Lecture #13 - Distributed Memory and MPI

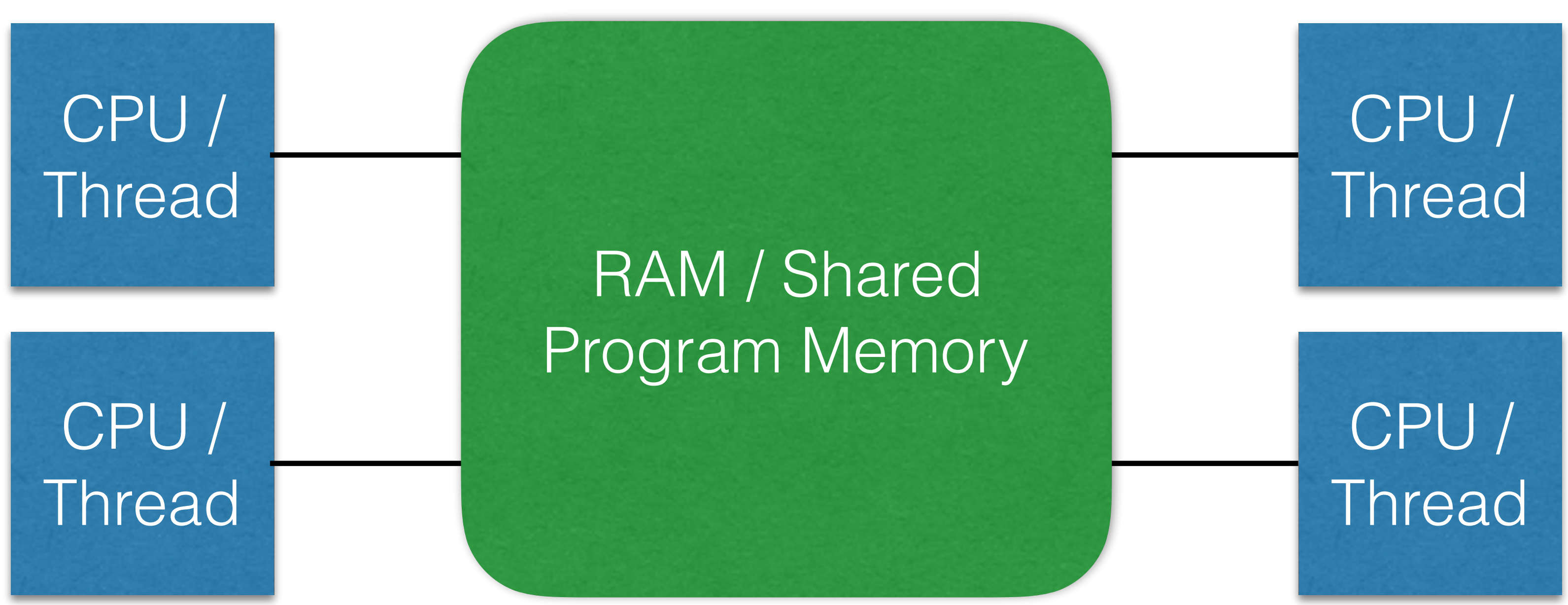
AMath 483/583

Parallelism Grain

- **Fine Grain**: parallelize at level of individual loops, splitting work for each loop between threads
- **Coarse Grain**: split problem into large pieces and have each thread deal with one piece
    - may need to sync info at some points
    - similar to MPI

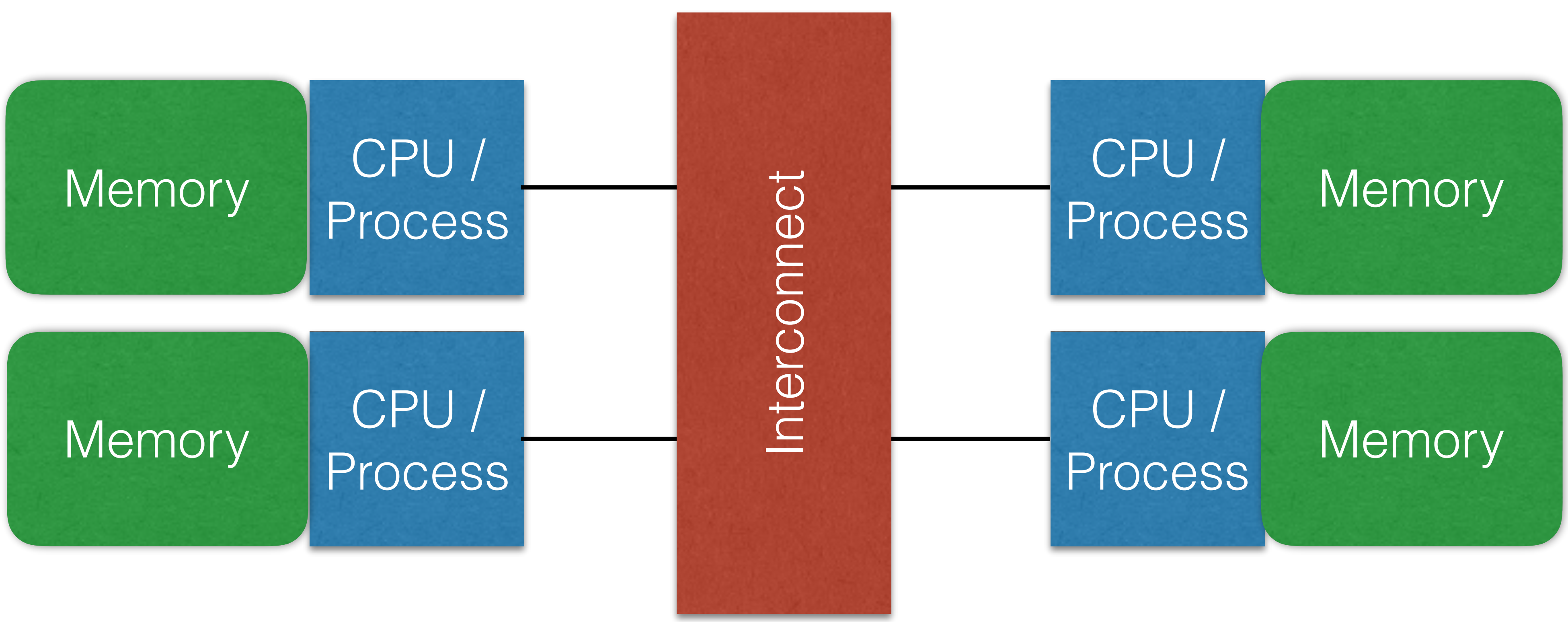
Shared Memory

- OpenMP - only shared memory environments
  - single address space, multiple threads



