MPI lecture and labs 2

Victor Eijkhout

2016

Collectives

Table of Contents

Collectives

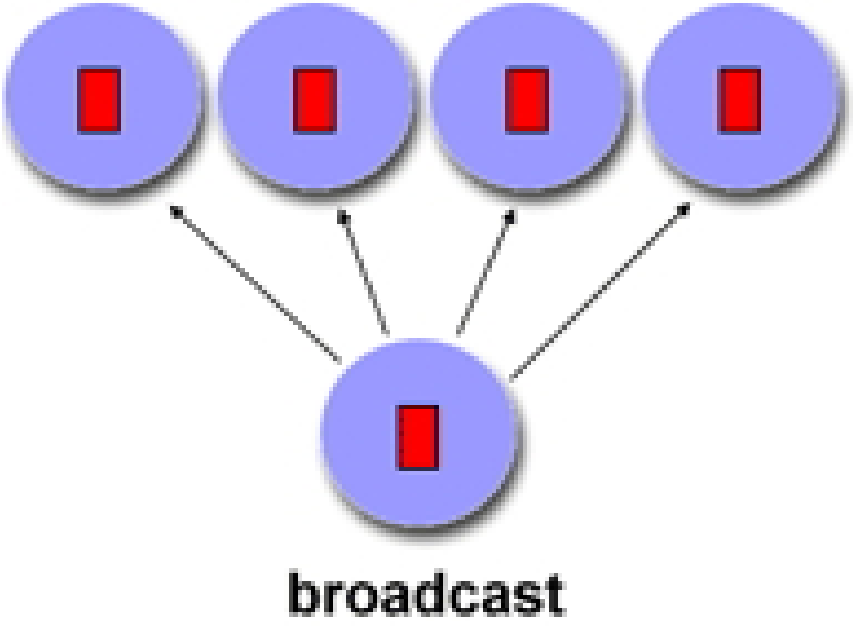
Gathering and spreading information:

- Every process has data, you want to bring it together;
- One process has data, you want to spread it around.

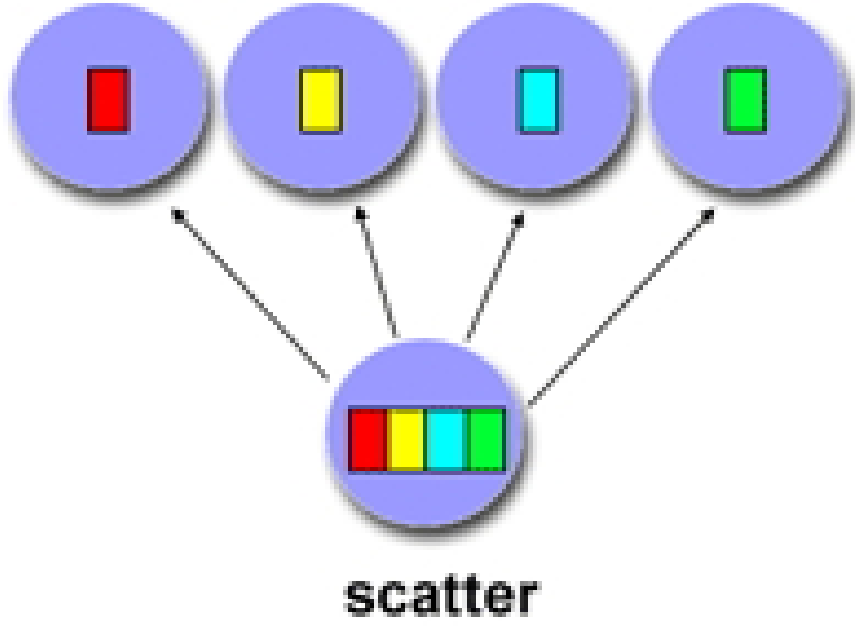
Root process: the one doing the collecting or disseminating.

