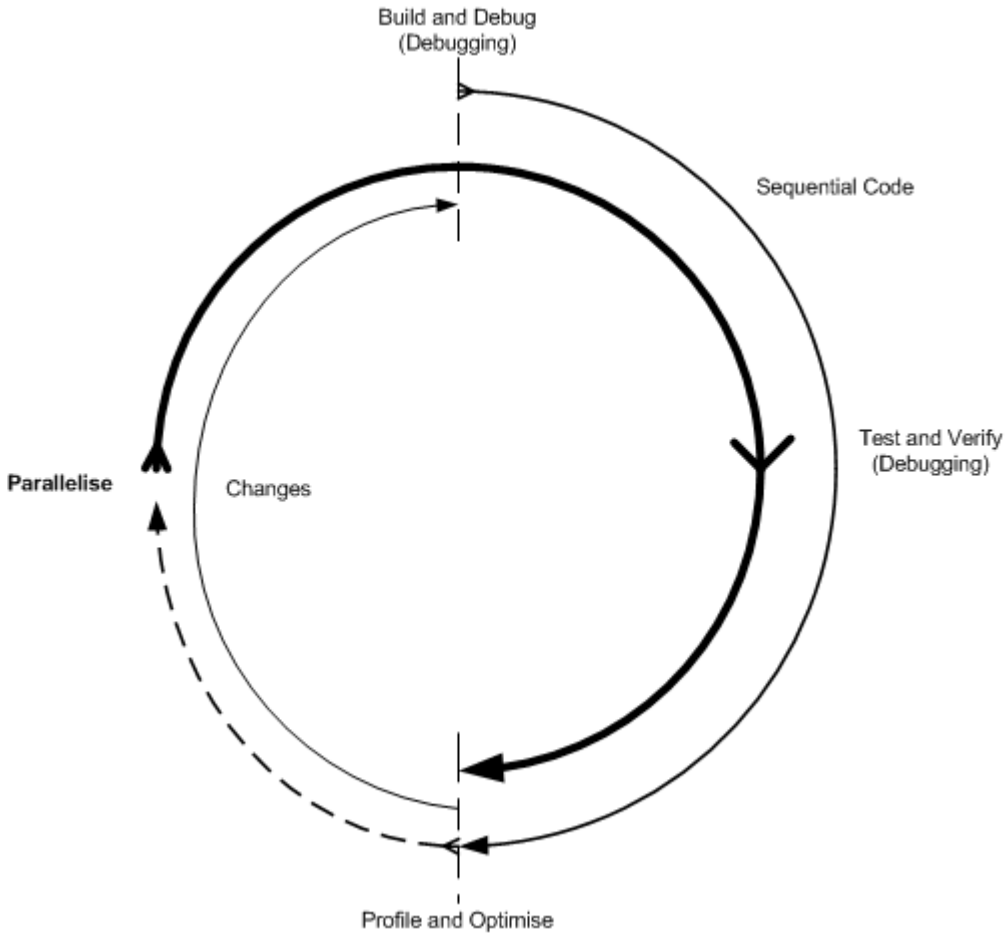
University of East Anglia

Parallelising Scientific Codes Using the Message Passing Interface (MPI)

Wadud Miah
Research Computing Group

Software Performance Lifecycle



Scientific Programming

- Early scientific codes were mainly sequential and were executed on single core CPUs;
- CPU performance accelerated from 1 MHz and has peaked at around 3.2 GHz;
- However, scientists want to a) refine and expand computational domains b) explore challenging scientific problems;
- Solution: parallel programming on many-core architectures.