Distributed Memory

- Each processor / machine has separate memory
  - “processes” - each have separate address space
  - process communication must be explicit



MPI - Message Passing Interface

- **Message Passing**
    - SIMD - Same Instruction Multiple Data
    - “Parent program” manages memory by placing data in processes
    - Data *explicitly* sent to/from processes.
- `MPI` = de facto standard for distributed computing

MPI - Message Passing Interface

- **Implementations**
    - OpenMPI (www.open-mpi.org)
    - MPICH — Argonne National Lab
    - MPIICC — Intel
- “MPI Standard” implemented by above compilers

Hello World

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

int main(int argc, char** argv)
{
  MPI_Init(NULL,NULL);

  int num_procs;
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

  int proc_id;
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);

  printf("Hello from proc %d of %d.\n",
         proc_id, num_procs);

  MPI_Finalize();
}
```

Hello World

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

int main(int argc, char** argv)
{
  MPI_Init(NULL, NULL);

  int num_procs;
  MPI_Comm_size(MPI_COMM_W

  int proc_id;
  MPI_Comm_rank(MPI_COMM_

  printf(“Hello from proc
         proc_id, num_proc

  MPI_Finalize();
}
```

`MPI_Init(int*, char***)`

- Required by every MPI program
- Must be first call.
- Talk about arguments later. NULL is fine for now.

Hello

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

int main(int argc, char** a
{
  MPI_Init(NULL, NULL);

  int num_procs;
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

  int proc_id;
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);

  printf(“Hello from proc %d of %d.\n”,
         proc_id, num_procs);

  MPI_Finalize();
}
```

`MPI_Comm_size(MPI_Comm, int*)`

- Returns size of given “*communicator*”
- *Communicator* = group of procs
- `MPI_COMM_WORLD` = default constructed by MPI

Hello World

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char**
{
  MPI_Init(NULL, NULL);

  int num_procs;
  MPI_Comm_size(MPI_COMM_WOR

  int proc_id;
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);

  printf("Hello from proc %d of %d.\n",
         proc_id, num_procs);

  MPI_Finalize();
}
```

`MPI_Comm_rank(MPI_Comm, int*)`

- Returns "*rank*" of proc running this code
- Communicator automatically assigns *rank*
- Primary method for identifying procs

Hello World

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

int main(int argc, char** argv)
{
  MPI_Init(NULL, NULL);

  int num_procs;
  MPI_Comm_size(MPI_COMM_WO

  int proc_id;
  MPI_Comm_rank(MPI_COMM_W

  printf(“Hello from pro
         proc_id, num        );

  MPI_Finalize();
}
```

`MPI_Finalize()`

- Clean up MPI environment
- No MPI calls allowed after this

Hello World

```
#include
#include

int mai
{
  MPI_I

  int num_procs;
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

  int proc_id;
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);

  printf("Hello from proc %d of %d.\n",
         proc_id, num_procs);

  MPI_Finalize();
}
```

Most arguments are passed by reference:

- Give `MPI_Comm_size` the address of `num_procs`
- `MPI_Comm_size` sets the value pointed to by `num_procs` to the number of processes
- `num_procs` is now equal to # of procs

Compiling Hello World

- Compile using `mpicc`
- Execute using `mpiexec`: specify number of processes

