UCIrvine | The Henry Samueli School of Engineering

Message Passing Interface (MPI); Collective Communication

EECS 221: Intro to High-Performance Computing

Aparna Chandramowlishwaran

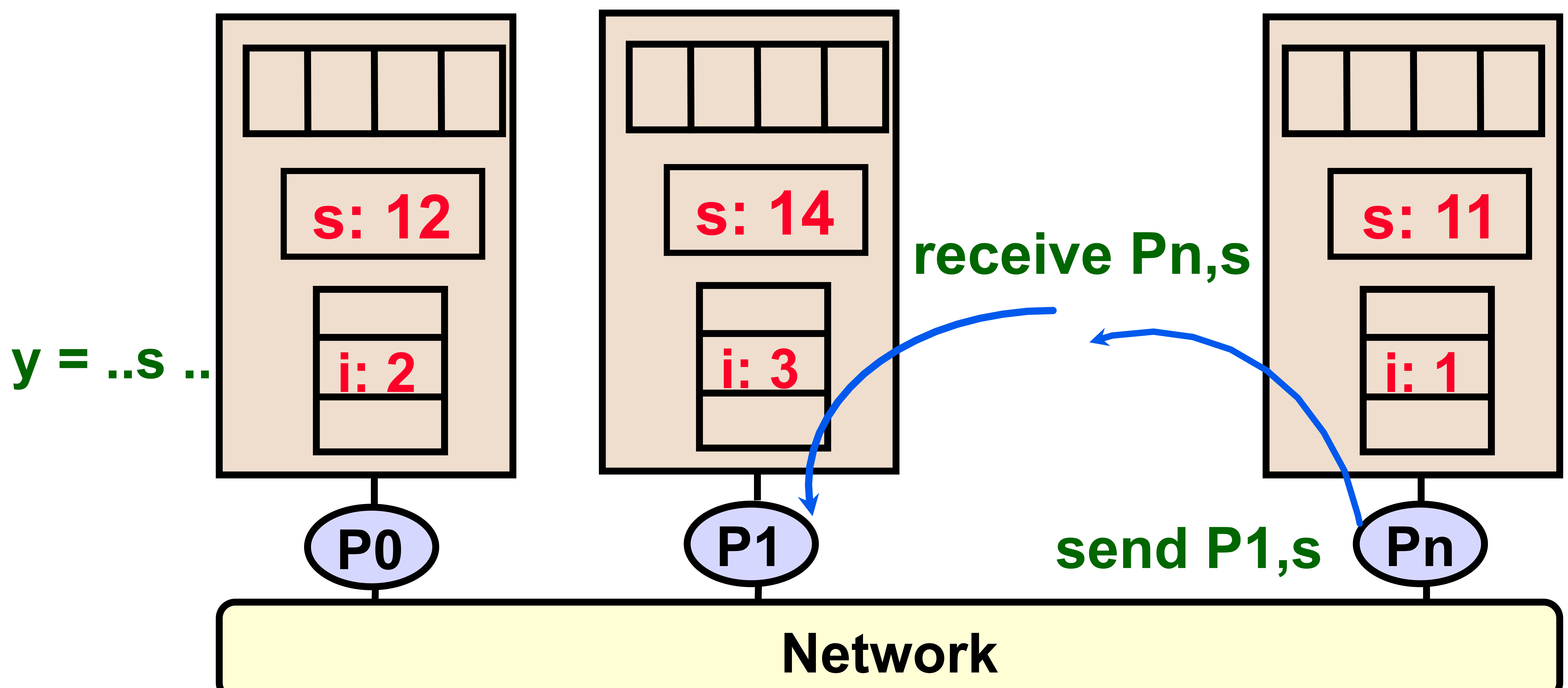
April 27, 2017

Administrivia

- Homework 2: **Hands-on lab** on Tuesday May 2nd (next class)
- Readings posted on the class website

Message Passing Model

- Program consists of a collection of named processes
- Processes communicate by explicit send/receive messages
- Coordination is implicit in every communication
- MPI has become the de facto standard



Message Passing Interface

- All communication, synchronization require subroutine calls
  - No shared variables
- Communication primitives
  - Pairwise, or point-to-point: send & receive ([non]blocking, [a]synchronous)
  - Collectives
    - **Move data**: Broadcast, Scatter/gather
    - **Compute and move**: sum, product, max, prefix sum, etc,.
- Barrier synchronization
  - No locks because there are no shared variables to protect
- **See**: http://www.mpi-forum.org and http://www.mcs.anl.gov/research/projects/mpi

MPI environment

- Two important questions arise in a parallel program
  - How many processes are participating in this computation?
  - Which one am I?

MPI environment

- Two important questions arise in a parallel program
  - How many processes are participating in this computation? MPI_Comm_size returns the number of processors, ***size***
  - Which one am I? MPI_Comm_rank reports the ***rank***, a number between **0** and **size-1**, identifying the calling process

“Hello World” example in MPI

```
int main(int argc, char *argv[])
{
  int rank, size;



  printf (“I am %d of %d\n”, rank, size); // Execute in parallel

  return 0;
}
```

“Hello World” example in MPI

```
#include <mpi.h>

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




  printf (“I am %d of %d\n”, rank, size);

  return 0;
}
```

“Hello World” example in MPI

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init (&argc, &argv);


  printf (“I am %d of %d\n”, rank, size);
  MPI_Finalize ();
  return 0;
}
```

“Hello World” example in MPI

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  printf (“I am %d of %d\n”, rank, size);
  MPI_Finalize ();
  return 0;
}
```

“Hello World” example in MPI

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  printf (“I am %d of %d\n”, rank, size);
  MPI_Finalize ();
  return 0;
}
```

Compiling:

```
$ mpicc -o hello hello.c
```

“Hello World” example in MPI

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  printf ("I am %d of %d\n", rank, size);
  MPI_Finalize ();
  return 0;
}
```

Compiling:

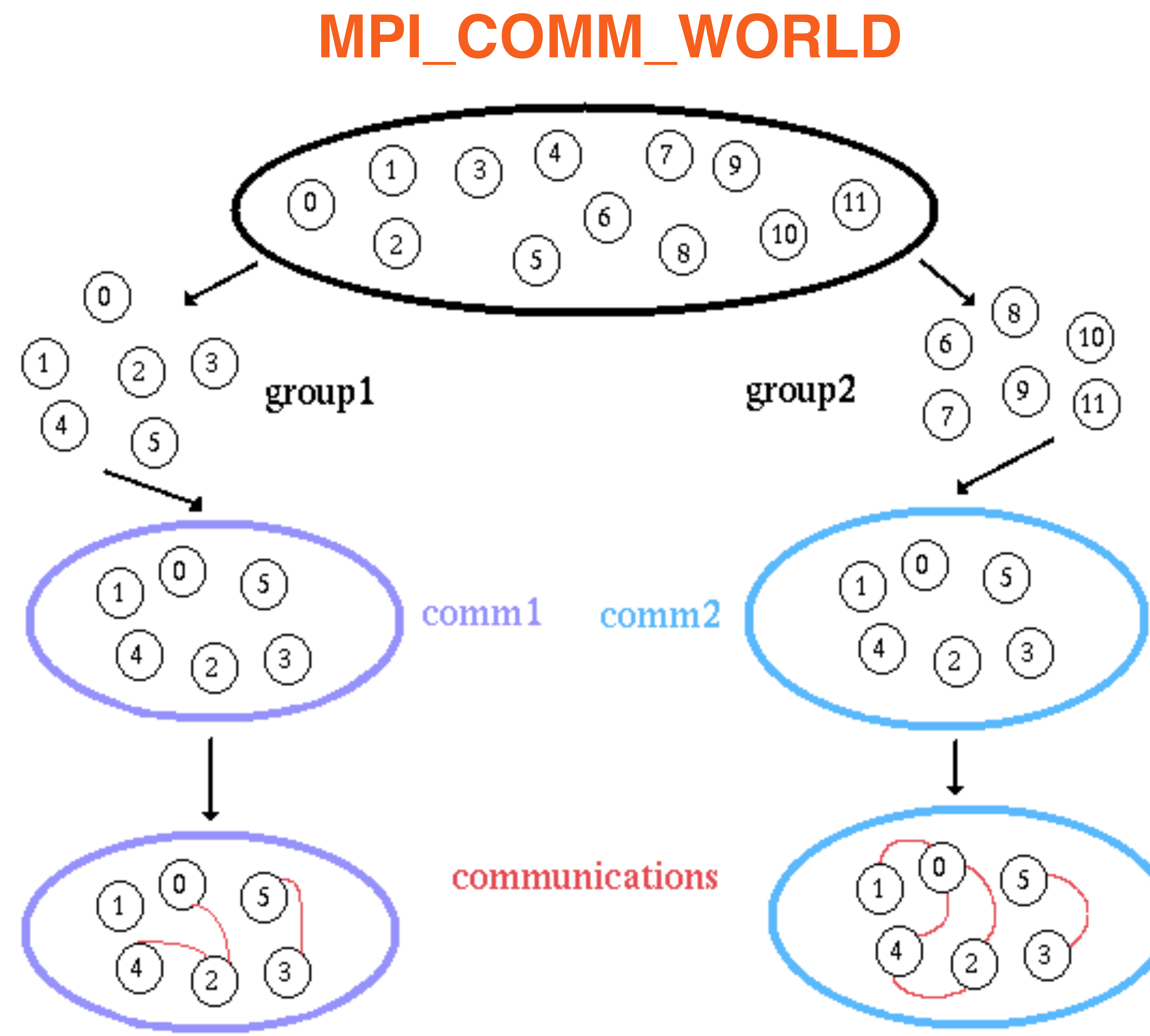
```
$ mpicc -o hello hello.c
```

Run:

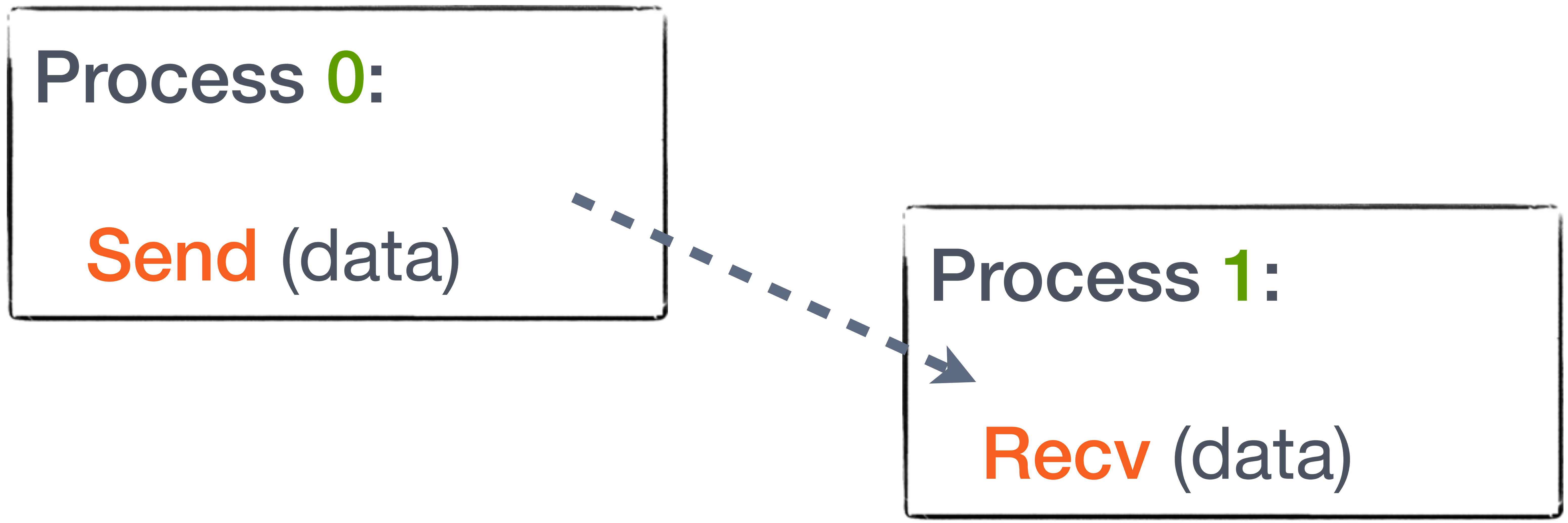
```
$ mpirun -np 4 ./hello
```

Basic concept: Communicators

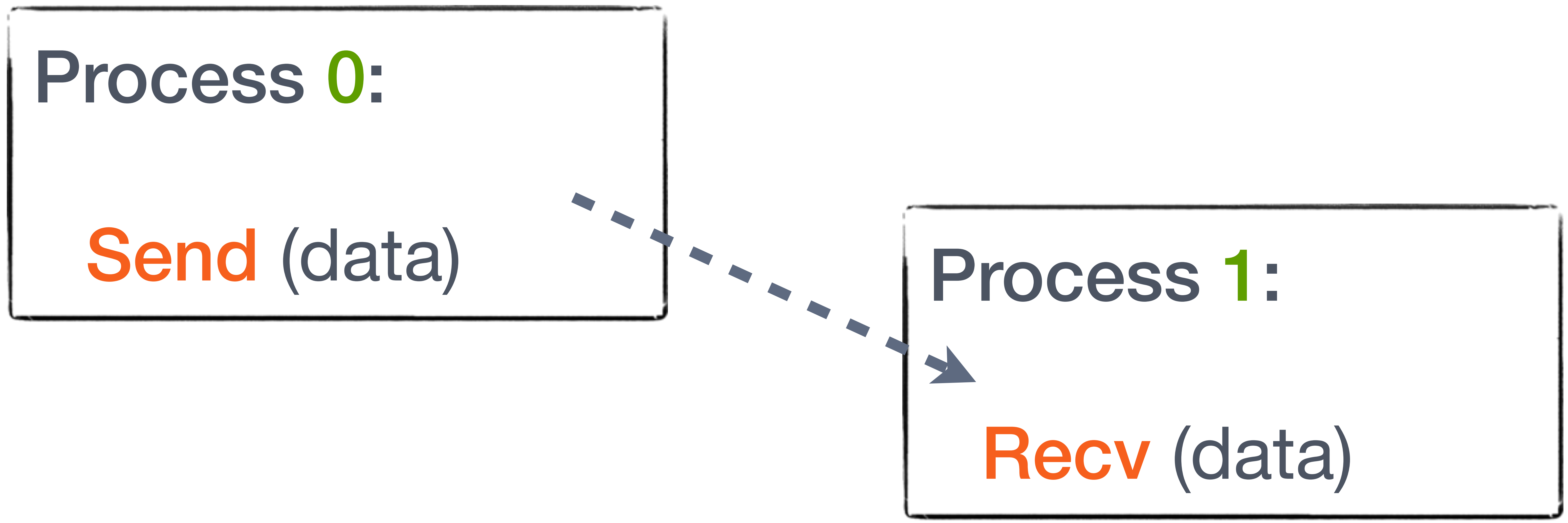
- **Group**: collection of processors
- Each message is sent in a **context**, and must be received in the same context
- **Communicator** = group + context
- **Rank**: process ID in its communicator
- MPI_COMM_WORLD = Group consisting of all processes
- MPI_ANY_RANK = Wildcard rank (receive only)



