Advanced communications

Coding allgather

Let’s implement our own allgather functions, using MPI_Send and MPI_Recv

Many possible implementations possible

- most basic (i.e. not good), all processes send to root, all receive from root. Estimated cost of startups (i.e. essentially total cost for small data sizes) would be $2(p-1)t_s$ where t_s is the communication startup time

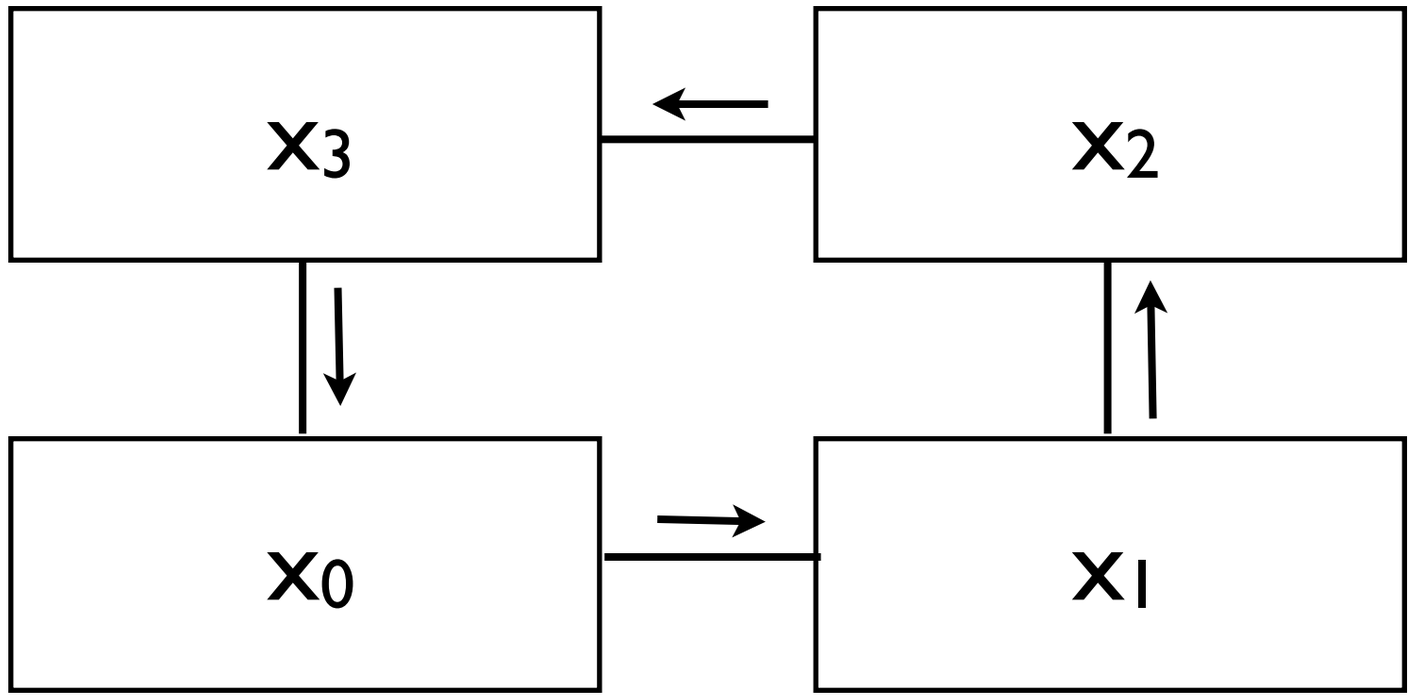
- if our processes can be conveniently arranged in a tree, we could do tree-structured gather to root and then tree-structured broadcast from room. Good performance, cost of startups is $2[\log_2(p)-1)]t_s$. However, this is an unlikely physical arrangement.

- if processors are physically arranged in a ring, a ring pass would be a natural solution. Estimated cost of startups is $(p-1)t_s$

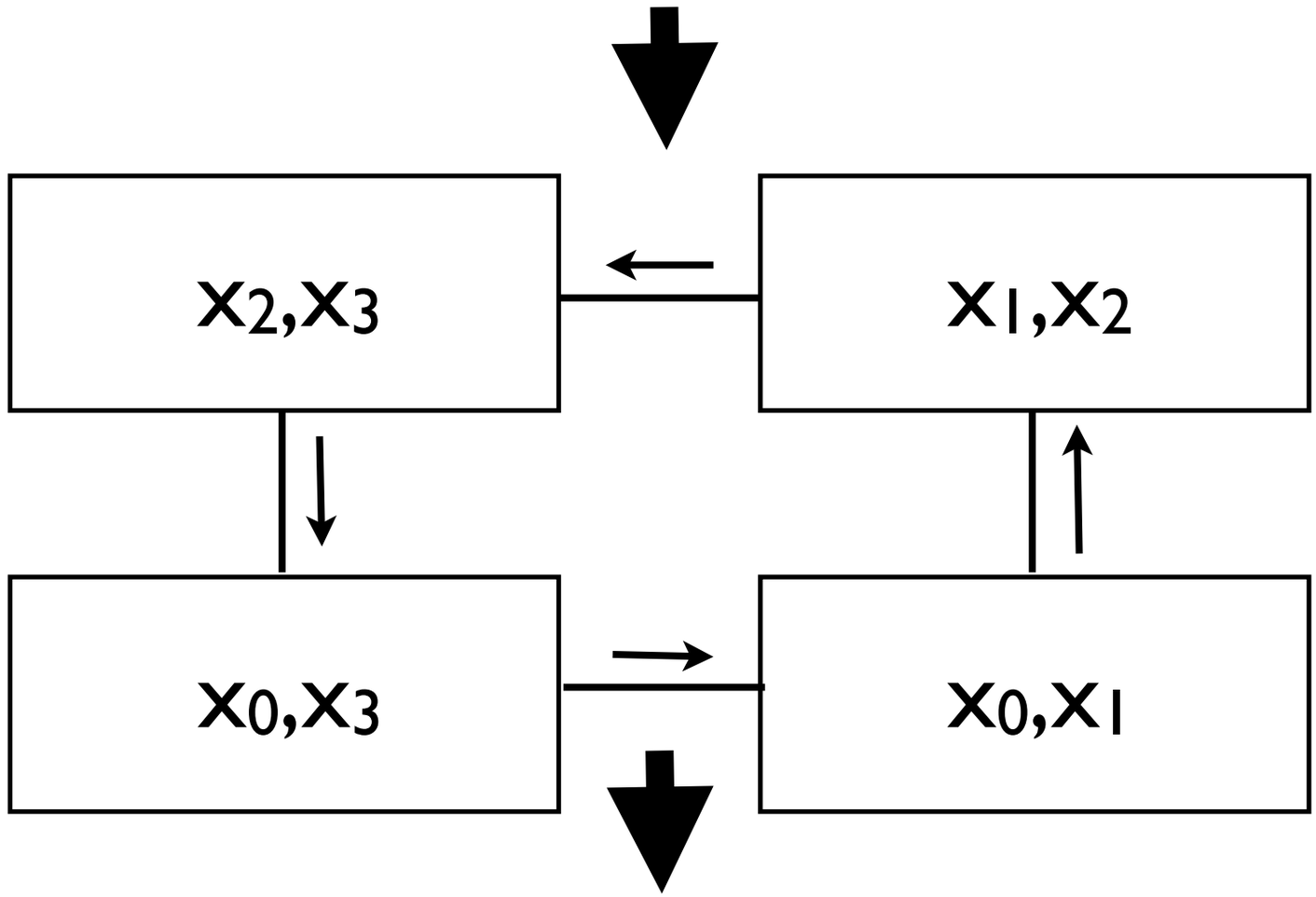
- if processes topology can be adequately modelled as a hypercube, we can have a sequence of pairwise exchanges, where at each stage processes are split into groups of equal sizes (assuming number of processes p is some power of 2 in this case) Cost of startups is then $\log_2(p)t_s$