```
$ cd uwhpsc-2016/lectures/lecture16
$ mpicc hello.c
$ mpiexec -n 4 ./a.out
```

Demo

MPI Hello World

Key Observation

- Every process runs the same program `hello.c`
- Manage which processes perform which tasks using their rank / process id.
- Separate programs, separate data.

Communicators

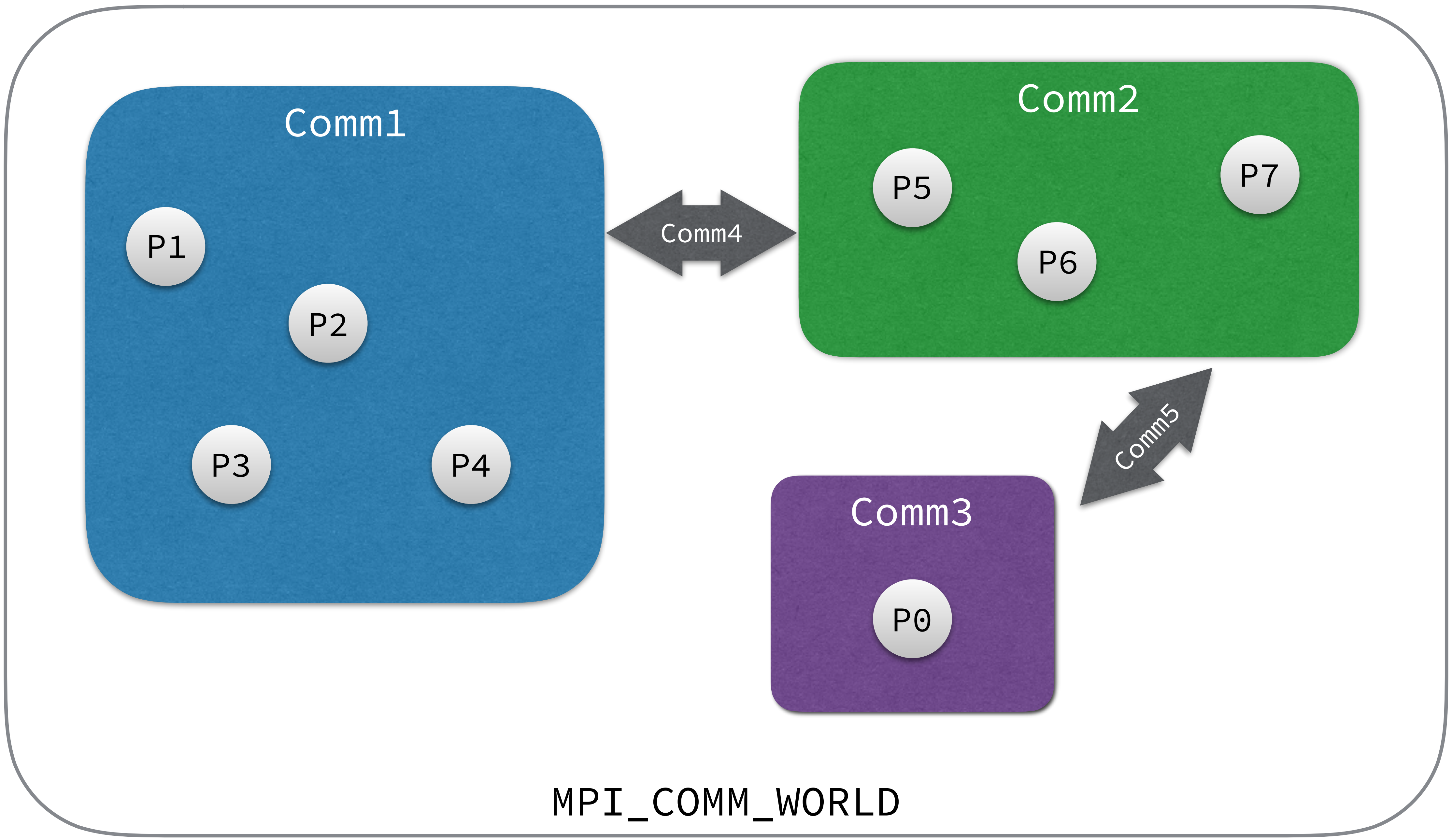
- `MPI_Comm` type:
  - all communication within group of processes
  - communication takes place in some context