Basic cases:

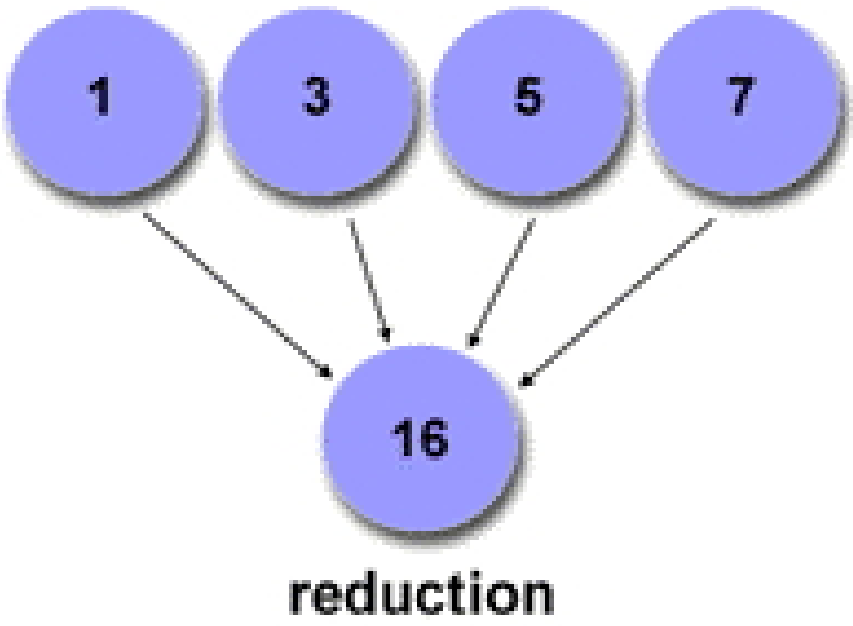
- Collect data: gather.
- Collect data and compute some overall value (sum, max): reduction.
- Send the same data to everyone: broadcast.
- Send individual data to each process: scatter.



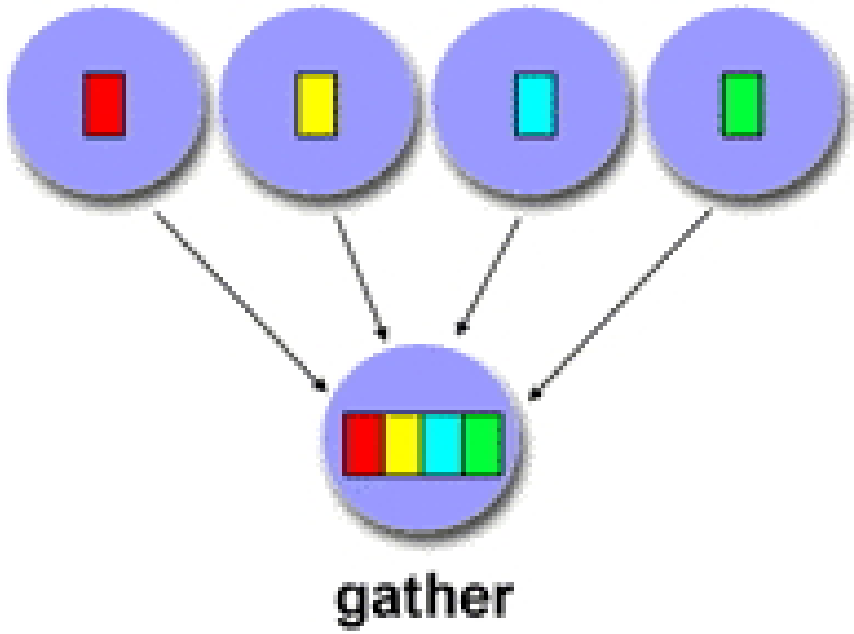
broadcast



scatter



reduction



gather

Exercise 1

How would you realize the following scenarios with MPI collectives?

- Let each process compute a random number. You want to print the maximum of these numbers to your screen.
- Each process computes a random number again. Now you want to scale these numbers by their maximum.
- Let each process compute a random number. You want to print on what processor the maximum value is computed.

More collectives

- Instead of a root, collect to all: `MPI_All...`
- Scatter individual data, but also individual size: `MPI_Scatterv`
- Everyone broadcasts: all-to-all
- Scan: like a reduction, but with partial results

. . . and more

Table of Contents

Broadcast

```
int MPI_Bcast(
    void *buffer, int count, MPI_Datatype datatype,
    int root, MPI_Comm comm )
```

- `root` is the rank of the process doing the broadcast
- Each process allocates buffer space;
  root explicitly fills in values,
  all others receive values through broadcast call.
- Datatype is `MPI_FLOAT`, `MPI_INT` et cetera, different between C/Fortran.
- `comm` is usually `MPI_COMM_WORLD`

Buffers in C

General principle: buffer argument is address in memory of the data.

