Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

- Different sending modes: Bsend, Ssend, Rsend
- Non-blocking send/recv: Isend, Irecv
- Collective operations: MPI_Reduce

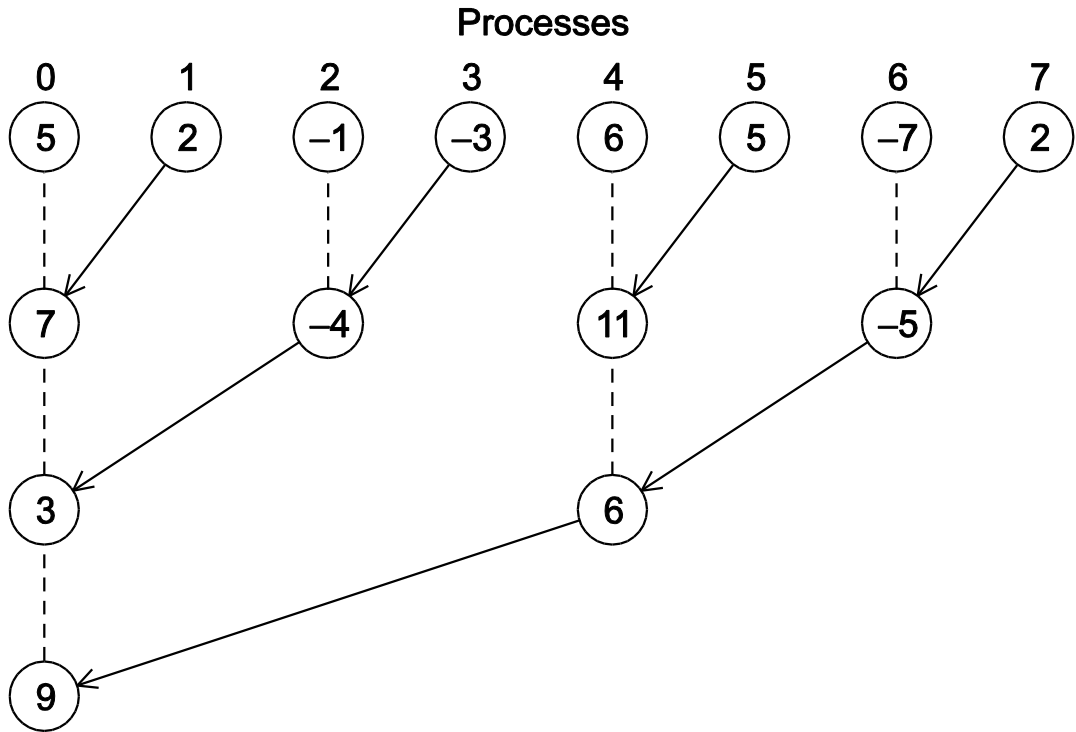
Issues with the trapezoidal rule implementation (1)

When doing the global sum, p-1 processes send their data to one process, which then computes all the sums. Unbalance! How long does it take?

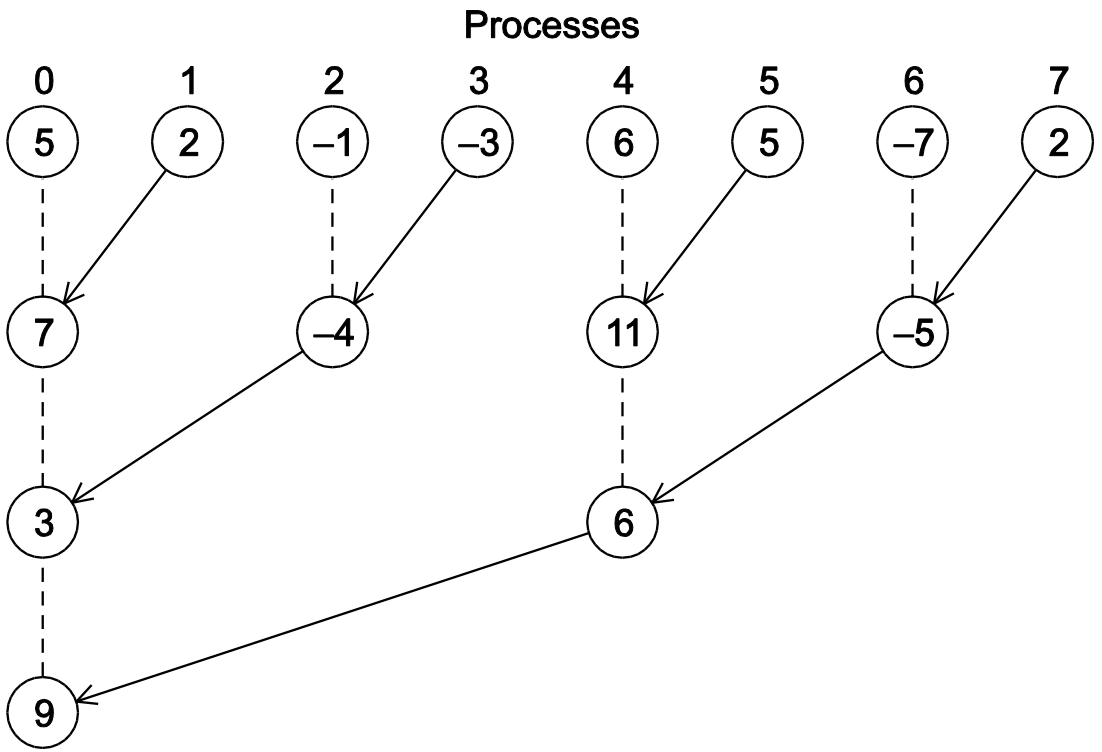
- For process 0: $(p-1)*(T_{sum} + T_{recv})$
- For all the other processes: T_{send}

