MPI Collectives

Computational Science II (CAAM 520)

Christopher Thiele

Rice University, Spring 2021

What are collectives?

So far we have introduced MPI communicators as well as `MPI_Send()`, `MPI_Recv()`, and `MPI_Sendrecv()` to pass messages between ***individual*** ranks within a communicator.

Collectives are operations that are performed by ***all*** ranks in a communicator.

Collectives provide convenient and efficient implementations of common communication patterns.

Synchronizing ranks with `MPI_Barrier`

We are already familiar with one example of a collective operation: `MPI_Barrier`.

```
int MPI_Barrier(MPI_Comm comm)
```

→ Each rank waits for ***all*** other ranks to reach the barrier.

Broadcasting data with MPI_Bcast

To send data from one rank to all other ranks in the communicator, use MPI_Bcast:

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

→ The rank specified by the root argument broadcasts to all other ranks.

Broadcasting data with `MPI_Bcast`

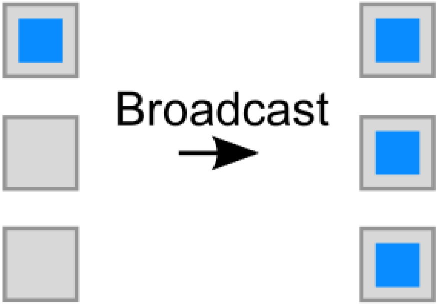

Image source: https://www.wikiwand.com/de/Message_Passing_Interface

Scattering data with MPI_Scatter

MPI_Scatter works much like MPI_Bcast, but it sends different data to each rank.

```
int MPI_Scatter(const void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                int root,
                MPI_Comm comm)
```

$\longrightarrow$ MPI_Scatter sends sendcount items to ***each*** rank.

Scattering data with `MPI_Scatter`

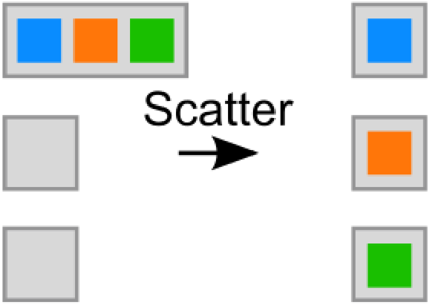

Image source: https://www.wikiwand.com/de/Message_Passing_Interface

Scattering data with MPI_Scatter

If we need to send a different amount of data to each rank, we can use MPI_Scatterv.

```
int MPI_Scatterv(const void *sendbuf,
                 const int *sendcounts,
                 const int *displs,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm)
```

→ Sends sendcounts[r] items to rank r starting with sendbuf[displs[r]].

Gathering data with MPI_Gather

The inverse operation to MPI_Scatter is MPI_Gather. It collects data from all ranks on the root rank.

```
int MPI_Gather(const void *sendbuf,
               int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf,
               int recvcount,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm)
```

→ MPI_Gather receives recvcount items from ***each*** rank.

Gathering data with `MPI_Gather`

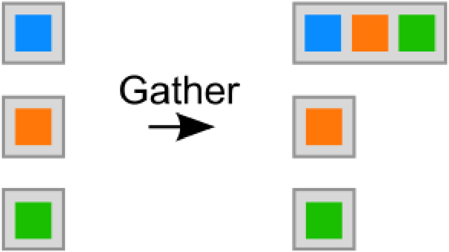

Image source: https://www.wikiwand.com/de/Message_Passing_Interface

Gathering data with `MPI_Gather`

To collect a different amount of data from each rank, use `MPI_Gatherv`.

```
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)
```

→ The root rank receives `recvcounts[r]` from rank `r` and stores them at `recvbuf + displs[r]`.

Gathering data with `MPI_Allgather`

`MPI_Allgather` works just like `MPI_Gather`, but ***all*** ranks gather all data.

```
int MPI_Allgather(const void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm comm)
```

→ Again, there is a more general `MPI_Allgatherv` function, too.