- Buffer is void pointer:
- write `&x` or `(void*)&x` for scalar
- write `x` or `(void*)x` for array

Buffers in Fortran

General principle: buffer argument is address in memory of the data.

- Fortran always passes by reference:
- write `x` for scalar
- write `x` for array

Buffers in Python

For many routines there are two variants:

- lowercase: can send Python objects;
  output is `return` result
  this uses `pickle`: slow.
- uppercase: communicates `numpy` objects;
  input and output are function argument.

Exercise 2

If you give a commandline argument to a program, that argument is available as a character string as part of the `argv,argc` pair that you typically use as the arguments to your main program. You can use the function `atoi` to convert such a string to integer.

Write a program where process 0 looks for an integer on the commandline, and broadcasts it to the other processes. Initialize the buffer on all processes, and let all processes print out the broadcast number, just to check that you solved the problem correctly.

Reduction

```
int MPI_Reduce
  (void *sendbuf, void *recvbuf,
   int count, MPI_Datatype datatype,
   MPI_Op op, int root, MPI_Comm comm)
```

- Compare buffers to bcast
- `recvbuf` is ignored on non-root processes
- `MPI_Op` is `MPI_SUM`, `MPI_MAX` et cetera.

Exercise 3

Write a program where each process computes a random number, and process 0 finds and prints the maximum generated value. Let each process print its value, just to check the correctness of your program.

Now let each process scale its value by this maximum.

Random numbers

C:

```
// Initialize the random number generator
srand((int)(mytid*(double)RAND_MAX/ntids));
// compute a random number
randomfraction = (rand() / (double)RAND_MAX);
```

Fortran:

```
  integer :: randsize
  integer,allocatable,dimension(:) :: randseed
  real :: random_value

  call random_seed(size=randsize)
  allocate(randseed(randsize))
  do i=1,randsize
     randseed(i) = 1023*mytid
  end do
```

Exercise 4

Create on each process an array of length 2 integers, and put the values $1, 2$ in it on each process. Do a sum reduction on that array. Can you predict what the result should be? Code it. Was your prediction right?

Gather/Scatter

```
int MPI_Gather(
  void *sendbuf, int sendcnt, MPI_Datatype sendtype,
  void *recvbuf, int recvcnt, MPI_Datatype recvtype,
  int root, MPI_Comm comm
);
int MPI_Scatter
  (void* sendbuf, int sendcount, MPI_Datatype sendtype,
   void* recvbuf, int recvcount, MPI_Datatype recvtype,
   int root, MPI_Comm comm)
```

- Compare buffers to ▸ reduce
- Scatter: the `sendcount` / Gather: the `recvcount`:
  this is not, as you might expect, the total length of the buffer; instead, it is the amount of data to/from each process.

Exercise 5

Let each process compute a random number. You want to print on what processor the maximum value is computed. What collective do you use? Write a short program.

Table of Contents

Scan

Scan or ‘parallel prefix’: reduction with partial results

- Useful for indexing operations:
- Each processor has an array of n_p elements;
- My first element has global number $\sum_{q<p} n_q$.

```
C:
int MPI_Scan(const void* sendbuf, void* recvbuf,
    int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
IN sendbuf: starting address of send buffer (choice)
OUT recvbuf: starting address of receive buffer (choice)
IN count: number of elements in input buffer (non-negative integer)
IN datatype: data type of elements of input buffer (handle)
IN op: operation (handle)
IN comm: communicator (handle)

Fortran:
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
res = Intracomm.scan( sendobj=None,recvobj=None,op=MPI.SUM)
```

All

- `MPI_Allreduce`: do a reduction, but leave the result everywhere.
- `MPI_AllGather`: gather, and leave the result everywhere.

```
C:
int MPI_Allreduce(const void* sendbuf,
  void* recvbuf, int count, MPI_Datatype datatype,
  MPI_Op op, MPI_Comm comm)

Semantics:
IN sendbuf: starting address of send buffer (choice)
OUT recvbuf: starting address of receive buffer (choice)
IN count: number of elements in send buffer (non-negative integer)
IN datatype: data type of elements of send buffer (handle)
IN op: operation (handle)
IN comm: communicator (handle)

Fortran:
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Exercise 6

How can you simulate an allreduce with routines you already know? What is the point of having this one routine?

V-type collectives

- Gather/scatter but with individual sizes
- Requires displacement in the gather/scatter buffer