Distributed Memory Machines

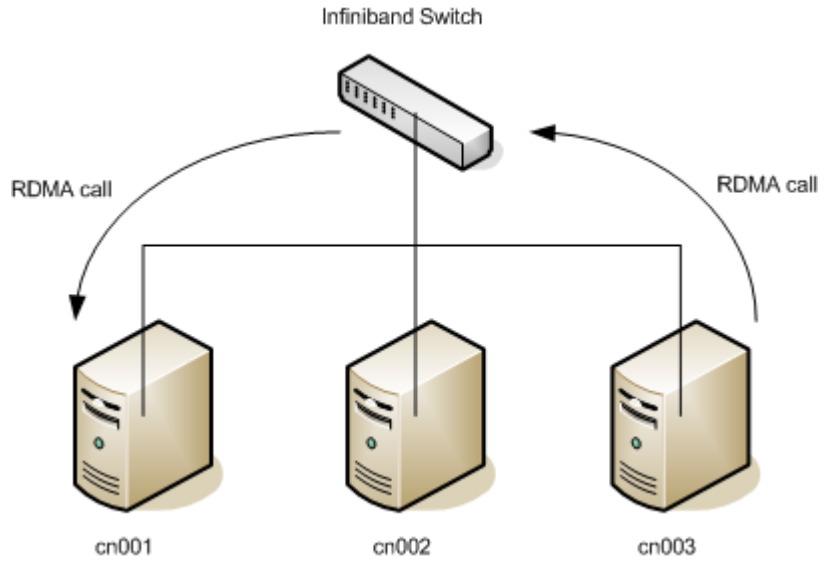
- A distributed memory machine is where processes have two levels of memory: **local memory** and **remote memory**;
- A process accesses **local memory** in the usual manner;
- A process accesses **remote memory** by making a remote direct memory access (RDMA) call;
- An example of a distributed memory machine is a computational cluster, e.g. Grace.

Program Scalability

- OpenMP can only run on a virtual memory machine, e.g. a single compute node;
- MPI programs can run on a number of compute nodes that are defined in the Infiniband queues;
- If efficiently programmed, MPI programs can scale better than OpenMP programs;
- This will depend on the parallel and underlying numerical algorithm.

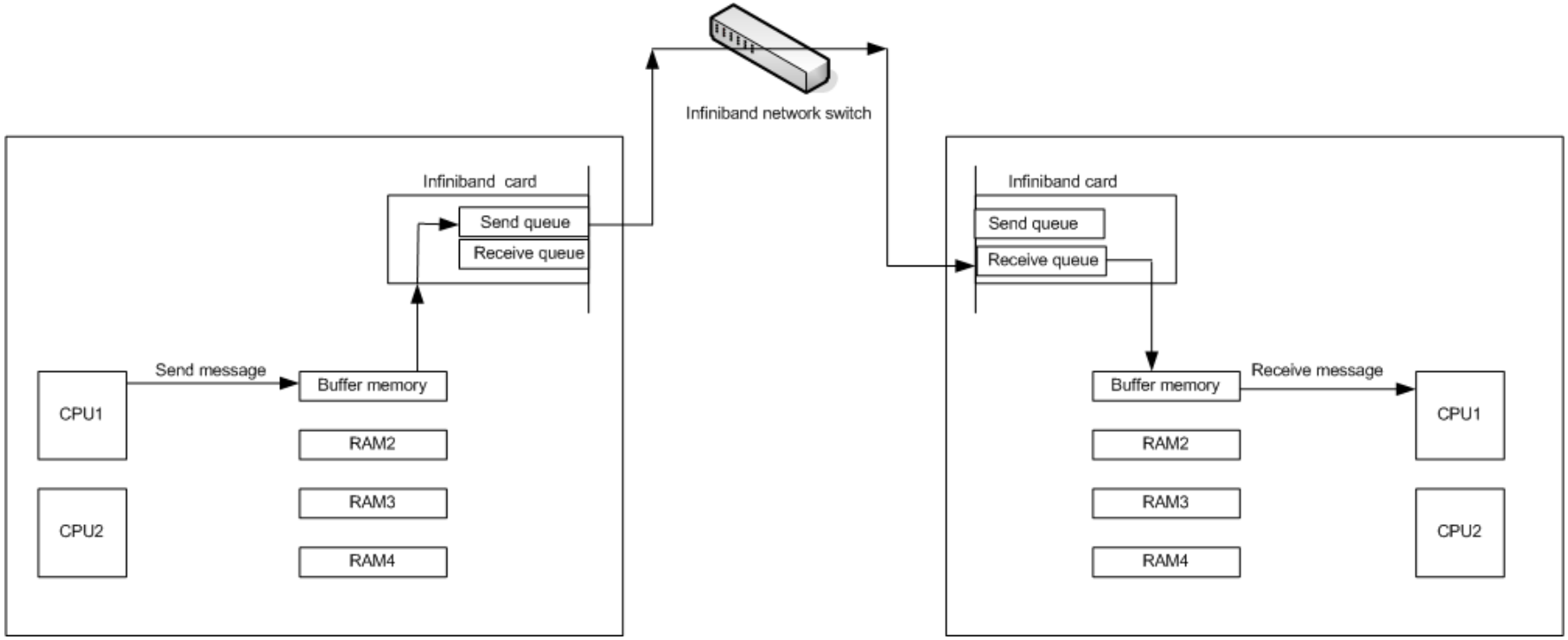
Computational Cluster

- A node has local memory which it access in the normal manner;
- To access remote memory, it has to traverse the Infiniband network:



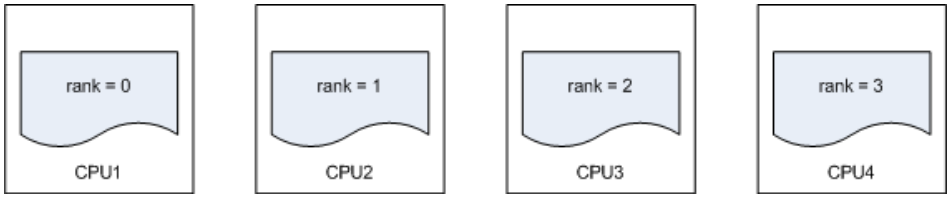
Remote Direct Memory Access

- An RDMA call traverses the network stack:



Single Program Multiple Data

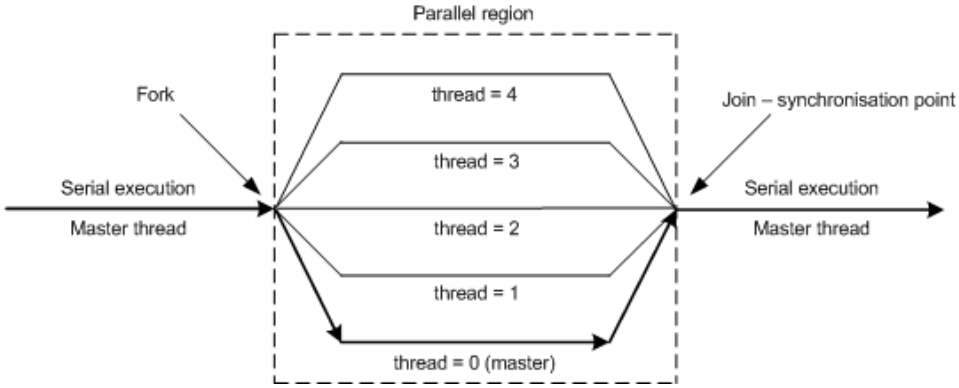
- The MPI specification uses the SPMD model;
- Each process has to use a communication function to send data to another process;
- The same program is executed on all processes with a different rank value numbered from 0 to *N* - 1:



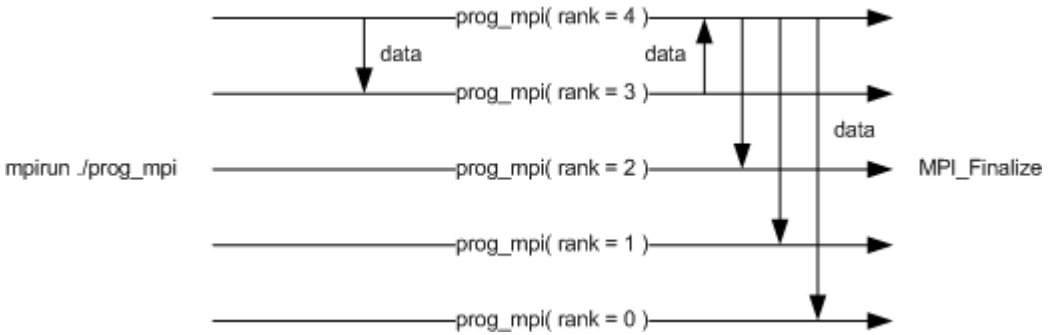
- The whole program is a “parallel region” and the process that finishes last is arbitrary.

Process Execution

- In OpenMP, the fork-join model is used:



- In MPI, all processes run for all time and there is no master process:



Process Memory Structure

- In OpenMP, threads shared the memory of the main process where race conditions are more likely to occur;
- In MPI, each process has its own memory stack and there is no sharing of memory;
- MPI processes have to communicate with each other to access another process' memory stack;
- Less chance of race conditions occurring in MPI.

Message Passing Interface - MPI

- MPI is a standardised and portable message passing specification for distributed memory systems;
- MPI allows developers to write parallel programs using a catalogue of functions or subroutines;
- Processes in MPI programs can communicate with each other by passing messages, hence the name MPI.

