Message-Passing and MPI Programming

Datatypes and Collectives

N.M. Maclaren

nmm1@cam.ac.uk

July 2010

3.1 Transfer Procedures

These are the procedures that actually transfer data; the point-to-point ones will be described later. All of these need to specify one or more transfer buffers, which are used to send or receive data, or both. Transfer buffer are specified using three arguments:

- The address of the buffer
- The size of the buffer
- The base datatype of the buffer

They also need to specify some control information:

- The root process for one-to-all or all-to-one transfers
- The communicator to be used for the collective

All MPI transfer buffers are *vectors* (i.e. 1-D arrays), and their base element datatypes are always scalars. Their size is an element count argument (i.e. the length of the vector in elements). The transfer buffer arguments are type-generic ("choice arguments" in MPI's terminology). These are declared as "`void *`" in C; In Fortran, MPI relies on no checking, which is technically a breach of standard, and is described later. The datatype of the base elements is passed as a separate argument.

The vectors are always *contiguous* arrays (i.e. each element immediately follows its predecessor). This is like Fortran 77 arrays (i.e. explicit shape or assumed size) or C arrays, and not like Fortran 90 assumed-shape ones; again, we will come back to Fortran 90 issues later. For example, consider transferring 100 integers. The element count is 100 and the datatype is `MPI_INTEGER` (in Fortran) or `MPI_INT` (in C).

Fortran: `INTEGER BUFFER ( 100 )`
C: `int buffer [ 100 ] ;`

Most C++ programs use containers from the C++ library rather than the built-in arrays but, in general, the data in a container is not contiguous. However, the data of `<vector>` (though not `<vector<bool> >`), `<array>` and `<string>` are required to be contiguous. The `front()` method returns a reference to the first element, so its address is the address of the data and C++ automatically converts the pointer to `void *` in the MPI calls. Many people use `&object[0]` instead of `&object.front()`.

This applies far more generally than just to MPI (e.g. it applies for uses of LAPACK or POSIX) and similar remarks apply to libraries like Boost, though be careful to check their documentation about whether they promise contiguity.

3.2 Datatypes

Because neither Fortran 77 nor C supports polymorphic programming, MPI assumes that arrays are passed as blocks of typeless data and requires the caller to specify the base type of the array as a separate argument. Because you cannot pass a type as a function argument in those languages, MPI defines some constants to indicate the type, and provides mechanisms for programmers to create new type constants for derived types and classes. Note that these datatypes are MPI constants, not necessarily language constants, and so cannot be used in initialisers.

Each datatype has an associated size, and all counts and offsets are in units of that. That is exactly the same rule as used for Fortran, C or C++ arrays. The following is how to pass an array of double of length 100 to MPI in C (Fortran is very similar):

```
double buffer [ 100 ] ;
MPI_Bcast ( buffer , 100 , MPI_DOUBLE ,
    root , MPI_COMM_WORLD , error )
```

In general, the MPI and language datatypes must match – there are some exceptions, but they are best avoided. You will **not** get warned if you make an error. That is exactly the same as in K&R C, C casts and Fortran 77; there is no equivalent of C++ class- or Fortran 90 type-checking. In theory, a compiler could detect a mismatch (as it could for those other language errors), but it would have to be 'MPI aware' and few (if any) are.

Here is a sample of recommended datatypes that are enough for the first examples. We will come back to datatypes in more detail later:

Fortran:	**C**:
MPI_INTEGER	MPI_INT
MPI_DOUBLE_PRECISION	MPI_DOUBLE

3.3 Collectives

We have already used the simplest collective, MPI_Barrier; all of the others involve some data transfer. The rules for their use are the same:

- All processes in a communicator are involved, and all must make the same call at the same time. For use on a subset, you need to create another communicator, which we shall come back to later.
- All datatypes and counts must be the same in all of the calls that match (i.e. on all processes). There are a a few, obscure exceptions, and using them is not recommended. Obviously the communicator must be the same!
- All of the buffer addresses may be different, because MPI processes do not share any addressing. There is no need for each matching call to use the same array. This generalises in more advanced use, where even the data layout may be different, and that is covered later.

The easiest way of ensuring the communicator, datatypes and counts match, and all of the collectives are called 'at the same time' is to the SPMD programming model, because you can code just one collective call.