Any given architecture is unlikely to fit these arrangements perfectly, but some will work better than others

Four-processor ring pass



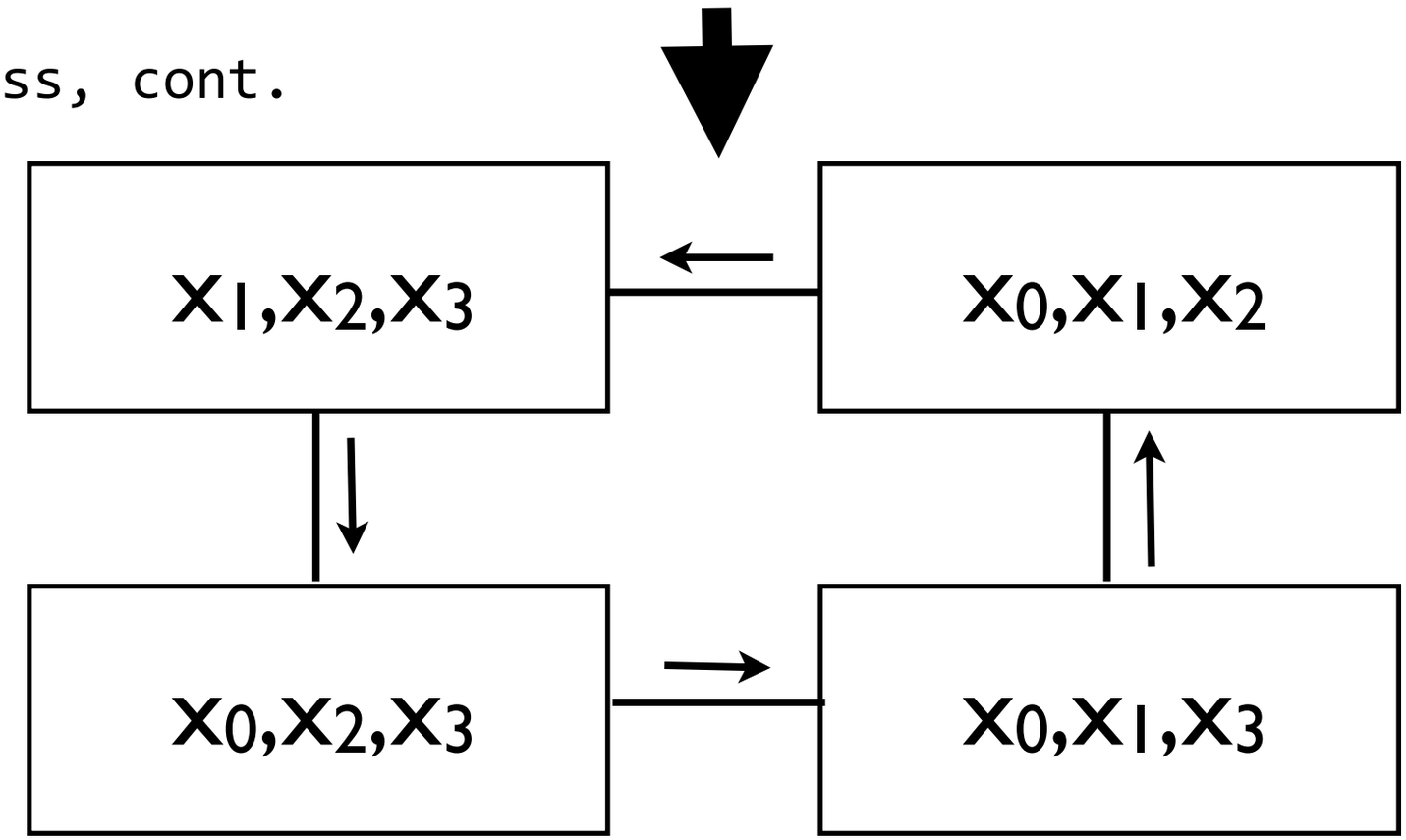
Start



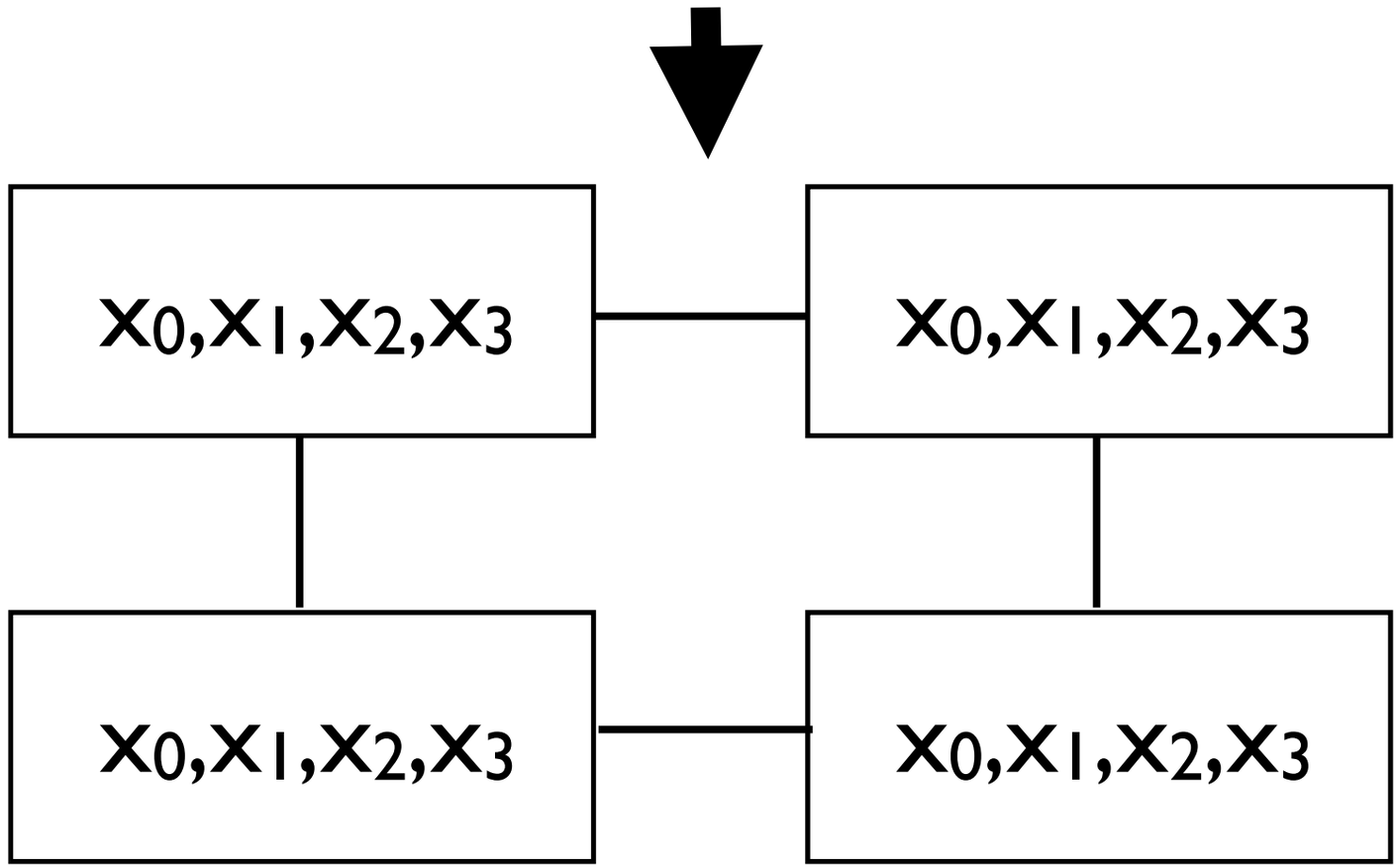
After first pass

Four-processor ring pass, cont.

