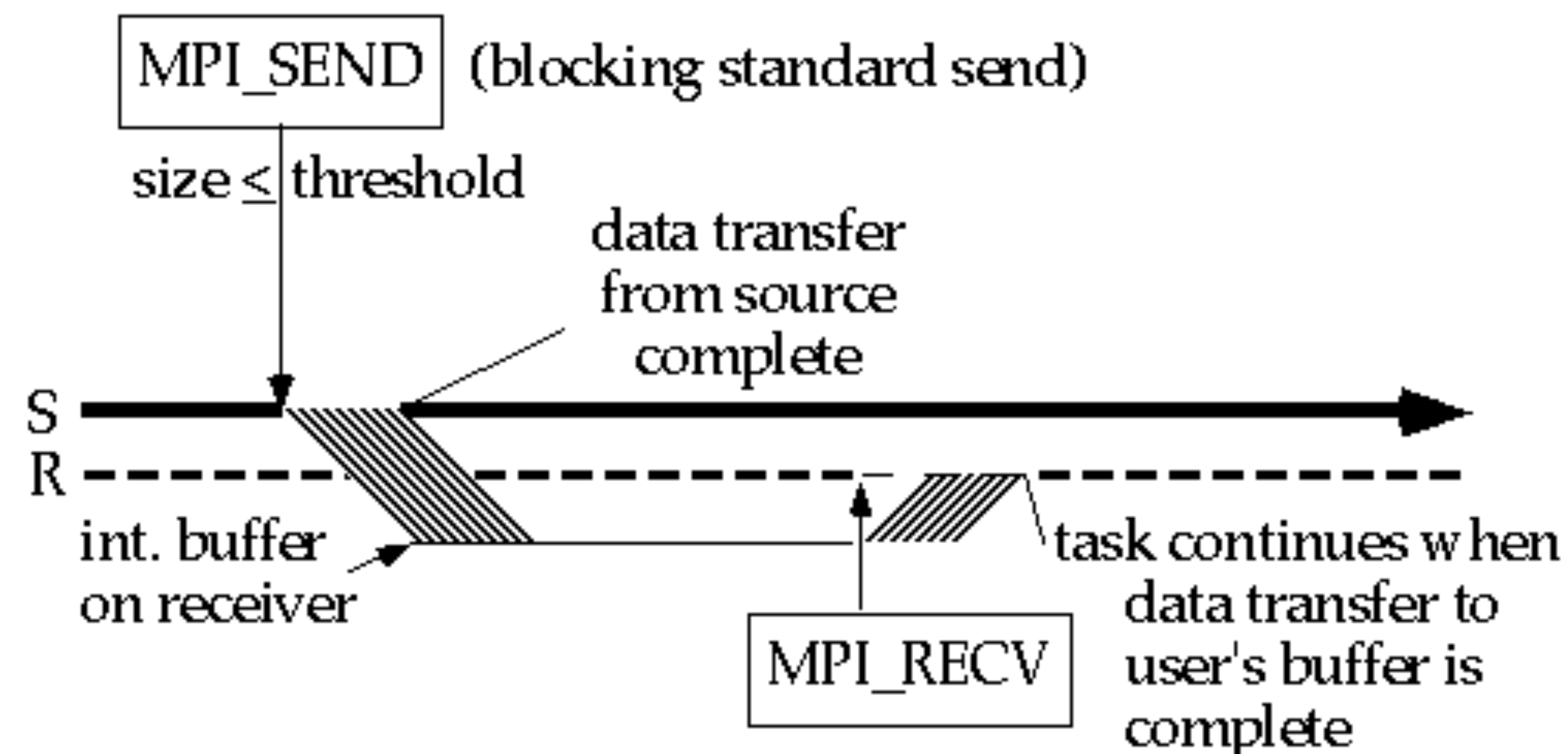
(continuation from last lecture)

Blocking and Non-Blocking Communication

So far we have been using blocking communication:

- MPI_Recv does not complete until the buffer is full (available for use).
- MPI_Send does not complete until the buffer is empty (available for use).

Completion depends on size of message and amount of system buffering.



https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/blocking-communication

Non-Blocking Send/Receive

Non-blocking operations **return (immediately)** “request handles” that can be tested and waited on:

- `MPI_Request request;`
- `MPI_Status status;`
- `MPI_Isend(…);`
- `MPI_Irecv(…);`
- `MPI_Wait(&request, &status);` (each request must be Waited on)

One can also test without waiting:

`MPI_Test(&request, &flag, &status);`

Accessing the data buffer without waiting is undefined

MPI_Isend

```
int MPI_Isend(const void* buffer,
              int count,
              MPI_Datatype datatype,
              int recipient,
              int tag,
              MPI_Comm communicator,
              MPI_Request* request);
```

Handle on non-blocking operation (used later to check for completion)

- `MPI_Isend` is the standard **non-blocking send** (I stands for immediate return).
- The user must not attempt to reuse the buffer after `MPI_Isend` returns without explicitly **checking for completion**.

https://rookiehpc.org/mpi/docs/mpi_isend/index.html

`MPI_Wait` and `MPI_Test`

```
int MPI_Wait(MPI_Request* request,
             MPI_Status* status);
```

`MPI_Wait` **waits** for a non-blocking operation to complete and will block until the underlying operation is done.

```
int MPI_Test(MPI_Request* request,
             int* flag,
             MPI_Status* status);
```

`MPI_Test` checks if a non-blocking operation is complete at a given time. **It will not wait** for the underlying non-blocking operation to complete.

https://rookiehpc.org/mpi/docs/mpi_wait/index.html
https://rookiehpc.org/mpi/docs/mpi_test/index.html

Two processes

MPI_Isend Example - Sender

```
   switch(my_rank)
    {
        case SENDER:
        {
            int buffer_sent = 12345;
            MPI_Request request;
            printf("MPI process %d sends value %d.\n", my_rank, buffer_sent);
            MPI_Isend(&buffer_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

            // Do other things while the MPI_Isend completes
            // <...>

            // Let's wait for the MPI_Isend to complete before progressing
further.
            MPI_Wait(&request, MPI_STATUS_IGNORE);
            break;
        }
        // RECEIVER CASE ON NEXT SLIDE
    }
```

https://rookiehpc.org/mpi/docs/mpi_isend/index.html

Two processes

MPI_Isend Example - Sender

```
    switch(my_rank)
     {
         case SENDER:
         {
             int buffer_sent = 12345;
             MPI_Request request;
             printf("MPI process %d sends value %d.\n", my_rank, buffer_sent);
             MPI_Isend(&buffer_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

             // Do other things while the MPI_Isend completes
             // <...>

             // Let's wait for the MPI_Isend to complete before progressing
further.
             MPI_Wait(&request, MPI_STATUS_IGNORE);
             break;
         }
         // RECEIVER CASE ON NEXT SLIDE
     }
```

MPI process 0 sends value 12345

https://rookiehpc.org/mpi/docs/mpi_isend/index.html

MPI_Isend Example - Sender

Two processes

```
   switch(my_rank)
    {
        case SENDER:
        {
            int buffer_sent = 12345;
            MPI_Request request;
            printf("MPI process %d sends value %d.\n", my_rank, buffer_sent);
            MPI_Isend(&buffer_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

            // Do other things while the MPI_Isend completes
            // <...>

            // Let's wait for the MPI_Isend to complete before progressing
further.
            MPI_Wait(&request, MPI_STATUS_IGNORE);
            break;
        }
        // RECEIVER CASE ON NEXT SLIDE
    }
```

MPI process 0 sends value 12345

Only continue after the send is done

https://rookiehpc.org/mpi/docs/mpi_isend/index.html

Two processes

MPI_Isend Example - Receiver

```
switch(my_rank)
 {
     // CASE SENDER FROM BEFORE NOT SHOWN
     case RECEIVER:
     {
         int received;
         MPI_Recv(&received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("MPI process %d received value: %d.\n", my_rank, received);
         break;
     }
 }
```

Does not need to be a special recv

MPI process 1 received value 12345

https://rookiehpc.org/mpi/docs/mpi_isend/index.html

MPI_Irecv

```
int MPI_Irecv(void* buffer,
              int count,
              MPI_Datatype datatype,
              int sender,
              int tag,
              MPI_Comm communicator,
              MPI_Request* request);
```

Handle on non-blocking operation (used later to check for completion)

- MPI_Isend is the standard **non-blocking receive** (I stands for immediate return).
- To know whether the message has been received, you must use `MPI_Wait` or `MPI_test` on the `MPI_Request`.

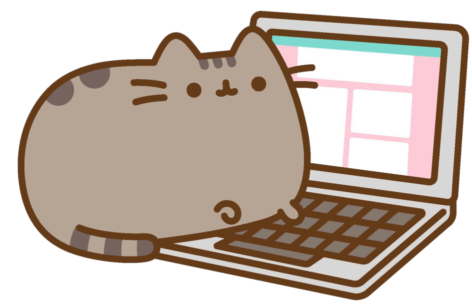
https://rookiehpc.org/mpi/docs/mpi_irecv/index.html

Summary

- There are many different network topologies, each with their own strengths and weaknesses.
- The main performance metrics of a network are **latency and bandwidth**.
- MPI defines a standard for programming distributed-memory machines.
- The core functionality for message passing is **send and receive** (and variants)

CSE 6230:
HPC Tools and Applications

+



Lecture 12: Advanced MPI & Analysis of Distributed Algorithms

Helen Xu

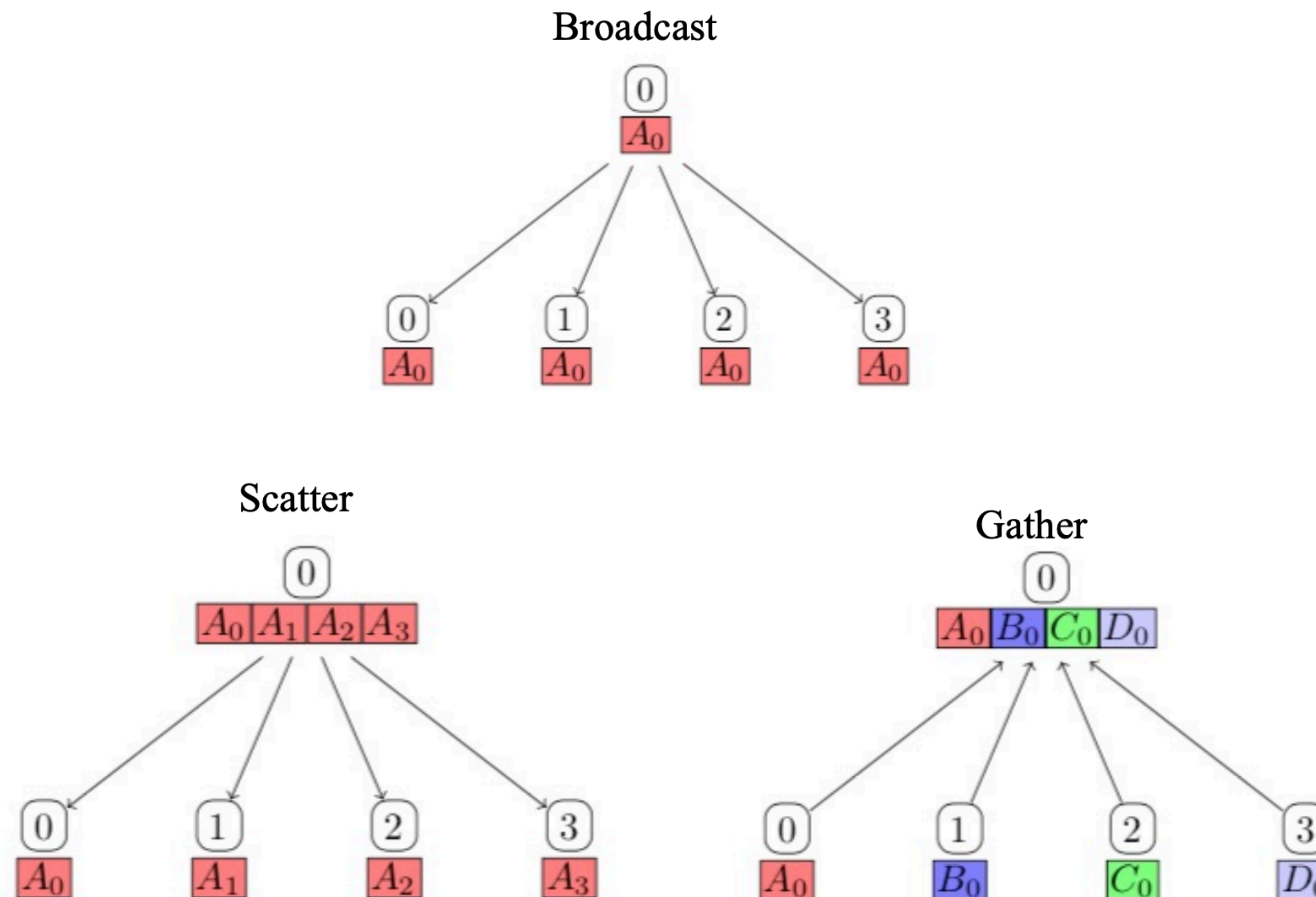
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

Collective Communication

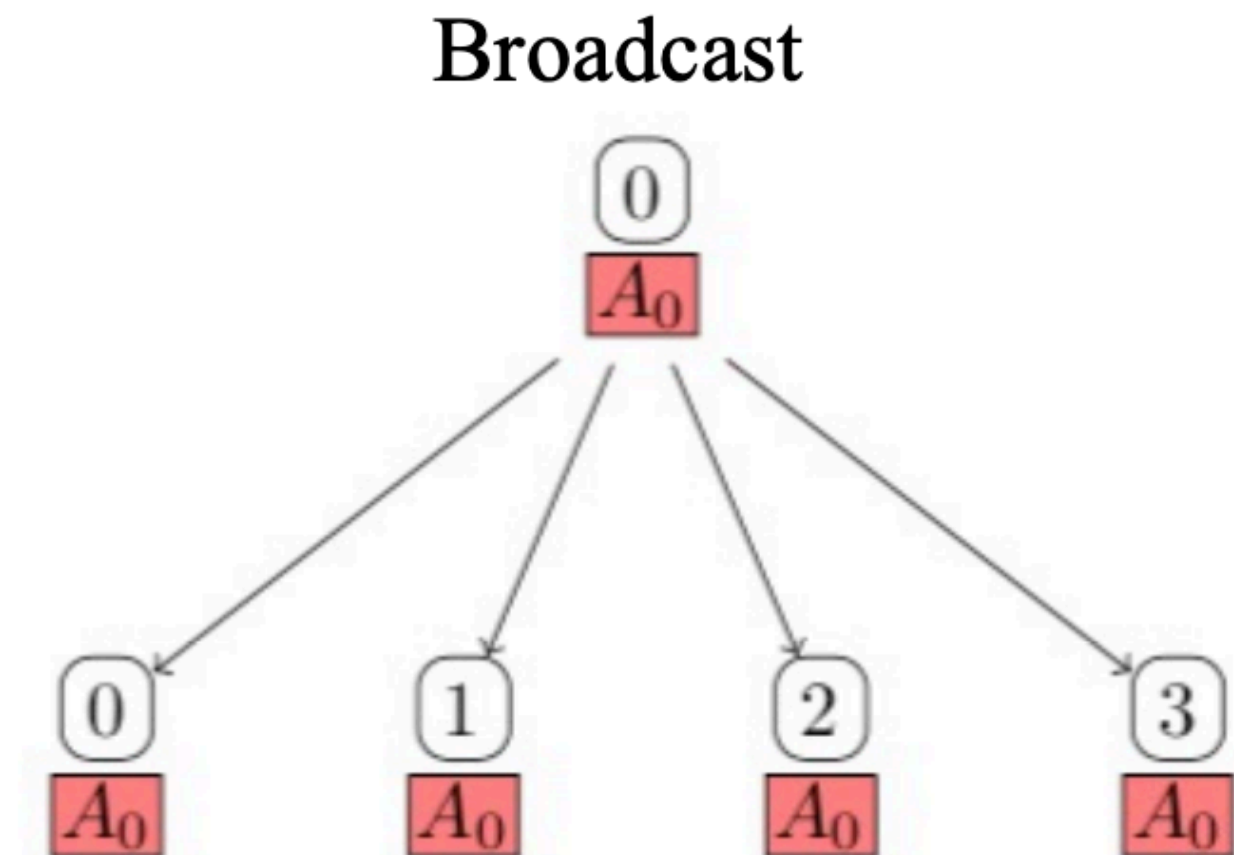
In collective communication, one process wants to communicate with multiple processes.

