Introduction to Parallel Computing

Irene Moulitsas

Programming using the Message-Passing Paradigm

MPI Background

- MPI : Message Passing Interface
- Began in Supercomputing ’92
  - Vendors
    - IBM, Intel, Cray
  - Library writers
    - PVM
  - Application specialists
    - National Laboratories, Universities

Why MPI ?

- One of the oldest libraries
- Wide-spread adoption. Portable.
- Minimal requirements on the underlying hardware
- Explicit parallelization
  - Intellectually demanding
  - Achieves high performance
  - Scales to large number of processors

MPI Programming Structure

- Asynchronous
  - Hard to reason
  - Non-deterministic behavior
- Loosely synchronous
  - Synchronize to perform interactions
  - Easier to reason
- SPMD
  - **S**ingle **P**rogram **M**ultiple **D**ata

MPI Features

- Communicator Information
- Point to Point communication
- Collective Communication
- Topology Support
- Error Handling

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

Six Golden MPI Functions

- MPI is 125 functions
- MPI has 6 most used functions

Table 6.1 The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of the calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

MPI Functions: Initialization

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- Must be called by all processes
- MPI_SUCCESS
  - "mpi.h"

MPI Functions: Communicator

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- MPI_Comm
  - MPI_COMM_WORLD

Hello World !

```
#include <mpi.h>

main(int argc, char *argv[])
{
  int npes, myrank;
  MPI_Comm_size(MPI_COMM_WORLD, &npes);

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf("From process %d out of %d, Hello World!\n",
         myrank, npes);
  MPI_Finalize();
 }
```

Hello World ! (correct)

```
#include <mpi.h>

main(int argc, char *argv[])
{
   int npes, myrank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf("From process %d out of %d, Hello World!\n",
          myrank, npes);
  MPI_Finalize();
}
```

MPI Functions: Send, Recv

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

- source
  - MPI_ANY_SOURCE
- MPI_Status
  - MPI_SOURCE
  - MPI_TAG
  - MPI_ERROR

MPI Functions: Datatypes

Table 6.2 Correspondence between the datatypes supported by MPI and those supported by C.

MPI Datatype	C Datatype
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	
`MPI_PACKED`	

