NTNU – Trondheim
Norwegian University of
Science and Technology

Six-function MPI

Jan.Christian.Meyer@ntnu.no

Today’s topic

- We have taken a broad look at the major concepts in MPI, without any details
    - Just so you’ll recognize them
- I have claimed that it’s possible to start using MPI with only a very few of them
    - Six, to be exact
- Next, we will make a poorly designed MPI version of our advection example
    - Just to show that you can do it with the six basic functions

A quick recap

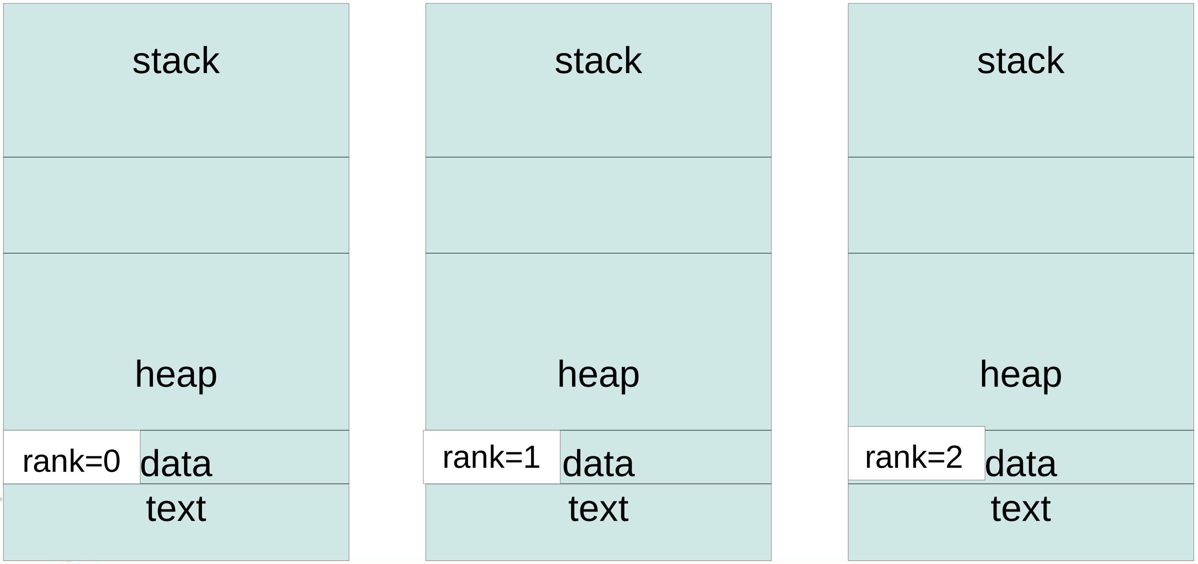
- We’ve mentioned that launching a program executable gives you a process image with the state of all its code and data in:

```
./my_program
```

MPI works with parallel processes

- It achieves this by launching your executable via an included program-launching thing
- It's usually called 'mpirun', but particular parallel systems may ask you to use another mechanism with some other name

```
mpirun -np 3 ./my_program
```



Initialization

- In order to be put in contact with its siblings, each rank must begin by initializing the internal state of the MPI library
- This can require information from the command line arguments array, so you have to pass those along

```
int main ( int argc, char **argv ) {
        MPI_Init ( &argc, &argv );
        <…rest of program comes here...>
```

Finalization

- What goes up must come down, so there’s a function that cleans up all memory that was allocated during initialization as well
- That one doesn’t need any arguments, all relevant information has been established internally

```
int main ( int argc, char **argv ) {
        MPI_Init ( &argc, &argv );
        <…rest of program goes in the middle...>
        MPI_Finalize();
}
```

We can observe a few things already

- Every MPI function is called something like

    MPI_Abcd_efg_h

    - "MPI_" to begin with
    - First letter in the function name is capitalized
    - The rest of the name is all in lowercase, with underscore separation
- MPI uses arguments to pass variables in and out of functions
    - For the vast, vast majority of functions, the return value is an error code that indicates whether the function completed in style or not
    - In order to obtain the answer from a function, you pass it a pointer to an area you have sized up to contain it, and let the function write it there

Why use pointer-arguments instead of C’s own return values?

