7. MPI: Asynchronous and collective communication

Part 1

Communication modes

1 – Communication modes

- Standard communication: MPI chooses buffering strategy
  - Example: MPI_Send –- MPI_Recv pair
  - Non-local completion: Send depends on remote receive
- Alternative: User chooses communication mode
  - Buffered communication
  - Synchronous communication
  - Ready (and unbuffered) communication
- Advantage: performs better, when communication pattern is known in advance
  - One memory copy less, when buffered (DMA possible)
  - No synchronization messages, when immediate

1 – Buffered communication

- MPI will buffer the data, if matching receive has not been issued
  - Therefore: Local completion semantics
- Can fail, without enough buffer space

```
MPI_Bsend(void* buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm)
```

- User must provide buffer space
  - `MPI_Buffer_attach(void *buffer, int size)`
  - `MPI_Buffer_detach(void *buffer, int *size)`
- Usage
  - Decouple sender and receiver
  - Can increase latency
  - Allows to overlap communication and computation

1 – More communication modes

- Synchronous communication
  - MPI will wait until matching receive has been issued
    - » Therefore: non-local completion semantics
  - Data will be sent directly to receiver
  - MPI function: MPI_Ssend
- Ready communication
  - Even more optimized: Send can be called only, if receive is issued already
  - Data will be sent immediately to receiver
  - If corresponding receive is not issued, call will fail
  - MPI-Function: MPI_Rsend

Part 2

Asynchronous communication

2 – Non-blocking communication

- Issue only a communication request, but do not wait for completion
- Meanwhile: computation can continue
- After that, we can
  - check, if the communication has terminated
  - wait for the end of the communication
- Advantage: Overlapping of communication and computation

2 – Non-blocking communication

- `MPI_Isend` start a non-blocking send, and `MPI_Irecv` start a non-blocking receive
- `MPI_Test` checks the status of the communication request
- `MPI_Wait` waits for the completion of the communication
- `MPI_Isend` and `MPI_Irecv` return a communication request of type `MPI_Request`
  - This serves as a handle for later `MPI_Test` and `MPI_Wait` calls

2 - Nicht-blockierende Kommunikation

```
MPI_Isend(void *buf, int count, MPI_Datatype datatype,
       int dest, int tag, MPI_Comm comm,
       MPI_Request *request);

MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
   int source, int tag, MPI_Comm comm,
    MPI_Request *request);

MPI_Test(MPI_Request *request, int *flag,
    MPI_Status *status);

MPI_Wait(MPI_Request *request, MPI_Status *status);
```


2 – Non-blocking communication

```
/* Asynchronous send */

MPI_Request request;
MPI_Status st;
char message[20];

MPI_Isend(message, 20, MPI_CHAR, 0, 0,
   MPI_COMM_WORLD, &request);
/* do your job ... */
MPI_Wait(&request, &st);
```


2 – Non-blocking communication

- Non-blocking communication can be done also
  - buffered: MPI_IBsend
  - synchronous: MPI_ISsend
  - ready: MPI_IRsend

Part 3

Collective communication

3 - Synchronization

- Exchange of messages can be used for (implicit) synchronization
- Frequently: want to ensure, that all processes terminated some task
- This kind of synchronization is called a barrier
  - Every process calls MPI_Barrier
  - MPI_Barrier terminates only, if all processes reached it

3 - Synchronization

```
MPI_Barrier(MPI_Comm comm);
```

```
MPI_Comm comm;
...
/* Synchronization */
MPI_Barrier(comm);
```


3 – Collective communication

- Overview
  - Broadcast (1-n-communication)
  - Scatter and gather
  - Reductions and scans
- Communication patters
  - From or to a single process
  - Between all processes

3 - Collective communication

- In collective communication calls, all processes exchange messages in a single call
- All processes usually execute the call at the *same time* with the *same arguments*
  - Collective communication calls synchronize process execution
- Advantage: Optimization of communication
  - Mapping to known communication architecture
  - Improved communiciation strategies

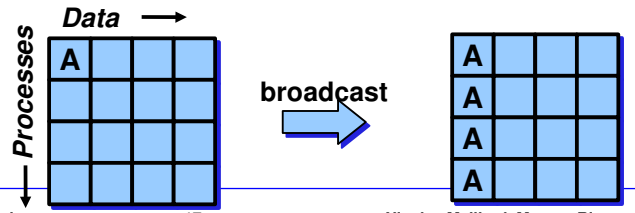
3 - Collective communication

- MPI_Bcast sends a message from one process to all others

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

- Example

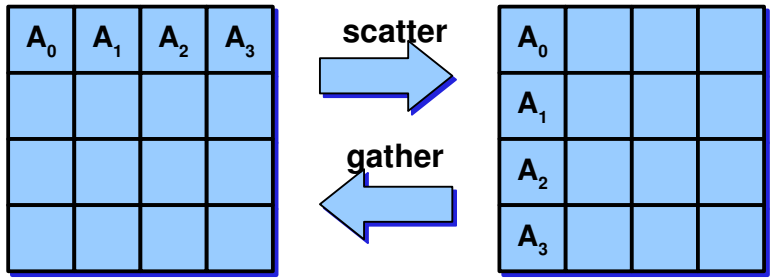
```
char message[20];
MPI_BCAST(message, 20, MPI_CHAR, 0, MPI_COMM_WORLD);
```



