting array

MPI_Reduce()

```
MPI_Reduce (dataArray, resultArray, count,  
            type, MPI_SUM, root, com);
```

dataIn	data sent from each processor
Result	stores result of combining operation
count	number of items in each of dataIn, result
MPI_SUM	combining operation, one of a predefined set
root	rank of processor receiving data

- Multiple elements can be reduced in one shot
- Illegal to alias input and output arrays

MPI_Reduce variants

- MPI_Reduce: result is sent out to the root
 - the operation is applied element-wise for each element of the input arrays on each processor
- MPI_Allreduce: result is sent out to everyone
- MPI_Reduce_scatter: functionality equivalent to a reduce followed by a scatter
- User defined operations

User-defined reduce operations

```
void rfunction(void *invec,  
              void *inoutvec, int *len, MPI_Datatype  
              *datatype);
```

```
MPI_Op op;
```

```
MPI_Op_create(rfunction, commute, &op);
```

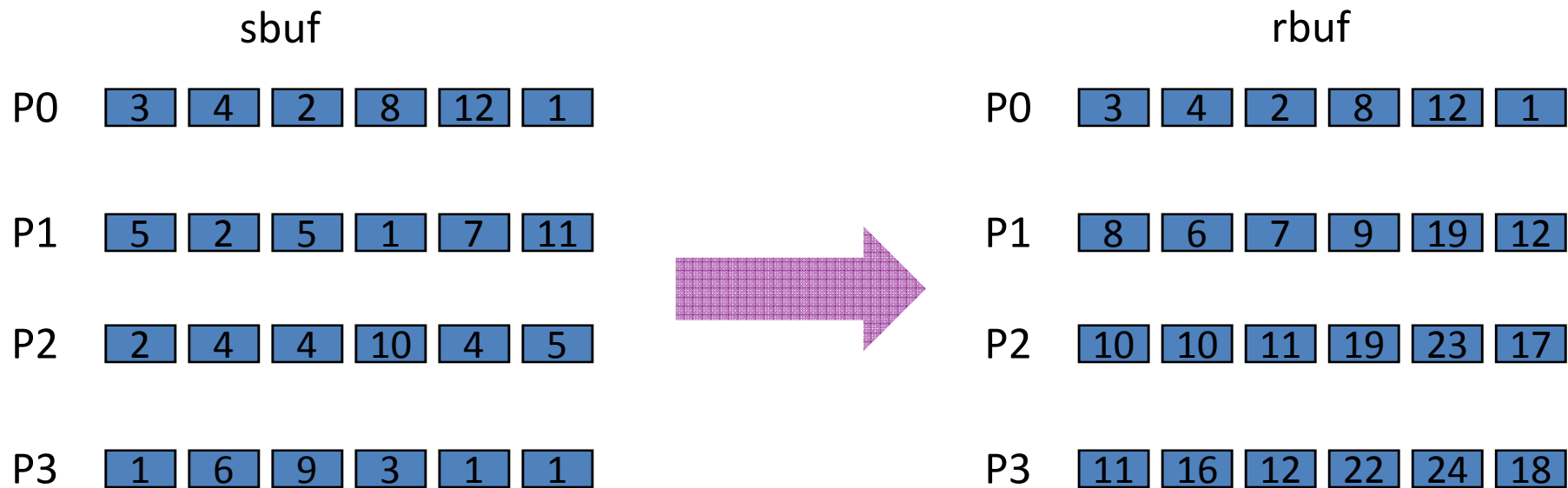
```
...
```

Later:

```
MPI_op_free(&op);
```

MPI_Scan: Prefix reduction

- process i receives data reduced on process 0 to i.



`MPI_Scan`(sbuf, rbuf, 6, MPI_INT, MPI_SUM, MPI_COMM_WORLD)

Process Start

```
MPI_Comm_Spawn(command, argv,  
maxprocs, info, root, comm,  
intercomm, array_of_errcodes)
```

- The children have their own
MPI_COMM_WORLD
- May not return until MPI_INIT has been called
in the children
- More efficient to start all processes at once

More MPI-2

- Remote Memory
 - put/get
 - weak synchronization
- Parallel I/O
- Refined notion of groups
- Socket-style communication:
 - open_port
 - Accept
 - connect