COL380

Introduction to
Parallel & Distributed Programming

Send/Receive

Blocking calls (Progress condition is satisfied on return)

int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

MATCHING (Per context)

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

s block of memory
number of items in messag
MPI_Datatype of each iten
rank of recipient
integer "message identifier

- message contents — memory buffer to store received message
- count — space in buffer, overflow error if too small
- message type — type of each item
- source — sender's rank (or MPI_ANY_SOURCE)
- tag — message identifier (or MPI_ANY_TAG)
- communicator
- status — information about message received

Eager vs Rendezvous

Eager

- Send-stub packetizes and transmits
  (May save a local message copy)
- Send-stub signals Done
- Recv-stub continuously accepts
- Delivered when Recv call matches

Envelope =
<source,dest,tag,communicator>

Rendezvous

- Send-stub transmits envelope info
  (May save local message copy)
- Recv-stub continuously accepts envelope info
- Recv-stub may signal OK (if it has space)
  Or, wait for matching Recv call to be made
- Recv-stub sets up "RDMA" with Send-stub
- Data transmitted
- Recv-stub signals Done
- Send-stub signals Done

Subodh Kumar

Send/Recv Synchronization

- Blocking
  - Send returns after some progress guarantee
    - Receive completed?
    - Synchronization (up to network delay)
- Immediate
  - Send returns with no progress guarantee
  - Receiver may also proceed immediately (message arrives later)

Send Semantics

- Standard mode: MPI_Send
  - ➡ implementation dependent
- Buffered mode MPI_Bsend
  - ➡ MPI saves a copy of message, Receiver can post later
  - ➡ User provided buffer — See MPI_Buffer_attach
- Synchronous mode MPI_Ssend
  - ➡ Will complete only once a matching receive has started
- Ready mode MPI_Rsend
  - ➡ Send may start only if a matching receive has already been called
  - ➡ Helps performance

- MPI_Send/MPI_Recv are blocking
  - ➡ Recv blocks until output buffer is filled
  - ➡ Send blocks until some 'progress'

Subodh Kumar

Message Semantics

- In order (per pair and tag)
  - ➡ Multi-threaded applications need to be coordinated
- Progress
  - ➡ For a matching send/Recv pair, at least one of these two will complete
- Fairness not guaranteed
  - ➡ A Send or a Recv may starve because all matches are satisfied by others
- Resource limitation can cause deadlocks
- Ready/Synchronous sends requires the least resources
  - ➡ Also used for debugging

Example

If (rank == 0):

```
    Send(sbuffer0, to 1);
    Recv(rbuffer0, from 1);
else:
    Send(sbuffer1, to 0);
    Recv(rbuffer1, from  0);
```

match

match

Deadlock
if neither send can copy out its sbuffer

```
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 calls

Returns even before buf copied out; caller must not use.

Example

```
If (rank == 0):
        MPI_Isend(sbuffer0, to 1);
        MPI_Irecv(rbuffer0, from 1);
        Wait for earlier calls to finish
else:
        MPI_Isend(sbuffer1, to 0);
        MPI_Irecv(rbuffer1, from  0);
        Wait for earlier calls to finish
```

Will not deadlock

Detecting Completion

- MPI_Wait(&request, &status)
  - status similar to MPI_recv
  - Blocks as per the blocking version's semantics
    - Send: message was copied out, Recv was started, etc.
    - Recv: Wait for data to fill
  - Request is freed as a side-effect
- MPI_Test(&request, &flag, &status)
  - Non-blocking poll
  - flag indicates whether operation is complete
  - Request is freed as a side-effect

Use MPI_Request_get_status to retain request

Later MPI_Request_free

Detecting Completion

- MPI_Wait(&request, &status)
  - status similar to MPI_recv
  - Blocks as per the blocking version's semantics
    - Send: message was copied out, Recv was started, etc.
    - Recv: Wait for data to fill
  - Request is freed as a side-effect
- MPI_Test(&request, &flag, &status)
  - Non-blocking poll
  - flag indicates whether operation is complete
  - Request is freed as a side-effect

Also see:

MPI_Waitany, MPI_Waitall, MPI_Waitsome

MPI_Testany, MPI_Testall, MPI_Testsome

Point to Point Basics

- Send - Recv is point-to-point
  - ➡ Call-to-call matching
  - ➡ Integer tag to control matching
  - ➡ Wildcard matching: MPI_ANY_SOURCE and MPI_ANY_TAG
- Recv buffer must contain enough space for message
  - ➡ Receiving fails otherwise
  - ➡ Can query the actual count received (MPI_Get_count)
    - Send determines the actual number sent
  - ➡ type parameters determines data structure ↔ message buffer copying

MPI Data types

MPI type	C type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG	signed long long int
MPI_SIGNED_CHAR	signed char
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_WCHAR	wchar_t
MPI_BYTE	

MPI Data types

- MPI_CHAR — signed char
- MPI_SHORT — signed short int
- MPI_INT — signed int
- MPI_LONG — signed long int
- MPI_LONG_LONG_INT — signed long long int
- MPI_LONG_LONG — signed long long int
- MPI_SIGNED_CHAR — signed char
- 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_WCHAR — wchar_t
- MPI_BYTE

Objects of type MPI_Datatype

- MPI does not understand language's layout (struct, e.g.)
  - ➡ Too system architecture dependent

MPI_INT, MPI_FLOAT ..

- Typemap:
  - ➡ (type_0, disp_0), ..., (type_n, disp_n)
  - ➡ i^{th} entry is of type_i and starts at byte base + disp_i

- MPI does not understand language's layout (struct, e.g.)
  - ➡ Too system architecture dependent

MPI_INT, MPI_FLOAT ..

- Typemap:
  - ➡ (type_0, disp_0), ..., (type_n, disp_n)
  - ➡ i^{th} entry is of type_i and starts at byte base + disp_i

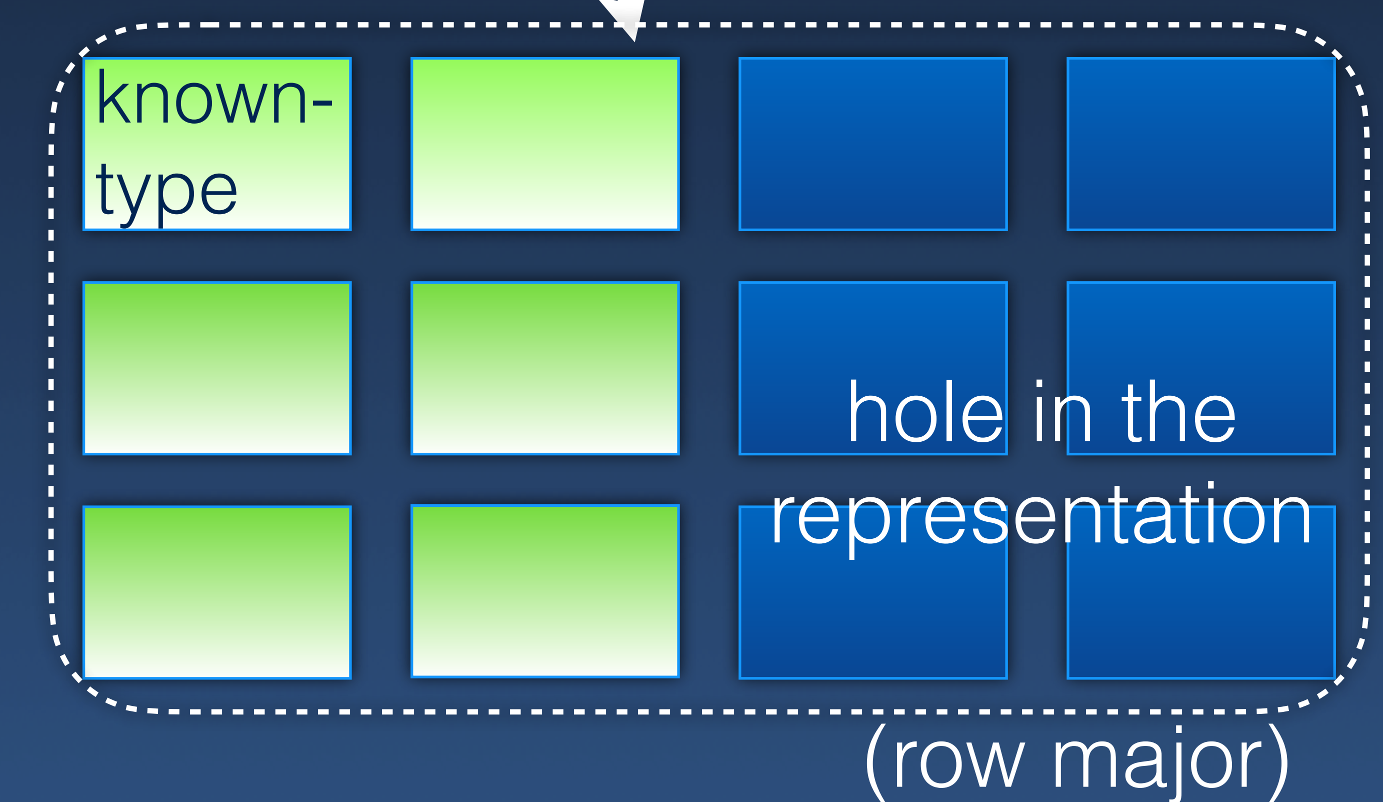
```
MPI_Datatype newtype;
MPI_Type_contiguous(count, MPI_INT, &newtype);
```

Blocks

- Equally-spaced blocks of the known datatype

➡ MPI_Type_vector(blockcount, blocklength, blockstride, knowntype, &newtype);
(blockcount = 3, blocklength = 2, blockstride = 4)

  - Assume contiguous copies of 'knowntype'
  - Stride between blocks specified in units of knowntype
  - All picked blocks are of the same length



➡ MPI_Type_create_hvector(blk_count, blk_length, bytestride, knowntype, &newtype); Gap between blocks is in bytes

Generalized Blocks

- MPI_Type_indexed(count, array_of_blocklengths, array_of_offsets, knowntype, &newtype);

  count = 5, array_of_blocklengths = 2,1,1,1,2, array_of_offsets = 0,4,6,8,10

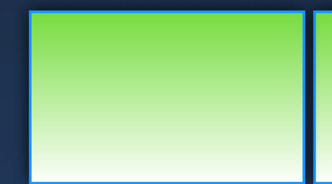
  - ➡ Blocks can contain different number of copies
  - ➡ And may have different strides
  - ➡ But the same data type

Struct

- MPI_Type_create_struct(count, array_of_blocklengths, array_of_byteoffsets, array_of_knowntypes, &newtype)
  - Example:
    - Suppose Type0 = {(double, 0), (char, 8)},
    - int BL[] = {2, 1, 3}, Disp[] = {0, 16, 26};
    - MPI_Datatype Typ[] = {MPI_FLOAT, Type0, MPI_CHAR)
  - MPI_Type_create_struct(3, BL, Disp, Typ, &newtype):
    - (float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27), (char, 28)

Struct

- MPI_Type_create_struct(count, array_of_blocklengths, array_of_byteoffsets, array_of_knowntypes, &newtype)
  - ➡ Example:
    - Suppose Type0 = {(double, 0), (char, 8)},



    - int BL[] = {2, 1, 3}, Disp[] = {0, 16, 26};

    - MPI_Datatype Typ[] = {MPI_FLOAT, Type0, MPI_CHAR)
  - ➡ MPI_Type_create_struct(3, BL, Disp, Typ, &newtype):
    - (float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27), (char, 28)

MPI_Type_get_contents(..)

Data Type Functions

- MPI_Type_commit(&datatype)
  - ➡ A datatype object must be committed before communication
- MPI_Type_size(datatype, &size)
  - ➡ Total size in bytes
- MPI_Type_get_extent(datatype, &beg, &extent);
- MPI_Type_create_resized(datatype, beg, extent, &newtype);
- MPI_Get_address(data, &Address[0]);
- MPI_BOTTOM

Data Type Functions

- MPI_Type_commit(&datatype)
  - ➡ A datatype object must be committed before communication
- MPI_Type_size(datatype, &size)
  - ➡ Total size in bytes
- MPI_Type_get_extent(datatype, &b
- MPI_Type_create_resized(datatype, beg, extent, &newtype);
- MPI_Get_address(data, &Address[0]);
- MPI_BOTTOM