Basic concepts: Send and receive

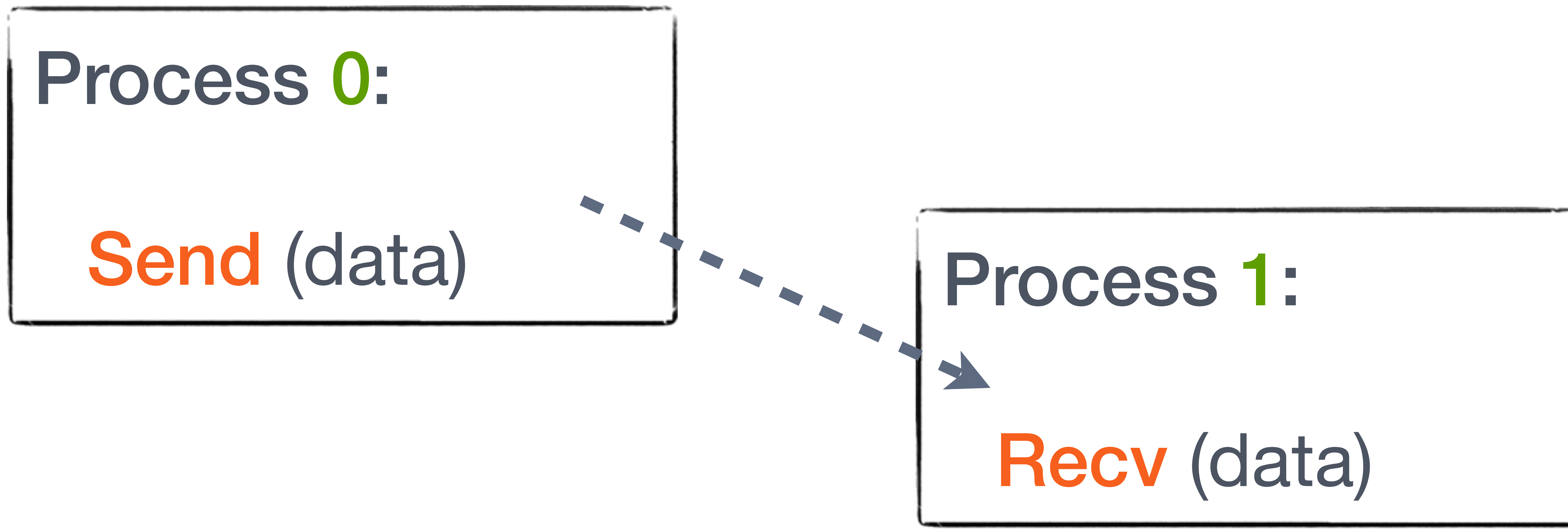


Basic concepts: Send and receive



- How to identify tasks/processes?
- How to describe “data”?
- How will receiver recognize and screen messages?
- What does it mean for operations to complete?

Basic concepts: Send and receive

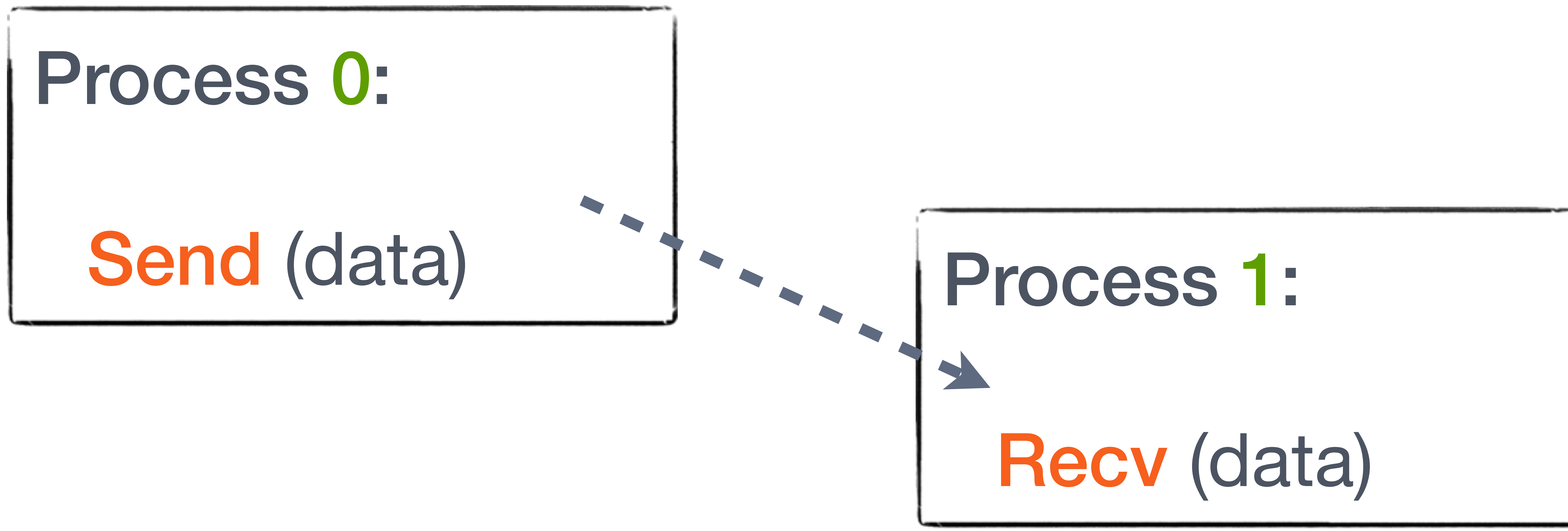


- How to identify tasks/processes?
- **How to describe "data"?**
- How will receiver recognize and screen messages?
- What does it mean for operations to complete?

Basic concept: MPI datatypes

- In MPI call, "data" is described a triple: (**address, count, type**)
- Datatype = (recursively defined)
  - "Standard" predefined scalar types: MPI_INT, MPI_DOUBLE, MPI_CHAR, etc,.
  - A contiguous array of datatypes
  - A strided block of datatypes
  - An indexed array of blocks of datatypes
  - An arbitrary structure of datatypes
- Can construct custom datatypes, in particular ones for subarrays

Basic concepts: Send and receive



- How to identify tasks/processes?
- How to describe "data"?
- **How will receiver recognize and screen messages?**
- What does it mean for operations to complete?

