Message Passing Programming

Tips and tricks

Reusing this material

Aims

- To write correct MPI programs
  - that are portable to many systems
  - that are efficient
  - that are easy to maintain

Common problems in MPI

- Assuming MPI_Send is asynchronous
- Non-portability
- Programs with specific process counts
- Not calling collectives collectively
- Incorrect use of non-blocking
- Sending lots of small messages
- Array allocation issues in C
- Array syntax issues in Fortran
- Code readability
- Debugging problems

Assuming MPI_Send is asynchronous

- Potential deadlock
  - you may be assuming that **`MPI_Send`** is asynchronous
  - it often *is* buffered for small messages
    - but threshold will vary with implementation
  - you code may run on one machine and deadlock on another
  - correct code will run with all **`MPI_Send`** calls replaced by **`MPI_Ssend`**

- Buffer space
  - cannot assume that there will be space for **`MPI_Bsend`**
  - default buffer space may be zero!
  - be sure to use **`MPI_Buffer_attach`**
    - some advice in MPI standard regarding required size
    - allow for space for message headers: **`MPI_BSEND_OVERHEAD`**

Data Sizes

- Be careful of data sizes or layout
  - use runtime enquiry functions for Fortran types
  - be careful of compiler-dependent padding for structures

- Do not use magic compiler flags to change precision!

```
cc -convert-floats-to-doubles *.c
```

- 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`**

Non-portability

- Correct C code should compile correctly with *any* C compiler
- Correct MPI code should also run correctly with *any* MPI library

- Run on more than one machine
  - assuming the MPI libraries are different
  - many parallel clusters will use the same open-source MPI
    - e.g. OpenMPI or MPICH2
    - running on two different HPC systems 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@cluster$ module switch mpich2-gcc mpich2-intel`

Code Readability

- 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)
```

Example: Initialisation

```
// Ugly code
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORDL, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// Nicer code
par_begin(&size, &rank);

// parallel function in libpar.c
void par_begin(int *mpisize, int *mpirank)
{
  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
  MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
}

// dummy routine in libser.c
void par_begin(int *sersize, int *serrank)
{
   sersize = 1;
   serrank = 0;
}
```

Example: Global Sum

```
// Globally sum the double precision rainfall value
  rainfall = par_dsum(rainfall);

// parallel function in libpar.c
double par_dsum(double dval)
{
  double dsum;
  MPI_Allreduce(&dval, &dsum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  return dsum;
}

// dummy routine in libser.c
double par_dsum(double dval)
{
  // do nothing!
}
```

Example Makefile

```
# No explicit calls to MPI in any of these files

DEMSRC= \
  demparams.c demrand.c demcoord.c demhalo.c \
  demforce.c demlink.c demcell.c dempos.c demons.c

# All MPI calls contained here
MPISRC= libpar.c
SERSRC= libser.c

# Define serial or parallel source code
PARSRC=$(MPISRC)
#PARSRC=$(SERSRC)

SRC=$(DEMSRC) $(PARSRC)
```

Advantages of Comms Library

- Can compile serial program from same source
  - makes parallel code more readable
  - only need to write error checking or debugging code once, e.g. for all calls to global reductions

- Enables code to be ported to other libraries
  - more efficient but less versatile routines may exist
  - e.g. Cray-specific SHMEM library
  - can choose to only port a subset of the routines

- Comms can be optimised for different MPI libraries
  - e.g. choose the fastest send (**`Ssend`**, **`Send`**, **`Bsend`**?)

Not calling collectives correctly

- Collectives must be called by all processes in communicator
  - this will not work correctly on more than a single process

```
if (rank == 0) MPI_Bcast(x, 10, MPI_INT, 0, MPI_COMM_WORLD);
```

  - an **`Allreduce`** called like this would deadlock

- Compute everything everywhere
  - e.g. use routines such as **`Allreduce`** in preference to **`Reduce`**
  - perhaps the value only really needs to be know on the master
    - but using **`Allreduce`** makes things simpler
    - no serious performance implications

Error checking and reductions

