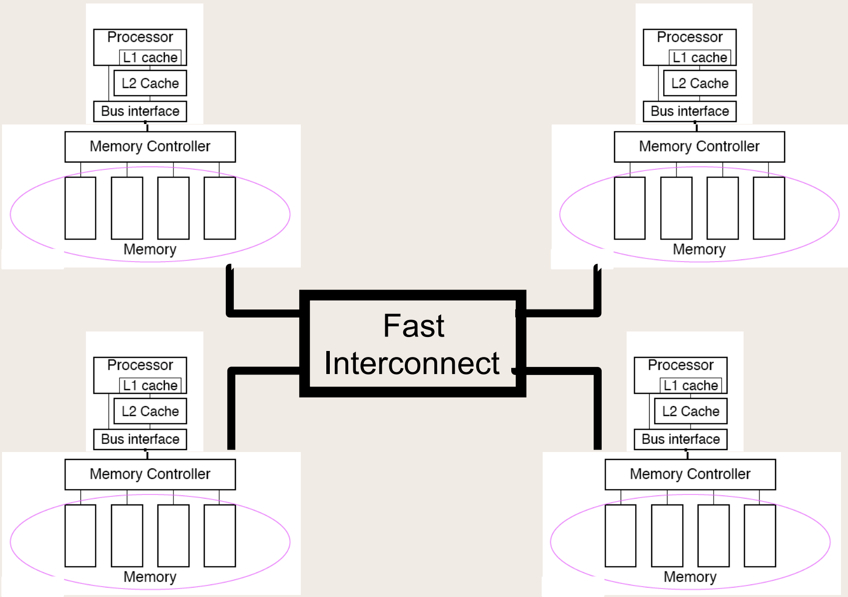
CS 6230: High-Performance Computing and Parallelization – Introduction to MPI

Dr. Mike Kirby

School of Computing and
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT, USA

Distributed Computing



(BlueGene/L - Image courtesy of IBM / LLNL)

MPI is the de facto standard for programming distributed processes.

A large API with over 300 functions exists and is widely supported.

Several popular and robust (free) implementations: MPICH and OpenMPI

How Widely Used Is MPI?

	Language/Prog Models----------										RT---		Dev--		I/O------------------						Math--															Sys------------------					
	Fortran77	Fortran90	Fortran95	C/C++	CAF	OpenMP	MPI	MPI2	Global Arrays	Python	ARMCI	BLACS	GNU Make	CCA-Tools	HDF5	PHDF5	MPI-IO	netCDF	pnetCDF	XML	BLAS	Cray Scilib	FFTPACK	FFTW	LAPACK	MUMPS	PBLAS	PEIGS	Parmetis	PARPACK	PetSC	Scalapack	SGI SCSL	SPRNG	ZOLTAN	ctime	tar	CP	Mkdir	Rm	Other
LSMS	1	1		1				1					1		1					1	1				1											1					
QMC		1					1														1				1									1							
CHIMERA		1					1								1				1						1																
POP/CICE		1			x	x	1											1																							
SD3		1					1																														1	1	1	x	
GTC		1		1		x	1								1	1	1	1		1											1										1
MADNESS		1					1														1																				
AORSA	1	1										1						1					1				1					1									
LAMMPS				1			1																	1																	
FLASH		1		1			1			1			1		1				1																						
Milc/Chroma				1			1																																		1
PFLOTRAN		1					1														1										1										
QBOX				1			1					1								1	1			1	1							1									
CAM		1		1	x													1				1											1								x
CCSD		1					1										1				1																				
T3P				1			1											1								1			1						1						1
VASP		1					1														1											1									x
NEWTRNX		1		1						1			1	1	1										1					1											
NWChem	1			1					1		1		1								1		1					1				1									
OReTran			1				1														1				1																
CASINO		1					1														1																				1

Must have (1) Can use (x)

The success of MPI (Courtesy of Al Geist, EuroPVM / MPI 2007)

Why MPI is Complex: Collision of Features

- Send
- Receive
- Send / Receive
- Send / Receive / Replace
- Broadcast
- Barrier
- Reduce

- Rendezvous mode
- Blocking mode
- Non-blocking mode
- Reliance on system buffering
- User-attached buffering
- Restarts/Cancels of MPI Operations

- Non Wildcard receives
- Wildcard receives
- Tag matching
- Communication spaces

An MPI program is an interesting (and legal) combination of elements from these spaces

So What is MPI Anyway?

MPI is not a language. It is an API.

Application Programming Interface (API): An API defines the calling conventions and other information needed for one software module (typically an application program) to utilize the services provided by another software module.

MPI provides a collection of functions that allow inter-process communication through an MPI communications “layer”.

One compiles “with” MPI.

Programming and Compiling

C++/MPI Code From Practical 1

```
#include <iostream>
#include "mpi.h"

using namespace std;

int main(int argc, char ** argv){
  int mynode, totalnodes;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  cout << "I am process " << mynode << " out of " << totalnodes << endl;

  MPI_Finalize();

  return 0;
}
```

