

Message Passing Programming

Designing MPI Applications

Overview

- Lecture will cover
 - MPI portability
 - maintenance of serial code
 - general design
 - debugging
 - verification

MPI Portability

- Potential deadlock
 - you may be assuming that **MPI_Send** is asynchronous
 - it often is buffered for small messages
 - but threshold can vary with implementation
 - a correct code should run if you replace all **MPI_Send** calls with **MPI_Ssend**
- Buffer space
 - cannot assume that there will be space for **MPI_Bsend**
 - default buffer space is often zero!
 - be sure to use **MPI_Buffer_Attach**
 - some advice in MPI standard regarding required size

Data Sizes

- Cannot assume data sizes or layout
 - e.g. C `float` / Fortran `REAL` were 8 bytes on Cray T3E
 - can be an issue when defining `struct` types
 - use `MPI_Type_extent` to find out the number of bytes
 - be careful of compiler-dependent padding for structures
- Changing precision
 - when changing from, say, `float` to `double`, must change all the MPI types from `MPI_FLOAT` to `MPI_DOUBLE` as well
- Easiest to achieve with an include file
 - e.g. every routine includes `precision.h`

Changing Precision: C

- Define a header file called, e.g. `precision.h`
 - `typedef float RealNumber`
 - `#define MPI_REALNUMBER MPI_FLOAT`
- Include in every function
 - `#include "precision.h"`
 - ...
 - `RealNumber x;`
 - `MPI_Routine(&x, MPI_REALNUMBER, ...);`
- Global change of precision now easy
 - edit 2 lines in one file: `float`->`double`, `MPI_FLOAT`->`MPI_DOUBLE`

Changing Precision: Fortran

- Define a module called, e.g., `precision`
 - `integer, parameter :: REALNUMBER=kind(1.0e0)`
 - `integer, parameter :: MPI_REALNUMBER = MPI_REAL`
- Use in every subroutine
 - `use precision`
 - `...`
 - `REAL(kind=REALNUMBER) :: x`
 - `call MPI_ROUTINE(x, MPI_REALNUMBER, ...)`
- Global change of precision now easy
 - change `1.0e0` -> `1.0d0`, `MPI_REAL`-> `MPI_DOUBLE_PRECISION`

Testing Portability

- Run on more than one machine
 - assuming the implementations are different
 - many parallel clusters will use the same open-source MPI
 - e.g. OpenMPI or MPICH2
 - running on two different mid-sized machines may not be a good test
- More than one implementation on same machine
 - e.g. run using both MPICH2 *and* OpenMPI on your laptop
 - very useful test, and can give interesting performance numbers
- More than one compiler
 - `user@morar$ module switch mpich2-pgi mpich2-gcc`

Serial Code

- Adding MPI can destroy a code
 - would like to maintain a serial version
 - i.e. can compile and run identical code without an MPI library
 - not simply running MPI code with $P=1$!
- Need to separate off communications routines
 - put them all in a separate file
 - provide a dummy library for the serial code
 - no explicit reference to MPI in main code

Example: Initialisation

```
! parallel routine
subroutine par_begin(size, procid)
  implicit none
  integer :: size, procid
  include "mpif.h"
  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD, size, ierr)
  call mpi_comm_rank(MPI_COMM_WORLD, procid, ierr)
  procid = procid + 1
end subroutine par_begin
```

```
! dummy routine for serial machine
subroutine par_begin(size, procid)
  implicit none
  integer :: size, procid
  size = 1
  procid = 1
end subroutine par_begin
```

Example: Global Sum

```
! parallel routine
subroutine par_dsum(dval)
  implicit none
  include "mpif.h"
  double precision :: dval, dtmp
  call mpi_allreduce(dval, dtmp, 1, MPI_DOUBLE_PRECISION, &
                    MPI_SUM, comm, ierr)

  dval = dtmp
end subroutine par_dsum

! dummy routine for serial machine
subroutine par_dsum(dval)
  implicit none
  double precision dval
end subroutine par_dsum
```

Example Makefile

```
SEQSRC= \  
    demparams.f90 demrand.f90 demcoord.f90 demhalo.f90 \  
    demforce.f90 demlink.f90 demcell.f90 dempos.f90 \  
    demons.f90
```

```
MPISRC= \  
    demparallel.f90 \  
    demcomms.f90
```

```
FAKESRC= \  
    demfakepar.f90 \  
    demfakecomms.f90
```

```
#PARSRC=$(FAKESRC)  
PARSRC=$(MPISRC)
```

Advantages of Comms Library

- Can compile serial program from same source
 - makes parallel code more readable
- Enables code to be ported to other libraries
 - more efficient but less versatile routines may exist
 - e.g. Cray-specific SHMEM library
 - can even choose to only port a subset of the routines
- Library can be optimised for different MPIs
 - e.g. choose the fastest send (**Ssend**, **Send**, **Bsend**?)