Some collectives are asymmetric; i.e. they transfer from one process to all processes or vice versa. An example is broadcasting from one process to the whole of the communicator – and that means **all** processes, including itself. Those all have a root process argument (i.e. the 'one' process), which also must be the same on all processes ; any process can be specified – not just zero. Symmetric ones do not have that argument; for example, `MPI_Barrier` does not.

Most collectives use separate send and receive buffers, both for flexibility and for standards conformance. They usually (but not always) have datatype and count arguments for each buffer; this is needed for advanced features not covered in this course.

In some cases, especially for the asymmetric collectives, not all arguments are needed on all processes, and MPI uses only the arguments it needs; unused ones are completely ignored. However, you are advised to set them all compatibly, because it is much safer! In particular, keep all datatypes and counts the same, even if they are not used.

3.4 Broadcast

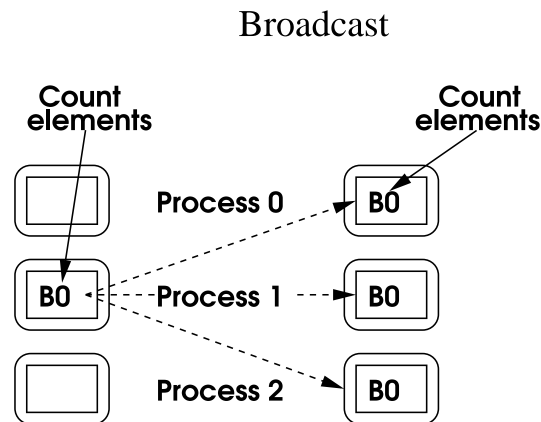

Figure 3.1

Broadcast copies the same data from the root process to all processes in the communicator. It is the second simplest collective.

Fortran:

```
REAL(KIND=KIND(0.0D0)) :: buffer ( 100 )
INTEGER , PARAMETER :: root = 3
INTEGER :: error
CALL MPI_Bcast ( buffer , 100 , MPI_DOUBLE_PRECISION ,    &
    root , MPI_COMM_WORLD , error )
```

C:

```
double buffer [ 100 ] ;
int root = 3 , error ;
error = MPI_Bcast ( buffer , 100 , MPI_DOUBLE ,
    root , MPI_COMM_WORLD ) ;
```

C++ using the **C** interface:

```
vector<double> buffer ( 100 ) ;
int root = 3 ;
error = MPI_Bcast ( & buffer . front () , 100 , MPI_DOUBLE ,
    root , MPI_COMM_WORLD ) ;
```

3.5 Multiple Transfer Buffers

Many collectives need one buffer per process. That is, process X needs one buffer for each of the processes in the collective. For example, take a 1⇒N scatter operation; the root process sends different data to each process. MPI handles these by treating each buffer as the concatenation of one pairwise transfer buffer for each process, in the order of process numbers (i.e. 0...N-1), so: <size of source> = × <size of each result>.

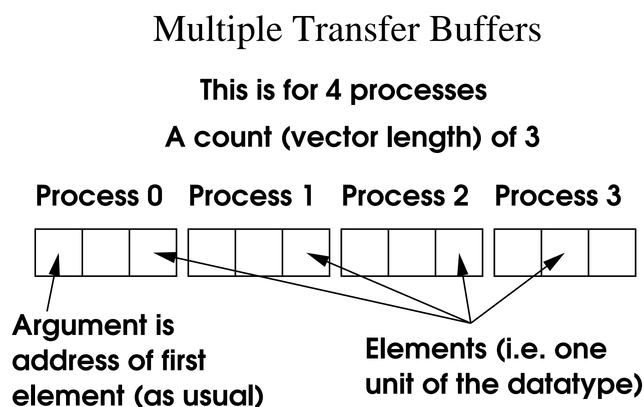

Figure 3.2

The size specifications are slightly counter-intuitive, though it is done for consistency and simplicity; if you think the design through, you will see why. You specify the size of each pairwise transfer, and MPI will deduce the total size of the buffers – i.e. it will multiply by process count, if needed. The process count is implicit, and is taken from the communicator – i.e. the result from `MPI_Comm_size`.

‘`void *`’ defines no length in C , nor does ‘`<type> :: buffer(*)`’ in Fortran. It is up to you to get it right; no compiler can trap an error with such an interface.

We shall use scatter as our first example; this is one process sending different data to every process in the communicator.

