Exercise 5

Introduction to High-Performance Computing

WS 2019/2020

Dr. rer. nat. Marc-André Hermanns

contact@hpc.rwth-aachen.de

Exercise Tasks

1. **Cyclic reduction**
    1. Remove possible deadlocks
    2. Write a complete MPI program
2. **Binomial-tree broadcast**
    1. Visualize communication pattern
    2. Write a complete MPI program
3. **Dissemination barrier**
    1. Visualize communication pattern
    2. Write a complete MPI program
4. **Derived Datatypes**
    1. Transpose matrix during data exchange
    2. Print `size` and `extent`

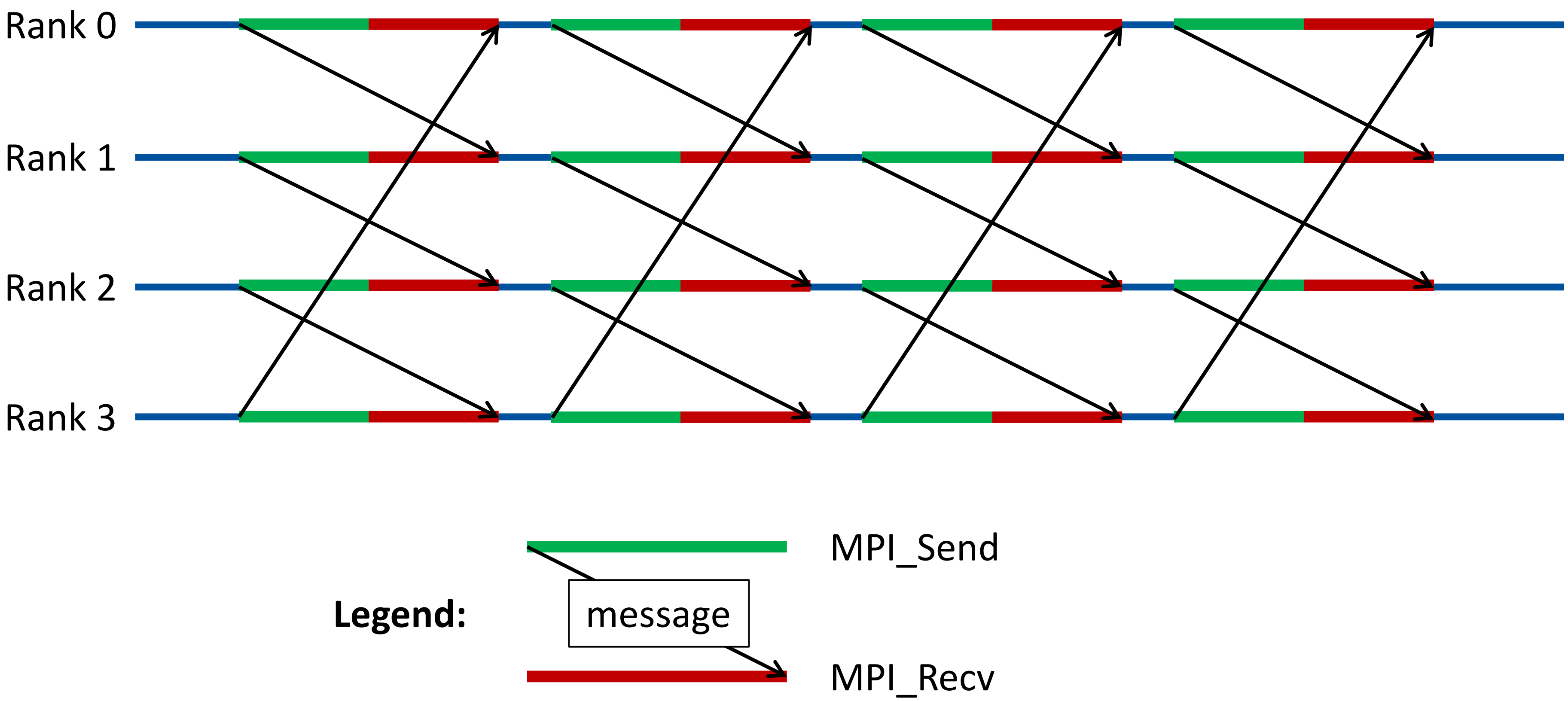
1a. Remove possible deadlocks

- **Original code**

```
MPI_Comm_size(MPI_COMM_WORLD, &size);

for (step = 0; step < size; step++)
{
   /* Send partial result to next rank */
   MPI_Send(result, LEN, MPI_INT, next, 0,
            MPI_COMM_WORLD);
   /* Receive partial result from previous rank */
   MPI_Recv(result, LEN, MPI_INT, prev, 0,
            MPI_COMM_WORLD, &status);
   /* Update local partial result */
   for (i = 0; i < LEN; i++)
      result[i] += local[i];
}
```

