THE UNIVERSITY OF CHICAGO | Research Computing Center

Distributed Programming with MPI in C

Igor Yakushin
ivy2@uchicago.edu

June 21, 2018

Introduction

- **M**essage **P**assing **I**nterface
- Standard for distributed computing implemented as a library
- MPI-1 (1994)
- Current standard: MPI-3.1 (2015)
- About 250 routines
- Available for C (native), Fortran, Python (mpi4py), ...
- Some popular implementations:
  - mpich
  - mvapich
  - Intel MPI - we use in this tutorial
  - openmpi
- MPI runs the same program on different nodes/cores. Each process is assigned a **rank**. Using if statement, a program can do different things for different ranks, exchange messages between different ranks
- In general (with one exception), there is no shared memory like in multithreading.

When to use MPI?

- Use MPI when your program requires more computing power or more memory than a single node can provide and different parts of the program need to exchange data. For example:
  - Solving PDE on a big grid:
    - divide the grid into patches and distribute the patches across MPI ranks
    - make one time step on each patch independently
    - exchange boundaries via MPI messages
  - Working with matrixes that do not fit in memory of a single node
- You do not need MPI to run embarassingly parallel computations that can run on different nodes without exchanging data. For example:
  - Simulations that use different set of parameters can run independently on different nodes
  - Data analysis pipeline parallelized over data

Where MPI can be used?

- One can use MPI on a single host running on local cores (useful for learning)

  ```
  mpirun -np 2 <yourprogram>
  ```

- One can use MPI on a cluster without a scheduler provided that a user can ssh from any host to any host without password (using ssh key with empty passphrase)

  ```
  mpirun -np 2 -hostfile <file with a list of hosts>
               <yourprogram>
  ```

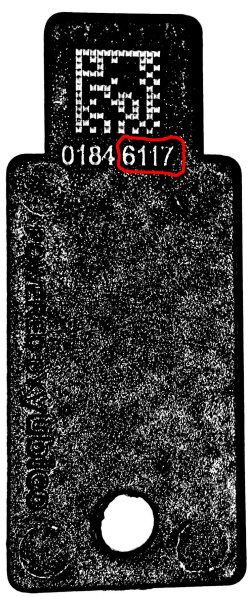
- One can use MPI on a cluster with a scheduler, for example Slurm. In that case one prepares a batch script requesting resources (number of nodes, cores per node, memory, time, etc.) and specifies the command to run:

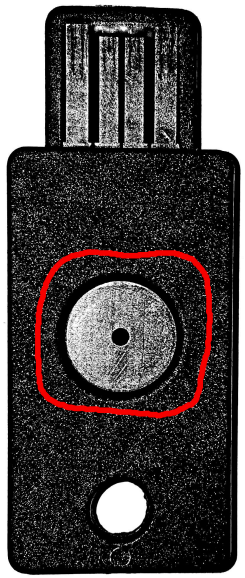
  ```
  mpirun <yourprogram>
  ```

  The scheduler takes care of distributing the computation among the nodes/cores given to the application

How to login to midway 2

- `ssh -Y <userid>@midway2.rcc.uchicago.edu` or use ThinLinc: `https://midway2.rcc.uchicago.edu`.
- If you do not have an account on midway, use yubikeys:






- Use last 4 digits of yubikey `<XXXX>` as part of userid:

  `ssh -Y rccguest<XXXX>@midway2.rcc.uchicago.edu`

  Push the button when asked for password

```
git clone https://git.rcc.uchicago.edu/ivy2/MPI_Introduction_C.git
cd MPI_Introduction_C/labs; source env.sh; cd 1
```

Simplest MPI program: Lab1: C/C++

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  int size, rank;
  char host[1024];
  int hostlen;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(host, &hostlen);
  printf("size=%d, rank=%d hostname=%s\n",
         size, rank, host);
  MPI_Finalize();
}
```

Simplest MPI program: Lab1: C/C++

- An MPI header is included
- MPI is initialized at the beginning and finalized at the end
- We use Intel MPI implementation of MPI that comes with Intel Parallel Studio
- To build the program, use mpicc instead of gcc or mpiicc instead of icc:

```
mpicc -o hello hello.c  // uses gcc
or
mpiicc -o hello hello.c // uses Intel icc
```

- mpicc/mpiicc is simply a wrapper around compiler, in this case gcc/icc, that links all the necessary MPI libraries and provides paths to headers
- To run a program using 2 processes on login node

```
mpirun -n 2 ./hello
```

- Each MPI process executes the same program

Simplest MPI program: Lab1: C/C++

- Only MPI processes that use the same **communicator** can talk to each other. For now we are using `MPI_COMM_WORLD` communicator so that all the processes can talk to each other. In general, you might want to use several communicators to restrict communication in the program.
- The program queries the communicator to find out how many processes use it (`size`) and what is the rank of the current process. Ranks are counted separately for each communicator from `0` to `size-1`. We also ask for the hostname.
- Each process prints size, rank and hostname:

```
size=2, rank=1 hostname=ivy2-XPS-15
size=2, rank=0 hostname=ivy2-XPS-15
```

- The corresponding C++ program looks identical, as far as MPI is concerned, but is compiled with

```
mpic++ -o hello++ hello.C  // using gcc
or
mpiicpc -o hello++ hello.C // using Intel's icpc
```

Simplest MPI program: Lab1: running under Slurm

- To run the program in batch under Slurm

```
sbatch hello.batch
  Submitted batch job 34723042
```

- One can use `squeue` to monitor the job
  - by userid:

    `squeue -u ivy2`
  - by jobid:

    `squeue -j 34723042`
- To kill the job, use

  `scancel 34723042`
- While the job is running, one can ssh to the corresponding compute node and, for example, run top to see what's going on
- In `hello.out`

```
size=2, rank=0 hostname=midway2-0002.rcc.local
size=2, rank=1 hostname=midway2-0003.rcc.local
```

P2P communication: blocking send/receive: Lab2

- `MPI_Send` function can be used to send data to another rank:

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

  where
  - `buf` - array with data
  - `count` - number of units of datatype (`MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, etc.)
  - `dest` - rank of the destination process
  - `tag` - can be used to distinguish one set of messages from another

P2P communication: blocking send/receive: Lab2

- The receiver process must execute `MPI_Receive` to get the message:

```
int MPI_Recv(void* buf, int count,
  MPI_Datatype datatype, int source, int tag,
  MPI_Comm comm, MPI_Status *status)
```

- The receiver might use wildcards instead of source (`MPI_ANY_SOURCE`) and tag (`MPI_ANY_TAG`).
- To find out the actual source and tag of the received message, one can use `status` which is a structure with the following fields: `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`. A user must allocate `MPI_Status` variable before using it or use `MPI_STATUS_IGNORE` instead.
- The status argument can also be used to find out the length of the received message:

```
int MPI_Get_count(const MPI_Status *status,
  MPI_Datatype datatype, int *count)
```

P2P communication: blocking send/receive: Lab2

- MPI receive calls specify a receive buffer and its size has to be enough for any data sent.
- In case you really have no idea how much data is being sent and you don't want to overallocate the receive buffer, you can use a probe call.
- `MPI_Probe` do not copy data but tell that there is a message and you can use `MPI_Get_count` to determine its size, allocate a large enough receive buffer and do a regular receive to have the data copied:

```
if(mytid == receiver) {
  MPI_Status status;
  MPI_Probe(sender,0,comm,&status);
  int count;
  MPI_Get_count(&status, MPI_FLOAT, &count);
  float recv_buffer[count];
  MPI_Recv(recv_buffer, count, MPI_FLOAT,
           sender, 0, comm, MPI_STATUS_IGNORE);
```

P2P communication: blocking send/receive: Lab2

- Sometimes it is convenient to use combined send/receive command
- A good use case is when two patches on the grid need to exchange boundaries
- Executing a single sendrecv is faster and less error prone then making two calls

```
int MPI_Sendrecv(const void *sendbuf, int sendcount,
  MPI_Datatype sendtype, int dest, int sendtag,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
  int source, int recvtag, MPI_Comm comm,
  MPI_Status *status)
```

P2P communication: blocking send/receive: Lab2

- `MPI_Send` and `MPI_Receive` functions are **blocking**:
  - sender can procede only after it is safe to overwrite the message: it was copied to some buffer but not necessarily delivered yet
  - receive can procede only after the message is delivered
- There are several problems with that:
  - One needs to be very careful while ordering send/receive to avoid **deadlock**. Consider the situation:

```
Process 0:
  receive from 1
  send to 1
Process 1:
  receive from 0
  send to 0
```

    Both processes are waiting for each other and get stuck.
  - While waiting for send/receive to complete, neither sender nor receiver can do any useful job

P2P communication: non-blocking send/receive: Lab3

- Non-blocking send/receive calls solve these two problems:
  - The calls return immediately
  - One can either query later if calls are finished or wait for the end of all or some of such calls at some point in the program
  - This allows to overlap computation and communication
  - One needs to be careful not to overwrite the message storage before it is delivered

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

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

P2P communication: non-blocking send/receive: Lab3

- `request` is used in `MPI_Wait` and `MPI_Test` to test the completion of non-blocking calls