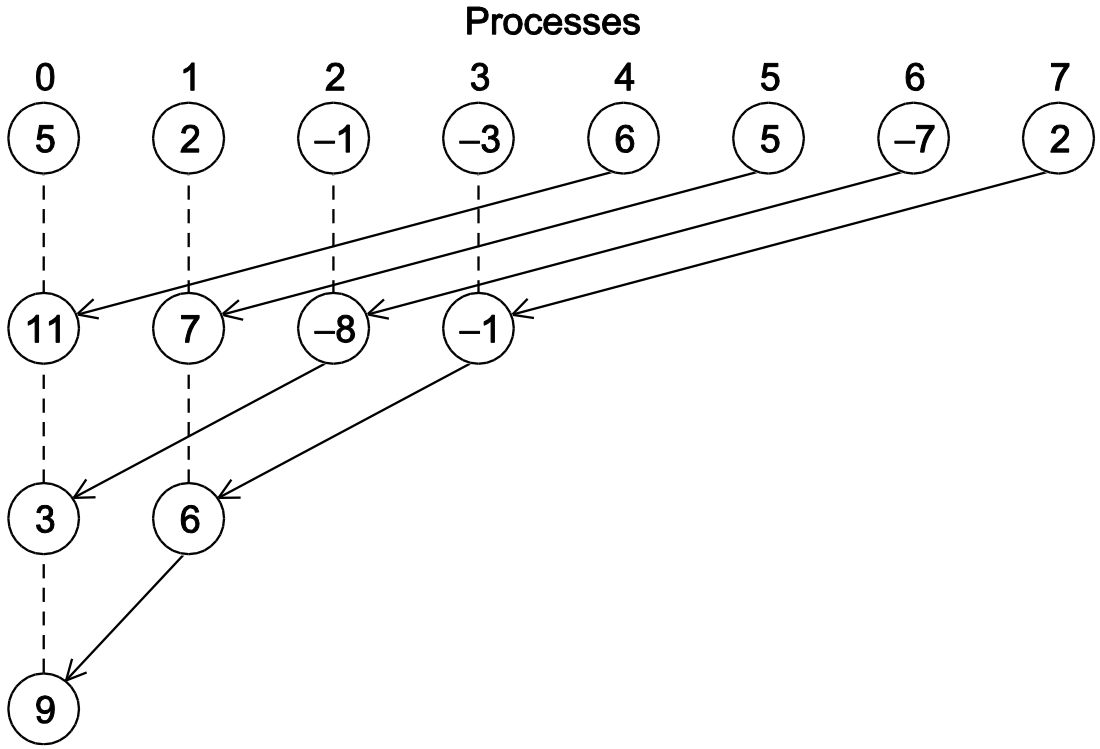
Alternative

- For process 0: $\log2(p)*(T_{sum} + T_{recv})$



Different valid trees

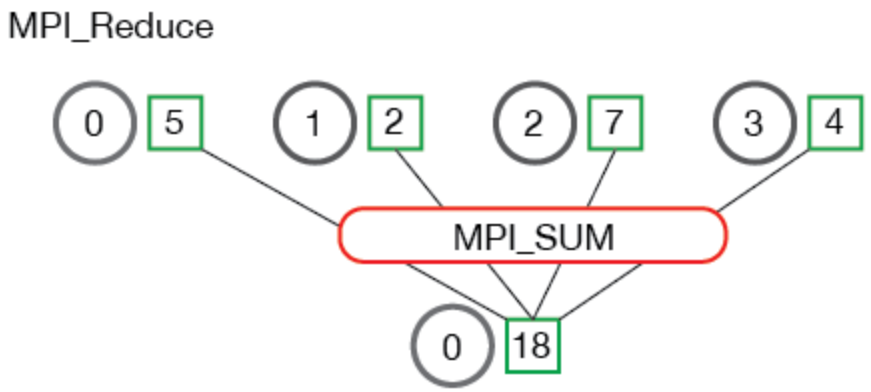


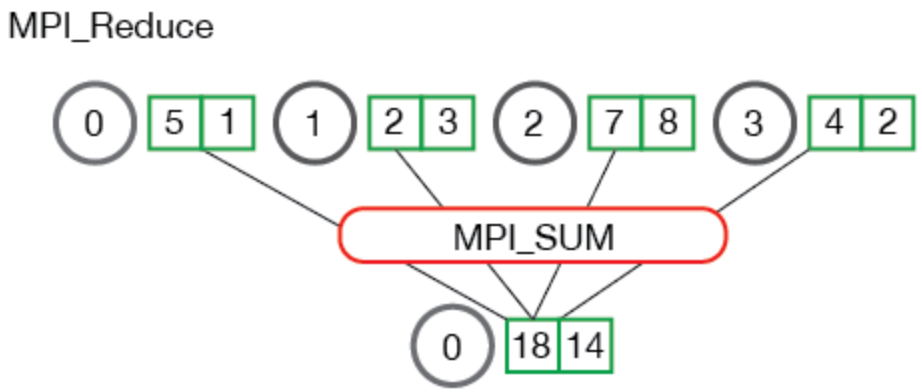


- The optimal way to compute a global sum depends on the number of processes, the size of the data, and the system we are running on (how many NICs, how the nodes are connected, etc…)
- Having a native way to express the global sum would simplify programming and improve performance

Collective Communication

MPI_Reduce


MPI_Reduce
0
5
1
2
2
7
3
4
MPI_SUM
0
18


MPI_Reduce
0
5
1
1
2
3
2
7
8
3
4
2
MPI_SUM
0
18
14

MPI_Reduce

```
int MPI_Reduce(
      void*          input_data_p   /* in  */,
      void*          output_data_p  /* out */,
      int            count          /* in  */,
      MPI_Datatype   datatype       /* in  */,
      MPI_Op         operator       /* in  */,
      int            dest_process   /* in  */,
      MPI_Comm       comm           /* in  */);
```

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

MPI_Reduce

One call for all the processes

```c
int main(void) {
   int my_rank, comm_sz, n, local_n;
   double a, b, h, local_a, local_b;
   double local_int, total_int;

   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   Get_input(my_rank, comm_sz, &a, &b, &n);

   h = (b-a)/n;          /* h is the same for all processes */
   local_n = n/comm_sz;  /* So is the number of trapezoids  */

   /* Length of each process' interval of
    * integration = local_n*h.  So my interval
    * starts at: */
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   local_int = Trap(local_a, local_b, local_n, h);

   /* Add up the integrals calculated by each process */
   MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   /* Print the result */
   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.15e\n",
          a, b, total_int);
   }

   /* Shut down MPI */
   MPI_Finalize();

   return 0;
} /*  main  */
```

MPI_Reduce Operators

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

You can create custom operators with **MPI_Op_create**

Questions?

Caveats

- All the processes in the communicator must call the same collective function.
- For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.
- The arguments passed by each process to an MPI collective communication must be "compatible."
- For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.
- The output_data_p argument is only used on dest_process.
- However, all of the processes still need to pass in an actual argument corresponding to output_data_p, even if it's just NULL.
- Point-to-point communications are matched on the basis of tags and communicators, collective communications don't use tags.
- They're matched solely on the basis of the communicator and the order in which they're called.

Matching Example

All the following calls are done on MPI_COMM_WORLD, have 0 as destination, and MPI_SUM as operator

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

b = 1+2+1=4
d = 2+1+2=5

b = ???
d = ???

b = ???
d = ???

Questions?

Issues with the trapezoidal rule implementation (2)

```
void Get_input(
      int       my_rank    /* in  */,
      int       comm_sz    /* in  */,
      double*   a_p        /* out */,
      double*   b_p        /* out */,
      int*      n_p        /* out */) {
   int dest;

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
      for (dest = 1; dest < comm_sz; dest++) {
         MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      }
   } else { /* my_rank != 0 */
      MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
      MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
      MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
   }
}  /* Get_input */
```

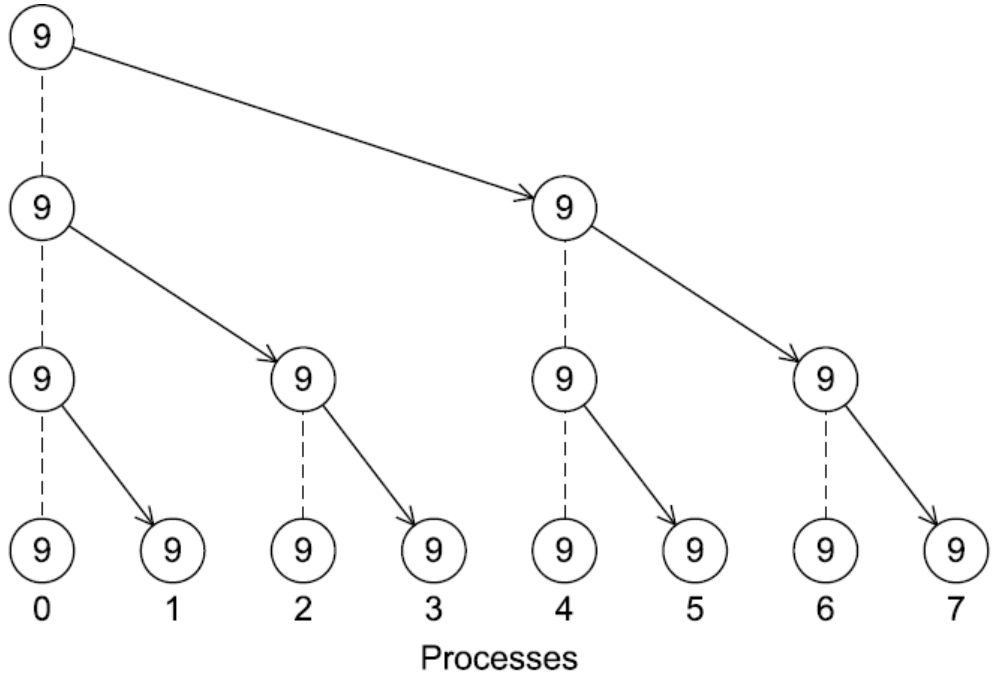
When doing the global sum, p-1 processes recv the data from one process. Unbalance! How long does it take?