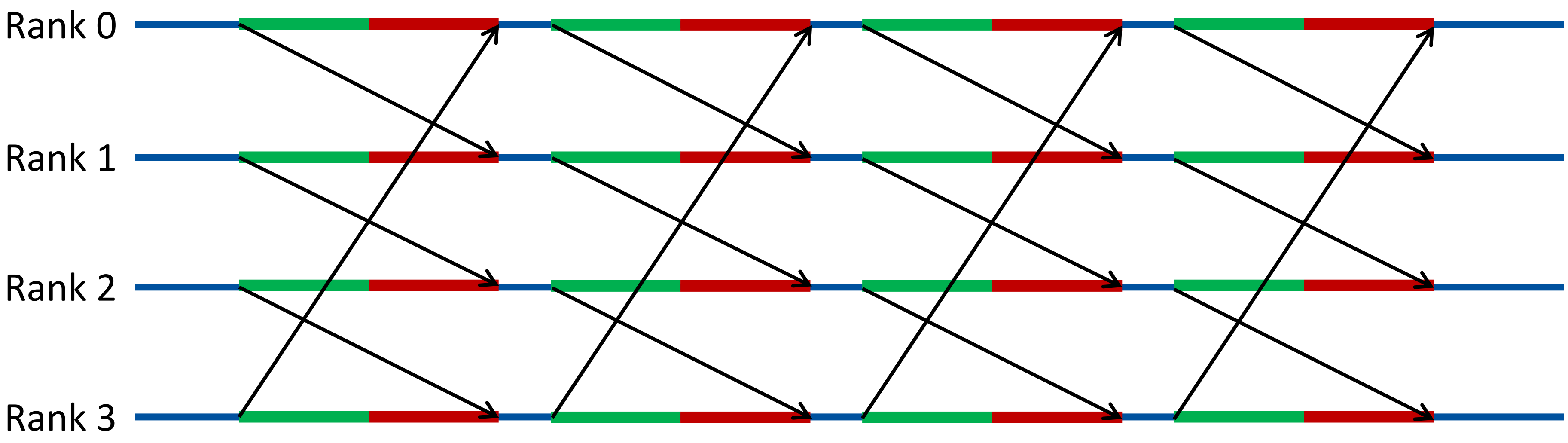
1a. Remove possible deadlocks

- **Intended behaviour**



1a. Remove possible deadlocks

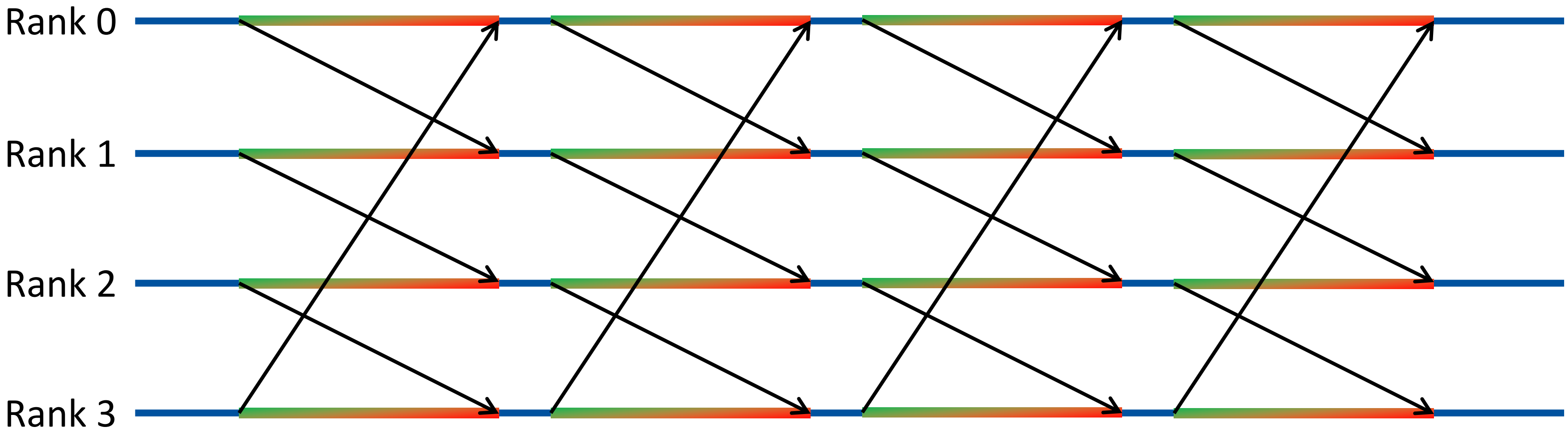
- **Intended behaviour**



- **Problem: Cyclic sender-receiver dependency chain**
  - → It only works if at least one operation in the chain is buffered or non-blocking in order to break the dependency chain