3 - Collective communication

- MPI_Scatter distributes data from one process to all others
  - Similar to MPI_Bcast, but each process receives a different part of the data
- The inverse operation is MPI_Gather
- All calls are executed simultaneously
- Both operations have so-called vector variants, which allow to scatter or gather messages of different length

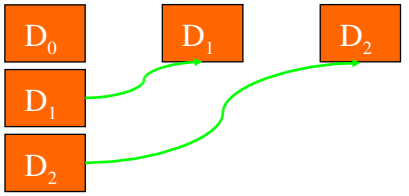
3 - Collective communication



3 – Distributing data

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

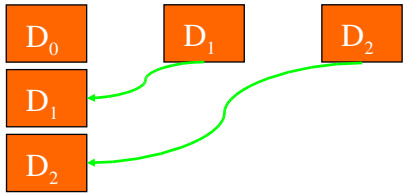
```
MPI_Scatter(globaldata, gsize, MPI_CHAR,
            localdata, lsize, MPI_CHAR, 0);
```



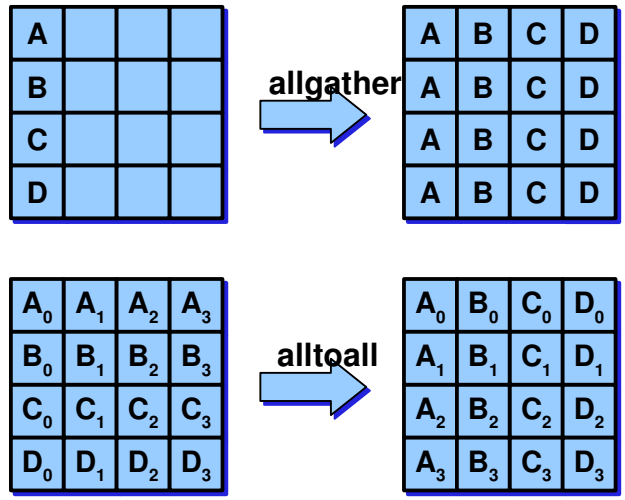
3 – Gathering data

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

```
MPI_Gather(localdata, lsize, MPI_CHAR,
           globaldata, gsize, MPI_CHAR, 0);
```



3 – More collective communication



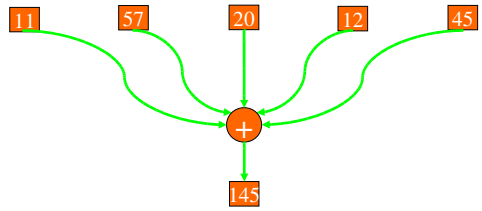
3 – Data reductions

- A reduction combines distributed data by some operation and returns the result to one process
- The corresponding collective call is `MPI_Reduce`.
- MPI provides a couple of predefined operations
  `MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD`
- The user can define other operations
- The `variant MPI_Allreduce` return the result to *all* processes

3 – Data reductions

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

```
MPI_Reduce(number, sum, 2, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(number, sum, 2, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



3 – Data reductions

- `MPI_Op_create` defines new reduction operations
- `The type MPI_Op` represents them
- `MPI_OP_free` deallocates the operator
- A user-defined operator has to be of type `MPI_User_function`

3 – Data reductions

```
MPI_Op_create(MPI_User_function *function,
              int commute, MPI_Op *op);

MPI_Op_free(MPI_Op *op);

typedef
  void MPI_User_function(void *invec,
                         void *inoutvec,
                         int *len,
                         MPI_Datatype *datatype);
```


4 – MPI 2

- We have seen (most of) MPI 1.2
- MPI 2 adds
  - Parallel file I/O
  - One-sided communication
  - Dynamic process creation and management
- one some other details
  - portable startup with mpiexec
  - generalized requests
  - extended collective communication calls

4 – MPI 2: Parallel file I/O

- Frequently, the I/O performance is a bottleneck
  - Think of implementing checkpointing when running 256 machines with of 1 GB data each
- Recent parallel computers provide parallel I/O systems
- Doing I/O in parallel with system calls is tedious
  - Some problem as in message passing
- MPI 2 parallel file I/O fills this gap
  - Portable way to do (collective) file I/O

4 – MPI-2: One-sided communication

- Provides a remote shared memory model
  - Each process can expose a memory window
  - The other process can read and write this window
  - The host process is not involved in the data transfer
- Advantages
  - Better for dynamic communication patterns
  - Can be faster, if architecture supports it

4 – MPI-2: process creation and managment

- MPI 1.2 is *static*: Number of processes is fixed at startup
- Process creation and management overcomes this
  - A parallel application can spawn new processes at runtime
  - The is support for client/server communication patterns
  - In particular: two independent MPI applications can connect to each other