Gathering data with MPI_Allgather

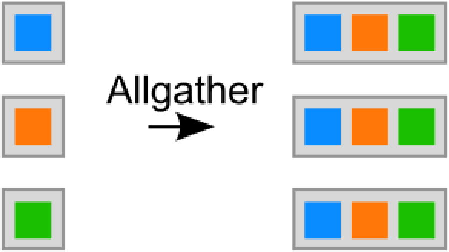

Image source: https://www.wikiwand.com/de/Message_Passing_Interface

Redistributing data with `MPI_Alltoall`

`MPI_Alltoall` resembles `MPI_Allgather`, but each rank now gathers different data. The action of `MPI_Alltoall` is best explained in a picture (next slide).

```
int MPI_Alltoall(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 MPI_Comm comm)
```

Redistributing data with `MPI_Alltoall`

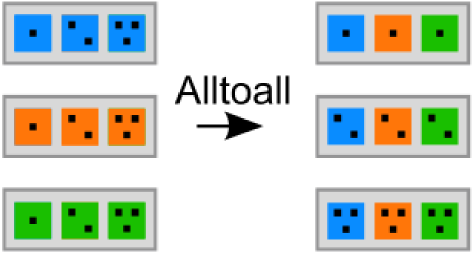

Image source: https://www.wikiwand.com/de/Message_Passing_Interface

Redistributing data with MPI_Alltoall

Again, there is a function MPI_Alltoallv that can be used if each rank receives a different amounts of data.

```
int MPI_Alltoallv(const void *sendbuf,
                  const int *sendcounts,
                  const int *sdispls,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  const int *recvcounts,
                  const int *rdispls,
                  MPI_Datatype recvtype,
                  MPI_Comm comm)
```

Reductions

Some collective operations perform computations in addition to message passing.

`MPI_Reduce` works much like `MPI_Gather`, but the gathered data is combined using an operator.

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

→ Compare this to OpenMP's `reduction` clause.

Reductions

Possible operations include

```
typedef enum {
  MPI_MAX,
  MPI_MIN,
  MPI_SUM,
  MPI_PROD,
  MPI_REPLACE,
  // etc.
} MPI_Op;
```

→ Users can define their own operations, too.

Reductions

`MPI_Allreduce` works like `MPI_Reduce`, but now ***every*** rank performs a reduction.

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

Reductions

Example: Compute the Euclidean norm of a ***distributed*** vector.

```
double norm2(const double *x, int n_local)
{
  double sum, sum_local = 0.0;

  for (int i = 0; i < n_local; i++) {
    sum_local += x[i]*x[i];
  }
  MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE,
                MPI_SUM, MPI_COMM_WORLD);

  return sqrt(sum);
}
```

Collectives and deadlocks

Collective operations ***must*** be called by all ranks in the communicator. Otherwise, the collective operation results in a deadlock!

This can be trickier than it seems:

```
compute_vector(x, n_local);

if (rank == 0) {
  // Deadlock!
  printf("||x|| = %e\n", norm2(x, n_local));
}
```

Collectives and performance

Question: Is the code below a good implementation of `MPI_Bcast`?

```
if (rank == root) {
  for (int r = 0; r < size; r++) {
    if (r == rank) continue;
    MPI_Send(buffer, count, datatype,
             r, 999, comm);
  }
}
else {
  MPI_Recv(buffer, count, datatype,
           root, 999, comm, MPI_STATUS_IGNORE);
}
// ...
```

Collectives and performance

Answer: No! We can accelerate the broadcast using a tree structure.

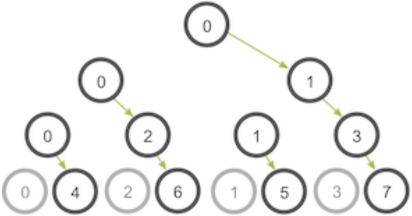

Image source: https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/