Design

- Separate the communications into a library
- Make parallel code similar as possible to serial
 - e.g. use of halos in case study
 - could use the same update routine in serial and parallel

```
serial:    update(new, old, M, N );  
parallel: update(new, old, MP, NP);
```
 - may have a large impact on the design of your serial code
- Don't try and be too clever
 - don't agonise whether one more halo swap is really necessary
 - just do it for the sake of robustness

General Considerations

- Compute everything everywhere
 - e.g. use routines such as **Allreduce**
 - perhaps the value only really needs to be know on the master
 - but using **Allreduce** makes things simpler
 - no serious performance implications
- Often easiest to make P a compile-time constant
 - may not seem elegant but can make coding much easier
 - e.g. definition of array bounds
 - put definition in an include file
 - a clever **Makefile** can reduce the need for recompilation
 - only recompile routines that define arrays rather than just use them
 - pass array bounds as arguments to all other routines

Debugging

- Parallel debugging can be hard
- Don't assume it's a parallel bug!
 - run the serial code first
 - then the parallel code with $P=1$
 - then on a small number of processes ...
- Writing output to separate files can be useful
 - e.g. log.00, log.01, log.02, for ranks 0, 1, 2, ...
 - need some way easily to switch this on and off
- Some parallel debuggers exist
 - Totalview is the leader across all largest platforms
 - Allinea DDT is becoming more common across the board

General Debugging

- People seem to write programs **DELIBERATELY** to make them impossible to debug!
 - my favourite: the silent program
 - “my program doesn’t work”
 - \$ `mprun -np 6 ./program.exe`
 - \$ `SEGV core dumped`
 - where did this crash?
 - did it run for 1 second? 1 hour? in a batch job this may not be obvious
 - did it even start at all?

Why don’t people write to the screen!!!

Program should output like this

```
$ mprun -np 6 ./program.exe
Program running on 6 processes
Reading input file input.dat ...
... done
Broadcasting data ...
... done
rank 0: x = 3
rank 1: x = 5
etc etc
Starting iterative loop
iteration 100
iteration 200
finished after 236 iterations
writing output file output.dat ...
... done
rank 0: finished
rank 1: finished
...
Program finished
```

Typical mistakes

- Don't write raw numbers to the screen!

- what does this mean?

```
$ mprun -np 6 ./program.exe
```

```
1 3 5.6
```

```
3 9 8.37
```

- programmer has written

```
$ printf("%d %d %f\n", rank, j, x);
```

```
$ write(*,*) rank, j, x
```

- Takes an extra 5 seconds to type:

```
$ printf("rank, j, x: %d %d %f\n", rank, j, x);
```

```
$ write(*,*) `rank, j, x: `, rank, j, x
```

- and will save you HOURS of debugging time

- Why oh why do people write raw numbers?!?!

Debugging walkthrough

- My case study code gives the wrong answer
- Stages:
 - read data in
 - distribute to processes
 - update many times
 - requiring halo swaps
 - collect data back
 - write data out
- Final stage shows the error
 - but where did it first go wrong?

Common mistake

- I changed something
 - and it now works (but I don't know why)
- All is OK!
- No!
 - there is a bug
 - you **MUST** find it
 - if not, it will come back later to bite you **HARD**
- Debugging is an experimental science

Where is it going wrong?

- On input?
- On distribute?
- On update?
 - on halo swaps?
 - on left/right swaps?
 - on up/down swaps?
- On collection?
- On output?

- All these can be checked with simple tests

Verification: Is My Code Working?

- Should the output be identical for any P ?
 - very hard to accomplish in practice due to rounding errors
 - may have to look hard to see differences in the last few digits
 - typically, results vary slightly with number of processes
 - need some way of quantifying the differences from serial code
 - and some definition of “acceptable”
- What about the same code for fixed P ?
 - identical output for two runs on same number of processes?
 - should be achievable with some care
 - not in specific cases like dynamic task farms
 - possible problems with global sums
 - MPI doesn't force reproducibility, but some implementations can
 - without this, debugging is almost impossible

Parallelisation

- Some parallel approaches may be simple
 - but not necessarily optimal for performance
 - case study example is very simple due to 1D decomposition
 - but not particularly efficient for large P
 - often need to consider what is the realistic range of P
- Some people write incredibly complicated code
 - step back and ask: what do I actually want to do?
 - is there an existing MPI routine or collective communication?
 - should I reconsider my approach if it prohibits me from using existing routines, even if it is not quite so efficient?

Optimisation

- Keep running your code
 - on a number of input data sets
 - with a range of MPI processes
- If scaling is poor
 - find out what parallel routines are the bottlenecks
 - again, much easier with a separate comms library
- If performance is poor
 - work on the serial code
 - return to parallel issues later on

Conclusions

- Run on a variety of machines
- Keep it simple
- Maintain a serial version
- Don't assume all bugs are parallel bugs
- Find a debugger you like (good luck to you)