NTNU – Trondheim
Norwegian University of
Science and Technology

Communicators

Jan.Christian.Meyer@ntnu.no

Today’s topic

- As we said in the very beginning of this MPI journey

  (and used in every program thereafter)

  the ranks that participate in MPI communication are all members of an MPI_Comm *(...unicator)*
- These can be created and manipulated in a couple of ways, we will examine two
  - By choosing arbitary groups of ranks
  - By arranging them in a general graph structure
- We’ll arrange them in rectangular grids next lecture

Choosing arbitrary sets of ranks

- An MPI_Group is just a set of ranks
- There is a set like this associated with every communicator
- It doesn’t contain any contact information, but we can use them to partition our collective into subsets

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

fetches the group of ranks in the communicator

```
MPI_Group everyone;
MPI_Comm_group ( MPI_COMM_WORLD, &everyone );
// This gets us a group that contains all the ranks
```

Including ranks in groups

- We can make them up from existing groups
- The ingredients are
  - The group we have chosen as a basis
  - An integer member count
  - A list of that many integers representing ranks in a group we already have
  - A pointer to the new (sub-)group we want to create

```
int MPI_Group_incl (
        MPI_Group old_group,
        int number_of_members,
        const int rank_list[],          ← ‘number_of_members’ long list
        MPI_Group *new_group
);
```

Excluding ranks from groups

- This one wins no points for originality, I’m sure you can guess what it does

```
int MPI_Group_excl (
        MPI_Group old_group,
        int number_of_rejects,
        int ranks_to_remove[],        ← ‘number_of_rejects’ long list
        MPI_Group *new_group
);
// The new group contains all the ranks of the old one, except for
//   the ones listed by arguments 2 and 3
```

Set operations: union

- When we have some groups that we want to join, they can be merged together

```
int MPI_Group_union (
        MPI_Group g1,
        MPI_Group g2,
        MPI_Group *new_group
);
```

- The new group will contain exactly one copy of each distinct rank from g1, g2

Set operations: intersection

- If some groups contain some of the same ranks, we can make one out of only the ranks that they share

```
MPI_Group_intersection (
    MPI_Group g1,
    MPI_Group g2,
    MPI_Group *new_group
);
```

- The new group will contain only ranks that are contained both within g1 and g2

Why this additional group concept?

- Sets of ranks *are* remarkably close to what we've called communicators
- Since they don't contain contact information, though, ranks that aren't members of a group can still put it together
  - They're just bags of numbers, in a sense
- It's all part of allowing for most of the code to be collectively executed

Making an MPI_Comm

- When all your groups are collected and ready, they can be made into communicators:

```
int MPI_Comm_create (
    MPI_Comm old_communicator,        ← Communicator to start with
    MPI_Group group,                  ← Subset of ranks in it
    MPI_Comm *new_communicator        ← The new communicator
);
```

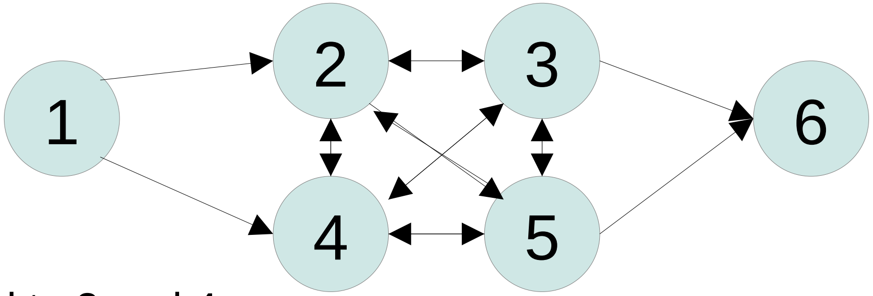
- When a rank that is a member of the group makes this call, it gets a communicator handle back
- When a rank that is **not** a member of the group makes this call, it gets MPI_COMM_NULL back

I have prepared an example...

- In the example code archive, the directory “01_group_communicator” contains a program that
  - Gets the group from the world comm
  - Divides them into two subgroups: odd and even world-ranks
  - Creates communicators out of both
  - Has one rank report back on
    - which half it became a member of, and
    - what its rank is *among that subset* (not the same as its world-rank)
- World-rank 0 reports by default, unless another one is given as the first command line argument
- The program doesn’t do anything useful, but it’s a demonstration of one way in which we can partition the world comm.