- There is actually a reasonable rationale behind this, you will find that system libraries and many other libraries do it as well
- The purpose is to give the programmer complete control over allocation
- If you’re coming from an OO language, it’s tempting to build ‘constructors’ for your structs like this:

  ```
  my_thing * create_thing( int a, int b, int c) { /* malloc in here */ }
  void destroy_thing ( my_thing *dead ) { free ( dead ); }
  ```

  and use them like this

  ```
  my_thing *newThing = create_thing (1,2,3);
  destroy_thing ( newThing );
  ```

- This will force all my_things into the heap

Allocation on the user side

- If create_thing (…) only writes at pointers you pass it, you can make things in both of these ways:

```
// On heap
my_thing *heapThing = malloc ( sizeof(my_thing) );
create_thing ( heapThing, 1,2,3 );

// On stack
my_thing stackThing;
create_thing ( &stackThing, 4,5,6 );
```

- You don’t have to like this style or use it yourself, but MPI does, and this is the reason
  - I also tend to use it, but again, you don’t have to, it’s just a common practice

Back to MPI

- Now that we can start some processes, we'll need their ranks and total number
- As we know, the rank of a process is always connected with the communicator it is acting as a member of
- Two functions tell us what we need for now:

```
int rank, size;
MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
MPI_Comm_size ( MPI_COMM_WORLD, &size );
```

- The first returns different numbers (0 through p-1) for each process, the second returns the same number everywhere

That was 4 functions

- Only two more to go
- We already have enough to write an MPI-enabled hello program, though

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main ( int argc, char **argv ) {
    int size, rank;
    MPI_Init ( &argc, &argv );
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    MPI_Comm_size ( MPI_COMM_WORLD, &size );
    printf ( “Hello world, I am rank %d out of %d\n”, rank, size );
    MPI_Finalize ();
    exit ( EXIT_SUCCESS );
}
```

It can already be (slightly) useful

- Suppose you have a problem where every piece is independent from all the others

  (running the same program on 256 files, for instance)

  you could
  - Start some processes and get their ranks
  - Deduce a separate set of file names for each rank
  - Handle all the files in exactly the same way
- There are easier ways to do just this, though
- The literature calls this type of task

  “embarrassingly parallel”

NTNU – Trondheim
Norwegian University of
Science and Technology

Sending and receiving

- The function signature for sending looks like this:

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

- The return value is usually the constant MPI_SUCCESS, its other possible values are in the documentation

Sending: what

These arguments are straightforward:

```
int MPI_Send (
    const void *buf,            // Pointer to the data to send
    int count,                  // Number of elements to send from it
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm
);
```

Sending: where

The destination process will be the one that has rank *dest* in the communicator given as final argument

```
int MPI_Send (
    const void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,                  // Rank of the recipient
    int tag,
    MPI_Comm comm              // Communicator to send in
);
```

Sending: how much

Message length (in bytes) is the count multiplied by a size that comes from the 3rd argument

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

- There’s a list of primitive data types to choose from, like MPI_INT, MPI_DOUBLE, MPI_BYTE, *etc.*

NTNU – Trondheim
Norwegian University of
Science and Technology

Why these MPI_*** data types?

- First and foremost, because a type isn’t a value that you can pass as an argument in C or Fortran
- Because MPI needs to pass types around, it has lists of constant values that mirror basic types instead
  - Slightly impractical
- There’s a lot more to say about MPI_Datatype, though, but we will save it for another day

Receiving

The argument list is almost the same

```
int MPI_Recv (
    const void *buf,   // Where to put the result
    int count,            // Number of elements
    MPI_Datatype datatype,   // Type of elements
    int src,   // Rank of sender
    int tag,
    MPI_Comm comm,          // Communicator to send in
    MPI_Status *status
);
```

- The pointer to a status object allows you to get information about how the message was sent after you have received it
- When we don't need it for anything, it can have the value MPI_STATUS_IGNORE instead

Sending and receiving: tags

- Both MPI_Send and MPI_Recv have an ‘int tag’ argument we haven’t mentioned
  - Ordinarily, MPI pairs the correct Send with the right Recv by checking size, type, source and destination

  BUT
  - It is also possible to have multiple messages on the way at the same time
  - They might have the same sizes, types, sources and destinations
- The ‘tag’ is used to distinguish between messages in such situations
- You can just choose any number for a tag, but it has to be the same number in an MPI_Send call as in the MPI_Recv call that is intended to get the message

That was all six

- It is possible to implement all the rest of MPI’s facilities using these six functions

    MPI_Init
    MPI_Finalize
    MPI_Comm_rank
    MPI_Comm_size
    MPI_Send
    MPI_Recv

- In other words, all communication patterns can be reduced to some sequence of point-to-point messages
- We have some reasons not to do that anyway
    - It’s extra work
    - It can be quite complicated for some of the patterns
    - There may be machine-specific tricks for certain patterns that make their implementations faster than what you can portably do with Send+Recv

NTNU – Trondheim
Norwegian University of
Science and Technology

Example time

- Now that we have a working set of operations, I’ll discuss how we can use them to parallelize the advection example from before
- It’s not going to be super smooth, because we’ll be doing it only with the functions we have discussed
  - Just to prove that we can
- Hopefully, doing everything manually first will demonstrate what happens when it’s done semi-automatically later