```
MPI_Datatype atype;
MPI_Type_contiguous(4, MPI_CHAR, &atype);
int asize;
MPI_Type_size(atype, &asize);
MPI_Type_commit(&atype);
MPI_Send(buf, nItems, atype, dest, ..);
MPI_Recv(...);
```

Derived Type Example

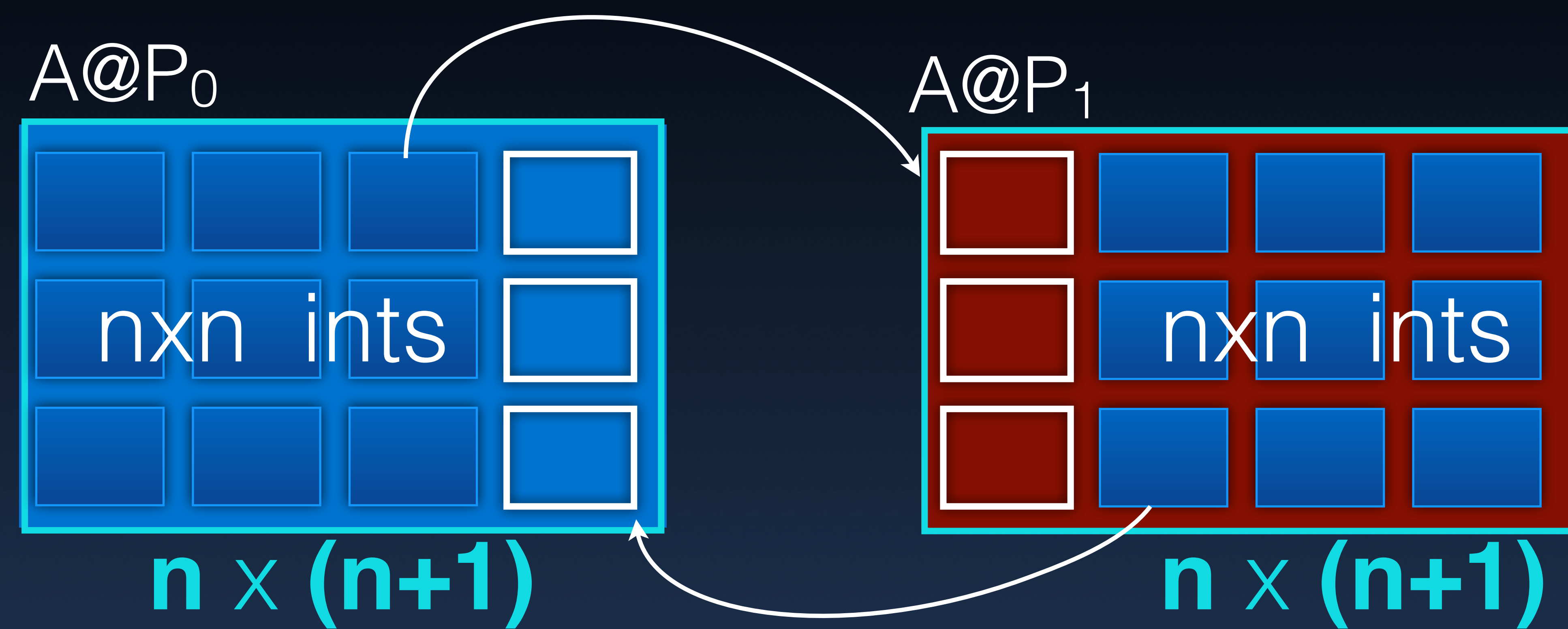
```
sendParticles(struct Particle particle[], int N):
    MPI_Datatype Particletype;
    MPI_Datatype types[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
    int blockcount[3] = {1, 6, 7};

    /* compute displacements of structure components */
    MPI_Aint disp[3];
    MPI_Address(particle, disp);
    MPI_Address(particle[0].d, disp+1);
    MPI_Address(particle[0].b, disp+2);
    for (int i=2; i >= 0; i--) disp[i] -= disp[0];

    MPI_Type_struct(3, blockcount, disp, types, &Particletype);
    MPI_Type_commit( &Particletype);
    MPI_Send(particle, N, Particletype, dest, tag, comm);
```

```
struct Particle
{
    int class;      // particle class
    double d[6];    // particle coordinates
    char b[7];      // some additional info
};
```

Data Transfer




```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
   MPI_Send(A+n-1, 1, column, 1, tag, MPI_COMM_WORLD);
   MPI_Recv(A+n, 1, column, 1, tag, MPI_COMM_WORLD, &status);
}
if(rank == 1) {
   MPI_Recv(A, 1, column, 0, tag, MPI_COMM_WORLD, &status);
   MPI_Send(A+1, 1, column, 0, tag, MPI_COMM_WORLD);
}
```

- MPI_Barrier
  - Barrier synchronization across all members of a group
- MPI_Bcast
  - Broadcast from one member to all members of a group
- MPI_Scatter, MPI_Gather, MPI_Allgather
  - Gather data from all members of a group to one
- MPI_Alltoall
  - complete exchange or all-to-all
- MPI_Reduce, MPI_Allreduce,
  - Reduction operations
- MPI_Reduce_Scatter
  - Combined reduction and scatter operation
- MPI_Scan, MPI_Exscan
  - Prefix

Barrier

- Synchronization of the calling processes
  – the call blocks until all of the processes have placed the call