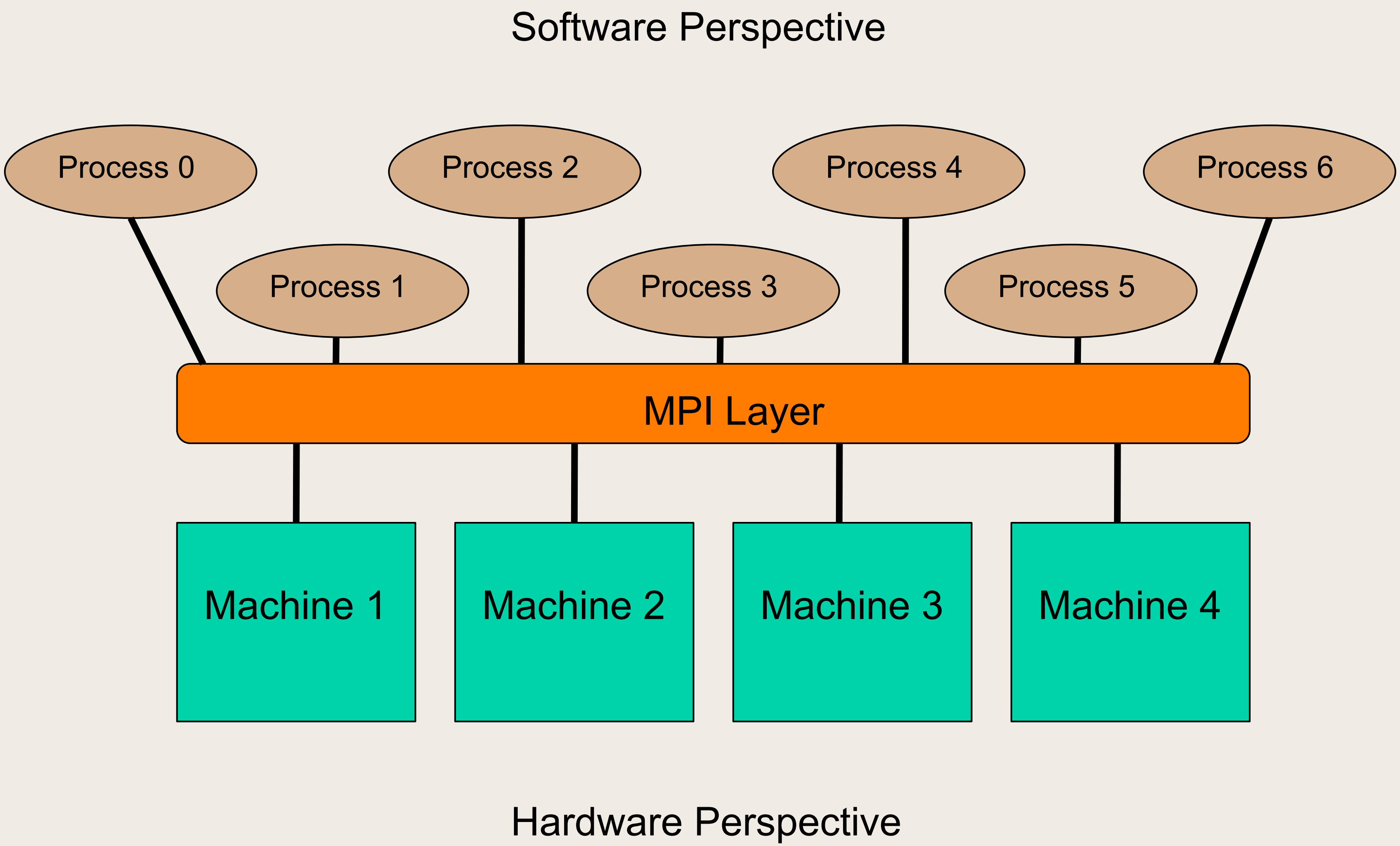
mpicc –o prac1 prac1.cpp

or

g++ -o prac1 -I <header path> -L <MPI library path> -lmpi prac1.cpp

This produces an executable prac1

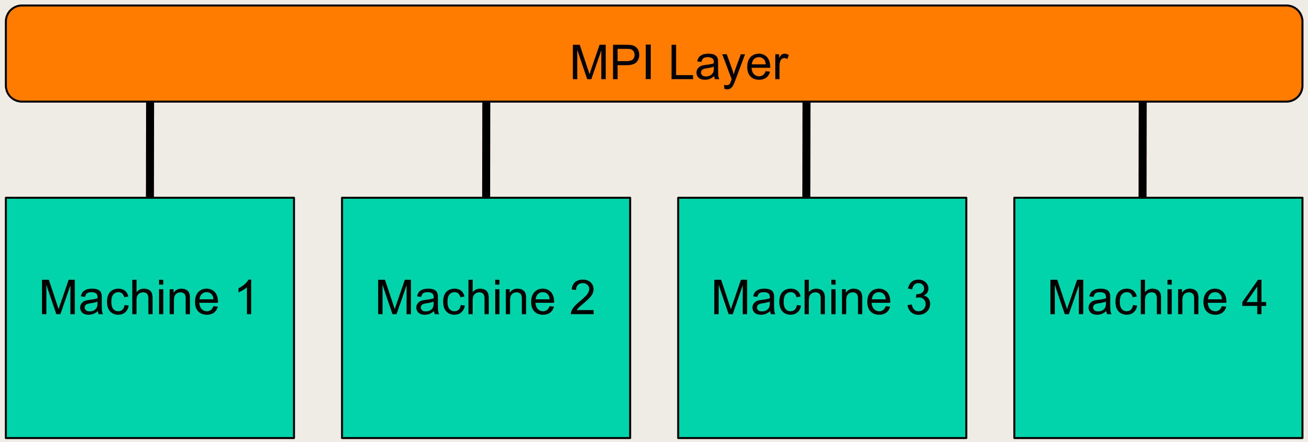
Conceptual View of MPI



MPI "Boot"

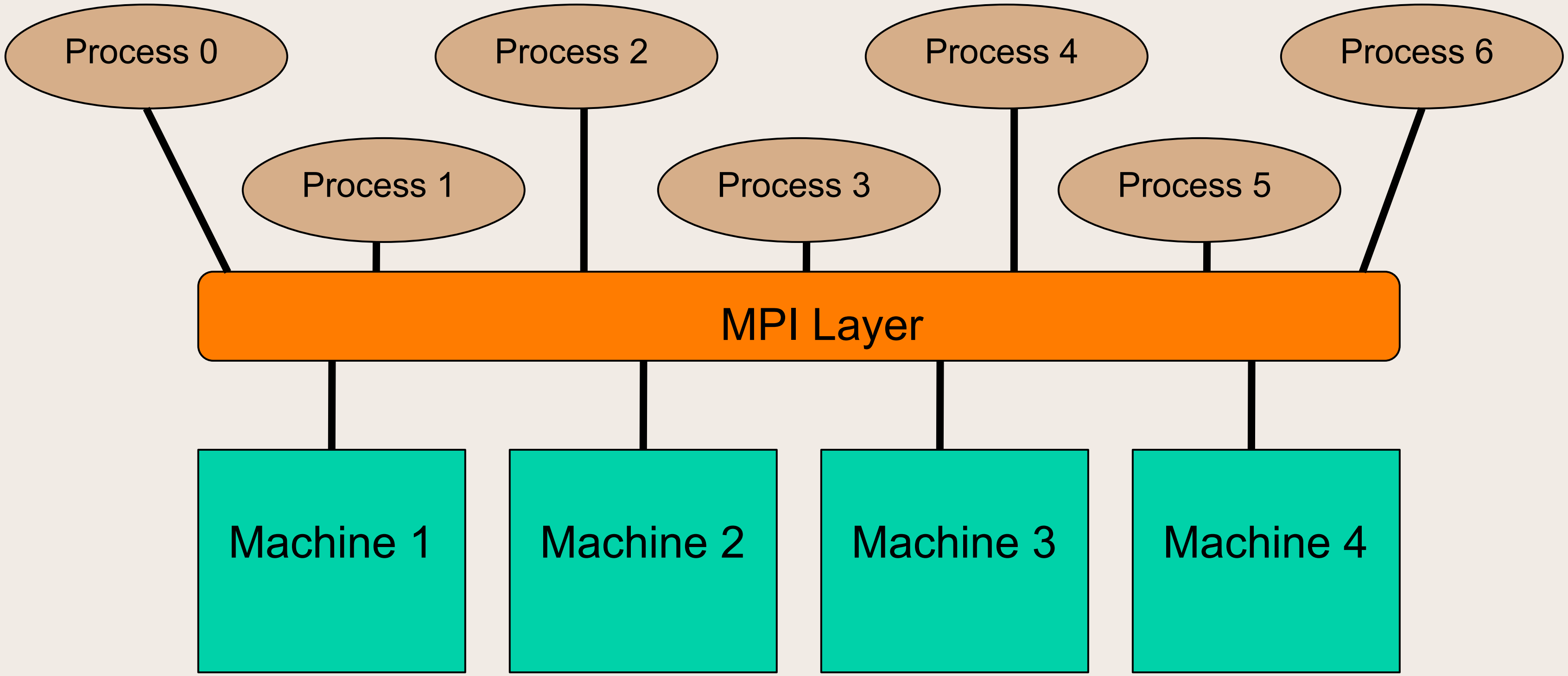
MPI "Boot" (called different things per implementation) starts a daemon per machine – sometimes called the MPI daemon.

This daemon waits for an MPI job to be started using mpirun.



Running MPI

mpirun –np 7 prac1



Hardware Perspective

Groups and Communicators

- A group defines the participants in the communication of a communicator. It is actually an ordered collection of processes, each with a rank.

- Message passing in MPI is via communicators, each of which specifies a set (group) of processes that participate in the communication.