For process 0: $(p-1)*(T_{send})$

For all the other processes: T_{recv}

Alternative

- For process 0: $\log2(p)*(T_{send})$



MPI_Bcast

- Data belonging to a single process is sent to all of the processes in the communicator.

```
int MPI_Bcast(
        void*           data_p          /* in/out */,
        int             count           /* in     */,
        MPI_Datatype    datatype        /* in     */,
        int             source_proc     /* in     */,
        MPI_Comm        comm            /* in     */);
```

MPI_Bcast

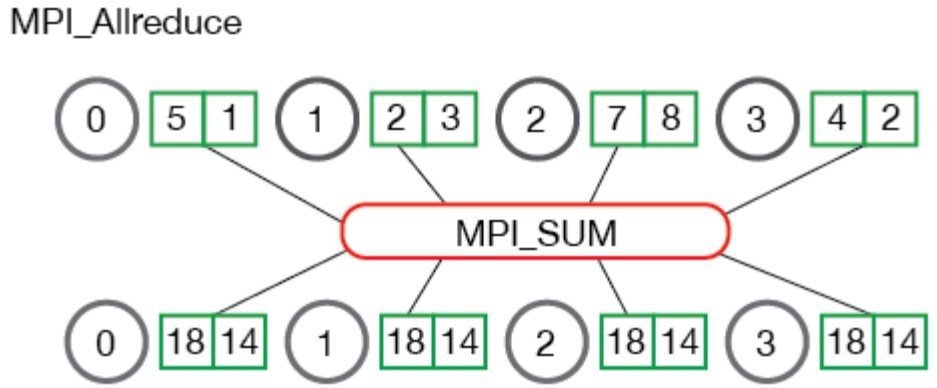
```
void Get_input(
      int       my_rank  /* in  */,
      int       comm_sz  /* in  */,
      double*   a_p      /* out */,
      double*   b_p      /* out */,
      int*      n_p      /* out */) {

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}  /* Get_input */
```

Questions?

MPI_Allreduce

- Conceptually, an MPI_Reduce followed by MPI_Bcast (i.e., compute a global sum and distribute the result to all the processes)

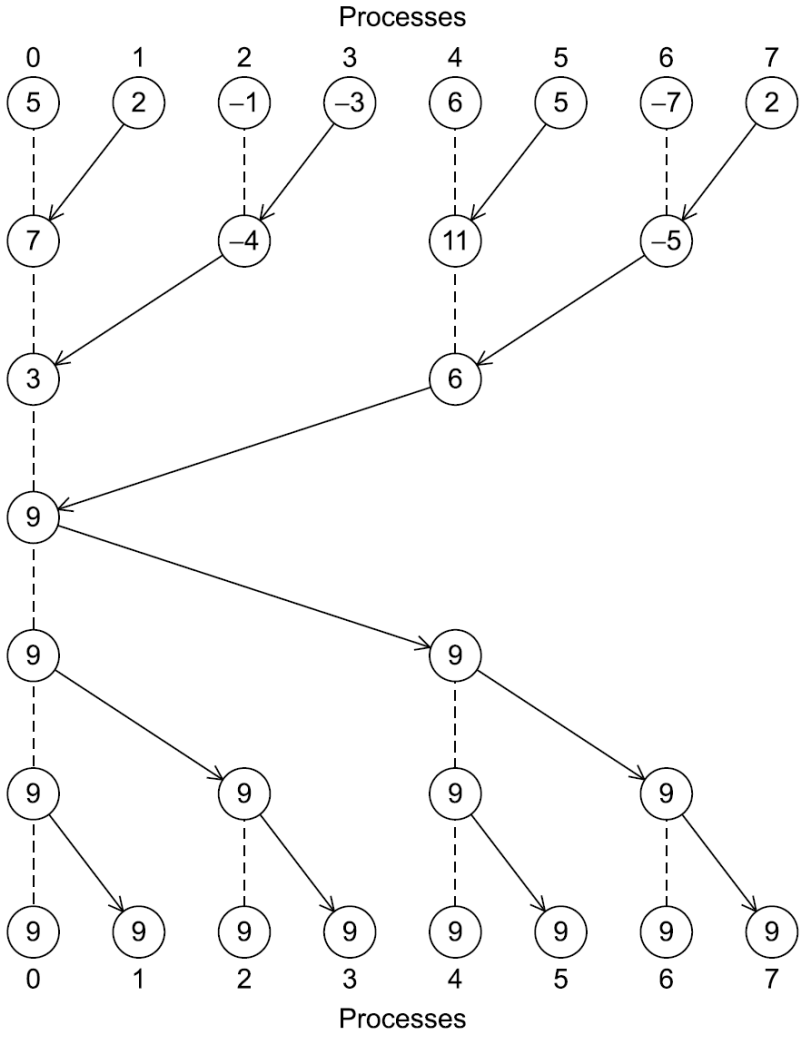



```
int  MPI_Allreduce(
        void*          input_data_p    /* in  */,
        void*          output_data_p   /* out */,
        int            count           /* in  */,
        MPI_Datatype   datatype        /* in  */,
        MPI_Op         operator        /* in  */,
        MPI_Comm       comm            /* in  */);
```

The argument list is identical to that for `MPI_Reduce`, except that there is no `dest_process` since all the processes should get the result.

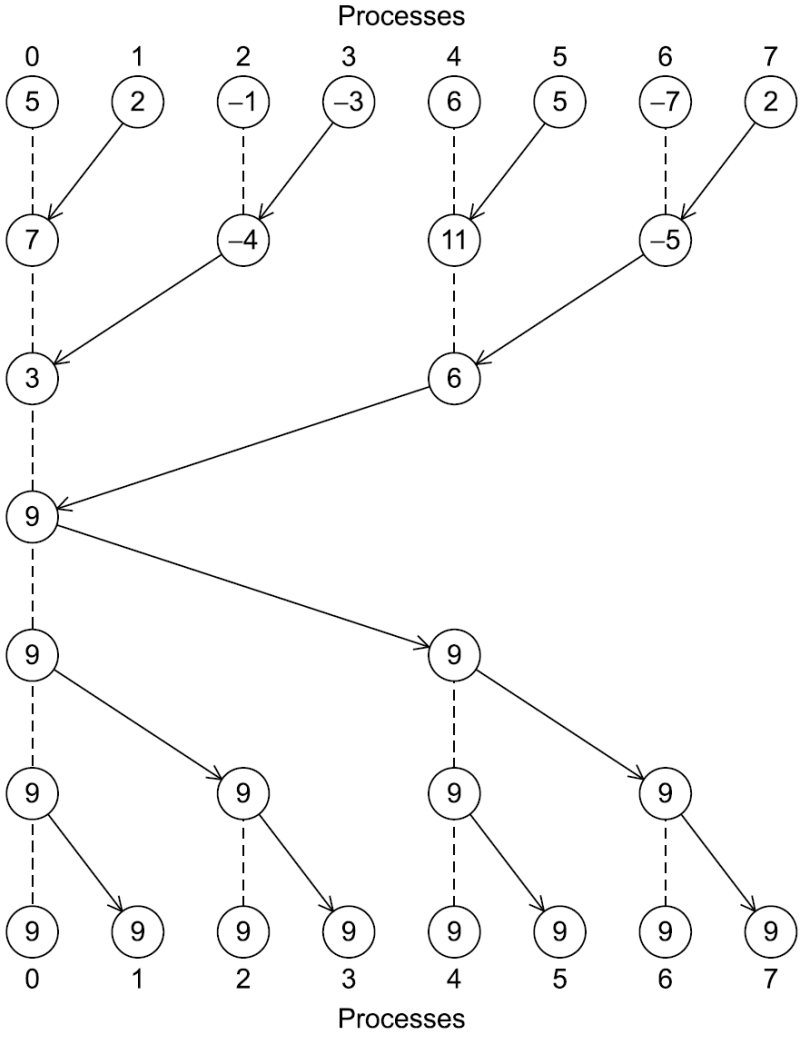
MPI_Allreduce

- Conceptually, an MPI_Reduce followed by MPI_Bcast (i.e., compute a global sum and distribute the result to all the processes)