Basic concepts: MPI tags

- **Message tags**
  - Every message has a user-defined integer ID to assist the receiver in identifying the message
  - Messages can be screened at the receiver by specifying a tag
  - Wildcard: **MPI_ANY_TAG**

Basic concept: Status object

- Opaque structures that contains more information (e.g. size of the message, error)

```
int received_tag, received_from, received_count;
MPI_Status status;
MPI_Recv (…, MPI_ANY_SOURCE, MPI_ANY_TAG, …, &status);
received_tag = status.MPI_TAG;
received_from = status.MPI_SOURCE;
MPI_Get_count (status, datatype, &received_count);
```

Blocking send

MPI_Send (buffer, count, datatype, dest-rank, tag, comm)

Process 0:
MPI_Send (A, 10, MPI_INT, 1, *tag, comm*);

Process 1:
MPI_Recv (B, 20, MPI_INT, 0, *tag, comm, &stat*);

- Message buffer = (**buffer-start, count, datatype**)
- Target = (**dest-rank, tag, comm**)
- On return:
  - Data delivered to the system
  - Buffer can be reused
  - Target may not yet have received the message

Blocking receive

MPI_Recv (buffer, count, datatype, source-rank, tag, comm, status)

```
Process 0:
   MPI_Send (A, 10, MPI_INT, 1, tag, comm);
```

```
Process 1:
   MPI_Recv (B, 20, MPI_INT, 0, tag, comm, &stat);
```

- Message buffer = (**buffer-start, count, datatype**) Source = (**source-rank, tag, comm**)
- Returns when matching message (match on source triplet) is received
  - Source can be a wildcard, MPI_ANY_SOURCE
  - Tag can be a wildcard, MPI_ANY_TAG
  - Receiving fewer than *count* items is OK, but more is an error
  - May query *status* for more information (e.g. size of the message, source rank)

Simple MPI program

```
#include <mpi.h>
int main(int argc, char *argv[])
{
  int rank, buffer;
  MPI_Status status;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    buffer = 123;
    MPI_Send (&buffer, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv (&buffer, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    printf ("Received %d\n", buffer);
  }
  MPI_Finalize ();
  return 0;
}
```

Beware of deadlock

Process 0:	Process 1:
Recv (data ← 1)	Recv (data ← 0)
Send (data → 1)	Send (data → 0)

Beware of deadlock

```
Process 0:
  Recv (data ← 1)
  Send (data → 1)
```

```
Process 1:
  Recv (data ← 0)
  Send (data → 0)
```

```
Process 0:
  Send (data → 1)
  Recv (data ← 1)
```

```
Process 1:
  Send (data → 0)
  Recv (data ← 0)
```

Beware of deadlock

Process 0:	Process 1:
Recv (data ← 1) Send (data → 1)	Recv (data ← 0) Send (data → 0)

Process 0:	Process 1:
Send (data → 1) Recv (data ← 1)	Send (data → 0) Recv (data ← 0)

- Unsafe orderings of send/recv
- How to avoid?

Beware of deadlock

- Use **safe orderings** of send/recv

```
Process 0:
  Send (data → 1)
  Recv (data ← 1)
```

```
Process 1:
  Recv (data ← 0)
  Send (data → 0)
```

- Use **simultaneous** send/recv

```
Process 0:
  SendRecv (data → 1)
```

```
Process 1:
  SendRecv (data → 0)
```

MPI_SendRecv (Sbuffer, Scount, Stype, dest, Stag, Rbuffer, Rcount, Rtype, src, comm, &stat)

Beware of deadlock

- Unsafe orderings of send/recv

Process 0:	Process 1:
Send (data → 1)	Send (data → 0)
Recv (data ← 1)	Recv (data ← 0)

- Use non-blocking send/recv

Process 0:	Process 1:
Isend (data1 → 1)	Isend (data1 → 0)
Irecv (data2 ← 1)	Irecv (data2 ← 0)
Waitall	Waitall

MPI_Isend (buffer, count, datatype, dest-rank, tag, comm, request)

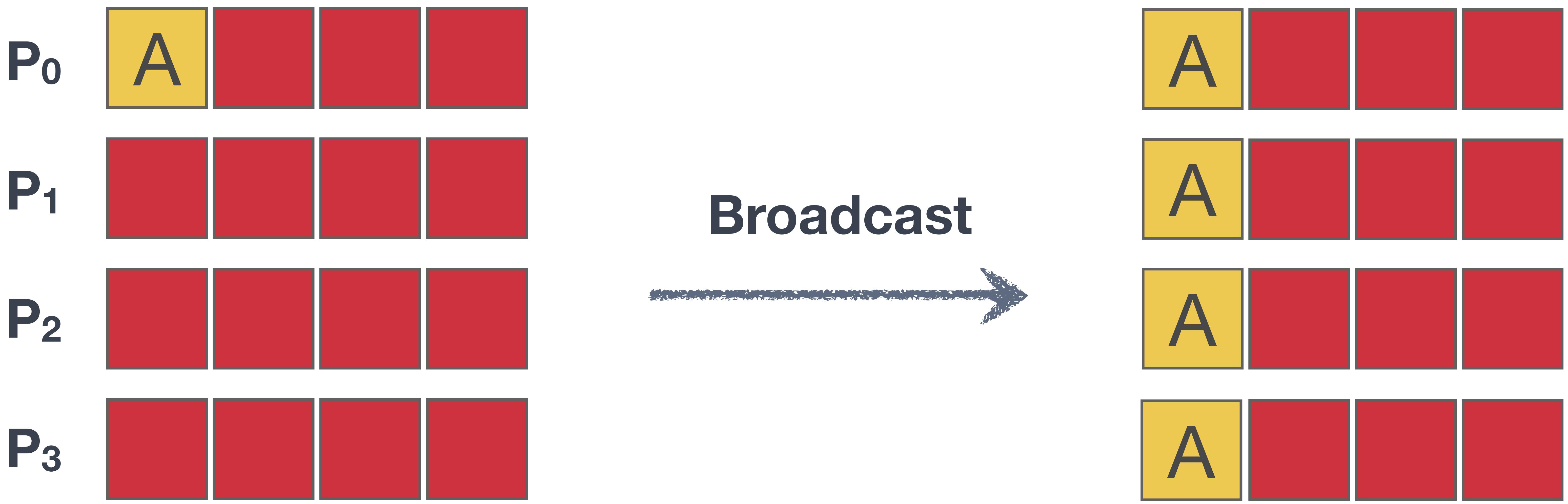
MPI_Waitall (count, request[ ], status[ ])