Send/Receive Examples

```
P0

a = 100;
send(&a, 1, 1);
a = 0;
```

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

Blocking Non-Buffered Communication

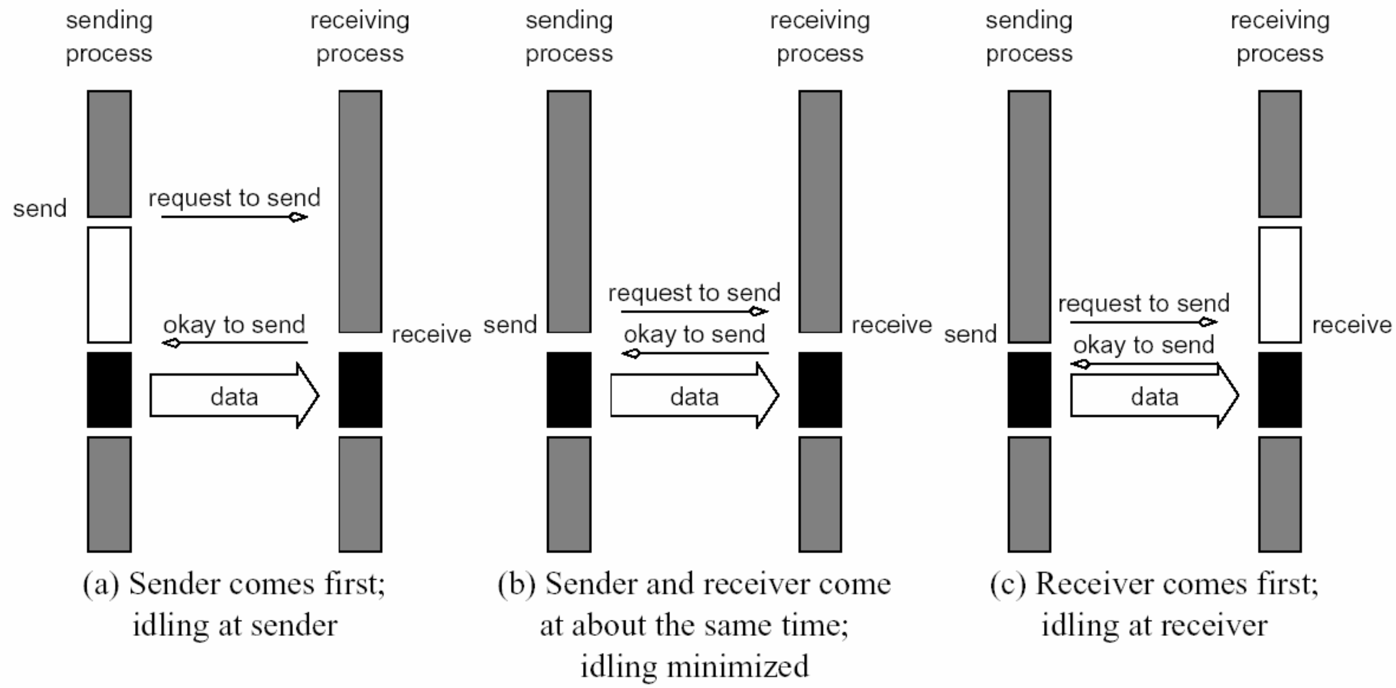


Figure 6.1 Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Send/Receive Examples

```
P0

a = 100;
send(&a, 1, 1);
a = 0;
```

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

```
P0

send(&a, 1, 1);
receive(&b, 1, 1);
```

```
P1

send(&a, 1, 0);
receive(&b, 1, 0);
```

Blocking Buffered Communication

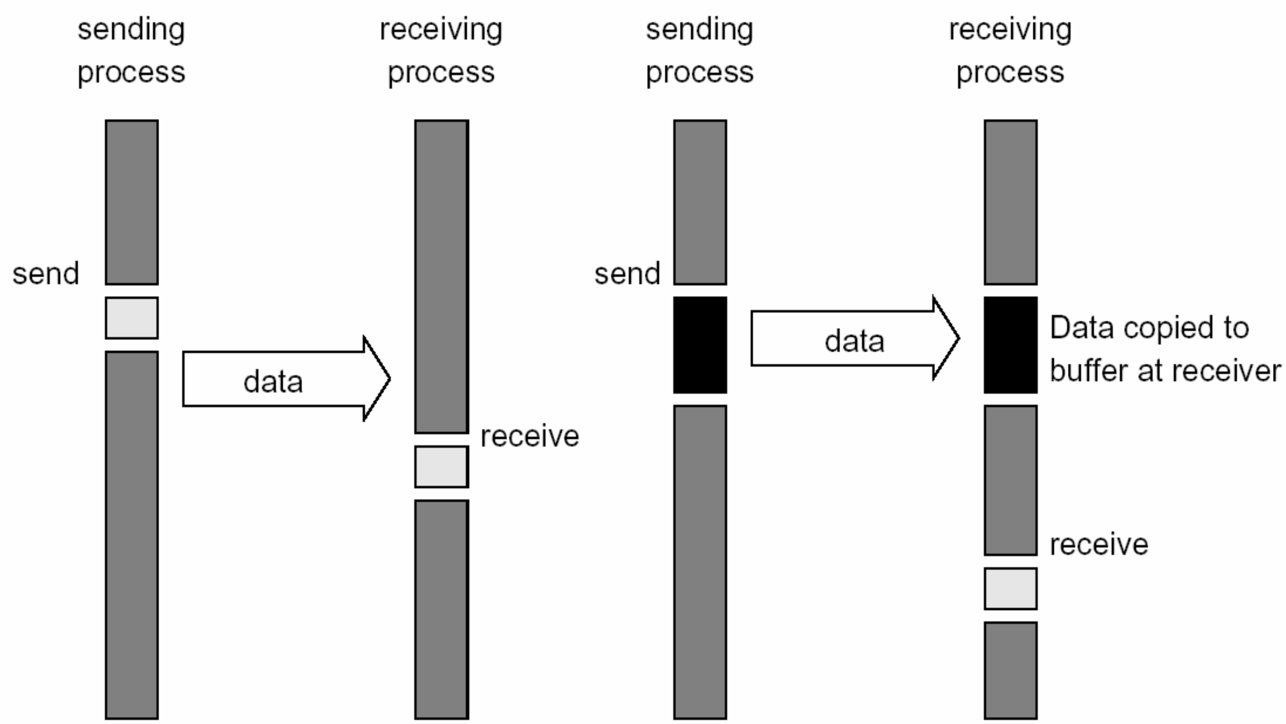


Figure 6.2 Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Send/Receive Examples

```
P0                                      P1

for (i = 0; i < 1000; i++) {            for (i = 0; i < 1000; i++) {
  produce_data(&a);                       receive(&a, 1, 0);
  send(&a, 1, 1);                         consume_data(&a);
}                                       }
```

```
P0                                      P1

receive(&a, 1, 1);                      receive(&a, 1, 0);
send(&b, 1, 1);                         send(&b, 1, 0);
```

