Lecture #14 - More MPI

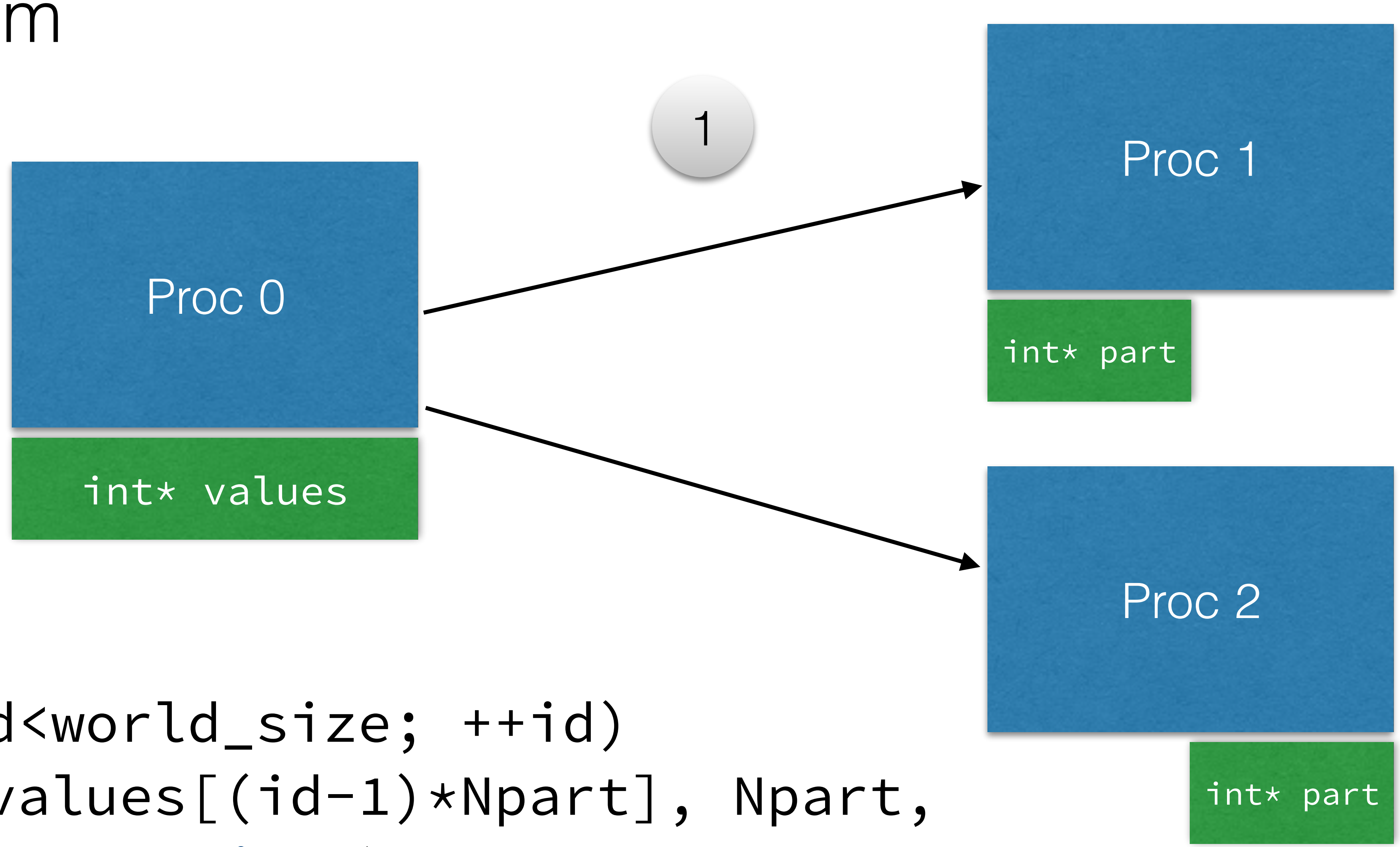
AMath 483/583

Announcements

- Homework #3 — Due next Friday
  - remember to pull corrections (Issue #5)
- Primary and Secondary References (finally) online for Week #7 (this week)
- Homework #2 — perhaps graded by tomorrow