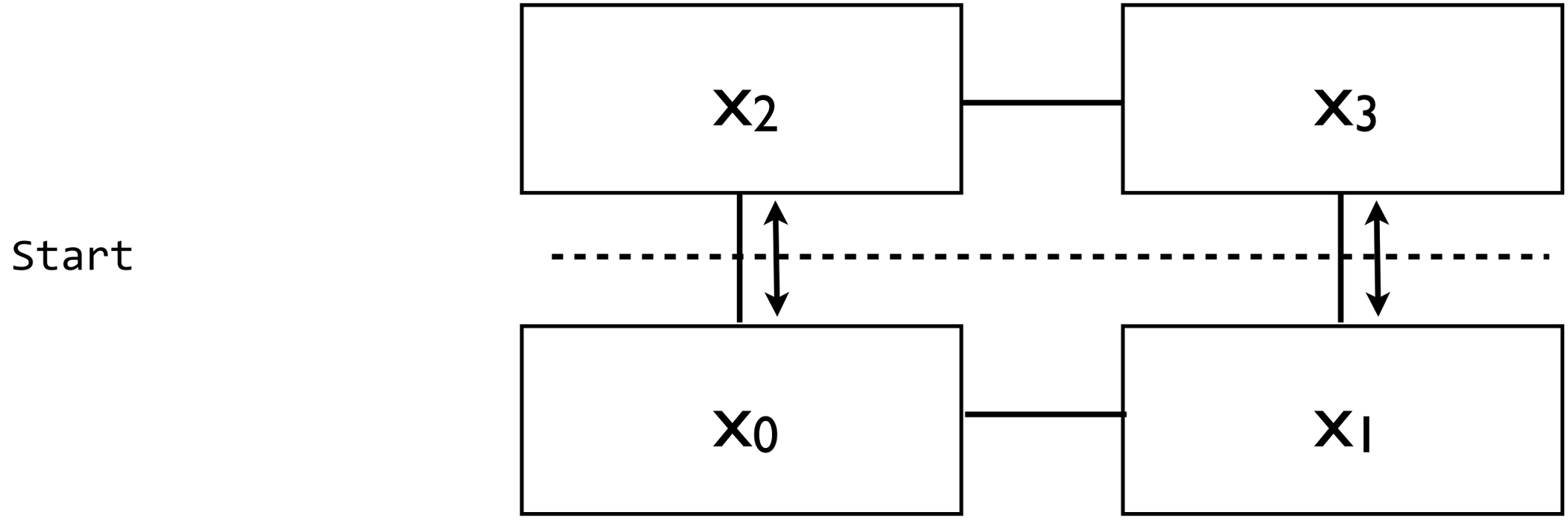
After second pass

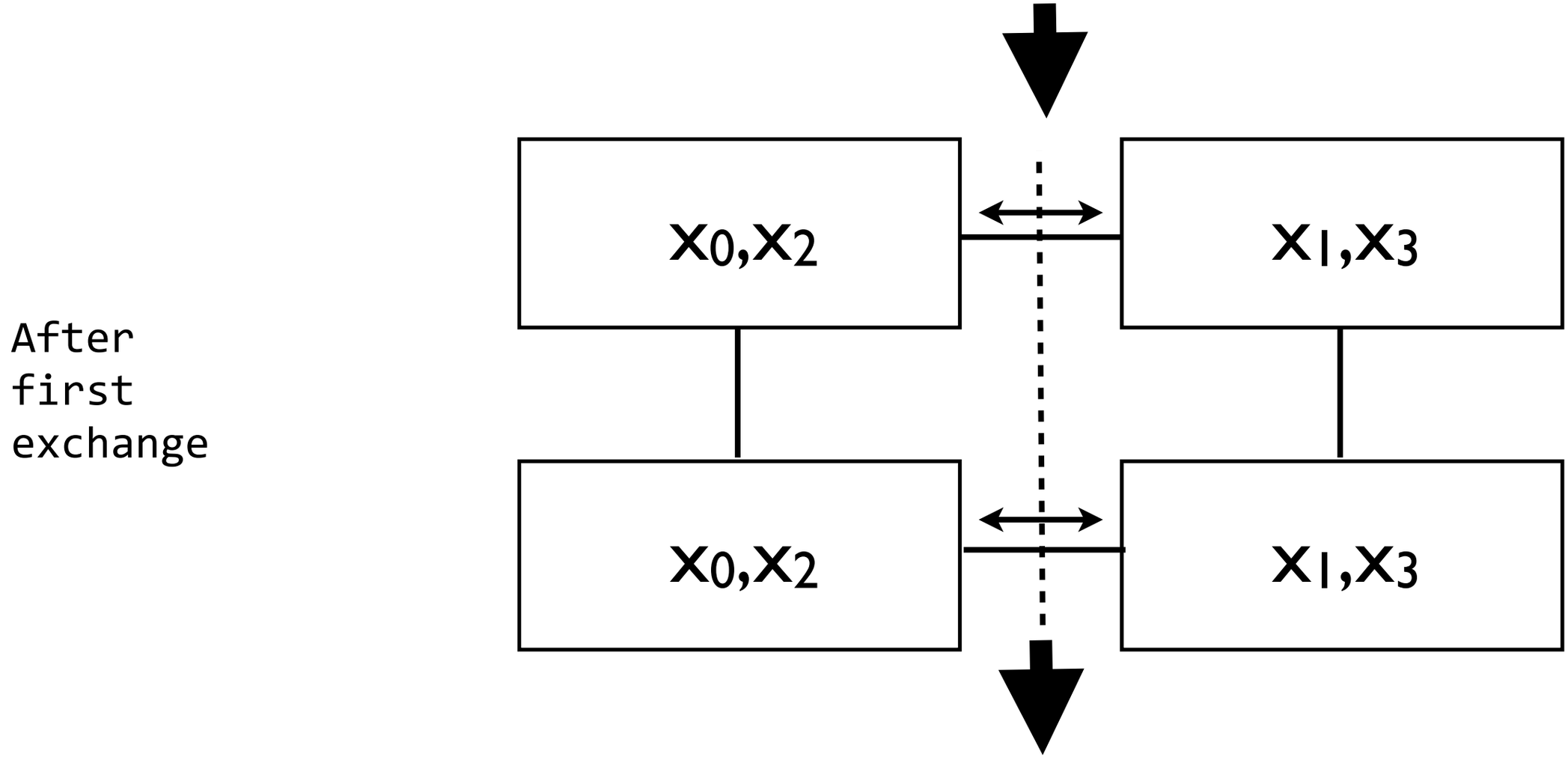


After third pass

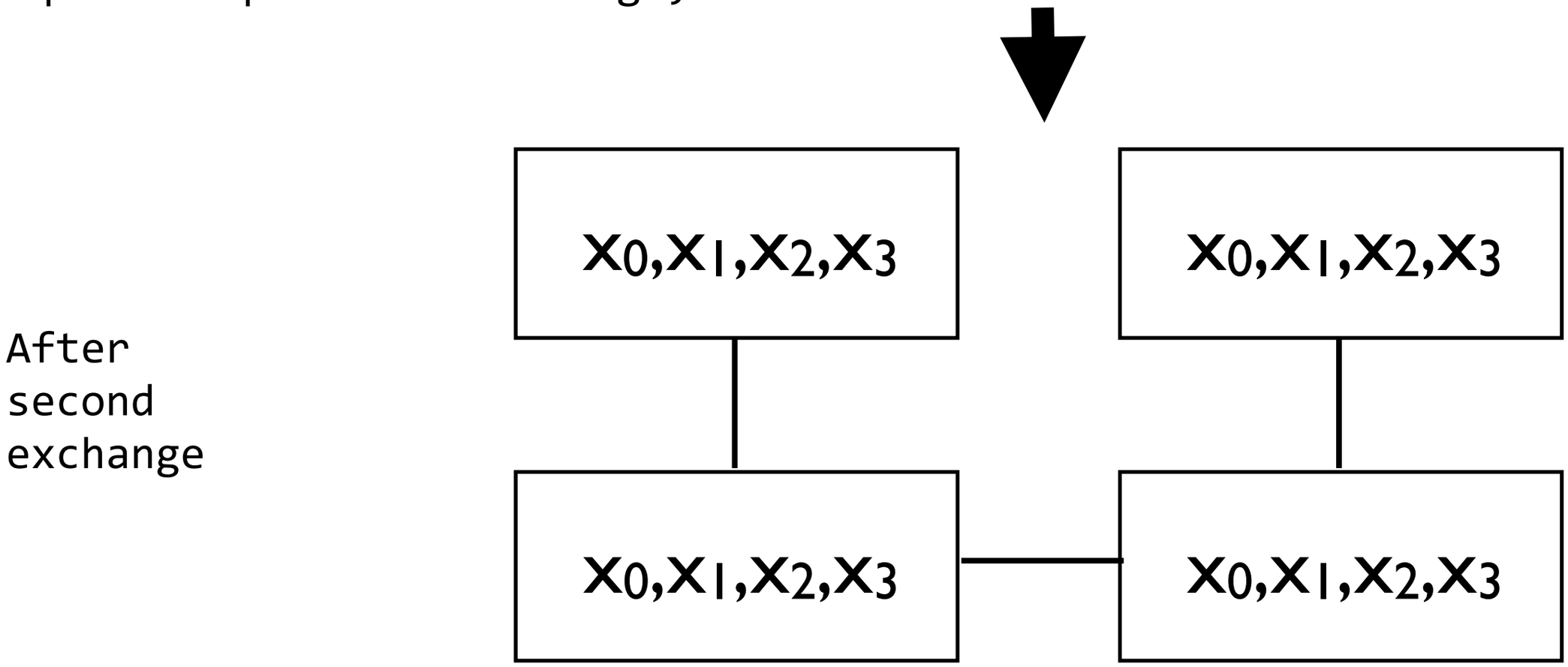


Four-processor pairwise exchange





Four-process pairwise exchange, cont



Ring Pass Allgather

Assume data consists of floats, each process holds a block initially with size “blocksize”

The block we send to on pass i, i=0,1,...,p-1 is

(my_rank-i)mod p

in C safe implementation is

(my_rank-i+p) % p

to compute offset

((my_rank-i+p) % p) * blocksize

the block we are receiving will be the block immediately preceding the one we are sending, so just subtract one from above