MPI Functions: SendRecv

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

MPI Functions: ISend, IRecv

```
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
- MPI_Request

MPI Functions: Test, Wait

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- MPI_Test tests if operation finished.
- MPI_Wait blocks until operation is finished.

Non-Blocking Non-Buffered Communication

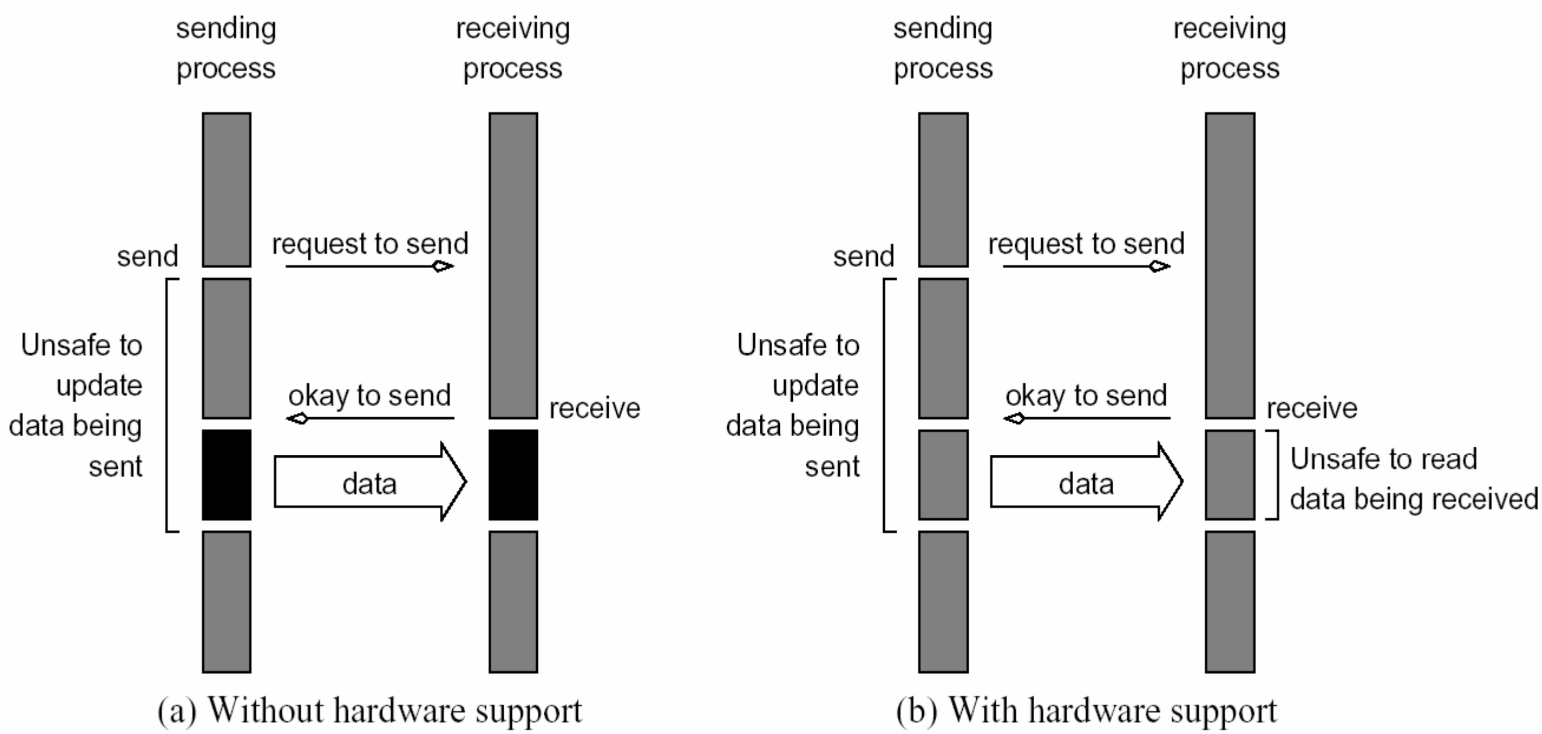


Figure 6.4 Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

Example

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...
```