Last Time

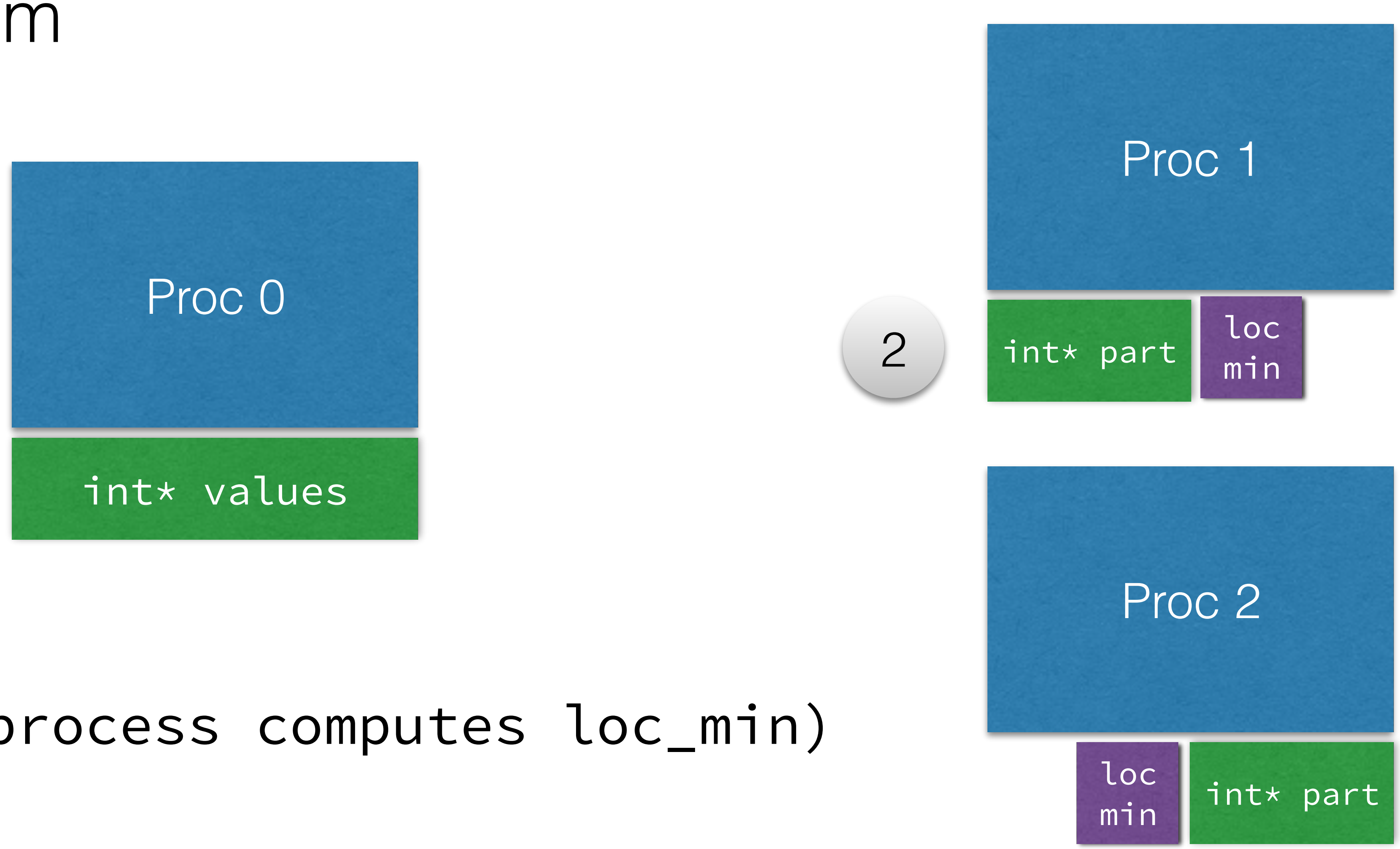
- `MPI_Send` / `MPI_Recv` - Application: distributed minimum




```
for (id=1; id<world_size; ++id)
  MPI_Send(&values[(id-1)*Npart], Npart,
           MPI_INT,id,…);
```