3.6 Scatter

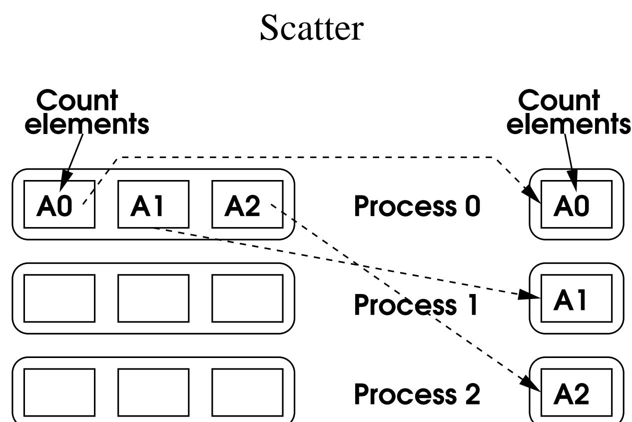

Figure 3.3

Scatter copies data from the root to all processes in the communicator – unlike broadcast, different data is sent to each process. The send buffer is used only on the root, but the receive buffer is used on all processes (including the root).

The Following examples assume $\leq$30 processes; this is specified **only** in the send buffer size and not in the call. Note the differences in the buffer declarations, and consider what you would have to do to handle 50 processes.

Fortran:

```
REAL(KIND=KIND(0.0D0)) :: sendbuf ( 100 , 30 ) , recvbuf ( 100 )
INTEGER , PARAMETER :: root = 3
INTEGER :: error
CALL MPI_Scatter (    &
    sendbuf , 100 , MPI_DOUBLE_PRECISION ,    &
    recvbuf , 100 , MPI_DOUBLE_PRECISION ,    &
    root , MPI_COMM_WORLD , error )
```

C:

```
double sendbuf [ 30 ] [ 100 ] , recvbuf [ 100 ] ;
int root = 3 , error ;
error = MPI_Scatter ( sendbuf , 100 , MPI_DOUBLE ,
    recvbuf , 100 , MPI_DOUBLE , root , MPI_COMM_WORLD )
```

C++ using the **C** interface:

```
vector < double > sendbuf ( 30 * 100 ) , recvbuf ( 100 ) ;
int root = 3 , error ;
error = MPI_Scatter (
        & sendbuf . front () , 100 , MPI_DOUBLE ,
        & recvbuf . front () , 100 , MPI_DOUBLE ,
        root , MPI_COMM_WORLD )
```

Remember that only the contents are contiguous, so do **not** create multiple buffers like this:

```
array< array< double , 100 > , 30 > sendbuf ;
```

3.7 Hiatus

Those are the basic principles of collectives. Now might be a good time to do some examples, and the first few questions cover the material so far. Once you are happy with the basic principles, then you should read on.

3.8 Fortran Datatypes

The following datatypes are recommended – all match the obvious Fortran types except where explained:

```
MPI_CHARACTER
MPI_LOGICAL
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_COMPLEX
MPI_DOUBLE_COMPLEX
```

`MPI_CHARACTER` matches `CHARACTER(LEN=1)`, which can be used for other lengths of character variables using Fortran's sequence association rules. This is covered briefly in a much later lecture, and in more detail in the course *An Introduction to Modern Fortran.*

`MPI_DOUBLE_COMPLEX` matches `COMPLEX(KIND=KIND(0.0D0))`, i.e. double precision complex, which is not a standard type in Fortran 77. However, almost all compilers have supported it (under various names) since Fortran 66.

MPI also supports Fortran 90 parameterized types (i.e. with `KIND` value selected by precision); we shall return to these later.

For use from Fortran, that is all I recommend. There are two more built-in datatypes, that you may want to use for advanced work, but which are not covered in this course. What you can do with them, especially the latter, is a bit restricted.

MPI_PACKED for MPI derived datatypes
MPI_BYTE (uninterpreted 8-bit bytes)

You should definitely avoid MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, MPI_REAL2, MPI_REAL4 and MPI_REAL8. MPI_<type>N translates to <type>*N. While that notation is common, it is non-standard and outmoded. Many people believe that the N is the size in bytes, but that is **not** true! It may be so, on some compilers, but it may mean several other things. E.g. REAL*2 works only on *Cray* vector systems.