Example

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
...
```

Example

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
             b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
             MPI_COMM_WORLD, &status);
...
```

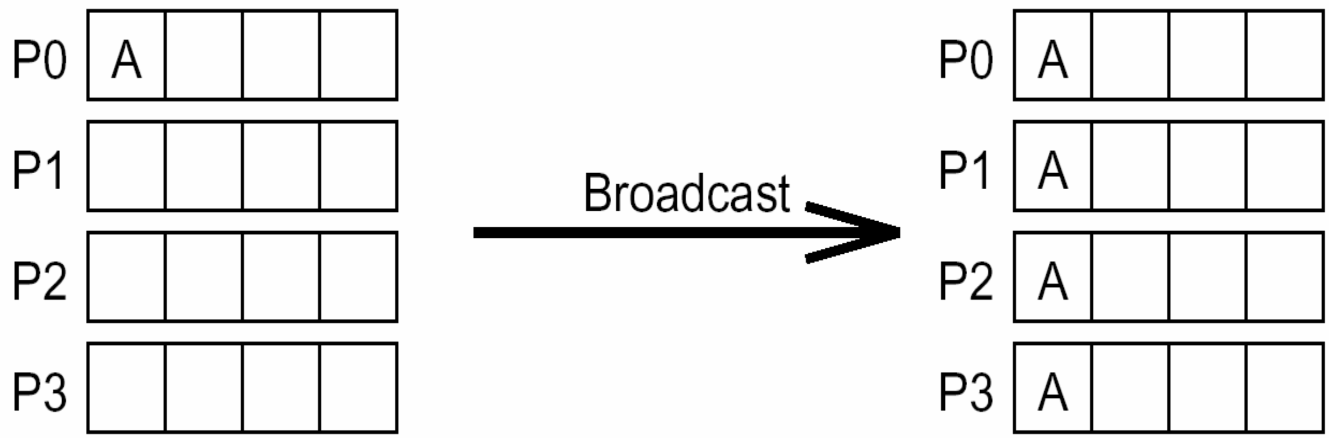
MPI Functions: Synchronization

```
int MPI_Barrier(MPI_Comm comm)
```

Collective Communications

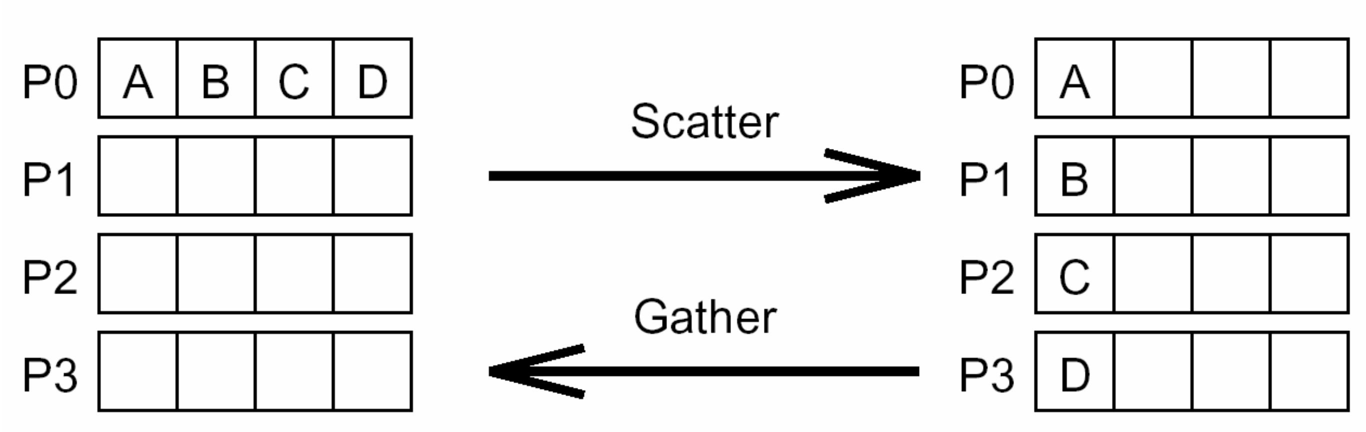
- One-to-All Broadcast
- All-to-One Reduction
- All-to-All Broadcast & Reduction
- All-Reduce & Prefix-Sum
- Scatter and Gather
- All-to-All Personalized