```
MPI_Barrier(comm);
```

Broadcast

```
MPI_Bcast(mesg, count, MPI_INT, root, comm);
```

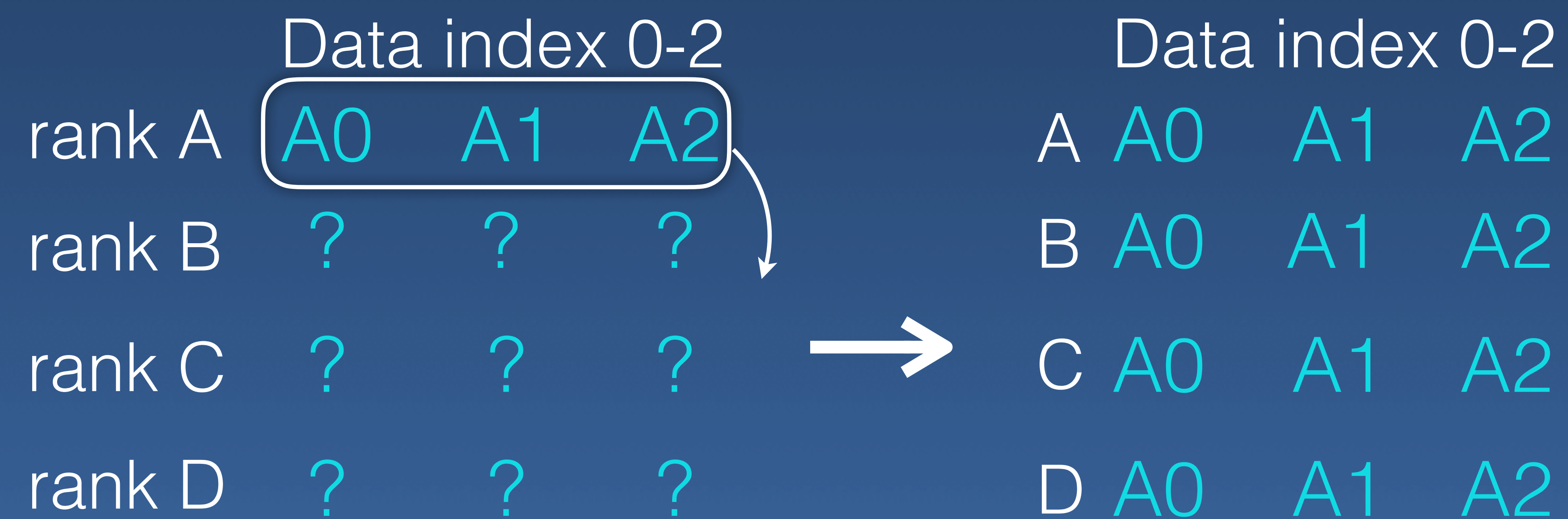
pointer on all (mesg)

number & type (count, MPI_INT)

identified sender (root)

can be intercommunicator (comm)

- All participants must call, match by comm and root
- No implicit synchronization



Subodh Kumar

Broadcast

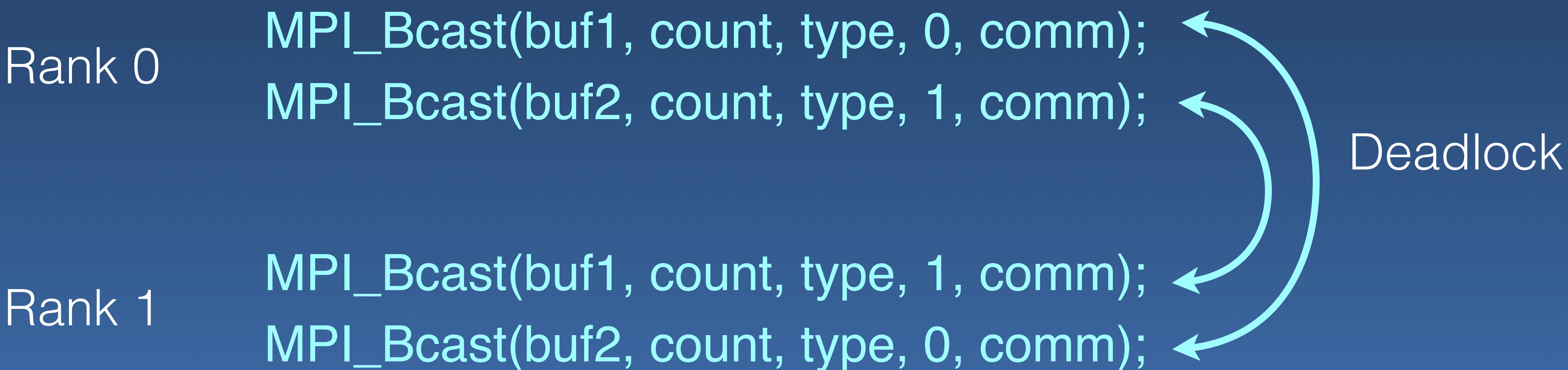
See: MPI_Reduce, MPI_Scan

```
MPI_Bcast(mesg, count, MPI_INT, root, comm);
```

- mesg: pointer on all
- count, MPI_INT: number & type
- root: identified sender
- comm: can be intercommunicator

- All participants must call, match by comm and root
- No implicit synchronization

see `MPI_Ibcast`



Subodh Kumar

MPI_Gather

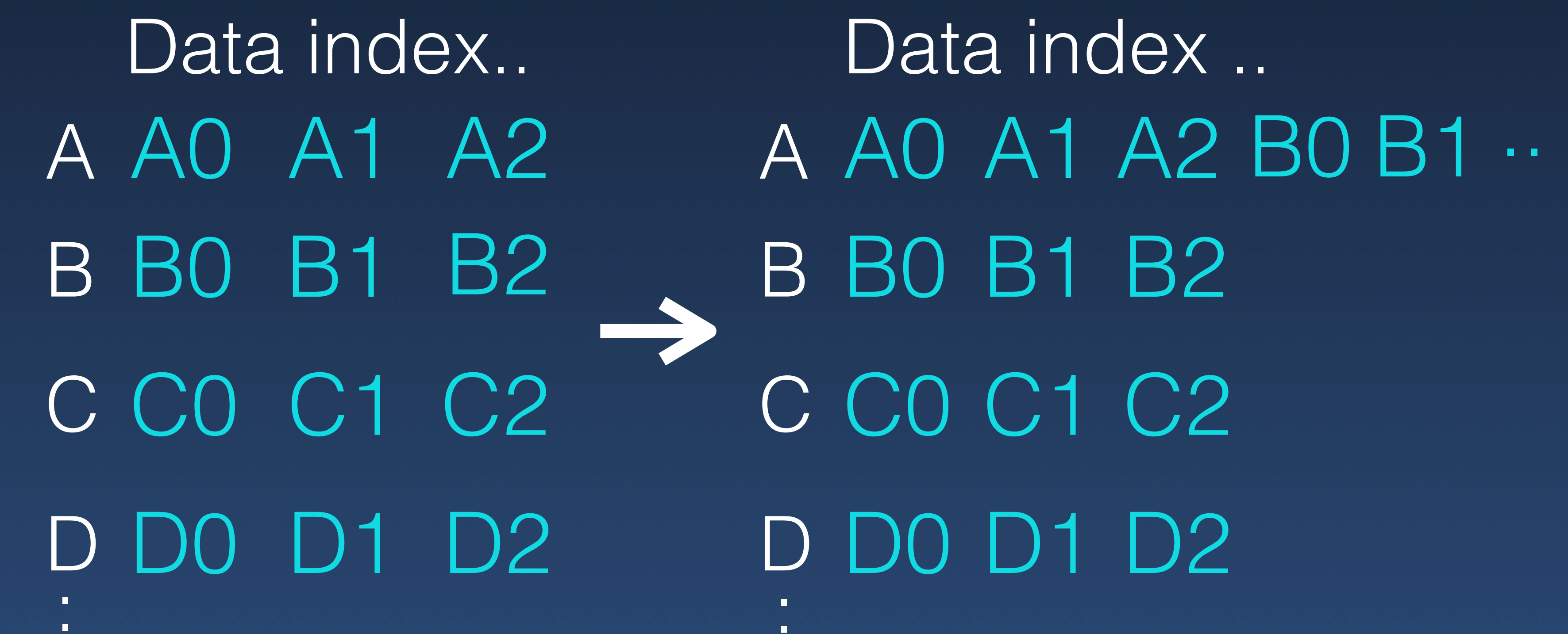
```
MPI_Gather(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype, root, comm);
```

- Similar to non-roots sending:
  - MPI_Send(sendbuf, sendcount, sendtype, root, ...),
- and the root receiving n times:
  - MPI_Recv(recvbuf + i * recvcount *extent(recvtype), recvcount, recvtype, i, ...)
- **`MPI_Gatherv`** allows different size data to be gathered
- **`MPI_Allgather`** has no *root*; all nodes receive similarly

MPI_Gather