((my_rank-i+p-1)%p)*blocksize

```
main(int argc, char* argv[]) {
    int        p;
    int        my_rank;
    float      x[LOCAL_MAX];
    float      y[MAX];
    int        blocksize;
    MPI_Comm   io_comm;
    int        i;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);
    Cache_io_rank(MPI_COMM_WORLD, io_comm);

    Cscanf(io_comm,"Enter the local array size","%d", &blocksize);

    while(blocksize > 0) {
        for (i = 0; i < blocksize; i++)
            x[i] = (float) my_rank;
        Allgather_ring(x, blocksize, y, MPI_COMM_WORLD);
        Print_arrays(io_comm, "Gathered_arrays", y, blocksize);
        /* Enter 0 to stop. */
        Cscanf(io_comm,"Enter the local array size",
            "%d", &blocksize);
    }
    MPI_Finalize();}  /* main */
```

```
/*****************************************************************/
void Allgather_ring(
          float     x[]          /* in  */,
          int       blocksize    /* in  */,
          float     y[]          /* out */,
          MPI_Comm  ring_comm    /* in  */) {

    int         i, p, my_rank;
    int         successor, predecessor;
    int         send_offset, recv_offset;
    MPI_Status  status;

    MPI_Comm_size(ring_comm, &p);
    MPI_Comm_rank(ring_comm, &my_rank);


    /* Copy x into correct location in y */
    for (i = 0; i < blocksize; i++)
        y[i + my_rank*blocksize] = x[i];
```

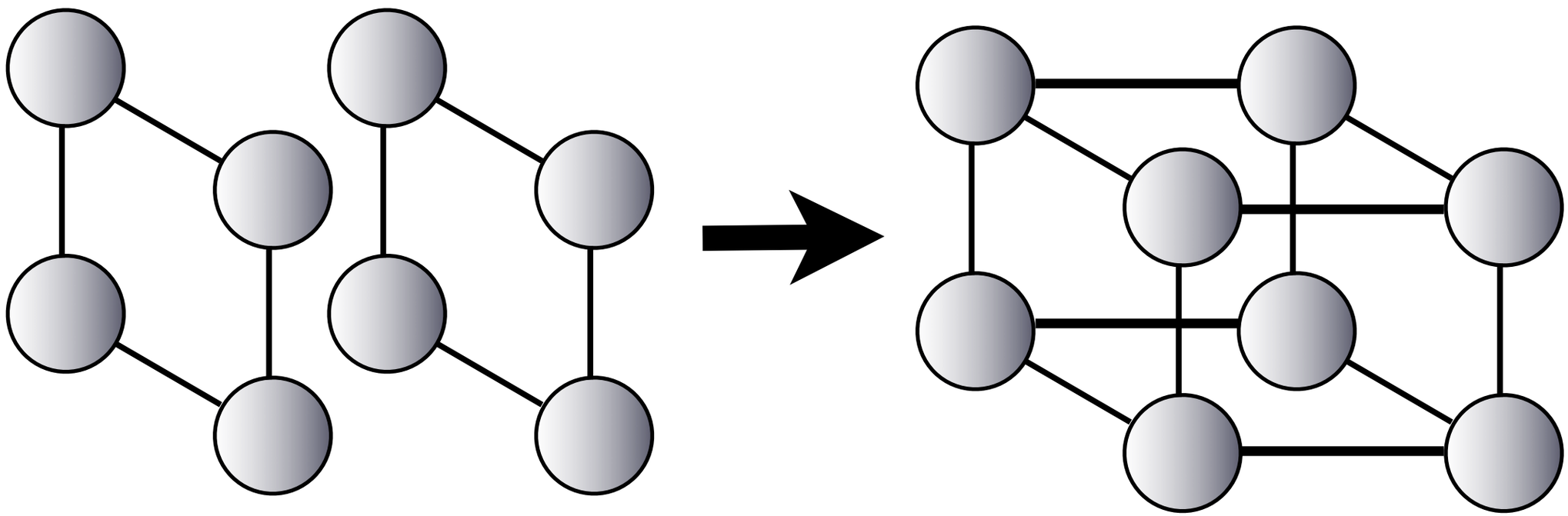
```
successor = (my_rank + 1) % p;
predecessor = (my_rank - 1 + p) % p;

for (i = 0; i < p - 1; i++) {
    send_offset = ((my_rank - i + p) % p)*blocksize;
    recv_offset =
        ((my_rank - i - 1 + p) % p)*blocksize;
    MPI_Send(y + send_offset, blocksize, MPI_FLOAT,
        successor, 0, ring_comm);
    MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
            predecessor, 0, ring_comm, &status);
    }
} /* Allgather_ring */
```

Notice that this implementation assumes that a buffer is available for MPI_Send calls i.e. they will not block the code

A safer implementation would be better (with non-blocking communications, for example)