1a. Remove possible deadlocks

- **Approach 1:**



- **Use a combined send and receive call.**

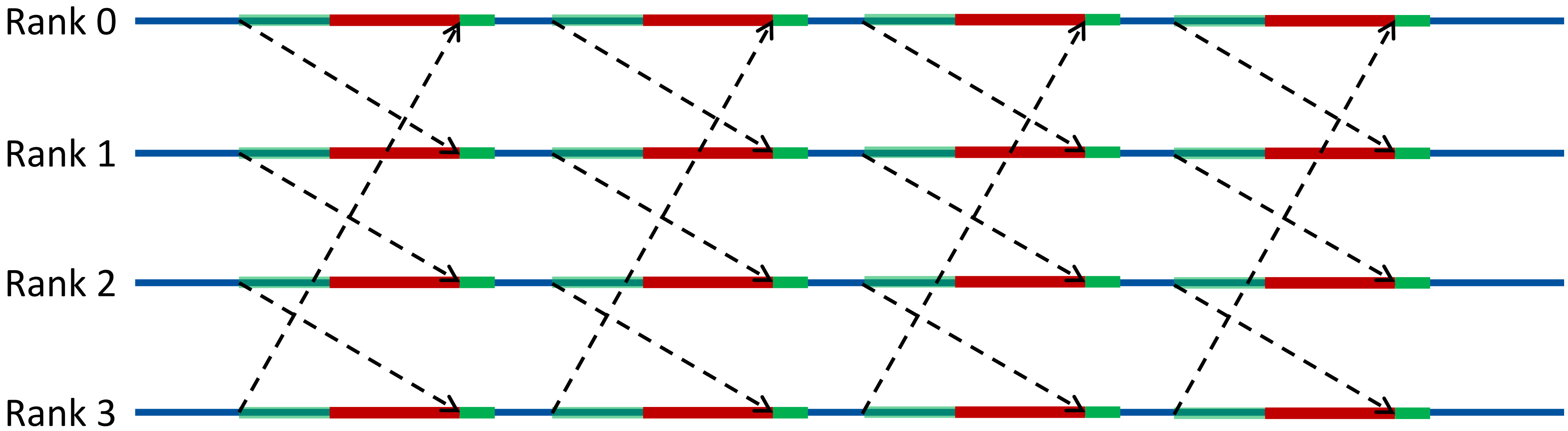
1a. Remove possible deadlocks

- **Use combined send-receive**

```
int temp[LEN];

for (step = 0; step < size; step++)
{
   /* Send partial result to next rank */
   /* Receive partial result from previous rank */
   MPI_Sendrecv(result, LEN, MPI_INT, next, 0,
                temp, LEN, MPI_INT, prev, 0,
                MPI_COMM_WORLD, &status);

   /* Update local partial result */
   for (i = 0; i < LEN; i++)
      result[i] = temp[i] + local[i];
}
```

1a. Remove possible deadlocks

- **Approach 2:**



- **Non-blocking send, receive and finally make sure all is sent.**

1a. Remove possible deadlocks

- **Use non-blocking send**

```
MPI_Request request;
for (step = 0; step < size; step++)
{
   /* Non-blocking send partial result to next rank */
   MPI_Isend(result, LEN, MPI_INT, next, 0,
             MPI_COMM_WORLD, &reqest);
   /* Receive partial result from previous rank */
   MPI_Recv(temp, LEN, MPI_INT, prev, 0,
            MPI_COMM_WORLD, &status);
   /* Wait for the send to complete */
   MPI_Wait(&request, MPI_STATUS_IGNORE);
   /* Update local partial result */
   for (i = 0; < LEN; i++)
      result[i] = temp[i] + local[i];
}
```

1a. Remove possible deadlocks

- **Can a non-blocking receive be used?**

```
MPI_Request request;
for (step = 0; step < size; step++)
{
   /* Send partial result to next rank */
   MPI_Send(result, LEN, MPI_INT, next, 0,
            MPI_COMM_WORLD);
   /* Non-blocking receive from previous rank */
   MPI_Irecv(temp, LEN, MPI_INT, prev, 0,
             MPI_COMM_WORLD, &request);
   /* Wait for the send to complete */
   MPI_Wait(&request, &status);
   /* Update local partial result */
   for (i = 0; < LEN; i++)
      result[i] = temp[i] + local[i];
}
```