```
MPI_Gather(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype, root, comm);
```

- Similar to non-roots sending:
  - MPI_Send(sendbuf, sendcount, sendtype, root, ...),
- and the root receiving n times:
  - MPI_Recv(recvbuf + i * recvcount *extent(recvtype), recvcount, recvtype, i, ...)

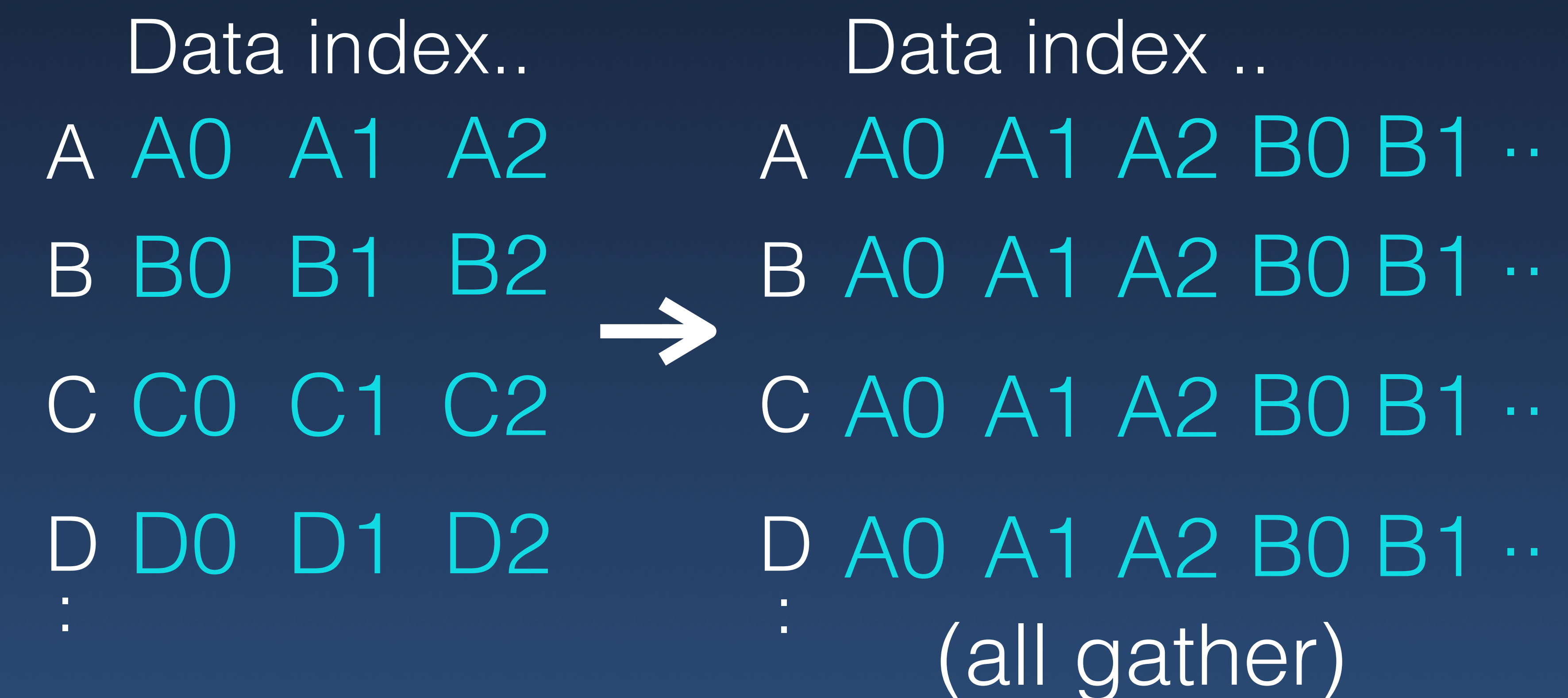


- **MPI_Gatherv** allows different size data to be gathered
- **MPI_Allgather** has no *root*; all nodes receive similarly

MPI_Gather

```
MPI_Gather(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype, root, comm);
```

- Similar to non-roots sending:
  - MPI_Send(sendbuf, sendcount, sendtype, root, ...),
- and the root receiving n times:
  - MPI_Recv(recvbuf + i * recvcount *extent(recvtype), recvcount, recvtype, i, ...)

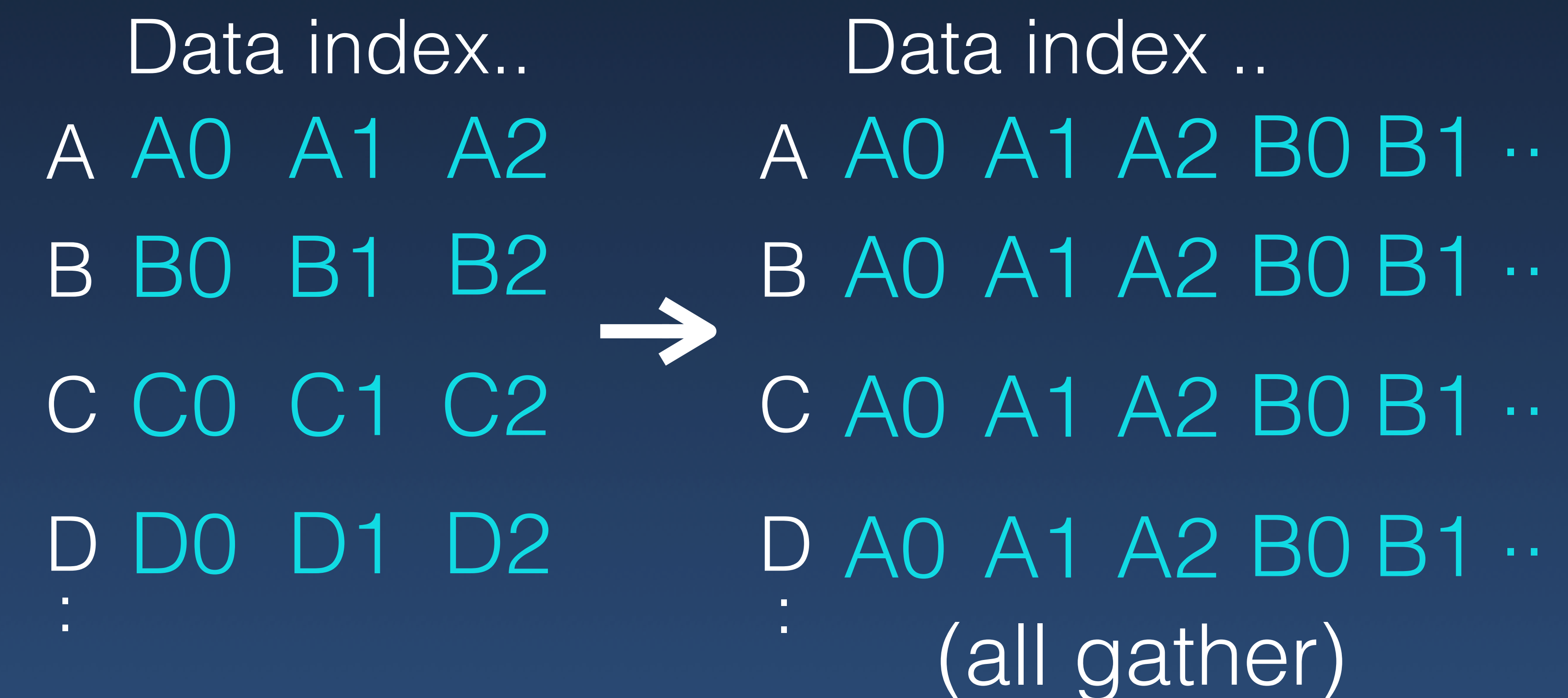


- **MPI_Gatherv** allows different size data to be gathered
- **MPI_Allgather** has no *root*; all nodes receive similarly

MPI_Scatter is opposite of MPI_Gather

```
MPI_Gather(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype, root, comm);
```

- Similar to non-roots sending:
  - MPI_Send(sendbuf, sendcount, sendtype, root, ...),
- and the root receiving n times:
  - MPI_Recv(recvbuf + i * recvcount *extent(recvtype), recvcount, recvtype, i, ...)
- **MPI_Gatherv** allows different size data to be gathered
- **MPI_Allgather** has no *root*; all nodes receive similarly



Gather Example