3.9 C Datatypes

The following integer datatypes are recommended – all match the obvious C types except where explained:

MPI_CHAR
MPI_UNSIGNED_CHAR
MPI_SIGNED_CHAR
MPI_SHORT
MPI_UNSIGNED_SHORT
MPI_INT
MPI_UNSIGNED
MPI_LONG
MPI_UNSIGNED_LONG

Note that MPI_CHAR is for char, meaning characters. Do not use use it for small integers and arithmetic, as MPI does not support it for that purpose – use the appropriate one of MPI_UNSIGNED_CHAR or MPI_SIGNED_CHAR instead.

Also note that it is MPI_UNSIGNED and not MPI_UNSIGNED_INT. I do not know why, nor why MPI did not define the latter as an alias.

The following floating-point datatypes are recommended, and match the obvious C types (but remember that long double is deceptive):

MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE

There is one more recommended datatype for C, though what you can do with it is a bit restricted. It is useful primarily for C++ PODs and C structures that would be PODs in C++.

MPI_BYTE (uninterpreted 8-bit bytes)

MPI 3.0 has added several C++ datatypes to the C interface, so that programs that use the above C++ datatypes can be converted. Most implementations will not have them yet, but should have them shortly; if you are using the C interface from C++, it is worth checking if they exist.

MPI_CXX_BOOL
MPI_CXX_FLOAT_COMPLEX
MPI_CXX_DOUBLE_COMPLEX
MPI_CXX_LONG_DOUBLE_COMPLEX

There are types for C99 `_Complex`, if you use it, but I do not advise using that (or most of C99, for that matter). C99 `_Complex` may not be compatible with C++ `complex`, and WG14 (the ISO C standard committee) have now made `_Complex` optional, at the choice of the implementor. There are other problems with it, too.

MPI_C_BOOL
MPI_C_FLOAT_COMPLEX
MPI_C_DOUBLE_COMPLEX
MPI_C_LONG_DOUBLE_COMPLEX

`MPI_PACKED` can be used for MPI derived datatypes in C, just as in Fortran, but this course does not cover them.

You should avoid `MPI_LONG_LONG_INT`, `MPI_UNSIGNED_LONG_LONG` and `MPI_WCHAR`, as they are not reliably portable; it is a good idea to avoid using `long long` and `wchar_t`, anyway. There is no support for C99's new types (the extended integer types), which is good, because they are an obstacle to long-term portability.

3.10 Gather

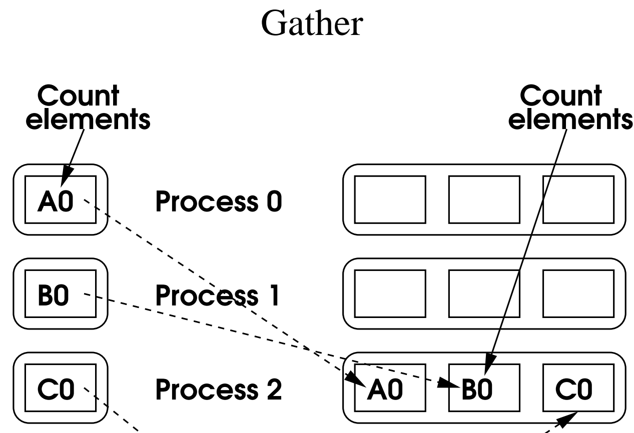

Figure 3.4

Gather is precisely the converse of scatter; it collects data rather than distributing it. All you have to do is to change the `Scatter` to `Gather` in the function names. Of course, the array sizes need changing, because it is now the receive buffer that needs to be bigger. Similarly, the send buffer is used on all processes, and the receive buffer is used only on the root.

3.11 Allgather

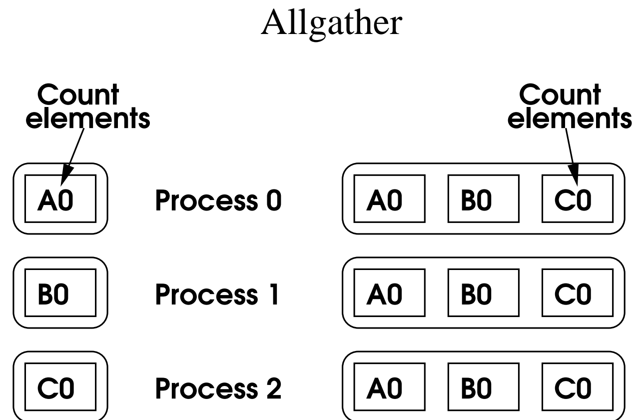

Figure 3.5

You can gather data and then broadcast it to all processes; the interface is very similar, with one difference. This is now a symmetric operation, so has no argument specifying the root process. You just Change `Gather` to `Allgather` in the calls and remove the root process argument. The receive buffer is now used on all processes, so you need to make sure that it is large enough.