```
int MPI_Wait(MPI_Request *request,
  MPI_Status *status)
int MPI_Test(MPI_Request *request, int *flag,
  MPI_Status *status)
```

- A call to `MPI_Wait` returns when the operation identified by request is complete and can be used once
- A call to `MPI_Test` returns `flag=true` if the operation identified by request is complete and can be used multiple times
- A request object can be deallocated without waiting for the associated communication to complete:

```
int MPI_Request_free(MPI_Request *request)
```

P2P communication: non-blocking send/receive: Lab3

- `MPI_Waitall` allows you to wait for a number of requests and it does not matter in what sequence they are satisfied:

```
int MPI_Waitall(int count,
  MPI_Request array_of_requests[],
  MPI_Status array_of_statuses[])
```

- There is also corresponding `MPI_Testall`
- Instead of waiting/testing for all the requests in the list, one can wait/test for some to happen:

```
int MPI_Waitsome(int incount,
  MPI_Request array_of_requests[],
  int *outcount, int array_of_indices[],
  MPI_Status array_of_statuses[])
```

- Waits until at least one of the operations associated with active handles in the list have completed. Returns in outcount the number of requests from the list `array_of_requests` that have completed.

P2P communication: non-blocking send/receive: Lab3

- An Isend or Irecv call has an `MPI_Request` parameter. This is an object that gets created in the send/recv call and deleted in the wait call.
- If the same communication happens many times, you can avoid this overhead by reusing the request object: **MPI persistent communication**
- You describe the communication with `MPI_Send_init`/`MPI_Recv_init` which has the same calling sequence as `MPI_send`/`MPI_Recv`
- The actual communication is performed by calling `MPI_Start`, for a single request, or `MPI_Startall` for an array of requests.
- Completion of the communication is confirmed with `MPI_Wait` or similar routine
- The wait call does not release the request object: that is done with `MPI_Request_free`.

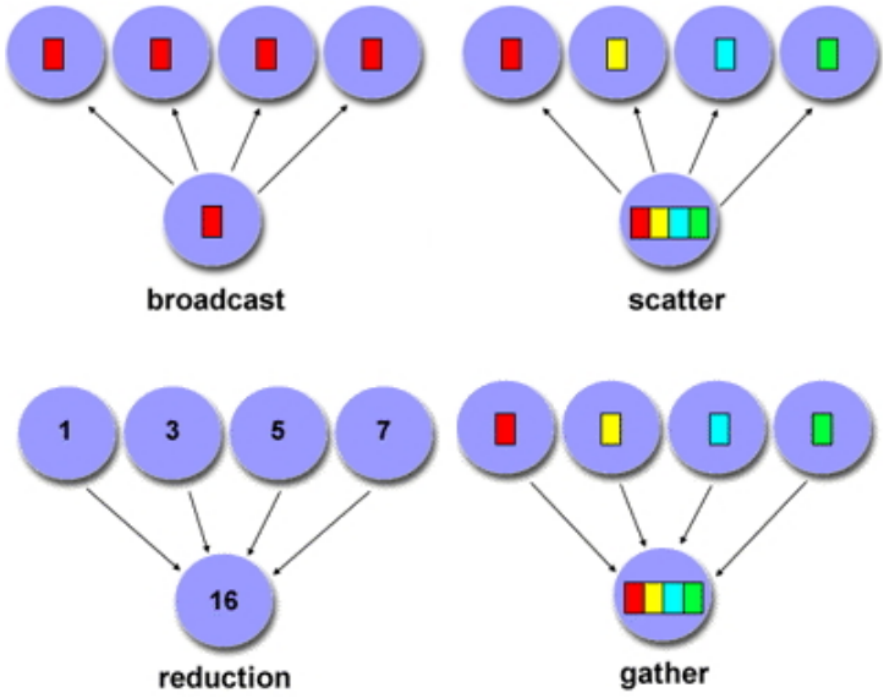
P2P communication: non-blocking send/receive: Lab3

- There are several other more exotic send/receive versions:
  - `MPI_Bsend` - buffered
  - `MPI_Ssend` - synchronous
  - `MPI_Rsend` - ready

  and the corresponding non-blocking calls
- One can mix different types of send and receive calls
- Whether the standard blocking send/receive calls are blocked or not, might depend on the size of the message: small messages might not be blocked and sent immediately to be stored in a remote buffer (therefore avoiding deadlocks that would happen for larger messages)

Collectives

- Collectives are operations that involve all processes in a communicator
- A collective is a single call and it blocks on all processes



Collectives: Broadcast: Lab4

- A process has data that needs to be shared with all the others. For example:
  - Process 0 parses command line arguments
  - Process 0 reads input file
- `MPI_Bcast` needs to be executed on all the processes

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

here `root` is the rank of the process that sends data

Collectives: Scatter: Lab5

- A process has an array `sendbuf` that should be divided into subarrays of `sendcount` elements which are sent into `recvbuf` on each process:

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

Collectives: Scatterv

- There is version of scatter that gives more control over which part of `sendbuf` sent where

```
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)
```

  Here `sendcounts` and `displs` specify how many elements from which location in `sendbuf` to send to the processes

Collectives: Gather: Lab5

- A process needs to collect into a single array `recvbuf` arrays of `sendcount` elements from all the processes

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

  where `root` is a rank of the receiving process

Collectives: Gatherv

- There is a version of gather that allows more control over how many elements are received from each process and where they are inserted into the `recvbuf`

```
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)
```

  Here
  - `recvcounts` is an array of non-negative integers containing the number of elements received from each process
  - `displs` an array of indexes where to put received data into `recvbuf`.

Collectives: Allgather

- It might be necessary to distribute gathered result to all the nodes. Instead of doing two separate calls, gather and broadcast, it is more efficient to use

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

Collectives: Alltoall

- `MPI_Alltoall` is an extension of `MPI_Allgather` to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of `recvbuf`.

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

Collectives: Reduce: Lab6

- Reduce aggregates data from all the processes into sum, product, maximum, minimum, minloc, maxloc, and, or, etc:

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

- On root rank, one can use `MPI_IN_PLACE` instead of `sendbuf` to avoid allocating two buffers but reuse `recvbuf` to return the result of reduce in place
- One can define custom aggregation operation:

```
int MPI_Op_create(MPI_User_function* user_fn,
  int commute, MPI_Op* op)
```

- The user-defined operation is assumed to be associative.

Collectives: Reduce: Lab6

- The argument `user_fn` is the user-defined function of the following type:

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

  where

```
inoutvec[i] = invec[i] op inoutvec[i], i = 1..len-1
```

- After the user defined operation is used, it should be deallocated:

```
int MPI_Op_free(MPI_Op *op)
```

Collectives: Allreduce

- There is a modification of reduce where each process gets aggregated value:

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

- It is faster then doing two operations: reduce and broadcast
- There is also `MPI_Reduce_scatter` available

Collectives: Scan

- If process i contains a number x_i and $\oplus$ is the operation, then the **inclusive scan** operation leaves $x_0 \oplus ... \oplus x_i$ on processor i

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

- An **exclusive scan** operation leaves $x_0 \oplus x_i$ on processor i and its result is undefined on processor 0

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

Collectives: Barrier

- The call blocks until all the processes reached it

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

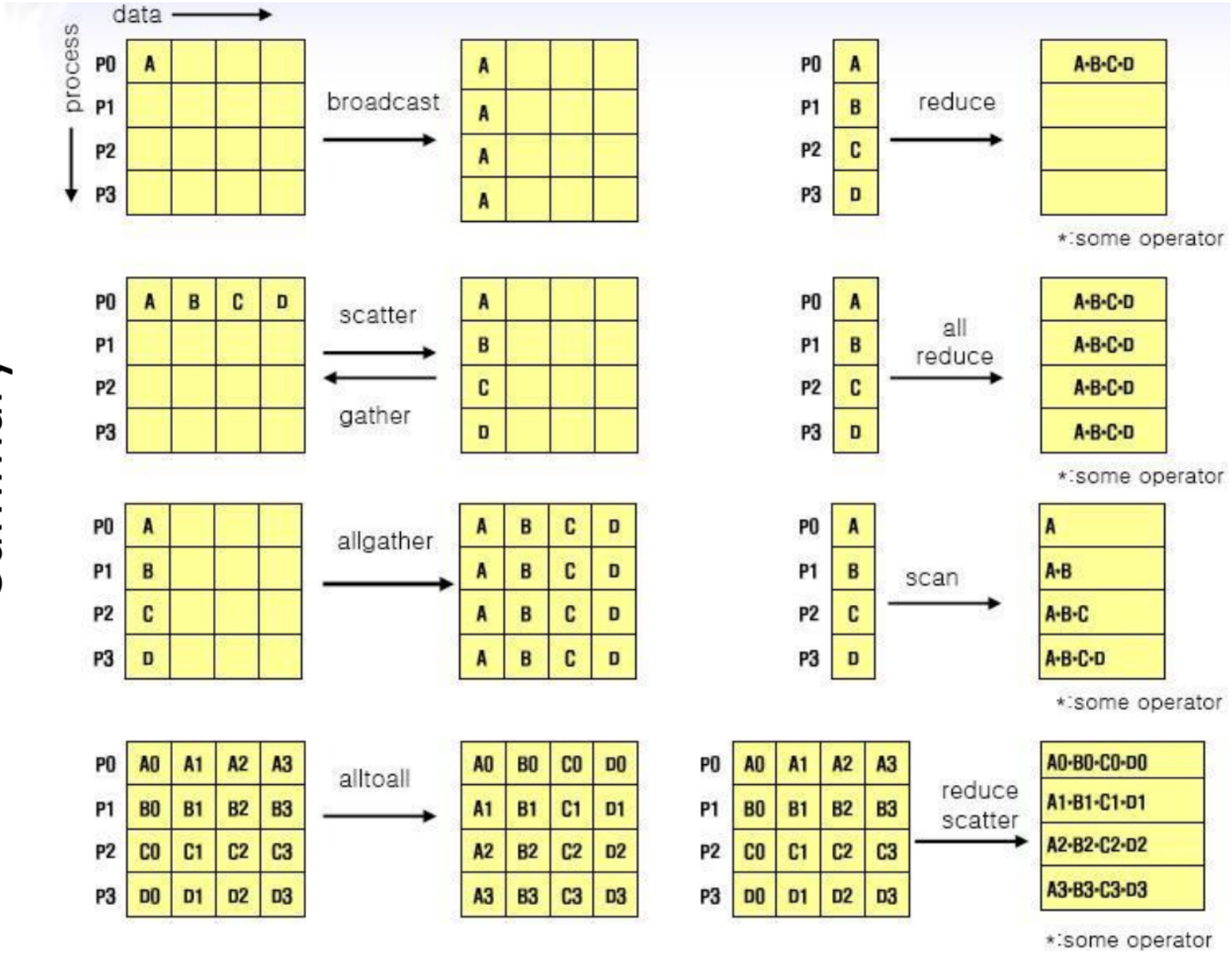
- It is almost never necessary to synchronize processes through a barrier: for most purposes it does not matter if processes are out of sync. Conversely, collectives introduce a barrier of sorts themselves.

Collectives: non-blocking collective operations

There are non-blocking versions of collective operations:

- `MPI_Ibcast`
- `MPI_Igather`, `MPI_Igatherv`
- `MPI_Iscatter`, `MPI_Iscatterv`
- `MPI_Iallgather`, `MPI_Iallgatherv`
- `MPI_Ialltoall`, `MPI_Ialltoallv`
- `MPI_Ireduce`, `MPI_Iallreduce`
- `MPI_Iscan`, `MPI_Iexscan`

Collective Communication: Summary



MPI data types: elementary

- So far we were sending contiguous buffer with elements of a single type:

  `MPI_CHAR, MPI_UNSIGNED_CHAR, MPI_SIGNED_CHAR`
  `MPI_SHORT, MPI_UNSIGNED_SHORT`
  `MPI_INT`
  `MPI_UNSIGNED`
  `MPI_LONG, MPI_UNSIGNED_LONG`
  `MPI_FLOAT`
  `MPI_DOUBLE`
  `MPI_LONG_DOUBLE`

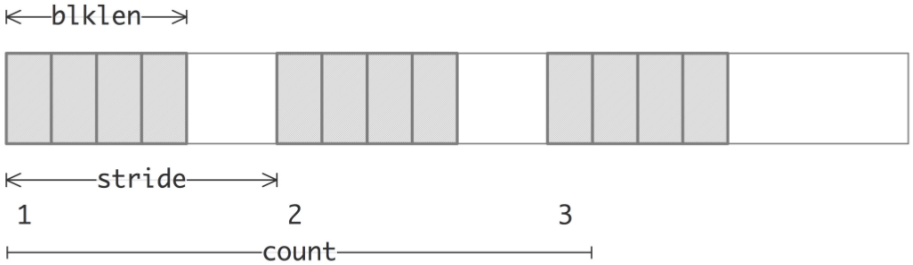
- MPI allows creating your own types that are not contiguous or are composite. Those are called **derived**.

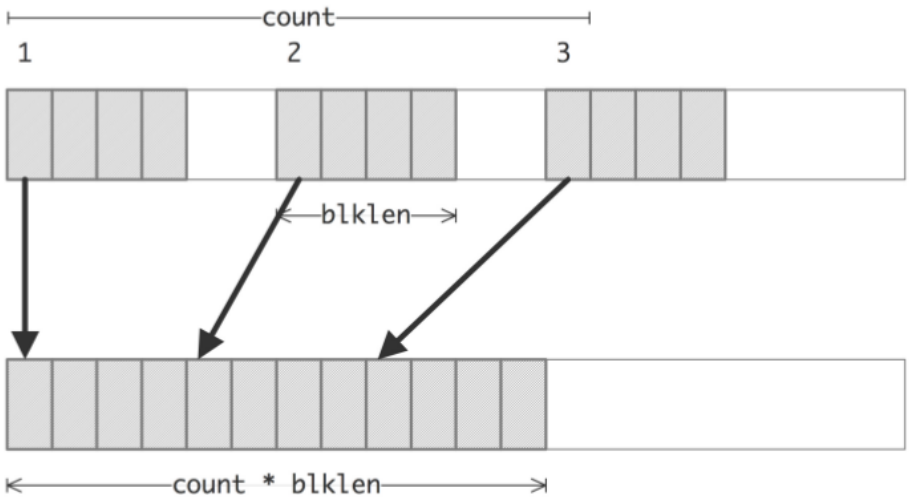
MPI data types: derived: vector: Lab7

- `MPI_Type_vector` describes a series of blocks, all of equal size, spaced with a constant stride

```
int MPI_Type_vector(int count, int blocklength,
  int stride, MPI_Datatype oldtype,
  MPI_Datatype *newtype)
```

- In the next example a vector type is created only on the sender, in order to send a strided subset of an array; the receiver receives the data as a contiguous block.





MPI data types: derived: vector

```
  source = (double*) malloc(stride*count*sizeof(double));
  target = (double*) malloc(count*sizeof(double));
  MPI_Datatype newvectortype;
  if (procno==sender) {
    MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    MPI_Type_free(&newvectortype);
  } else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
             &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
}
```

MPI data types: derived: subarray

- The vector datatype can be used for example, for cutting out blocks in an array of dimension 2.
- It still can be used recursively for higher dimensions but it gets tedious. Instead, there is an explicit subarray type:

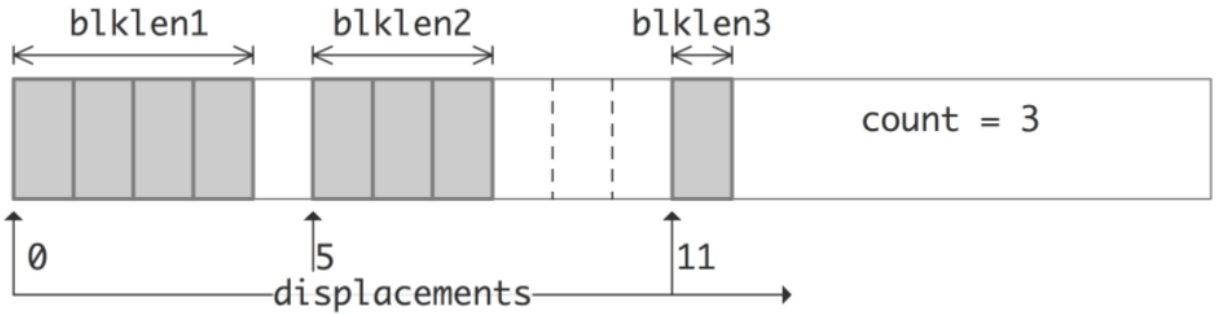
```
int MPI_Type_create_subarray(int ndims,
  const int array_of_sizes[],
  const int array_of_subsizes[],
  const int array_of_starts[], int order,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
```

  - `array_of_sizes` - dimensions of full `ndims`-array,
  - `array_of_subsizes` - dimensions of a subarray which starts at `array_of_starts`

MPI data types: derived: indexed: Lab7

- The indexed datatype can send arbitrary located elements from an array of a single datatype:

```
int MPI_Type_indexed(int count,
  const int array_of_blocklengths[],
  const int array_of_displacements[],
  MPI_Datatype oldtype, MPI_Datatype *newtype)
```

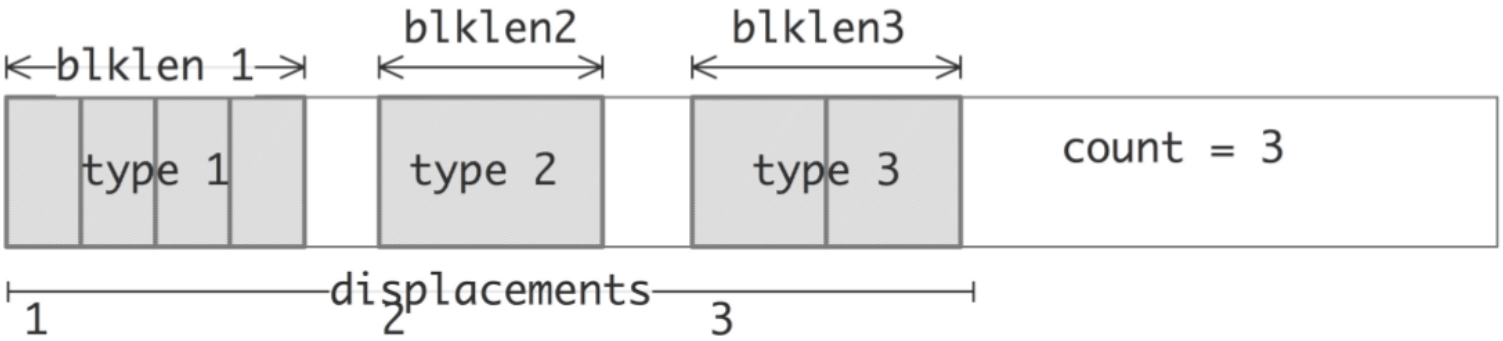


- There is also `MPI_Type_create_hindexed` that measures index locations in bytes, rather than multiples of the old type.

MPI data types: derived: struct: Lab7

- The structure type can contain multiple data types.