  - `MPI_COMM_WORLD` — provided communicator that includes all processors by default

Communicators

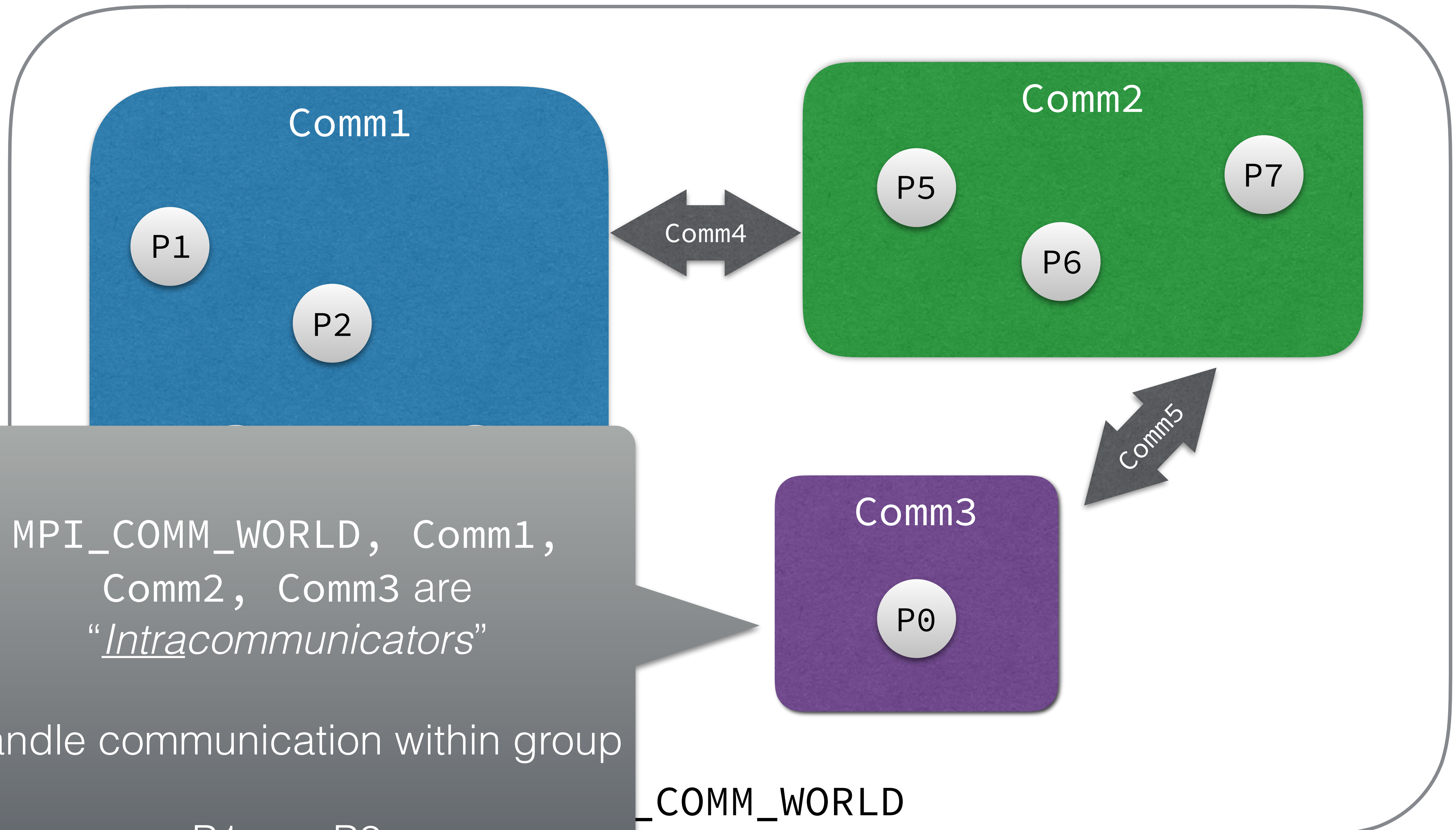
- MPI_C
  - all co … up of processes
  - com … n some context
  - MPI_COMM_WORLD — provided communicator that includes all processors by default

Sufficient for many applications

Communicators



Communicators

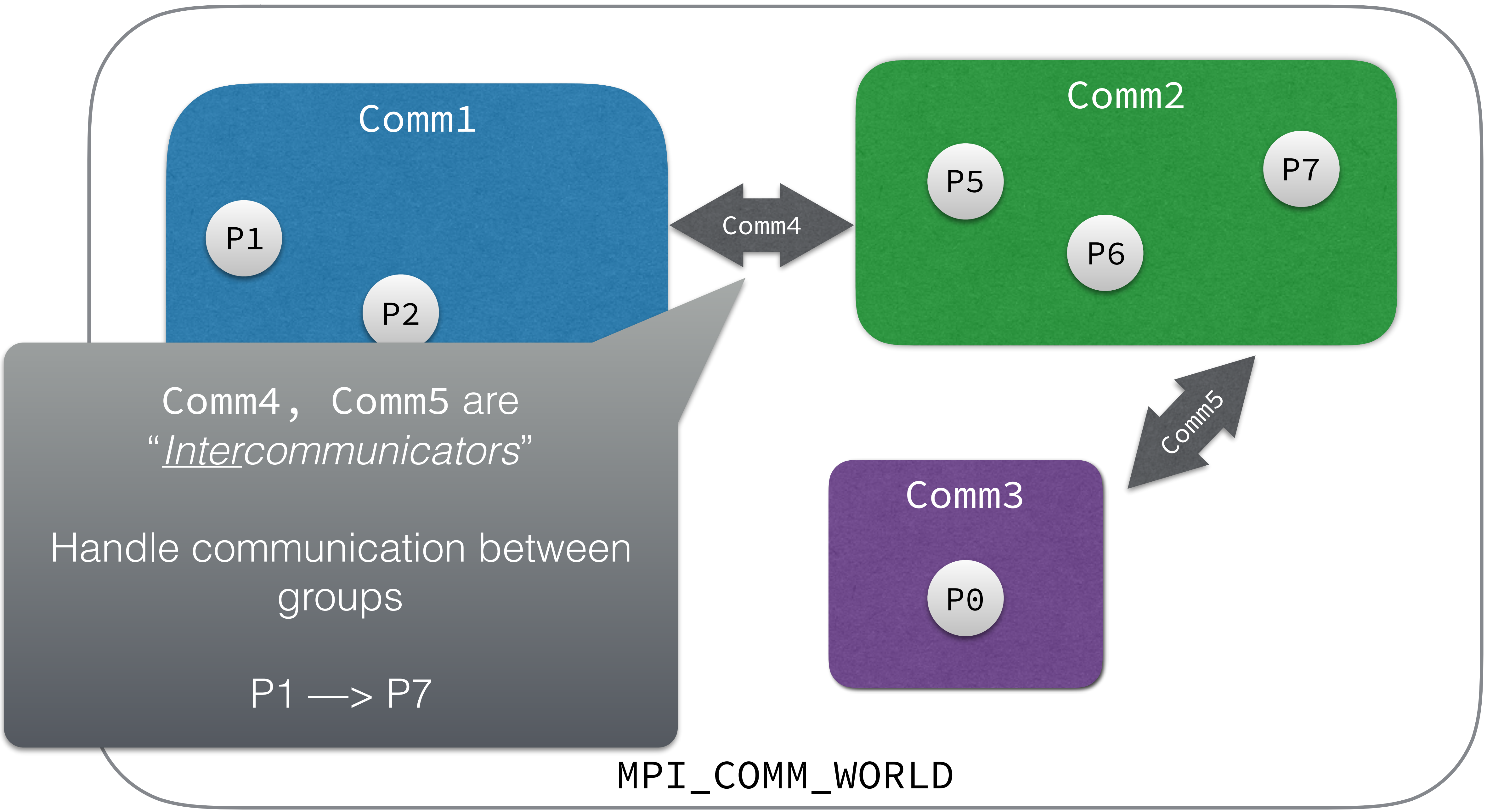


`MPI_COMM_WORLD, Comm1, Comm2, Comm3` are "*Intracommunicators*"

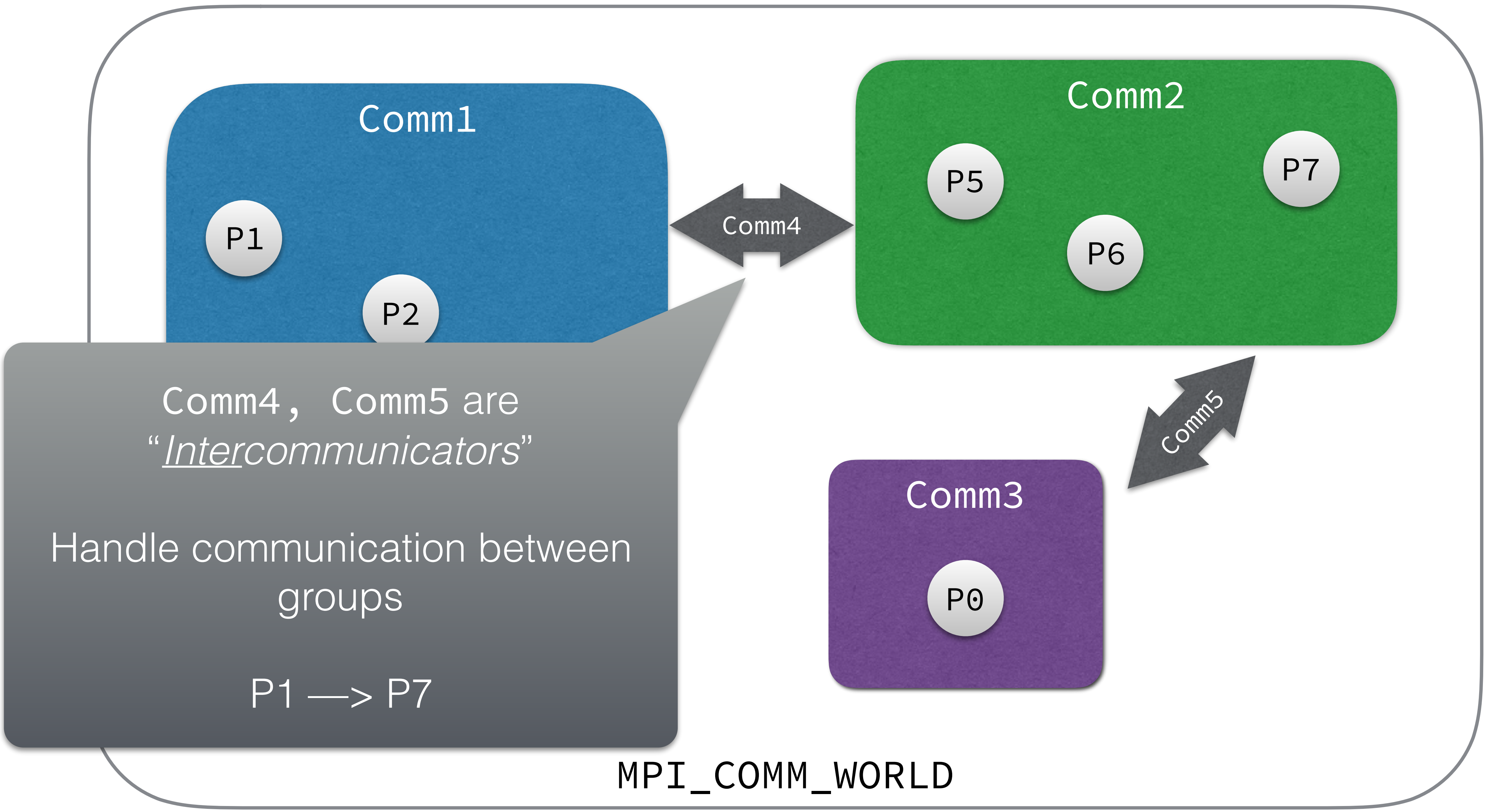