Fortran:

```
REAL(KIND=KIND(0.0D0)) :: sendbuf ( 100 ) , recvbuf ( 100 , 30 )
INTEGER :: error
CALL MPI_Allgather (    &
    sendbuf , 100 , MPI_DOUBLE_PRECISION ,    &
    recvbuf , 100 , MPI_DOUBLE_PRECISION ,    &
    MPI_COMM_WORLD , error )
```

C:

```
double sendbuf [ 100 ] , recvbuf [ 30 ] [ 100 ] ;
int error ;
error = MPI_Allgather (
    sendbuf , 100 , MPI_DOUBLE ,
    recvbuf , 100 , MPI_DOUBLE ,
    MPI_COMM_WORLD )
```

3.12 Alltoall

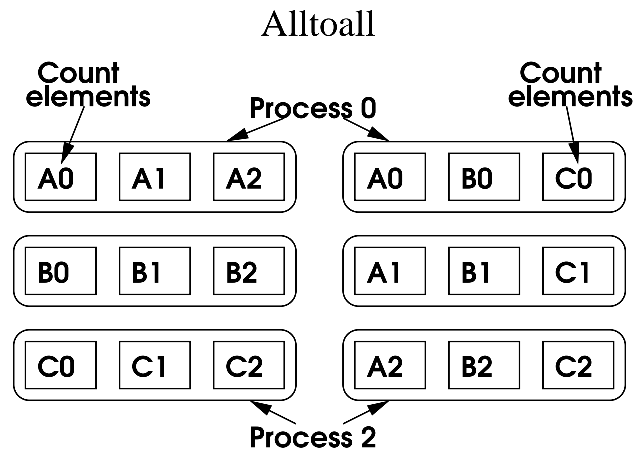

Figure 3.6

You can do a composite gather/scatter operation, which uses essentially the same interface as `MPI_Allgather`. You just change `Allgather` to `Alltoall` in the calls but, now, **both** buffers need to be bigger.

The easiest way to think of this is as a sort of parallel transpose, changing from blocks of rows being distributed across processes to blocks of columns being distributed; in fact, it is used when implementing matrix transpose. It is very powerful and is a key for performance in many programs. You are advised to learn how to use it (even if you put it on one side while you get more experienced with MPI).

3.13 Global Reductions

Global reductions are one of the basic parallelisation primitives. Logically, these start with a normal gather operation and then sum the values across all processes. By implementing that as a single call, it can often can be implemented much more efficiently. Note that summation is not the only reduction – anything that makes mathematical sense can be used, all of the standard ones are provided, and you can define your own (though that is advanced use).

Aside: mathematically, any operation that is associative can be used in a MPI style reduction; the values do not have to be numbers, and can be composite values (classes, structures or derived makes that optionaltypes). I prefer a stricter form that requires commutativity as well, but MPI makes that optional.

Reduce

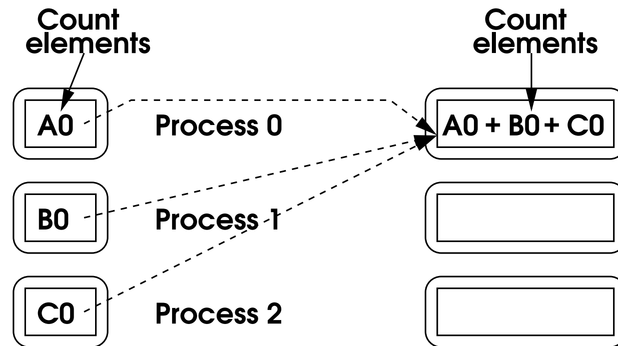

Figure 3.7

Note that `MPI_Reduce` specifies the datatype and count once, and not separately for the source and result, unlike for the copying operations. It makes no mathematical sense to do have different source and result types, so MPI does not support that.