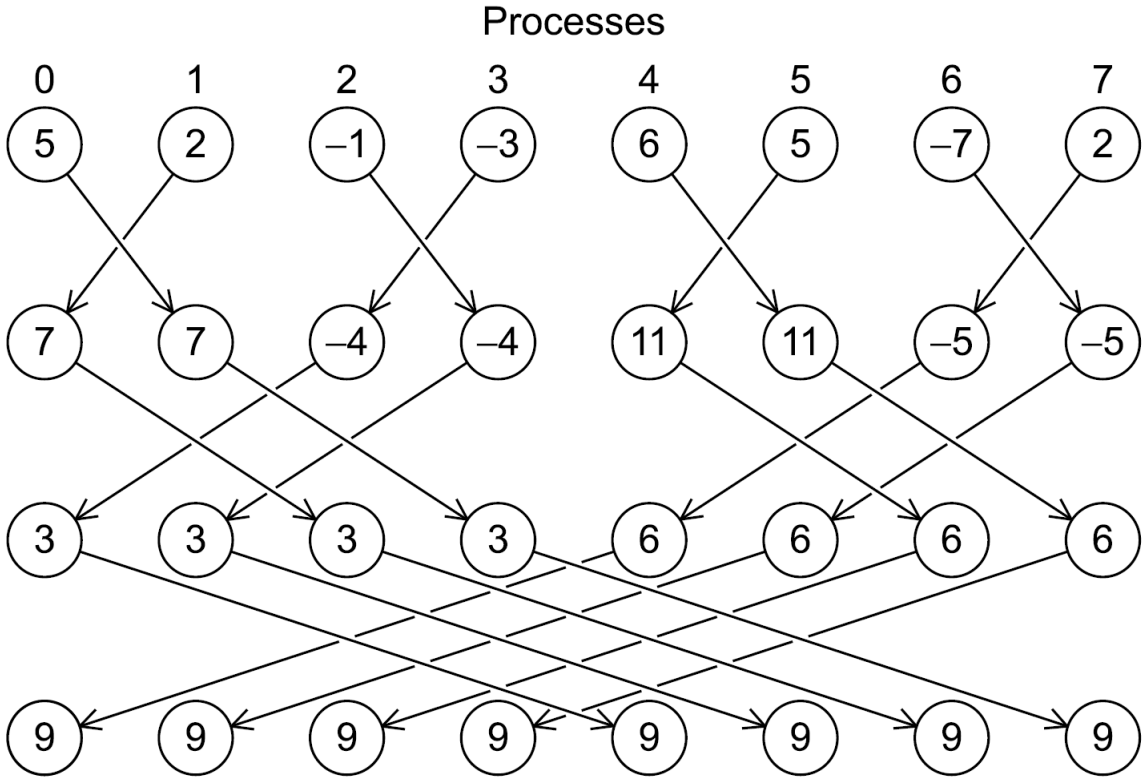


Is this the best way of doing it?

MPI_Allreduce



$T = 2*\log2(p)*T_{send}$



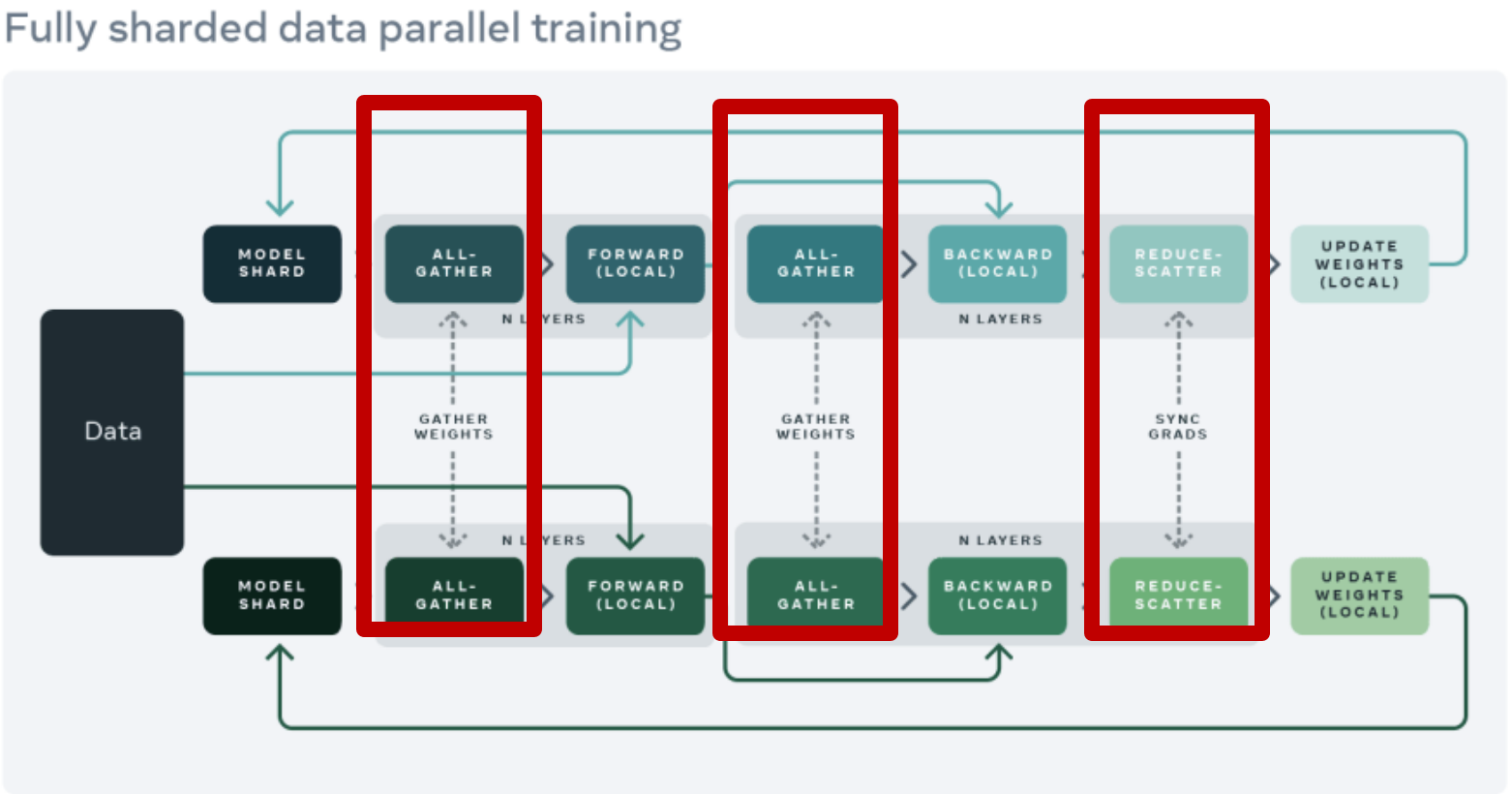
This is also known as **butterfly** pattern
(sometimes as recursive distance doubling)

$T = \log2(p)*(T_{send})$ **(Assuming send and recv happen at the same time)**

2x faster (other algos might be better depending on the data size)

Relevance of collective algorithms

- Widely used in large-scale parallel applications from many domains
- Account for a large fraction of the total runtime
- Highly relevant for distributed training of deep-learning models, e.g., Meta's FSDP training system:



https://engineering.fb.com/2021/07/15/open-source/fsdp/

Relevance of collective algorithms

- That's the reason why all the big players are designing their own collective communication library. E.g.,
  - NCCL (NVIDIA)
  - RCCL (AMD)
  - OneCCL (Intel)
  - MSCCL (Microsoft)
  - ….
- Given a collective (e.g., MPI_Reduce), how to select the best algorithm?
  - Automatically through heuristic
  - Manually
  - MPI implementations such as Open MPI do not make assumption on the underlying hardware, *CCL does
- Active research area, both from algorithmic and implementations standpoints

Questions?

Exercises

Exercises

3.2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and it's area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

i.e., number in circle : pi = total number of tosses : 4

Remember that if origin (0,0), the circle must respect the condition $x^2+y^2=r^2$

We can use this formula to estimate the value of with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

This is called a "Monte Carlo" method, since it uses randomness (the dart tosses). Write an MPI program that uses a Monte Carlo method to estimate π.

Exercises

3.2. Modify the trapezoidal rule so that it will correctly estimate the integral even if *comm_sz* doesn't evenly divide *n*. (You can still assume that $n \geq comm\_sz$)

3.4. Modify the program that just prints a line of output from each process (*mpi_output.c*) so that the output is printed in process rank order: process Os output first, then process 1s, and so on.

