

Parallel programming

MPI 2





Today's topic

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



Synchronous communication

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



Asynchronous communication

- Copying to/from the buffer (yellow color) is usually much slower than own computation of the process (blue color)
- 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



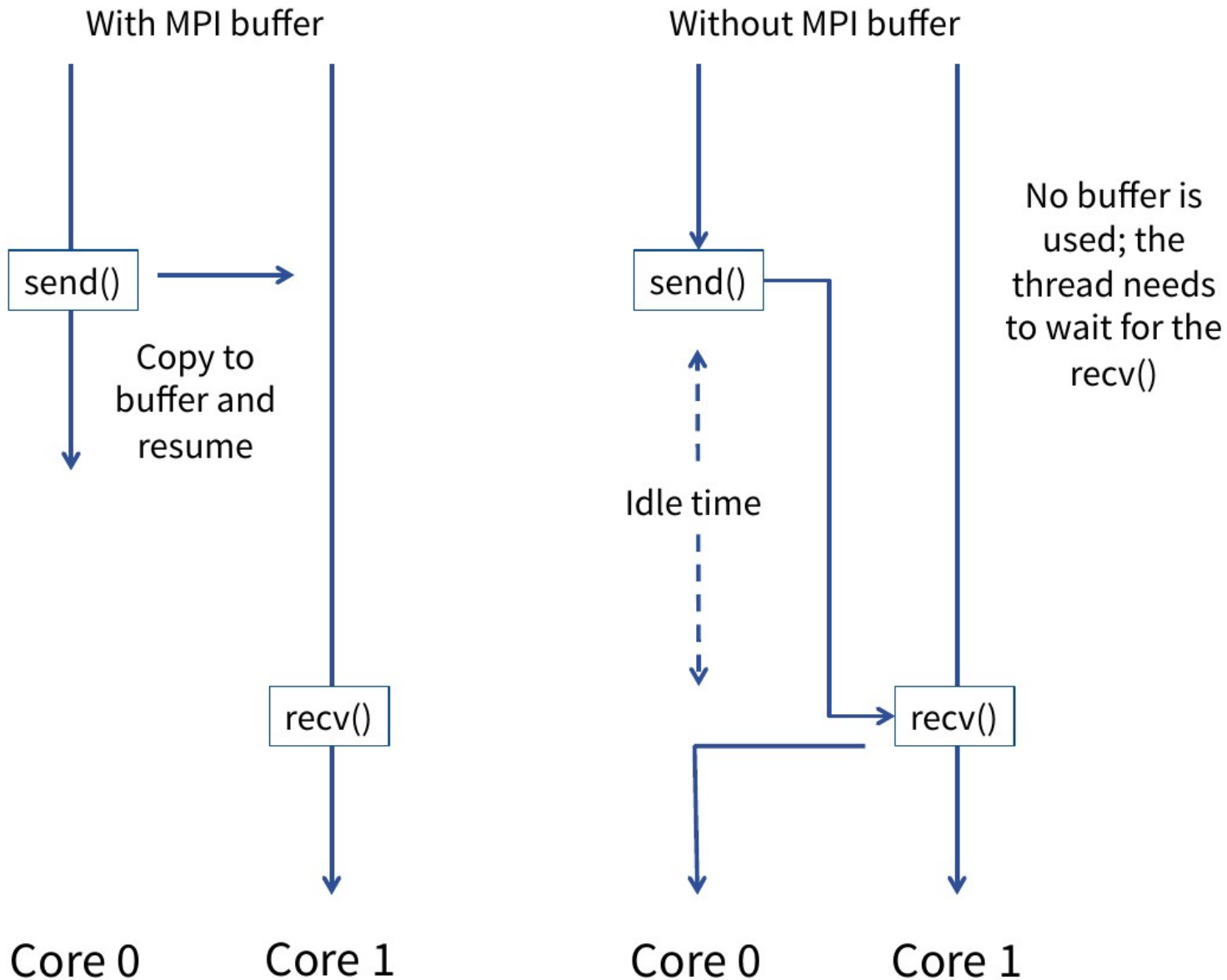
Blocking case



Non-blocking case



Non-blocking and Blocking





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` except:
 - Request -> MPI request 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 except:
 - Request -> MPI request 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 operations
- `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
 - Useful for **bussy waiting** loops