All collective operations must be called by all processes in the communicator



MPI_Bcast

```
int MPI_Bcast(void* buffer,
              int count,
              MPI_Datatype datatype,
              int emitter_rank,
              MPI_Comm communicator);
```

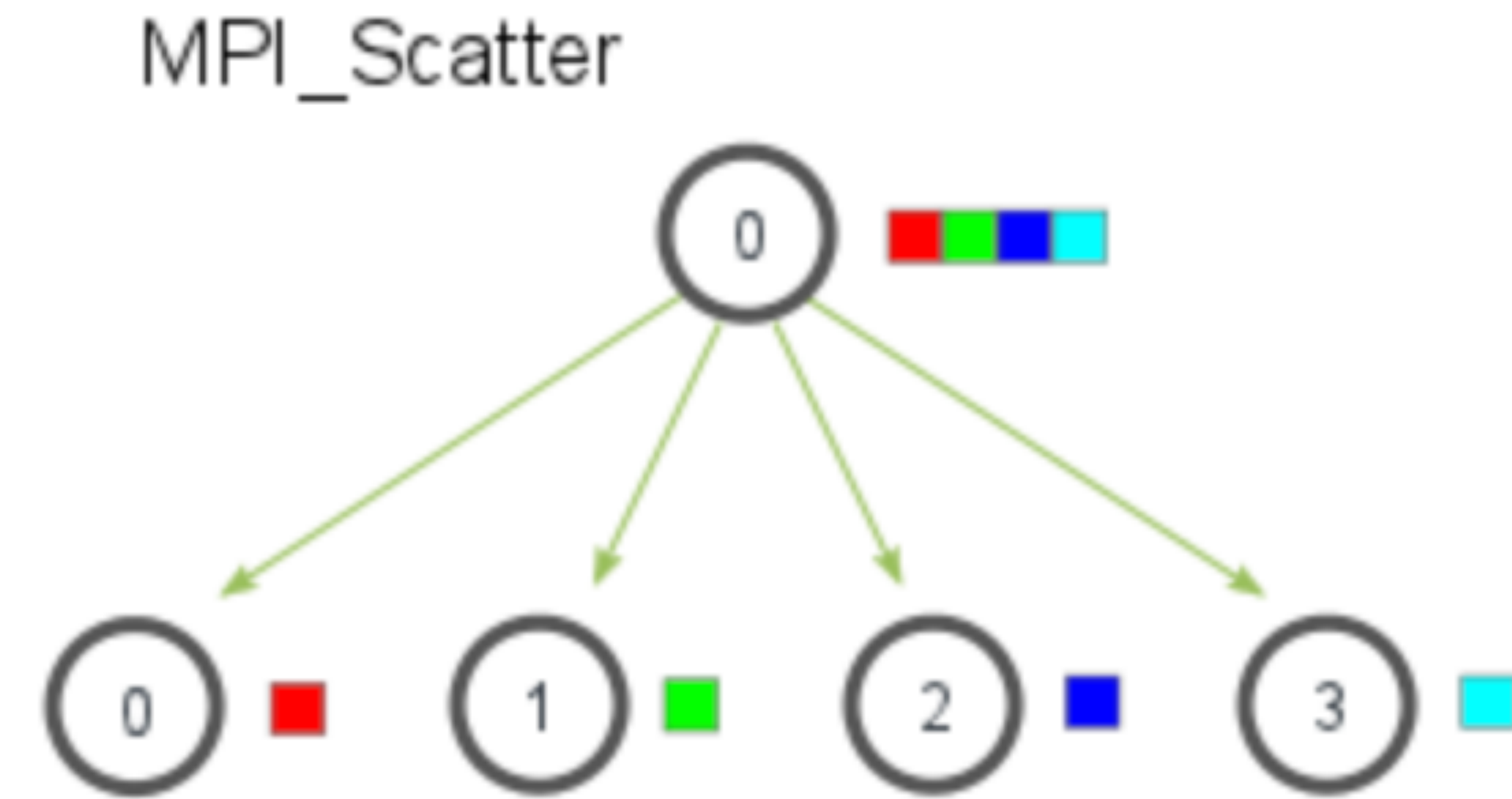


- `MPI_Bcast` broadcasts a message from a process to all other processes in the same communicator.
- This is a collective operation; it must be called by **all processes in the communicator**.

https://rookiehpc.org/mpi/docs/mpi_bcast/index.html

MPI_Scatter

```
MPI_Scatter(
   void* send_data,
   int send_count,
   MPI_Datatype send_datatype,
   void* recv_data,
   int recv_count,
   MPI_Datatype recv_datatype,
   int root,
   MPI_Comm communicator)
```

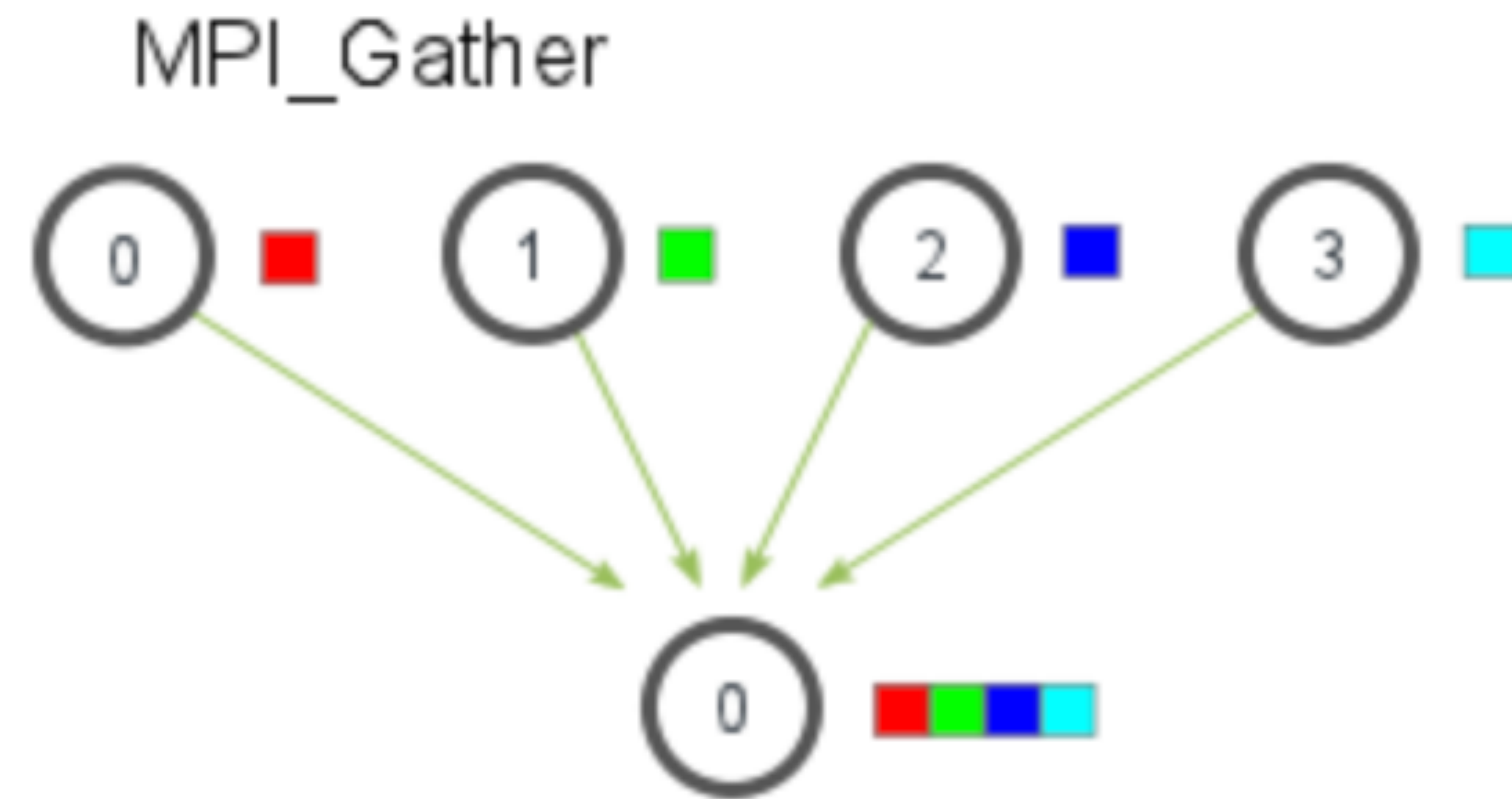


`MPI_Scatter` takes an array of elements and **distributes the elements in order of process rank**. Process 0 gets the first chunk, process 1 gets the next, and so on.

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

MPI_Gather

```
MPI_Gather(
   void* send_data,
   int send_count,
   MPI_Datatype send_datatype,
   void* recv_data,
   int recv_count,
   MPI_Datatype recv_datatype,
   int root,
   MPI_Comm communicator)
```



`MPI_Gather` is the **inverse of scatter** - it takes elements from many processes and gathers them into one process.

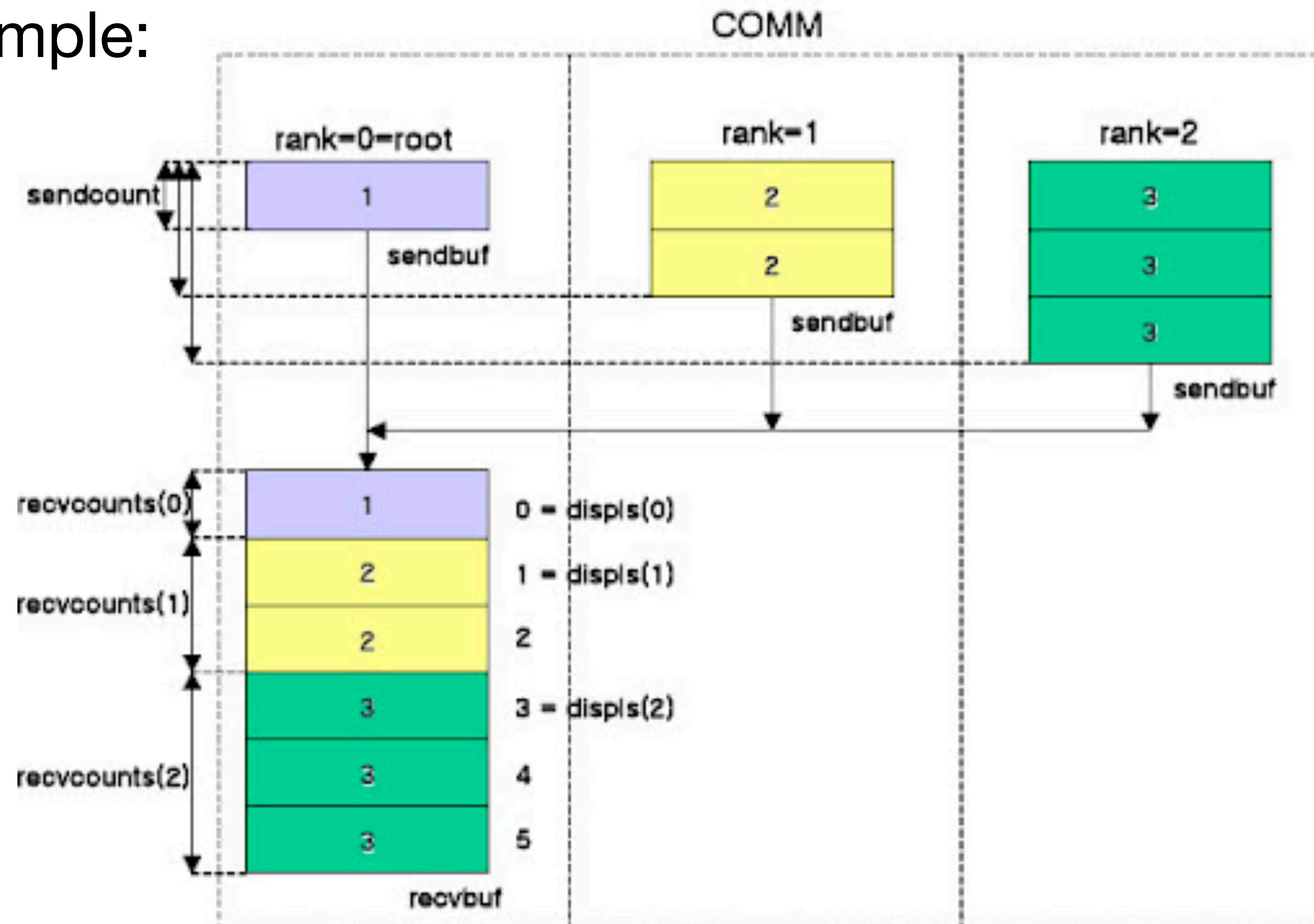
It is often used in parallel algorithms such as sorting and searching.

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Variable Length Collectives

Variable length: e.g. `gatherv, scatterv`, etc.

`gatherv` example:



http://k-atoms.ksc.re.kr/mpi/2_3_2.html

Example: Averaging Numbers with Gather/Scatter

Root process creates array of random numbers

```
if (world_rank == 0) {
  rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
…
```

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Example: Averaging Numbers with Gather/Scatter

Every process creates a buffer to store their subset of random numbers

```
if (world_rank == 0) {
  rand_nums = create_rand_num
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
…
```

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Example: Averaging Numbers with Gather/Scatter

```
if (world_rank == 0) {
  rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subs
float *sub_rand_nums = malloc(sizeof(flo

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
…
```

After the scatter, each process now contains elements_per_proc elements of the original data

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Example: Averaging Numbers with Gather/Scatter

```
if (world_rank == 0) {
  rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
…
```

Each process computes the average of their subset

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Example: Averaging Numbers with Gather/Scatter

```
if (world_rank == 0) {
  rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
…
```

Root creates buffer to hold all subset averages

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Example: Averaging Numbers with Gather/Scatter

```
…
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
           MPI_COMM_WORLD);

// Compute the total average of all numbers.
if (world_rank == 0) {
  float avg = compute_avg(sub_avgs, world_size);
  printf(“Avg computed across original data is %f\n”, avg);
}
```

Root gathers subset averages

```
Avg computed across original data is 0.479736
```

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Example: Averaging Numbers with Gather/Scatter

```
…
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
           MPI_COMM_WORLD);

// Compute the total average of all numbers.
if (world_rank == 0) {
  float avg = compute_avg(sub_avgs, world_size);
  printf("Avg computed across original data is %f\n", avg);
}
```

Root gathers subset averages

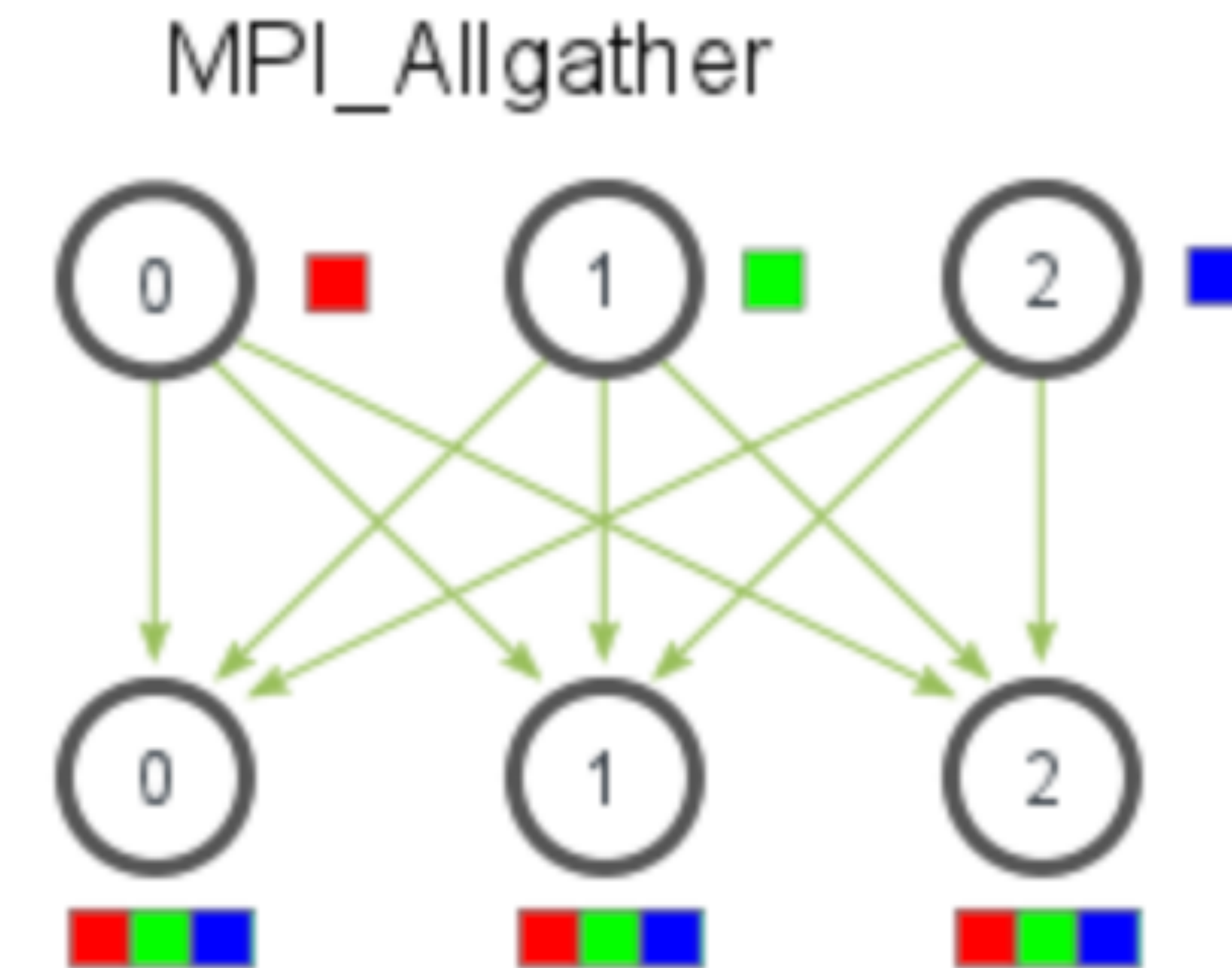
Root computes final average

```
Avg computed across original data is 0.479736
```

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

MPI_Allgather

```
MPI_Allgather(
  void* send_data,
  int send_count,
  MPI_Datatype send_datatype,
  void* recv_data,
  int recv_count,
  MPI_Datatype recv_datatype,
  MPI_Comm communicator)
```



- `MPI_Allgather` implements **many-to-many communication** - similar to `MPI_Gather` then `MPI_Bcast`.

- The function declaration is the same as `MPI_Gather` except that there is no root.

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

MPI_Allgather Example

All processes get all subset averages and compute the total average.

```
// Gather all partial averages down to all the processes
float *sub_avgs = (float *)malloc(sizeof(float) * world_size);
MPI_Allgather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT,
              MPI_COMM_WORLD);

// Compute the total average of all numbers.
float avg = compute_avg(sub_avgs, world_size);

printf("Avg of all elements from proc %d is %f\n", world_rank, avg);
```

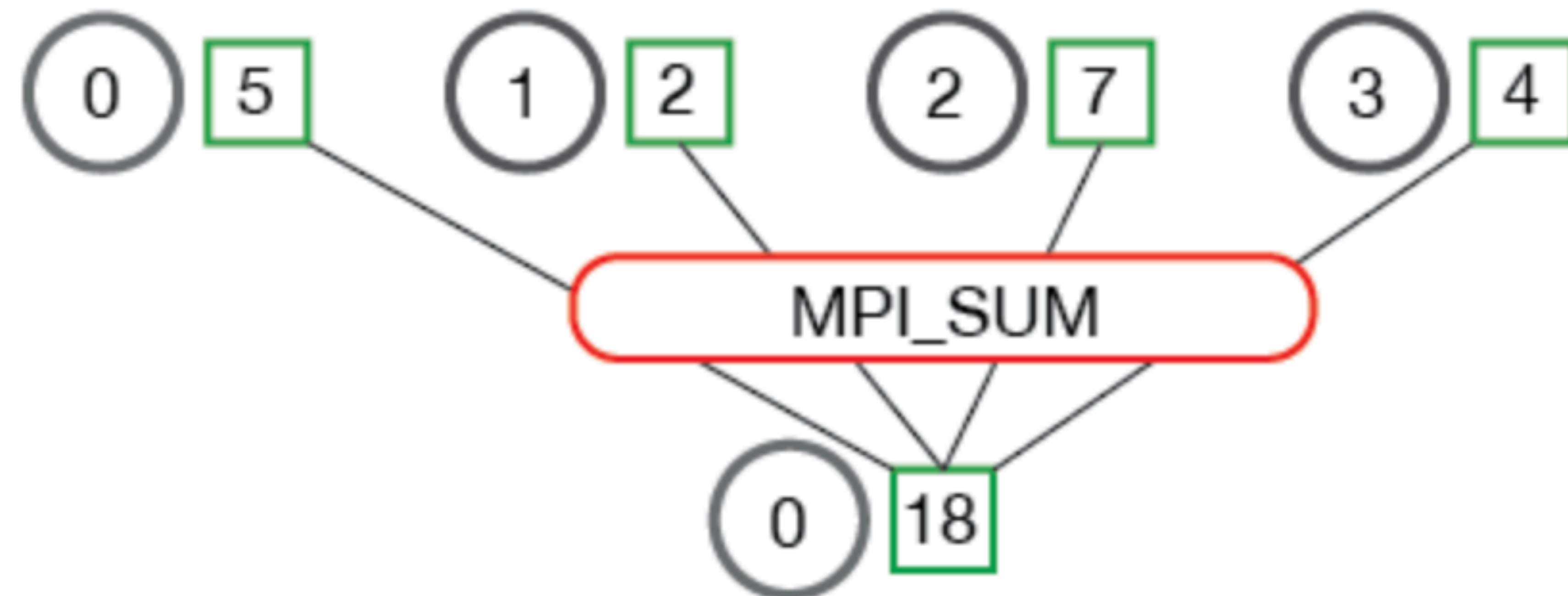
```
Avg of all elements from proc 1 is 0.479736
Avg of all elements from proc 3 is 0.479736
Avg of all elements from proc 0 is 0.479736
Avg of all elements from proc 2 is 0.479736
```