```
MPI_Comm com;
int gsize, sendarray[100];
int root, *recvbuf;
MPI_Datatype rtype;
 ...
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
recvbuf = (int *) malloc(gsize * 100 * sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, recvbuf, 1, rtype, root, comm)
```

All to All

	Data index..		
A	A0	A1	A2
B	B0	B1	B2
C	C0	C1	C2
:			

→

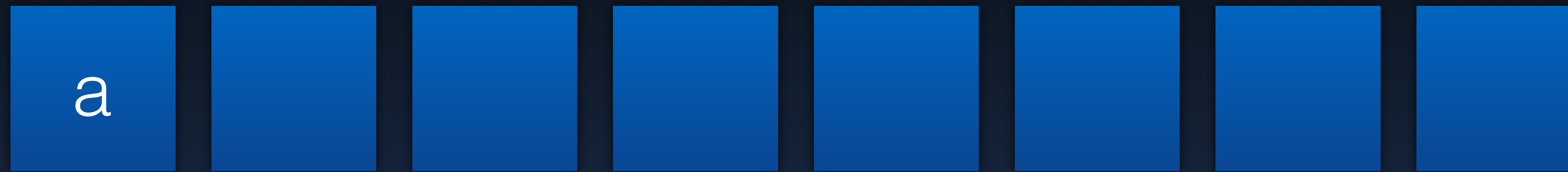
	Data index ..		
A	A0	B0	C0
B	A1	B1	C1
C	A2	B2	C2
:			

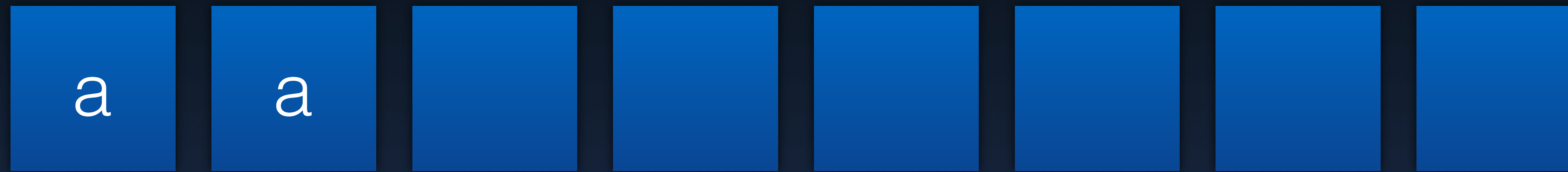
	Data index..					
A	A0	A1	A2	A3	A4	A5
B	B0	B1	B2	B3	B4	B5
C	C0	C1	C2	C3	C4	C5
:						

	Data index..					
A	A0	A1	B0	B1	C0	C1
B	A2	A3	B2	B3	C2	C3
C	A4	A5	B4	B5	C4	C5
:						

Can all-to-all multiple data items

Collective Implementation




Bcast
a
a

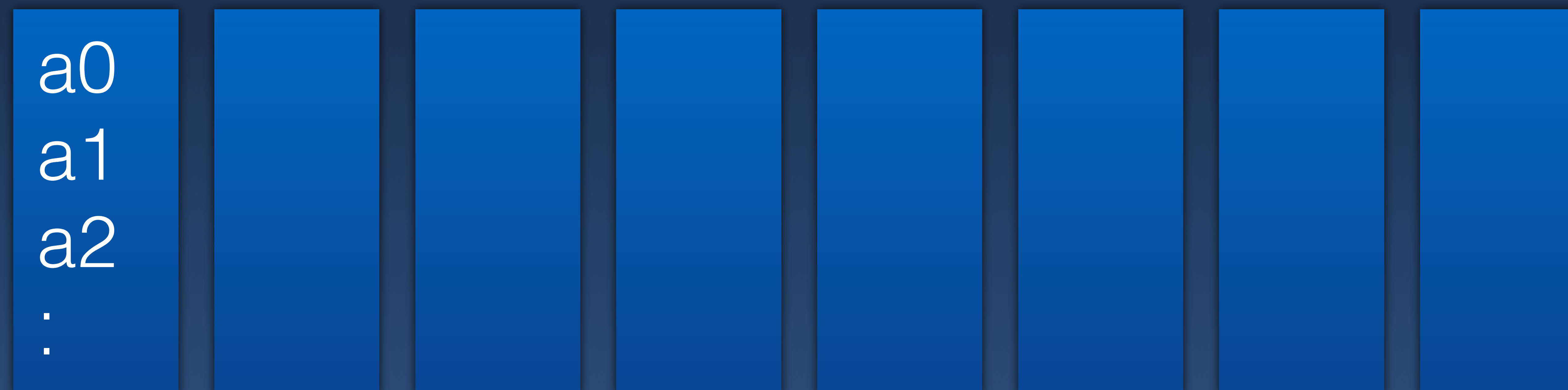
Bcast
a
a
a
a

Collective Implementation

Collective Implementation