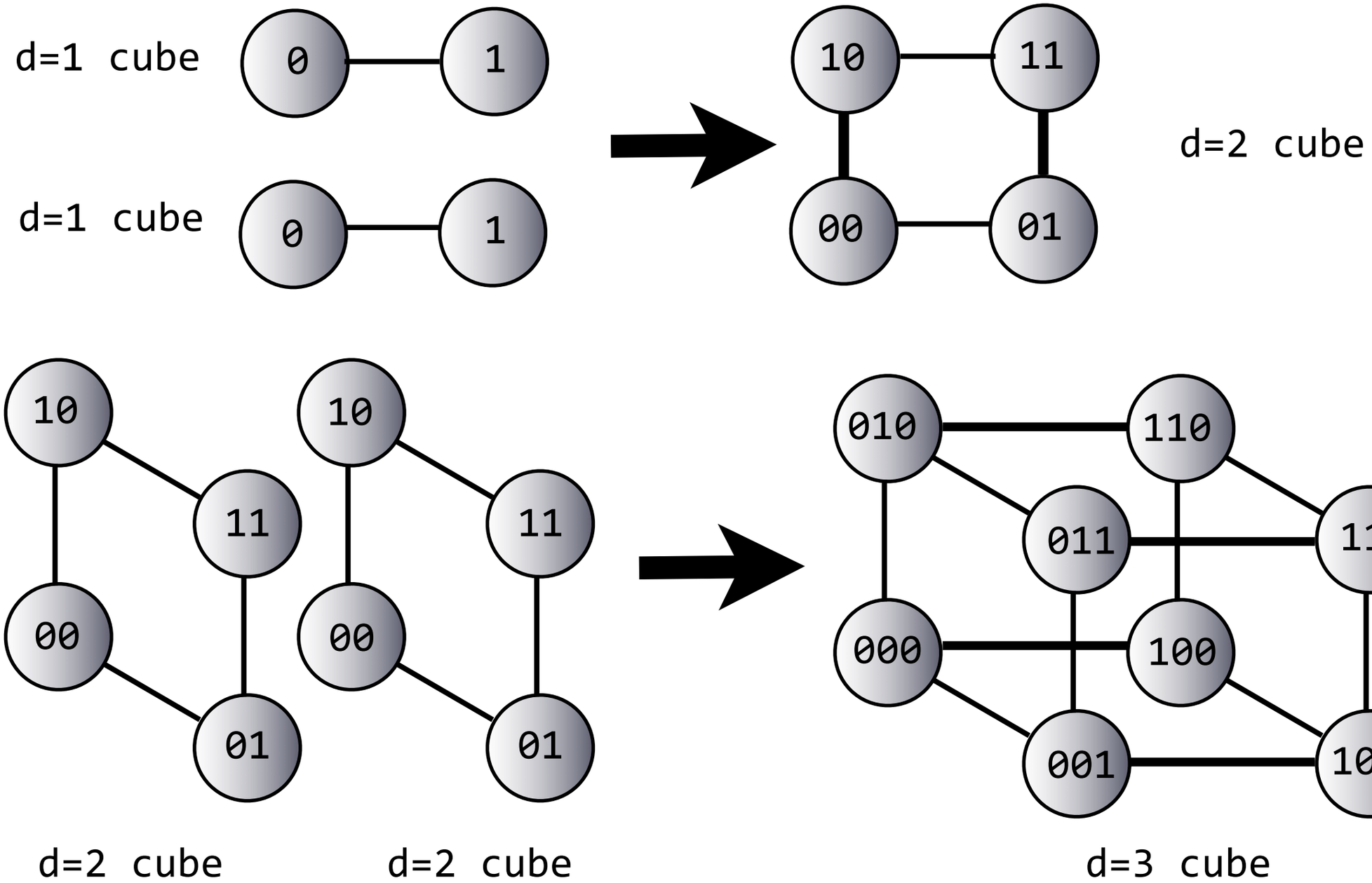
Hypercubes



2d cube 2d cube 3d cube

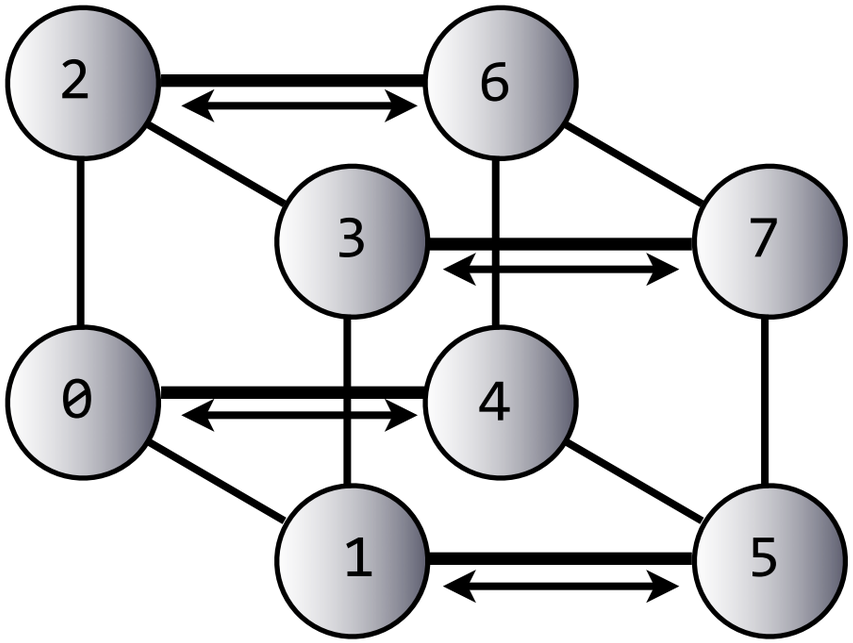
(d+1) dimensional hypercube has 2d vertices

Addressing hypercubes



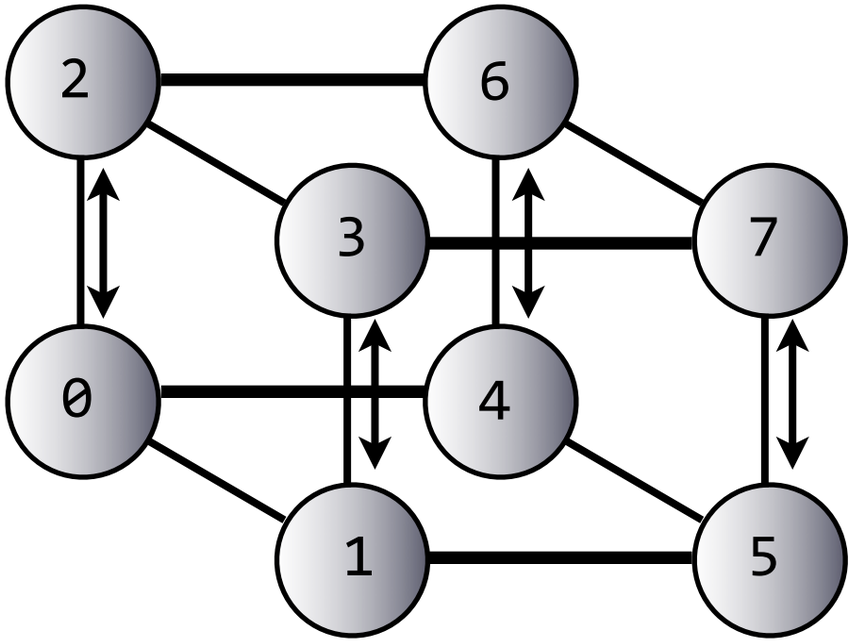
Each vertex thus has a binary address of length d
Vertices are adjacent if they differ by exactly one bit
In stage 1, can split vertices into 2 groups based on first bit
In stage i, do the same based on ith bit