```
int MPI_Type_create_struct(int count,
  int blocklengths[], MPI_Aint displacements[],
  MPI_Datatype types[], MPI_Datatype *newtype);
```



MPI data types: derived: struct

```
struct object {
  char c;
  double x[2];
  int i;
};
MPI_Datatype newstructuretype;
int structlen = 3;
int blocklengths[structlen];
MPI_Datatype types[structlen];
MPI_Aint displacements[structlen];
blocklengths[0] = 1; types[0] = MPI_CHAR;
displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;
blocklengths[1] = 2; types[1] = MPI_DOUBLE;
displacements[1] = (size_t)&(myobject.x[0]) - (size_t)&myobject;
blocklengths[2] = 1; types[2] = MPI_INT;
displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;
```

MPI data types: derived: struct

```
MPI_Type_create_struct(structlen, blocklengths,
  displacements, types, &newstructure);
MPI_Type_commit(&newstructuretype);
{
  MPI_Aint typesize;
  MPI_Type_extent(newstructuretype,&typesize);
  if(procno==0)
    printf("Type extent: %d bytes\n",typesize);
}
if (procno==sender) {
  MPI_Send(&myobject,1,newstructuretype,the_other,0,comm);
} else if (procno==receiver) {
  MPI_Recv(&myobject,1,newstructuretype,the_other,0,comm,
           MPI_STATUS_IGNORE)
}
MPI_Type_free(&newstructuretype);
```

MPI data types: derived: contiguous

- There is also a contiguos data type:

```
int MPI_Type_contiguous(int count,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- At first glance, there is no difference between using an array of elementary data types and contiguous type.
- The main reason to use contiguous type is to overcome a limitation on how many elements can be in the array of elementary data types (`MPI_int` allows $2^{31} - 1$ elements): to send more data one can use an array of contiguous data types instead.

Communicators

- So far we have been using `MPI_COMM_WORLD` communicator that includes all the MPI processes requested by `mpirun`
- While only two processes communicate in send/receive operations, in collectives all the processes in the communicator talk to each other. This might be quite expensive for a program with millions of processes
- Often there is a natural way to divide communication in the program into subgroups:
  - Example 1. Simulation
    - inputs are generated
    - computations are performed on them
    - the results of the computations are analyzed and rendered graphically

    Consider dividing your processes in three groups corresponding to generation, computation, rendering.
  - Example 2. You are doing some computation with a matrix and sometimes you need to make a collective calls only on a row/column (for example, to sum them with reduce). Create separate communicators for each row/column.

Communicators: split by color: Lab8

- One way to make processes to use different communicators it to use

```
int MPI_Comm_split(MPI_Comm comm,
  int color, int key, MPI_Comm* newcomm)
```

- This is a collective call that runs on all the processes belonging to `comm`. For each rank it computes `color` and as a result ranks are divided into groups with the same color.
- `key` is used to decide how to sort ranks within the new communicator `newcomm`. For example, if you use `comm`'s rank as `key`, the ranks within each `newcomm` will preserve the same order as in the original communicator
- When creating separate communicator for each column in a matrix, use column number as color for a column to end up in the same communicator
- After done using `newcomm`, free it with `MPI_Comm_free`.

Communicators: split by color

```
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int color = world_rank / 4;
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);
int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);
printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
world_rank, world_size, row_rank, row_size);
MPI_Comm_free(&row_comm);
```

Communicators: groups

- Internally MPI communicator has two parts:
  - ID that differentiates one communicator from another
  - group of processes using the communicator
- Using

  ```
  int MPI_Comm_group(MPI_Comm comm, MPI_Group* group)
  ```

  one can get `group` object, then one can use the existing groups to create new groups that are union, intersection, subset, and finally one can create new communicator with the resulting group.

Communicators: groups

```
int MPI_Group_union( MPI_Group group1,
  MPI_Group group2, MPI_Group* newgroup)

int MPI_Group_intersection(MPI_Group group1,
  MPI_Group group2, MPI_Group* newgroup)

int MPI_Group_incl(MPI_Group group, int n,
  const int ranks[], MPI_Group* newgroup)

int MPI_Comm_create_group(MPI_Comm comm,
  MPI_Group group, int tag, MPI_Comm* newcomm)
```

Communicators: dup

- Suppose MPI program is using library that internally also uses MPI.
- Even if the program and the library operate on the same set of processes, one should use two different copies of communicators to avoid messages intended for your program to be mistakenly intercepted by a library or vice versa.

```
int MPI_Comm_dup(MPI_Comm comm,
    MPI_Comm *newcomm)
```

Communicators: topology: Lab8

- In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes
- Often the processes are logically arranged in n-dimensional grids, sometimes in a more general graph
- We call logical process arrangement of processes - “virtual topology”.
- A topology can provide a convenient naming mechanism for the processes of a group
- Based on existing communicator with linear ranking, one can create a new communicator that in addition has topology information.
- There are routines to query the topology of the communicator
- For communicators with cartesian topology, there are routines to translate between linear and n-dimensional cartesian ranks
- There are routines to find the neighbor in cartesian grid along certain dimension.
- There are collectives that do data exchange between neighbors in virtual topology.

Communicators: topology: cartesian: Lab8

```
int MPI_Cart_create(MPI_Comm comm_old,
  int ndims, const int dims[],
  const int periods[], int reorder,
  MPI_Comm *comm_cart)
```

- `ndims` - number of dimensions of Cartesian grid
- `dims` - integer array of size `ndims` specifying the number of processes in each dimension
- `periods` - logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
- `reorder` - ranking may be reordered (true) or not (false)

Communicators: topology: cartesian: Lab8

- One can query the topology to find parameters of the cartesian grid and translate between linear rank and cartesian coordinates

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

int MPI_Cart_get(MPI_Comm comm, int maxdims,
  int dims[], int periods[], int coords[])

int MPI_Cart_rank(MPI_Comm comm, const int coords[],
  int *rank)

int MPI_Cart_coords(MPI_Comm comm, int rank,
  int maxdims, int coords[])
```

here `coords` are coordinates of the calling process

Communicators: topology: cartesian: Lab8

- One can find neighbors on the grid or in a graph:

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank,
  int *nneighbors)

int MPI_Graph_neighbors(MPI_Comm comm, int rank,
  int maxneighbors, int neighbors[])
```

- If the process topology is a Cartesian structure, an `MPI_SENDRECV` operation may be used along a coordinate direction to perform a shift of data. As input, `MPI_SENDRECV` takes the rank of a source process for the receive, and the rank of a destination process for the send. To get this information, call

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
  int disp, int *rank_source, int *rank_dest)
```

shift `disp` units along dimension `direction`

Communicators: topology: cartesian: Lab8

- One can partition cartesian space into subspaces:

```
int MPI_Cart_sub(MPI_Comm comm,
  const int remain_dims[], MPI_Comm *newcomm)
```

Communicators: topology: cartesian: collectives

- Neighborhood gather:

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

- In this function, each process i gathers data items from each process j if an edge (j, i) exists in the topology graph, and each process i sends the same data items to all processes j where an edge (i, j) exists. The send buffer is sent to each neighboring process and the l-th block in the receive buffer is received from the l-th neighbor.

Communicators: topology: cartesian: collectives

- The vector version of gather allows to get different number of elements from different neighbors:

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

- `recvcounts` - Non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor
- `displs` - Integer array (of length indegree). Entry i specifies the displacement (relative to `recvbuf`) at which to place the incoming data from neighbor i.

Communicators: topology: cartesian: collectives

- In this function, each process i receives data items from each process j if an edge (j, i) exists in the topology graph or Cartesian topology. Similarly, each process i sends data items to all processes j where an edge (i, j) exists. This call is more general than `MPI_Neighbor_allgather` in that different data items can be sent to each neighbor. The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in the receive buffer is received from the l-th neighbor.

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

- `MPI_Neighbor_alltoallv` allows sending/receiving different numbers of elements to and from each neighbor.
- `MPI_Neighbor_alltoallw` allows one to send and receive with different data types to and from each neighbor.
- Non-blocking neighbor collectives... Collectives for graphs...

Communicators: inter-communicators

- If two disjoint communicators exist, it may be necessary to communicate between them.
- `MPI_Intercomm_create` is used to bind two intra-communicators into an **inter-communicator**
- `MPI_Intercomm_merge` creates an intra-communicator by merging the local and remote groups of an inter-communicator
- Overlap of local and remote groups that are bound into an inter-communicator is prohibited.
- The function `MPI_Intercomm_create` can be used to create an inter-communicator from two existing intra-communicators, in the following situation: At least one selected member from each group (the "group leader") has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator. Furthermore, members of each group know the rank of their leader.

Communicators: inter-communicators