log P rounds
1 message/round/pair
of ‘unit’ size

Collective Implementation



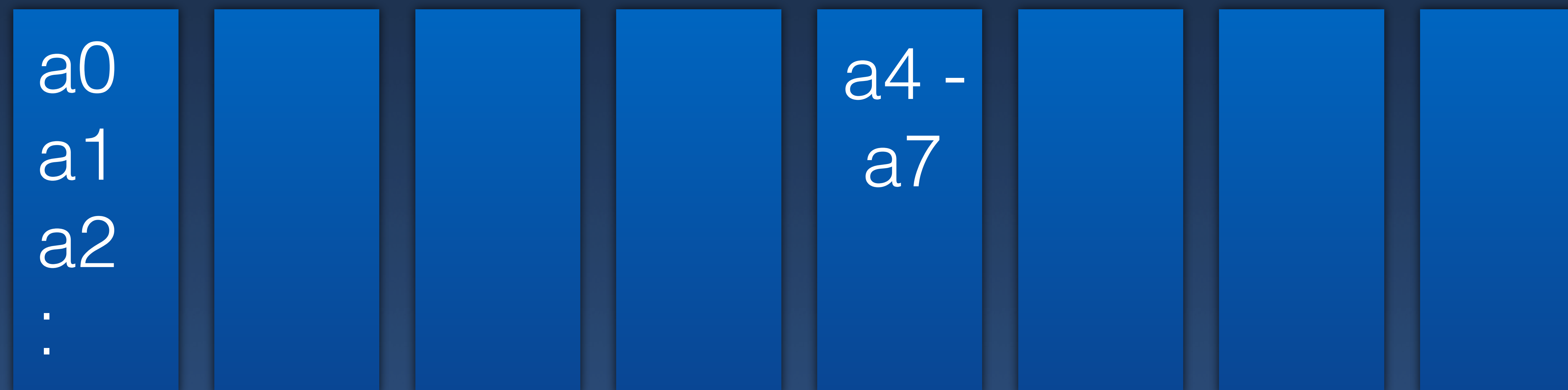
log P rounds

1 message/round/pair

of ‘unit’ size

Subodh Kumar

Collective Implementation



log P rounds
1 message/round/pair
of ‘unit’ size

Collective Implementation

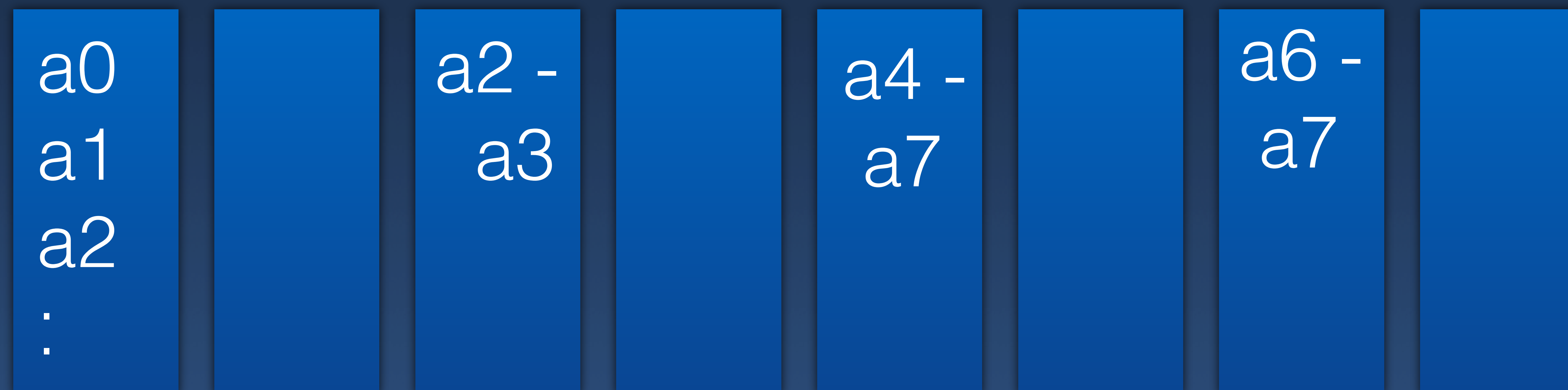
Bcast

log P rounds

1 message/round/pair

of 'unit' size

Scatter



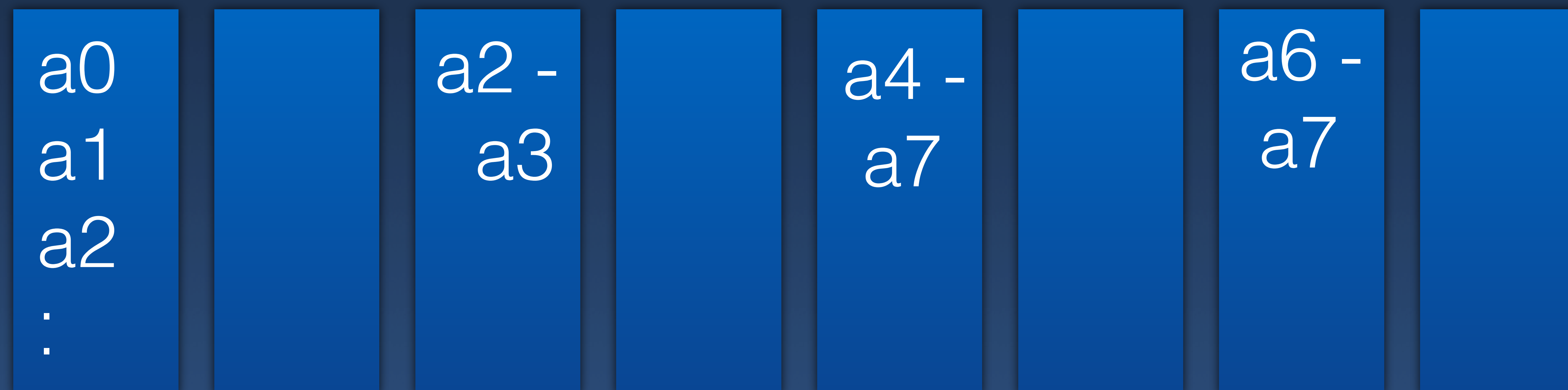
Subodh Kumar

Collective Implementation

Bcast

log P rounds
1 message/round/pair
of ‘unit’ size

Scatter



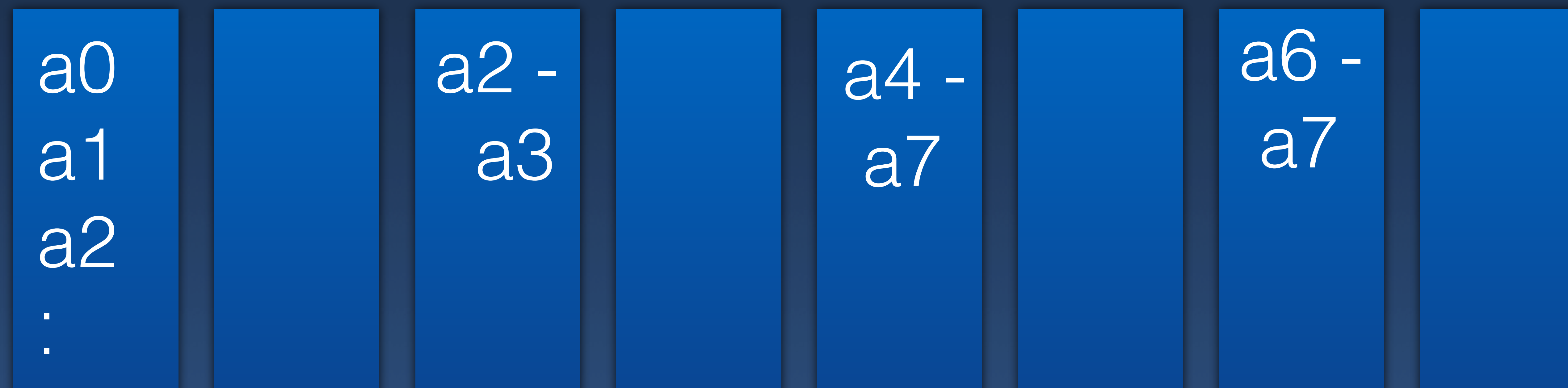
log P rounds
1 message/round/pair
of P/2, P/4, P/8 .. units

Collective Implementation

log P rounds

1 message/round/pair

of ‘unit’ size



log P rounds

1 message/round/pair

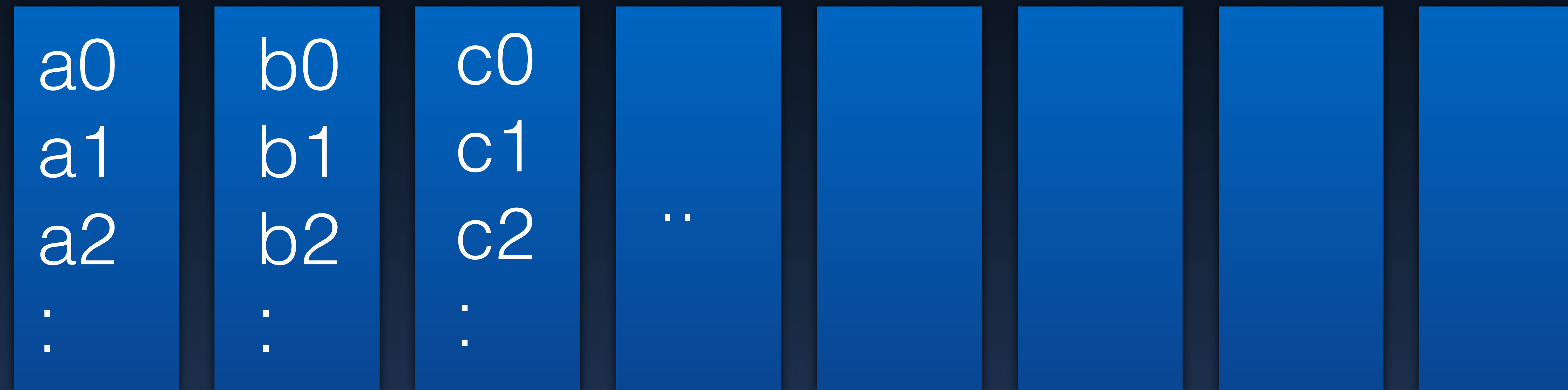
of P/2, P/4, P/8 .. units