```
// Check for valid results
MPI_Reduce(&partial, &sum,
           1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Bcast(&sum, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
if (sum < 0)
{
  if (rank == 0) printf("Error: sum = %f\n", sum)
  MPI_Finalize();
}
```

- Do not use reduce + broadcast!
  - use allreduce

Sending lots of small messages

```
for (j=0; j < N; j++)
  {
    MPI_Send(&x[0][j], 1, MPI_INT, dest, 0, comm);
  }
```

- Send a single message of size *N*

```
MPI_Send(&x[0][0], N, MPI_INT, dest, 0, comm);
```

- Use a derived type, e.g. a vector, for equivalent loop over *i*
  - e.g. to send x[0][0], x[1][0], ..., x[N-1]][0]

```
MPI_Send(&x[0][0], 1, my_mpi_vector, dest, 0, comm);
```

Programs with specific process counts

- Do not write code like:

```
if (rank == 0) {
  for (i=1; i <= N/4; i++)
    pi = pi + 1.0/(1.0 + pow((((double)i)-0.5)/((double) N),2.0));
} else if (rank == 1)
  for (i=N/4+1; i <= N/2; i++)
    pi = pi + 1.0/(1.0 + pow((((double)i)-0.5)/((double) N),2.0));
} else ...
```

- 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 and *check at runtime* that size = *P* !!
  - a clever Makefile can reduce the need for recompilation
    - only recompile routines that define arrays rather than use them

Incorrect use of non-blocking

```
if (rank == 0) {
  for (i=1; i < size; i++) {
    MPI_Issend(x, 10, MPI_INT, i, 0, comm, &request);
  }
} else MPI_Irecv(x, 10, MPI_INT, 0, 0, comm, &request);

// now start computation
```

- Need multiple requests on rank 0
  - and they *must* be waited on at some later point
- Why use non-blocking here at all?
  - avoid complication unless this is performance critical

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
  - Allinea DDT is becoming more common across the board
  - a commercial product
  - debuggers can powerful tools but also very complicated

General Debugging

- People seem to write programs DELIBERATELY to make them impossible to debug!
  - my favourite: the silent program
  - "my program doesn't work"

```
$ mpirun -n 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?!?!

Common mistake

- There was a bug, but 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
  - start with the serial code
  - then $P = 1$
  - then a small process count ...

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 require reproducibility, but most implementations are
  - without this, debugging is almost impossible

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

Fortran array syntax

- MPI derived types enable strided data to be sent/received
  - no explicit copy in/out required
- For Fortran
  - why not use Fortran array syntax?
- Some subtleties for non-blocking operations

Non-blocking operations

- What is wrong with this code?

```
allocate(buf(n))
call MPI_Issend(buf, n, ....)
deallocate(buf)
```

- Non-blocking send may still be ongoing at deallocation
  - code could crash or give unpredictable behaviour
  - only safe to deallocate the memory after the matching wait
- Identical issues in C using malloc and free
  - however, the problem arises in a more subtle way in Fortran
  - due to its more sophisticated array handling

Fortran array syntax

```
real, dimension(m,n) :: array
call MPI_Issend(array(1,1:n), n, MPI_REAL, ...)
...
```

- Looks ok but compiler will probably do:

```
allocate buf(n)
buf(1:n) = array(1,1:n)
call MPI_Issend(buf, n, MPI_REAL, ...)
array(1,1:n) = buf(1:n)
deallocate(buf)
```

  - so buf may not exist when message is sent
  - issue even more severe for non-blocking receive

Solutions

- Note this *only an issue for non-blocking operations*
  - e.g. can do normal blocking send and receive using array syntax

- Advice
  - avoid array syntax, even for contiguous sections (e.g. columns)

```
call MPI_Issend(array(1,1), m, ...)
```

  - rather than

```
call MPI_Issend(array(1:m,1), m, ...)
```

- Derived datatypes (e.g. vectors) for non-contiguous rows

```
call MPI_Issend(array(1,1), 1, rowtype, ...)
```

Full array support

- Some MPI libraries fully support Fortran array syntax
  - I have seen it mostly via the Fortran 2008 interface

- Check value of variable:

  MPI_SUBARRAYS_SUPPORTED

- I wrote a small test code for this:
  - https://github.com/davidhenty/subarraytest