```
int MPI_Intercomm_create(MPI_Comm local_comm,
  int local_leader, MPI_Comm peer_comm,
  int remote_leader, int tag,
  MPI_Comm *newintercomm)
```

- This call creates an inter-communicator.
- It is collective over the union of the local and remote groups.
- Processes should provide identical `local_comm` and `local_leader` arguments within each group
- After this, the intercommunicator can be used in collectives.

Communicators: inter-communicators

```
int MPI_Intercomm_merge(MPI_Comm intercomm,
  int high, MPI_Comm *newintracomm)
```

- This function creates an intra-communicator from the union of the two groups that are associated with intercomm.
- All processes should provide the same high value within each of the two groups.
- If processes in one group provided the value *high* = *false* and processes in the other group provided the value *high* = *true* then the union orders the "low" group before the "high" group.
- If all processes provided the same high argument then the order of the union is arbitrary.
- This call is blocking and collective within the union of the two groups.

MPI-IO: motivation

- How to get the data from hundreds thousands of nodes on the cluster to disks?
- I/O is much slower then computing and might become major bottleneck
- Number of tasks keeps rising rapidly
- The size of the data is also rapidly increasing
- The gap between computational power and I/O rates keeps increasing
- Letting each process write its own file and later gluing them all together? - Might be very inefficient if done on the same file system, overloads metadata server.
- MPI-IO can often do it several hundered times faster by providing collective access to parallel file system (GPFS, Lustre, ...) and underlying storage hardware to all the processes in the communicator

MPI-IO: opening and closing files

```
int MPI_File_open(MPI_Comm comm, const char *filename,
  int amode, MPI_Info info, MPI_File *fh)
```

- It is a collective routine: all the processes in the communicator must provide the same value for `amode` and `filename`
- Some possible values for `amode`:
  - `MPI_MODE_RDONLY` - read only
  - `MPI_MODE_RDWR` - write only
  - `MPI_MODE_WRONLY` - write only
  - `MPI_MODE_CREATE` - create the file if it does not exist
  - `MPI_MODE_SEQUENTIAL` - file will only be accessed sequentially
- Via `info` object one can provide hints to help to optimize I/O, `MPI_INFO_NULL` - no hints, hints are implementation and hardware specific
- The output of the call is the file handle `fh`
- The file is closed with

  `int MPI_File_close(MPI_File *fh)`

MPI-IO: file pointer: Lab9

- Each process moves its local file pointer (individual file pointer) with

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset,
  int whence)
```

- `whence` can be
  - `MPI_SEEK_SET` - the pointer is set to `offset` counting from the beginning of the file
  - `MPI_SEEK_CUR` - the pointer is set to the current position plus `offset`
  - `MPI_SEEK_END` - the point is set to the end of file plus `offset`
- One can query the current position of the file pointer with

```
int MPI_File_get_position(MPI_File fh,
  MPI_Offset *offset)
```

- `MPI_Offset` is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type `MPI_Offset`.

MPI-IO: file pointer: Lab9

- Read file at individual file pointer (can be different for different processes)

```
int MPI_File_read(MPI_File fh, void *buf,
  int count, MPI_Datatype datatype, MPI_Status *status)
```

- Updates position of file pointer after reading
- Not thread safe
- Similarly, one can write using individual file pointer:

```
int MPI_File_write(MPI_File fh, const void *buf,
  int count, MPI_Datatype datatype, MPI_Status *status)
```

MPI-IO: explicit offset

- Instead of using individual (per process) file pointers that get changed by read/write and are not thread safe, one can use explicit offsets:

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset,
  void *buf, int count, MPI_Datatype datatype,
  MPI_Status *status)

int MPI_File_write_at(MPI_File fh, MPI_Offset offset,
  const void *buf, int count, MPI_Datatype datatype,
  MPI_Status *status)
```

- Thread-safe
- The file pointer is neither referred nor incremented

MPI-IO: collective and non-blocking operations

- I/O can be performed collectively by all processes in a communicator
  - `MPI_File_read_all`
  - `MPI_File_write_all`
  - `MPI_File_read_at_all`
  - `MPI_File_write_at_all`
- Same parameters as in independent I/O functions
- All processes in communicator that opened file must call function
- Performance potentially better than for individual functions
  - Even if each processor reads a non-contiguous segment, in total the read is contiguous
- There are also corresponding non-blocking functions that allow to overlap I/O with computations

MPI-IO: Non-contiguous data access: fileview: Lab9

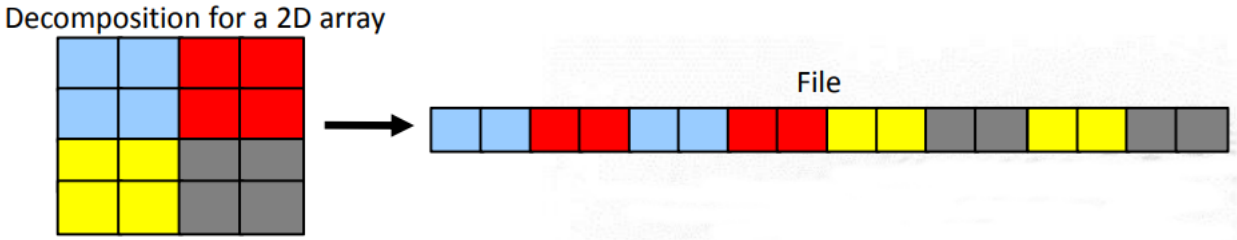
- By default, file is treated as consisting of bytes, and process can access (read or write) any byte in the file
- The fileview defines which portion of a file is visible to a process
- A fileview consists of three components
  - `displacement`: number of bytes to skip from the beginning of file
  - `etype`: type of data accessed, defines unit for offsets
  - `filetype`: portion of file visible to a process - can be non-contiguous like such datatypes as vector, indexed, subarray, etc.

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
  MPI_Datatype etype, MPI_Datatype filetype,
  const char *datarep, MPI_Info info)
```

- `datarep` - data representation, controls file portability vs performance

MPI-IO: Non-contiguous data access: fileview: Lab9

- MPI-IO writes data into a file in a binary format. Possible values for `datarep`
  - native - default; data is written to file exactly as it is stored in memory; best performance; might be non-portable between different hardware or MPI implementations
  - internal - portable for the same MPI implementation on different hardware; worse performance
  - external32 - completely portable; possibly worse performance; possibly some loss in precision
- Each process has to access small pieces of data scattered throughout a file; this is controlled by `filetype`, `displacement` and `etype` (in terms of which everything is measured). Here different processes write to subarrays of different colors scattered in the file:



MPI-IO: Non-contiguous data access: fileview: Lab9

- This would be very expensive if implemented with separate reads/writes:
  - I/O is done in blocks: the whole block has to be read from disk even if a small piece of data from it is required
  - Blocks are accessed sequentially to avoid race condition
  - If two different data items in the same block are needed by different processes, this would be done sequentially - false sharing
  - MPI-IO sees the whole picture and can read necessary blocks into memory and then send different pieces of it to different processes avoiding false sharing
- If you do not use fileview, the data layout in a file would depend on the number of processes used in a job and you would have to do later some translation into a more portable format

HDF5: Lab10

- If you want to sacrifice some performance for a lot of convenience, you can use parallel version of HDF5 library that knows how to talk to MPI and do parallel I/O
- The advantage is that you are working with a standard portable format for scientific computing that can be used in combination with many other tools and languages: ParaView, VisIt, python, C/C++, ROOT, etc.

T3PIO

- Some sites, for example TACC, developed libraries to provide reasonable hints to MPI-IO given the existing hardware.
- The parallel I/O performance depends on three parameters:
  - Number of writers
  - Number of stripes
  - Stripe size
- T3IO library at TACC is claimed to be able to speed up MPI-IO by a factor of 10s by providing reasonable default parameters to Lustre file system
- MPI-IO can take hints, if it was compiled with the support for the particular file system
- The hints are passed via `info` object in the form of key-value pairs. Those pairs that are not understood, are ignored. Also, even if the hints are understood, there is no guarantee that they will be used. It is implementation dependent.
- Since parallel HDF5 is built on top of MPI-IO, T3PIO and hints in general might help it as well.