Last Time

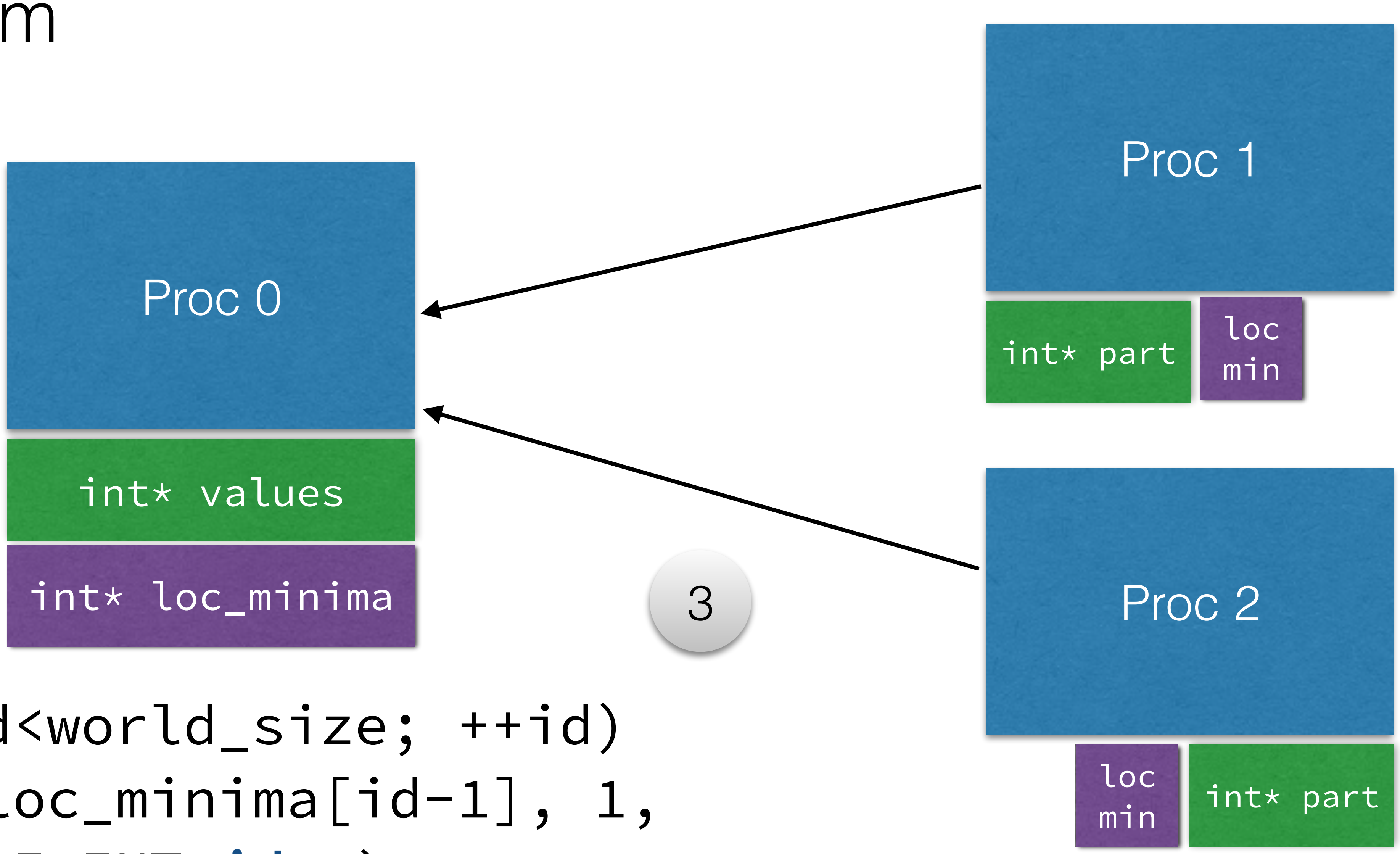
- `MPI_Send` / `MPI_Recv` - Application: distributed minimum



`(each other process computes loc_min)`

Last Time

- `MPI_Send` / `MPI_Recv` - Application: distributed minimum




```
for (id=1; id<world_size; ++id)
  MPI_Recv(&loc_minima[id-1], 1,
           MPI_INT,id,…);
```

Last Time

MPI implementation-dependent!

- `MPI_Send` / `MPI_Recv` — note:
  - both are "*blocking*" — processes will **not(*)** continue until corresponding `Send` / `Recv` is reached

```
if (rank == 0) {
    // big computation
    MPI_Send(data_buffer, buffer_length,
                 MPI_INT, 1, …)
} else if (rank == 1) {
   MPI_Recv(receive_buffer, len, MPI_INT, 0, …)
}
```

Last Time

- `MPI_Send` / `MPI_Recv` — note:
    - both are "*blocking*" — processes will **<u>not(*)</u>** continue until corresponding `Send` / `Recv` is reached

```
if (rank == 0) {
  MPI_Send(data_buffer, buffer_length,
                 MPI_INT, 1, …)
} else if (rank == 1) {
  // big computation
  MPI_Recv(receive_buffer, len, MPI_INT, 0, …)
}
```

Deadlock

- *(Not the next Marvel villain.)* What happens here?

```
if (rank == 0)
{
  int* foo, bar;
  MPI_Send(foo, N, TYPE, 1, …)
  // compute
  MPI_Recv(bar, M, TYPE, 1, …)
}
else if (rank == 1)
{
  int* a, b;
  MPI_Send(a, N, TYPE, 0, …)
  // compute
  MPI_Recv(b, M, TYPE, 0, …)
}
```

Deadlock

- *(Not the next Marvel villain.)* What happens here?

```
if (rank == 0)
{
  int* foo, bar;
  MPI_Send(foo, N, TYPE, 1, …)
  // compute
  MPI_Recv(bar, M, TYPE, 1, …)
}
else if (rank == 1)
{
  int* a, b;
  MPI_Send(a, N, TYPE, 0, …)
  // compute
  MPI_Recv(b, M, TYPE, 0, …)
}
```

Send data to Proc 1.