MPI Functions: Broadcast




```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
        int source, MPI_Comm comm)
```

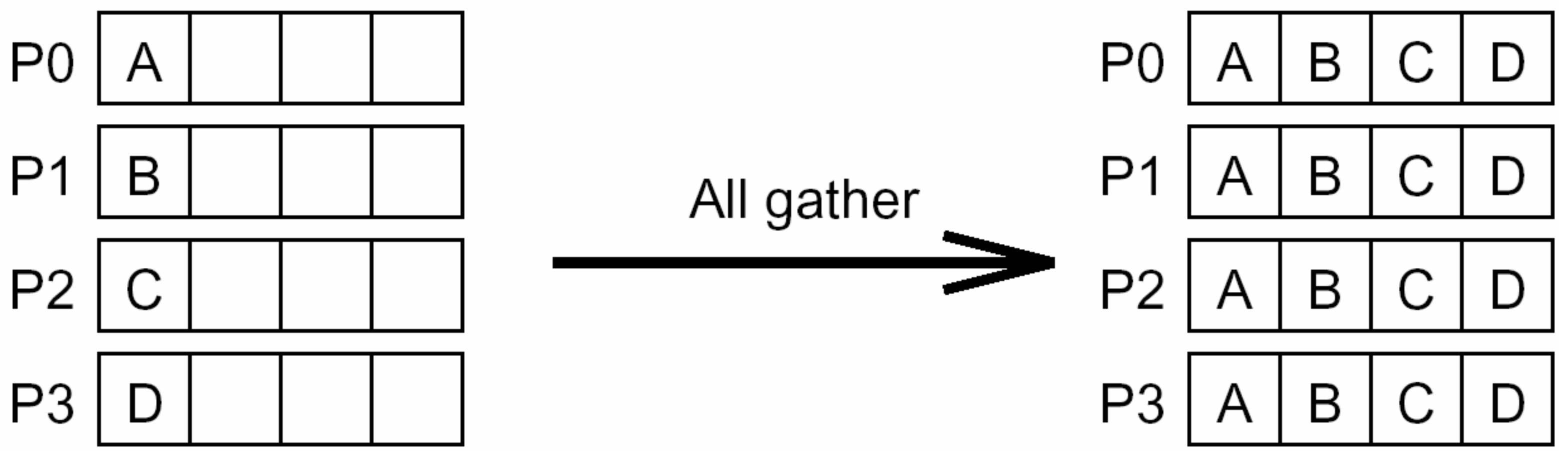
MPI Functions: Scatter & Gather




```
int MPI_Scatter(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int source, MPI_Comm comm)
```

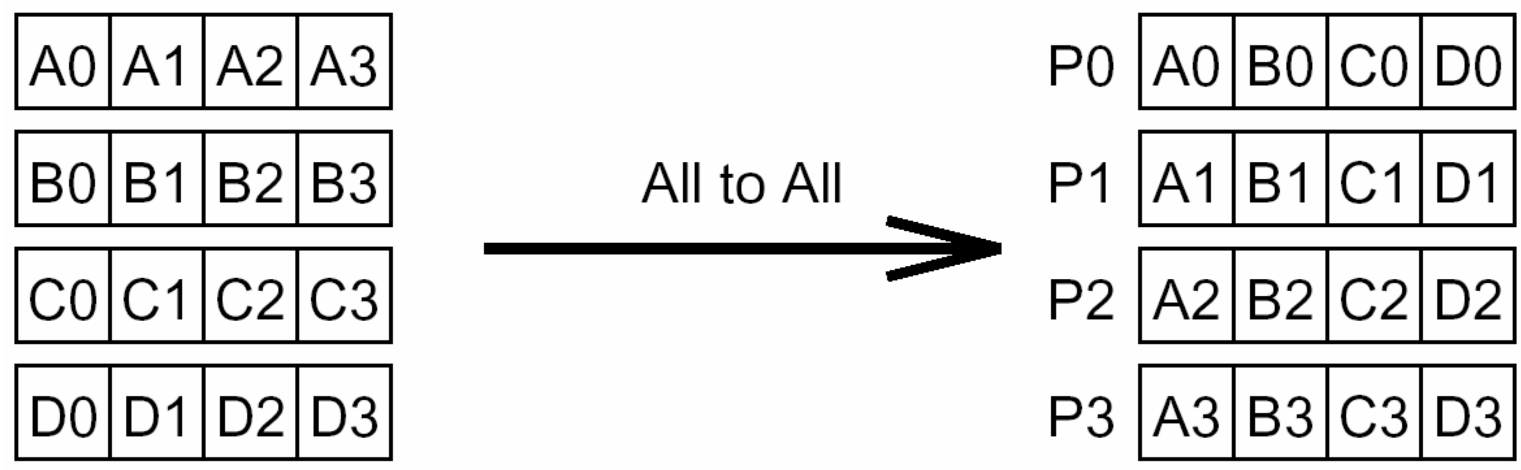
```
int MPI_Gather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int target, MPI_Comm comm)
```

