Parallel programming

MPI 2

Today's topic

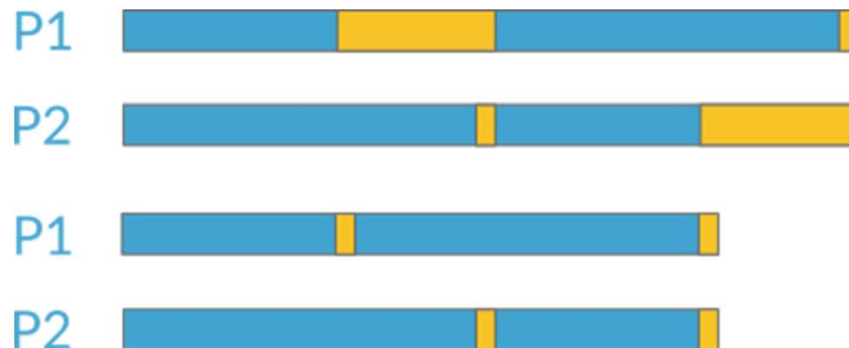
- From the previous seminar we know
  - General principle of MPI
  - Synchronous communication (point to point, collective)
- Questions:
  - Can we AVOID IDLING of point-to-point communication?
  - Is there any way how to create our own datatypes?
  - Can we create any network topology from lectures?
  - What if we don't know the size of message?

Synchronous communication

- At the moment, we have only seen **blocking** point-to-point communication.
  - After sending a message the process has to wait **(Delay)**
  - Process waits until it receives the message **(Delay)**
- Both send and receive operations use buffers
  - Send waits until the data are copied from the buffer (send started, send finished)
  - Receive waits until the data are copied them to the buffer (receive started, receive finished)
- Waiting for the calls causes IDLING in communication.
- On the other hand, buffer can be used right after the operation.

Asynchronous communication

- Copying to/from the buffer is usually much slower than own computation of the process
- We can continue in own computation, and when we want to use buffer again, we can check, if the copying is finished
  - **Check** if the communication operation is finished



Blocking case

Non-blocking case

MPI Isend

- int **MPI_Isend**(void* buf,
  int count,
  MPI_Datatype datatype,
  int dest,
  int tag,
  MPI_Comm comm,
  MPI_Request *request)
- All parameters are the same as of function MPI_Send exept:
  - Request -> MPI reqeuest object (stores information about the communication operation)
- An MPI_Isend creates a **send request** and returns a **request** object.
- It may or may not have sent the message, or buffered it. The caller is responsible for **not changing the buffer** until after waiting upon the resulting request object.

MPI Irecv

- int **MPI_Irecv**(void* buf,
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm comm,
  MPI_Request *request)
- All parameters are the same as of function MPI_Recv exept:
  - Request -> MPI reqeuest object (stores information about the communication operation)
  - See that MPI_Status is missing
- An MPI_Irecv creates a receive request and returns a receive request in an MPI_Request object.
- The caller is responsible for **not changing the buffer** until after waiting upon the resulting request object

Checking if communication is finished

- int **MPI_Wait**(MPI_Request *request,
  MPI_Status *status)
- An MPI_Wait call waits for completion of the operation that created the request object passed to it.
  - For a send, the semantics of the sending mode have been fulfilled
  - For a receive, the buffer is now valid for use
  - Implicit for blocking send and receive opperations
- int **MPI_Test**(MPI_Request *request,
  int *flag,
  MPI_Status *status)
- An MPI_Test call returns immediately a **flag** value indicating whether a corresponding MPI_Wait would return immediately.
  - Flag is 1 if request has been completed
  - Flag is 0 if request has not been completed
  - Usefull for **bussy waiting** loops

Task

- Write program with 2 MPI processes. One will sleep for 5 seconds and then recives data, other will send data and then go sleep for 5 seconds. Use blocking and non-blocking communications and compare times.
- Write program with multiple MPI processes where process 0 has array of chars. Other processes will dynamically requesting for data. All processes are using loops. In the beginning of the loop process 0 sleeps for 2 seconds. Other processes sleeps for 1 second. We are searching for char “E” => When some process founds it, program is finished.

