Programming with MPI

Composite Types and Language Standards

Nick Maclaren

nmm1@cam.ac.uk

March 2008

Composite Types (1)

So far, mainly contiguous arrays of basic types
n–D arrays handled in array element order
Fortran 77 and C are similar

Advanced collectives allow one level of separation

- Fortran 90 arrays not always contiguous
  n–D assumed shape array may have n 'strides'
- C and C++ have structures and pointers
  And "objects" are often built using them
- Fortran 90 and C++ have "classes"

Composite Types (2)

- MPI defines language–independent support

It isn't a great success, and is rarely used

Will explain how to do the job properly

Will give simpler, useful, partial solutions

Will explain why general solution is intractable

Proper Derived Types

- A case where the right solution is easiest

It is a language issue, but few get it right

All derived types (classes) need these methods:

- A validator to check correctness
- A displayer to use for diagnostics
- A packer to convert to exportable form
- An unpacker to convert back again

It is the last two that we are interested in here
Python gets this largely right (see “pickle”)

Complicated Data Structures

- Write proper pack and unpack functions

You don't have to make them into methods

- You can transfer pointers as hash codes

Also need the hash codes of the targets
Can then match up on receipt, and fix up properly

- Put proper checking and diagnostics into them

You will then get your program working a lot faster

See "Building Applications out of Multiple Programs"
And/or ask me for help with your problem

Shortcuts (Hacks)

In a simple case, you can put the code inline
Or pack multiple transfers into one function

- Do whichever is simplest and cleanest

1: Pack up your data for export
2: Do the actual data transfer
3: Unpack the data you have imported

OR

1: Transfer the first simple array
2: Transfer the second simple array

. . .

n: Rebuild them into a consistent structure

Simple Packing

- Can simply convert default integers to doubles
  In practice, that is almost always safe
  Adequate for 99% of MPI data

May not be adequate for long, MPI_Aint or
INTEGER(KIND=MPI_ADDRESS_KIND)

- Or you may prefer to use ''byte streams''

You use MPI_BYTE to transfer byte streams
The count is the number of bytes

C++ PODs and C structs

C++ PODs and similar C structs are easy
Use as array of sizeof bytes (type MPI_BYTE)

But you must follow these rules
Or, occasionally, everything will fall apart

- Do it only when using the same executable
- Do it only between identical types
- Don't do it if they contain pointers
- Don't do it if have any environment data

And watch out for variable sized structs

C, C++ and POSIX

Some C, C++ and POSIX features are toxic
Often cause chaos to almost all other interfaces
Can be used safely, but only by real experts

<signal.h>, <setjmp.h>, <fenv.h>
POSIX threading, signal handling, scheduling
    timer control, alarm, sleep, ...

It's easy to break MPI's rules using C++ exceptions
E.g. releasing an in–use non–blocking buffer

Fortran Type Checking

A routine must use compatible arguments everywhere
MPI buffers can be of any supported type
So the compiler may object to your use of them

- This is also fixed in MPI 3

If compiler objects to buffer argument type:

- Keep all calls in one module the same

Fortran compilers rarely check over all program

- Or write trivial wrappers in external procedures

E.g. My_Send_Integer and My_Send_Double

Fortran Derived Types

Fortran 2003 supports BIND(C) for interoperability
BIND(C) derived types are like C++ PODs

In general, don't treat them like PODs
And never do if they contain allocatable arrays

- No option but to transfer them as components
Tedious, messy, but not difficult

- Don't assume SEQUENCE means C−compatible
Has its uses for MPI, but too complicated to describe

Fortran Assumed Shape Arrays (1)

Good Fortran 90 uses assumed shape arrays
MPI 3 supports them properly, but not covered here

- MPI 2 uses assumed size arrays (i.e. Fortran 77)

Generally requires a copy, on call and return
Ignore this if not a performance problem
See Fortran course for some more details

- If you need to convert to/from contiguous arrays

Can simply write your own DO–loops
But Fortran 90 has several useful procedures

Fortran Assumed Shape Arrays (2)

To transfer a general array:

- You can extract the bounds or shape
  Using LBOUND/UBOUND or SHAPE
- You can flatten array elements
  Using PACK or RESHAPE
- You can build arrays for receiving those
  Using ALLOCATE
- You can unpack a flattened array
  Using UNPACK or RESHAPE

Fortran Precisions (1)

Fortran 90 allows selectable precisions
KIND=SELECTED_INTEGER_KIND(precision)
KIND=SELECTED_REAL_KIND(precision[,range])

Can create a MPI derived datatype to match these
Then can use it just like a built–in datatype

- Call the datatype constructor

Surprisingly, it is a predefined datatype
Do NOT commit or free it
[Don't worry if that makes no sense to you]

Fortran Precisions (2)

```
INTEGER ( KIND =          &
        SELECTED_INTEGER_KIND(15) ) ,      &
    DIMENSION(100) :: array
INTEGER :: root , integertype , error

CALL MPI_Type_create_f90_integer (       &
    15 , integertype , error )
CALL MPI_Bcast ( array , 100 ,       &
    integertype , root ,       &
    MPI_COMM_WORLD , error )
```

Fortran Precisions (3)

REAL and COMPLEX are very similar

```
REAL ( KIND =          &
        SELECTED_REAL_KIND(15,300) ) ,        &
    DIMENSION(100) :: array
CALL MPI_Type_create_f90_real (        &
    15 , 300 , realtype , error )

COMPLEX ( KIND =          &
        SELECTED_REAL_KIND(15,300) ) ,        &
    DIMENSION(100) :: array
CALL MPI_Type_create_f90_complex (        &
    15 , 300 , complextype , error )
```