Wait until it receives until proceeding.

Meanwhile…

Send data to Proc 0.

Wait until it receives until proceeding.

Deadlock

- *(Not the next Marvel villain.)* What happens here?

```
if (rank == 0)
{
  in     foo, bar;
  M      nd(f      N, TYPE, 1, …)

                   M, TYPE, 1, …)


                      1)


              ;
  MPI_Se      a, N, TYPE, 0, …)
  // compute
  MPI_Recv(b, M, TYPE, 0, …)
}
```

Both processes are
"stuck" in waiting position.

Deadlock

Send data to Proc 1.

Wait until it receives until proceeding.

Meanwhile…

Send data to Proc 0.

Wait until it receives until proceeding.

Deadlock - Solution #1

- Impose non-deadlocking ordering of send/recv

```
if (rank == 0)
{
  int* foo, bar;
  MPI_Send(foo, N, TYPE, 1, …)
  // compute
  MPI_Recv(bar, M, TYPE, 1, …)
}
else if (rank == 1)
{
  int* a, b;
  MPI_Recv(b, M, TYPE, 0, …)
  MPI_Send(a, N, TYPE, 0, …)
  // compute
 }
```

Accept Proc 0's request before making own.

Deadlock - Solution #2

- Use non-blocking variants: `ISend` / `IRecv`

```
if (rank == 0)
{
  int* foo, bar;
  MPI_Isend(foo, N, TYPE, 1, …)
  // compute
  MPI_Recv(bar, M, TYPE, 1, …)
}
else if (rank == 1)
{
  int* a, b;
  MPI_Send(a, N, TYPE, 0, …)
  // compute
  MPI_Recv(b, M, TYPE, 0, …)
}
```

Send data to Proc 1.

Initiate the send and continue executing code.

Send data to Proc 0.

This proc will still wait until Proc 0 receives until proceeding.

MPI_Isend / MPI_Irecv

```
MPI_Isend(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator,
    MPI_Request* request)
```

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

MPI_Isend / MPI_Irecv

```
MPI_Isend(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator,
    MPI_Request* request)
```

```
MPI_Irecv(
    void* data
```

`MPI_Request`

Identifies communication operations and matches the operation that initiates the communication with the operation that terminates it.

MPI_Wait

- `MPI_Irecv()` — What's wrong with this situation?

```
MPI_Request req;
MPI_Irecv(data, N, …, &req);

// … compute some things …

result = foo(data, N);
```

MPI_Wait

- What's wrong with this situation?

```
MPI_Request req;
MPI_Irecv(data, N, …, &req);

// … compute some things

result = foo(data, N);
```

Transfer of "data" may not have completed by the time foo() is called!

MPI_Wait

- What's wrong with this situation?

```
MPI_Request req;
MPI_Irecv(data, N, …, &req);

// … compute some things …

MPI_Wait(&req, MPI_STATUS_IGNORE);
result = foo(data, N);
```

Make sure the associated non-blocking call finishes

Keep track of call using `MPI_Request`

`MPI_Wait`

- An `MPI_Irecv(…)` immediately followed by an `MPI_Wait(…)` is equivalent to an `MPI_Recv(…)`

- **Caution**: `MPI_Send` may not block on some implementations!
  - `MPI_Ssend()` — guaranteed blocking
  - See [MPI Send Modes]() and [Differences Between Send() and Ssend()]()

