ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

Point to point communication

- to send a message

```
int MPI_Send(
      void* buffer, int count, MPI_Datatype datatype,
      int destination, int tag, MPI_Comm communicator
      );
```

- to receive a message

```
int MPI_Recv(
      void* buffer, int count, MPI_Datatype datatype,
      int source, int tag, MPI_Comm communicator
      );
```

- the `tag` enables choosing the order you may receive pending messages
- but for a given (`source`,`tag`,`communicator`) messages are received in the order they were sent
- receiving via wildcards: `MPI_ANY_SOURCE` and `MPI_ANY_TAG`
- in *standard* communication mode, sending and receiving messages are *blocking*, so the function does not return until you can safely access the `buffer`
  - to read, free, etc.

Communication modes

- in standard mode, the specification does not explicitly mention buffering strategy
  - buffering messages would remove some of the access constraints but it requires time and storage for the multiple copies
  - portability across implementations implies conservative assumptions about the order of initiation of sends and receives to avoid deadlock
- in *ready* mode, you must post a receive before the matching send can be initiated
  - `MPI_Rsend`, `MPI_Rrecv`
- in *buffered* mode, sends can be initiated, and may complete, regardless of when the matching receive is initiate
  - `MPI_Bsend`, `MPI_Brecv`
- in *synchronous* mode, sends can be initiated regardless of whether the matching receive has been initiated, but the send will not return until the message has been received
  - `MPI_Ssend`, `MPI_Srecv`

Asynchronous communication

- there are non-blocking versions of all these

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

  - faster, but you must take care to not access the message buffers until the messages have been delivered
  - more details later in the course, as needed
- for sends
  - standard mode: `MPI_Isend`
  - ready mode: `MPI_Irsend`
  - buffered mode: `MPI_Ibsend`
  - synchronous mode: `MPI_Issend`
- only one call for receives: `MPI_Irecv`
- extra `request` argument to check for completion of the request
  - `MPI_Test`, `MPI_Wait` and their relatives

Creating communicators and groups

- communicators and groups are intertwined
  - you cannot create a group without a communicator
  - you cannot create a communicator without a group
- the cycle is broken by MPI_COMM_WORLD

```c
#include <mpi.h>

int main(int argc, char* argv[]) {
   /* declare a communicator and a couple of groups */
   MPI_Comm workers;
   MPI_Group world_grp, workers_grp;

   /* initialize MPI; for brevity all status checks are omitted */
   MPI_Init(&argc, &argv);

   /* get the world communicator to build its group */
   MPI_Comm_group(MPI_COMM_WORLD, &world_grp);

   /* build another group by excluding a process */
   MPI_Group_excl(world_grp, 1, 0, &workers_grp);

   /* now build a communicator out of the processes in workers_grp */
   MPI_Comm_create(MPI_COMM_WORLD, worker_grp, &workers);

   /* etc.... */

   /* shut down MPI */
   MPI_Finalize();

   return 0;
}
```

Manipulating communicators and groups

- releasing resources

```
int MPI_Group_free(MPI_Group* group);
int MPI_Comm_free(MPI_Comm* communicator);
int MPI_Comm_disconnect(MPI_Comm* communicator);
```

- you can make a new group by adding or removing processes from an existing one

```
int MPI_Group_incl(
   MPI_Group grp, int n, int* ranks, MPI_Group* new_group);
int MPI_Group_excl(
   MPI_Group grp, int n, int* ranks, MPI_Group* new_group);
```

- or by using set operations

```
int MPI_Group_union(
   MPI_Group grp1, MPI_Group grp2, MPI_Group* new_group);
int MPI_Group_intersection(
   MPI_Group grp1, MPI_Group grp2, MPI_Group* new_group);
int MPI_Group_difference(
   MPI_Group grp1, MPI_Group grp2, MPI_Group* new_group);
```

Timing

- the function

```c
double MPI_Wtime();
```

  returns the time in seconds from some arbitrary time in the past
  - guaranteed not to change only for the duration of the process
- you can compute the elapsed time for any program segment by making calls at the beginning and the end and computing the difference
- no guarantees about synchronized clocks among different processes
- you can compute the clock resolution by using

```c
double MPI_Wtick();
```

Other collective operations

- MPI_Scan computes partial reductions: the p^{th} process receives the result from processes 0 through $p - 1$

```
int MPI_Scan(
      void* send_buffer, void* recv_buffer,
      int count, MPI_Datatype datatype, MPI_Op operation,
      MPI_Comm communicator
      );
```

- MPI_Reduce collects the result at only the given process root

```
int MPI_Reduce(
      void* send_buffer, void* recv_buffer,
      int count, MPI_Datatype datatype, MPI_Op operation,
      int root, MPI_Comm communicator
      );
```

- synchronization is also a global operation:

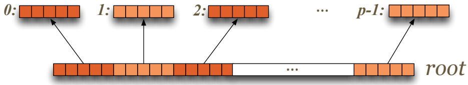
```
int MPI_Barrier(MPI_Comm communicator);
```

participating processes block at a barrier until they have all reached it

Scatter

- `MPI_Scatter` sends data from `root` to all processes

```
int MPI_Scatter(
        void* send_buffer, int send_count, MPI_Datatype send_datatype,
        void* recv_buffer, int recv_count, MPI_Datatype recv_datatype,
        int root, MPI_Comm communicator
        );
```

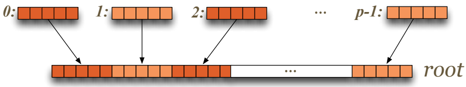


- it is as if the data in `send_buffer` were split in p segments, and the i^{th} process receives the i^{th} segment
- the `send_xxx` arguments are only meaningful for `root`; they are ignored for other processes
- the arguments `root` and `communicator` must be passed identical values by all processes

Gather

- the converse is `MPI_Gather` with `root` receiving data from all processes

```
int MPI_Gather(
      void* send_buffer, int send_count, MPI_Datatype send_datatype,
      void* recv_buffer, int recv_count, MPI_Datatype recv_datatype,
      int root, MPI_Comm communicator
      );
```



- it is as if *p* messages, one from each processes, were concatenated in rank order and placed at `recv_buffer`
- the `recv_xxx` arguments are only meaningful for `root`; they are ignored for other processes
- the arguments `root` and `communicator` must be passed identical values by all processes

Broadcasting operations

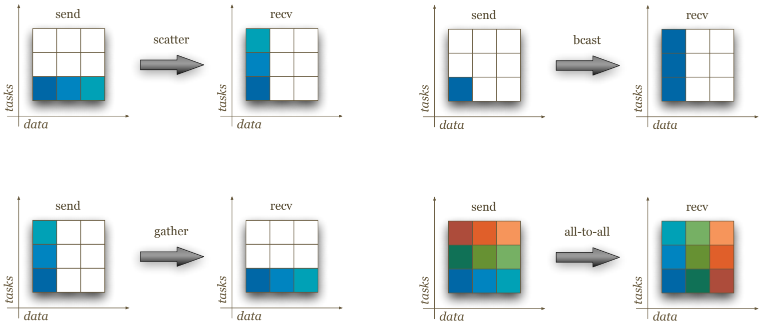
- `MPI_Alltoall` sends data from all processes to all processes in a global scatter/gather

```
int MPI_Alltoall(
      void* send_buffer, int send_count, MPI_Datatype send_datatype,
      void* recv_buffer, int recv_count, MPI_Datatype recv_datatype,
      MPI_Comm communicator
      );
```

- use `MPI_Bcast` to send the contents of a buffer from `root` to all processes in a communicator

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

Data movement patterns for the collective operations



Virtual topologies

-