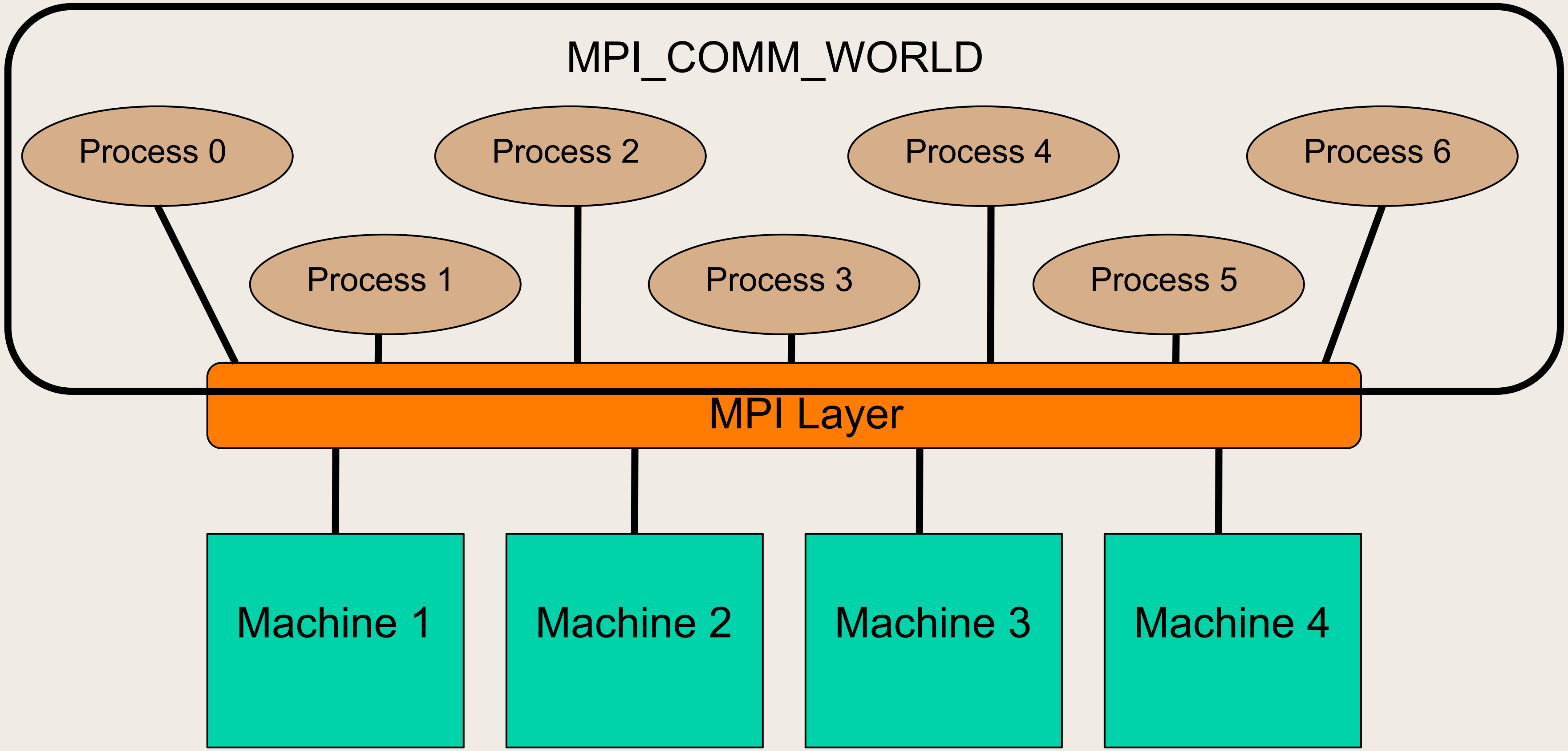
- Communicators can be created and destroyed dynamically by coordinating processes.

- Information about topology and other attributes of a communicator can be updated dynamically.

Groups and Communicators

- Group Functions start with MPI_Group_*
  - MPI_Group_rank
  - MPI_Group_size
  - MPI_Group_create
- Communicator Functions start with MPI_Comm_*
  - MPI_Comm_rank
  - MPI_Comm_size
  - MPI_Comm_compare
  - MPI_Comm_dup

Guaranteed Communicator



Predefined MPI Datatypes

MPI datatype	*C datatype*
MPI_CHAR	signed char
MPI_SHORT	singed short int
MPI_INT	signed int
MPI_LONG	singed 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	

Predefined MPI Operations

Operation Name	*Meaning*
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Predefined MPI Operations

Defined in header file

```
#include <iostream.h>
#include <mpi.h>

int main(int argc, char * argv[]){
  int mynode, totalnodes;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  cout << "Hello world from processor " << mynode << " of " << totalnodes << endl;

  MPI_Finalize();

  return 0;
}
```

Used as arguments in MPI function calls

Output Considerations

This is probably what you see as output on your screen:

```
Hello world from processor 0 of 4
Hello world from processor 3 of 4
Hello world from processor 2 of 4
Hello world from processor 1 of 4
```

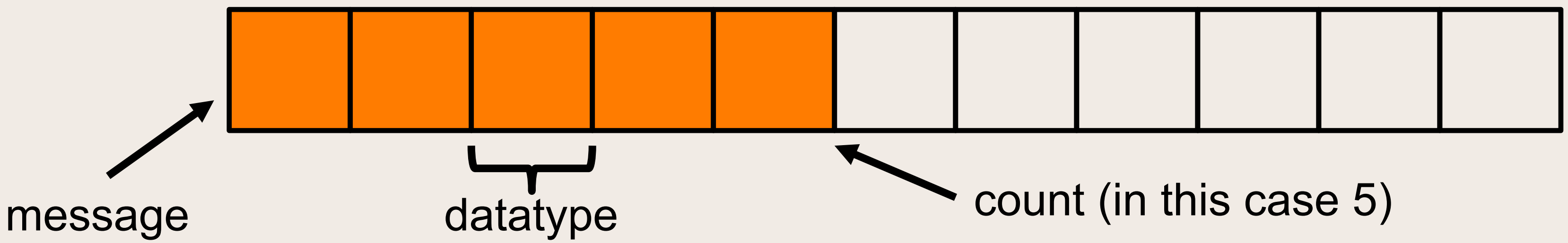
Note: This makes assumptions about the output device and how the MPI subsystem is handling standard output.