Hypercube data exchange
Pairing between processes

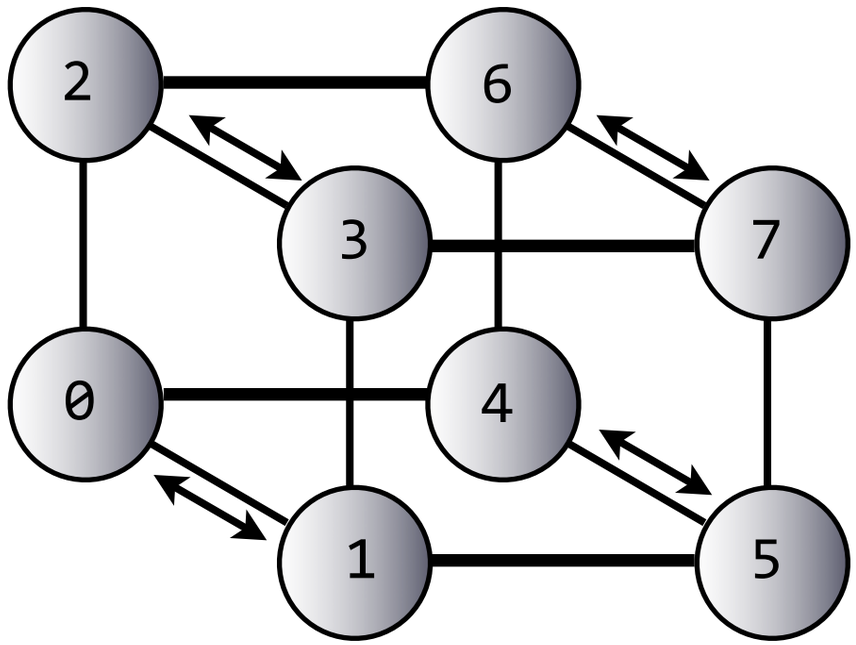


Stage 0

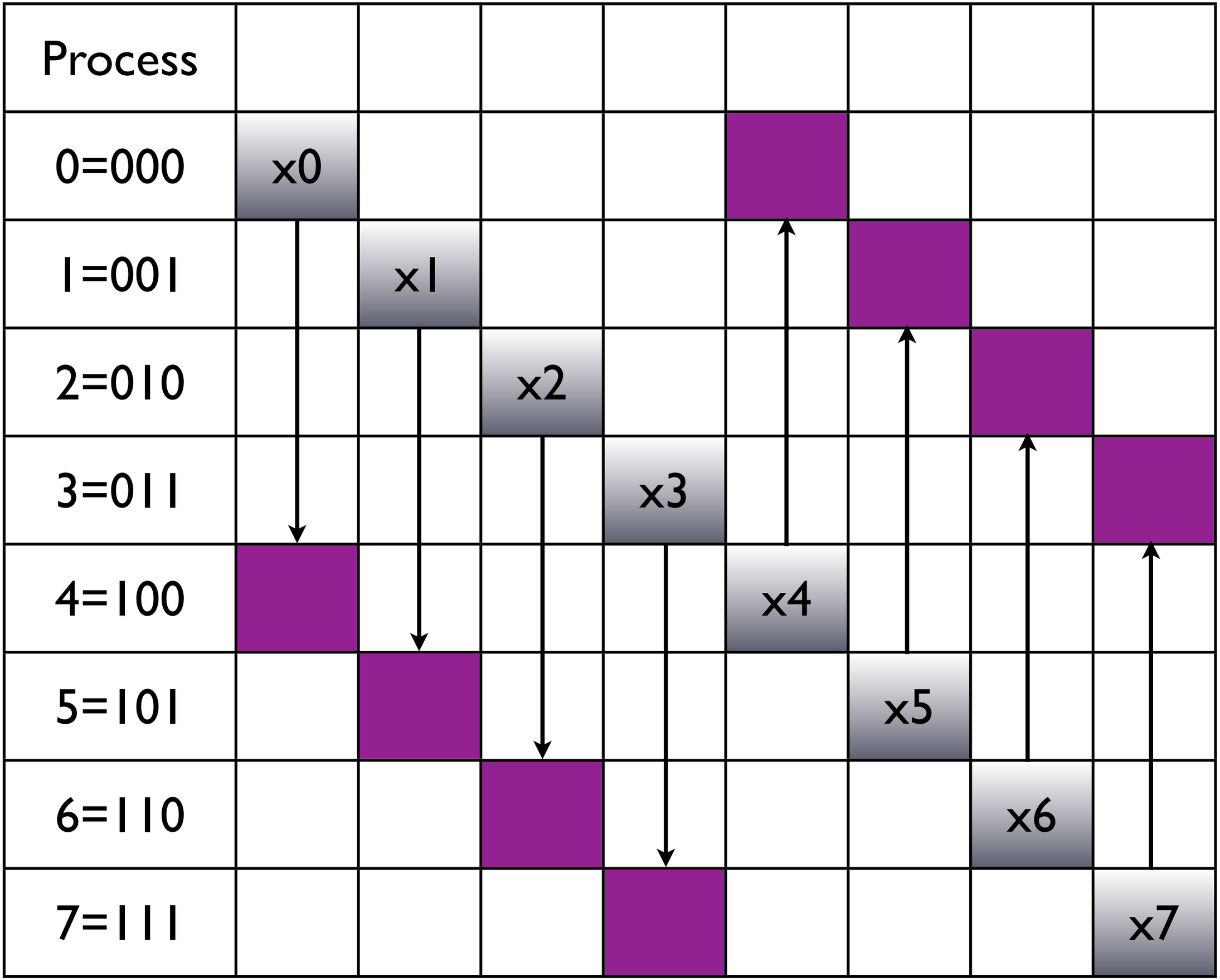
Stage 1



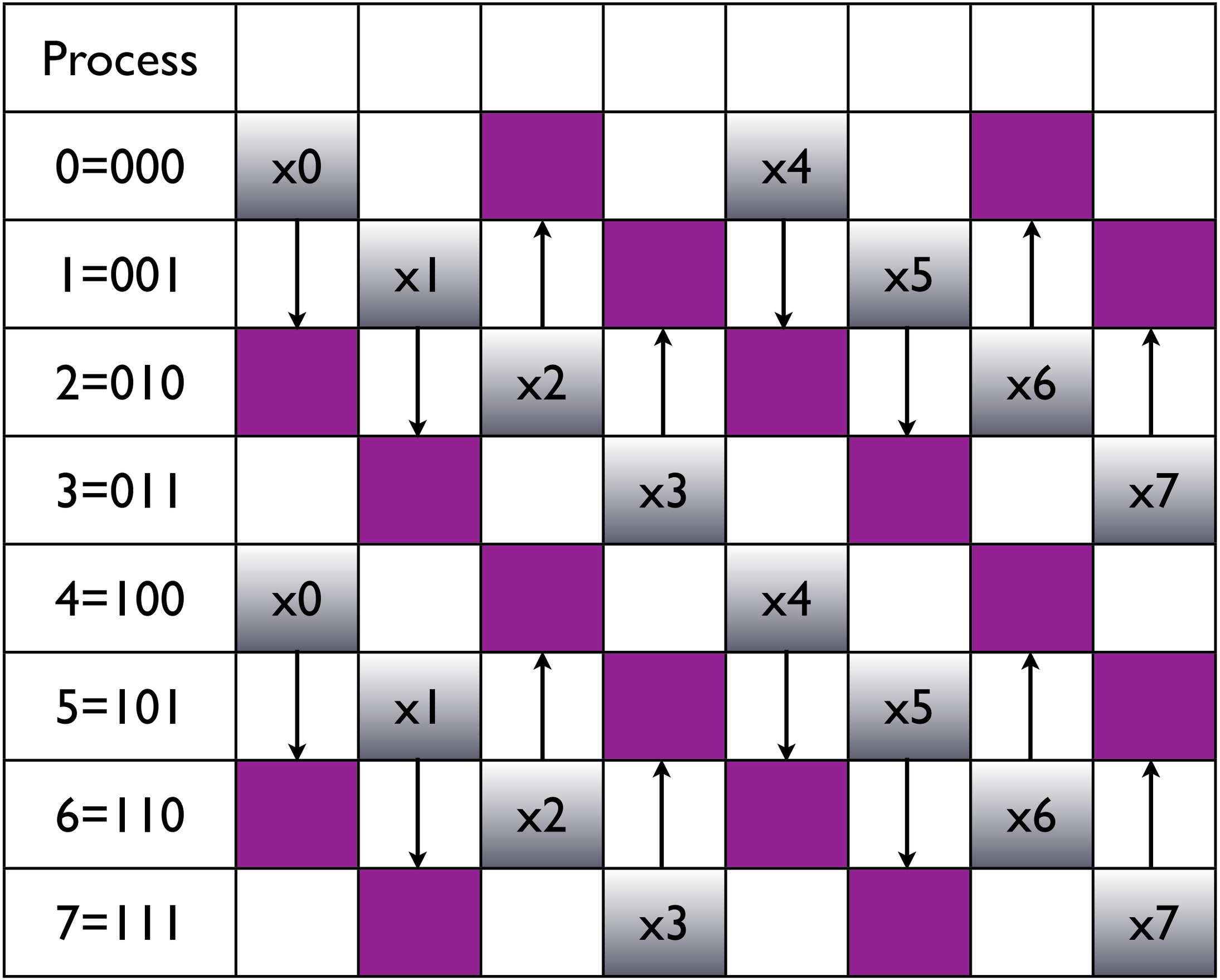
Stage 2



Exchanges during stage 0



Exchanges during stage 1



Exchanges during stage 2

Process								
0=000	x0		x2		x4		x6	
1=001		x1		x3		x5		x7
2=010	x0		x2		x4		x6	
3=011		x1		x3		x5		x7
4=100	x0		x2		x4		x6	
5=101		x1		x3		x5		x7
6=110	x0		x2		x4		x6	
7=111		x1		x3		x5		x7

How do we send this data? The amount and pattern to send is different at each stage.

Various possibilities

1. Build new datatype for each stage (inside the allgather function)
2. Pack/unpack data before/after exchange
3. Use send/receive into contiguous memory locations at every stage and sort data after finishing
4. Prebuild datatypes outside of the allgather function (assume it will be called many times)

We need to determine, at each stage, the rank of the partner process with which data will be exchanged

Remember, during stage 0, the exchange pairs are in a way set by 100

000 001 010 011
100 101 110 111

During stage 1, set by 010

000 001 100 101
010 011 110 111

During stage 2, set by 001

000 010 100 110
001 011 101 111

A	B	A eor
0	0	0
0	1	1
1	0	1
1	1	0

It follows that the process should translate its rank into sequence of bits, then during stage m flip the m-th bit, and take the resulting number as the rank of the process with which communication should be established

```
int log_base2(int p) {
/* Just counts number of bits to right of most significant
 * bit.  So for p not a power of 2, it returns the floor
 * of log_2(p).
 */
    int return_val = 0;
    unsigned q;

    q = (unsigned) p;
    while(q != 1) {
        q = q >> 1;
        return_val++;
    }
    return return_val;
}  /* log_base2 */
```

---

now to find partner for 0 stage exchange, use

```
unsigned eor_bit;
int d;
int partner;

d = log_base2(p);
eor_bit = 1 << (d-1) /* fills low-order bits with 0s */
partner = my_rank ^ eor_bit; /* ^ = bitwise eor in C */
/* eor = “exlusive or” */
```

Outline of algorithm for all the stages

```
/* eor_bit should be unsigned so that right shift will fill leftmost
bits with 0 */
d = log_base2(p);
eor_bit = 1 << (d-1);

for (stage = 0 ; stage < d; stage++) {
partner = my_rank ^ eor_bit;
Build derived data type;
Exchange data;
eor_bit = eor_bit >> 1;
}
```

Exchanges will consist of equally spaced blocks of floats