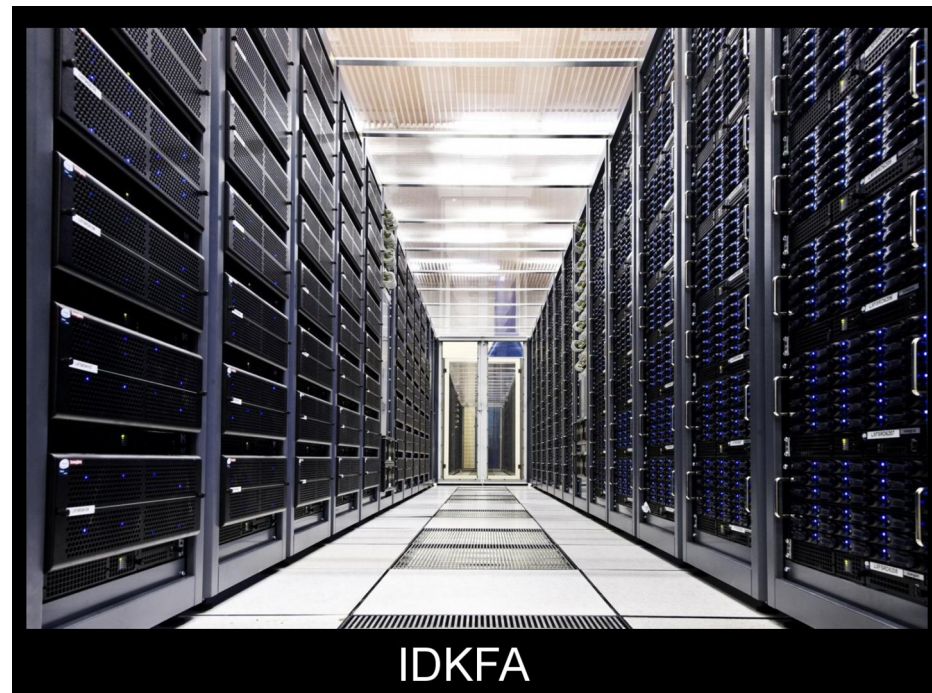
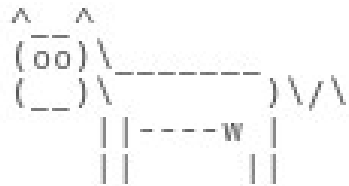


Task

AsyncSendAndReceive.cpp

- Write a program which sends short message “IDKFA” in **non-blocking way** from one process to another one and prints the result.

< IDKFA >



IDKFA



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 datatypes in original structure and their offsets
- Creation of structure must be followed by **MPI_Type_Commit**
- When you are working with structure consist of same datatypes you can represent your structure as a vector using **MPI_Type_contiguous**
- See **CustomTypeDemo.cpp**



Custom operation

- **MPI_Reduce** is a collective operation for data reduction using specific reduce operation
- **MPI_Reduce** supports custom operations to perform reductions other than basic mathematical operations
- Custom operations are user-defined and allow **flexibility in data reduction**
- Custom operations enable performing operations **on user-defined data types**



Custom operation in MPI

- `int MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)`
 - `user_fn` -> User-defined function for the operation
 - `commute` -> A flag indicating commutativity
 - `op` -> MPI_Op object for storing the created operation
- A custom reduction operation is a user-defined function for reducing data
 - It can be applied with **MPI_Reduce** to combine data from multiple processes
- **MPI_Op_create** is a function in MPI for creating custom reduction MPI_Op operations
- See **CustomOperationDemo.cpp**