Demo

Deadlock: `deadlock.c`

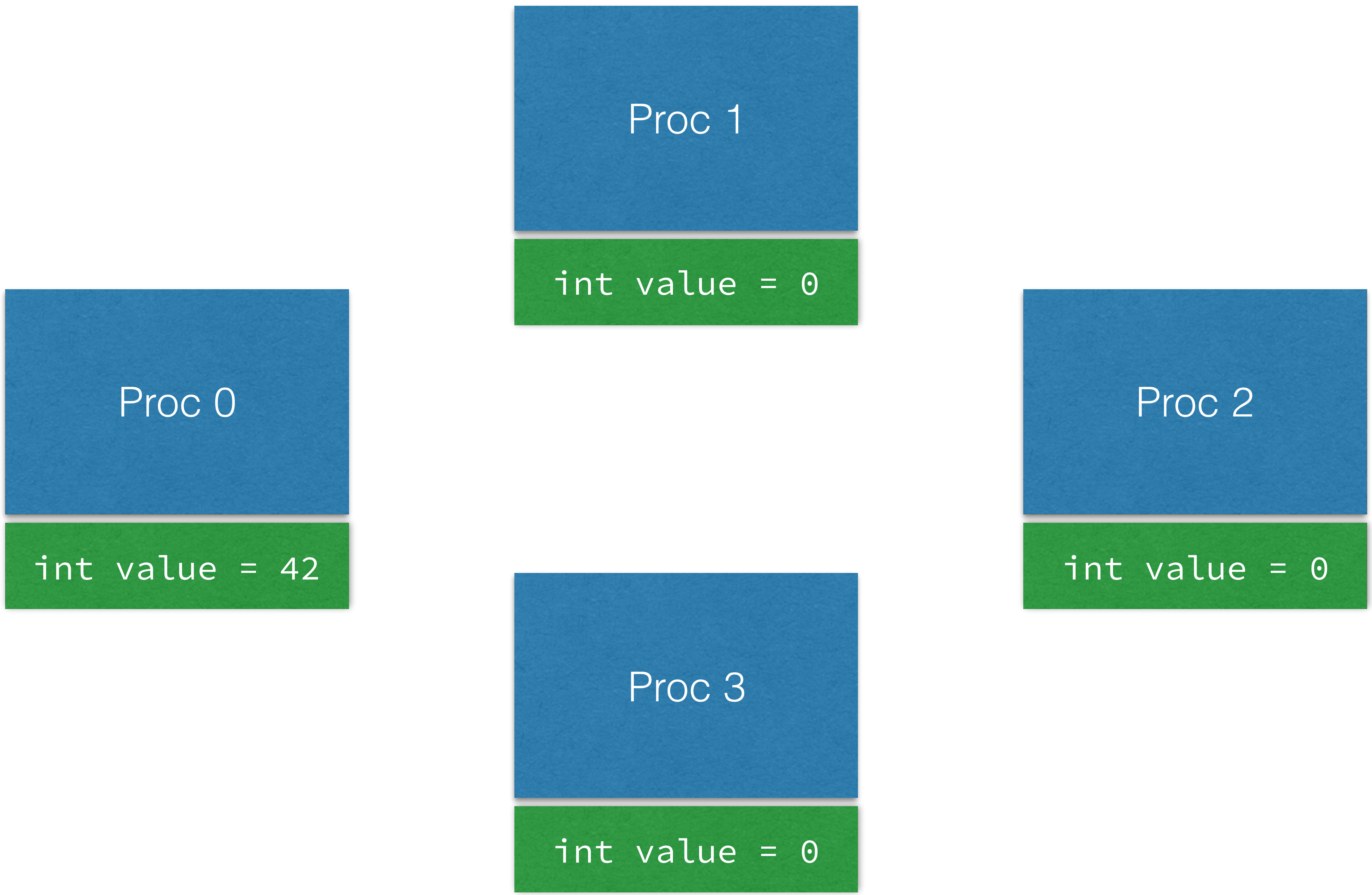
Latency Hiding

- The point of non-blocking calls is to hide latency
  - data transfer is expensive
  - waste of time to wait if you can compute something in the meantime