1a. Remove possible deadlocks

- **Can a non-blocking receive be used?**

```
MPI_Request request;
for (step = 0; step < size; step++)
{
   /* Se[illegible]d partial result to next ra[illegible]k */
   MPI_[illegible]esult, LEN, MPI_INT[illegible] 0,
   [illegible]OMM_WORLD);
   /* Non-bloc[illegible]ei[illegible] previous rank */
   MPI_Irecv(temp,[illegible]INT, prev, 0,
             MPI[illegible]request);
   /* Wait f[illegible]send to[illegible]e */
   MPI_[illegible]quest, &status);
   /* Up[illegible]e local partial result [illegible]
   for (i = 0; < LEN; i++)
      result[i] = temp[i] + local[i];
}
```

MPI_Recv

1a. Remove possible deadlocks

- Use non-blocking receive (the right way)

```
MPI_Request request;
for (step = 0; step < size; step++)
{
   /* Post non-blocking receive */
   MPI_Irecv(temp, LEN, MPI_INT, prev, 0,
             MPI_COMM_WORLD, &request);
   /* Send partial result to next rank */
   MPI_Send(result, LEN, MPI_INT, next, 0,
            MPI_COMM_WORLD);
   /* Wait for the receive to complete */
   MPI_Wait(&request, &status);
   /* Update local partial result */
   for (i = 0; < LEN; i++)
      result[i] = temp[i] + local[i];
}
```

1a. Remove possible deadlocks

- **Non-blocking procedures return before operation is complete**
  - → Cannot use the same buffer in both send and receive operations
  - → Temporary buffer needed
    - → No data dependency between the send and receive operations, therefore can be executed in any order using two separate buffers
    - → Watch out for the increased memory footprint
  - → Space/time trade-off
- **A single buffer could be used too (but slower)**

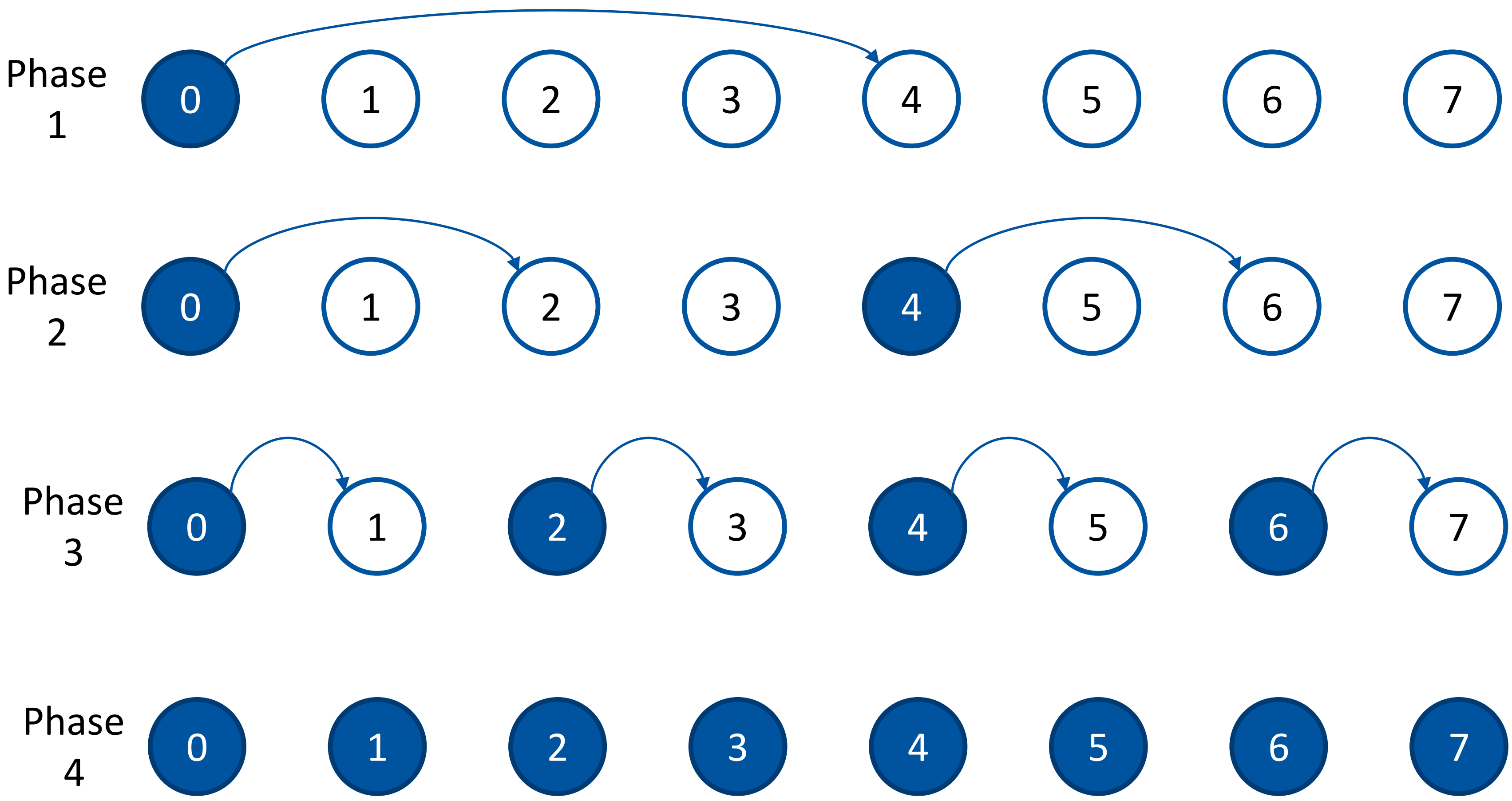
```
/* Send partial result to next rank */
/* Receive partial result from previous rank */
MPI_Sendrecv_replace(result, LEN, MPI_INT, next, 0,
             prev, 0, MPI_COMM_WORLD, &status);
```