Exercises

3.1. Use MPI to implement the histogram program. Have process 0 read in the input data and distribute it among the processes. Also have process 0 print out the histogram.

Exercises

3.3. Write an MPI program that computes a tree-structured global sum. First write your program for the special case in which *comm_sz* is a power of two. Then, after you've gotten this version working, modify your program so that it can handle any *comm_sz*

3.9. Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 0, which prints them. You can assume that *n*, the order of the vectors, is evenly divisible by comm sz.

3.13. MPI Scatter and MPI Gather have the limitation that each process must send or receive the same number of data items. When this is not the case, we must use the MPI functions MPI Gatherv and MPI Scatterv. Look at the man pages for these functions, and modify your vector sum, dot product program so that it can correctly handle the case when *n* isn't evenly divisible by comm sz.

Example: Sum Between Vectors

$$
\begin{aligned}
\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\
&= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\
&= (z_0, z_1, \ldots, z_{n-1}) \\
&= \mathbf{z}
\end{aligned}
$$

Compute a vector sum.

Serial implementation of vector addition

```
void Vector_sum(double x[], double y[], double z[], int n) {
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}  /* Vector_sum */
```

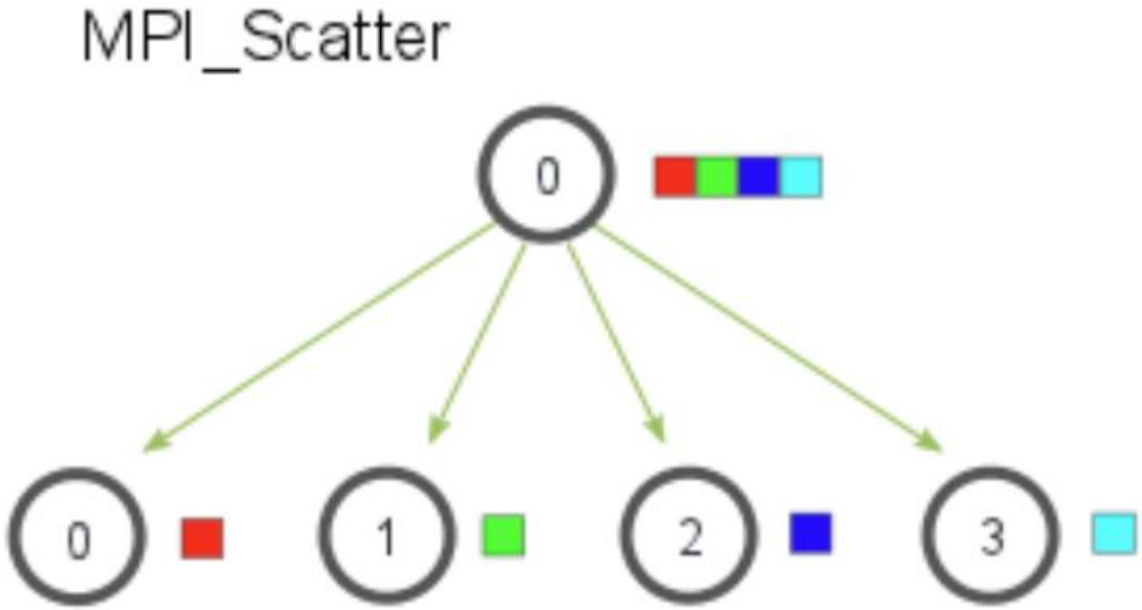
Parallel implementation of vector addition

```
void Parallel_vector_sum(
      double  local_x[]  /* in  */,
      double  local_y[]  /* in  */,
      double  local_z[]  /* out */,
      int     local_n    /* in  */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}  /* Parallel_vector_sum */
```

Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.



ATTENTION:
Different from
MPI_Bcast

Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

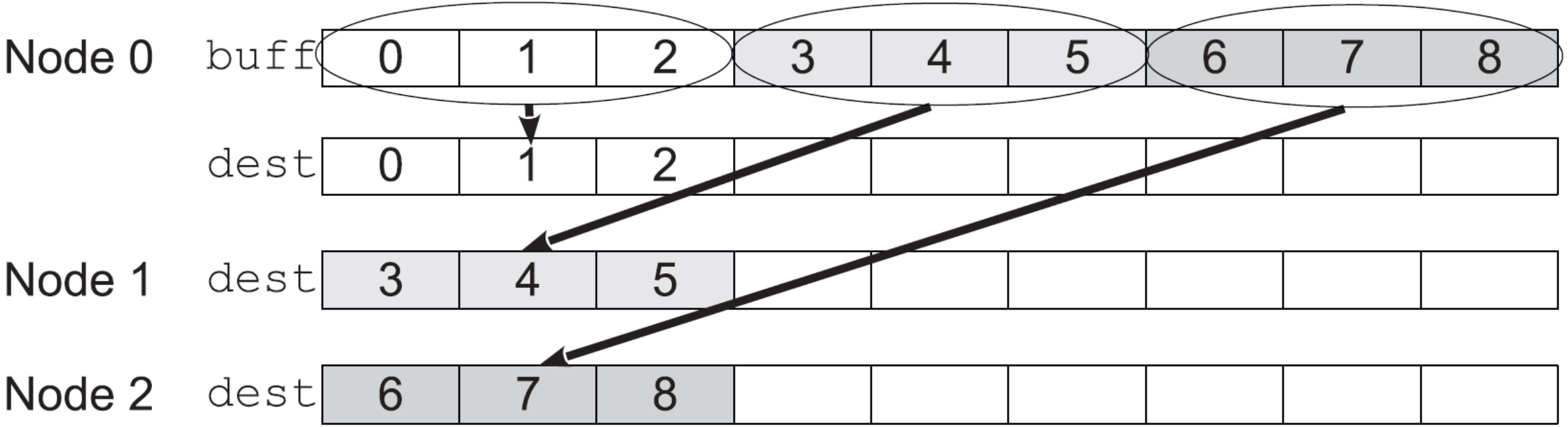
```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            src_proc    /* in  */,
      MPI_Comm       comm        /* in  */);
```

ATTENTION: This is the number of elements to send to each process, not the total number of elements!!!

What if I want to send a different number of elements to each rank? MPI_Scatterv

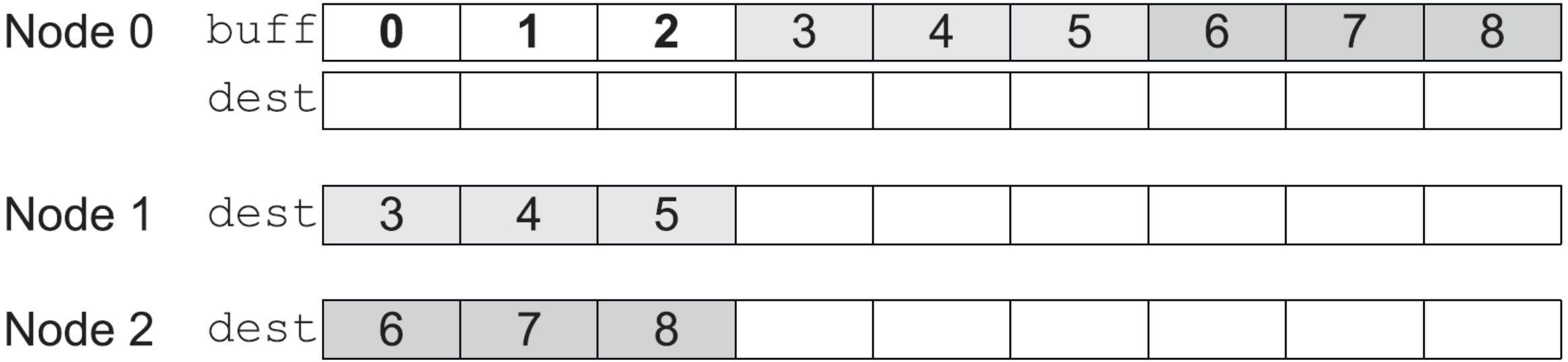
Scatter

```
MPI_Scatter(buff, 3, MPI_INT,
            dest, 3, MPI_INT, 0, MPI_COMM_WORLD);
```



