futurelearn.com

Collective communication: many to many

4-5 分钟

- 

In this article we discuss how to use collective communication to exchange data between all the processes in a communicator.

Collective communication routines in MPI include also routines for global communication between all the processes.

Global collective communication is extremely costly in terms of performance, so if possible one should avoid using them. Nevertheless, in some situations they are exactly the correct approach to implement an parallel algorithm.

Let us look next how to use collective communication to exchange data between all the processes in a communicator,

i.e. how to move data from *many to many*.

Allreduce

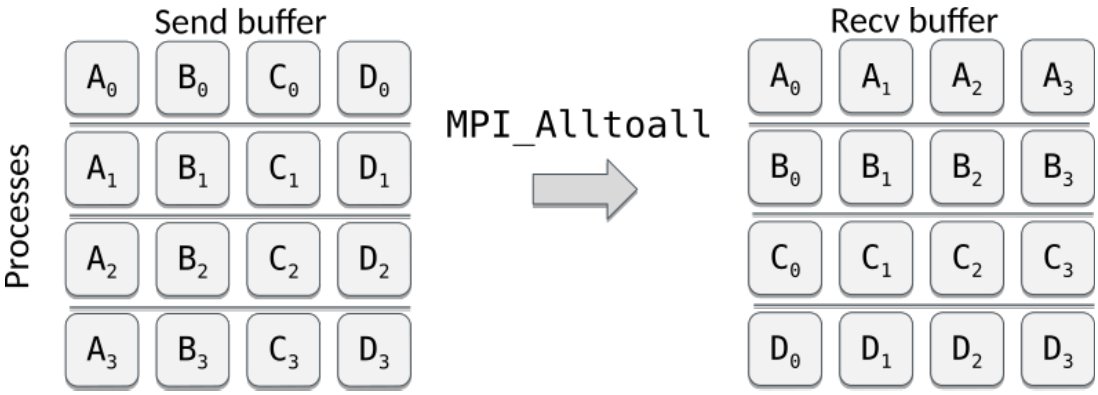
Allreduce is in principle just a Reduce operation followed by Broadcast, so that in the end of the operation all processes have the results of reduction. The MPI library can, however, implement the operation more efficiently than when using two successive calls.

Only difference in the function call compared to Reduce is that there is no root argument, as seen in the following example:

```
from mpi4py import MPI
from numpy import arange, empty
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

data = arange(10 * size, dtype=float) * (rank
+ 1)
buffer = empty(size * 10, float)

n = comm.allreduce(rank, op=MPI.SUM) #
returns the value


comm.Allreduce(data, buffer, op=MPI.SUM) #
in-place modification
```

Alltoall

In Alltoall operation each process sends and receives to/from each other, and
can be considered as combination of Scatter and Gather. The operation can be
also viewed as “transpose”.



An example of Alltoall both with a Python list and a NumPy array:

```
from mpi4py import MPI
from numpy import arange, empty, zeros_like
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

py_data = range(size)
data = arange(size**2, dtype=float)

new_data = comm.alltoall(py_data) # returns
the value


buffer = zeros_like(data) # prepare a receive
buffer
comm.Alltoall(data, buffer) # in-place
```

```
modification
```

Some common mistakes to avoid when using collectives include:

1. Using a collective operation within one branch of an if-else test based on
   the rank of the process
   ```
   if rank == 0:
    comm.bcast(...)
   ```
   All processes in a communicator must call a collective routine!
2. Assuming that all processes making a collective call would complete at
   the same time.Even a collective operation such a barrier only ensures that a process
   holds until everyone reaches the call. With data movement call (scatter,
   bcast etc.) even this may not be true, since MPI only guarantees that the
   process will proceed only when it is *safe* to do so. MPI implementations
   can (and do!) use communication caches that may allow some of the processes
   to continue from a collective call *even before communication happens*.

3. Using the input buffer also as an output buffer.

   ```
   comm.Scatter(a, a, MPI.SUM)
   ```

   Always use different memory locations (arrays) for input and output!

This article is from the free online

Python in High Performance Computing

Created by

Our purpose is to transform access to education.

We offer a diverse selection of courses from leading universities and cultural institutions from around the world. These are delivered one step at a time, and are accessible on mobile, tablet and desktop, so you can fit learning around your life.

We believe learning should be an enjoyable, social experience, so our courses offer the opportunity to discuss what you're learning with others as you go, helping you make fresh discoveries and form new ideas.
You can unlock new opportunities with unlimited access to hundreds of online short courses for a year by subscribing to our Unlimited package. Build your knowledge with top universities and organisations.

Learn more about how FutureLearn is transforming access to education

Hi there! We hope you're enjoying our article: Collective communication: many to many

This article is part of our course: Python in High Performance Computing