Summary: MPI

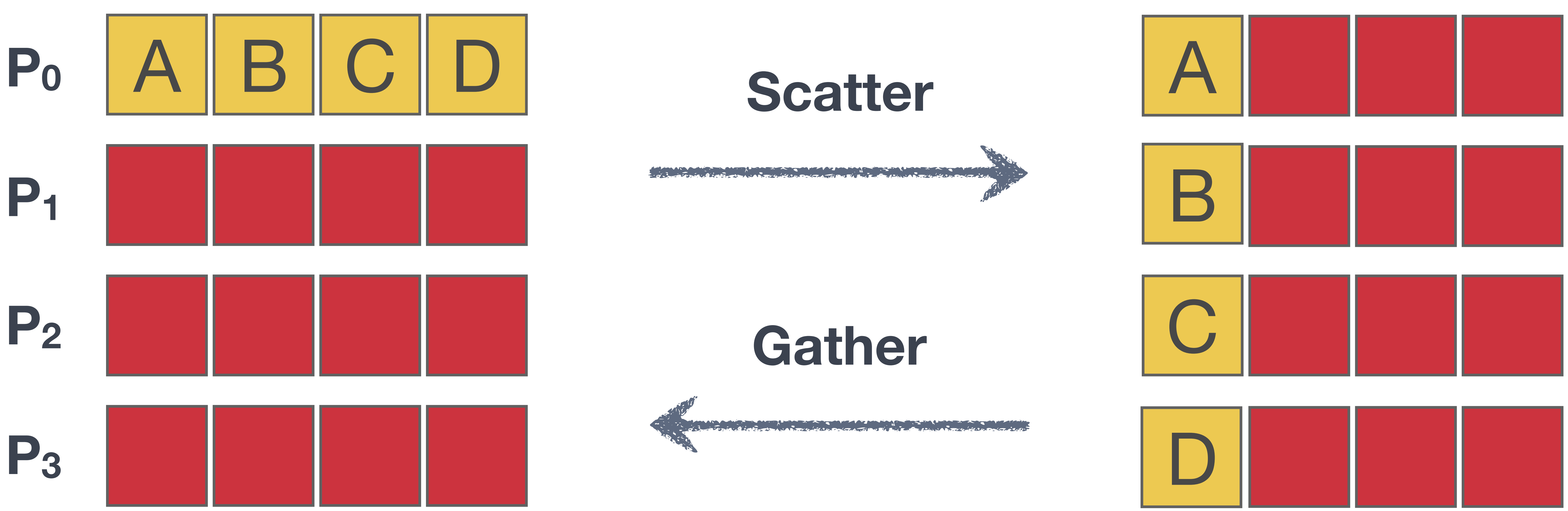
- Many parallel programs can be written using six commonly used MPI primitives
  - **MPI_Init**
  - **MPI_Finalize**
  - **MPI_Comm_rank**
  - **MPI_Comm_size**
  - **MPI_Send**
  - **MPI_Recv**
- Non-blocking primitives for correctness and performance

Alternate approach

- **Collective communication**
  - Higher-level communication primitives
    - **MPI_Bcast**: Broadcast data to all processes
    - **MPI_Reduce**: Combine data from all processes to one process
    - **MPI_Barrier**
- Each process executes the same communication operation
- MPI provides a rich set of collective operations
- Presumably optimized/tuned for hardware



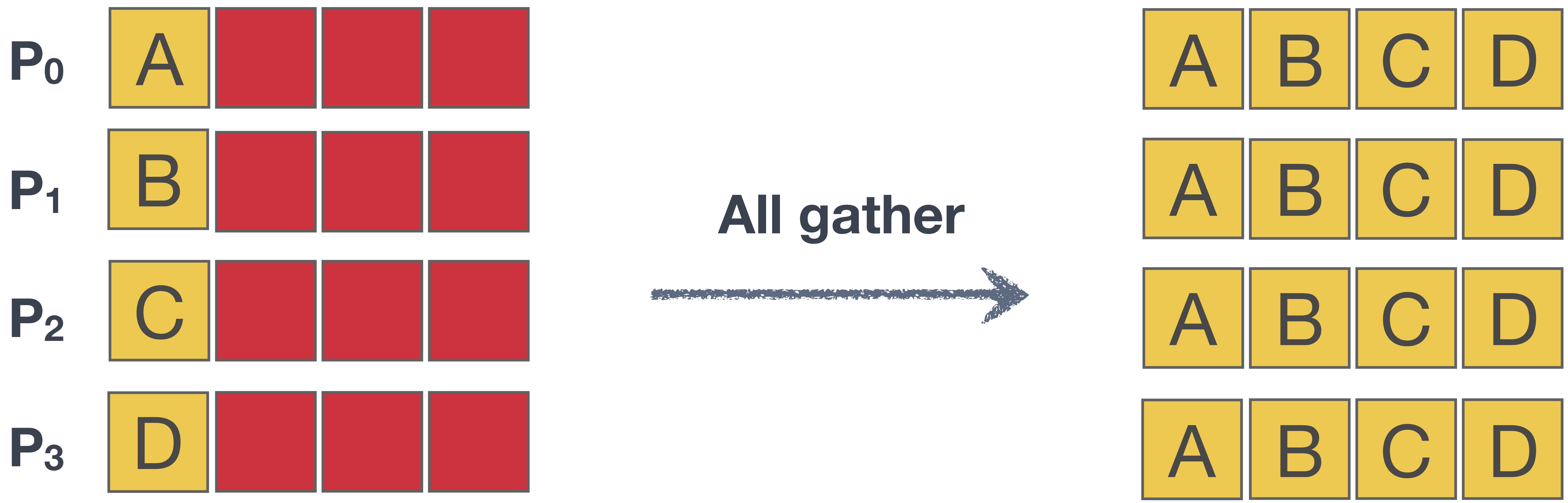
MPI_Bcast (buffer, count, datatype, root, comm)