Task

BestStudent.cpp

- Write a program for finding best student at school
 - Use provided struct Student and find_best_student function
 - Create custom MPI_Datatype Student based on struct Student
 - Create custom MPI_Op for data reduction
 - Scatter student data generated by process 0
 - Each process finds its best student using find_best_student function
 - Use MPI_Reduce to get best student at process 0
 - Print the best student
 - Follow the provided guidelines



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

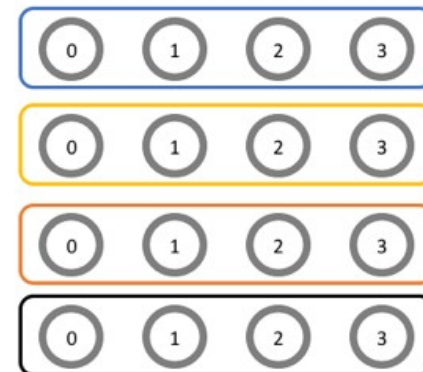
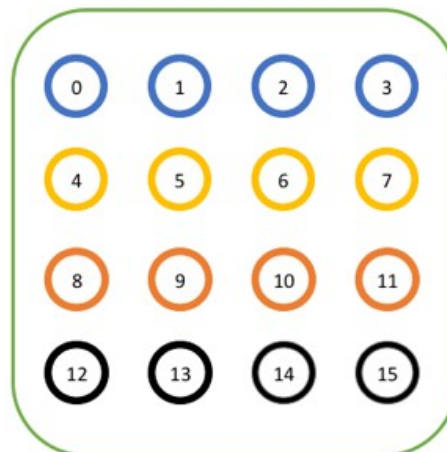
- `int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)`
- Allow checking of incoming messages
- The user can then decide how to receive them, based on the information returned by the probe in the status variable.
- See **ProbeMessageDemo.cpp**





Communicator Management

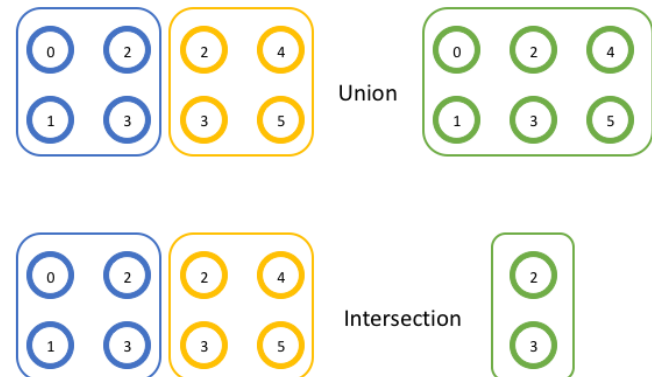
- At the start of an MPI program all its processes belong to the communicator **MPI_COMM_WORLD**
- In many application we want to partition processes into **n subgroups** forming separate communicators (intra-communicator)
- Intra-communicator
 - Set of all processes which share that communicator
 -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





MPI Groups

- Group is set of processes inside 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



Communicator Splitting

- `int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)`
 - **Comm** -> Communicator
 - **Color** -> control of subset assignment
 - **Key** -> control of rank assignment
 - **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`
- See **CommunicatorDemo.cpp**



Task

MasterSlave.cpp

- Write a program for master slave communication
 - Master will be process with rank 0
 - Master creates a integer array Tasks of integers greater than 0
 - Master makes sets element in Tasks to -1
 - Slave will ask master for one task from Tasks
 - Master must have an incrementing index and sends task at index position
 - If received element is not -1 then slaves sleeps for task value seconds and sends work request
 - If received element is -1 then slaves sends end request
 - After master received end request from some slave it terminates all slave
 - Slave can be terminated only from master
 - Master is terminated automatically after all slaves are terminated