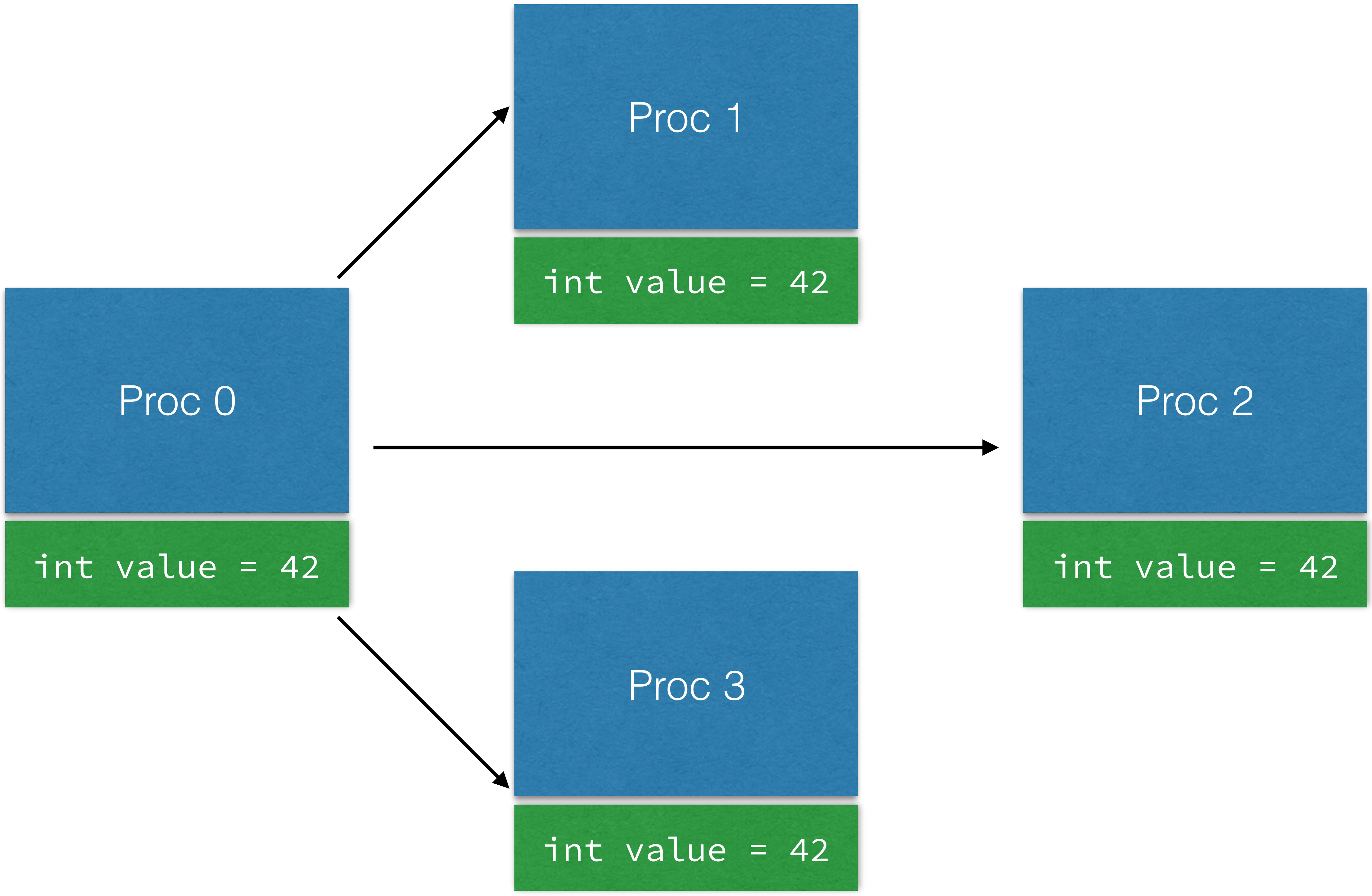
MPI_Bcast

- "Broadcasting" data to all processes on communicator
- "*Collective Operation*" — every process takes part in communication
  - same data
  - different "source" data
  - one-to-all

MPI_Bcast


Proc 1
int value = 0
Proc 0
int value = 42
Proc 2
int value = 0
Proc 3
int value = 0

MPI_Bcast



Proc 1
int value = 42
Proc 0
int value = 42
Proc 2
int value = 42
Proc 3
int value = 42

MPI_Bcast

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    MPI_Comm communicator)
```

MPI_Bcast

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    MPI_Comm communicator)
```

Pointer to some location in memory (data buffer)

Size / length of data buffer

Type of data buffer (`MPI_INT, MPI_FLOAT`)

Process broadcasting the data.

Communicator. Every process within will obtain data buffer.

MPI_Bcast

Pointer to some location in memory (data buffer)