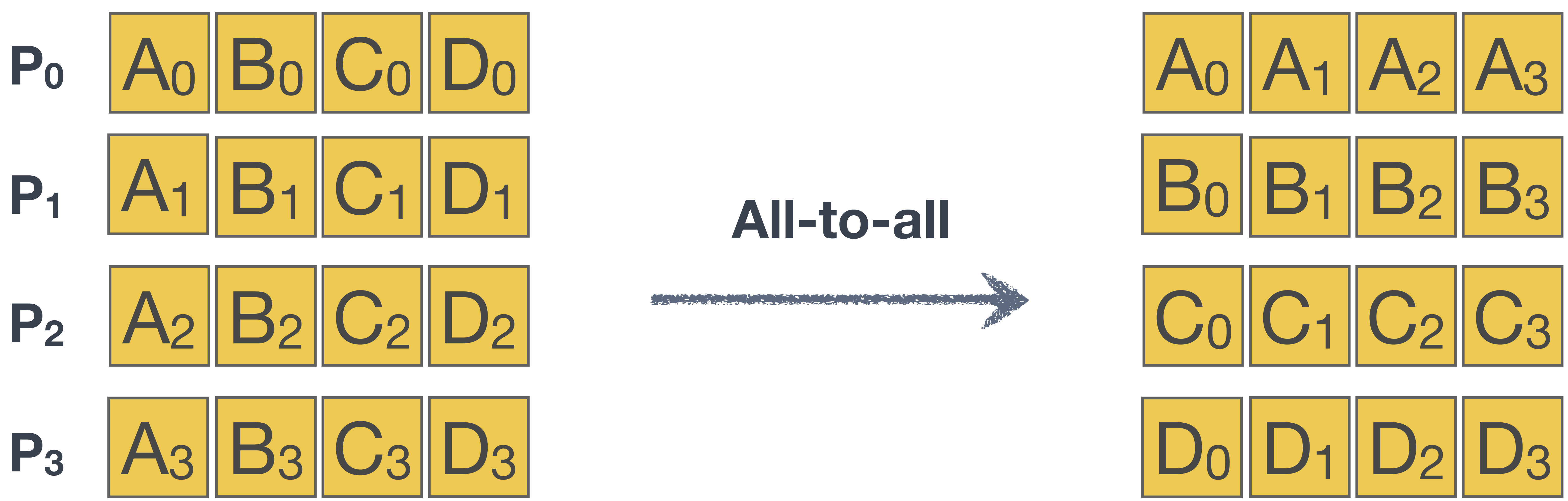
MPI_Scatter (Sbuffer, Scount, Stype, Rbuffer, Rcount, Rtype, root, comm)
MPI_Gather (Sbuffer, Scount, Stype, Rbuffer, Rcount, Rtype, root, comm)

Case study: Broadcast

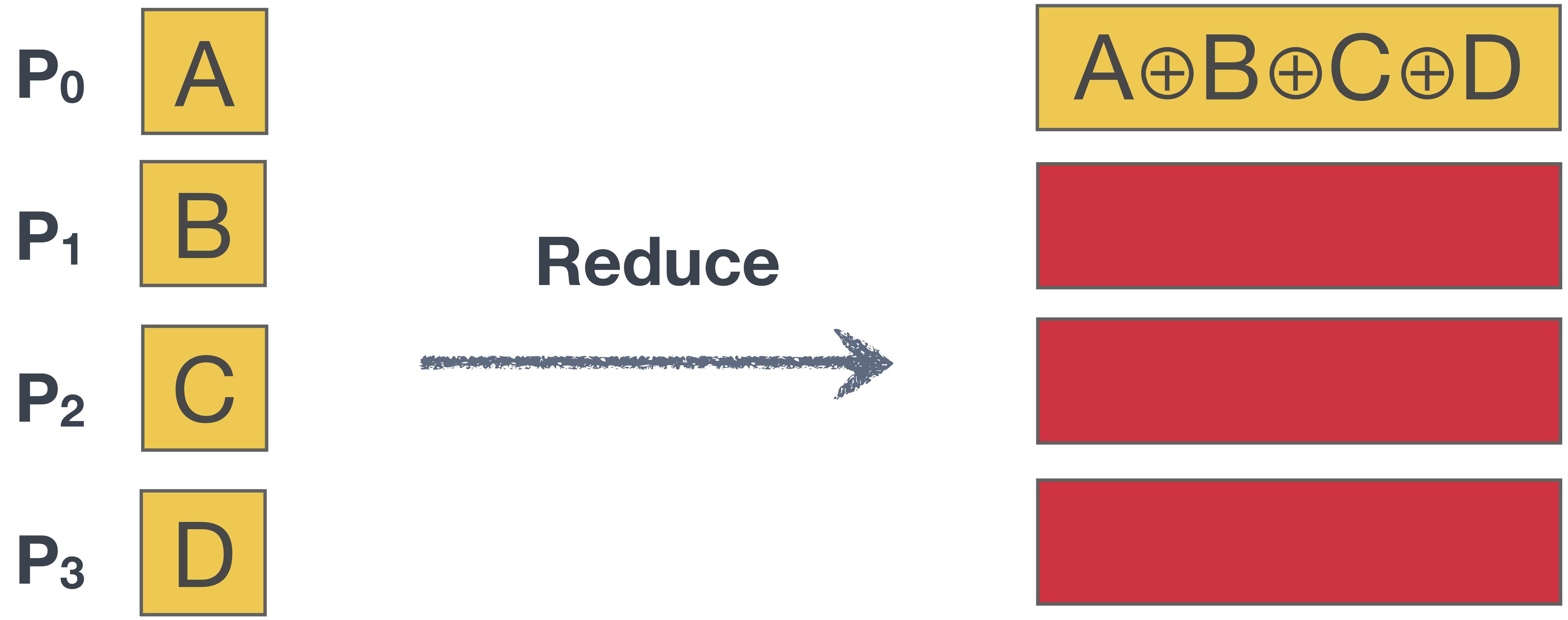
- All collective operations must be called by all processes in the communicator
- Eg,. MPI_Bcast is called by both the root process and all the processes that receive the broadcast
- Demo example on HPC cluster!