1a. Remove possible deadlocks

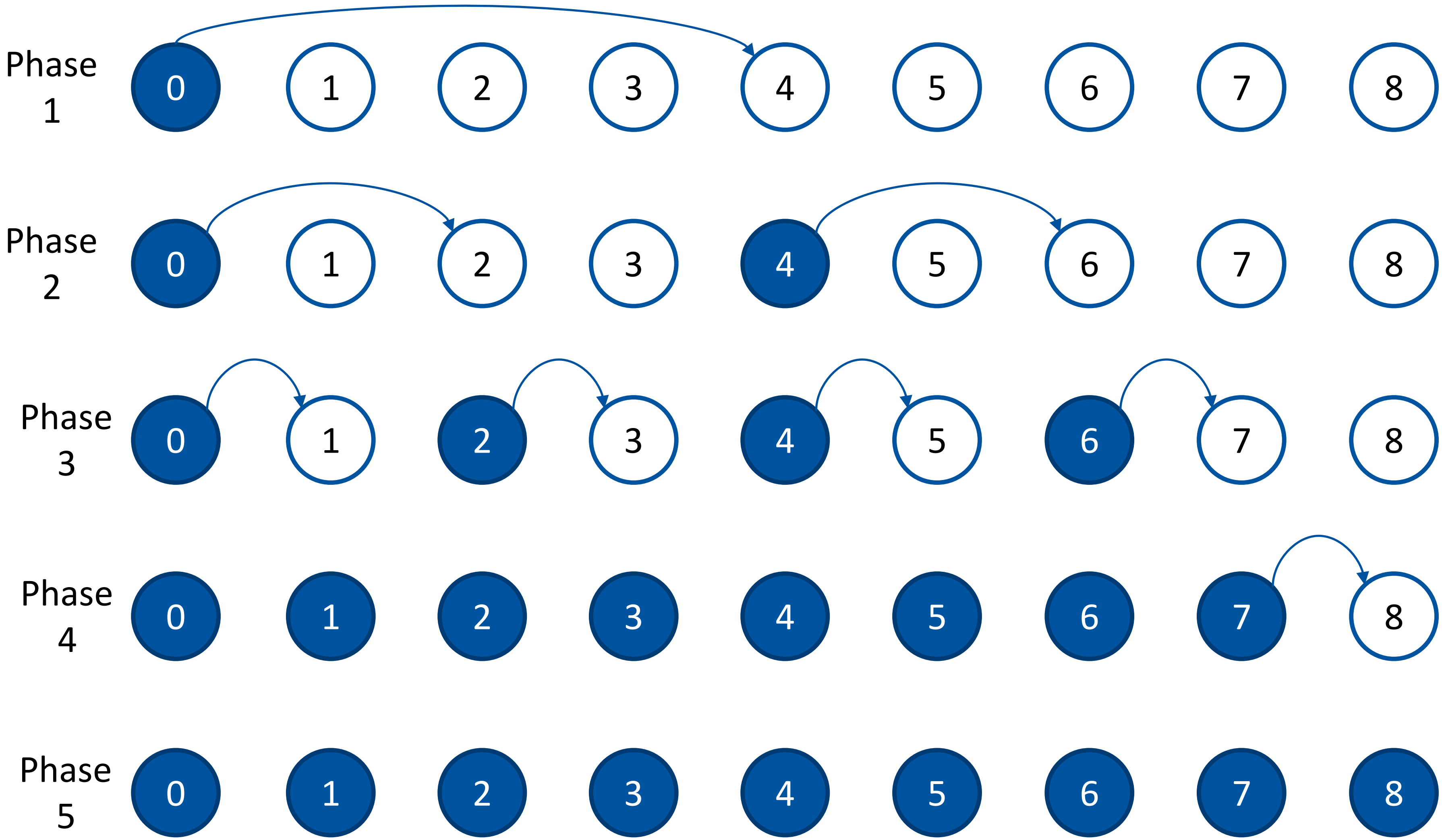
- **Approach 3: Use buffered send**

```
MPI_Pack_size(LEN, MPI_INT, MPI_COMM_WORLD, &msize);
int bsize = (msize + MPI_BSEND_OVERHEAD) * size;
void* buffer = malloc(bsize);
MPI_Buffer_attach(buffer, bsize);
for (step = 0; step < size; step++)
{
   /* Send partial result to next rank */
   MPI_Bsend(result, LEN, MPI_INT, next, 0,
            MPI_COMM_WORLD);
   /* Receive partial result from previous rank */
   MPI_Recv(result, LEN, MPI_INT, prev, 0,
            MPI_COMM_WORLD, &status);
   /* Update local partial result */
   for (i = 0; < LEN; i++)
      result[i] += local[i];
}
MPI_Buffer_detach(&buffer, &bsize);
```