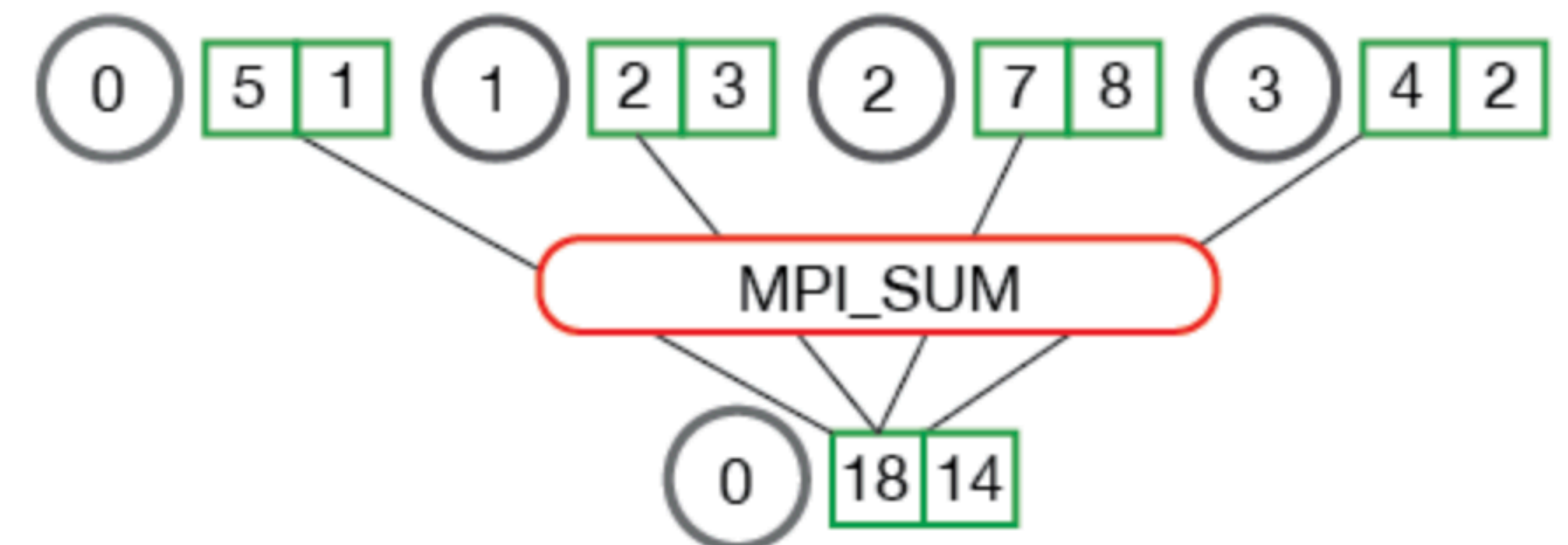
https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

MPI_Reduce

MPI_Reduce reduces multiple numbers per process into an array element for each index.

can be MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, etc.

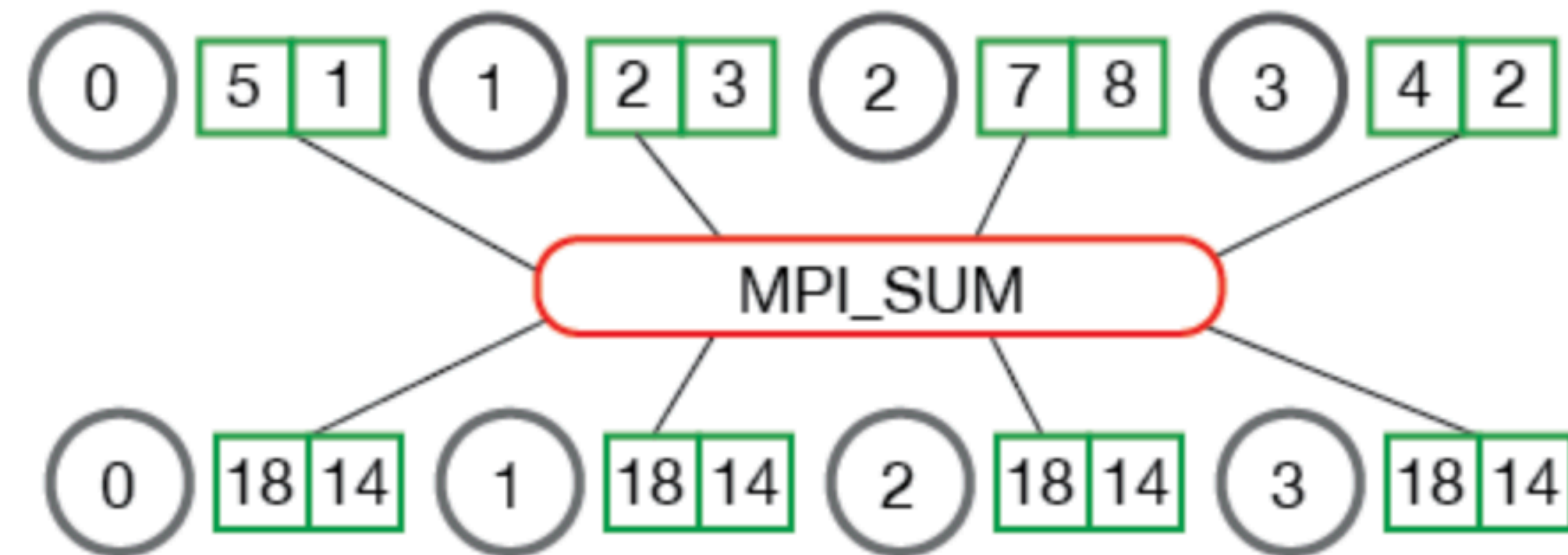




https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

MPI_Allreduce

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

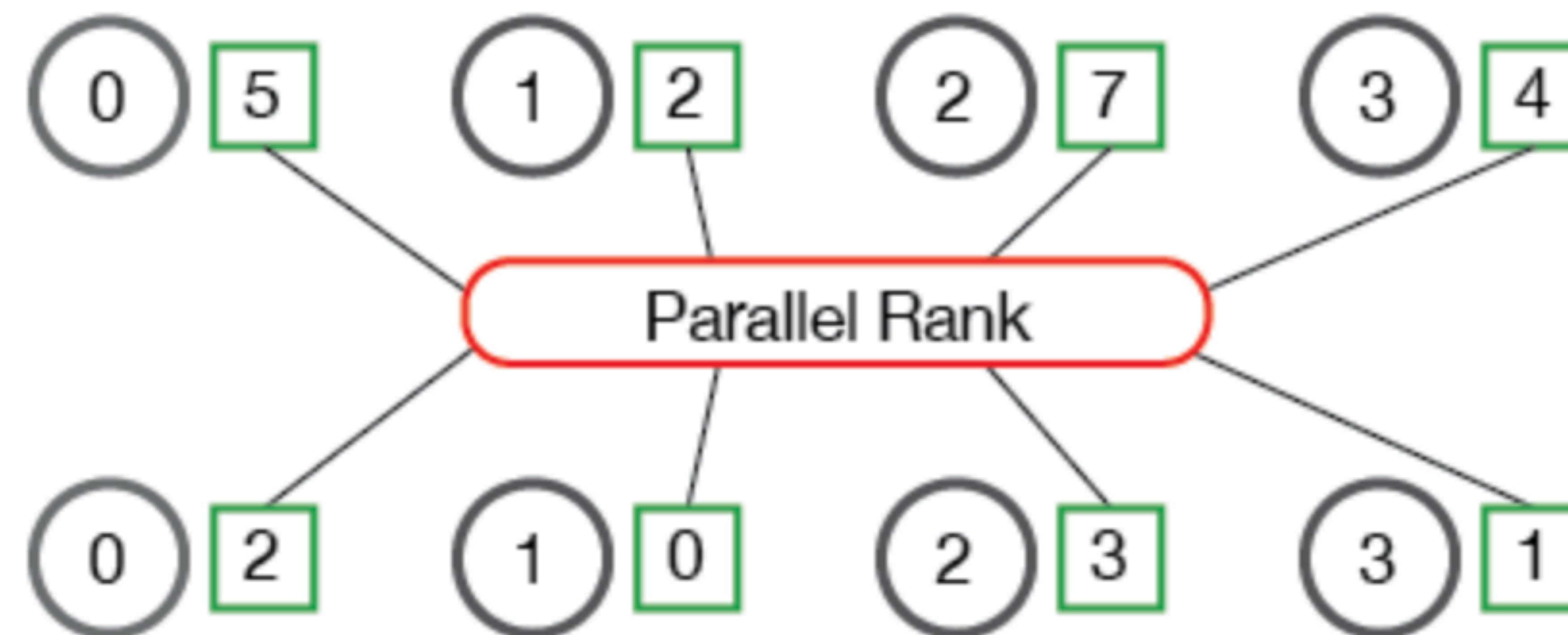


- MPI_Allreduce reduces the values and distributes the results to all processes.
- It is identical to MPI_Reduce except that it does not need a root.

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Example: Parallel Rank

- Input: All processes have **a single number stored in their local memory**
- Output: The **order of the processes' numbers** with respect to the entire set of numbers over all processes.
- **Example application:** benchmarking processors in an MPI cluster and trying to find the order of how fast each processor is relative to the others.



https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Parallel Rank API

```
TMPI_Rank(
    void *send_data,
    void *recv_data,
    MPI_Datatype datatype,
    MPI_Comm comm)
```

One number of type `datatype`

One integer per process that contains rank data for `send_data`

- Input: One number per processor
- Output: Return the associated rank with respect to all numbers across all processors

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Step 1: Gather numbers to root

```
void *gather_numbers_to_root(void *number, MPI_Datatype datatype,
                             MPI_Comm comm) {
  int comm_rank, comm_size;
  MPI_Comm_rank(comm, &comm_rank);
  MPI_Comm_size(comm, &comm_size);

  int datatype_size;
  MPI_Type_size(datatype, &datatype_size);
  void *gathered_numbers;
  if (comm_rank == 0) {
    gathered_numbers = malloc(datatype_size * comm_size);
  }

  // Gather all of the numbers on the root process
  MPI_Gather(number, 1, datatype, gathered_numbers, 1,
             datatype, 0, comm);

  return gathered_numbers;
}
```

Allocate array on root for all the numbers

Gather all numbers into root

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Step 2: Sorting Numbers and Maintaining Ownership

- Sorting is easily available (e.g., in the C++ STL), but we also have to maintain the ranks that sent the numbers to the root process.
- To attach the owning process to the numbers, we create a struct to hold the information:

```
typedef struct {
  int comm_rank;
  union {
    float f;
    int i;
  } number;
} CommRankNumber;
```

Owner process

Number to sort

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Step 2: Get Ranks

```
int *get_ranks(void *gathered_numbers, int gathered_number_count,
               MPI_Datatype datatype) {
  int datatype_size;
  MPI_Type_size(datatype, &datatype_size);

  // Convert the gathered number array to an array of CommRankNumbers.
  // This allows us to sort the numbers and also keep the information
  // of the processes that own the numbers intact.
  CommRankNumber *comm_rank_numbers = malloc(
    gathered_number_count * sizeof(CommRankNumber));
  int i;
  for (i = 0; i < gathered_number_count; i++) {
    comm_rank_numbers[i].comm_rank = i;
    memcpy(&(comm_rank_numbers[i].number),
           gathered_numbers + (i * datatype_size),
           datatype_size);
  }
…
```

Create array of CommRankNumber structs

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Step 2: Get Ranks

```c
int *get_ranks(void *gathered_numbers, int gathered_number_count,
               MPI_Datatype datatype) {
  int datatype_size;
  MPI_Type_size(datatype, &datatype_size);

  // Convert the gathered number array to an array of CommRankNumbers.
  // This allows us to sort the numbers and also keep the information
  // of the processes that own the numbers intact.
  CommRankNumber *comm_rank_numbers = malloc(
    gathered_number_count * sizeof(CommRankNumber));
  int i;
  for (i = 0; i < gathered_number_count; i++) {
    comm_rank_numbers[i].comm_rank = i;
    memcpy(&(comm_rank_numbers[i].number),
           gathered_numbers + (i * datatype_size),
           datatype_size);
  }
…
```

Create array of CommRankNumber structs

Attach rank of owner to numbers

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Step 2: Get Ranks

```
int *get_ranks(void *gathered_numbers, int gathered_number_count,
               MPI_Datatype datatype) {
  int datatype_size;
  MPI_Type_size(datatype, &datatype_size);

  // Convert the gathered number array to an array of CommRankNumbers.
  // This allows us to sort the numbers and also keep the information
  // of the processes that own the numbers intact.
  CommRankNumber *comm_rank_numbers = malloc(
    gathered_number_count * sizeof(CommRankNumber));
  int i;
  for (i = 0; i < gathered_number_count; i++) {
    comm_rank_numbers[i].comm_rank = i;
    memcpy(&(comm_rank_numbers[i].number),
           gathered_numbers + (i * datatype_size),
           datatype_size);
  }
…
```

Create array of CommRankNumber structs

Attach rank of owner to numbers

Copy in the number from the owner

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Step 2: Get Ranks

Sort based on datatype

```
  if (datatype == MPI_FLOAT) {
    qsort(comm_rank_numbers, gathered_number_count,
          sizeof(CommRankNumber), &compare_float_comm_rank_number);
  } else {
    qsort(comm_rank_numbers, gathered_number_count,
          sizeof(CommRankNumber), &compare_int_comm_rank_number);
  }

  int *ranks = (int *)malloc(sizeof(int) * gathered_number_count);
  for (i = 0; i < gathered_number_count; i++) {
    ranks[comm_rank_numbers[i].comm_rank] = i;
  }

  free(comm_rank_numbers);
  return ranks;
}
```

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Step 2: Get Ranks

Sort based on datatype

```
  if (datatype == MPI_FLOAT) {
    qsort(comm_rank_numbers, gathered_number_count,
          sizeof(CommRankNumber), &compare_float_comm_rank_number);
  } else {
    qsort(comm_rank_numbers, gathered_number_count,
          sizeof(CommRankNumber), &compare_int_comm_rank_n
  }

  int *ranks = (int *)malloc(sizeof(int) * gathered_number_count);
  for (i = 0; i < gathered_number_count; i++) {
    ranks[comm_rank_numbers[i].comm_rank] = i;
  }

  free(comm_rank_numbers);
  return ranks;
}
```

Make array of rank values for each process

Step 2: Get Ranks

```
  if (datatype == MPI_FLOAT) {
    qsort(comm_rank_numbers, gathered_number_count,
          sizeof(CommRankNumber), &compare_float_comm_rank_number);
  } else {
    qsort(comm_rank_numbers, gathered_number_count,
          sizeof(CommRankNumber), &compare_int_comm_rank_n
  }

  int *ranks = (int *)malloc(sizeof(int) * gathered_number_count);
  for (i = 0; i < gathered_number_count; i++) {
    ranks[comm_rank_numbers[i].comm_rank] = i;
  }

  free(comm_rank_numbers);
  return ranks;
}
```

Sort based on datatype

Make array of rank values for each process

The i-th element in the array contains the rank value for the number sent by process i

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Step 2: Get Ranks

Sort based on datatype

```
  if (datatype == MPI_FLOAT) {
    qsort(comm_rank_numbers, gathered_number_count,
          sizeof(CommRankNumber), &compare_float_comm_rank_number);
  } else {
    qsort(comm_rank_numbers, gathered_number_count,
          sizeof(CommRankNumber), &compare_int_comm_rank_n
  }

  int *ranks = (int *)malloc(sizeof(int) * gathered_number_count);
  for (i = 0; i < gathered_number_count; i++) {
    ranks[comm_rank_numbers[i].comm_rank] = i;
  }

  free(comm_rank_numbers);
  return ranks;
}
```

Make array of rank values for each process

The i-th element in the array contains the rank value for the number sent by process i

Cleanup

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Putting it all together

```
// Gets the rank of the send_data, which is of type datatype. The rank
// is returned in recv_data and is of type datatype.
int TMPI_Rank(void *send_data, void *recv_data, MPI_Datatype datatype,
             MPI_Comm comm) {
  // NOT SHOWN - INIT

  void *gathered_numbers = gather_numbers_to_root(send_data, datatype,
                                                  comm);

  // Get the ranks of each process
  int *ranks = NULL;
  if (comm_rank == 0) {
    ranks = get_ranks(gathered_numbers, comm_size, datatype);
  }

  // Scatter the rank results
  MPI_Scatter(ranks, 1, MPI_INT, recv_data, 1, MPI_INT, 0, comm);

  // NOT SHOWN - CLEANUP
}
```

Gather numbers to root process

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Putting it all together

```
// Gets the rank of the send_data, which is of type datatype. The rank
// is returned in recv_data and is of type datatype.
int TMPI_Rank(void *send_data, void *recv_data, MPI_Datatype datatype,
              MPI_Comm comm) {
  // NOT SHOWN - INIT

  void *gathered_numbers = gather_numbers_to_root(send_data, datatype,
                                                  comm);

  // Get the ranks of each process
  int *ranks = NULL;
  if (comm_rank == 0) {
    ranks = get_ranks(gathered_numbers, comm_size, datatype);
  }

  // Scatter the rank results
  MPI_Scatter(ranks, 1, MPI_INT, recv_data, 1, MPI_INT, 0, comm);

  // NOT SHOWN - CLEANUP
}
```