Graph communicators

- All this juggling of groups and their members can be circumvented if all you want is some typical rearrangement of all the ranks in a communicator
- One typical (and very general) construction is to arrange them as a directed graph
- Here’s a (hypothetical) 6-way example of that kind of thing:



Rank 1 can send to 2 and 4
Ranks 2-5 can all collaborate
Ranks 3 and 5 pass the result to rank 6

That example is contrived

- True.
  - We could imagine a pipeline where 1 feeds input into a fully connected cluster and 6 receives the output, but we’d typically want to let 1 talk to all of 2-5, and 6 to receive from all of them
  - The drawing comes out clearer this way, though
- Since we can do arbitrary graphs, we can just make a more elaborate example out of a familiar structure
  - A binary tree is a particular case of a graph
  - Let’s make a binary tree communicator
  - Code in example subdirectory “02_general_graph”

The heart of the matter

- This call creates a graph communicator out of another communicator, by just imposing the graph structure:

```
int MPI_Graph_create (
        MPI_Comm old_communicator,          ← Easy
        int number_of_nodes,                ← Easy
        const int index[],
        const int edges[],
        int reorder,                        ← Easy
        MPI_Comm *new_communicator          ← Easy
);
```

- Most of this is straightforward
  - “reorder” says whether or not MPI is allowed to give ranks in the new communicator different numbers, or whether it must keep the old values
  - 0 means don’t reorder
  - Not 0 means it’s ok to reorder (but it’s not an obligation)

The remaining two arguments

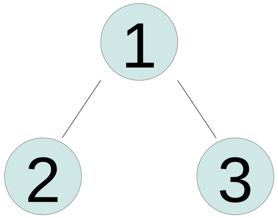
- ‘index’ and ‘edges’ are just some linear lists of integers
- Sizing and contents can be a bit finickety, though
- That’s why I’d like to illustrate them using one particular graph topology (*i.e.* the binary tree)

Indices in trees

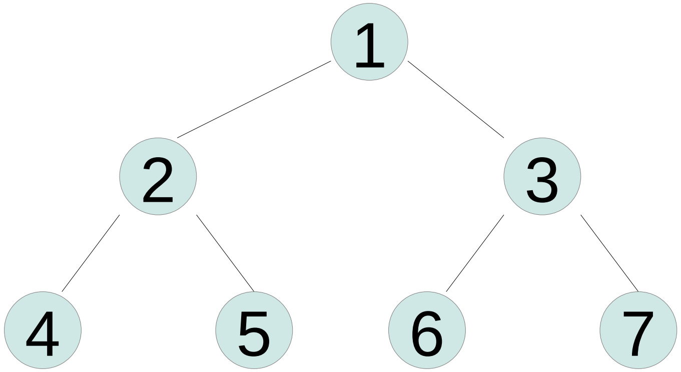
- In order to map ranks onto tree nodes, we'll need some schema for which rank goes where
- Let's use one which behaves like the indexing in a "heap" *data structure*
  - Usually covered in Algorithms&Data structures, but we'll repeat it
  - Mind that this is **not** the same thing as the "heap memory" we've been talking about
- This is not the only way to number tree nodes, but it's simple.

Top-to-bottom and left-to-right

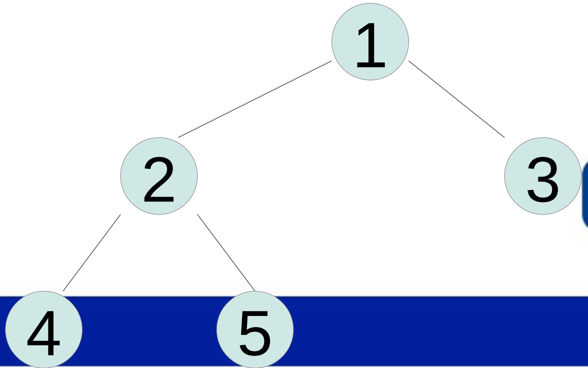
- Two levels:



- Three levels:



- Three levels, incomplete:



Most nodes have at most 3 neighbors

- Node 1 has at most 2 (left/right children)
- All other nodes have at least 1 (parent)
  - Optionally, they have a left child
  - Also optionally, a right child
- Here are the node-indices of all three, for node n>1

...if we assume that integer division truncates decimals…

parent = n/2

left_child = n*2

right_child = n*2+1

Translating for MPI ranks

- Those formulas work when we number tree nodes from 1
- MPI ranks begin at 0
- Therefore, the calculation becomes

  parent = (rank+1)/2 - 1

  left_child = (rank+1)*2 - 1

  right_child = (rank+1)*2

  instead:

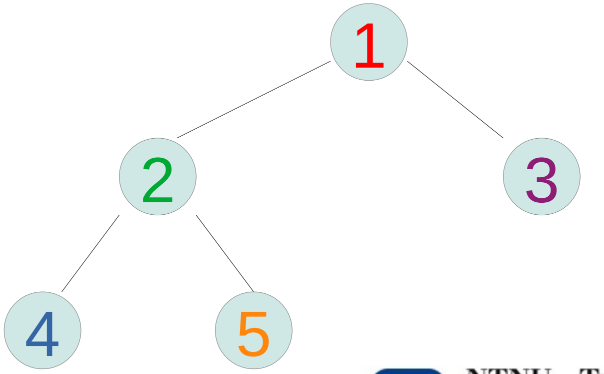
  - Add one to rank in order to get treenode indices
  - Subtract one afterwards in order to get back to rank numbers

The edges list

- The list of edges is just a sorted list of neighbor ranks for each rank
- Graph communicators are directed graphs
  - to make them undirected, we'll just add edges in both directions
- For our incomplete 3-level tree, it works out like this:

[2, 3, 1, 4, 5, 1, 2, 2]

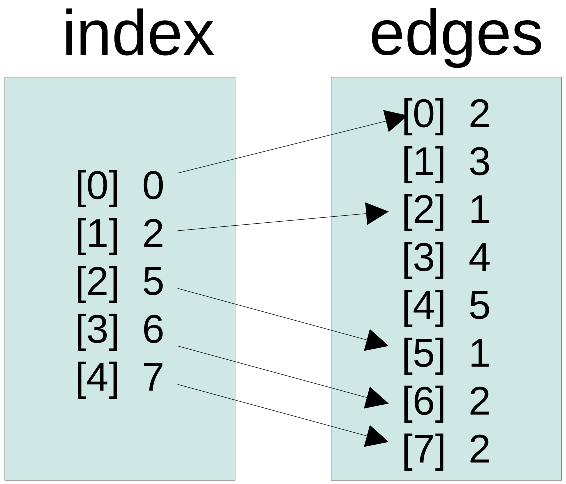
(The color coding indicates whose neighbors are listed where)



NTNU – Trondheim
Norwegian University of
Science and Technology

The edges list is a mess

- It’s not regularly ordered, because nodes can have different numbers of neighbors
- This is where the index list comes in
  - We can imagine one like this:



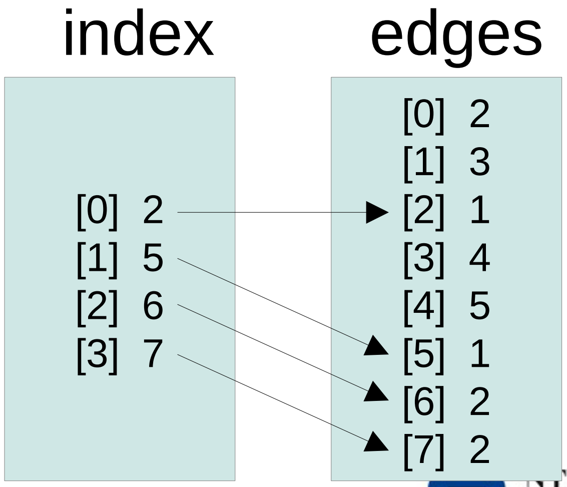
index	
[0]	0
[1]	2
[2]	5
[3]	6
[4]	7

edges	
[0]	2
[1]	3
[2]	1
[3]	4
[4]	5
[5]	1
[6]	2
[7]	2

  - This would give us the start of each rank’s neighbor list by its entry in the index list

That’s mildly redundant

- The first entry would always be 0
  - So we can skip it
- Each entry of the actual index list contains the sum of neighbors from all preceding ranks instead:



index

[0] 2
[1] 5
[2] 6
[3] 7

edges

[0] 2
[1] 3
[2] 1
[3] 4
[4] 5
[5] 1
[6] 2
[7] 2

(This is a ***prefix sum*** *operation on an array. You can implement it in parallel using MPI_Scan, if you want.*
The example code does it with a loop, because we’re honestly juggling enough indices already.)

Those were all the arguments

- Once we’ve configured the graph layout, actually creating the communicator is just a single call
- The communicator embeds all the neighborhood-info inside
- Once you’ve created it, you no longer need to juggle all the indexing logic
- The example code demonstrates this by passing the result to a function which recovers the structure from the communicator.

Usage

- In order to visualize that we got it right, the example code ends by having each rank print its neighbor information into its own text file
  - Using the notation “A -> B” to say that there’s an edge from A to B
  - This notation is used by the ‘dot’ graph plotting program
  - It’s part of a package called GraphicsMagick, which you can install at home
- After the MPI program has created all the partial text files, you can run ‘make graph.png’
  - It concatenates all the text into a common ‘graph.dot’ file
  - It then passes the input to ‘dot’, and obtains a picture