Scatter – In Place

```
if(rank == 0)
    MPI_Scatter(buff, 3, MPI_INT,
                MPI_IN_PLACE, 3, MPI_INT, 0, MPI_COMM_WORLD);
else
    MPI_Scatter(buff, 3, MPI_INT,
                dest, 3, MPI_INT, 0, MPI_COMM_WORLD);
```



Reading and distributing a vector

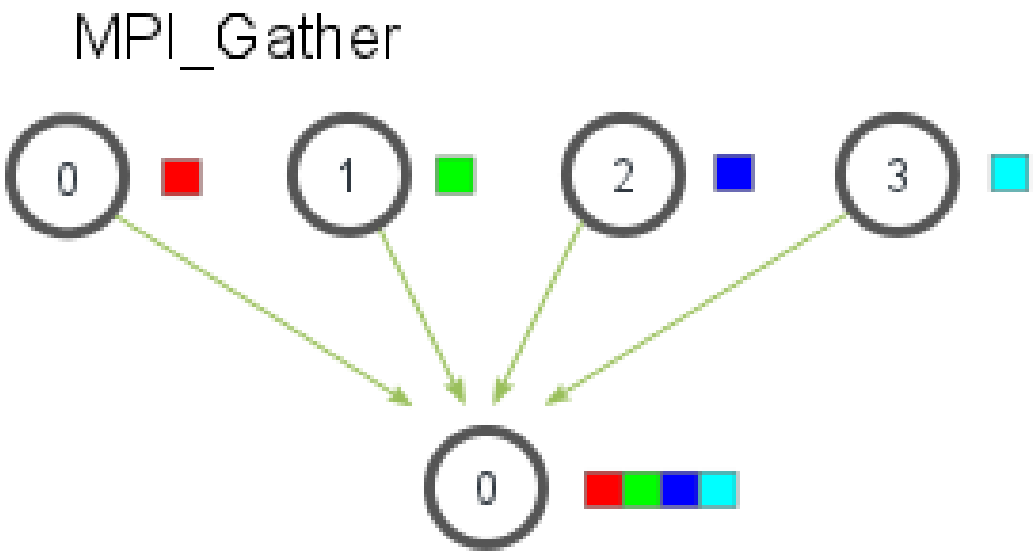
```
void Read_vector(
      double    local_a[]   /* out */,
      int       local_n     /* in  */,
      int       n           /* in  */,
      char      vec_name[]  /* in  */,
      int       my_rank     /* in  */,
      MPI_Comm  comm        /* in  */) {

   double* a = NULL;
   int i;

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
      free(a);
   } else {
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Read_vector */
```

Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.



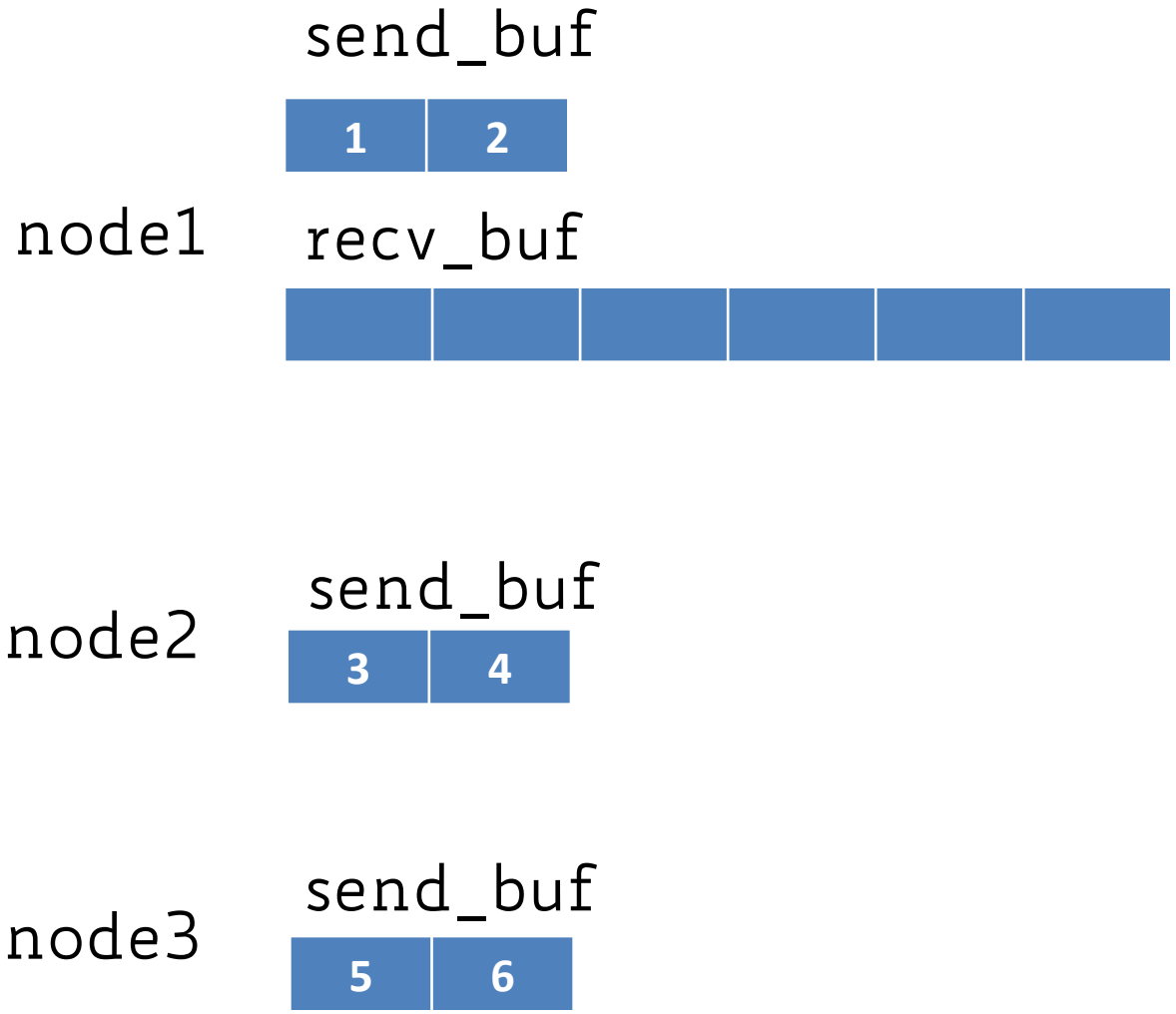
Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