MPI Functions: All Gather



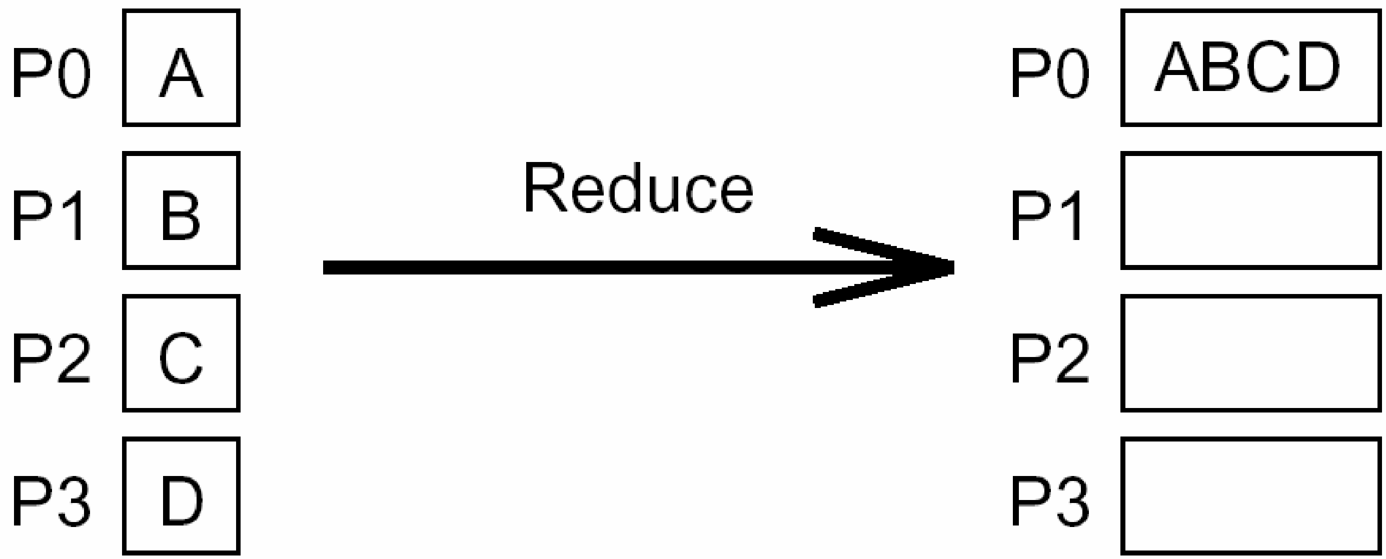

```
int MPI_Allgather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

MPI Functions: All-to-All Personalized




```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

MPI Functions: Reduction




```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int target,
        MPI_Comm comm)
```

MPI Functions: Operations

Table 6.3 Predefined reduction operations.