```
C:
int MPI_Gatherv(
  const void* sendbuf, int sendcount, MPI_Datatype sendtype,
  void* recvbuf, const int recvcounts[], const int displs[],
  MPI_Datatype recvtype, int root, MPI_Comm comm)

Semantics:
IN sendbuf: starting address of send buffer (choice)
IN sendcount: number of elements in send buffer (non-negative integer)
IN sendtype: data type of send buffer elements (handle)
OUT recvbuf: address of receive buffer (choice, significant only at ro
IN recvcounts: non-negative integer array (of length group size) conta
IN displs: integer array (of length group size). Entry i specifies the
IN recvtype: data type of recv buffer elements (significant only at ro
IN root: rank of receiving process (integer)
IN comm: communicator (handle)

Fortran:
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
```

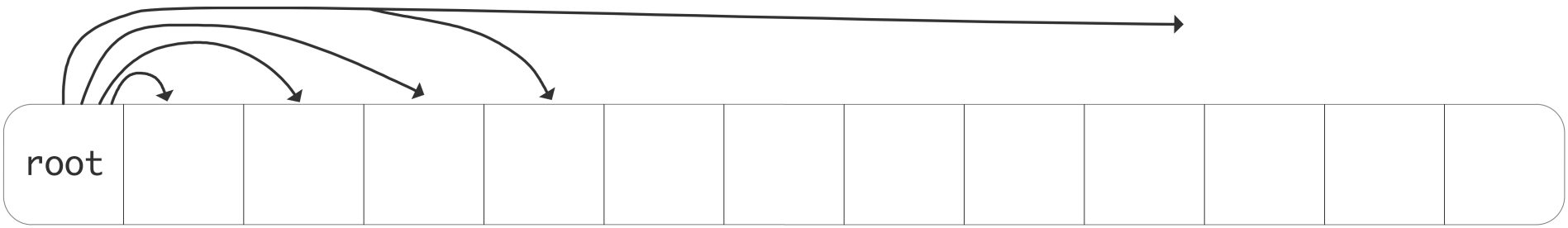
All-to-all

- Every process does a scatter;
- each individual data
- Very rarely needed.

Barrier

- Synchronize processors:
- each process waits at the barrier until all processes have reached the barrier
- **This routine is almost never needed**
- One conceivable use: timing

Naive realization of collectives

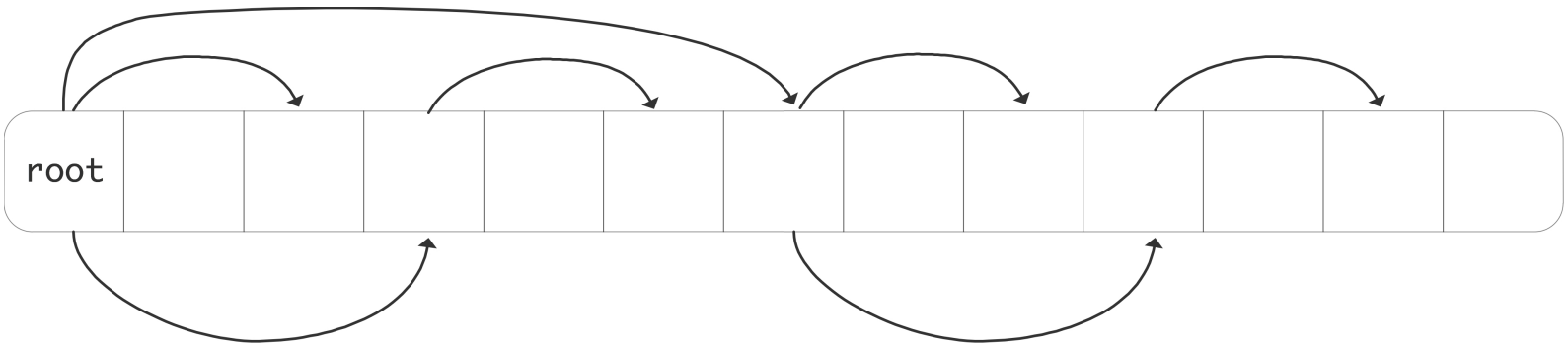


Message time is modeled as

$$\alpha + \beta n$$

Time for collective? Can you improve on that?

Better implementation of collective



What is the running time now?