MPI Function Declarations

```
int MPI_Init(
        int*       argc_ptr      /*  in/out  */,
        char**     argv_ptr[ ]   /*  in/out  */)

int MPI_Finalize(void)

int MPI_Comm_rank(
        MPI_Comm   comm     /*  in   */,
        int*       result   /*  out  */)

int MPI_Comm_size(
        MPI_Comm   comm     /*  in   */,
        int*       size     /*  out  */)
```

Sending in MPI

```
int MPI_Send(
        void*          message    /* in */,
        int            count      /* in */,
        MPI_Datatype   datatype   /* in */,
        int            dest       /* in */,
        int            tag        /* in */,
        MPI_Comm       comm       /* in */)
```

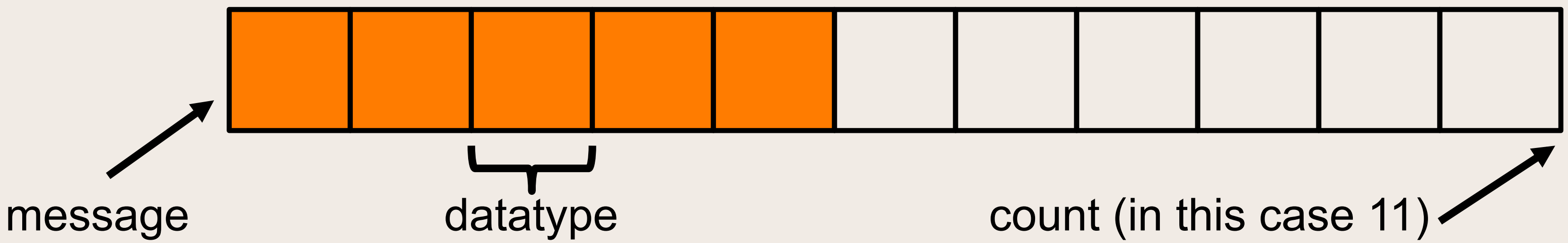


Notes on MPI_Send

- Must be specific as to the process to whom you are sending (no wildcard).
- dest and comm are used together in concert to determine to whom a process is sending.
- Send assumes that the message in memory to be sent is contiguous.
- Tags are integers which are used to distinguish between particular messages sent from one process to another.
- MPI_Send is blocking – the function will only return when the user can reuse the memory which was passed.

Receiving in MPI

```
int MPI_Recv(
        void*           message    /*  out  */,
        int             count      /*  in   */,
        MPI_Datatype    datatype   /*  in   */,
        int             source     /*  in   */,
        int             tag        /*  in   */,
        MPI_Comm        comm       /*  in   */,
        MPI_Status*     status     /*  out  */)
```



Notes on MPI_Recv

- The count within the MPI_Recv denotes the size of the buffer into which the system may place an incoming message. It is not used to select which message is received.
- Assuming the same tags, messages are received in their sending order. Tags are used to distinguish between messages on the incoming message stack.
- MPI_Recv is blocking. It will only return after the message has been received (otherwise an error has occurred which will be denoted in the error and status information).

Predefined MPI Constants

- MPI_ANY_SOURCE (Wildcard Source)
- MPI_ANY_TAG (Wildcard Tag)

- These can only be used with Receive (and its variants). There is no such thing as a wildcard Send.

Example Serial Program

```
#include<iostream.h>


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

  sum = 0;

  for(int i=1;i<=1000;i++)
     sum = sum + i;

  cout << "The sum from 1 to 1000 is: " << sum << endl;

  return 0;
}
```

Example Parallelization of Serial Program

```
#include<iostream.h>
#include<mpi.h>

int main(int argc, char * argv[]){
  int mynode, totalnodes;
  int sum,startval,endval,accum;
  MPI_Status status;

  MPI_Init(argc,argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  sum = 0;
  startval = 1000*mynode/totalnodes+1;
  endval =   1000*(mynode+1)/totalnodes;

  for(int i=startval;i<=endval;i++)
     sum = sum + i;
```

The Programmer Does the Partitioning Work

Example Parallelization of Serial Program

Common Place For A Bug

Pass by Reference

```
for(int i=startval;i<=endval;i++)
   sum = sum + i;

if(mynode!=0)
  MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
else
  for(int j=1;j<totalnodes;j++){
    MPI_Recv(&accum,1,MPI_INT,j,1,MPI_COMM_WORLD, &status);
    sum = sum + accum;
  }


if(mynode == 0)
  cout << "The sum from 1 to 1000 is: " << sum << endl;

MPI_Finalize();

return 0;
}
```

Key Concepts

Key Concept

- Almost everything in MPI can be summed up in the single idea of "Message Sent - Message Received".

Key Concept

- There **must** be a one-to-one correspondence between MPI_Send and MPI_Recv commands. For every message sent using MPI_Send, there must be an explicit receiver using MPI_Recv.

Key Concept

- There **must** be a one-to-one correspondence between MPI_Send and MPI_Recv commands. For every message sent using MPI_Send, there must be an explicit receiver using MPI_Recv.

Terminology: Correctness

Deadlock: An error condition common in parallel programming in which the computation has stalled because a group of processes are blocked and waiting for each other in a cyclic configuration.

Example of a Deadlock Scenario:

Process 0	Process 1
MPI_Send(…,1,…)	MPI_Send(…,0,…);
MPI_Recv(…,1,…)	MPI_Recv(…,0,…);

Terminology: Correctness

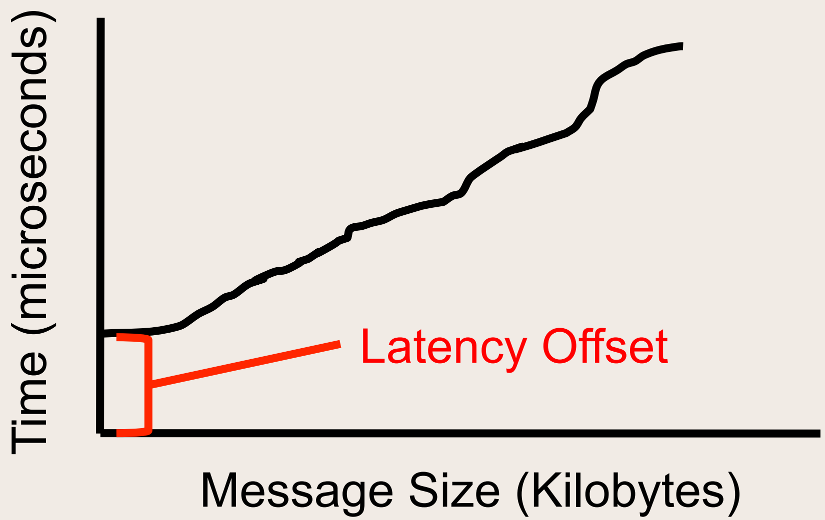
Race condition: An error condition peculiar to parallel programs in which the outcome of a program changes as the relative scheduling of processes varies.

Example of a Race Condition Scenario:

Process 0	Process 1	Process 2
MPI_Send(…,2,…)	MPI_Send(…,2,…)	MPI_Recv(a,MPI_ANY_SOURCE) // Accomplish Func A with data a
		MPI_Recv(b,MPI_ANY_SOURCE) // Accomplish Func B with data b

Terminology: Latency

Latency: The fixed cost of serving a request, such as sending a message or accessing information from a disk. In parallel computing, the term most often is used to refer to the time it takes to send an empty message over the communication medium, from the time the send routine is called to the time the empty message is received by the recipient.



Terminology: Bandwidth

Bandwidth: The capacity of a system, usually expressed as items per second. In parallel computing, the most common usage of the term “bandwidth” is in reference to the number of bytes per second that can be moved across a network link.

Notes:

- Can increase the bandwidth by making the “pipe” larger.
- Larger bandwidth does not equate to lower latency.

MPI_Isend

```
int MPI_Isend(
        void*          message   /*  in   */,
        int            count     /*  in   */,
        MPI_Datatype   datatype  /*  in   */,
        int            dest      /*  in   */,
        int            tag       /*  in   */,
        MPI_Comm       comm      /*  in   */,
        MPI_Request*   request   /*  out  */)
```

Notes on MPI_Isend

- MPI_Isend is non-blocking. The function is used to "initiate" a send and returns immediately. This does not mean that one can reuse the memory as the message may not have been read out of memory yet.