Temporary Problem

MPI_Type_free is broken in OpenMPI
So comment out the call when writing examples

The memory leakage isn't important in most programs

It could be if you create new types, repeatedly

That's All For Now

There are only two simple practicals on the above

- For C, transferring structures
- For Fortran, using the precision control

Rest of this lecture is about what not to do

Explain why have omitted MPI derived datatypes
Then describe some language standard issues

MPI Derived Types

MPI supports the following composition operations:

Contiguous replication
Constant stride (offset) vectors
Indexed vectors
Structures of different types

- Might help with Fortran arrays and CHARACTER

But sequence association means it isn't needed

- Doesn't help much with the other problems

In all cases, you need to know the exact layout

Advanced Structure Use

Fortran 90 derived type layout is easy
Implementation dependent and unspecified

C structure/union layout is a nightmare
Anything that explains it simply is just plain wrong
Even SC22/WG14 doesn't agree on the rules
Behaviour often changes with compiler option

C++ is a little better, but not much

- You don't want to open that can of worms

Please ask me offline if you want to know more

Pointers etc.

Many advanced composite types include pointers

- Should you copy the object pointed to?
- If not, what happens to the pointer value?

The same problem as copying directory trees
What do you do with hard and soft links?
All Unix utilities (and most versions) are different

MPI, sensibly, has no support for such types
Write proper pack/unpack functions – it's easiest

More on Language Interfaces

You need to know about these, to avoid problems
And if you use the advanced features in future

- This is mainly about what not to do

Especially ‘‘reliable’’ interfaces that aren’t

> It ain’t what we don’t know that causes trouble,
> it’s what we know for sure that ain’t so.

Probably Mark Twain or Josh Billings

Callbacks

Some MPI features have callback procedures
I.e. ones that you write and which MPI calls

- Avoid updating global data in callbacks

All languages have some nasty "gotchas"
Please ask if you want to know why and how

- Don't use longjmp out of MPI procedures
- Or jump out using C++ exceptions

Either will probably work, but ...

C/C++/POSIX Issues

- Here, C and C++ are similar; Fortran differs

Don't assume that MPI constants can be used in #if

- Don't use POSIX signal handling (e.g. masking)
- Don't call MPI in signal handlers, atexit or C++ destructors

MPI requests implementations to support that, but it is undefined behaviour in C and C++

OpenMP, SMP Libraries etc.

SMP libraries usually implemented using OpenMP
OpenMP usually coded using POSIX threads

One easy, fairly safe, path allowing you to use both:

- Use one MPI process per system
- Call MPI only from the master/initial thread
- Leave the other CPUs to the SMP library etc.

Alternate Approach to SMP

- Compile and link using only the serial libraries

- Run several MPI processes on a SMP system

However many is best – NOT more than CPUs

You can (with difficulty) run more for testing
But some very nasty "gotchas" lurk there

- Never mix this approach and SMP libraries

Or any other form of threading ...

Actually Using Threading

If you really must use threading directly:

- Call MPI only from the initial thread
- Never put that into a thread wait
- Watch out for evil race conditions

- POSIX signal handling is pure poison

That is a minefield, even without threading
MPI+threading+signals $\equiv$ CHAOS

C/C++ Standard Conformance

MPI is unavoidably incompatible with C/C++
No more than POSIX, TCP/IP or .NET are! †

Similar ones to the incompatibilities with Fortran
E.g. non–blocking calls do transfers in parallel

- That is undefined behaviour in the C standard

- As always with C/C++, program defensively

Occasionally need to reduce optimisation level
Please ask if you want to know more about this

† One cause of C/C++/POSIX/.NET unreliability

Fortran Standard Conformance

Some unavoidable breaches of Fortran standard

- Type–generic (“choice”) args mentioned above
- Assumes call–by–reference for all arguments
- Non–blocking calls do transfers in parallel

Can cause trouble with extremely stringent checking

- More often, with high levels of optimisation

Problems are rare – most people never see any

You may need to use special compiler options

- Ask for help if you have trouble here

Fortran and Type-Generics

Fortran 77 and Fortran 95 don't support them
Procedures must have same arg. types everywhere
⇒ So we have to try to fool the compiler

- Keep all calls in one module the same

Fortran compilers rarely check over all program

- Or write trivial wrappers in external procedures

E.g. My_Send_Integer and My_Send_Double

Fortran 2003 does support such things
But not in quite the same way that MPI does

Fortran and Non-Blocking (1)

Fortran 95 does not allow asynchronous actions
MPI non–blocking transfers are asynchronous
The only difficulty in specifying the transfer buffers

- MUST avoid them being copied on the call

That matters only for non–blocking transfers

There is one simple rule that usually works:

- Make the transfer buffer a Fortran 77 array
  either explicit size or assumed size

In a common parent of both send/receive and wait

Fortran and Non-Blocking (2)

- If that doesn't work, ask me for help (it's tricky)

However, you will be very unlucky for it not to

Despite common belief, it is NOT required to
Fortran does not require call–by–reference

Fortran 2003 does support asynchronous actions

MPI and Fortran 2003

Currently, it is mostly a Fortran 77 interface
With some features taken from Fortran 90

- Works with all current Fortran compilers

A lot could be done using Fortran 2003
I.e. like the C++ improvement to the C one
It would provide a much better interface

- Like the C++ one, it would be very different

It's not going to happen