```
r = 2 ⌈log n⌉
while(r > 1):
    if( PID & (r-1) == 0)
        Send items[r/2:end] to PID+r/2 (match recv)
    r  /= 2;
```

Subodh Kumar

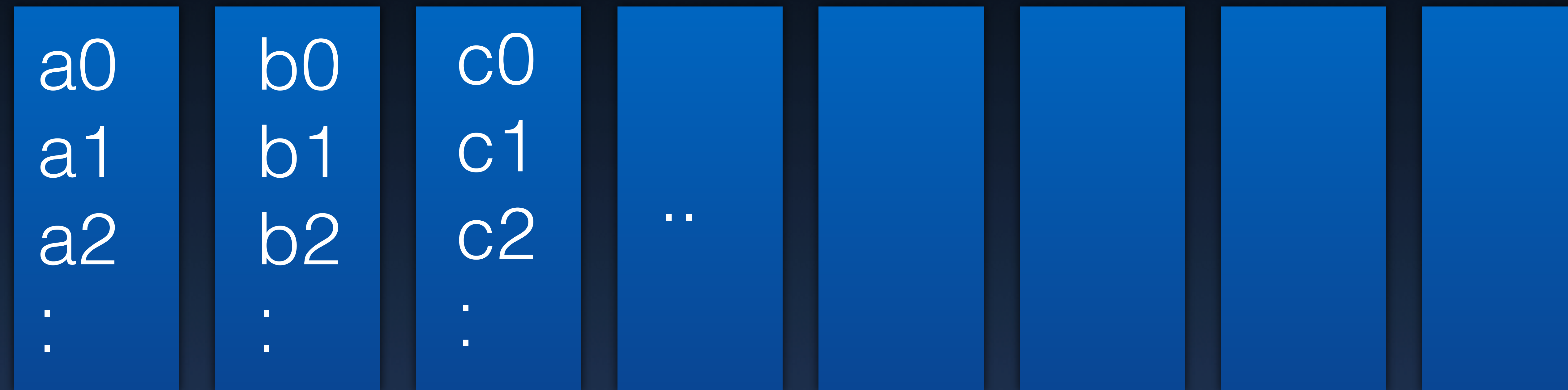
Collective Communication

All to All



Collective Communication

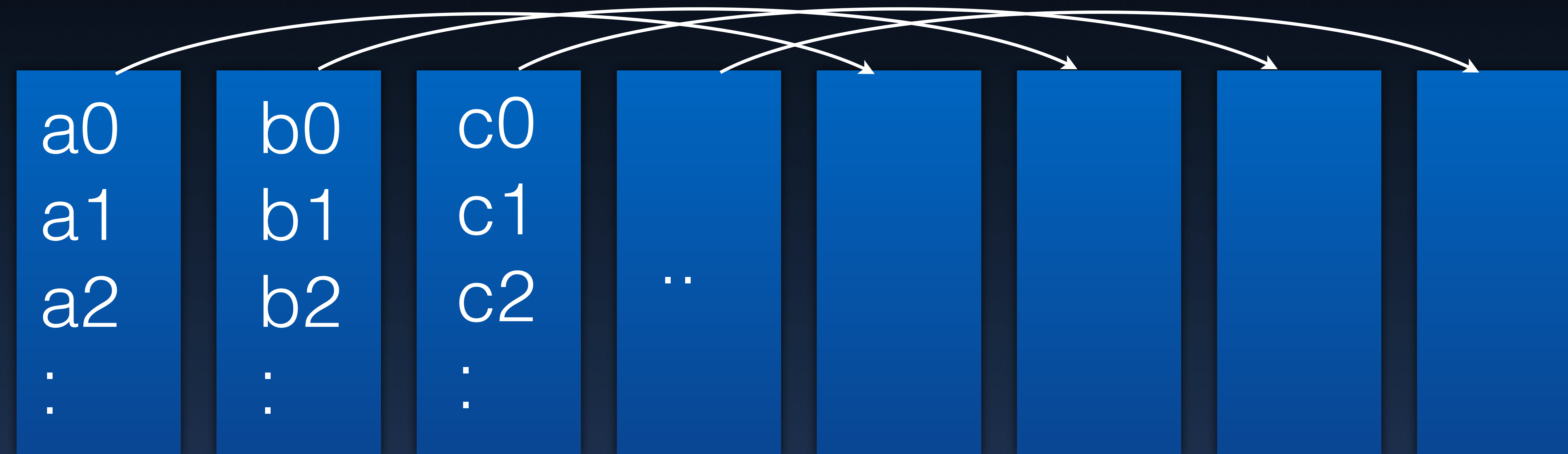
All to All



P sequential Scatters (P log P rounds)?

Subodh Kumar

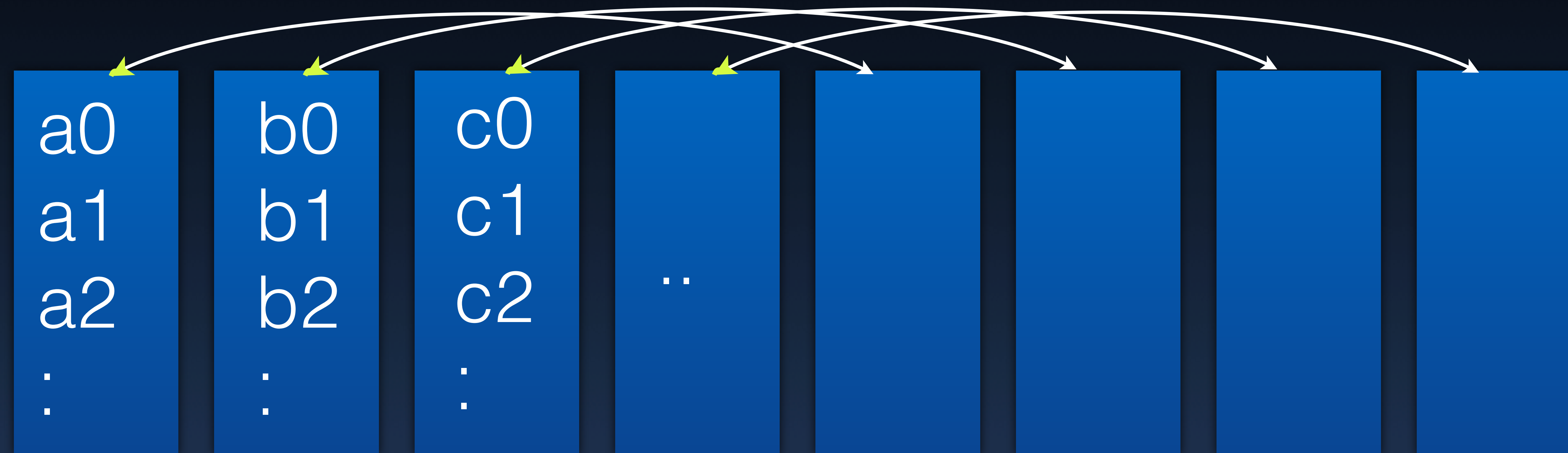
Collective Communication



P sequential Scatters (P log P rounds)?

Others could also begin in parallel..

Collective Communication

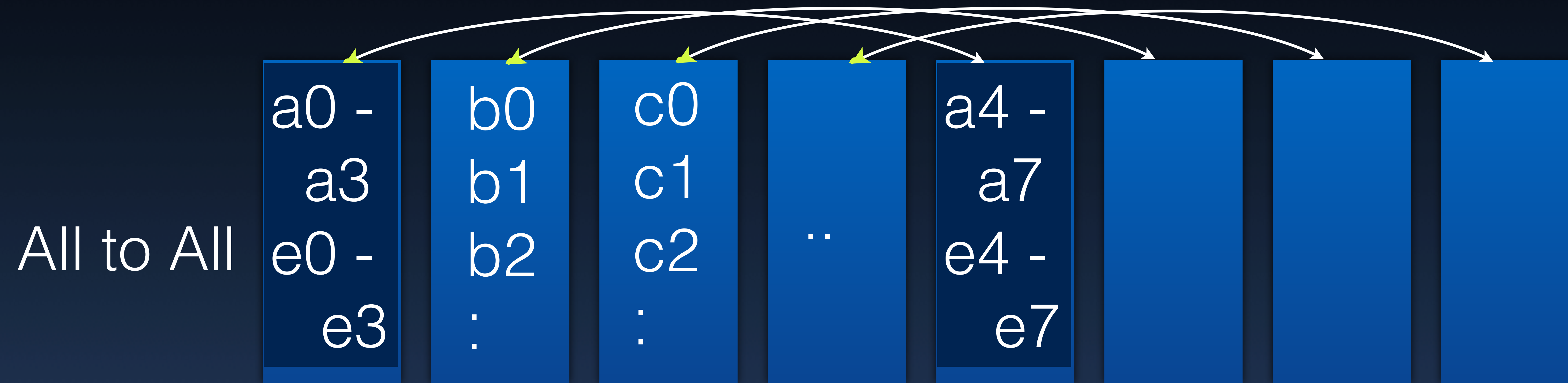


P sequential Scatters (P log P rounds)?

Others could also begin in parallel..

(Both ways)

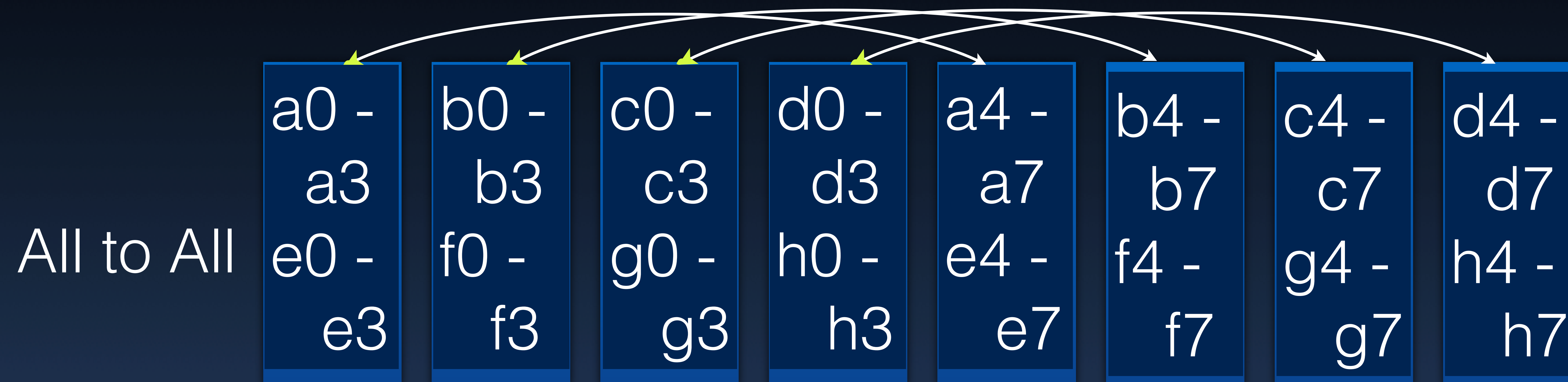
Collective Communication



P sequential Scatters (P log P rounds)?

Others could also begin in parallel..
(Both ways)

Collective Communication



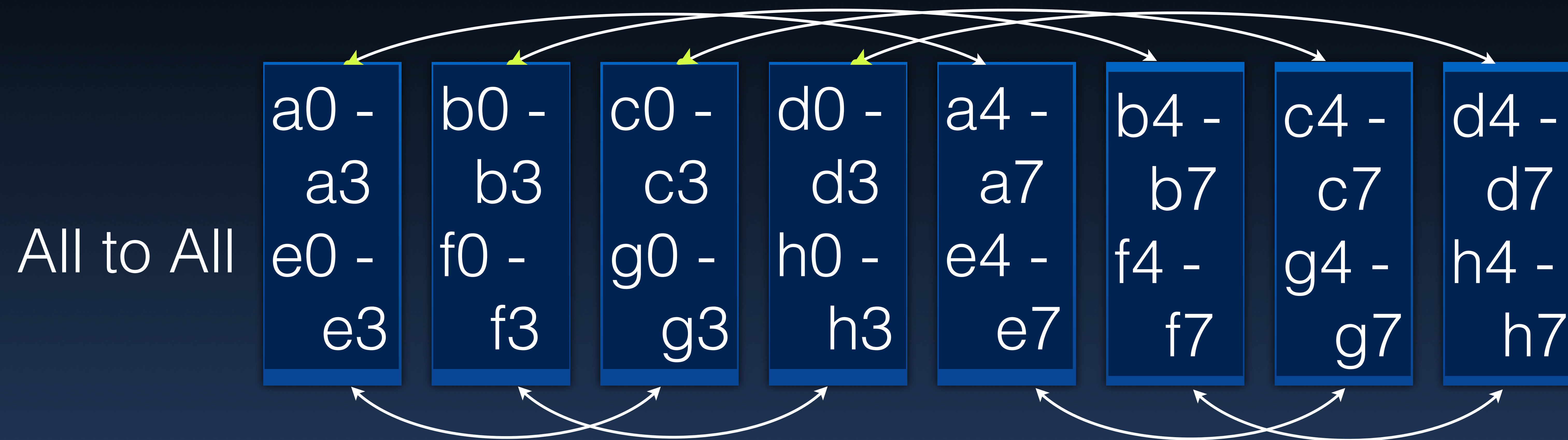
P sequential Scatters (P log P rounds)?

Others could also begin in parallel..

(Both ways)

- P/2 pairs exchange ½ their data (first/second half)
  - ➡ each P/2 apart

Collective Communication



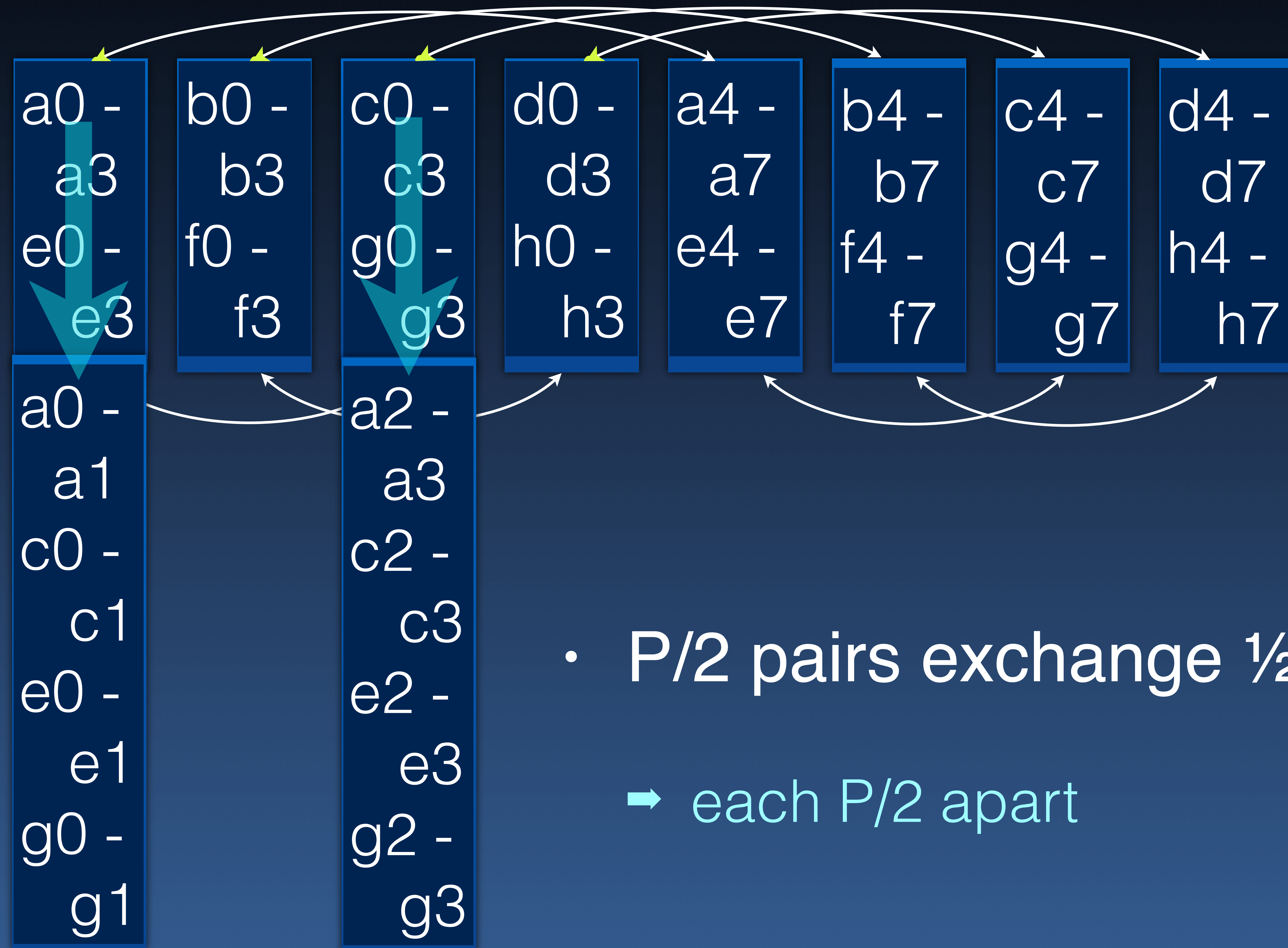
P sequential Scatters (P log P rounds)?

Others could also begin in parallel..

(Both ways)

- P/2 pairs exchange ½ their data (first/second half)
  - ➡ each P/2 apart

Collective Communication



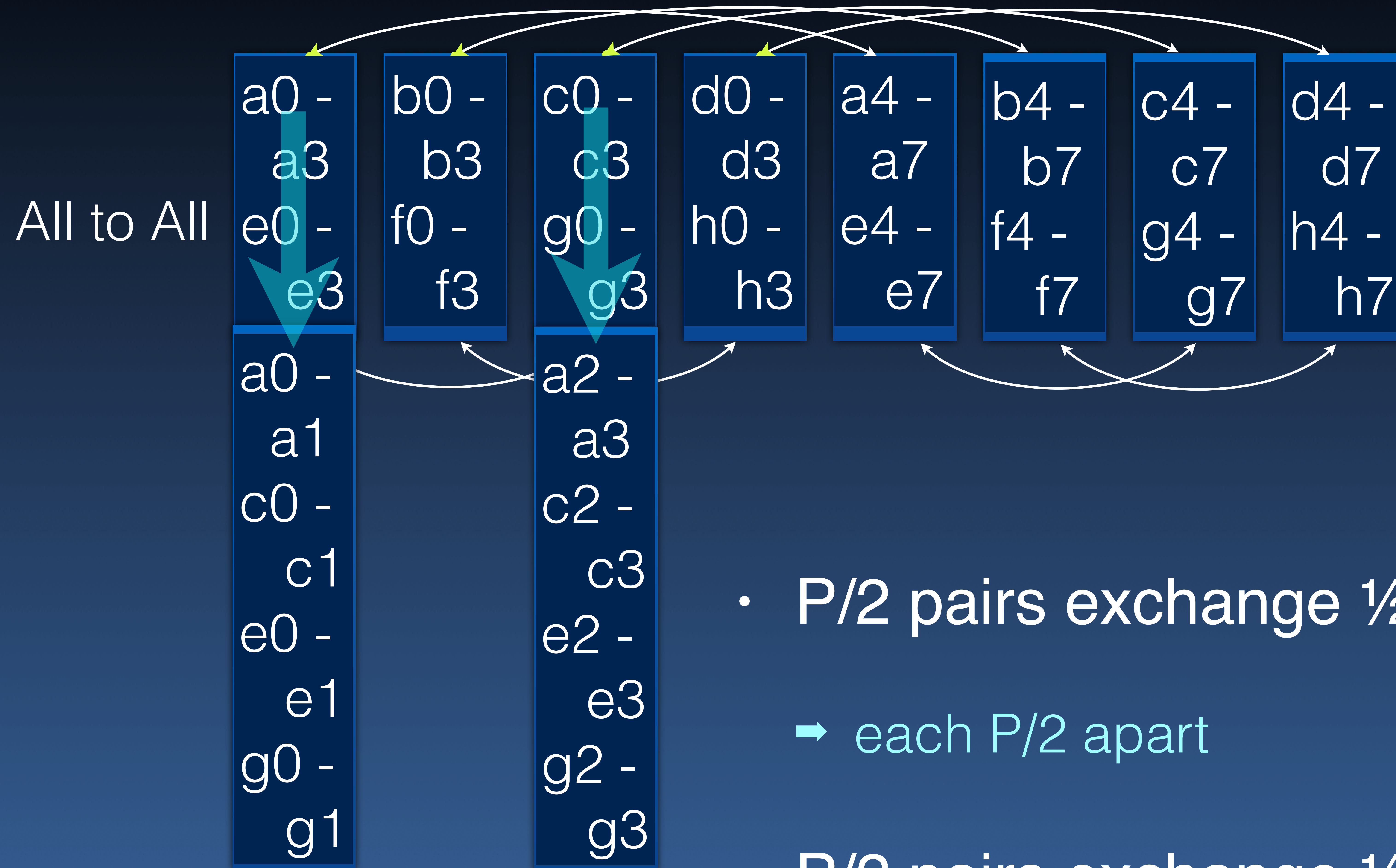
P sequential Scatters (P log P rounds)?

Others could also begin in parallel..

(Both ways)

- P/2 pairs exchange ½ their data (first/second half)
  - ➡ each P/2 apart

Collective Communication