2.1 Binomial-Tree Broadcast
Powers of 2

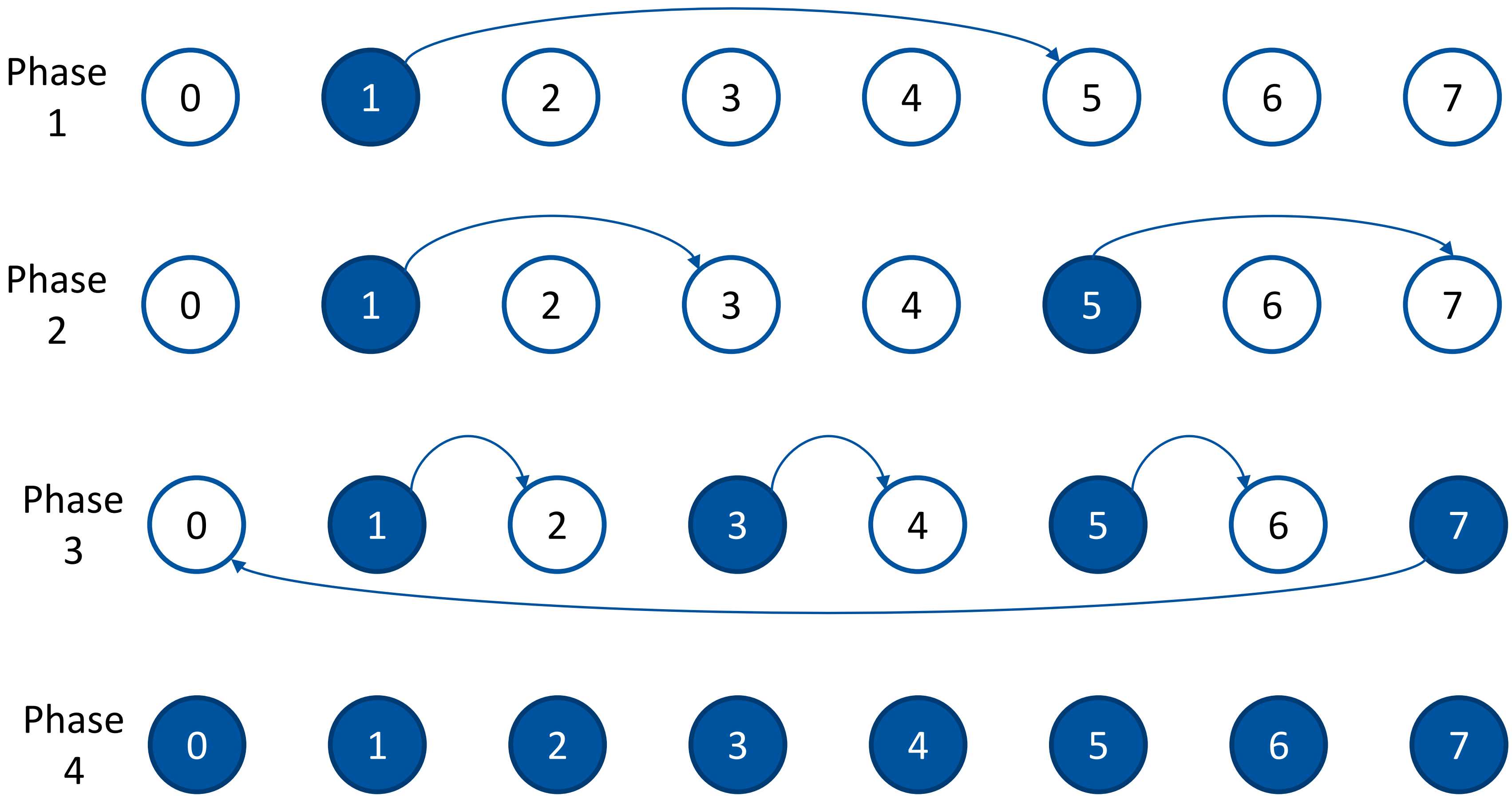


2.1 Binomial-Tree Broadcast
Non Powers of 2



2.2 Binomial-Tree Broadcast

Arbitrary root ranks



- **Setting up**

```
void my_broadcast(int* buffer, int count, MPI_Datatype datatype,
                  int root, MPI_Comm comm)
{
    int my_rank, num_ranks;
    MPI_Comm_rank(comm, &my_rank);
    MPI_Comm_size(comm, &num_ranks);

    int tag         = 0;
    int interval_ub = num_ranks;
    int distance    = interval_ub / 2;

    int sender = 0;
    int phase  = 0;
    int shifted_rank = (my_rank - root + num_ranks) % num_ranks;

    […]
}
```

- **Determining sender/destination and exchange data**