Handle communication within group

P1 —> P2

Communicators



Communicators

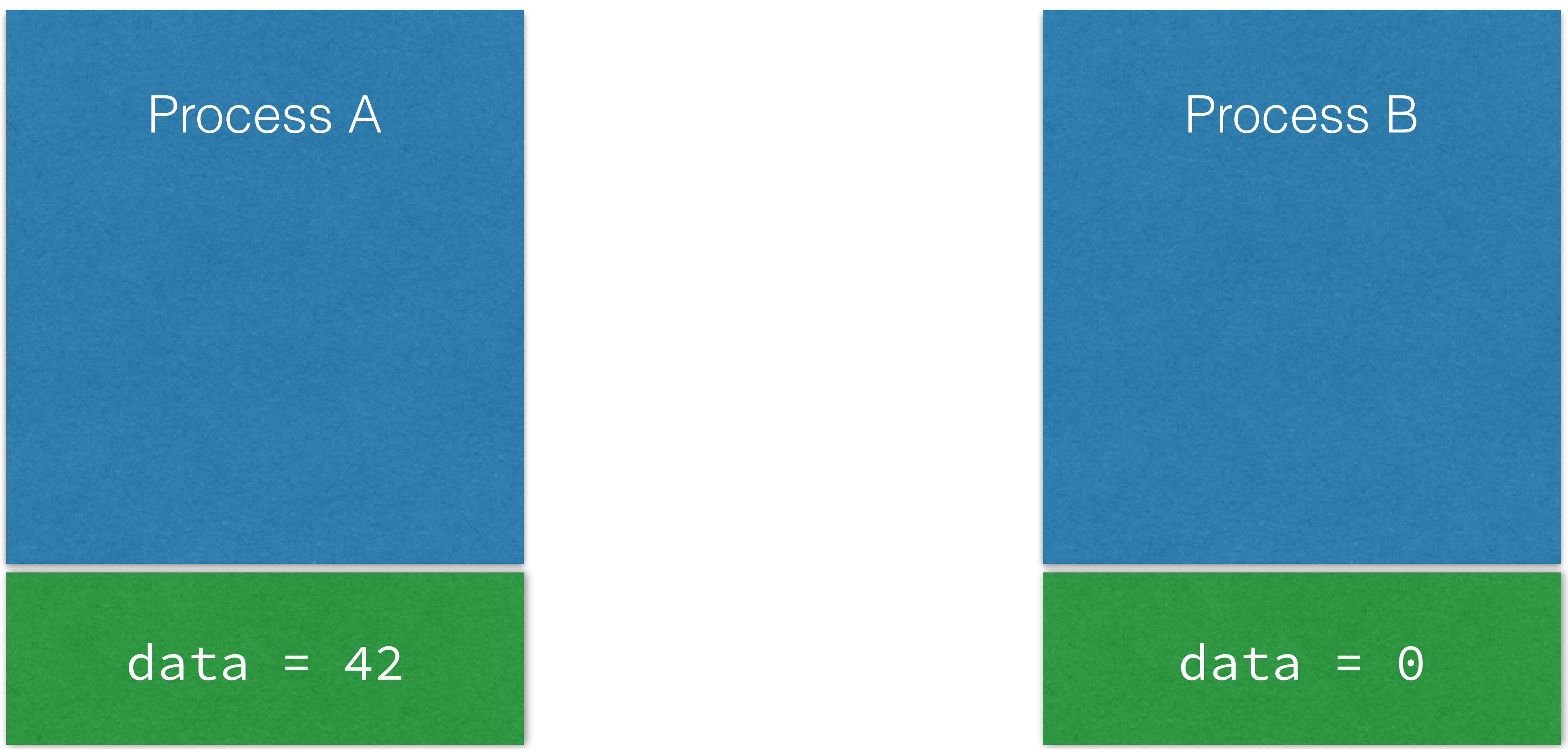


MPI

- Approx. 125 MPI functions
- Many programs can be written with the following eight and MPI_COMM_WORLD
  - `MPI_Init`
  - `MPI_Finalize`
  - `MPI_Comm_size`
  - `MPI_Comm_rank`
  - `MPI_Send`
  - `MPI_Recv`
  - `MPI_Bcast`
  - `MPI_Reduce`

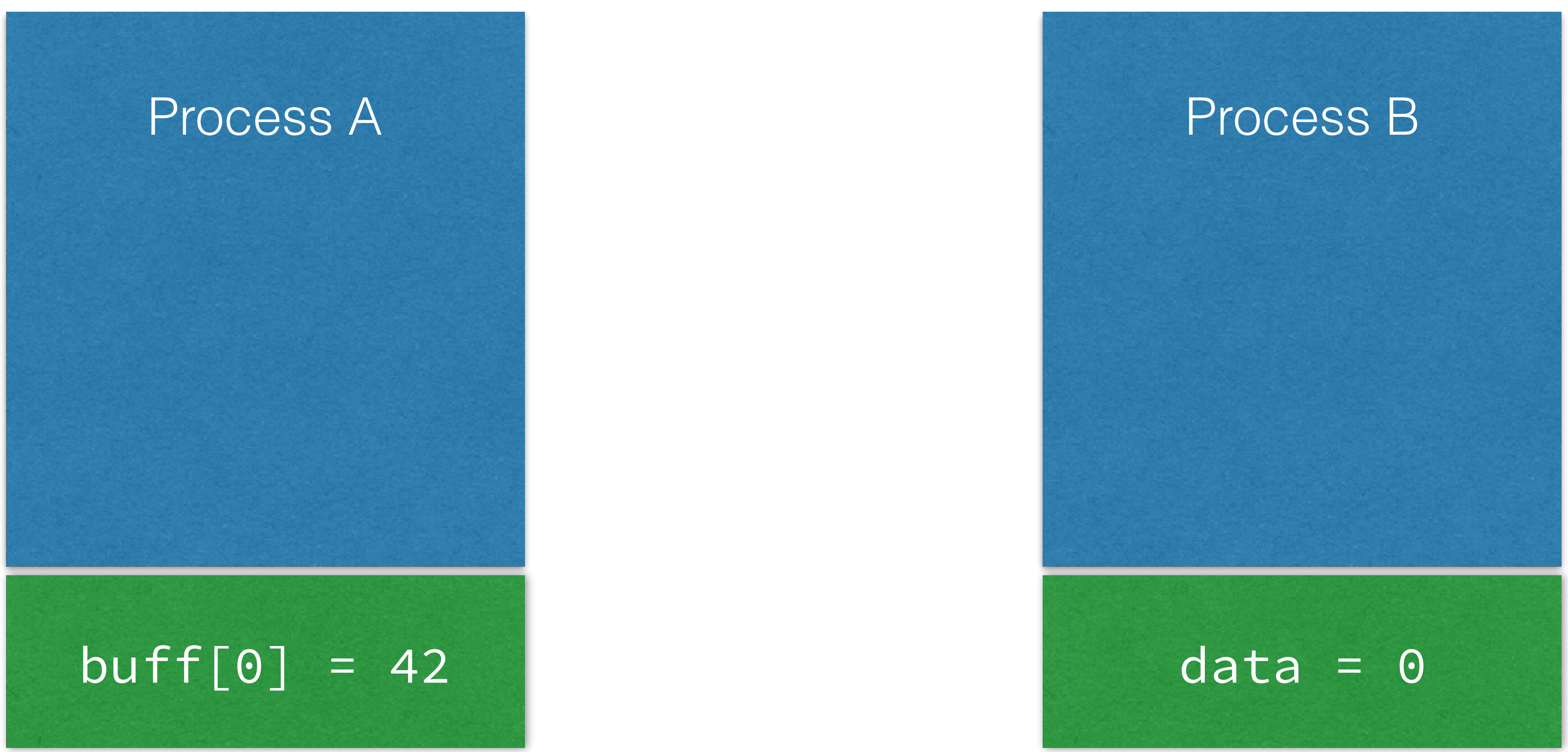
MPI_Send / MPI_Recv