Gather numbers to root process

Sort and get ranks of gathered numbers

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Putting it all together

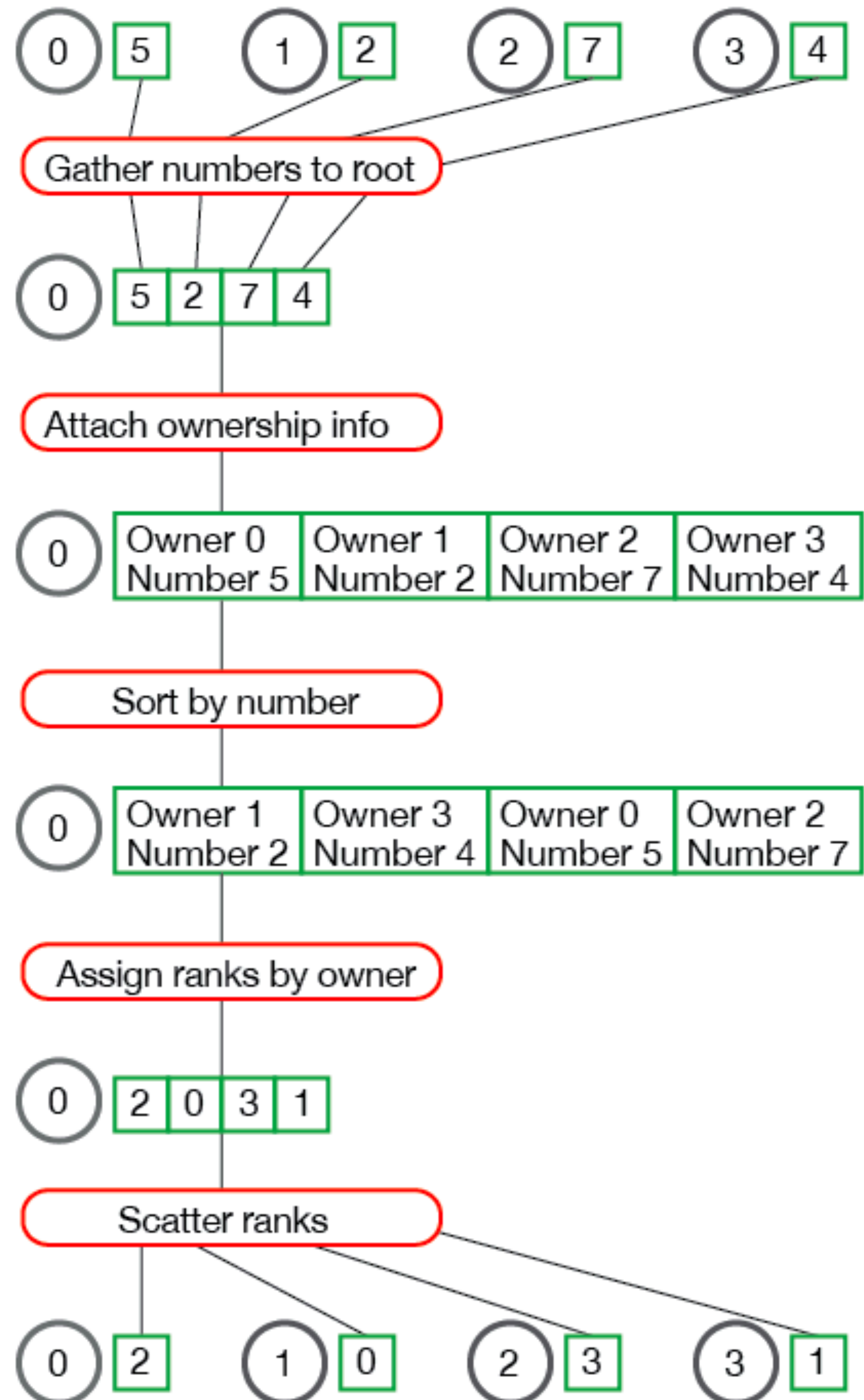
```c
// Gets the rank of the send_data, which is of type datatype. The rank
// is returned in recv_data and is of type datatype.
int TMPI_Rank(void *send_data, void *recv_data, MPI_Datatype datatype,
              MPI_Comm comm) {
  // NOT SHOWN - INIT

  void *gathered_numbers = gather_numbers_to_root(send_data, datatype,
                                                   comm);

  // Get the ranks of each process
  int *ranks = NULL;
  if (comm_rank == 0) {
    ranks = get_ranks(gathered_numbers, comm_size, datatype);
  }

  // Scatter the rank
  MPI_Scatter(ranks, 1, MPI_INT, recv_data, 1, MPI_INT, 0, comm);

  // NOT SHOWN - CLEANUP
}
```

Gather numbers to root process

Sort and get ranks of gathered numbers

Scatter rank results back to other processes

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/



https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

Assuming 4 processes

Example Output

```
Rank for 0.242578 on process 0 - 0
Rank for 0.894732 on process 1 - 3
Rank for 0.789463 on process 2 - 2
Rank for 0.684195 on process 3 - 1
```

https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

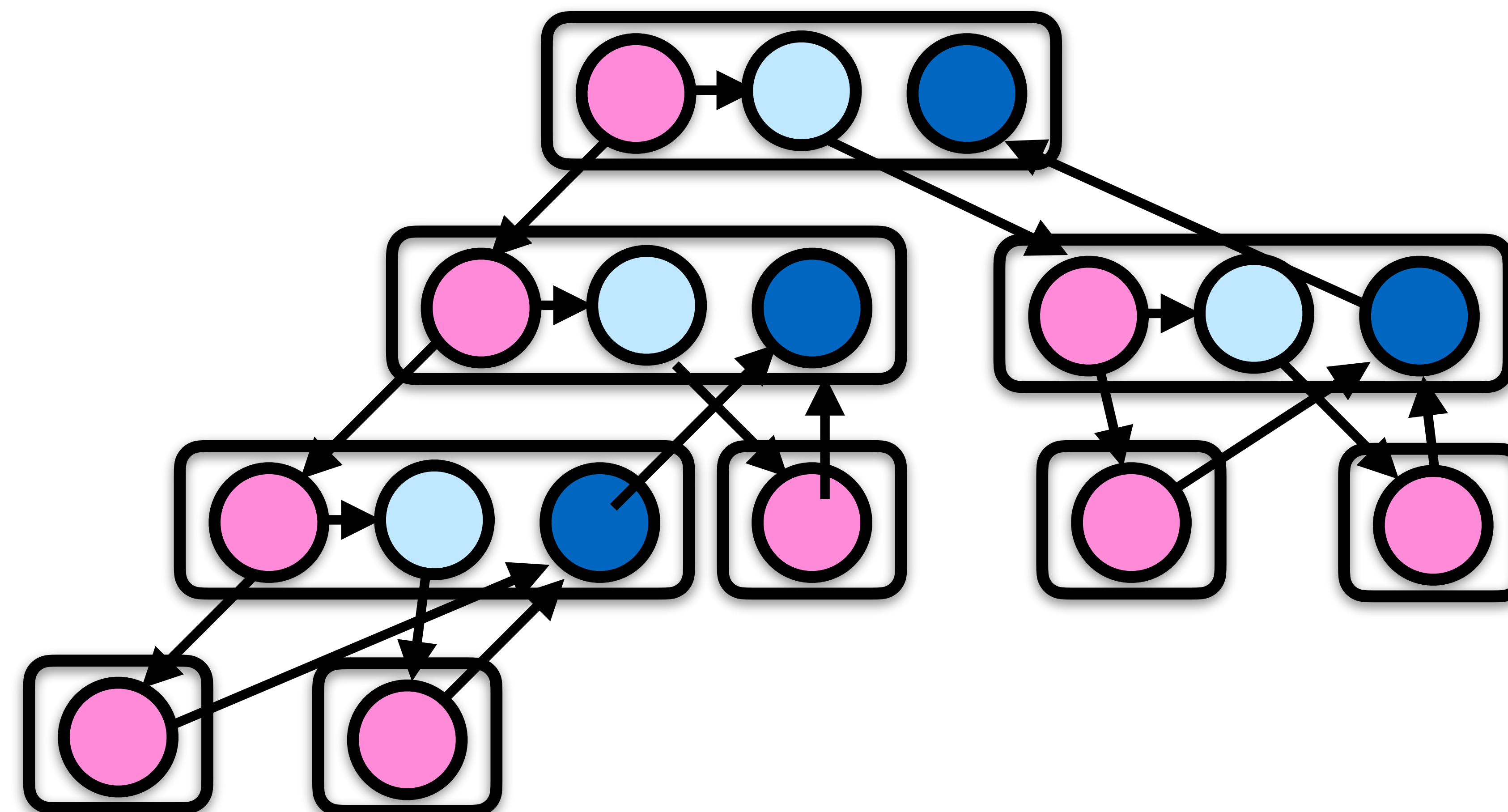
Performance Models

Recall: PRAM Model for Shared Memory

Parallel Random Access Memory (PRAM)

- All memory access operations complete in one clock period -- no concept of memory hierarchy ("too good to be true").
- OK for understanding whether an algorithm has enough parallelism at all.

Parallel algorithm design strategy: first do a PRAM algorithm, then worry about memory/communication time (sometimes works)



Motivation for Other Distributed-Memory Models

Processors are multi-core and many nodes are multi-chip.

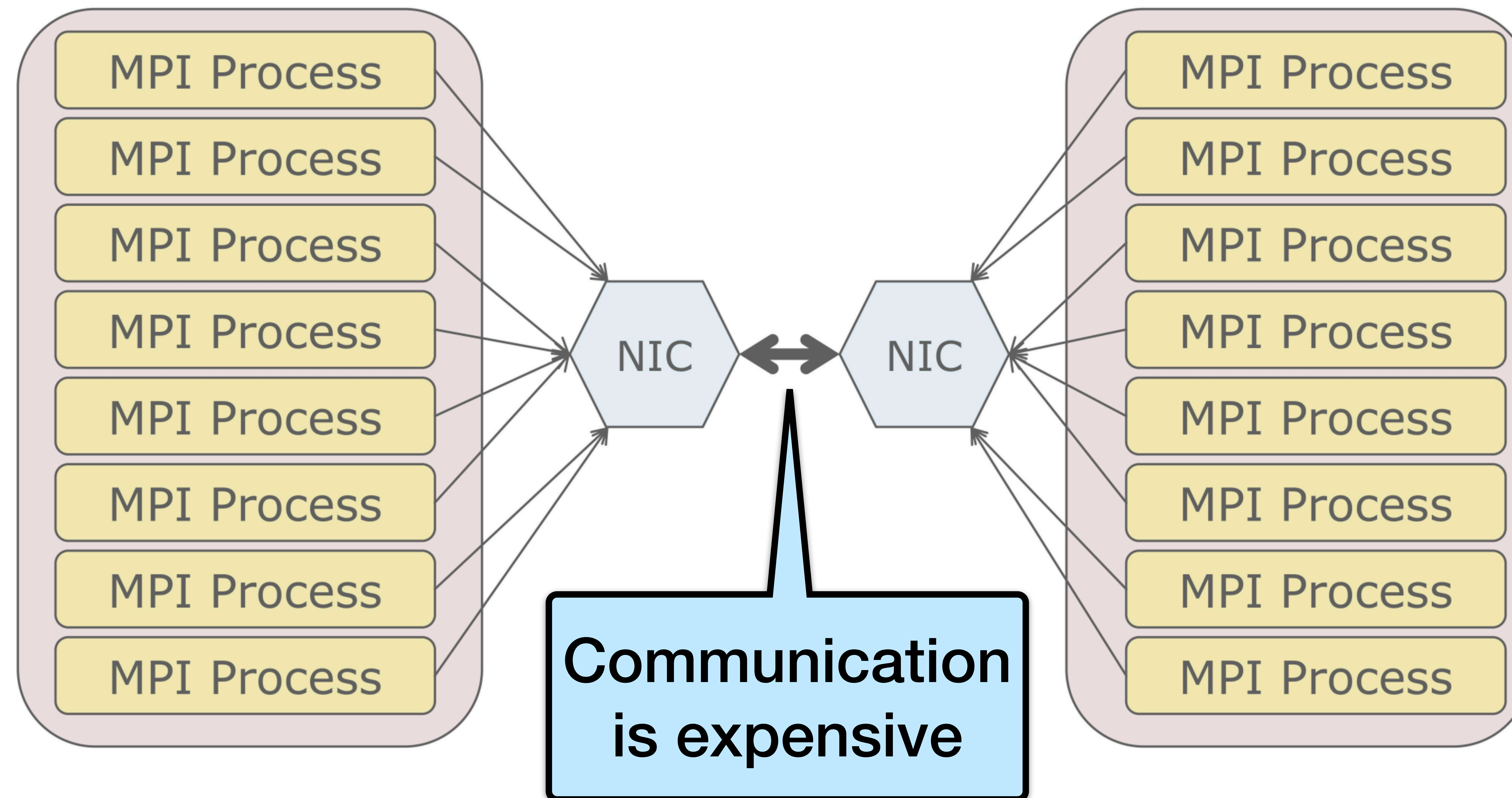


Figure source: Balaji, Gropp, Thakur

Latency and Bandwidth Model for Distributed-Memory Machines

Ignores topology

Time to send message of length n is roughly:

Time = latency + n * cost_per_word
= latency + n/bandwidth

Often called "a-b model" and written:

$$\text{Time} = \alpha + n\beta$$

Usually $\alpha \gg \beta \gg$ time per flop.

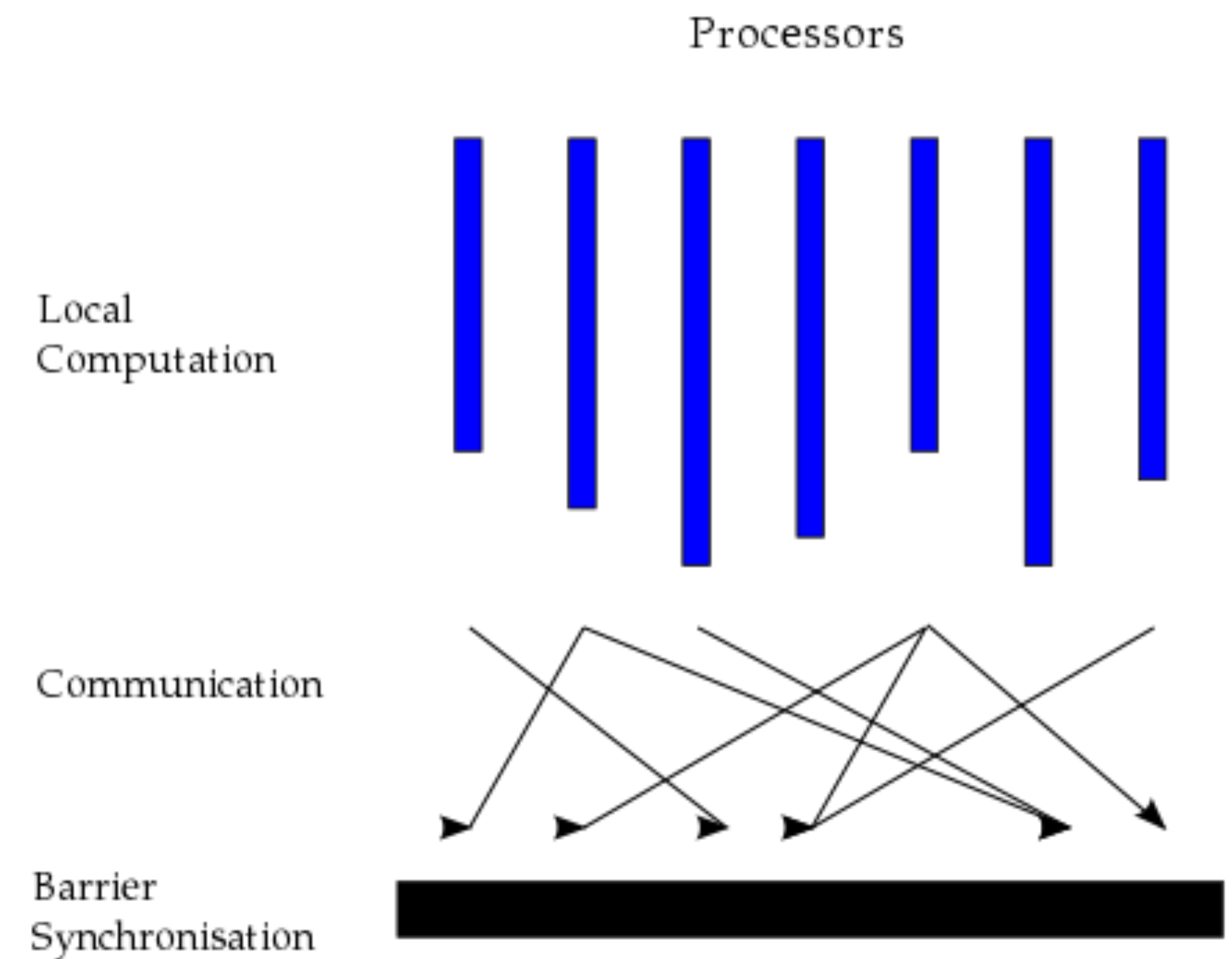
- One long message is cheaper than many short ones.
- Can do hundreds or thousands of flops for cost of one message.

Lesson: Need **large computation-to-communication ratio** to be efficient.

Bulk-Synchronous Parallel (BSP) Computer

A BSP computer consists of the following:

- Components capable of **processing and/or local memory transactions** (i.e., processors),
- A **network** that routes messages between pairs of such components, and
- A hardware facility that allows for the **synchronization** of all or a subset of components.



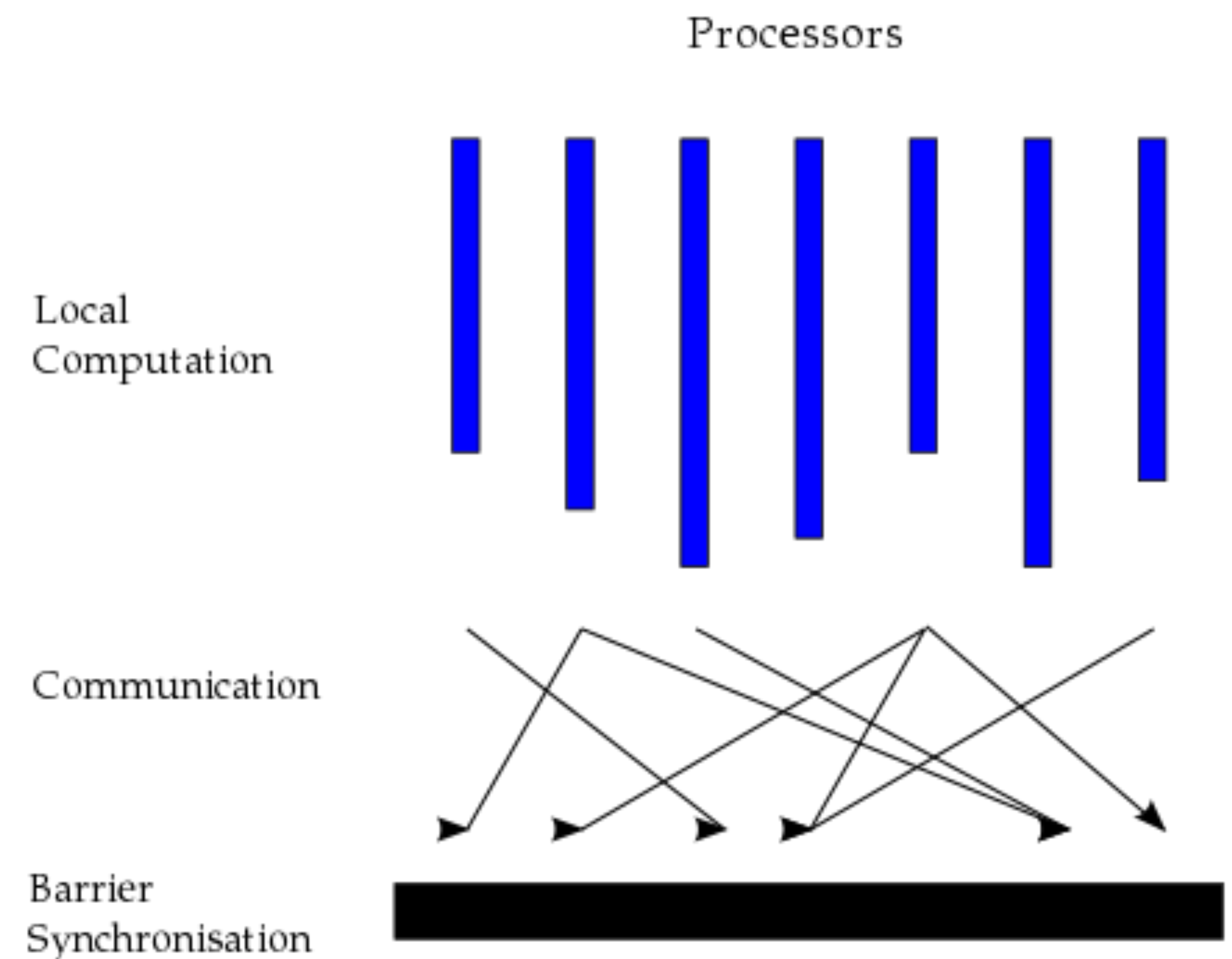
https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Leslie G. Valiant, “A bridging model for parallel computation,” Communications of the ACM, Volume 33 Issue 8, Aug. 1990

Bulk-Synchronous Parallel (BSP) Algorithms

A BSP computation proceeds in a series of global ***supersteps***, which consists of three components:

- **Concurrent computation**: every participating processor may perform local computations which may **overlap with communication.**
- **Communication**: processes exchange data.
- **Barrier synchronization**: When a process reaches this point (the *barrier*), it waits until all other processes have reached the same barrier.