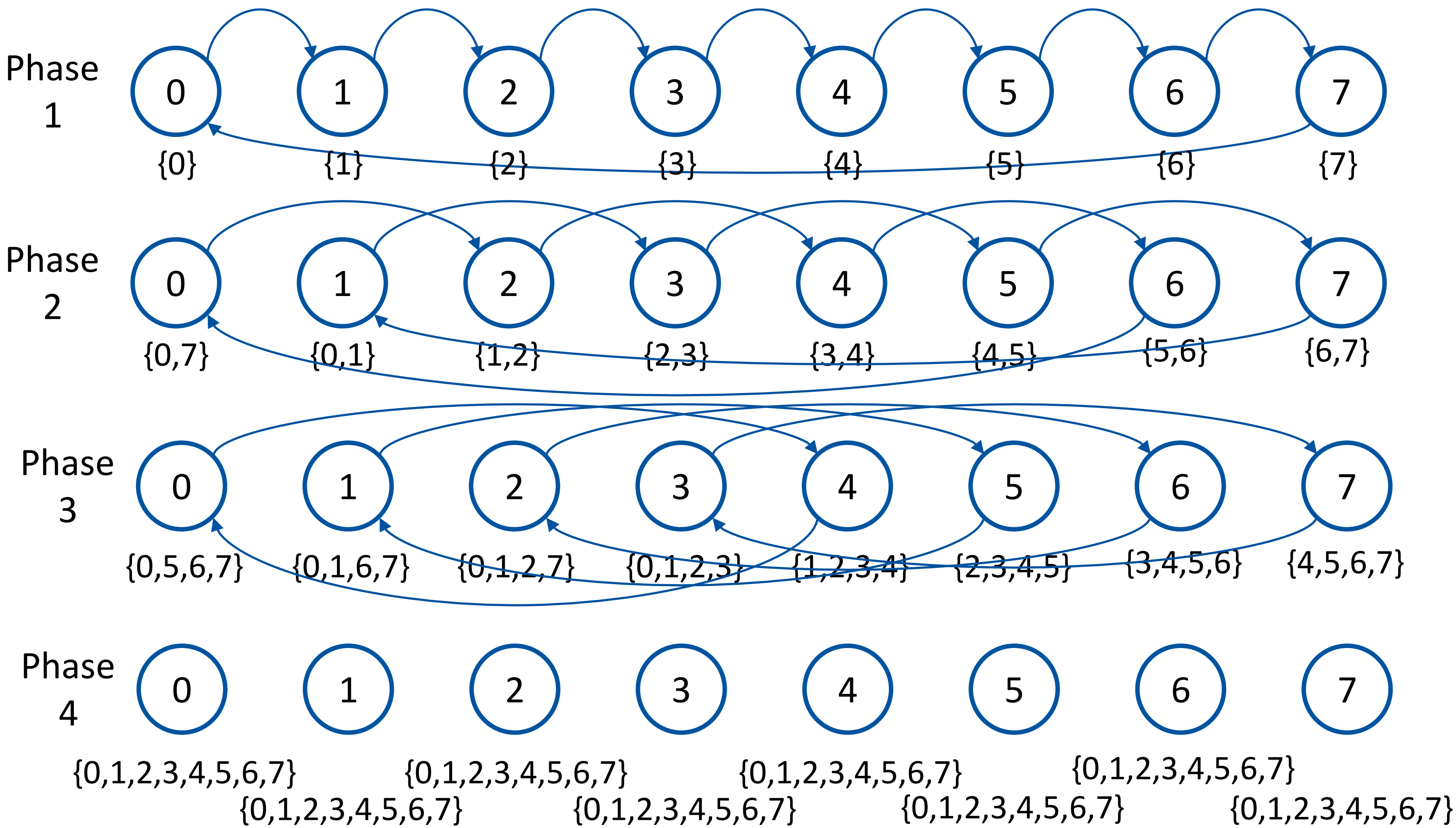
```
void my_broadcast(int* buffer, int count, MPI_Datatype datatype,
                  int root, MPI_Comm comm)
{
    […]
    while (distance > 0)
    {
        int destination = sender + (interval_ub - sender) / 2 ;
        int shifted_destination = (destination + root) % num_ranks;
        int shifted_sender = (sender + root) % num_ranks;

        if (my_rank == shifted_sender) {
            MPI_Send(buffer, count, datatype, shifted_destination,
                     tag, comm);
        } else if (my_rank == shifted_destination) {
            MPI_Recv(buffer, count, datatype, shifted_sender, tag,
                     comm, MPI_STATUS_IGNORE);
        }
        […]
    }
}
```