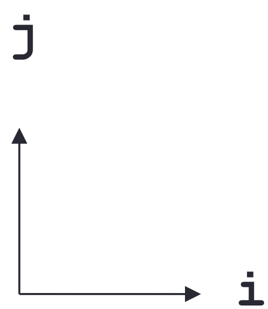
Array allocation issues with C

C: **`x[16]`** F: **`x(16)`**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

C: **`x[4][4]`**

4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13



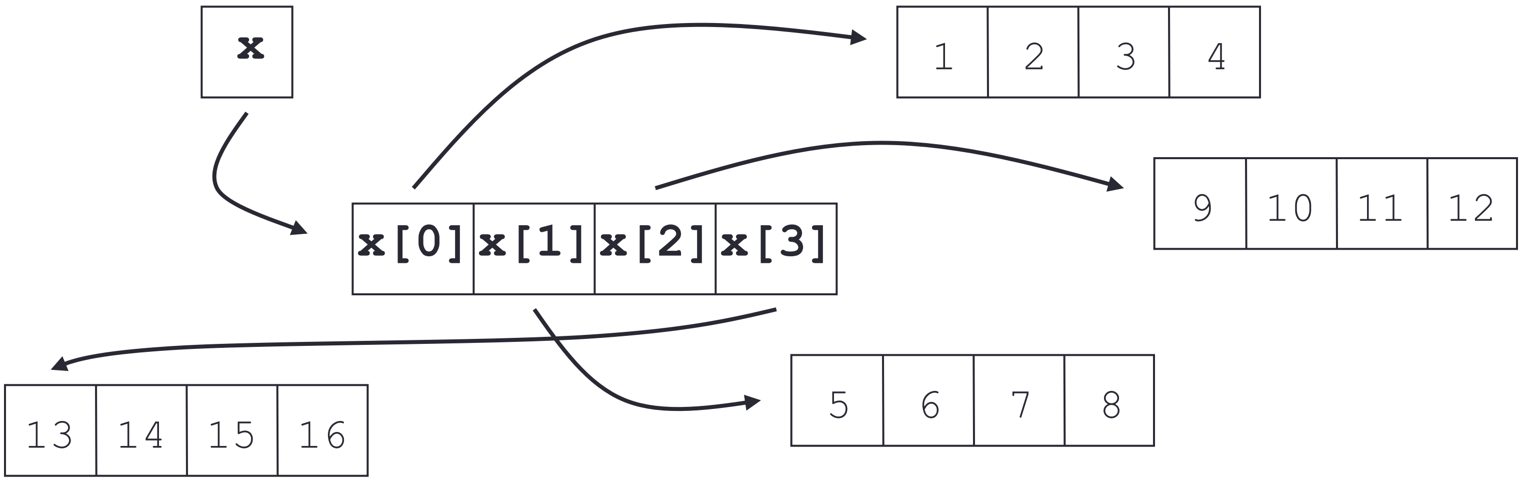
F: **`x(4,4)`**

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

- Data is contiguous in memory
  - different conventions in C and Fortran
  - for statically allocated C arrays **`x == &x[0][0]`**

Aside: Dynamic Arrays in C

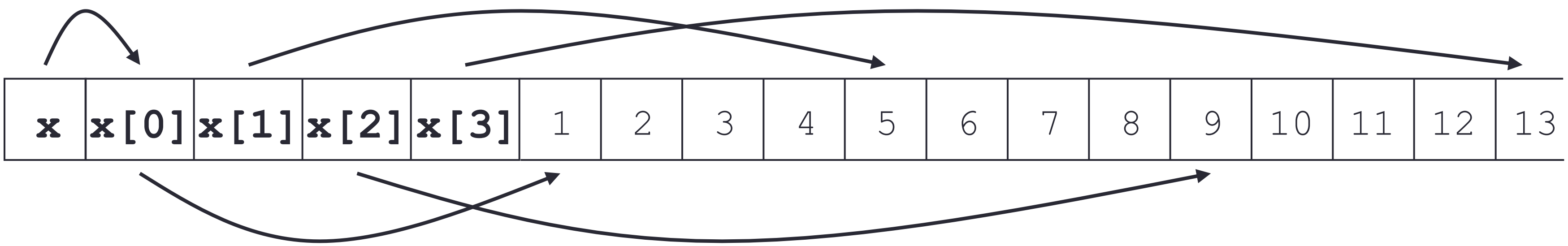
```
int **x = (int **) malloc(4, sizeof(int *));

for (i=0; i < 4; i++)
{
  x[i] = (int *) malloc(4, sizeof(int));
}
```