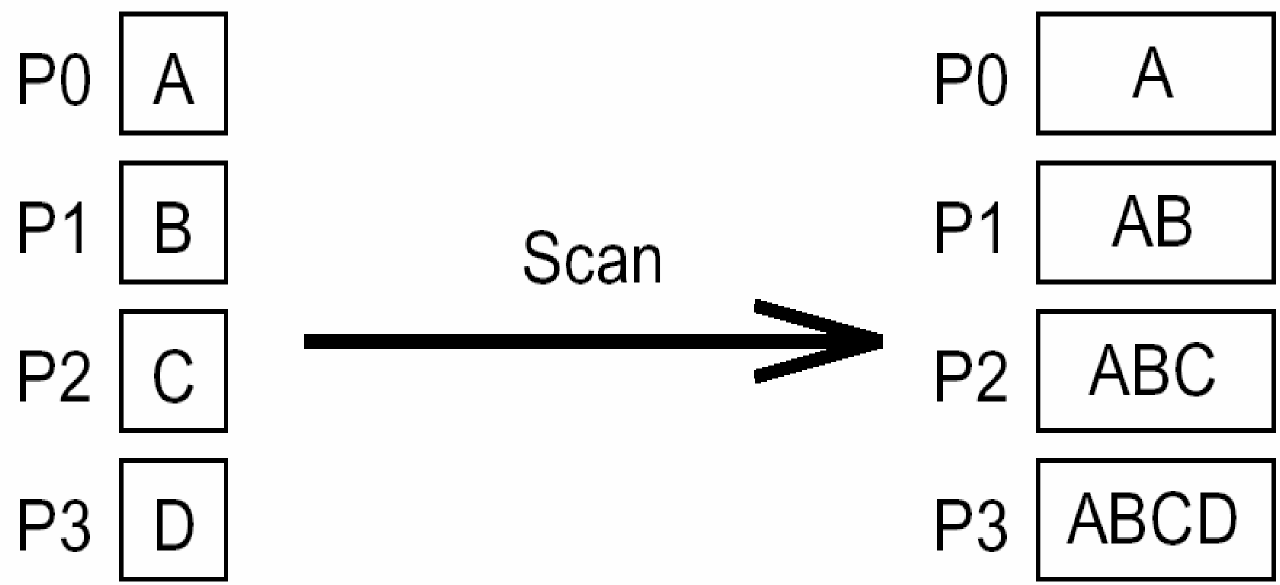
Operation	Meaning	Datatypes
`MPI_MAX`	Maximum	C integers and floating point
`MPI_MIN`	Minimum	C integers and floating point
`MPI_SUM`	Sum	C integers and floating point
`MPI_PROD`	Product	C integers and floating point
`MPI_LAND`	Logical AND	C integers
`MPI_BAND`	Bit-wise AND	C integers and byte
`MPI_LOR`	Logical OR	C integers
`MPI_BOR`	Bit-wise OR	C integers and byte
`MPI_LXOR`	Logical XOR	C integers
`MPI_BXOR`	Bit-wise XOR	C integers and byte
`MPI_MAXLOC`	max-min value-location	Data-pairs
`MPI_MINLOC`	min-min value-location	Data-pairs

MPI Functions: All-reduce

- Same as MPI_Reduce, but all processes receive the result of MPI_Op operation.

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

MPI Functions: Prefix Scan




```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

MPI Names

Table 4.2 MPI names of the various operations discussed in this chapter.

Operation	MPI Name
One-to-all broadcast	MPI_Bcast
All-to-one reduction	MPI_Reduce
All-to-all broadcast	MPI_Allgather
All-to-all reduction	MPI_Reduce_scatter
All-reduce	MPI_Allreduce
Gather	MPI_Gather
Scatter	MPI_Scatter
All-to-all personalized	MPI_Alltoall

MPI Functions: Topology

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
        int *periods, int reorder, MPI_Comm *comm_cart)
```

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims,
        int *coords)
```

Performance Evaluation

- Elapsed (wall-clock) time

```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf( "Elapsed time is %f\n", t2 - t1 );
```

Matrix/Vector Multiply

Program 6.4 Row-wise Matrix-Vector Multiplication

```
RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
                        MPI_Comm comm)
{
  int i, j;
  int nlocal;        /* Number of locally stored rows of A */
  double *fb;        /* Will point to a buffer that stores the entire vector b */
  int npes, myrank;
  MPI_Status status;

  /* Get information about the communicator */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Allocate the memory that will store the entire vector b */
  fb = (double *)malloc(n*sizeof(double));

  nlocal = n/npes;

  /* Gather the entire vector b on each processor using MPI's ALLGATHER operation */
  MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
        comm);

  /* Perform the matrix-vector multiplication involving the locally stored submatrix */
  for (i=0; i<nlocal; i++) {
    x[i] = 0.0;
    for (j=0; j<n; j++)
      x[i] += a[i*n+j]*fb[j];
  }

  free(fb);
}
```