- MPI_Wait or Test is used to bring closure to the non-blocking send operation.

- Isend can be received by all of the various blocking and non-blocking receives.

MPI_Irecv

```
int MPI_Irecv(
        void*           message   /*  out  */,
        int             count     /*  in   */,
        MPI_Datatype    datatype  /*  in   */,
        int             dest      /*  in   */,
        int             tag       /*  in   */,
        MPI_Comm        comm      /*  in   */,
        MPI_Request*    request   /*  out  */)
```

Notes on MPI_Irecv

- MPI_Irecv is non-blocking. The function is used to "initiate" a recv and returns immediately. This does not mean that one can use the memory as the message may not have been read into memory yet.
- MPI_Irecv can be used with any of the blocking or non-blocking MPI send calls.

MPI_Wait

```
int MPI_Wait(
        MPI_Request*   request   /*   in/out   */
        MPI_Status*    status    /*   out      */)
```

Notes on MPI_Wait

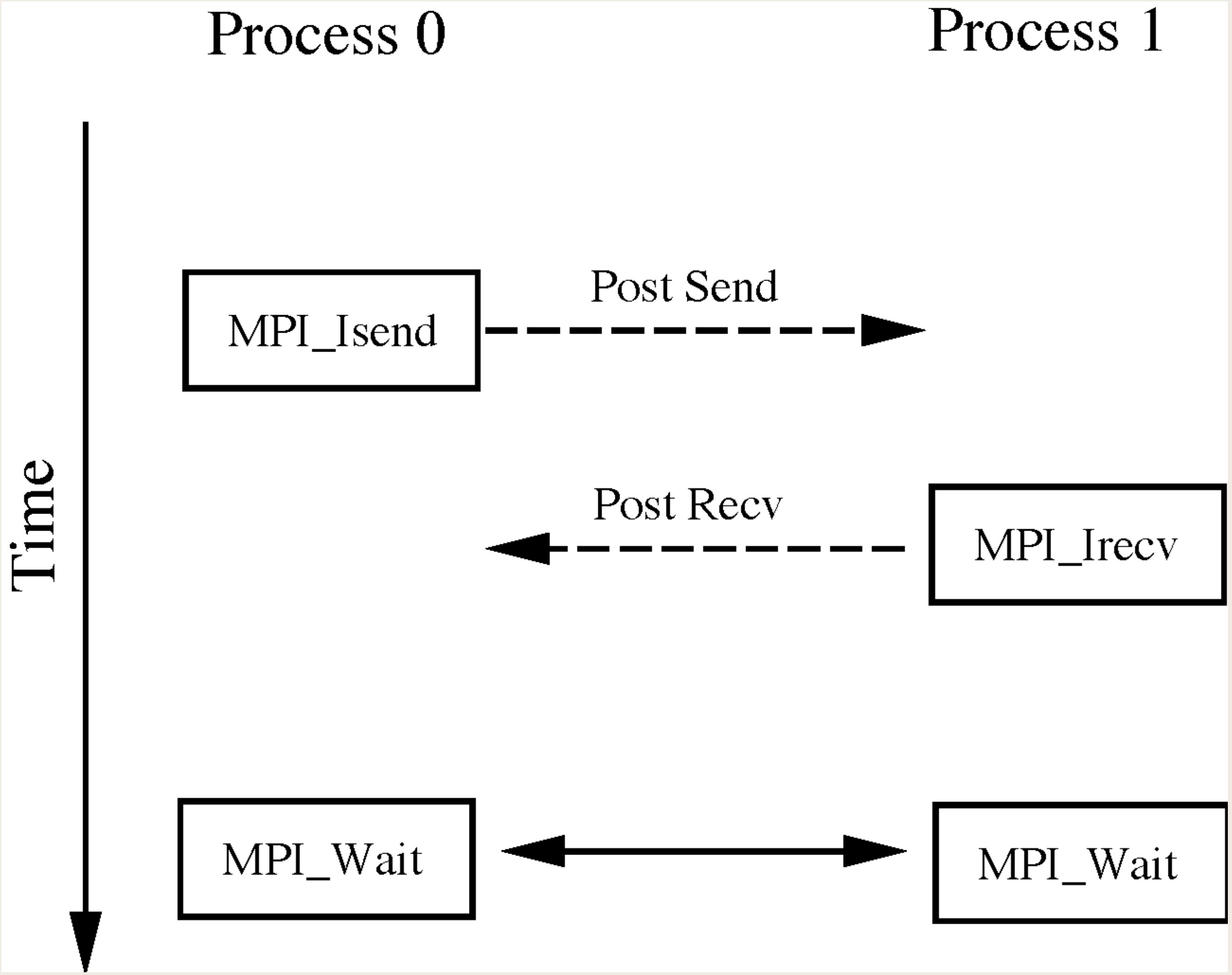
- The Wait function does not return until the request which was initiated by an Isend or Irecv has completed.

- The wait is the point at which the process blocks. If one does not want to block, one can use Test (but test requires polling to see when the process finally completes).

MPI_Sendrecv

```
int MPI_Sendrecv(
        void*          sendbuf    /*  in   */,
        int            sendcount  /*  in   */,
        MPI_Datatype   sendtype   /*  in   */,
        int            dest       /*  in   */,
        int            sendtag    /*  in   */,
        void*          recvbuf    /*  out  */,
        int            recvcount  /*  in   */,
        MPI_Datatype   recvtype   /*  in   */,
        int            source     /*  in   */,
        MPI_Datatype   recvtag    /*  in   */,
        MPI_Comm       comm       /*  in   */,
        MPI_Status*    status     /*  out  */)
```