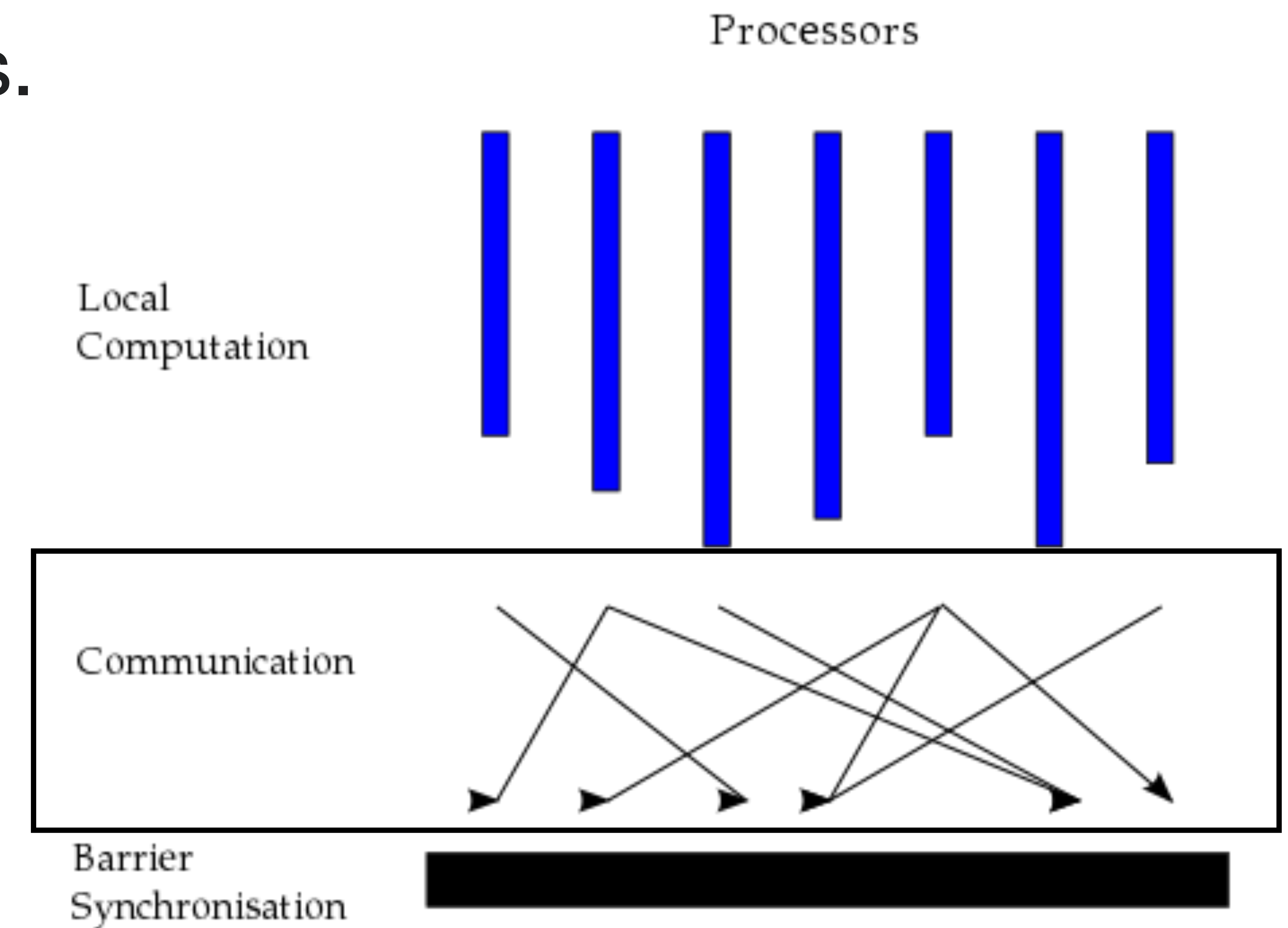


https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Leslie G. Valiant, “A bridging model for parallel computation,” Communications of the ACM, Volume 33 Issue 8, Aug. 1990

Communication in BSP

- It is difficult to determine the time any single communication action will complete since there are may be many simultaneous communications.
- The BSP model provides an **upper bound** on the time to communicate a set of data:
  - Let h be the number of incoming or outgoing message in a superstep.
  - The ability of a communication network to deliver data is captured by some parameter g, such that it takes **time hg for a process to deliver h messages of size 1**.



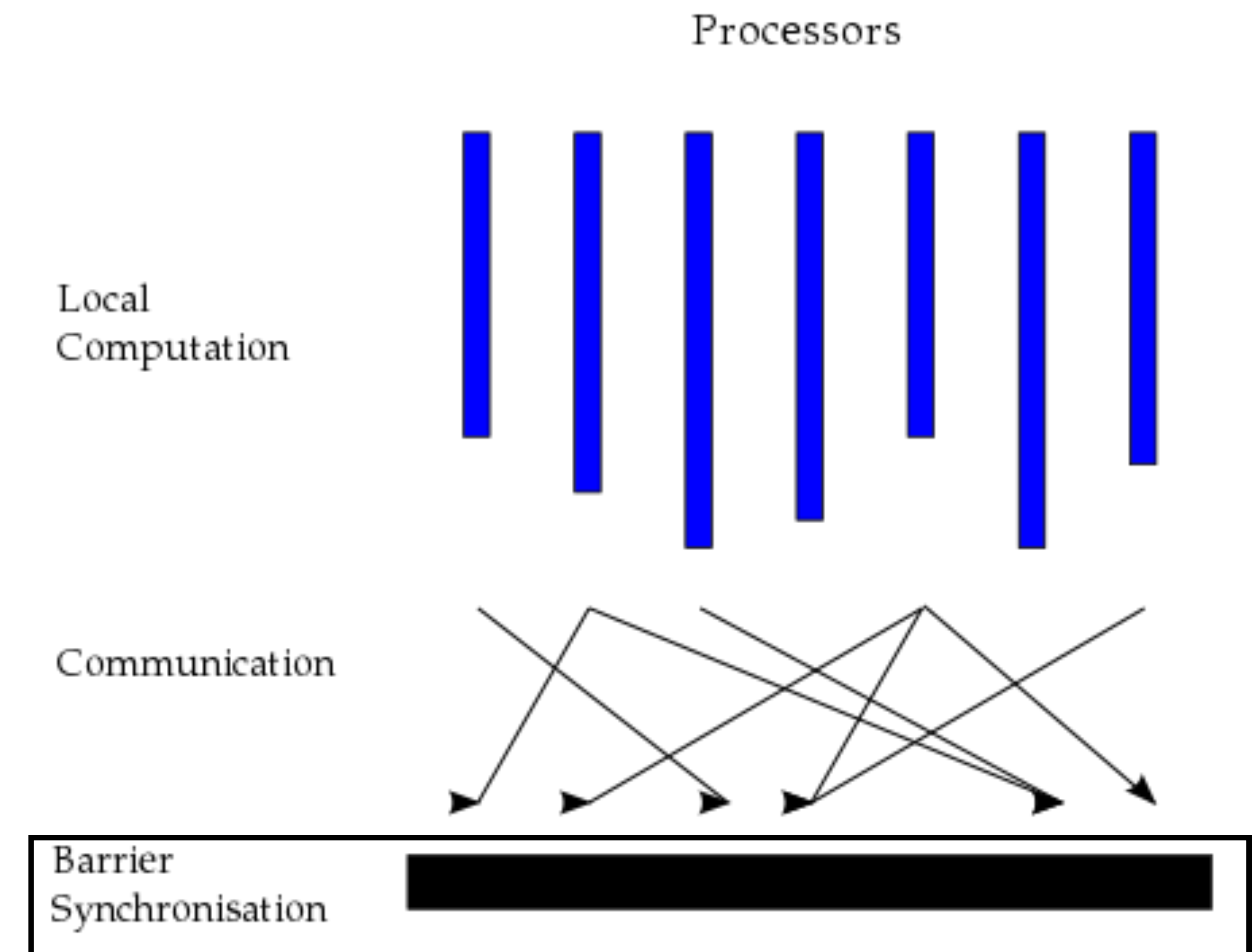
Ignores latency

https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Leslie G. Valiant, "A bridging model for parallel computation," Communications of the ACM, Volume 33 Issue 8, Aug. 1990

Barriers in BSP

- The cost of **barrier synchronization** is denoted by ℓ.
- The cost of barriers is influenced by a couple of issues:
  - The cost imposed by the **variation in the completion time** of the different processes.
  - The **cost of reaching a globally consistent state** in all of the processors.



https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Leslie G. Valiant, "A bridging model for parallel computation," Communications of the ACM, Volume 33 Issue 8, Aug. 1990

Analyzing Algorithms in the BSP Model

The cost of a superstep is the sum of three terms:

- The cost of the longest-running **computation**,
- The cost of global **communication** between the processors, and
- The cost of the barrier **synchronization** at the end of the superstep.

Cost for local computation in process i

Messages sent or received by process i

$$\max_{i=1}^{p}(w_i) + \max_{i=1}^{p}(h_i g) + \ell$$

Synchronization cost

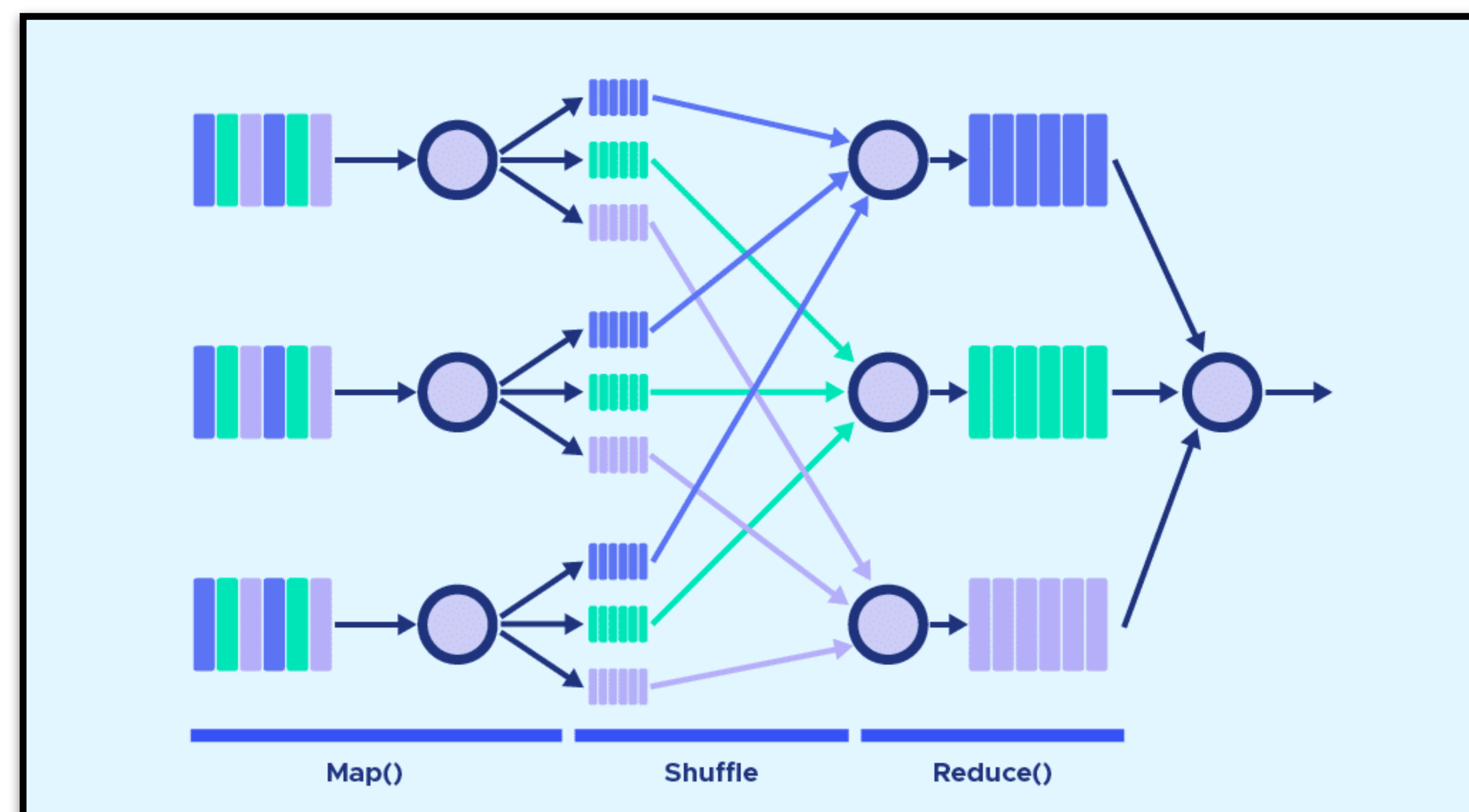
The cost of an entire BSP algorithm is the sum of the cost of **all the supersteps**.

https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

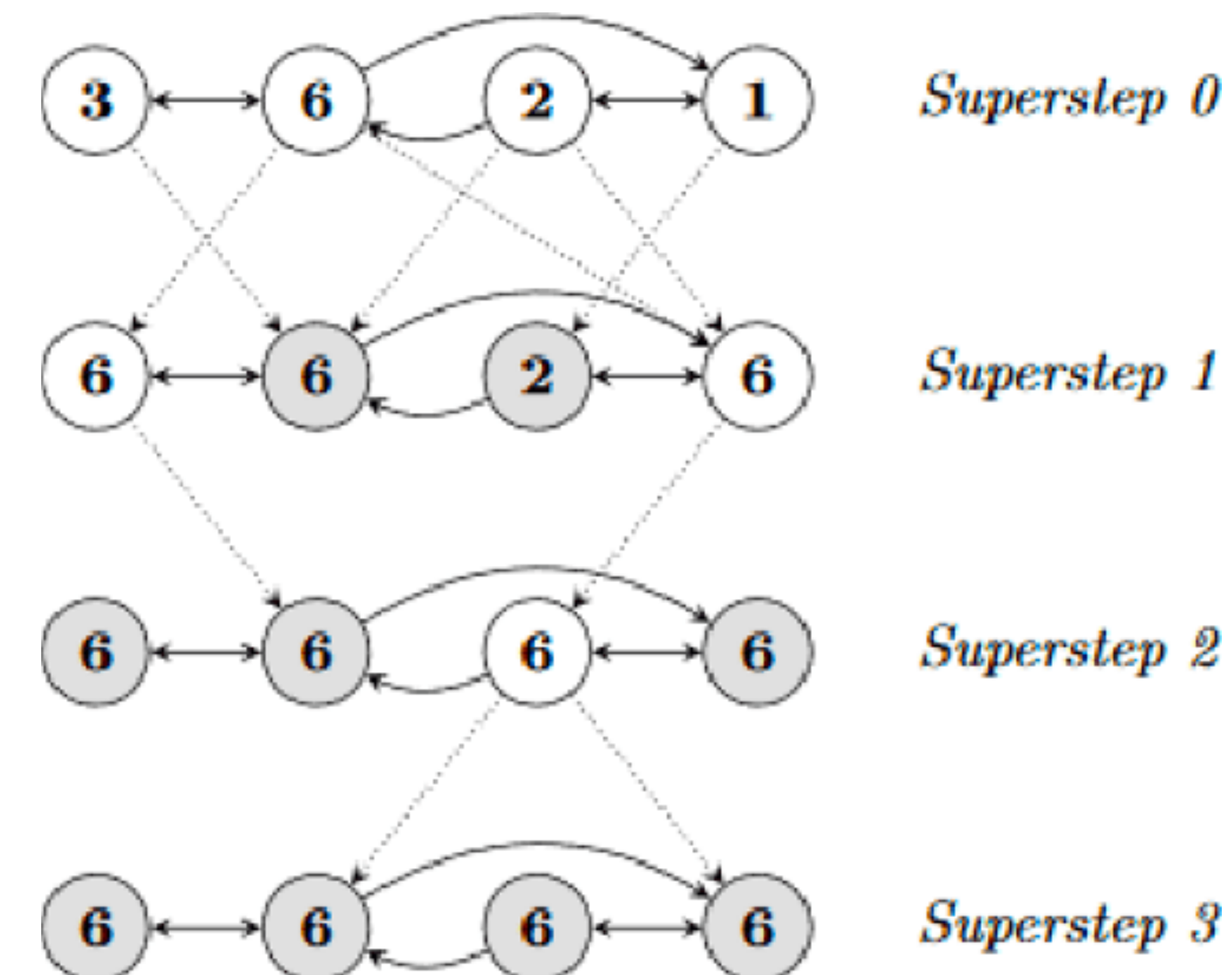
Leslie G. Valiant, "A bridging model for parallel computation," Communications of the ACM, Volume 33 Issue 8, Aug. 1990

Applications of BSP

One of the main applications for BSP is Google's graph analysis platforms Pregel and MapReduce.



https://datascientest.com/en/mapreduce-how-to-use-it-for-big-data



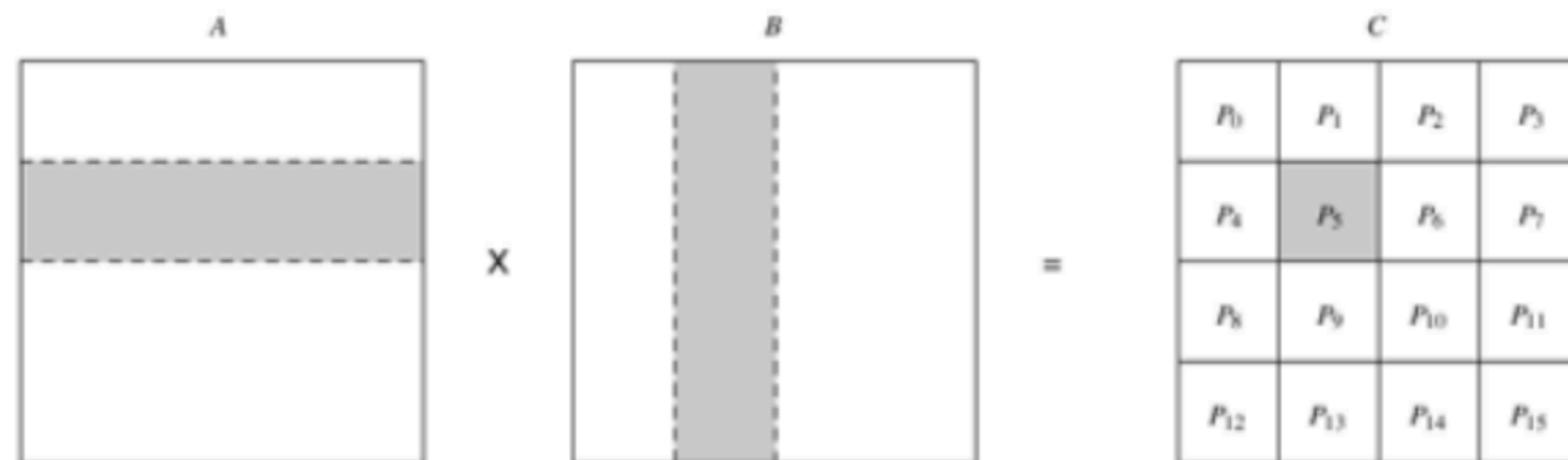
https://www.semanticscholar.org/paper/Pregel%3A-a-system-for-large-scale-graph-processing-Malewicz-Austern/2d867297dfe0d3ce2ed5b1d0f2dff88cac46ee94

https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Matrix Multiplication

A Simple Distributed Matrix-Matrix Multiplication

- Let A, B be n x n matrices. Goal: Compute $C = AB$.
- Partition A and B into p square blocks $A_{i,j}, B_{i,j}$ $(0 \leq i, j < \sqrt{p})$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each.
- Process $P_{i,j}$ initially stores $A_{i,j}, B_{i,j}$ and computes submatrix $C_{i,j}$.
- Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}, B_{k,j}$ for $0 \leq k < \sqrt{p}$.