One-sided communication: motivation

- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - one should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory
  - Can do multiple data transfers with a single synchronization operation
- Some irregular communication patterns can be more economically expressed
- Can be significantly faster than send/receive on systems with hardware support for remote memory access (RMA)

One-sided communication: window

- One can declare part of local memory, called **window**, to be publically accessible in four ways:
  1. You already have an allocated buffer that you would like to make remotely accessible

```
int MPI_Win_create(void *base, MPI_Aint size,
  int disp_unit, MPI_Info info,
  MPI_Comm comm, MPI_Win *win)
```

     - `base` - pointer to the existing buffer
     - `size` - size of window in bytes
     - `disp_unit` - local unit size for displacements, in bytes

  2. You want to create a buffer and directly make it remotely accessible. This is better since MPI might allocate data in a more optimal memory address for MPI communication. If the first option is used, use `MPI_alloc` instead of malloc.

```
int MPI_Win_allocate(MPI_Aint size, int disp_unit,
  MPI_Info info, MPI_Comm comm, void *baseptr,
  MPI_Win *win)
```

One-sided communication: window

4. You don't have a buffer yet, but will have one in the future

```
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm,
  MPI_Win *win)
```

   - Later you can use the following calls to attach/detach a buffer to/from the window:

```
int MPI_Win_attach(MPI_Win win, void *base,
  MPI_Aint size)
int MPI_Win_detach(MPI_Win win,
  const void *base)
```

5. You want multiple processes on the same node to share a buffer

```
int MPI_Win_allocate_shared(MPI_Aint size,
  int disp_unit, MPI_Info info, MPI_Comm comm,
  void *baseptr, MPI_Win *win)
```

- Window creation are collective calls executed on all the processes of the communicator
- The buffer sizes can be different for different processes
- To release the window, run on all the involved processes

```
int MPI_Win_free(MPI_Win *win)
```

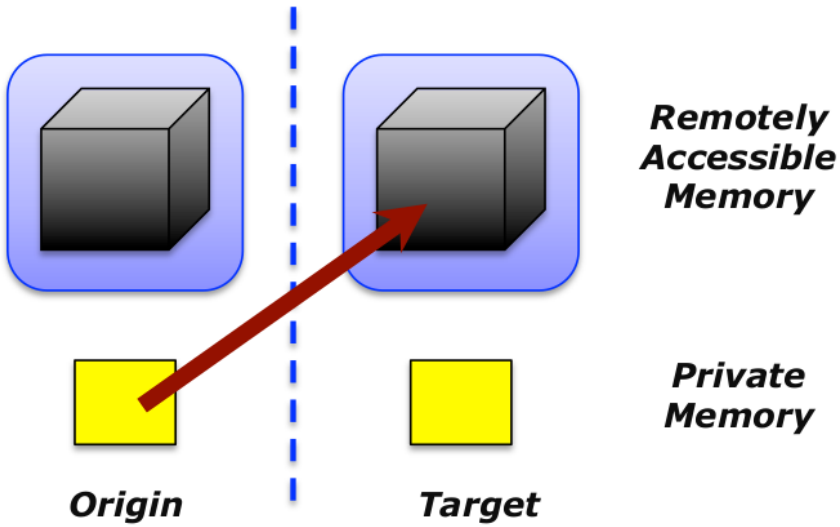
One-sided communication: data movement

- MPI provides ability to read, write and atomically modify data in remotely accessible memory regions
  - `MPI_Get`
  - `MPI_Put`
  - `MPI_Accumulate`
  - `MPI_Get_accumulate`
  - `MPI_Compare_and_swap`
  - `MPI_Fetch_and_op`
- All data movement operations are non-blocking

One-sided communication: data movement: put

- Move data from origin, to target

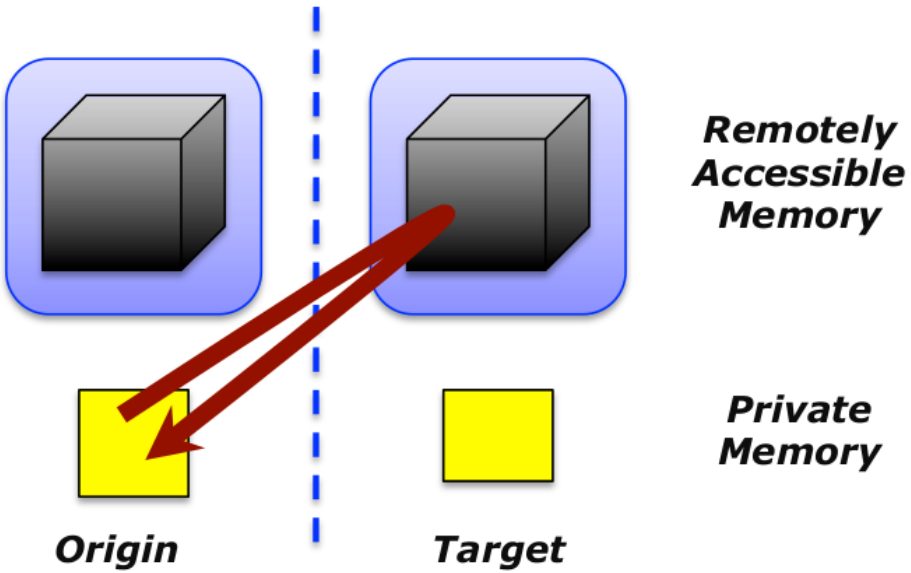
```
int MPI_Put(const void *origin_addr, int origin_count,
  MPI_Datatype origin_datatype, int target_rank,
  MPI_Aint target_disp, int target_count,
  MPI_Datatype target_datatype, MPI_Win win)
```



One-sided communication: data movement: get

- Move data to origin, from target

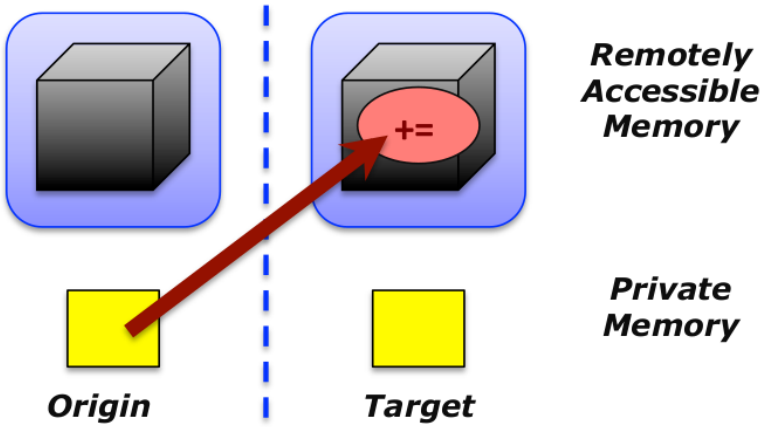
```
int MPI_Get(void *origin_addr, int origin_count,
  MPI_Datatype origin_datatype, int target_rank,
  MPI_Aint target_disp, int target_count,
  MPI_Datatype target_datatype, MPI_Win win)
```



One-sided communication: data movement: accumulate

- Element-wise atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Predefined ops only, no user-defined operations

```
int MPI_Accumulate(const void *origin_addr,
  int origin_count, MPI_Datatype origin_datatype,
  int target_rank, MPI_Aint target_disp,
  int target_count, MPI_Datatype target_datatype,
  MPI_Op op, MPI_Win win)
```



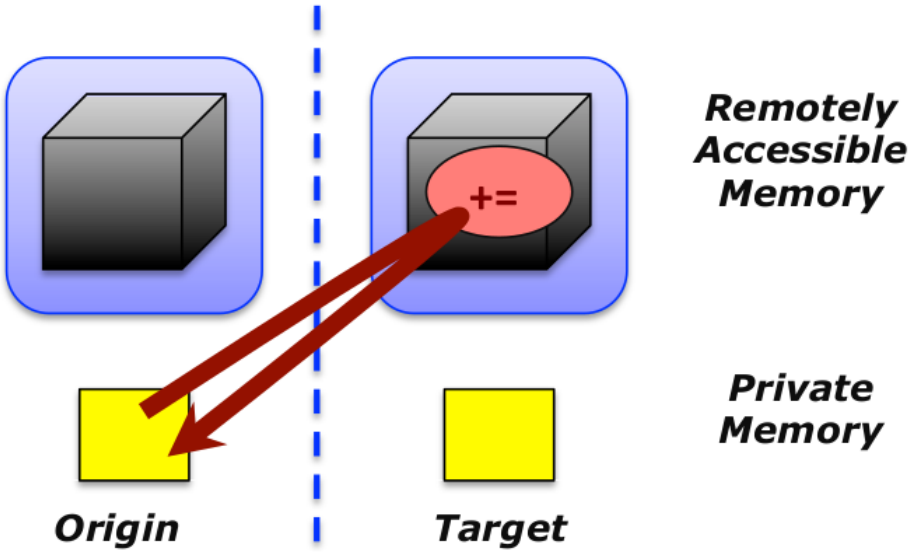
- `Op = MPI_REPLACE` corresponds to atomic put

One-sided communication: data movement: get accumulate

- Element-wise atomic read-modify-write
  - Op = `MPI_SUM`, `MPI_PROD`, `MPI_OR`, `MPI_REPLACE`, `MPI_NO_OP`, ...
- Result stored in target buffer
- Original data stored in result buf on the origin node
- Element-wise atomic get with `MPI_NO_OP`
- Element-wise atomic swap with `MPI_REPLACE`

```
int MPI_Get_accumulate(const void *origin_addr,
  int origin_count, MPI_Datatype origin_datatype,
  void *result_addr, int result_count,
  MPI_Datatype result_datatype, int target_rank,
  MPI_Aint target_disp, int target_count,
  MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

One-sided communication: data movement: fetch and op



- `MPI_Fetch_and_op` is a simpler version of `MPI_Get_accumulate`:
  - All buffers share a single predefined datatype
  - No count argument (it's always 1)
  - Simpler interface allows hardware optimization

```
int MPI_Fetch_and_op(const void *origin_addr,
  void *result_addr, MPI_Datatype datatype,
  int target_rank, MPI_Aint target_disp,
  MPI_Op op, MPI_Win win)
```

One-sided communication: data movement: compare and swap

- Another useful operation is an atomic compare and swap where the value at the origin is compared to the value at the target, which is atomically replaced by a third value only if the values at origin and target are equal.