Isend/Irecv/Wait

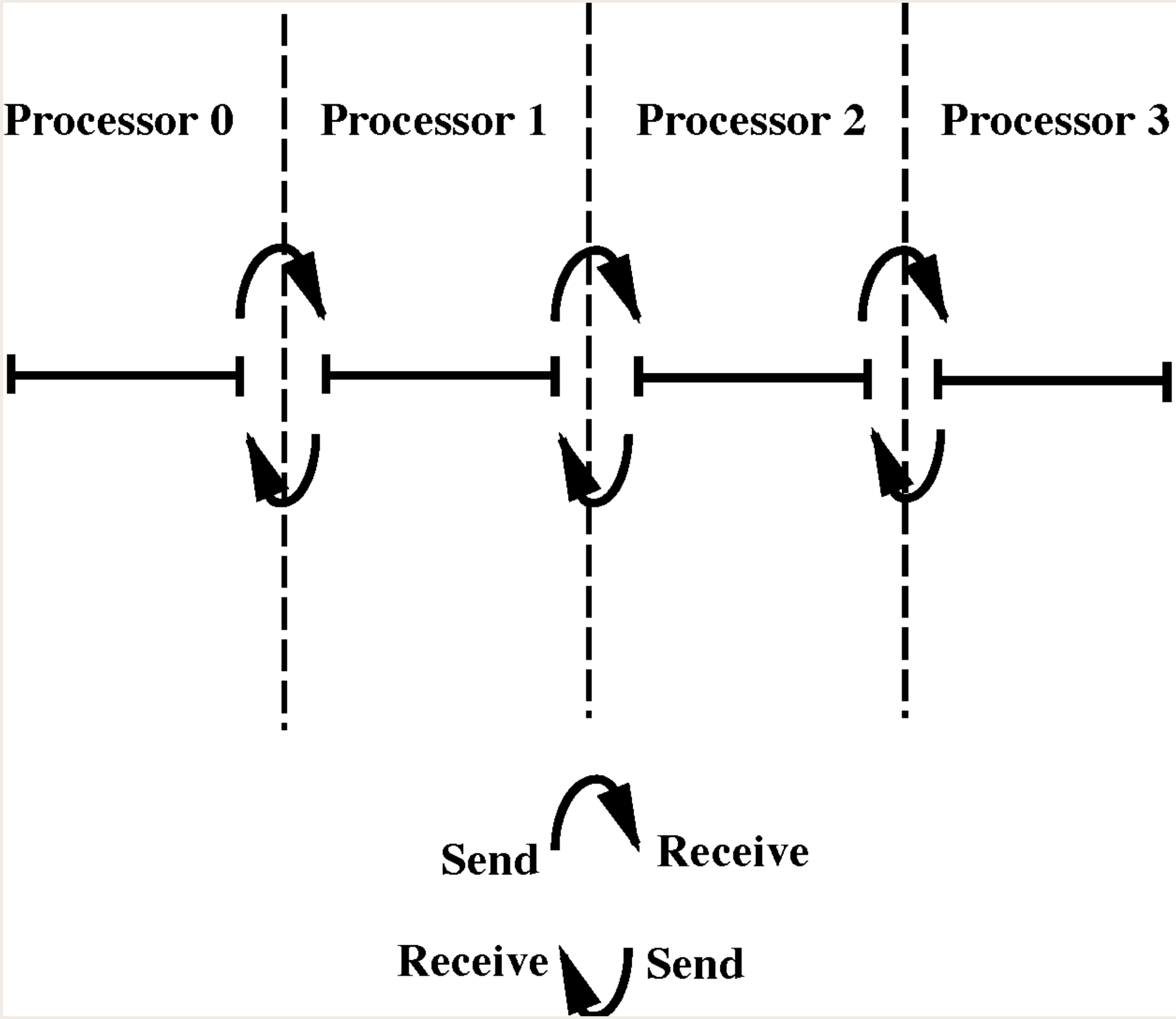


Notes on MPI_Sendrecv

- The sendrecv command is used whenever two processes are going to "swap" data. Note it is not required that the swapping be symmetrical – each process within the pair may send different data (different types and different number).

- MPI contains a Sendrecv_replace operator which technically only works when buffering exists within the system.

Sendrecv



Processor 0
Processor 1
Processor 2
Processor 3
Send
Receive
Receive
Send

MPI Collective Operations

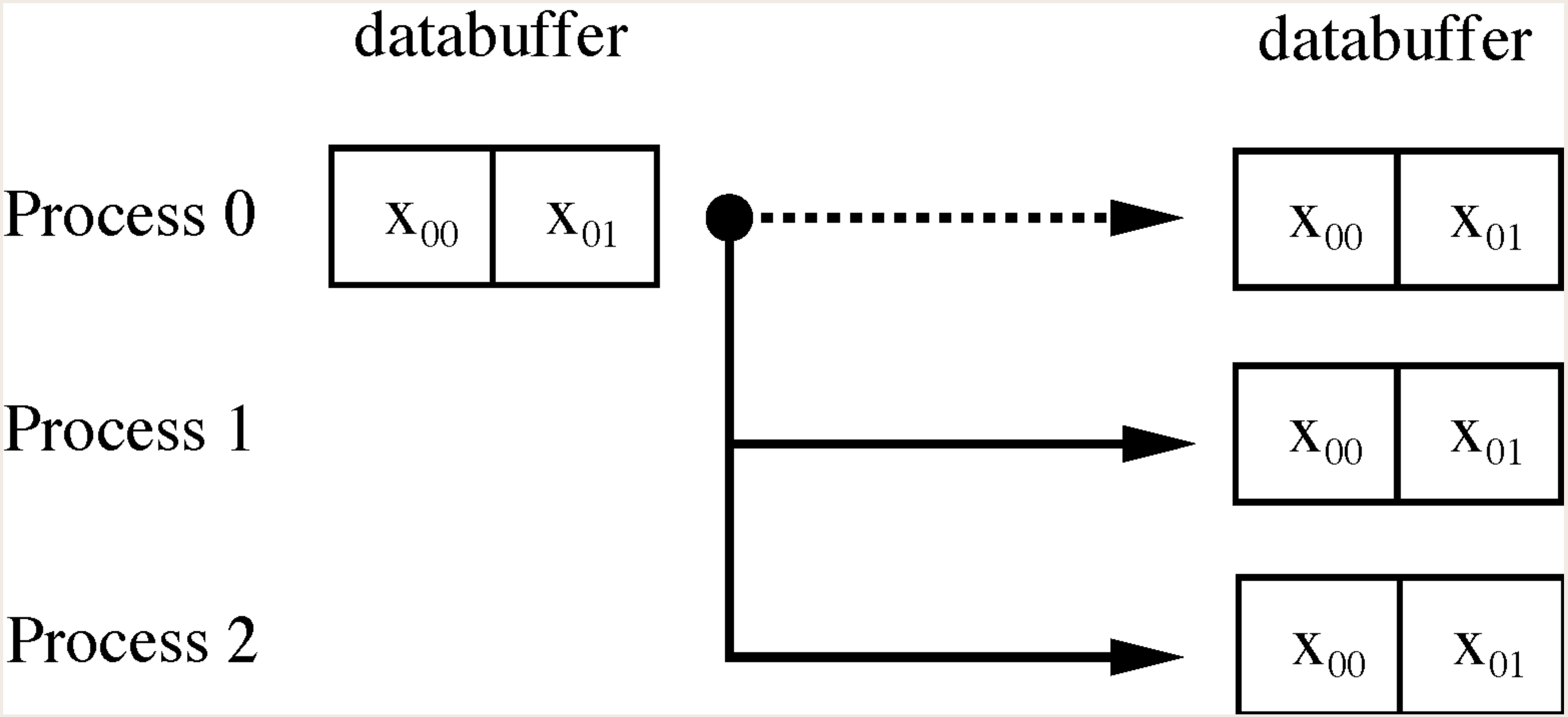
MPI_Barrier

```
int MPI_Barrier(
            MPI_Comm    comm    /*    in    */)
```

MPI_Bcast

```
int MPI_Bcast(
        void*          buffer     /*  in/out  */,
        int            count      /*  in      */,
        MPI_Datatype   datatype   /*  in      */,
        int            root       /*  in      */
        MPI_Comm       comm       /*  in      */)
```

MPI_Bcast



databuffer
databuffer
Process 0
Process 1
Process 2
x_{00}
x_{01}
x_{00}
x_{01}
x_{00}
x_{01}
x_{00}
x_{01}

MPI_Reduce

```
int MPI_Reduce(
        void*       operand   /* in  */,
        void*       result    /* out */,
        int         count     /* in  */,
        MPI_Op      operator  /* in  */,
        int         root      /* in  */,
        MPI_Comm    comm      /* in  */)
```

