NTNU – Trondheim
Norwegian University of
Science and Technology

Concepts of MPI

Jan.Christian.Meyer@ntnu.no

Today's topic

- MPI is quite rich in abstractions and mechanisms for various purposes
  - Many of them interact
- Before we look at the details of each, it may be helpful to briefly survey the whole
  - Otherwise, some functions will have arguments that make no sense until their purpose has been mentioned
- Today, we'll just cover a number of the general ideas, and then we can start digging into the details next time

Message Passing Interface (MPI)

- MPI is a *standard specification* for a (large) number of function calls, and what they're supposed to do
- It aims to avoid specifying *how* to do things, the idea is that
  - HPC vendors can supply implementations that fit their specific machines, and
  - your code will be runnable on all of them, just at different speeds
- It's been around since 1994
  - many implementations have come and gone
  - the ones we have now may disappear, and new ones will emerge

MPI runs parallel processes

- Multiple copies of the program are started through a launcher-program that tracks how to make connections between them
  - “mpirun”, “mpi_exec”, “srun”, or similar, depending on your platform
- When you send data from one process to another, it gets its own copy
  - If you want a number to be consistent between all processes, your code has to take care of it

MPI is SPMD
(Single Program Multiple Data)

- This four-letter acronym is not due to Flynn
  - It's more of a programming style than it is a type of machine
- The Wonderful Idea™ is that P copies of the same program can do P different things if they have an identity-number that sets the copies apart
- MPI calls this number the *rank* of a process

Communication is two-sided (mostly)

- If the process with rank 1 contains a statement like

  "take these 4 numbers and send them to rank 2"

  then the process with rank 2 must contain a statement like

  "receive 4 numbers from rank 1, and put them in this 4-element buffer I have prepared"

- Every send must have a matching receive at the other end
  - Sends that go nowhere may stop your program, or crash it later
  - Receives that never receive anything stop your program

The SPMD view

- Even though we want ranks 1 and 2 to do different things, we can merge their actions into 1 program
- This is a pseudo-code rendition of the principle:

```
rank = who_am_i ()
if ( rank == 1 )
    four_numbers = [1,2,3,4]
    send ( four_numbers, 4, rank_2 )
else if ( rank == 2 )
    four_numbers = [0,0,0,0]
    recv ( four_numbers, 4, rank_1 )
```

That's impractical

- Admittedly, yes
  - but only kind of
- In the extreme case that no two ranks ever do the same thing, we might
  - write P versions of the program
  - concatenate them all in one file
  - wrap each version in an if-statement that checked for a specific rank number

  but matching up all the sends and receives would quickly create a hot mess.
- How do we even plan how many ranks to have in total?
  - The P-versions approach would hardwire the code to run with a specific number of processes
  - No more
  - No less

MPI Communicators

- In addition to obtaining its own rank number, each process starts as a member of a “communicator”
  - By default, there’s always one that contains every process we launched
  - MPI calls it the “world” communicator
- Programs can find out how big it is
- For a p-process communicator, the ranks inside are always numbered from 0 to p-1