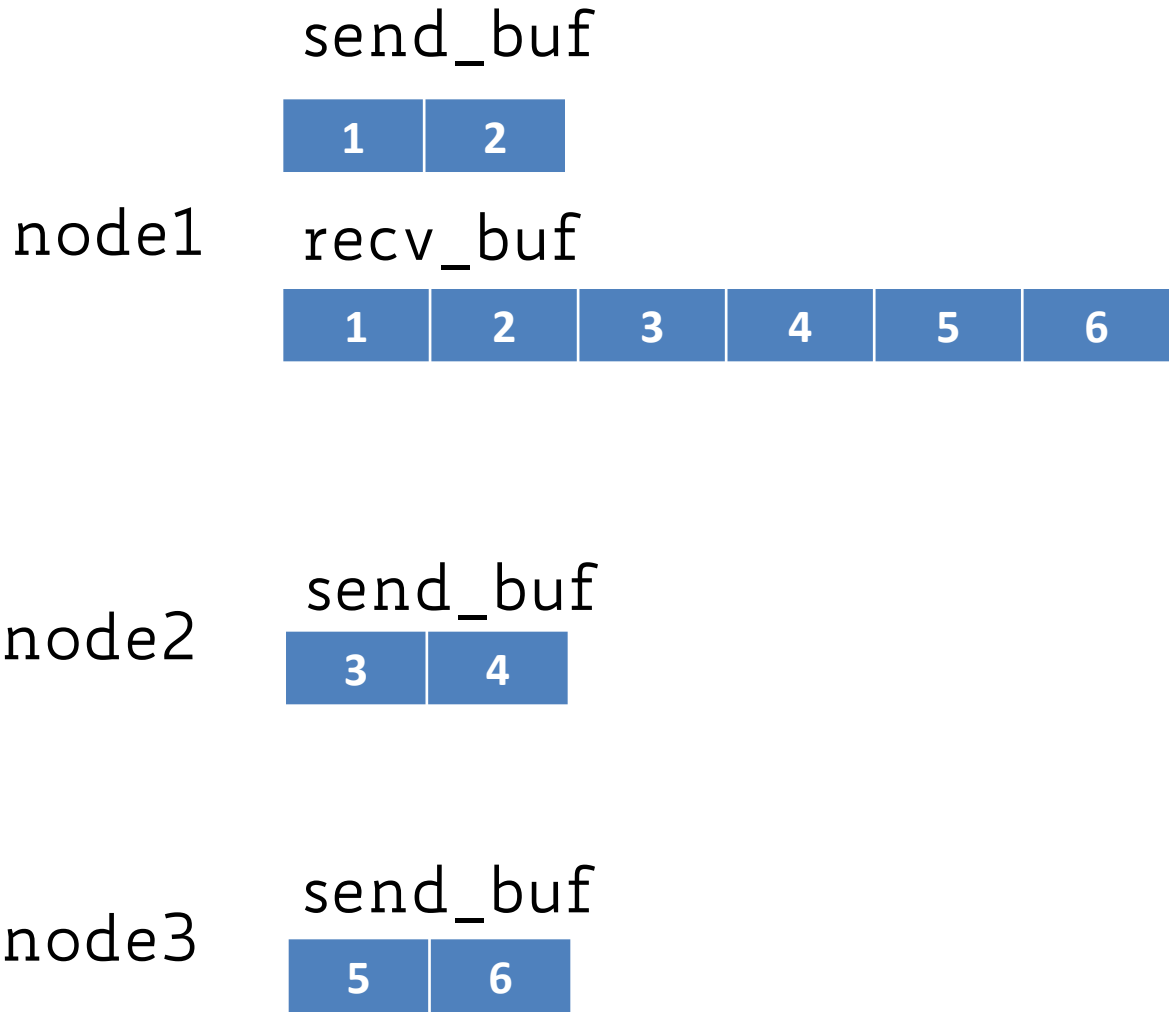
```
int MPI_Gather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      int            dest_proc    /* in  */,
      MPI_Comm       comm         /* in  */);
```

ATTENTION: This is the number of elements that each process sends, not the total number of elements in the final vector!!!

Gather



Gather


send_buf
1
2
node1
recv_buf
1
2
3
4
5
6
node2
send_buf
3
4
node3
send_buf
5
6

Print a distributed vector (1)

```
void Print_vector(
      double    local_b[]  /* in */,
      int       local_n    /* in */,
      int       n          /* in */,
      char      title[]    /* in */,
      int       my_rank    /* in */,
      MPI_Comm  comm       /* in */) {

   double* b = NULL;
   int i;
```

Print a distributed vector (2)

```
   if (my_rank == 0) {
      b = malloc(n*sizeof(double));
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(b);
   } else {
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
   }
} /* Print_vector */
```

MPI_Barrier

MPI_Barrier(MPI_Comm comm)

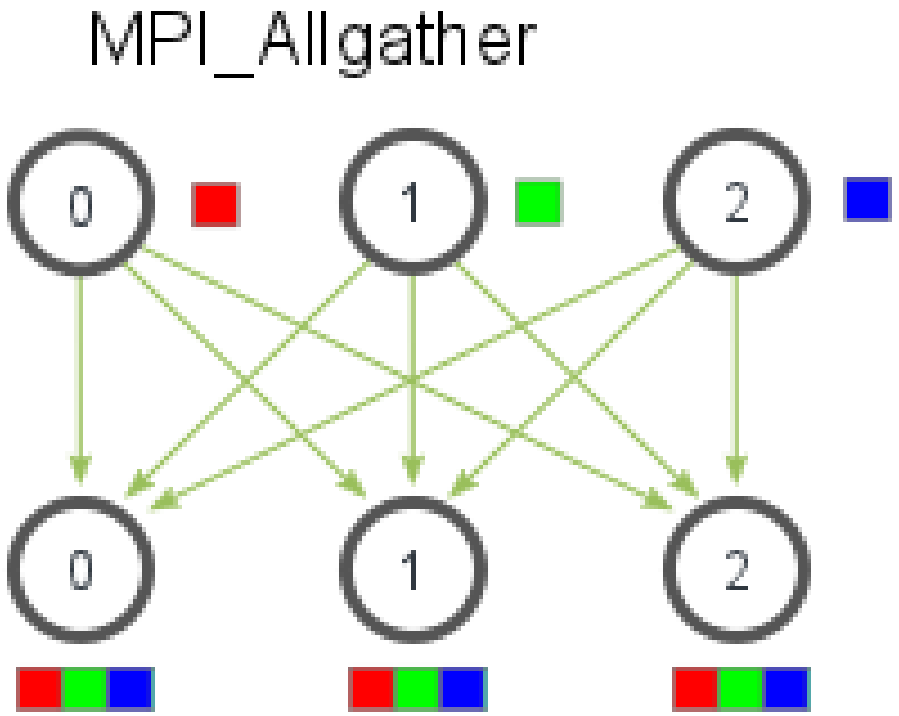
e.g., one rank is preparing files that all the other ranks will read. The other ranks will have to wait until those files are ready

```
if(rank == 0){
   prepare files
   MPI_Barrier(MPI_COMM_WORLD);
}else{
   MPI_Barrier(MPI_COMM_WORLD);
   read files
}
```

Questions?

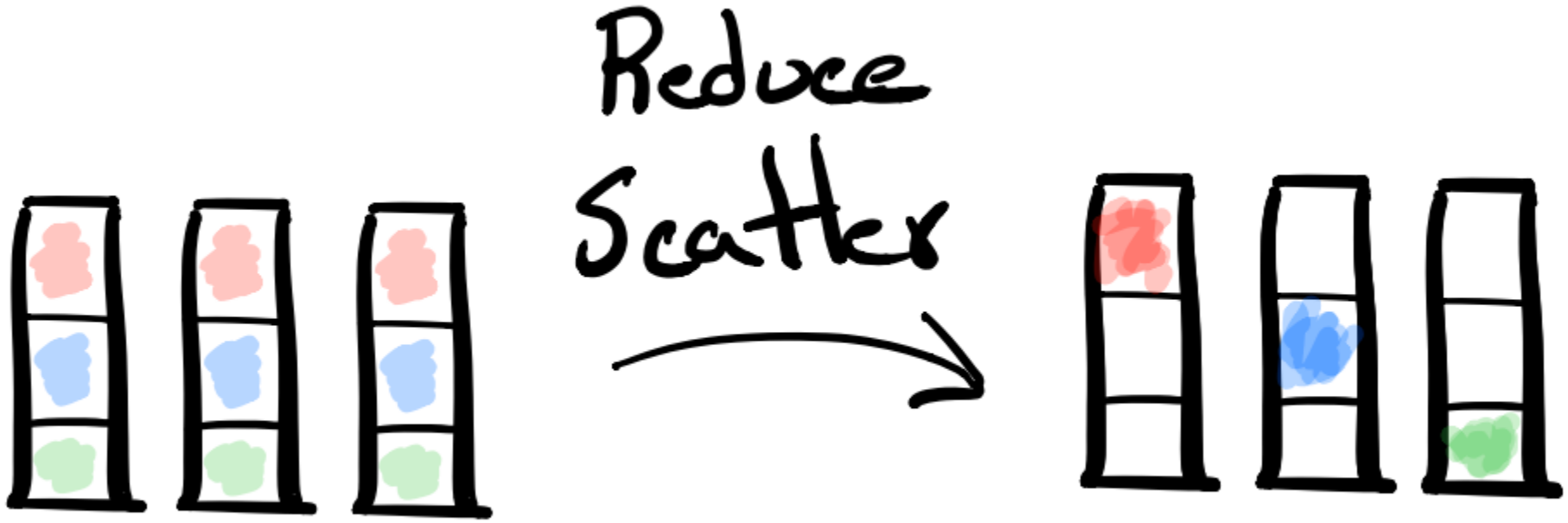
Allgather

- Conceptually, it is like a Gather + Broadcast
- In practice, it might be implemented in a more efficient way