~~P sequential Scatters (P log P rounds)?~~

Others could also begin in parallel..
(Both ways)

log P rounds
1 message/round/pair
of P units

- P/2 pairs exchange ½ their data (first/second half)
  - ➡ each P/2 apart
- P/2 pairs exchange ½ of each ½ assembled earlier
  - ➡ each P/4 apart

Remote Memory

MPI_Win_create(addr, size, displ_unit, info, MPI_COMM_WORLD, &win);

…

MPI_Win_free(&win);

- Weak synchronization
- Collective call
- Info specifies system-specific information (e.g., memory locking)
  - ➡ Designed to optimize performance
- Also see MPI_Alloc_mem/MPI_Win_allocate for <addr> allocation
  (RMA friendly)

- MPI_Put(my_addr, my_count, my_datatype,
  there_rank, there_disp, there_count, there_datatype, win);
  - ➡ Written in the dest window-buffer at address
    - window_base + disp×disp_unit
  - ➡ Must fit in the target buffer
  - ➡ there_datatype defined on the “putter”
    - But refers to memory “there”
    - Usually defined on both sides

- MPI_Put(my_addr, my_count, my_datatype,
  there_rank, there_disp, there_count, there_datatype, win);
  - ➡ Written in the dest window-buffer at address
    - window_base + disp×disp_unit
  - ➡ Must fit in the target buffer
  - ➡ there_datatype defined on the "putter"
    - But refers to memory "there"
    - Usually defined on both sides

MPI_Get does the reverse: there ➜ my

Also see:
MPI_Accumulate
performs an "op" at destination

Remote Memory Synchronization

- MPI_Win_fence
- MPI_Win_flush
- MPI_Win_lock
- MPI_Win_unlock
- MPI_Win_start
- MPI_Win_complete
- MPI_Win_post
- MPI_Win_Wait
- MPI_Win_Test