Also, note that it does **not** reduce over the vector; the count is the size of the result, too. It sums the values for each element separately (i.e. it does N simultaneous reductions, where N is the length of the vector). If you want to, you have to reduce over the vector yourself; doing it beforehand is more efficient, because you transfer less data.

Fortran:

```
REAL(KIND=KIND(0.0D0)) :: sendbuf ( 100 ) , recvbuf ( 100 )
INTEGER , PARAMETER :: root = 3
INTEGER :: error
CALL MPI_Reduce (
    sendbuf , recvbuf ,  100 , MPI_DOUBLE_PRECISION ,    &
    MPI_SUM , root , MPI_COMM_WORLD , error )
```

C:

```
double sendbuf [ 100 ] , recvbuf [ 100 ] ;
int root = 3 , error ;
error = MPI_Reduce (
    sendbuf , recvbuf , 100 , MPI_DOUBLE ,
    MPI_SUM , root , MPI_COMM_WORLD )
```

3.14 Allreduce

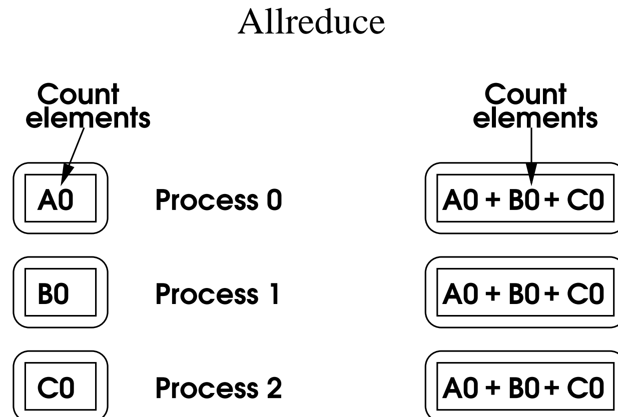

Figure 3.8

You can reduce data and then broadcast it. As with MPI_Allgather and MPI_Gather, the interface is essentially identical to MPI_Reduce, but this is now a symmetric operation and so has no argument specifying the root process. All you do is change Reduce to Allreduce and remove the root process argument. Again, the receive buffer is now used on all processes.

3.15 Reduction Operations

There are operations for the sum and product, which can be used for all precisions and all classes of number: integer, real and complex.

Fortran:	**C**:	
MPI_SUM	MPI_SUM	Summation
MPI_PROD	MPI_PROD	Product

The minimum and maximum are similar, but are not applicable to complex numbers.

Fortran:	**C**:	
MPI_MIN	MPI_MIN	Minimum
MPI_MAX	MPI_MAX	Maximum

There are no reductions on character data (i.e. MPI_CHARACTER and MPI_CHAR).

Remember that the Boolean type is LOGICAL in Fortran and int in C, and the supported values are **only** True and False (i.e. 1 and 0). It has the following operations.

Fortran:	**C**:	
MPI_LAND	MPI_LAND	Boolean *AND*
MPI_LOR	MPI_LOR	Boolean *OR*
MPI_LXOR	MPI_LXOR	Boolean *Exclusive OR*

You can also perform bitwise operations on integers (i.e. using them as bit masks), which has the same operations as for Boolean values, but with different names.

Fortran:	**C**:	
MPI_BAND	MPI_BAND	Integer bitwise *AND*
MPI_BOR	MPI_BOR	Integer bitwise *OR*
MPI_BXOR	MPI_BXOR	Integer bitwise *Exclusive OR*

3.16 Conclusion

There is a little more to say on collectives, but that is quite enough for now. The above has covered all of the essentials, and the remaining aspects to cover are only Fortran parameterised types, searching as a reduction, more flexible buffer layout, and using collectives efficiently. In fact, you have now covered enough to start writing real scientific applications, though there is more that you need to know in order to write efficient and reliable ones.

There are a lot of exercises on the above, which will take you through almost all aspects that have been shown, so that you get a little practice with every important feature. Each one should need very little editing or typing, and you can usually start from a previous one as a basis. But **please** check that you understand the point, that you get the same answers as are provided, and that you understand what it is doing and why. The exercises are pointless if you do them mechanically, because coding MPI is not where the problems arise; understanding how it works and how it should be used is.

You are recommended to make sure that you understand what has been covered so far before proceeding to new topics. This lecture covers topics that are critical to understanding MPI.