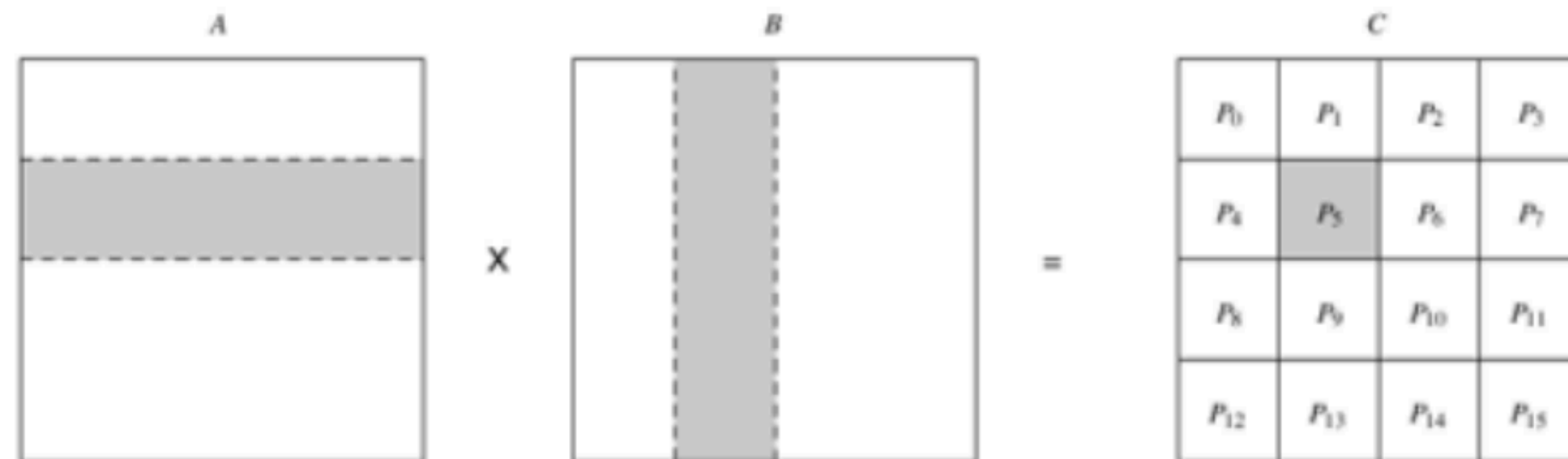
https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-07-3.pdf

A Simple Distributed Matrix-Matrix Multiplication

Algorithm:

Perform all-to-all broadcast of blocks of A in each row of processes.
Perform all-to-all broadcast of blocks of B in each column of processes.

Each process $P_{i,j}$ performs $C_{i,j} = \sum_{k=0}^{\sqrt{p}=1} A_{i,k} \times B_{k,j}$.



https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-07-3.pdf

Communication Analysis of Simple Algorithm in BSP Model

Step 1: $\sqrt{p}$ rows of all-to-all broadcasts, each of size $\frac{n^2}{p}$.

Communication time = $g \times \frac{n^2}{p}(\sqrt{p} - 1)$

Step 2: $\sqrt{p}$ columns of all-to-all broadcasts, each of size $\frac{n^2}{p}$.

Communication time = $g \times \frac{n^2}{p}(\sqrt{p} - 1)$

Total communication time = $2g \times \frac{n^2}{p}(\sqrt{p} - 1)$

https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-07-3.pdf

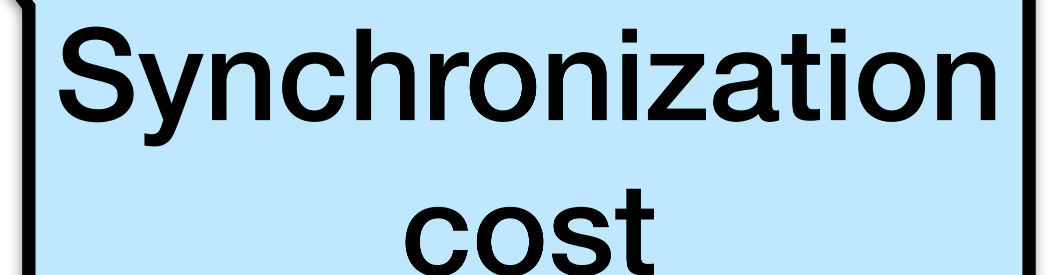
Full Analysis of Simple Algorithm in BSP Model

Total communication time = $2g \times \frac{n^2}{p}(\sqrt{p} - 1)$

Computation time: $\sqrt{p}$ multiplications of submatrices at $(n/\sqrt{p})^3$ work each

Total computation time per process = $\sqrt{p} \times (n/\sqrt{p})^3 = n^3/p$

Total time = communication time + computation time + ℓ



https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-07-3.pdf

Memory Usage of Simple Algorithm

Each process $P_{i,j}$ has $2\sqrt{p}$ blocks of $A_{i,k}, B_{k,j}$.

Each process therefore needs $2\sqrt{p} \times (n^2/p) = \Theta(n^2/\sqrt{p})$ memory.

There are p processes, so the total memory is $\Theta(n^2 \times \sqrt{p})$.

Extra $\sqrt{p}$ factor - can we do better?

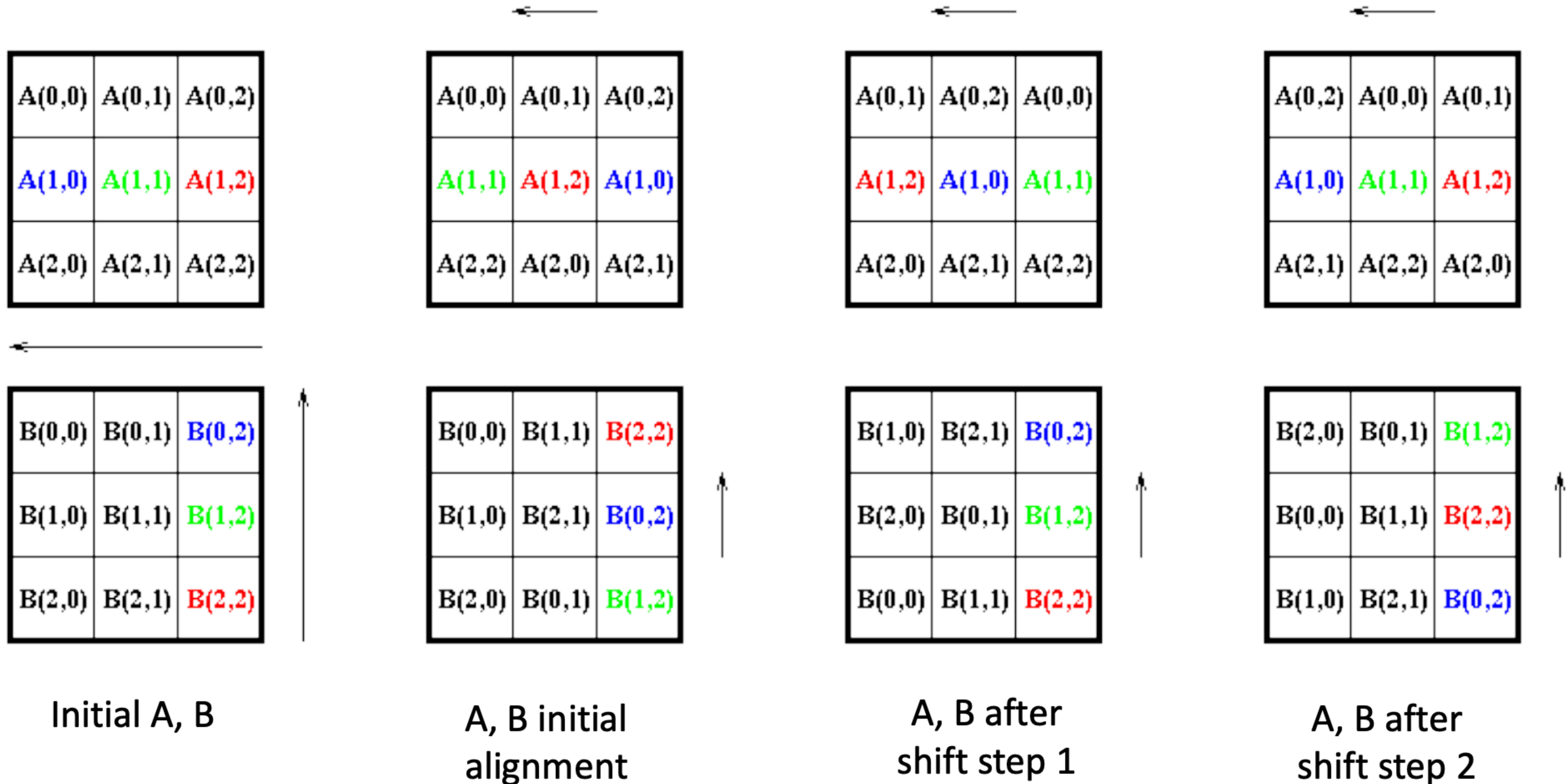
Cannon's Algorithm

- Goal: Improve the memory efficiency of the simple algorithm.
- Idea: Add more rounds and compute the addition of one tile addition per round.
- Based on the following contention-free formula:

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k) \bmod \sqrt{p}} \times B_{(i+j+k) \bmod \sqrt{p},j}$$

Lynn Elliot Cannon, *A cellular computer to implement the Kalman Filter Algorithm*, Technical report, Ph.D. Thesis, Montana State University, 14 July 1969.

Cannon's Algorithm for a 3x3 Grid



https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-07-3.pdf

Pseudocode for Cannon's Algorithm

// make initial alignment

for i, j :=0 **to** $\sqrt{p} - 1$ **do**

Send block $A_{i,j}$ to process $\left(i, \left(j - i + \sqrt{p}\right) mod \sqrt{p}\right)$ and block $B_{i,j}$ to process $\left(\left(i - j + \sqrt{p}\right) mod \sqrt{p}, j\right)$;

endfor;

Process $P_{i,j}$ multiply received submatrices together and add the result to $C_{i,j}$;

// compute-and-shift. A sequence of one-step shifts pairs up $A_{i,k}$ **and** $B_{k,j}$

// on process $P_{i,j}$. $C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}$

for step :=1 **to** $\sqrt{p} - 1$ **do**

Shift $A_{i,j}$ one step left (with wraparound) and $B_{i,j}$ one step up (with wraparound);

Process $P_{i,j}$ multiply received submatrices together and add the result to $C_{i,j}$;

Endfor;

Remark: In the initial alignment, the send operation is to: shift $A_{i,j}$ to the left (with wraparound) by i steps, and shift $B_{i,j}$ to the up (with wraparound) by j steps. 7

https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-07-3.pdf

Communication Analysis of Cannon's Algorithm in BSP Model

Initial step: every processor does a circular shift in row and column directions. It has to move two submatrices of size n^2/p each.

Communication cost for initial step = $2g \times (n^2/p)$

Subsequent steps: every processor does a circular shift of two submatrices of size n^2/p each. There are $\sqrt{p}$ of these steps.

Communication cost for following steps = $2g \times (n^2/\sqrt{p})$

The total communication cost sums these two up.

https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-07-3.pdf

Full Analysis of Cannon's Algorithm in BSP Model

There are $\sqrt{p}+1$ supersteps (one for initial alignment, the others for the actual multiply).

The total computation time is the same as the simple algorithm (n^3/p).

Therefore, the total time in BSP for Cannon's algorithm is:
communication time + computation time + ($\ell(\sqrt{p}+1)$

https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-07-3.pdf

Summary

- MPI collectives are a key part of distributed-memory programming.
- BSP model is a classical theoretical model for distributed memory.
- There are many other matrix-matrix multiplication algorithms in distributed memory, e.g., SUMMA.
- Significant effort has been devoted to communication-avoiding algorithms in distributed memory - some theoretical models ignore local computation.

BACKUP SLIDES

Multiple Completions

It is sometimes desirable to **wait on multiple requests**:

```
MPI_Waitall(count, array_of_requests, array_of_statuses)
MPI_Waitany(count, array_of_requests, &index, &status)
MPI_Waitsome(count, array_of_requests, array_of_indices,
array_of_statuses)
```

There are corresponding versions of test for each of these.

MPI_Irecv Example

```
    switch(my_rank)
    {
        // NOT SHOWN - The primary MPI process sends the value 12345 synchronously.
        case RECEIVER:
        {
            // The secondary MPI process receives the message.
            int received;
            MPI_Request request;
            printf("[Process %d] I issue the MPI_Irecv to receive the message as a
background task.\n", my_rank);
            MPI_Irecv(&received, 1, MPI_INT, SENDER, 0, MPI_COMM_WORLD, &request);

            // Do other things while the MPI_Irecv completes.
            printf("[Process %d] The MPI_Irecv is issued, I now moved on to print this
message.\n", my_rank);

            // Wait for the MPI_Recv to complete.
            MPI_Wait(&request, MPI_STATUS_IGNORE);
            printf("[Process %d] The MPI_Irecv completed, therefore so does the underlying
MPI_Recv. I received the value %d.\n", my_rank, received);
            break;
        }
    }
```

https://rookiehpc.org/mpi/docs/mpi_irecv/index.html

MPI_Irecv Example

```
    switch(my_rank)
    {
        // NOT SHOWN - The primary MPI process sends the value 12345 synchronously.
        case RECEIVER:
        {
            // The secondary MPI process receives the message.
            int received;
            MPI_Request request;
            printf("[Process %d] I issue the MPI_Irecv to receive the message as a 
background task.\n", my_rank);
            MPI_Irecv(&received, 1, MPI_INT, SENDER, 0, MPI_COMM_WORLD, &request);

            // Do other things while the MPI_Irecv completes.
            printf("[Process %d] The MPI_Irecv is issued, I now moved on to print this 
message.\n", my_rank);

            // Wait for the MPI_Recv to complete.
            MPI_Wait(&request, MPI_STATUS_IGNORE);
            printf("[Process %d] The MPI_Irecv completed, therefore so does the underlying 
MPI_Recv. I received the value %d.\n", my_rank, received);
            break;
        }
    }
```

Can do local work while waiting

https://rookiehpc.org/mpi/docs/mpi_irecv/index.html

MPI_Irecv Example

```
    switch(my_rank)
    {
        // NOT SHOWN - The primary MPI process sends the value 12345 synchronously.
        case RECEIVER:
        {
            // The secondary MPI process receives the message.
            int received;
            MPI_Request request;
            printf("[Process %d] I issue the MPI_Irecv to receive the message as a
background task.\n", my_rank);
            MPI_Irecv(&received, 1, MPI_INT, SENDER, 0, MPI_COMM_WORLD, &request);

            // Do other things while the MPI_Irecv completes.
            printf("[Process %d] The MPI_Irecv is issued, I now moved on to print this
message.\n", my_rank);

            // Wait for the MPI_Recv to complete.
            MPI_Wait(&request, MPI_STATUS_IGNORE);
            printf("[Process %d] The MPI_Irecv completed, therefore so does the underlying
MPI_Recv. I received the value %d.\n", my_rank, received);
            break;
        }
    }
```

Can do local work while waiting

value 12345

https://rookiehpc.org/mpi/docs/mpi_irecv/index.html

MPI_Bcast Example

```
// Get my rank in the communicator
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

int data;
if (world_rank == 0) {
  data = 100;
  printf("Process 0 broadcasting data %d\n", data);
  MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
} else {
  MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
  printf("Process %d received data %d from root process\n", world_rank, data);
}
```

```
Process 0 broadcasting data 100
Process 2 received data 100 from root process
Process 3 received data 100 from root process
Process 1 received data 100 from root process
```

https://rookiehpc.org/mpi/docs/mpi_bcast/index.html

`MPI_Bcast` Implementation

```
void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
              MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    int i;
    for (i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(data, count, datatype, i, 0, communicator);
      }
    }
  } else {
    // If we are a receiver process, receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator,
             MPI_STATUS_IGNORE);
  }
}
```

One way to implement `Bcast` is a simple wrapper around send / receive

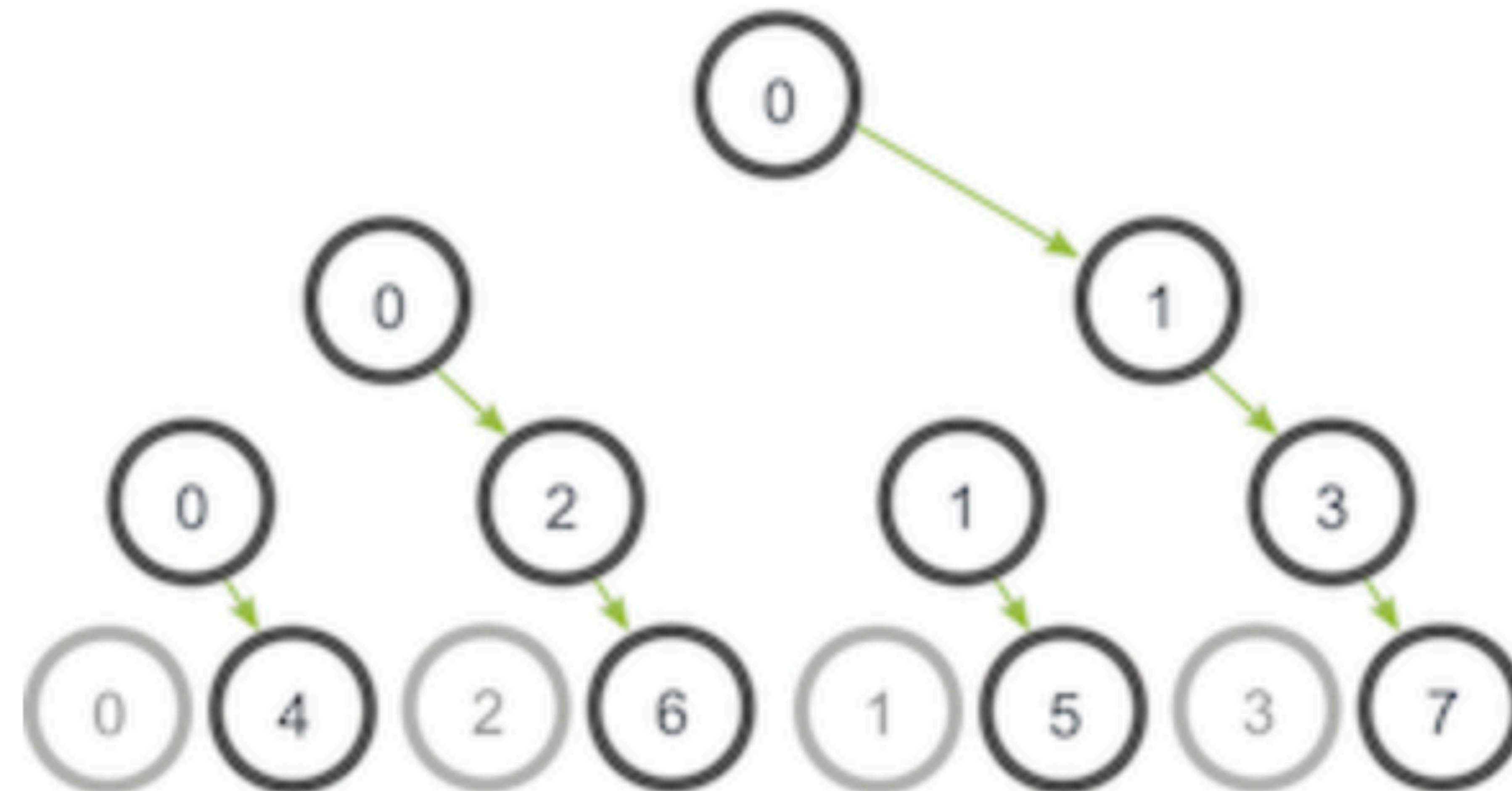
https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

Faster Broadcast via Trees

- The previous function is inefficient because it **serializes the broadcast**.