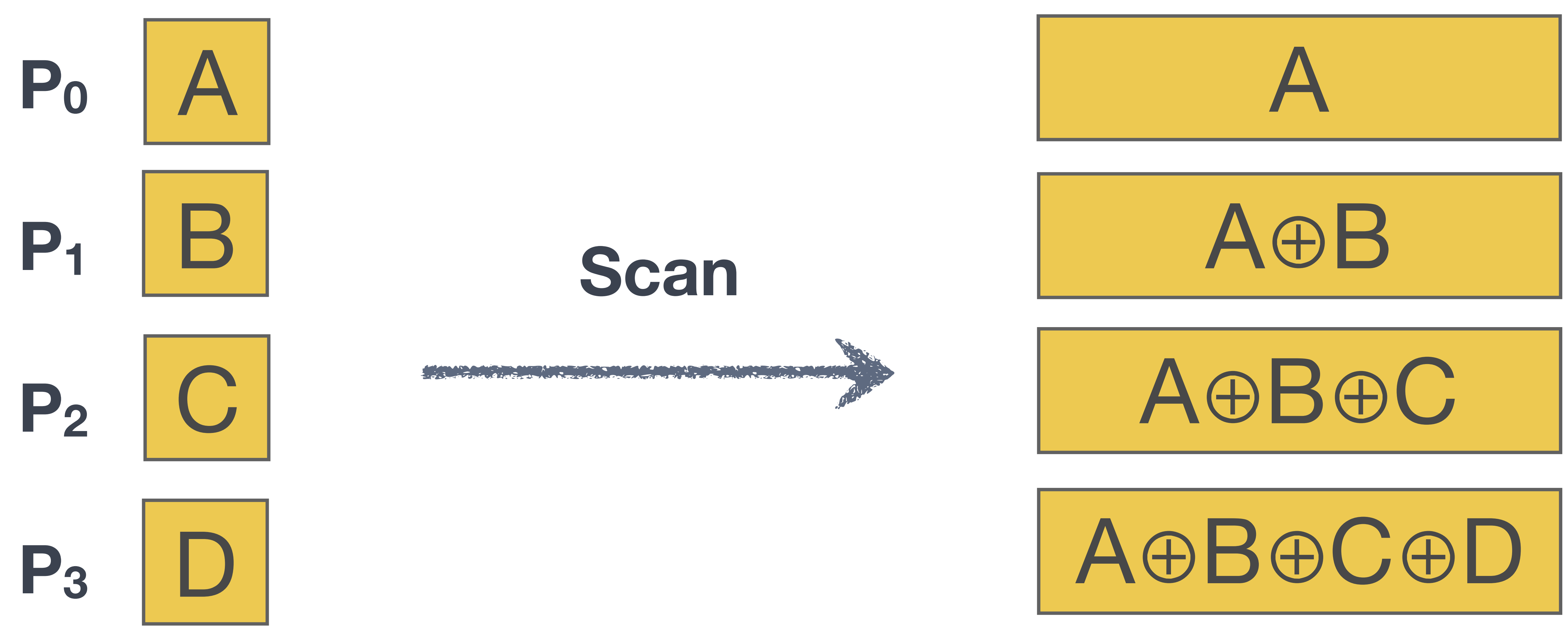
MPI_Allgather (Sbuffer, Scount, Stype, Rbuffer, Rcount, Rtype, comm)



MPI_Alltoall (Sbuffer, Scount, Stype, Rbuffer, Rcount, Rtype, comm)



MPI_Reduce (Sbuffer, Rbuffer, count, datatype, op, root, comm)



MPI_Scan (Sbuffer, Rbuffer, count, datatype, op, comm)

MPI Built-in Operations

MPI Reduction Operation	
`MPI_MAX`	maximum
`MPI_MIN`	minimum
`MPI_SUM`	sum
`MPI_PROD`	product
`MPI_LAND`	logical AND
`MPI_BAND`	bit-wise AND
`MPI_LOR`	logical OR
`MPI_BOR`	bit-wise OR
`MPI_LXOR`	logical XOR
`MPI_BXOR`	bit-wise XOR
`MPI_MAXLOC`	max value and location
`MPI_MINLOC`	min value and location

Other collective routines

- Many routines: **Gather, Gatherv, Allgather, Allgatherv, Reduce_scatter, Reduce, Allreduce, Scatter, Scatterv, Bcast, Scan**
- **All** = data to all processes
- **v** = allows data chunks to have variable sizes
- MPI-2 added **Alltoallw**, **Exscan**, inter-communicator variants for most routines