Custom types

- As you might have noticed, all datatypes in MPI communications are atomic types.
- Sometimes, it might be useful to create higher-level structures
  - MPI allows us to do that in the form of **derived** or custom datatypes.
- A datatype can be defined easily by specifying a sequence of couples. Each couple represent a block : (type, displacement).
  - Type could be atomic or also derived
  - Displacement indicates the offset in bytes in memory
- There are multiple types how to create or use own datatypes in MPI
  - One way is to serialize your datatypes into a **byte array** and send it as MPI_BYTE array
  - You can also create your own MPI_Datatype

MPI Structures

- int **MPI_Type_create_struct**(int count,
  const int *block_length,
  const MPI_Aint *displacement,
  const MPI_Datatype *types,
  MPI_Datatype *new_type)
  - **Count** -> number of elements
  - **Block_length** -> number of contiguous elements of that type
  - **Displacement** -> array of address offsets in the custom datatype (Aint = address integer)
  - **Types** -> array of all the different sub-types we are going to use in the custom type
  - **New_type** -> resulting datatype
- We can create our own MPI structure, when we know all datetypes in original structure and their offsets.
- Creation of structe must be followed by **MPI_Type_Commit**
- When you are working with structure consist of same datetypes you can represent your structure as a vector using **MPI_Type_contiguous**

Example of structure creation

```
typedef struct car_s {
        int shifts;
        int topSpeed;
} car;
const int nitems=2;
int          blocklengths[2] = {1,1};
MPI_Datatype types[2] = {MPI_INT, MPI_INT};
MPI_Datatype mpi_car_type;
MPI_Aint     offsets[2];
offsets[0] = offsetof(car, shifts);
offsets[1] = offsetof(car, topSpeed);
MPI_Type_create_struct(nitems, blocklengths, offsets, types, &mpi_car_type);
MPI_Type_commit(&mpi_car_type);
```

Task

- Write program with 4 MPI processes.
- Prosess 0 reads n lines form file representing information about cars
  - Representing Euro Car Segment (ECS) of the car (char)
  - Price of the car (int)
  - Year of fabrication of the car (short)
  - Average fuel consumption of the car (float)
- After reading the input, process 0 broadcast the data to other processes
- Process 0 will return the count of cars in ECS “C”
- Process 1 will return the average fuel consumption of all ECS
- Process 2 will return all cars older than 6 years
- Process 3 will return average price of cars in ECS “C” or lower.

Probing incoming communications

- The amount of data can be really big -> optimizing the size of the messages sent have a real influence on the performance of the system.
  - **1. Try to group as many data as possible in one communication.**
  - **2. Try to send the exact amount of data you are storing in your buffer and no more.**
- **Probing** the message = asking MPI to give you the size of the message.
  - Information of the message is stored in **MPI_Status**
  - Getting the count of elements we are about to receive
  - Getting the **ID** and **tags** of the processes we are receiving from
    - We can use MPI_ANY_SOURCE and MPI_ANY_TAG
- Probing only informs that the process is ready to receive a communication
  - Use MPI_Get_Count on the received status to retrieve the information we want

MPI_Probe and MPI_Iprobe

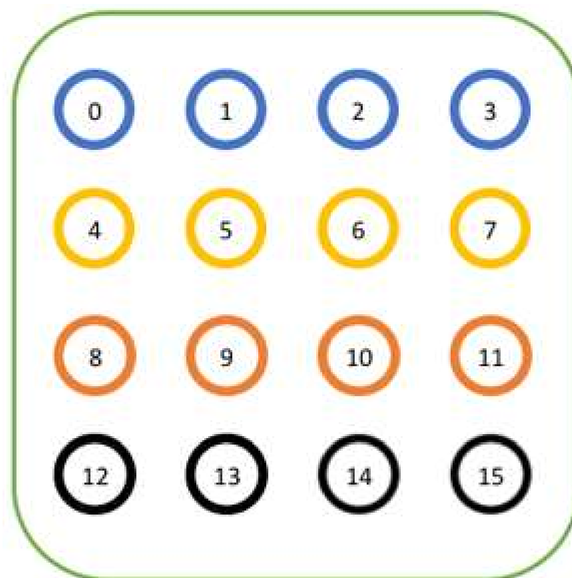
```
int MPI_Probe(  int source,
                int tag,
                MPI_Comm comm,
                MPI_Status *status)
```