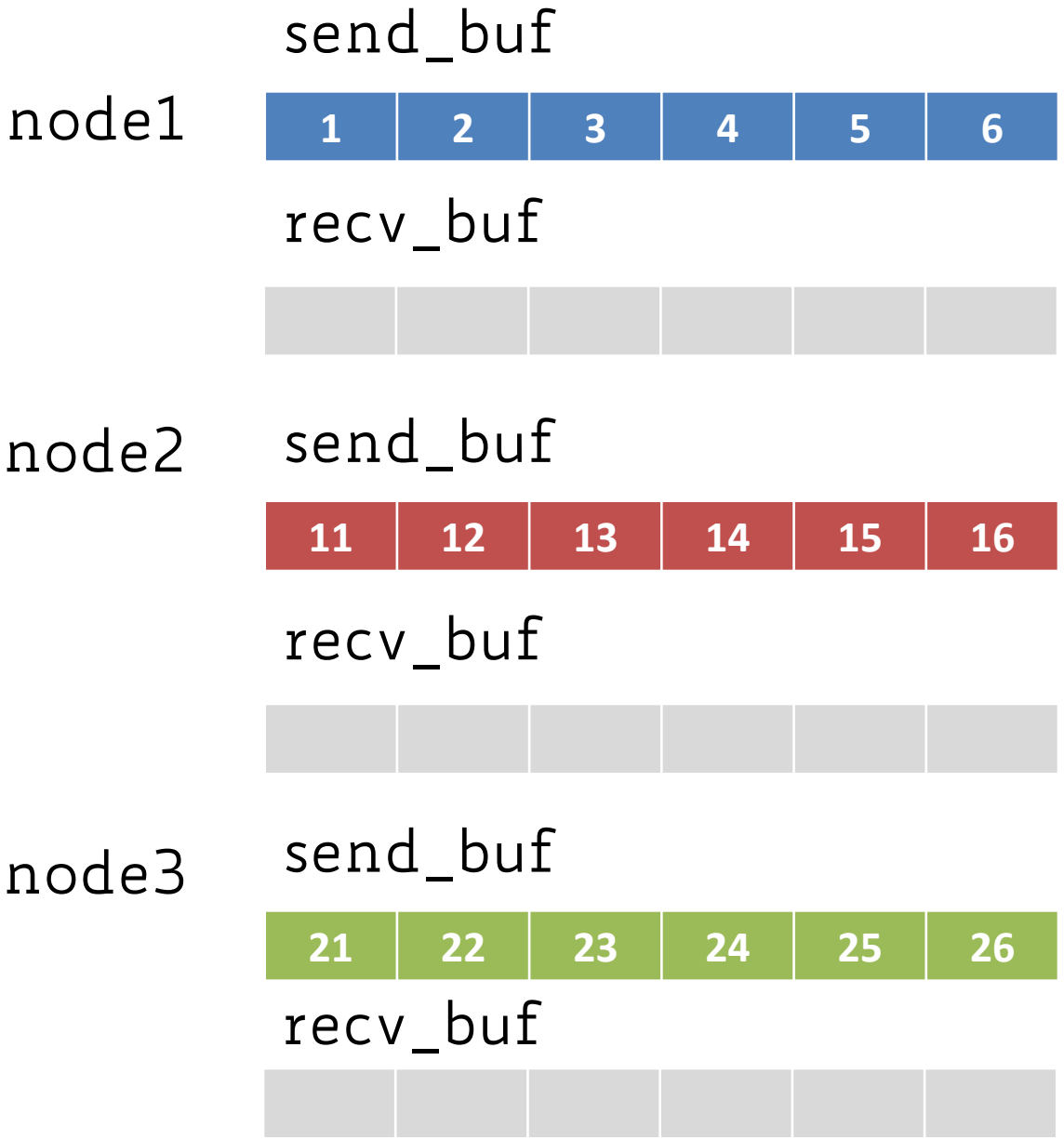


Reduce-Scatter

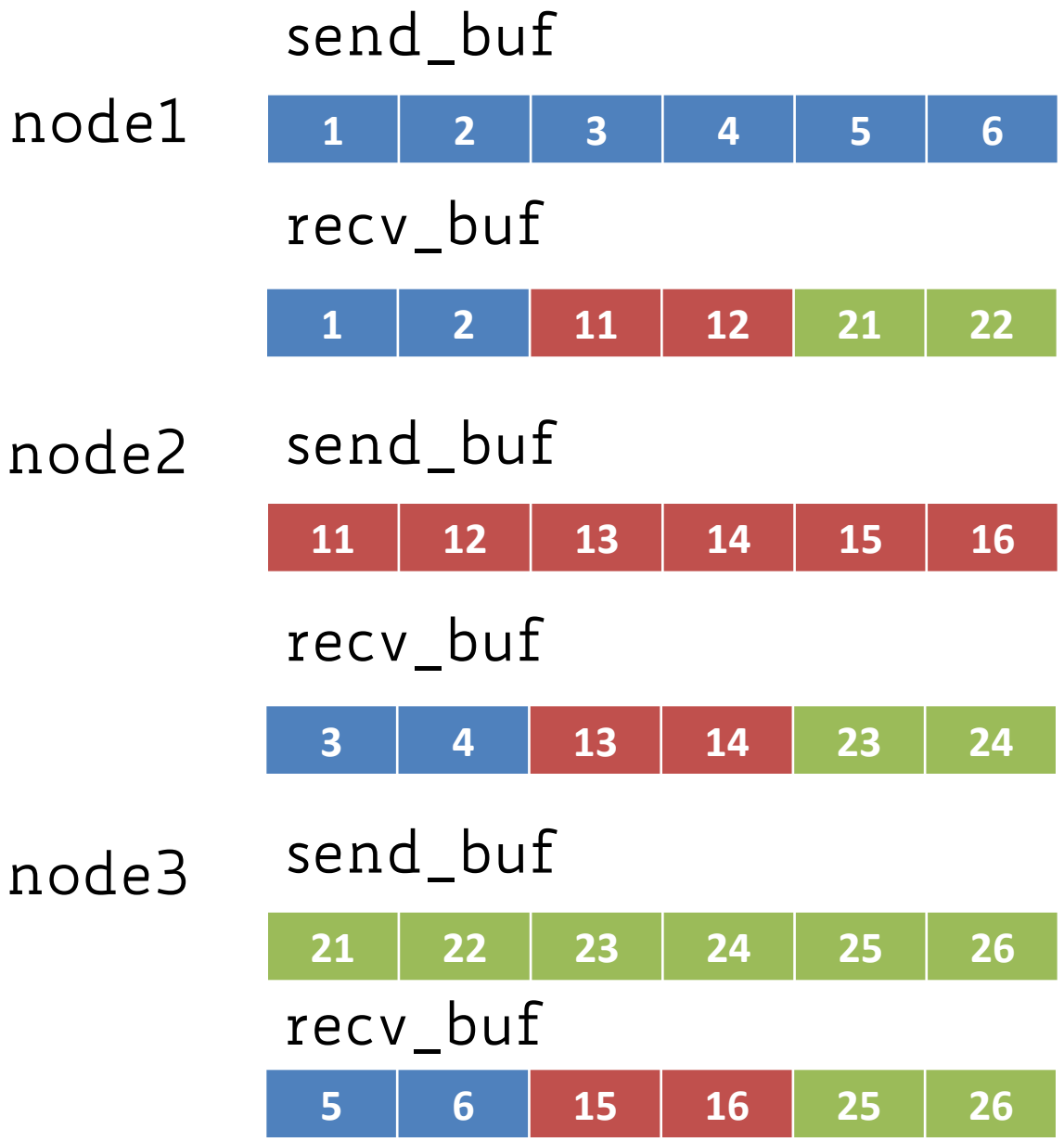
- Each rank gets the sum of just a part of the vector



MPI_Alltoall



MPI_Alltoall



MPI_Alltoall