- Imagine that each processor has **one ingoing/outgoing network link**. The previous algorithm just uses one network link from process 0 to send all the data.

- A tree-based algorithm would take $\lg n$ rounds.



https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

Broadcast Performance Comparison

The tree-based Bcast scales better with more processes.

Processors	my_bcast	MPI_Bcast
2	0.0344	0.0344
4	0.1025	0.0817
8	0.2385	0.1084
16	0.5109	0.1296

https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

MPI_Scatter Example

Assuming 4 processes

```
    // Determine root's rank
    int root_rank = 0;

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Define my value
    int my_value;

    if(my_rank == root_rank)
    {
        int buffer[4] = {0, 100, 200, 300};
        printf("Values to scatter from process %d: %d, %d, %d, %d.\n", my_rank, buffer[0],
buffer[1], buffer[2], buffer[3]);
        MPI_Scatter(buffer, 1, MPI_INT, &my_value, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Scatter(NULL, 1, MPI_INT, &my_value, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
    }

    printf("Process %d received value = %d.\n", my_rank, my_value);
```

https://rookiehpc.org/mpi/docs/mpi_scatter/index.html

MPI_Scatter Example

Assuming 4 processes

```
    // Determine root's rank
    int root_rank = 0;

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Define my value
    int my_value;

    if(my_rank == root_rank)
    {
        int buffer[4] = {0, 100, 200, 300};
        printf("Values to scatter from process %d: %d, %d, %d, %d.\n", my_rank, buffer[0], 
buffer[1], buffer[2], buffer[3]);
        MPI_Scatter(buffer, 1, MPI_INT, &my_value, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Scatter(NULL, 1, MPI_INT, &my_value, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
    }

    printf("Process %d received value = %d.\n", my_rank, my_value);
```

Values to scatter from process 0: 0, 100, 200, 300

https://rookiehpc.org/mpi/docs/mpi_scatter/index.html

MPI_Scatter Example

Assuming 4 processes

```
    // Determine root's rank
    int root_rank = 0;

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Define my value
    int my_value;

    if(my_rank == root_rank)
    {
        int buffer[4] = {0, 100, 200, 300};
        printf("Values to scatter from process %d: … er[0],
buffer[1], buffer[2], buffer[3]);
        MPI_Scatter(buffer, 1, MPI_INT, &my_value, … ORLD);
    }
    else
    {
        MPI_Scatter(NULL, 1, MPI_INT, &my_value, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
    }

    printf("Process %d received value = %d.\n", my_rank, my_value);
```

Values to scatter from process 0: 0, 100, 200, 300

Process 0 received value = 0
Process 1 received value = 100
Process 2 received value = 200
Process 3 received value = 300

https://rookiehpc.org/mpi/docs/mpi_scatter/index.html

MPI_Gather Example

Assuming 4 processes

```
    // Get root and self rank
    int root_rank = 0;
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Define my value
    int my_value = my_rank * 100;
    printf("Process %d, my value = %d.\n", my_rank, my_value);

    if(my_rank == root_rank)
    {
        int buffer[4];
        MPI_Gather(&my_value, 1, MPI_INT, buffer, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
        printf("Values collected on process %d: %d, %d, %d, %d.\n", my_rank, buffer[0],
buffer[1], buffer[2], buffer[3]);
    }
    else
    {
        MPI_Gather(&my_value, 1, MPI_INT, NULL, 0, MPI_INT, root_rank, MPI_COMM_WORLD);
    }
```

https://rookiehpc.org/mpi/docs/mpi_gather/index.html

`MPI_Gather` Example

Assuming 4 processes

```
    // Get root and self rank
    int root_rank = 0;
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Define my value
    int my_value = my_rank * 100;
    printf("Process %d, my value = %d.\n", my_rank, my_value);

    if(my_rank == root_rank)
    {
        int buffer[4];
        MPI_Gather(&my_value, 1, MPI_INT, buffer, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
        printf("Values collected on process %d: %d, %d, %d, %d.\n", my_rank, buffer[0],
buffer[1], buffer[2], buffer[3]);
    }
    else
    {
        MPI_Gather(&my_value, 1, MPI_INT, NULL, 0, MPI_INT, root_rank, MPI_COMM_WORLD);
    }
```

Only root needs a valid buffer

Others can pass in NULL

https://rookiehpc.org/mpi/docs/mpi_gather/index.html

MPI_Gather Example

Assuming 4 processes

```
    // Get root and self rank
    int root_rank = 0;
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my

    // Define my value
    int my_value = my_rank * 100;
    printf("Process %d, my valu = %d.\n", my_rank, my_value);

    if(my_rank == root_rank)
    {
        int buffer[4];
        MPI_Gather(&my_value, 1, MPI_INT, buffer, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
        printf("Values collected on process %d: %d, %d, %d, %d.\n", my_rank, buffer[0],
buffer[1], buffer[2], buffer[3]);
    }
    else
    {
        MPI_Gather(&my_value, 1, MPI_INT, NULL, 0, MPI_INT, root_rank, MPI_COMM_WORLD);
    }
```

Output: Values collected on process 0: 0, 100, 100, 300

https://rookiehpc.org/mpi/docs/mpi_gather/index.html

Assuming 4 processes

Averaging With MPI_Reduce

```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
for (int i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
         global_sum / (world_size * num_elements_per_proc));
}
```

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Assuming 4 processes

Averaging With MPI_Reduce

Each process creates random numbers

```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
for (int i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
         global_sum / (world_size * num_elements_per_proc));
}
```

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Assuming 4 processes

Averaging With MPI_Reduce

```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
for (int i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
         global_sum / (world_size * num_elements_per_proc));
}
```

Each process creates random numbers

Local sum per process

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Assuming 4 processes

Averaging With MPI_Reduce

```c
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
for (int i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
         global_sum / (world_size * num_elements_per_proc));
}
```

Each process creates random numbers

Local sum per process

Reduce local_sum on root

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Assuming 4 processes

Averaging With MPI_Reduce

```c
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
for (int i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
         global_sum / (world_size * num_elements_per_proc));
}
```

Each process creates random numbers

Local sum per process

Reduce local_sum on root

Global average

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Assuming 4 processes

Averaging With MPI_Reduce

Output:

```
Local sum for process 0 - 51.385098, avg = 0.513851
Local sum for process 1 - 51.842468, avg = 0.518425
Local sum for process 2 - 49.684948, avg = 0.496849
Local sum for process 3 - 47.527420, avg = 0.475274
Total sum = 200.439941, avg = 0.501100
```

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Example: Standard Deviation

Standard deviation is a measure of the dispersion of numbers from their mean.

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

Standard deviation

Number from list

Number of values

Mean of all numbers

How would you do it on a distributed set of numbers with MPI?

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Example: Standard Deviation With `MPI_Allreduce`

```
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
for (int i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Reduce all of the local sums into the global sum in order to
// calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);

// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean);
}
```

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Example: Standard Deviation With `MPI_Allreduce`

```
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
for (int i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Reduce all of the local sums into the global sum in order to
// calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);

// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean);
}
```

Compute local sum

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Example: Standard Deviation With `MPI_Allreduce`

```
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
for (int i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Reduce all of the local sums into the global sum in order to
// calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);

// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean);
}
```

Compute local sum

Combine local sums to get mean

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Example: Standard Deviation With `MPI_Allreduce`

```
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
for (int i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Reduce all of the local sums into the global sum in order to
// calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);

// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean);
}
```

Compute local sum

Combine local sums to get mean

Sum of squared diffs

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Example: Standard Deviation With `MPI_Allreduce`

Reduce local squared differences into global squared difference

```
// Reduce the global sum of the squared differences to the
// process and print off the answer
float global_sq_diff;
MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_FLOAT, MPI_SUM, 0,
           MPI_COMM_WORLD);

// The standard deviation is the square root of the mean of the
// squared differences.
if (world_rank == 0) {
  float stddev = sqrt(global_sq_diff /
                      (num_elements_per_proc * world_size));
  printf("Mean - %f, Standard deviation = %f\n", mean, stddev);
}
```

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Example: Standard Deviation With `MPI_Allreduce`

```c
// Reduce the global sum of the squared differences to the
// process and print off the answer
float global_sq_diff;
MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_FLOAT, MPI_SUM, 0,
           MPI_COMM_WORLD);

// The standard deviation is the square root of the mean of the
// squared differences.
if (world_rank == 0) {
  float stddev = sqrt(global_sq_diff /
                      (num_elements_per_proc * world_size));
  printf("Mean - %f, Standard deviation = %f\n", mean, stddev);
}
```

Reduce local squared differences into global squared difference

Final stddev uses global_sq_diff and N

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

MPI_Reduce

e.g., MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, etc.

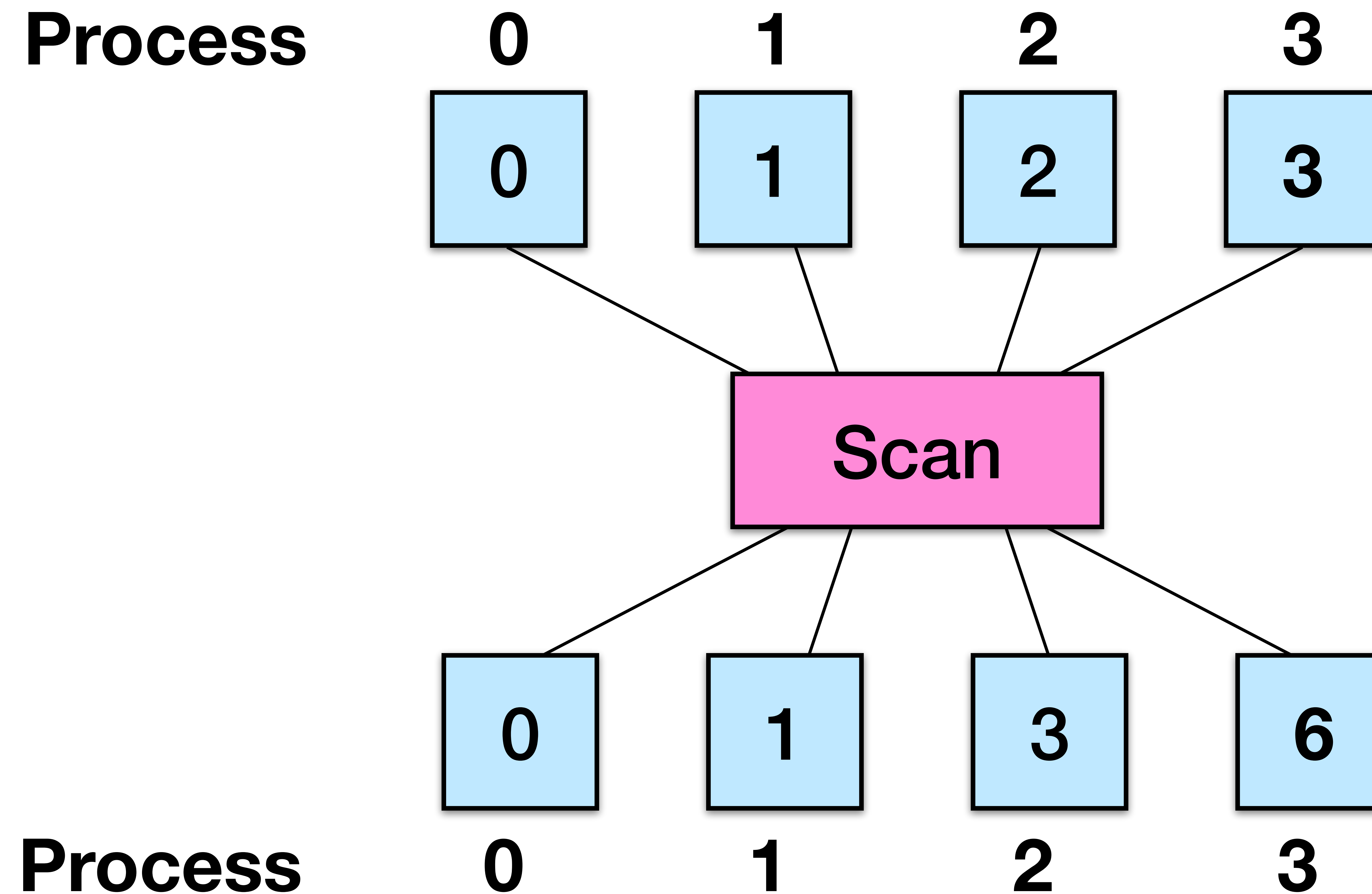
```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

- The send_data parameter is an array of elements that each process wants to reduce. Each send_data array has count elements.
- The recv_data is only relevant on the root and has # elements = count

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

`MPI_Scan`

`MPI_Scan` performs an **inclusive scan** across all MPI processes in the given communicator.

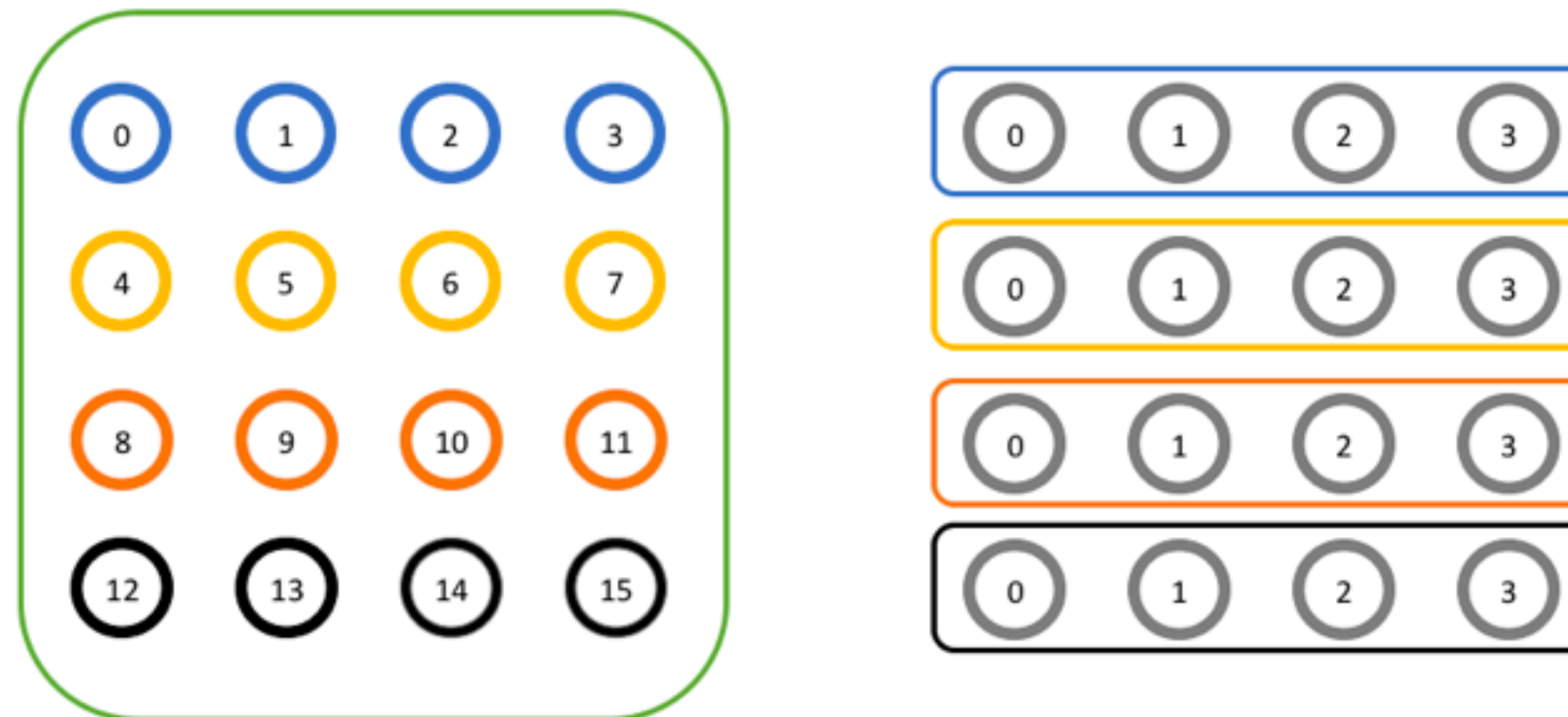


https://rookiehpc.org/mpi/docs/mpi_scan/index.html

MPI Communicators

- Up to now, we have only used the default communicator, `MPI_COMM_WORLD`.

- For more complex use cases, it might be helpful to have more communicators - e.g., performing operations on a **subset of the processors**.



https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/

MPI_Comm_split

- MPI_Comm_split creates new communicators by **"splitting" a communicator into a group of sub-communicators** based on the input values color and key.

- The original communicator doesn't go away - a new one is created on each process.