- Basic concept behind MPI — one process sends data, another receives data



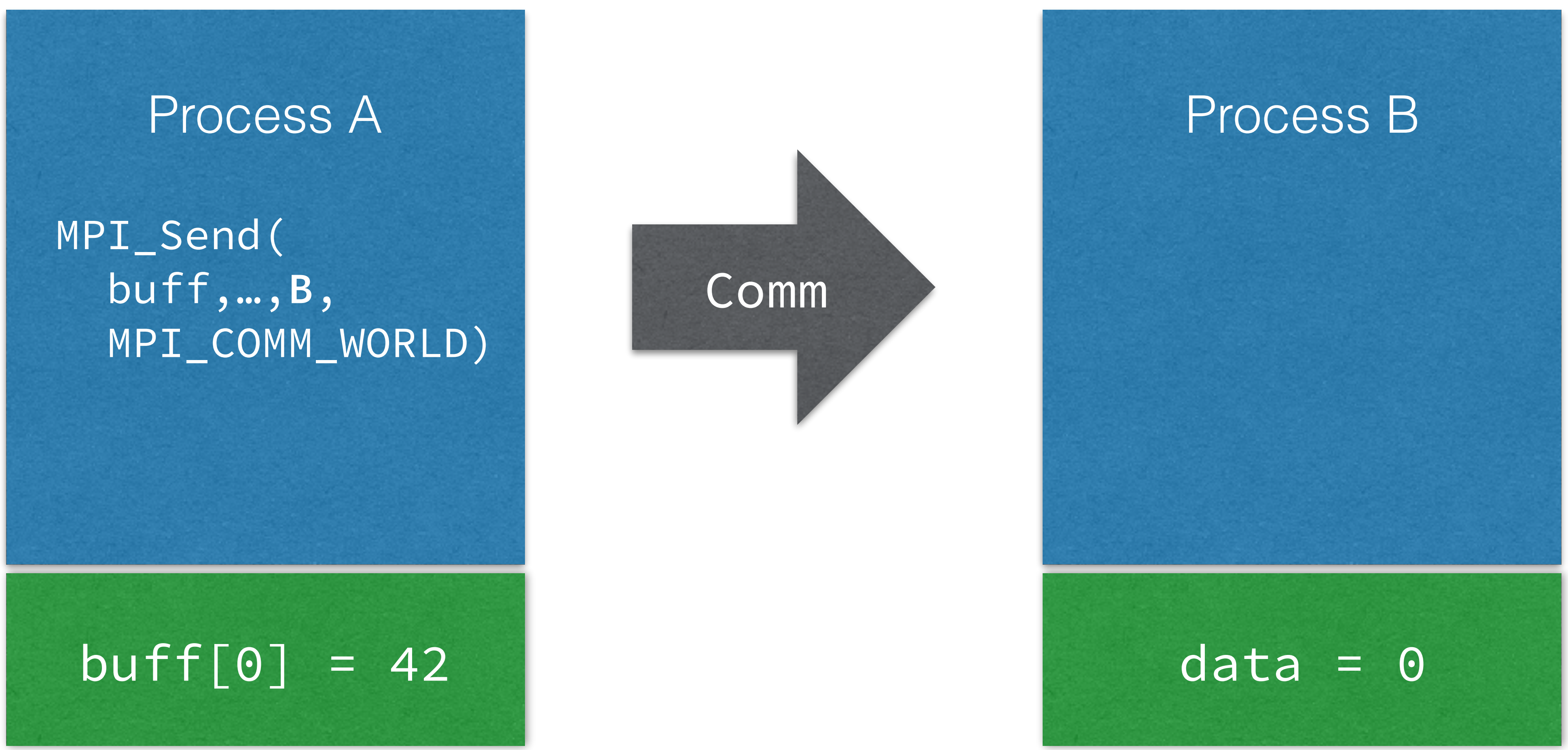
MPI_Send / MPI_Recv

- (1) Proc A packs up data to be sent into data buffer array / "*envelope*" (a pointer to data)



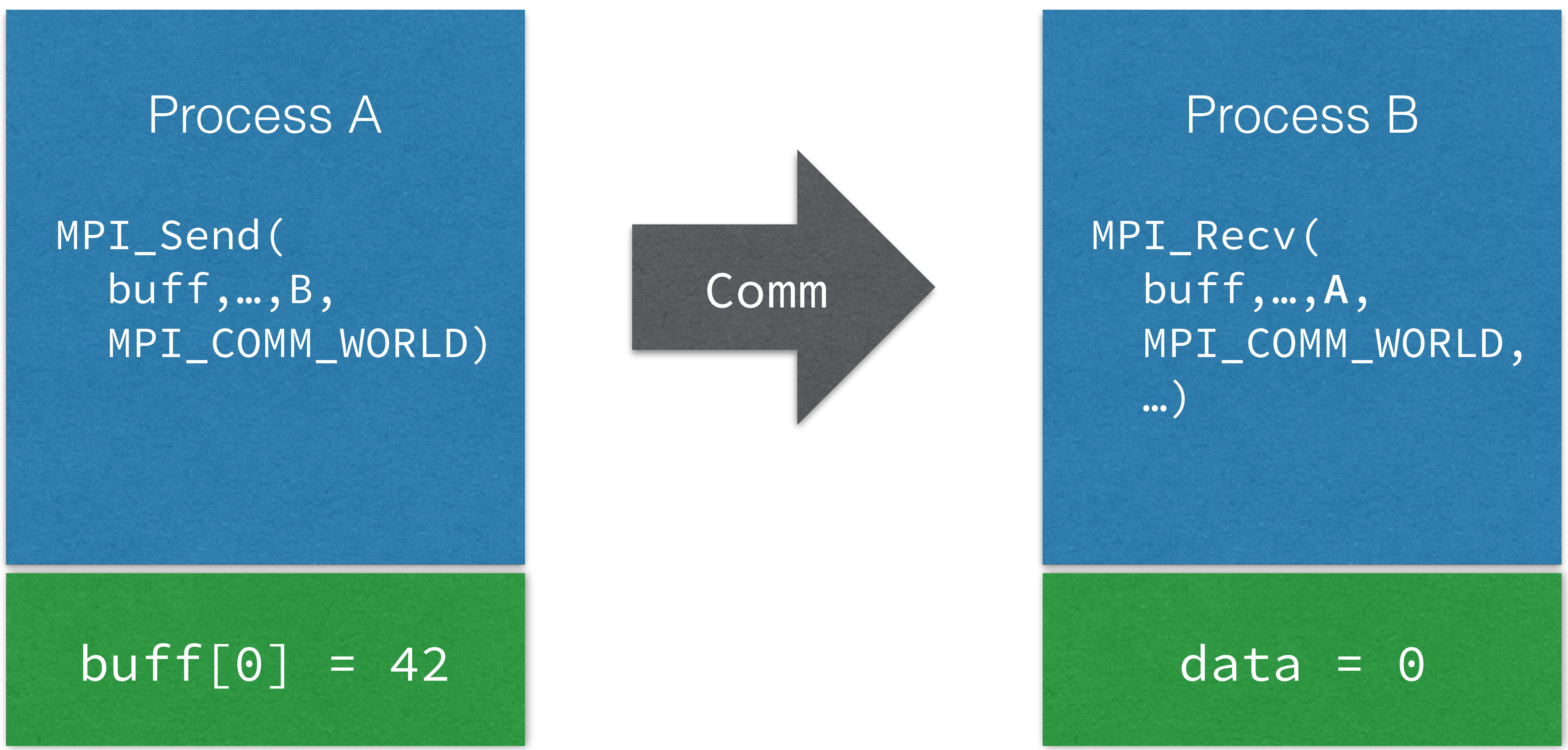
MPI_Send / MPI_Recv

- (2) Proc A announces to communicator it wants to send data buffer to Proc B with `MPI_Send(…)`



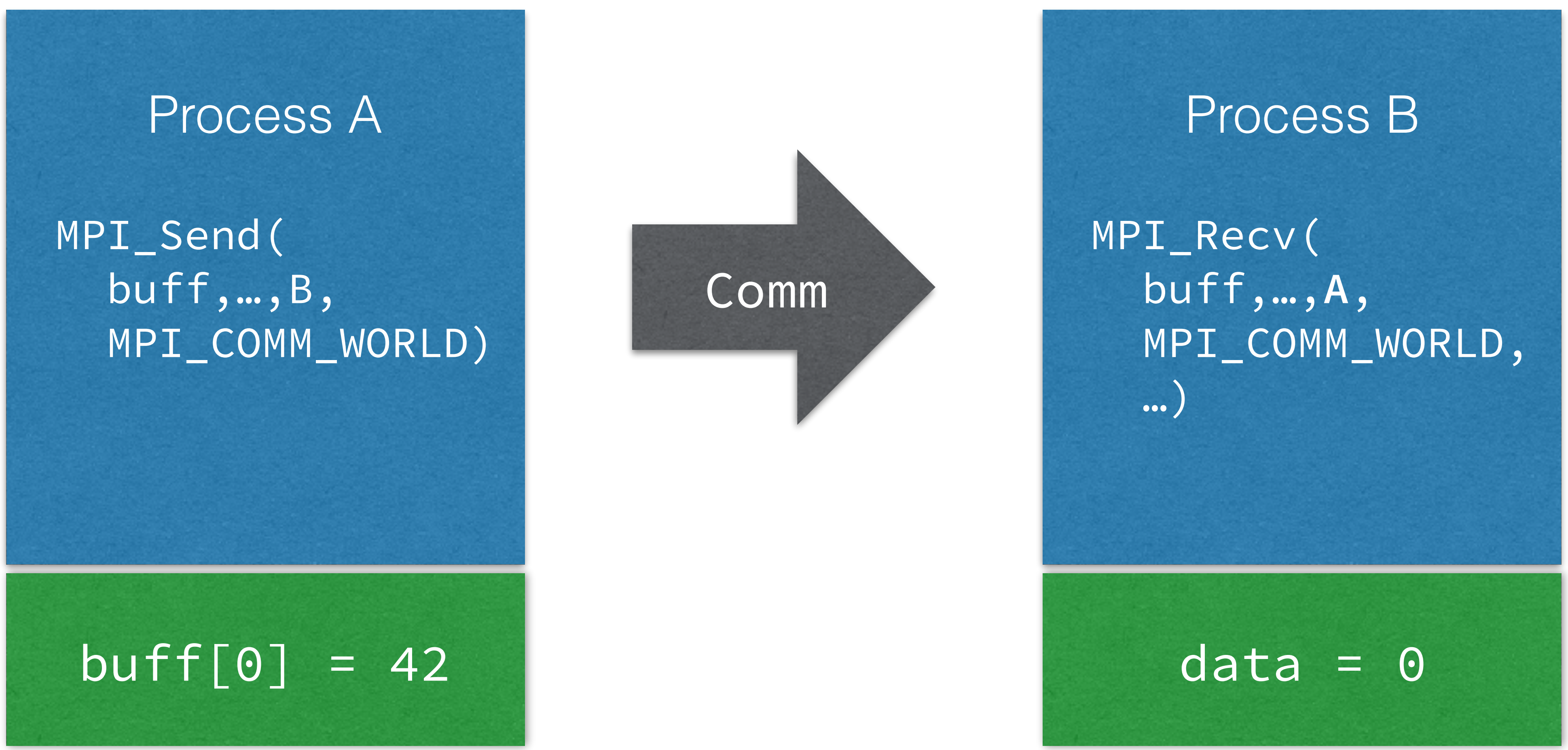
MPI_Send / MPI_Recv

- (3) Proc B acknowledges that it wants data buffer from Proc A using `MPI_Recv`



MPI_Send / MPI_Recv

- *Data is not sent until Proc A reaches* `MPI_Send()` *and Proc B reaches corresponding* `MPI_Recv()`



MPI_Send / MPI_Recv

- (4) Proc B receives data buffer and stores / uses information

MPI_Send / MPI_Recv

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

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

MPI_Send / MPI_Recv

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