- Data non-contiguous, and `x != &x[0][0]`
  - cannot use regular templates such as vector datatypes
  - cannot pass `x` to any MPI routine

Arralloc

```
int **x = (int **) arraymalloc2d(4, 4 ,sizeof(int));
/* do some work */
free(x);
```



- Data is now contiguous, but still **`x != &x[0][0]`**
  - can now use regular template such as vector datatype
  - must pass **`&x[0][0]`** (start of contiguous data) to MPI routines
  - see **`MPP-arraymalloc.tar`** for example of use in practice
- Clearest to use always use **`&x[i][j]`** syntax
  - correct for both static and (contiguously allocated) dynamic arrays

Passing arrays to functions (i)

```
#define N 100

void mycode()
{
  int x[N][N]; // this is allocated at compile time

  arrayinit(x);
  ...
}

void arrayinit(int x[N][N])
{
 for (int i = 0; i < N; i++)
    {
       for (int j = 0; j < N; j++)
       {
          x[i][j] = 0;
  ...
```

Passing arrays to functions (2)

```
void mycode()
{
  int n;
  int **x;
  ...
  // read value of n from a file
  ..
  x = (int **) arraymalloc2d(sizeof(int), n, n);  // allocated at runtime

  arrayinit(x, n, n);
  ...
  free(x):
}

void arrayinit(int **x, int n1, int n2)
{
  for (int i = 0; i < n1; i++)
    {
      for (int j = 0; j < n2; j++)
      {
        x[i][j] = 0;
  ...
```

Passing arrays to functions (3)

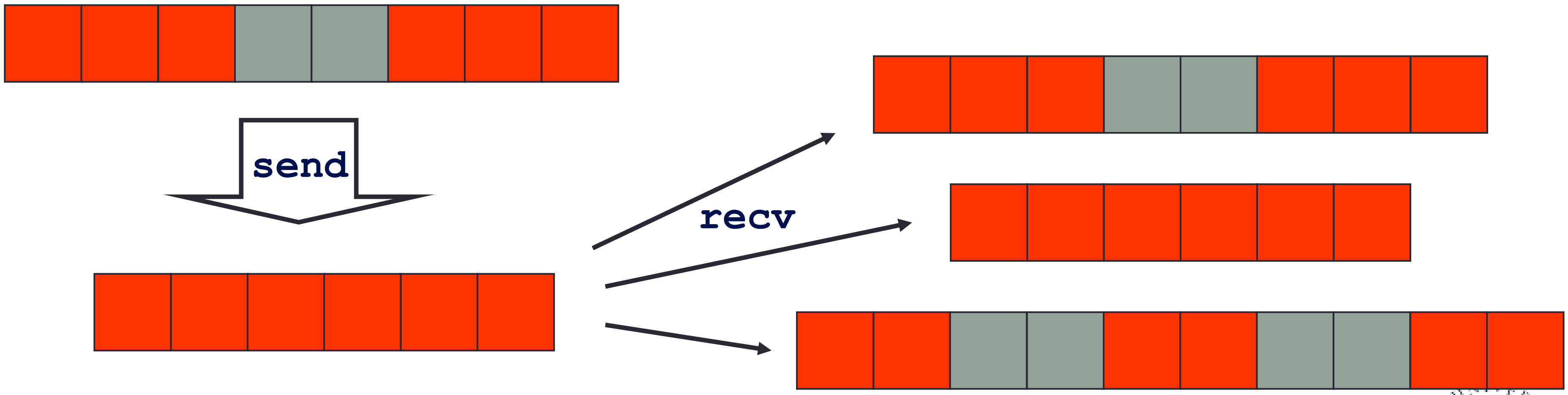
```
void mycode()
{
  int n;
  ...
  // read value of n from a file and define a Variable Length Array
  ..
  int x[n][n]; // allocated at runtime

  arrayinit(n, n, x);
  ...
}

void arrayinit(int n1, int n2, int x[n1][n2])
{
 for (int i = 0; i < n1; i++)
    {
       for (int j = 0; j < n2; j++)
       {
          x[i][j] = 0;
  ...
```