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    MPI_Comm communicator)
```

Process broadcasting the data.

- Each proc has the variable `data` (and alloc, if necessary)
- Proc source has version of `data` to be broadcast
- Other procs have their `data` "filled in"

Demo

`MPI_Bcast`

Begs the question…

- *…why not create copies of the data within the code? (Each proc runs same program.)*
- **Answer**: the decision to globally share data can be made at runtime

Reduce

- **Reduce** — common functional programming op
- Given: `list = [1,2,3,4]`

  `reduce(+, list) = 1 + 2 + 3 + 4 = 8`

  `reduce(*, list) = 1 * 2 * 3 * 4 = 24`
- `MPI` – *"Collective operation"*

MPI_Reduce

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

MPI_Reduce

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

Elements to send to each process for reduction

Where to receive reduction result in proc `root`.

Size of `recv_data`

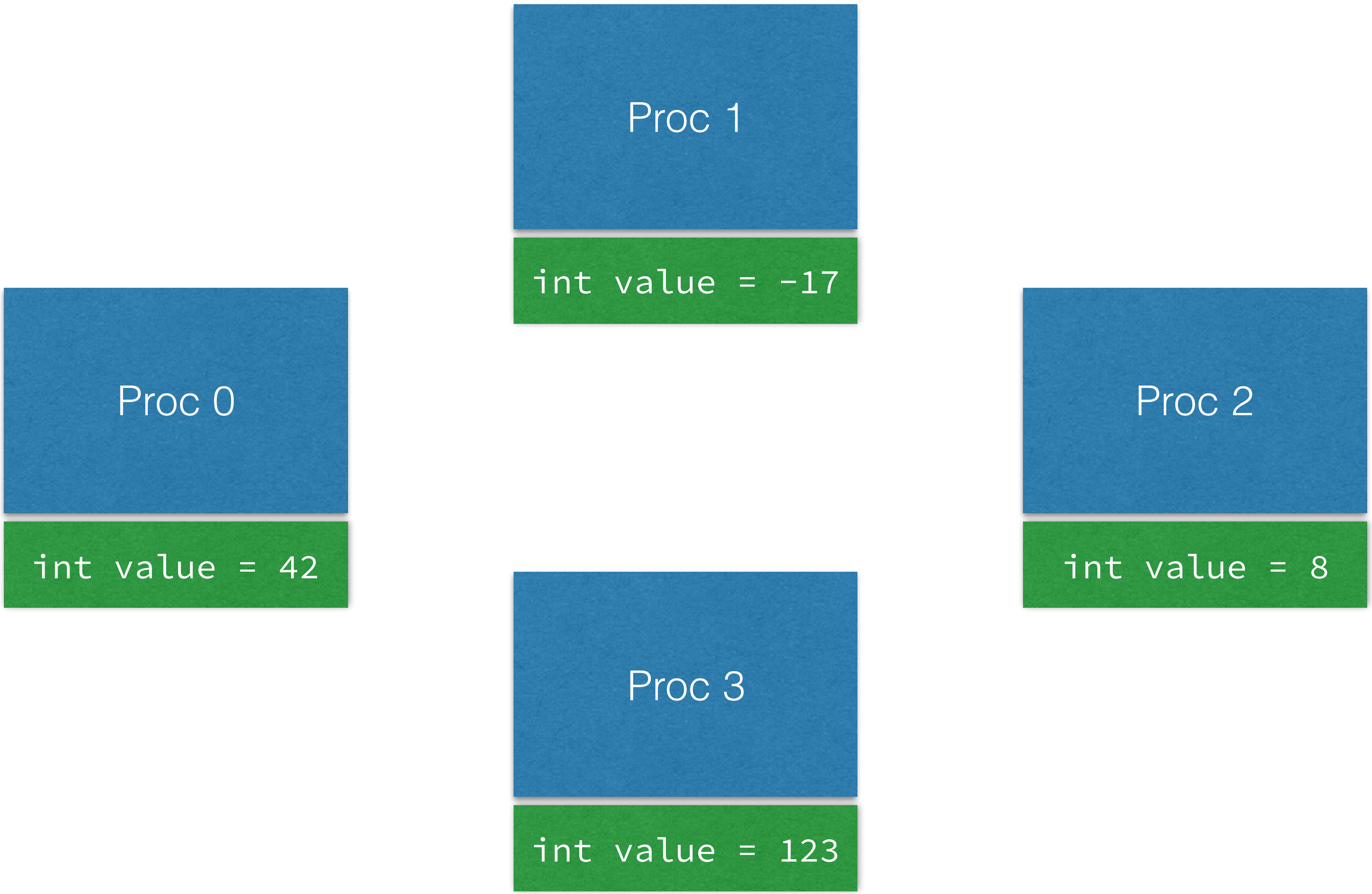
Reduction operation

Sending process.

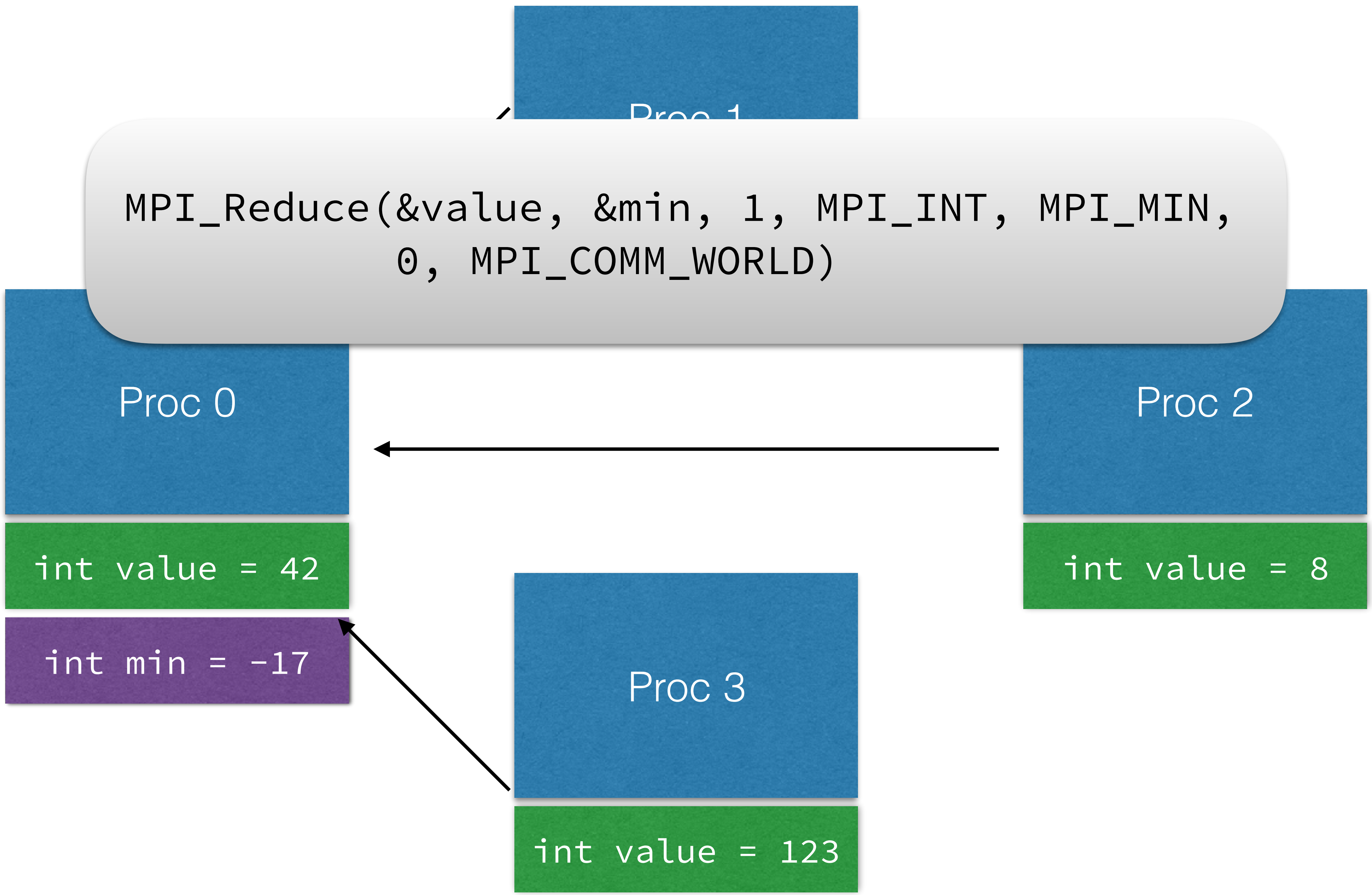