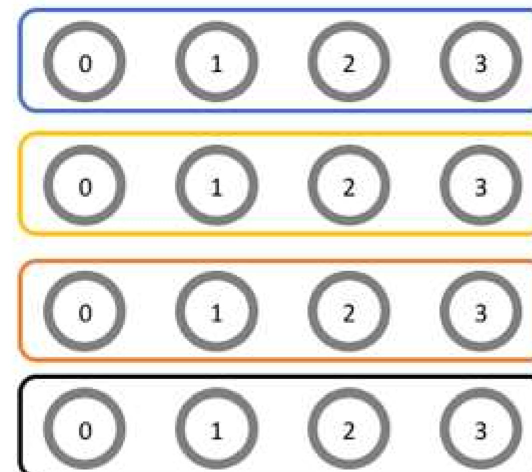
```
int MPI_Iprobe( int source,
                int tag,
                MPI_Comm comm,
                int *flag,
                MPI_Status *status)
```

- Allow checking of incoming messages
- The user can then decide how to receive them, based on the information returned by the probe in the status variable.

Communicator Management

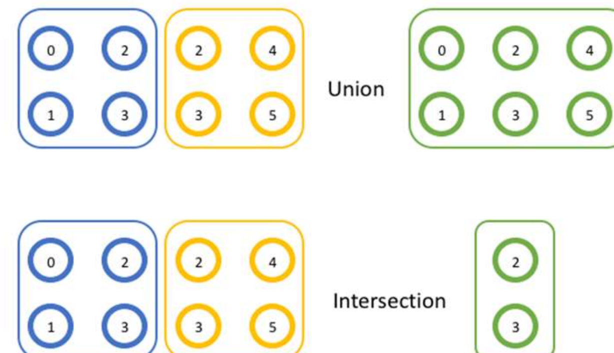
- At the start of an MPI program all its processes belong to the communicator **MPI_COMM_WORLD**.
- In many application we want to partition processes into n subgroups forming separate communicators (intra-communicator).
- Intra-communicator
  - Set of all processes which share that communicator
  - Collective and point to point communications can be performed with an intra-communicator





MPI Groups

- Group is set of precesses in the communicator.
- MPI uses these groups in the same way that set theory generally works.
  - MPI provides Union and Intersection operations on groups
- int **MPI_Comm_group**(MPI_Comm comm, MPI_Group* group)
  - Creates new MPI_Group group in communicator MPI_Comm comm
- int **MPI_Group_incl**(MPI_Group group, int n, const int *ranks, MPI_Group* newgroup)
  - contains the processes in group with ranks contained in ranks, which is of size n
- int **MPI_Group_union**(MPI_Group group1, MPI_Group group2, MPI_Group* newgroup)
  - Newgroup -> Union of group1 and group2
- int **MPI_Group_intersection**(MPI_Group group1, MPI_Group group2, MPI_Group* newgroup)
  - Newgroup -> Intersection of group1 and group2



Communicator Constructor

- int MPI_Comm_create_group(MPI_Comm comm,
  MPI_Group group,
  int tag,
  MPI_Comm * newcomm)
  - Comm -> communicator (handle)
  - Group -> group, which is a subset of the group of comm (handle)
  - Tag -> safe tag unused by other communication
  - Newcomm -> new communicator (handle)
- Requires that comm is an intra-communicator.
- Returns a new intra-communicator, newcomm, for which the group argument defines the communication group.
- No cached information propagates from comm to newcomm.
- Each process must provide a group argument that is a subgroup of the group associated with comm

Splitting Communicator

- int **MPI_Comm_split**(MPI_Comm comm,
  int color,
  int key,
  MPI_Comm *newcomm)
  - Comm -> Communicator
  - Color -> ontrol of subset assignment
  - Key -> control of rank assigment
  - Newcomm-> new communicator
- Partitions the group associated with comm into disjoint subgroups, one for each value of color.
- Within each subgroup, the processes are ranked in the order defined by the value of the argument key.