```
int MPI_Compare_and_swap(const void *origin_addr,
  const void *compare_addr, void *result_addr,
  MPI_Datatype datatype, int target_rank,
  MPI_Aint target_disp, MPI_Win win)
```

One-sided communication: ordering of operations

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location undefined
- Result of Get concurrent with Put/Accumulate undefined
- Result of concurrent accumulate operations to the same location are defined according to the order in which the occurred
- Remember that you can simulate atomic put and get:
  - Atomic put: `Accumulate` with `op = MPI_REPLACE`
  - Atomic get: `Get_accumulate` with `op = MPI_NO_OP`

One-sided communication: synchronization models

- RMA data access model
  - When is a process allowed to read/write remotely accessible memory?
  - When is data written by process X is available for process Y to read?
- Three synchronization models provided by MPI:
  - **Fence - active target synchronization**
  - **Post-start-complete-wait - generalized active target synchronization**
  - **Lock/Unlock - passive target synchronization**
- Data accesses occur within **epochs**:
  - **Access epochs**: contain a set of operations issued by an origin process
  - **Exposure epochs**: enable remote processes to access and/or update a target's window
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms for establishing epochs

One-sided communication: synchronization models: fence

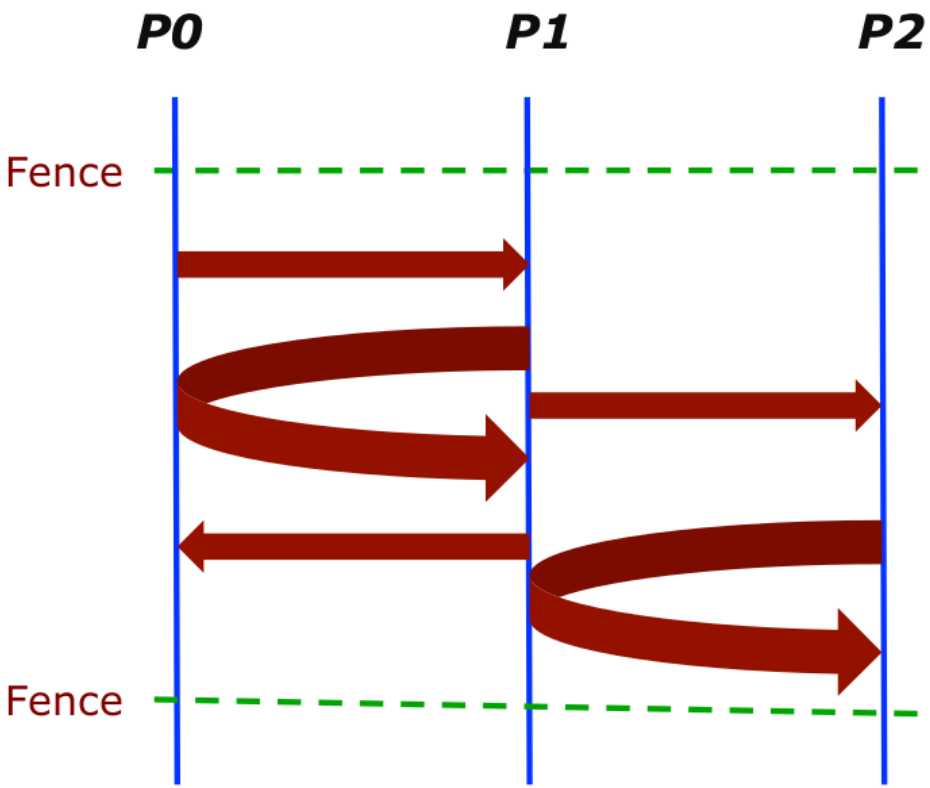
- Active target synchronization
- Collective synchronization model: starts and ends access and exposure epochs on all processes in the window

```
int MPI_Win_fence(int assert, MPI_Win win)
```

  - The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. A value of `assert = 0` is always valid.

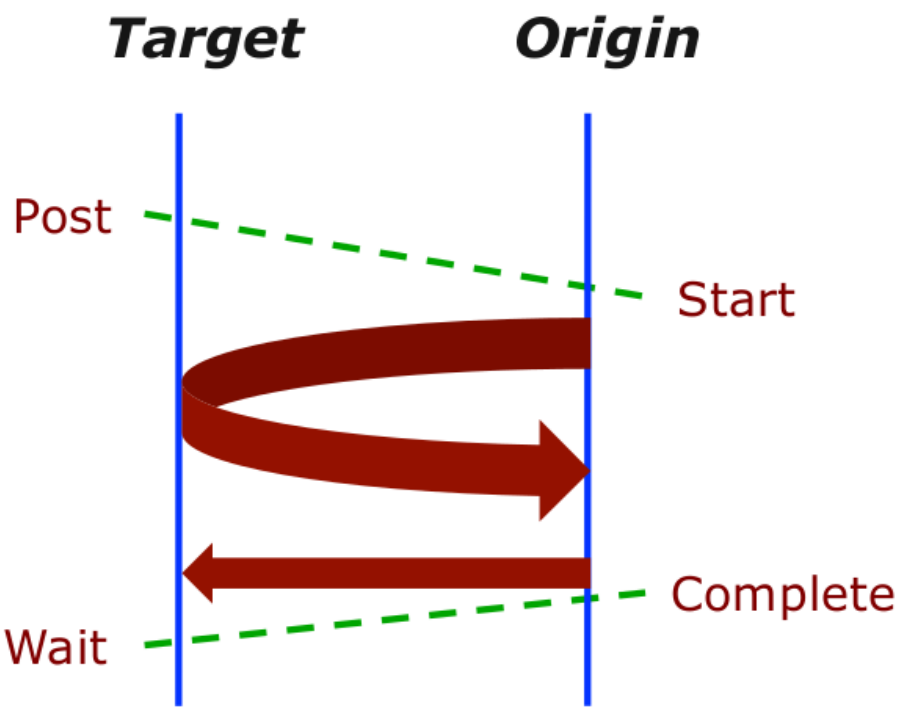
- All processes in group of `win` do an `MPI_Win_fence` to open an epoch
- Everyone can issue put/get/accumulate operations to read/write data
- Everyone does an `MPI_Win_fence` again to close the fence epoch
- All operations complete at the second fence synchronization

One-sided communication: synchronization models: fence



One-sided communication: synchronization models: PSCW

- PSCW = Post/Start/Complete/Wait
- Generalized Active Target Synchronization
- Like fence, but origin and target specify who they communicate with

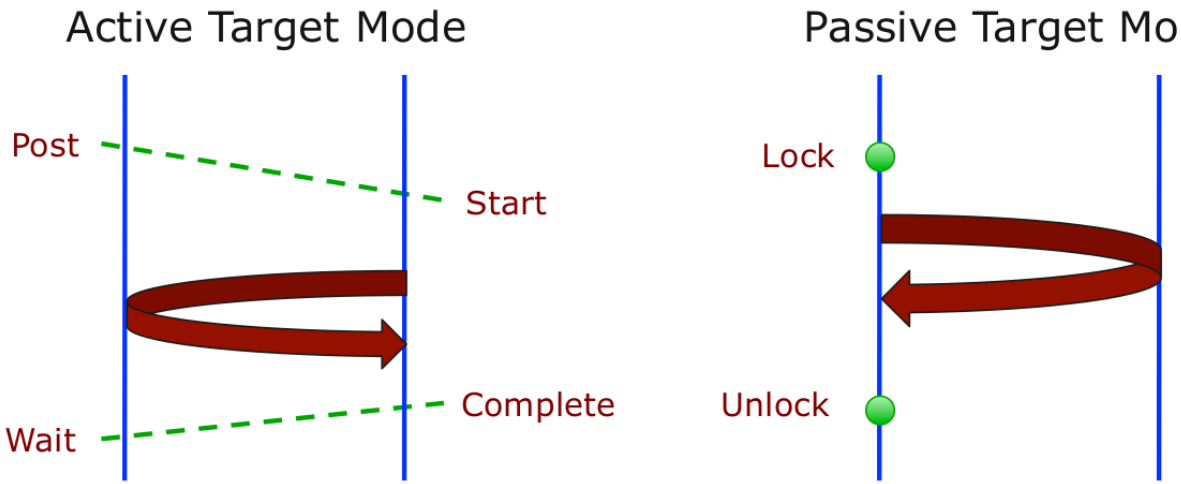