Predefined MPI Operations

Operation Name	*Meaning*
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

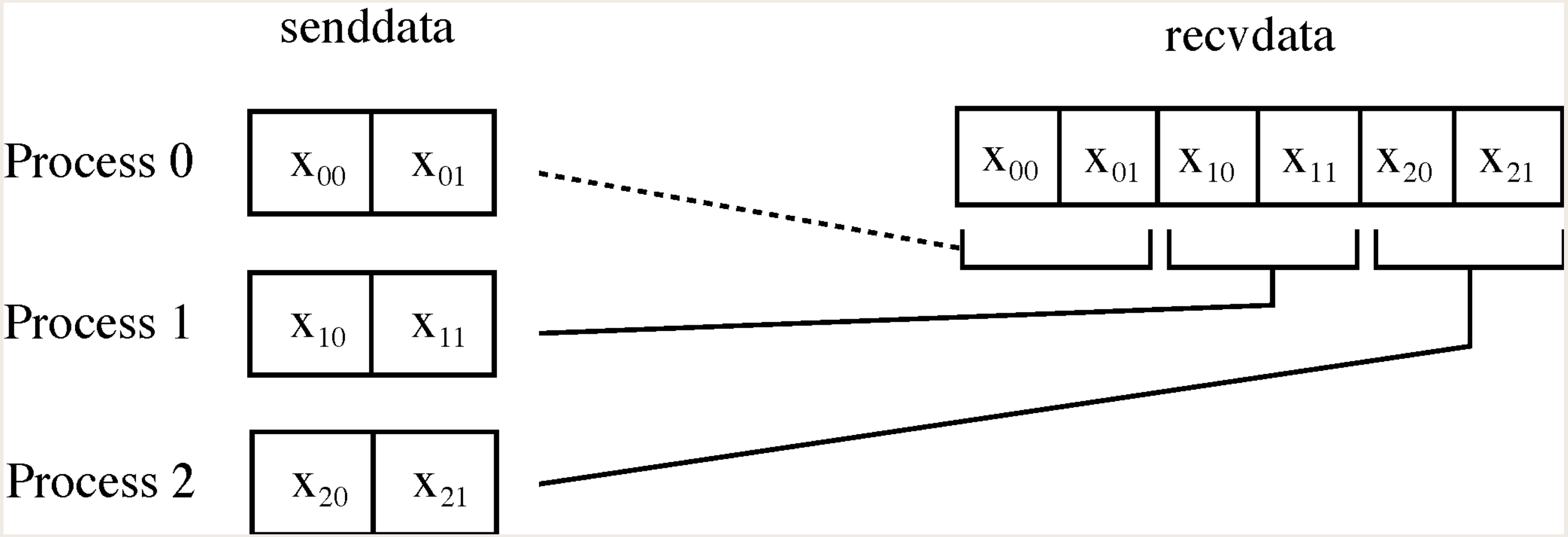
MPI_Allreduce

```
int MPI_Allreduce(
        void*          operand    /* in  */,
        void*          result     /* out */,
        int            count      /* in  */,
        MPI_Datatype   datatype   /* in  */,
        MPI_Op         operator   /* in  */,
        MPI_Comm       comm       /* in  */)
```

MPI_Gather

```
int MPI_Gather(
        void*           sendbuf     /*  in   */,
        int             sendcount   /*  in   */,
        MPI_Datatype    sendtype    /*  in   */,
        void*           recvbuf     /*  out  */,
        int             recvcounts  /*  in   */
        MPI_Datatype    recvtype    /*  in   */,
        MPI_Comm        comm        /*  in   */)
```

MPI_Gather



senddata
recvdata
Process 0
Process 1
Process 2
x_{00} x_{01}
x_{10} x_{11}
x_{20} x_{21}
x_{00} x_{01} x_{10} x_{11} x_{20} x_{21}

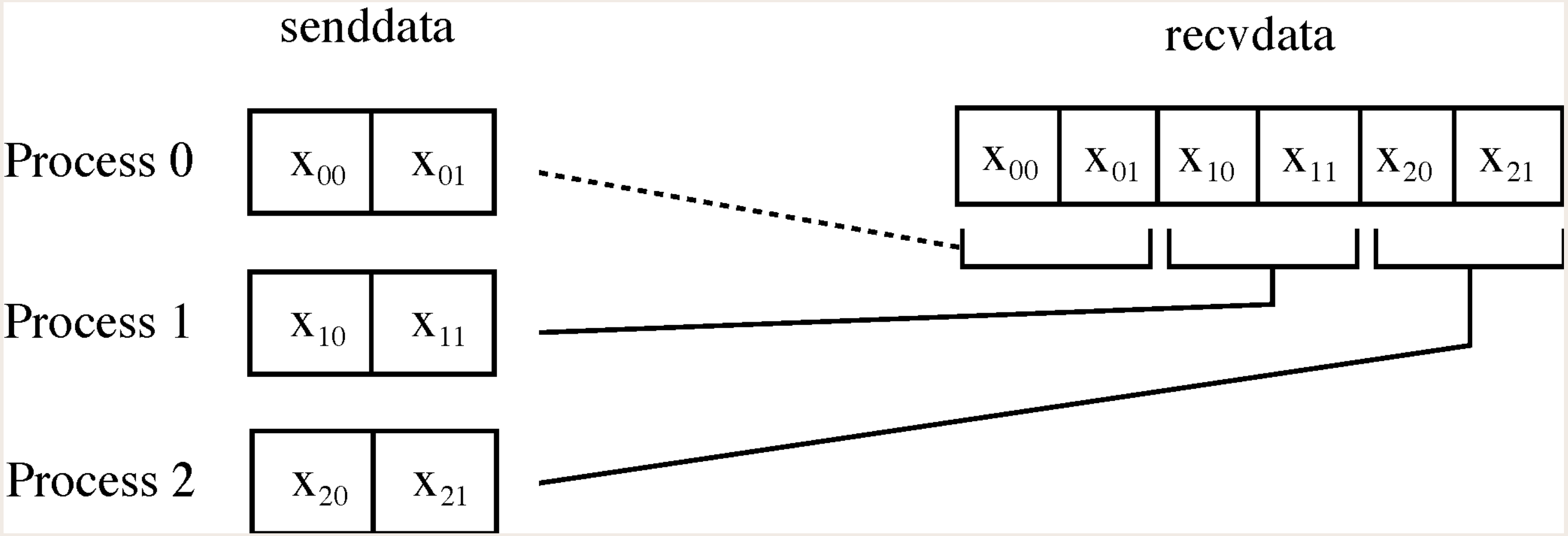
MPI_Gatherv

```
int MPI_Gatherv(
        void*           sendbuf            /* in   */,
        int             sendcount          /* in   */,
        MPI_Datatype    sendtype           /* in   */,
        void*           recvbuf            /* out  */,
        int             recvcounts[ ]      /* in   */
        int             displacements[ ]   /* in   */,
        MPI_Datatype    recvtype           /* in   */,
        int             root               /* in   */,
        MPI_Comm        comm               /* in   */)
```

MPI_Allgather

```
int MPI_Allgather(
        void*           sendbuf     /* in   */,
        int             sendcount   /* in   */,
        MPI_Datatype    sendtype    /* in   */,
        void*           recvbuf     /* out  */,
        int             recvcount   /* in   */
        MPI_Datatype    recvtype    /* in   */,
        MPI_Comm        comm        /* in   */)
```