```
MPI_Comm_split(
  MPI_Comm comm,
  int color,
  int key,
  MPI_Comm* newcomm)
```

Basis for the new communicators

All processes with the same color will be in the same communicator.

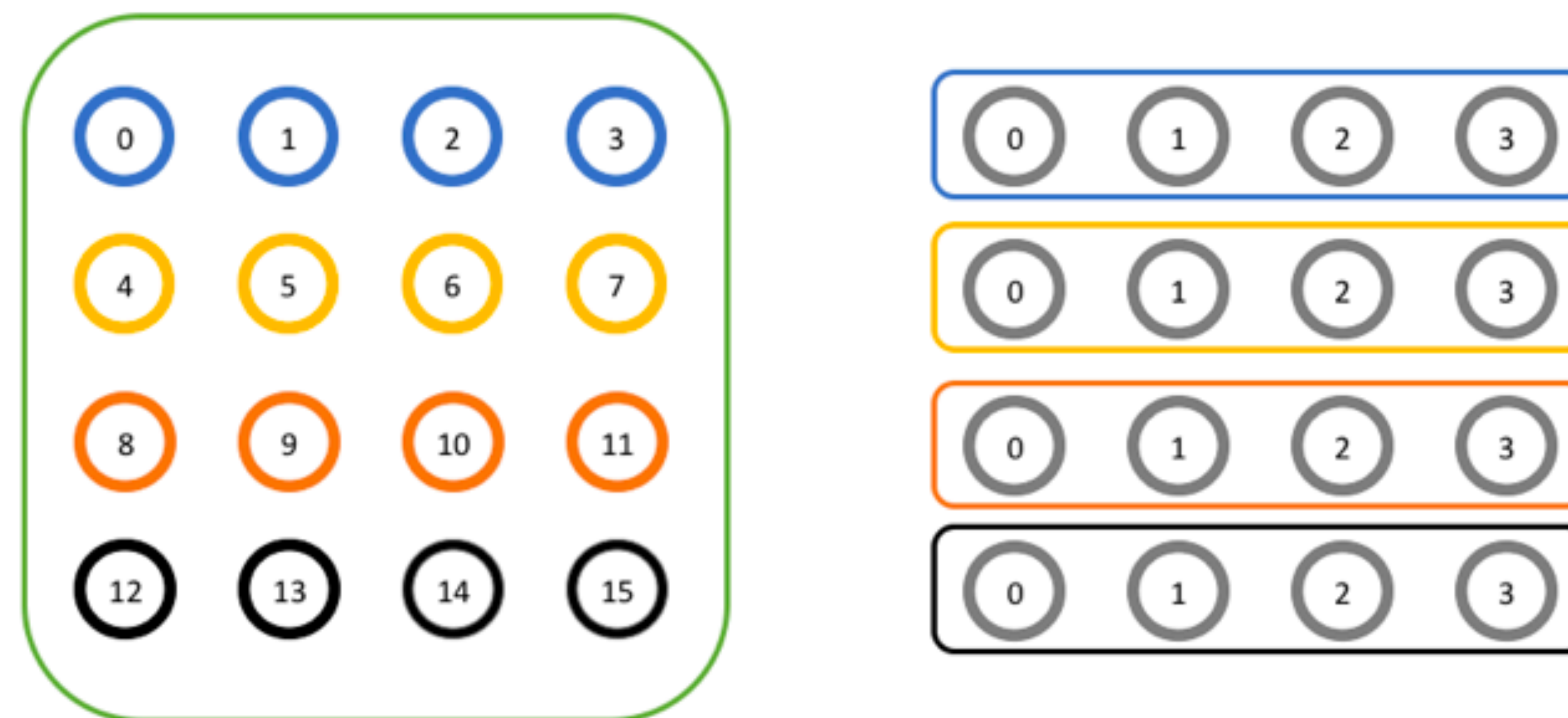
Ordering (rank) within the new communicator

Return to the user

https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/

Example: Split 16 processors into 4x4 grid

```
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);
```

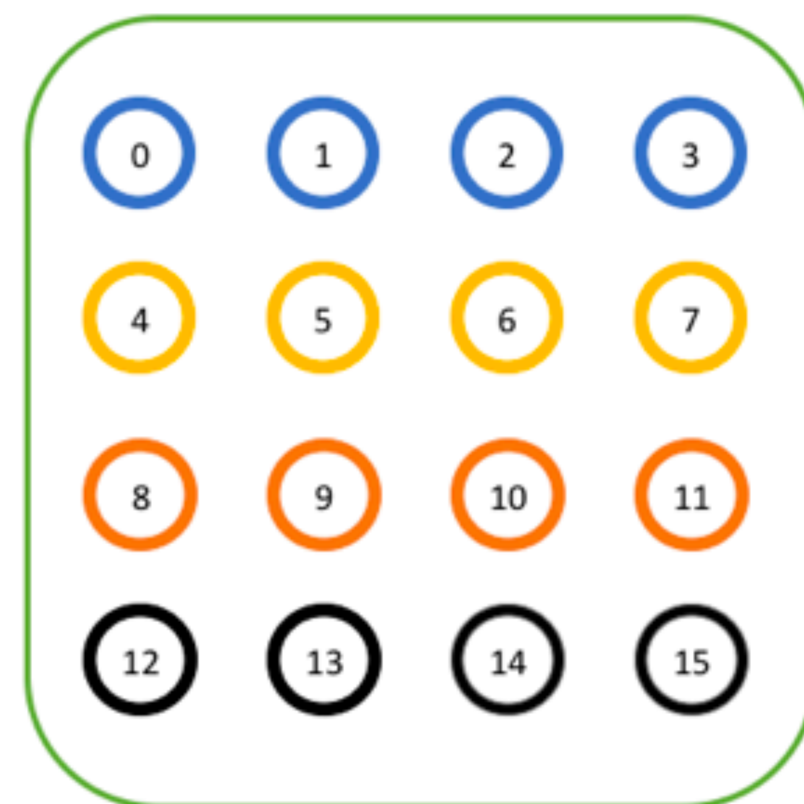


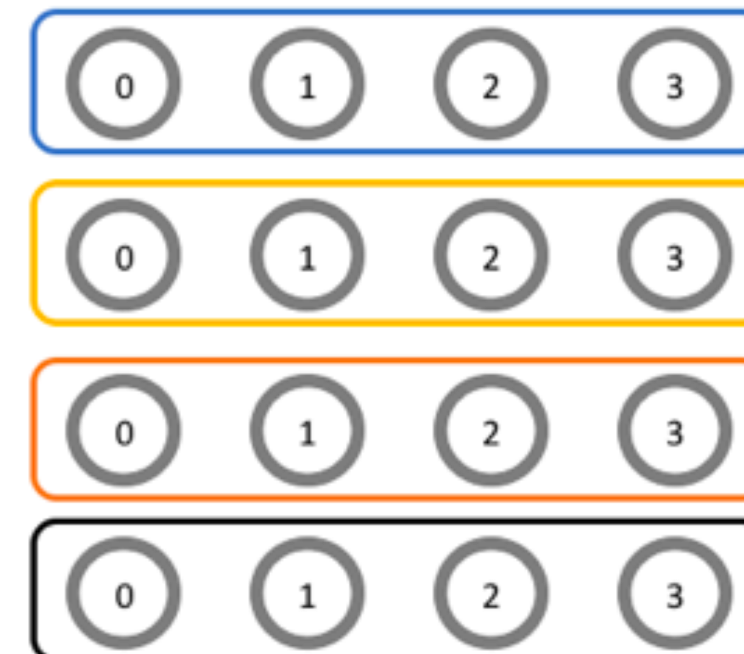
https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/

Example: Split 16 processors into 4x4 grid

```
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);
```

Get rank and size of original communicator





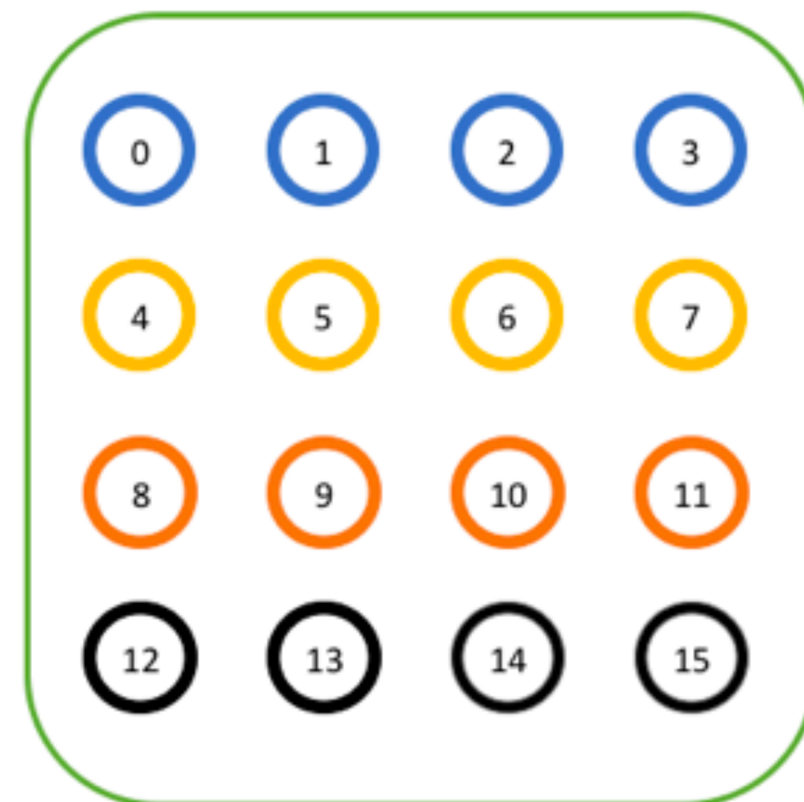
https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/

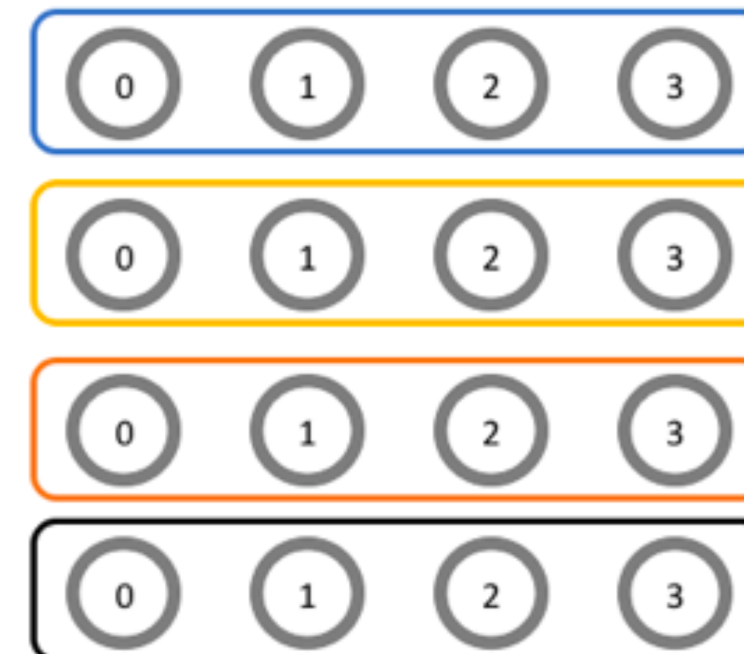
Example: Split 16 processors into 4x4 grid

```c
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);
```

Get rank and size of original communicator

Get color of local process





https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/

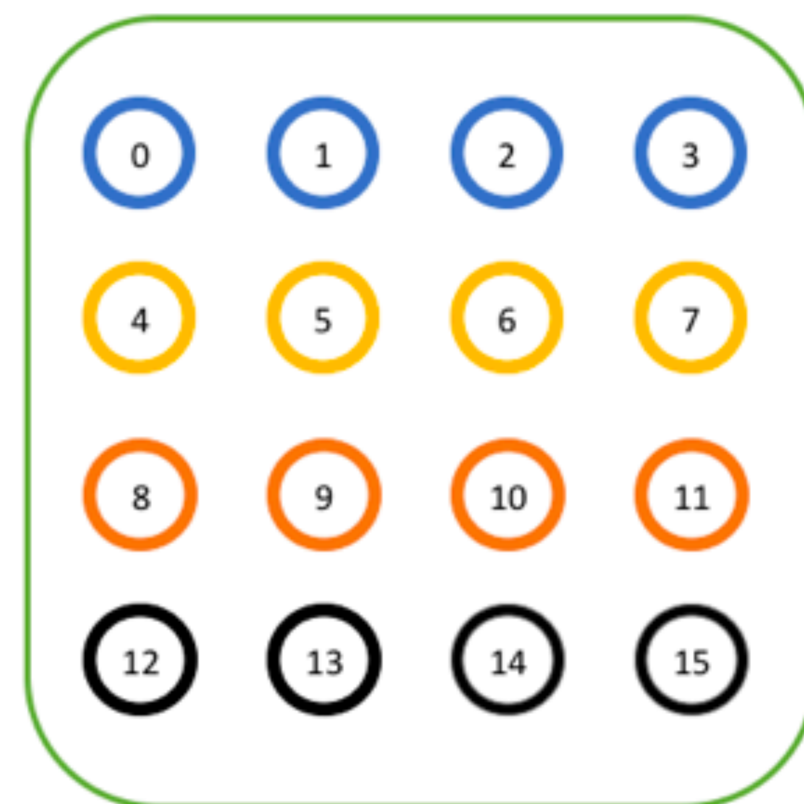
Example: Split 16 processors into 4x4 grid

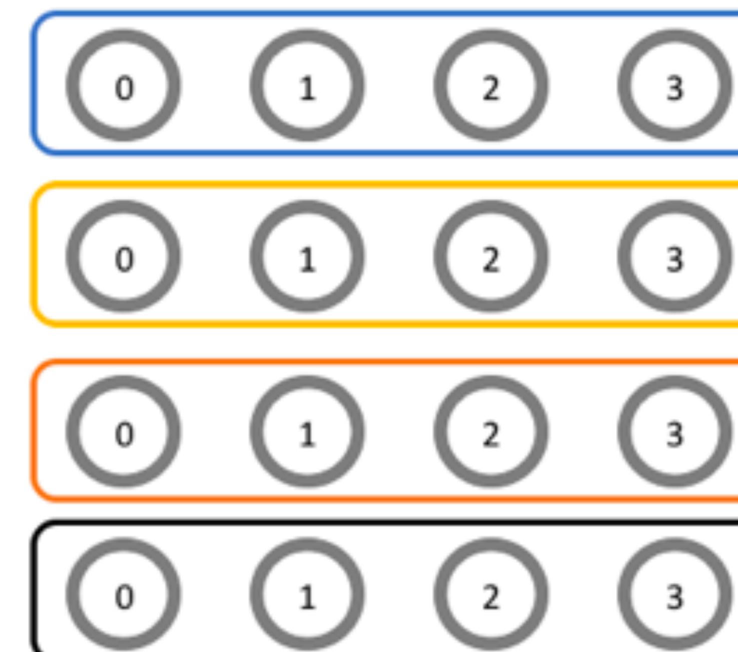
```
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on ...

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);
```

Get rank and size of original communicator

Get color of local process

Use original rank as the key for the split operation: processes are in the same order as in the original communicator





https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/

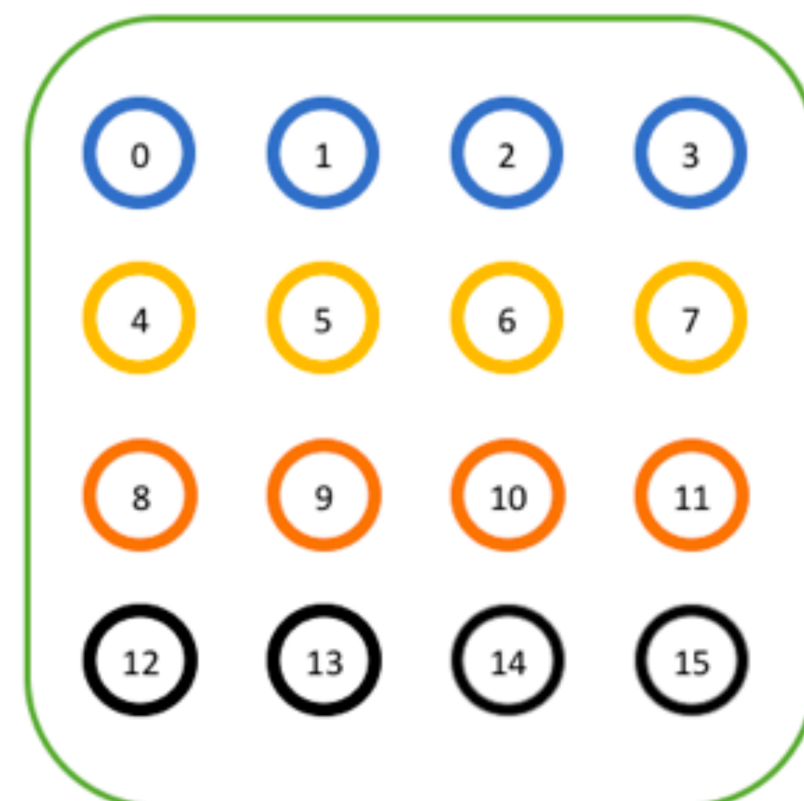
Example: Print and Cleanup

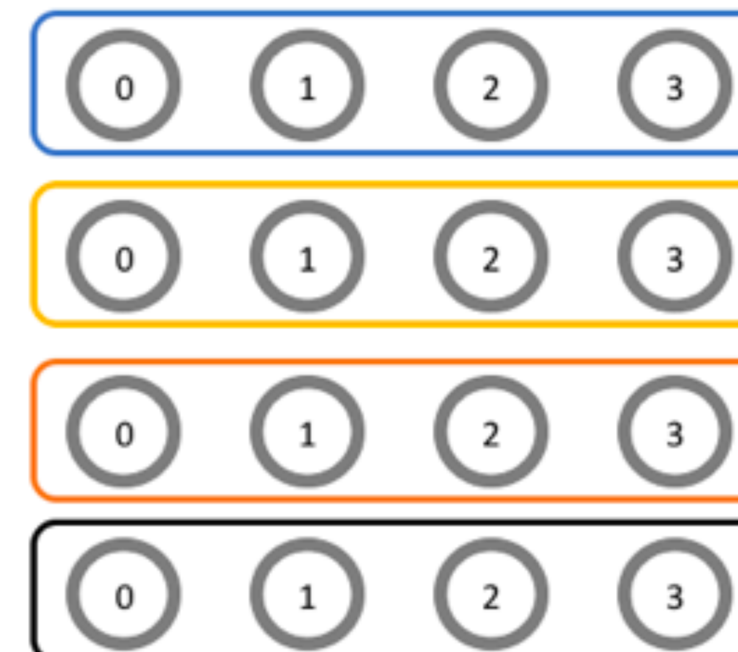
```
// DO SPLIT

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
  world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

Get rank and size in new communicator

Print them

Cleanup





https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/

Example: Print Split Results

Example output (probably will not be in order, but ordered here for ease of illustration):

```
WORLD RANK/SIZE: 0/16   ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 1/16   ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 2/16   ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 3/16   ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 4/16   ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 5/16   ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 6/16   ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 7/16   ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 8/16   ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 9/16   ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 10/16  ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 11/16  ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 12/16  ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 13/16  ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 14/16  ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 15/16  ROW RANK/SIZE: 3/4
```

https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/