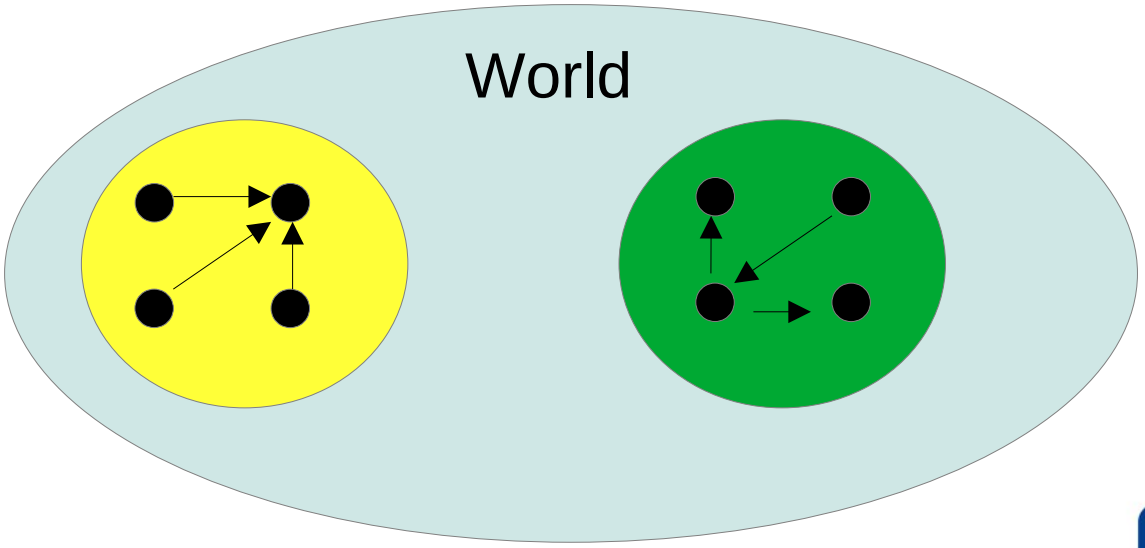
The SPMD thing again

- Armed with the size of the world communicator, we can, *e.g.* make a program that will send a message from the first to the last rank, regardless of the last rank's actual number
- Here's the updated pseudo-code:

```
rank = who_am_i ()
size = how_many_are_we ()
if ( rank == 0 )
    four_numbers = [1,2,3,4]
    send ( four_numbers, 4, size-1 )
else if ( rank == size-1 )
    four_numbers = [0,0,0,0]
    recv ( four_numbers, 4,  0 )
```

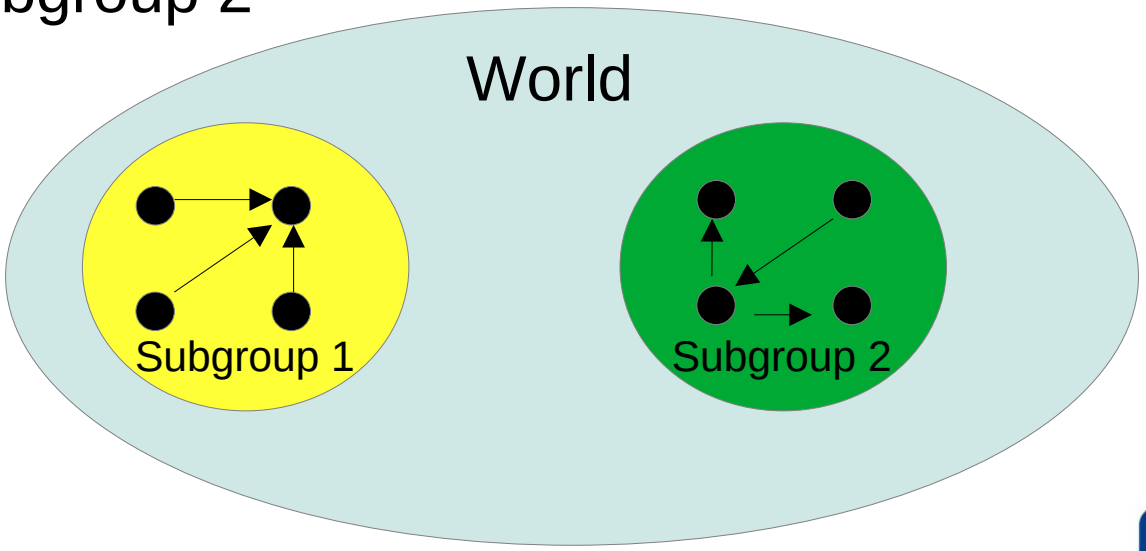
Other communicators

- If you want, you can divide the world communicator into sub-groups, to let some processes collaborate on one thing, and the rest to work on something else



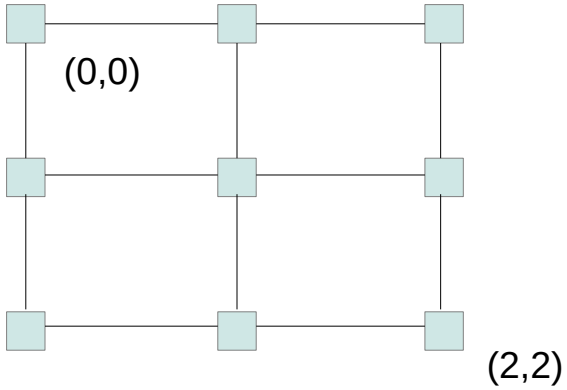
Messages associate with a communicator

- In order to send and receive messages within a group, we must know its size and our rank within it
- The process that has rank 5 in world can have rank 0 in subgroup 2



Virtual topologies

- The world communicator has no internal structure, everyone just gets a number for rank
- MPI lets you declare communicators that have structure, *e.g.* the *Cartesian* flavor, where every rank has a set of coordinates:



- This way you can send/receive messages with “rank at (1,1)” instead of having to calculate an indexing scheme yourself
- We can get communicators shaped like arbitrary graphs as well, but this rectangular thing is common

Communication modes

- The point-to-point messages can be sent with 4 different guarantees for how they are transmitted
  - Standard

    (Whatever your implementation has as default, more on this later)
  - Synchronized

    (Send-function will not return until reception is acknowledged)
  - Buffered

    (Explicitly manage the memory that's used for sending/receiving)
  - Ready

    (Assume that the receiver has already initiated the receive)

Non-blocking communication

- Usually, send and receive operations cause the program to stop and wait for the message to come through, and only resume the program afterwards
  - This is not **100%** true, but close enough for now
- Non-blocking sending and receiving immediately returns a *request* instead, so that you can continue calculating
- In order to make sure that the message has gone/come through, you must issue a wait-for-completion call for the request later on
  - Whenever you can no longer proceed without the comms being complete

Collective operations

- In order to save on the amount of point-to-point message passing, MPI offers a set of operations that every rank in a communicator must call before it completes
  - Things like broadcasting values, finding a maximum value between all ranks, global sums, etc.
- These don't require separate branches in code

```
rank = who_am_i ()
the_number = 0                          // Every rank sets 0
if ( rank == 0 )
    the_number = 42                     // Rank 0 sets something different
broadcast ( the_number, rank_0 )  // Everyone gets what rank 0 has
```

Scattering and gathering

MPI has some collective operations dedicated to

- splitting some large data set in equal parts, and distributing them among ranks
- receiving a number of equal parts, and putting them back together into a large data set

- This is a fairly common thing to want
- Kind of like specialized broadcasting operations
  - but only almost

Synchronization

- MPI has a *barrier* mechanism that makes all ranks in a communicator wait until everyone has reached the barrier
- It doesn't actually guarantee that communication is finished (because we can do background communication), but it lets you know that all ranks have reached a given statement
- Most valuable use: make sure that everyone is in the same place before you start measuring the execution time of what comes next

Derived data types

- The basic assumption of sending and receiving is that the message will be laid out as a contiguous array with equally sized elements
- This is not convenient if you want to send a data structure that has different types of variables inside
- It is also not convenient if you want to send *e.g.* a column from a 2D array that is stored in row-major order
- ...or indeed, a rectangular sub-array from it…
- *etc. etc.*

Derived data types

- MPI understands a handful of built-in data types
  - Integers, floating point numbers, characters, that sort of thing
- It lets you combine these into more elaborate structures
  - The outcome is a recipe for how to space out a non-uniform bunch of data
- Sending calls will automatically marshal the data type from its memory layout into a contiguous buffer
- Receiving calls will automatically un-marshal the contiguous buffer and recreate the memory layout at the receiving end

Parallel I/O

- When some data are distributed across a number of processes, the obvious way to collect them in a file is to appoint one of them as a kind of file-master, and
  - collect all the pieces
  - organize the file contents
- This increases the amount of sequential code in your application for no good reason
- MPI has a feature that lets all ranks open the same file, and write their data in different parts of it
  - Probably still serializes things on your laptop, unless you've customized the OS installation
  - Actually works simultaneously when writing is done on a parallel-friendly file system

Miscellaneous debris

- MPI *can* do some limited one-sided communication
  - Provided that the receiver has registered a buffer it's ok for others to write in
- It's possible to receive from "anywhere"
  - For searching problems, and other first-past-the-post methods that should report their result as soon as it has been found by anyone
- There are non-blocking collective operations
  - We won't need them for anything, though
- There is a performance profiling interface
  - You can inject code that is run before/after every kind of MPI operation, so that you can instrument it without changing the source

That's a lot of stuff

- Absolutely, I expect it to keep us entertained for a few weeks
- We won't cover everything, though
- It is also not necessary to digest the whole thing before you can do useful things

  ...or indeed, ever – very few programmers have used every corner of MPI
- We'll take it from the beginning
  - A selection of 6 functions are all we need in principle
  - The rest is convenience-functions and window dressing

Design philosophy

- MPI is the most extensive of the programming models we will discuss
  - Why start there then?
- MPI is made with the express intention that it should never occupy your CPU unless you have given it permission
  - By asking it to do something for you
- We have mentioned that data movement is the most expensive thing to do in modern computers
  - MPI is all about data movement
  - It makes all its data movement explicit, so that you can find the bottlenecks directly in the source code
- It's easier to reason about 'invisible' data movement when you've handled it manually first