MPI_Allgather



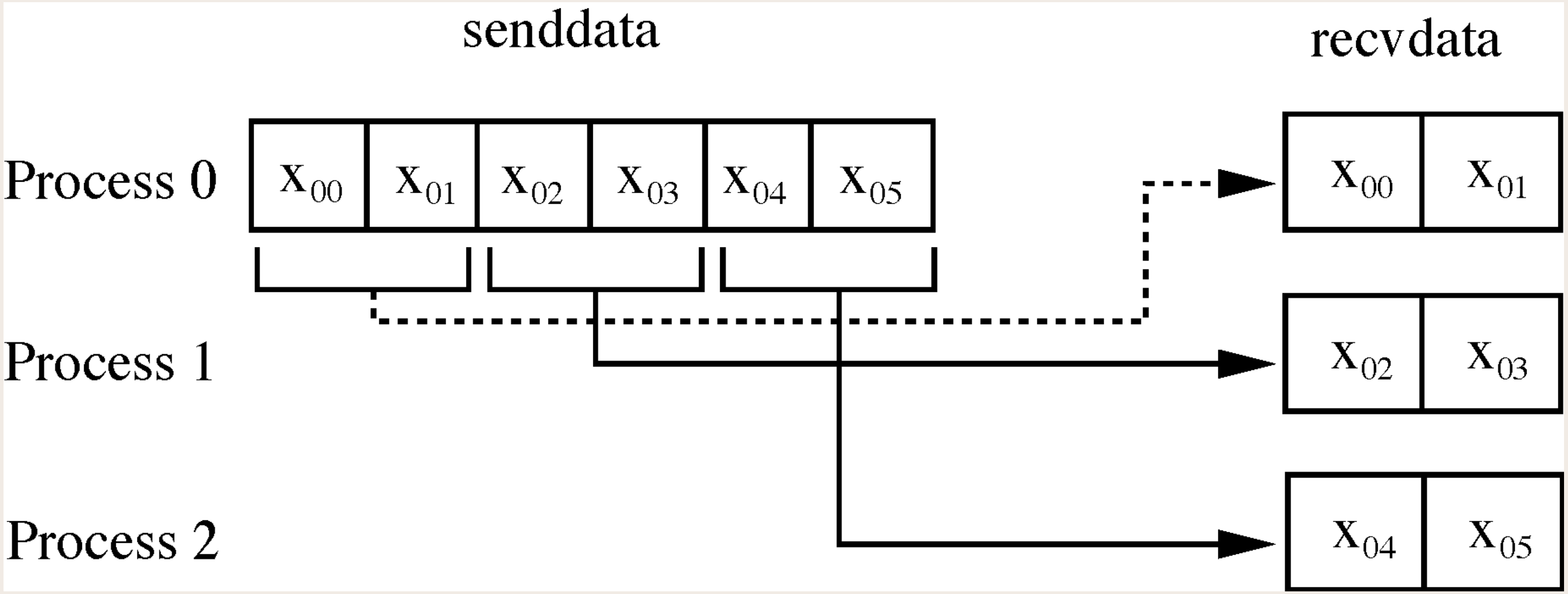
MPI_Allgatherv

```
int MPI_Allgatherv(
        void*           sendbuf           /*  in   */,
        int             sendcount         /*  in   */,
        MPI_Datatype    sendtype          /*  in   */,
        void*           recvbuf           /*  out  */,
        int             recvcounts[ ]     /*  in   */
        int             displacements[ ]  /*  in   */,
        MPI_Datatype    recvtype          /*  in   */,
        MPI_Comm        comm              /*  in   */)
```

MPI_Scatter

```
int MPI_Scatter(
        void*           sendbuf     /*  in   */,
        int             sendcount   /*  in   */,
        MPI_Datatype    sendtype    /*  in   */,
        void*           recvbuf     /*  out  */,
        int             recvcount   /*  in   */
        MPI_Datatype    recvtype    /*  in   */,
        int             root        /*  in   */,
        MPI_Comm        comm        /*  in   */)
```

MPI_Scatter


senddata
recvdata
Process 0
Process 1
Process 2
x_{00}
x_{01}
x_{02}
x_{03}
x_{04}
x_{05}
x_{00}
x_{01}
x_{02}
x_{03}
x_{04}
x_{05}

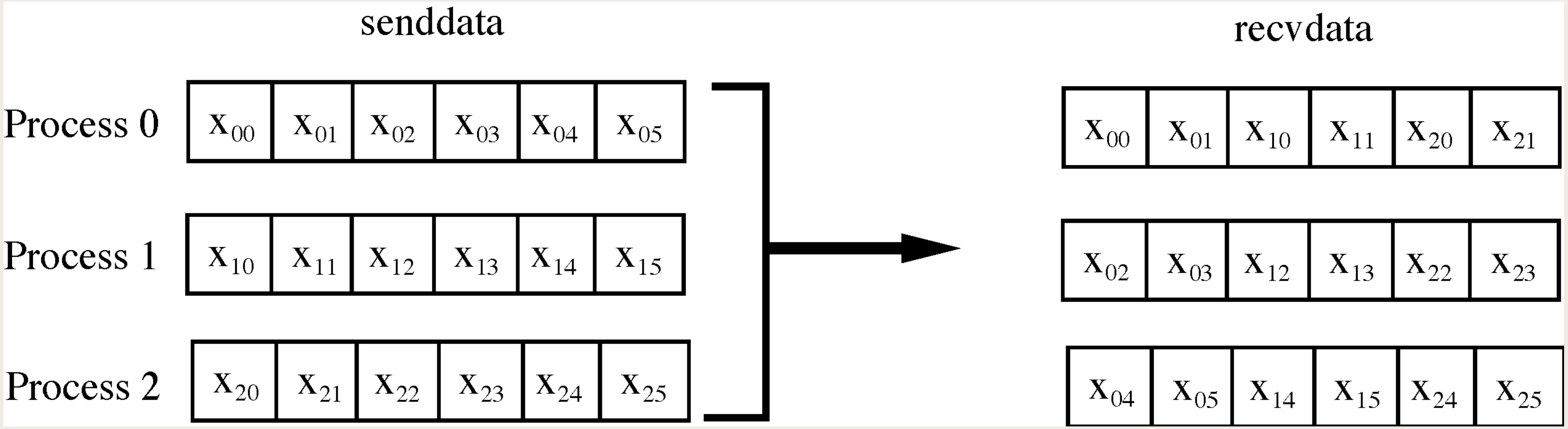
MPI_Scatterv

```
int MPI_Scatterv(
        void*           sendbuf             /*  in   */,
        int             sendcounts[ ]       /*  in   */,
        int             displacements[ ]    /*  in   */,
        MPI_Datatype    sendtype            /*  in   */,
        void*           recvbuf             /*  out  */,
        int             recvcount           /*  in   */
        MPI_Datatype    recvtype            /*  in   */,
        int             root                /*  in   */,
        MPI_Comm        comm                /*  in   */)
```

MPI_Alltoall

```
int MPI_Alltoall(
        void*          sendbuf    /* in  */,
        int            sendcount  /* in  */,
        MPI_Datatype   sendtype   /* in  */,
        void*          recvbuf    /* out */,
        int            recvcount  /* in  */
        MPI_Datatype   recvtype   /* in  */,
        MPI_Comm       comm       /* in  */)
```

MPI_Alltoall



MPI_Alltoallv

```
int MPI_Alltoallv(
        void*          sendbuf                 /* in   */,
        int            sendcounts[ ]           /* in   */,
        int            send_displacements[ ]   /* in   */,
        MPI_Datatype   sendtype                /* in   */,
        void*          recvbuf                 /* out  */,
        int            recvcounts[ ]           /* in   */
        int            recv_displacements[ ]   /* in   */,
        MPI_Datatype   recvtype                /* in   */,
        MPI_Comm       comm                    /* in   */)
```