```
int MPI_Win_post(MPI_Group group,
  int assert, MPI_Win win)

int MPI_Win_start(MPI_Group group,
  int assert, MPI_Win win)

int MPI_Win_complete(MPI_Win win)

int MPI_Win_wait(MPI_Win win)
```

- All synchronization operations may block, to enforce P-S/C-W ordering
- Processes can be both origins and targets

One-sided communication: synchronization models: passive target

- What if each process may need to independently access data?



- Passive target synchronization: one-sided, asynchronous communication
- Target does not participate in communication operation
- Shared memory-like model

```
int MPI_Win_lock(int lock_type,
  int rank, int assert, MPI_Win win)

int MPI_Win_unlock(int rank, MPI_Win win)
```

```
int MPI_Win_flush(int rank, MPI_Win win)

int MPI_Win_flush_local(int rank, MPI_Win win)
```

One-sided communication: synchronization models: passive target

- Lock/Unlock: Begin/end passive mode epoch
  - Target process does not make a corresponding MPI call
  - Can initiate multiple passive target epochs to different processes
  - Concurrent epochs to same process not allowed
- Lock type
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently
- `Flush`: Remotely complete RMA operations to the target process: After completion, data can be read by target process or a different process
- `Flush_local`: Locally complete RMA operations to the target process: after completion the user may reuse any buffers provided to put, get, or accumulate on the origin
- Flushes can be done inside epochs
- "Lock" is misleading: not a mutex, just a start and end of an epoch

One-sided communication: synchronization models: advanced passive target synchronization

```
int MPI_Win_lock_all(int assert, MPI_Win win)
int MPI_Win_unlock_all(MPI_Win win)
int MPI_Win_flush_all(MPI_Win win)
int MPI_Win_flush_local_all(int rank, MPI_Win win)
```

- `Lock_all`: Shared lock, passive target epoch to all other processes
  - Expected usage is long-lived: `lock_all`, `put/get`, `flush`, ..., `unlock_all`
- `Flush_all` – remotely complete RMA operations to all processes
- `Flush_local_all` - locally complete RMA operations to all processes

One-sided communication: synchronization models: which to use?

- RMA communication has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc...
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g. InfiniBand)
- Active mode: bulk synchronization
  - E.g. ghost cell exchange
- Passive mode: asynchronous data movement
  - Useful when dataset is large, requiring memory of multiple nodes
  - Also, when data access and synchronization pattern is dynamic
  - Common use case: distributed, shared arrays
- One-sided communication is still relatively new and not always well implemented in MPI, test performance

Hybrid programming: motivation

- MPI describes parallelism between processes, with separate address spaces, on the same or different physical machines
- Thread parallelism provides a shared-memory model within a process on one physical machine
- Using multithreading on a single machine is typically much faster then pure MPI since there is less overhead:
  - No need to maintain buffers on sending and receiving side, no need to copy data to/from buffers
  - Threads can share memory while each process has its own
- In HPC one typically uses OpenMP for multithreading
- Other possible multithreading options: pthreads, C++11 threads, Intel TBB, Intel Cilk Plus,...
- However, multithreading can only work on a single machine and to run on a cluster one can either use pure MPI or combine MPI with some multithreading framework, typicall OpenMP - **hybrid programming**

Hybrid programming: four levels of thread safety

- In MPI+threads hybrid programming, there can be multiple threads executing simultaneously
  - All threads share all MPI objects (communicators, requests)
  - The MPI implementation might need to take precautions to make sure the state of the MPI implementation is consistent
- MPI defines four levels of thread safety - these are commitments the application makes to the MPI
  - `MPI_THREAD_SINGLE`: only one thread exists in the application
  - `MPI_THREAD_FUNNELED`: multithreaded, but only the main thread makes MPI calls (the one that called `MPI_Init_thread`)
  - `MPI_THREAD_SERIALIZED`: multithreaded, but only one thread at a time makes MPI calls
  - `MPI_THREAD_MULTIPLE`: multithreaded and any thread can make MPI calls at any time
- Instead of `MPI_Init` use

```
int MPI_Init_thread(int *argc, char ***argv,
  int required, int *provided)
```

- You ask for `required` level of thread safety, you get - `provided`

Hybrid programming: four levels of thread safety

- An implementation is not required to support levels higher than `MPI_THREAD_SINGLE`; that is, an implementation is not required to be thread safe
- A fully thread-safe implementation will support `MPI_THREAD_MULTIPLE`. There is much more overhead needed from the system to support it and it might be slow. `MPI_THREAD_FUNNELED` seems to be the most optimal choice for most hybrid applications. Benchmark performance before deciding which level of thread safety to use.
- A program that calls `MPI_Init` (instead of `MPI_Init_thread`) should assume that only `MPI_THREAD_SINGLE` is supported
- A threaded MPI program that does not call `MPI_Init_thread` is an incorrect program (common user error) although for many MPI implementations it might actually work fine.

Hybrid programming: MPI-3 shared memory

- MPI-3 one-sided communication allows different processes to allocate shared memory through MPI
  - `MPI_Win_allocate_shared`
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads
- One can use `MPI_Comm_split_type` with `split_type` equal to `COMM_TYPE_SHARED` to find for the given MPI rank all the ranks on the same physical node, it is returned as a new communicator:

```
int MPI_Comm_split_type(MPI_Comm comm,
  int split_type, int key,
  MPI_Info info, MPI_Comm *newcomm)
```

- `key` controls rank ordering in the new communicator.

Timing

- MPI supplies its own timing routine `MPI_Wtime` which gives wall clock time
- You call it before and after an event and subtract the values:

```
t = MPI_Wtime();
// something happens here
t = MPI_Wtime() - t;
```

- This works on a single rank
- If you want to measure global time of a parallel application, you might need to use barriers in front of each `MPI_Wait` call
- `MPI_Wtick` gives the smallest possible timer increment

Random practical tips

- Do not ever build your program with one MPI implementation/version and run it with another! Most likely it will not work or work incorrectly.
- If MPI is compiled with a certain compiler, it is desirable to use the same compiler for your application and all the dependencies. Typically, when you load a module it is taken care of automatically but might be accidentally overwritten by subsequent module loads
- A typical mistake: first load some MPI and then some python with mpi4py. In that case, you would be using MPI implementation supplied with mpi4py instead of MPI you intended to use. Always check with "which mpirun" what you are actually using

Random practical tips

- mpirun/mpiexec and mpicc/mpiicc/mpic++/mpiicpc/mpif77/mpif90/mpifort have different options and might even be called differently for different MPI implementations.
- For example, openmpi has radically different command line syntax then Intel MPI and other mpich based implementations
- As far as command line options are concerned, Intel MPI is documented the best and is typically one of the best implementations to use, you might want to start with it
- To debug/profile MPI/OpenMP program use Arm (former Allinea) ddt/map. Just for profiling, Intel Parallel Studio comes with VTune, Intel Advisor, Intel Trace Analyzer and Collector, vectorization reports options to compiler, etc. Score-P is another good profiling tool.

References

http://mpi-forum.org
http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
https://www.mpich.org
https://www.mpich.org/static/docs/v3.2/www3/
http://mvapich.cse.ohio-state.edu
https://software.intel.com/en-us/intel-mpi-library
https://www.open-mpi.org
http://mpitutorial.com
https://computing.llnl.gov/tutorials/mpi/
https://hpc.llnl.gov/sites/default/files/DavidCronkSlides.pdf
http://wgropp.cs.illinois.edu/courses/cs598-s16/
https://bitbucket.org/VictorEijkhout/parallel-computing-book/src

"Using MPI: Portable Parallel Programming with the Message-Passing Interface", third edition,
by William Gropp, Ewing Lusk, Anthony Skjellum

"Using Advanced MPI: Modern Features of the Message-Passing Interface", by William Gropp, Torsten Hoefler,
Rajeev Thakur, Ewing Lusk

HDF5 page:
https://www.hdfgroup.org

Tutorials on demand

- Currently the following tutorials are available at any time:
  - Introduction to Linux and RCC
  - Introduction to Python
  - Distributed MPI programming in C/C++/Python
  - Multithreading programming in OpenMP
  - Introduction to Hadoop
  - BigData and Machine Learning with Spark and BigDL
  - Deep Learning with Keras and TensorFlow
  - Deep Learning with Theano
  - Deep Learning with PyTorch
  - Singularity Container
- More are coming: Sequence Models with Deep Learning, Reinforcement learning, Deep Learning autoencoders for dimensionality reduction...
- Requests for other topics?