Built-in Reduction Ops

- `MPI_MAX` - Returns the maximum element.
- `MPI_MIN` - Returns the minimum element.
- `MPI_SUM` - Sums the elements.
- `MPI_PROD` - Multiplies all elements.
- `MPI_LAND` - Performs a logical and across the elements.
- `MPI_LOR` - Performs a logical or across the elements.
- `MPI_BAND` - Performs a bitwise and across the bits of the elements.
- `MPI_BOR` - Performs a bitwise or across the bits of the elements.
- `MPI_MAXLOC` - Returns the maximum value and the rank of the process that owns it.
- `MPI_MINLOC` - Returns the minimum value and the rank of the process that owns it.

MPI_Reduce


Proc 1
int value = -17
Proc 0
int value = 42
Proc 2
int value = 8
Proc 3
int value = 123

MPI_Reduce



MPI_Reduce(&value, &min, 1, MPI_INT, MPI_MIN,
0, MPI_COMM_WORLD)
Proc 0
int value = 42
int min = -17
Proc 2
int value = 8
Proc 3
int value = 123

Demo

Rewriting distributed min

Next Time

- `MPI_Scatter / MPI_Gather`
- **Application**: distributed (adaptive) integration
- Load balancing
- (Time Permitting) Go over Homework #2 solutions