- **Prepare next phase**

```
void my_broadcast(int* buffer, int count, MPI_Datatype datatype,
                  int root, MPI_Comm comm)
{
    […]
    while (distance > 0)
    {
        […]

        if (destination <= shifted_rank)
            sender = destination;

        if (destination > shifted_rank)
            interval_ub = destination;

        distance = (interval_ub - sender) / 2;
    }
}
```

3.1 Dissemination Pattern

Powers of 2



3.2 Dissemination Pattern

Barrier Implementation

```
void my_barrier(MPI_Comm comm)
{
    int my_rank, num_ranks, tag = 0;
    MPI_Comm_rank(comm, &my_rank);
    MPI_Comm_size(comm, &num_ranks);

    for (int distance = 1; distance < num_ranks; distance *= 2)
    {
        int destination = (my_rank + distance) % num_ranks;
        int source      = (my_rank - distance + num_ranks) % num_ranks;
        MPI_Request request;
        MPI_Irecv(NULL, 0, MPI_INT, source, tag, comm, &request);
        MPI_Ssend(NULL, 0, MPI_INT, destination, tag, comm);
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        printf("Rank %i receiving from %i sending to %i\n",
          my_rank, source, destination);
    }
}
```

- **Derived datatypes are just descriptions on how to read/write data**
  - → `MPI_Type_contiguous` provides row-wise access
  - → `MPI_Type_vector` provides column-wise access

- **Combination of both access patterns transposes matrix**
  - → Read column and write row
  - → Read row and write column

- **Sender transposing matrix through vector type**

```
[…]
if (my_rank == 0)
{
    int count = 10, blocklength = 1, stride = 10;
    MPI_Type_vector(count, blocklength, stride, MPI_INT, &vectortype);
    MPI_Type_commit(&vectortype);

    print_typeinfo(vectortype, "vector");
    print_matrix(matrix, 10);

    for (int i = 0; i < 10; i++)
        MPI_Send(&matrix[i], 1, vectortype, 1, tag, MPI_COMM_WORLD);

    MPI_Type_free(&vectortype);
}
else
{
    […]
}
```

- **Receiver stores incoming data as contiguous buffer**

```
[…]
if (my_rank == 0)
{
    […]
}
else
{
    MPI_Type_contiguous(10, MPI_INT, &contigtype);
    MPI_Type_commit(&contigtype);

    for (int i = 0; i < 10; i++)
        MPI_Recv(&matrix[i*10], 1, contigtype, 0, tag, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

    print_typeinfo(contigtype, "contiguous");
    print_matrix(matrix, 10);

    MPI_Type_free(&contigtype);
}
```