```
MPI_Recv(
    * data,
    int count,
    int source,
    MPI_Comm communicator,
    MPI_Status* status)
```

Pointer to some location in memory (data buffer)

Size / length of data buffer

Type of data buffer (`MPI_INT, MPI_FLOAT`)

Target process / where to send data

Communicator. Interprets `destination`.

MPI_Send / MPI_Recv

```
MPI_Send(
    int count,
    int destination,
    int tag,
    MPI_Comm communicator)
```

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

Where to store incoming data.

Where to listen for incoming data.

Status message (for error handling)

MPI_Datatype

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
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_BYTE	char

Demo

- `send-receive.c`
- Proc 0 will send a data packet to Proc 1
- Proc 1 reports that it has the data.

Demo

`send-receive.c`

Example: Distributed Minimum

- Find minimum across large array
- **Strategy**:
  - Proc 0 gives chunk of array to each process
  - each process finds local minimum
  - process reports result back to Proc 0

Pointer Reminder

```
int a[8];
```

42	10	-24	5	0	2135	90	4
0x00	0x04	0x08	0x0c	0x10	0x14	0x18	0x1c

```
    a == 0x00;
 a[0] == 42;
&a[0] == 0x00;
 a[2] == -24;
&a[2] == 0x08;
```

Pointer Reminder

```
int a[8];
```

42	10	-24	5	0	2135	90	4
0x00	0x04	0x08	0x0c	0x10	0x14	0x18	0x1c

```
MPI_Send(&a[4], 2, …)
```

Demo

`distributed-min.c`

Example: Matmul

- AB = C (all N x N)

```
int i,j,k;
for (i=0; i<N; ++i) {
  for (j=0; j<N; ++j) {
    C[i*N+j] = 0;
    for (k=0; k<N; ++k)
      C[i*N+j] = A[i*N+k] * B[k*N+j];
  }
}
```