Recall that MPI_Type_vector is specifically designed for this purpose

```
int MPI_Type_vector(
        int count,
        int blocklen,
        int stride,
        MPI_Datatype old_type,
        MPI_Datatype *newtype )
```

In our example, each process sends one block at stage 0,
two at stage 1, four at stage 2

For a given stage then

```
number_of_blocks = 1 << stage;  /* 2^stage */
stride = (1 << (d-stage))*blocksize;  /* 2^(d-stage) */

MPI_Type_vector(number_of_blocks, blocksize, stride, MPI_FLOAT,
&hole_type);
```

Last, need to compute send and receive offsets

The pattern suggests using bitwise “and” (in C this is “&”)

```
unsigned and_bits;
int d;
int send_offset;
d=log_base2(p);
and_bits= (1 << d) -1 ; /* or just p-1 */
```

The send offset at each stage can be computed via

```
for (stage=0; stage < d; stage++){
send_offset = (my_rank & and_bits)*blocksize;
...
and_bits = and_bits >> 1;
} /* for stage */
```

finally, to compute the receive offset

```
recv_offset = (partner & and_bits)*blocksize
```

```
void  Allgather_cube(
         float    x[]         /* in  */,
         int      blocksize   /* in  */,
         float    y[]         /* out */,
         MPI_Comm comm        /* in  */) {

    int           i, d, p, my_rank;
    unsigned      eor_bit;
    unsigned      and_bits;
    int           stage, partner;
    MPI_Datatype  hole_type;
    int           send_offset, recv_offset;
    MPI_Status    status;

    int log_base2(int p);

    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &my_rank);

    /* Copy x into correct location in y */
    for (i = 0; i < blocksize; i++)
        y[i + my_rank*blocksize] = x[i];
```

```
/* Set up */
d = log_base2(p);
eor_bit = 1 << (d-1);
and_bits = (1 << d) - 1;

for (stage = 0; stage < d; stage++) {
    partner = my_rank ^ eor_bit;
    send_offset = (my_rank & and_bits)*blocksize;
    recv_offset = (partner & and_bits)*blocksize;

    MPI_Type_vector(1 << stage, blocksize,
        (1 << (d-stage))*blocksize, MPI_FLOAT,
        &hole_type);
        MPI_Type_commit(&hole_type);

        MPI_Send(y + send_offset, 1, hole_type,
            partner, 0, comm);
        MPI_Recv(y + recv_offset, 1, hole_type,
            partner, 0, comm, &status);

        MPI_Type_free(&hole_type);  /* Free type so we  */
                  /* can build new type during next pass */
        eor_bit = eor_bit >> 1;
        and_bits = and_bits >> 1;
    }
} /* Allgather_cube */
```