Comments

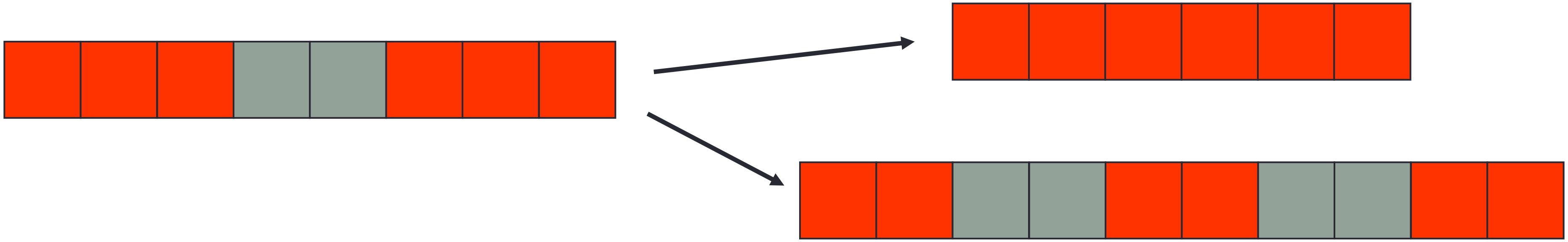
- Fixed sized arrays (#define'd) simple and easy to use
  - perhaps not very elegant as you need to recompile frequently
  - come from the stack
- Dynamically allocated (malloc'd) arrays more complex
  - much more flexible and code looks more elegant
    - remember to deallocate if you no longer need them!
  - come from the heap
- Variable length arrays
  - frowned upon by purists but simple and easy to use
  - come from the stack which is of limited size
- Stack size
  - your code may crash if you try and allocated large arrays
  - increase stack size with: `user@laptop$ ulimit -s unlimited`

Message Matching (i)

- A datatype is defined by two attributes:
  - type signature: a list of the basic datatypes in order
  - type map: the locations (displacements) of each basic datatype
- For a receive to match a send only signatures need to match
  - type map is defined by the receiving datatype
- Think of messages being packed for transmission by sender
  - and independently unpacked by the receiver



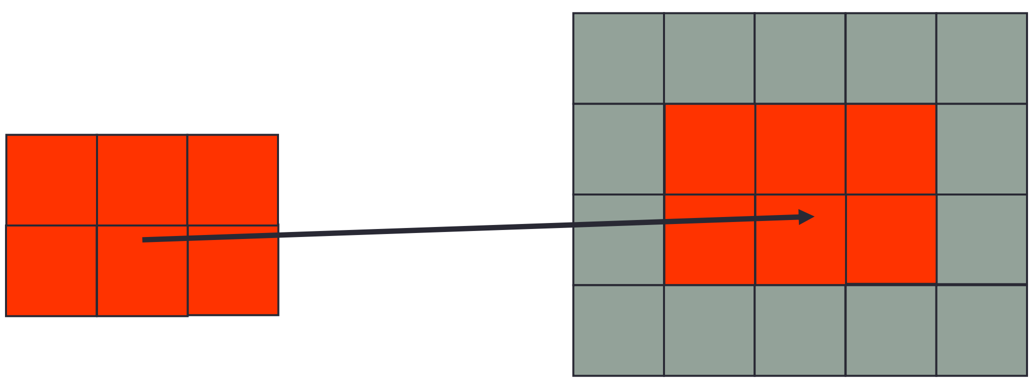
Message Matching (ii)



`Send(1, subarray3x2)` matches `Recv(6, MPI_INT)`

`Send(1, subarray3x2)` matches `Recv(1, subarray2x3)`

- Can be useful when scattering data directly to array with halos



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)