Types of Communication

- **Blocking** - the operation is blocked until the operation has completed and the buffer can be re-used;
- **Non-blocking** – the operation is not blocked and is only started; buffer cannot be re-used;
- **Synchronous** - data transfer is only completed when both send and receive is completed;
- **Asynchronous** - data send does not need to be coordinated with a receive. Data is held in the buffer until a receive call.

Communication Patterns

- **Point-to-point** (one-to-one): a single sending process and a single receiving process;
- **One-to-many** (collective): a single sending process and multiple receiving processes;
- **Many-to-one** (reduction): many sending processes and one receiving process;
- **Many-to-many** (collective): multiple sending processes and multiple receiving processes;
- The most common are point-to-point and one-to-many.

MPI Parallel Bugs (1)

- Race condition: non-blocking operation's buffer is re-used too quickly;
- Deadlock: processes waiting for each other forever:

```
if ( rank == 0 ) {
   receive from rank 1
   send to rank 1
} else if ( rank == 1 ) {
   receive from rank 0
   send to rank 0
}
```

MPI Parallel Bugs (2)

- A safe implementation is:

```
if ( rank == 0 ) {
   send to rank 1
   receive from rank 1
} else if ( rank == 1 ) {
   receive from rank 0
   send to rank 0
}
```

- Deadlocks can be easily identified using parallel debuggers as the program counter stops at the problematic location.

MPI Communicators

- MPI operations are bound within a communicator handle;
- A communicator is a set of processes where communication is bound within this set;
- Communicators can be created and destroyed dynamically;
- The default communicator is `MPI_COMM_WORLD` which contains all processes;
- All user defined communicators are a proper subset of `MPI_COMM_WORLD`

MPI Send (1)

- To send data, use the following function:

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

MPI Send (2)

- `buffer` - specifies the send buffer;
- `count` - number of elements to send;
- `datatype` - data type of the buffer;
- `dest` - specifies the rank of the destination process;
- `tag` - a value that is used to uniquely pair a send and receive;
- `comm` - specifies the communicator.

MPI Receive (1)

- To receive data, use the following function:

```
int MPI_Recv( void *buffer,
              int count,
              MPI_Datatype datatype,
              int source,
              int tag,
              MPI_Comm comm,
              MPI_Status *status );
```

MPI Receive (2)

- `buffer` - specifies the receive buffer;
- `count` - number of elements to send;
- `datatype` - data type of the buffer;
- `source` - specifies the rank of the source process;
- `tag` - a value that is used to uniquely pair a send and receive;
- `comm` - specifies the communicator;
- `status` - contains the status of the receive.

MPI Send and Receive

- A combined send and receive function which prevents deadlocks:

```
int MPI_Sendrecv(
 void *sendbuf, int sendcount,
 MPI_Datatype sendtype,
 int dest, int sendtag,
 void *recvbuf, int recvcount,
 MPI_Datatype recvtype
 int source, int recvtag,
 MPI_Comm comm,
 MPI_Status *status );
```

MPI C Data Types

MPI Data Type	Intrinsic Data Type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	single byte value

MPI Fortran Data Types

MPI Data Type	Intrinsic Data Type
MPI_CHARACTER	character
MPI_INTEGER	integer
MPI_REAL	real
MPI_REAL8	real*8
MPI_DOUBLE_PRECISION	double precision
MPI_LOGICAL	logical
MPI_COMPLEX	complex

MPI C Program

```
#include <stdio.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 );

  print( “I am %d of %d\n”, rank, size );
  MPI_Finalize( );
}
```

MPI Fortran Program

```
program mpi
 include 'mpif.h'
 integer ierr, size, rank

 call MPI_INIT( ierr )
 call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
 call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )

 print *, 'I am ', rank, ' of ', size
 call MPI_FINALIZE( ierr )

end program
```

Non-blocking Operations

- Blocking operations can result in idle processes as they have to wait for operation to complete;
- Waiting times can be overlapped with computation to hide latency;
- Non-blocking operations initiate the transfer and returns control to the process immediately;
- Developer will have to coordinate the buffer transfer and computation.

Non-blocking Send

```
int MPI_Isend( void *buffer,
               int count,
               MPI_Datatype type,
               int dest,
               int tag,
               MPI_Comm comm,
               MPI_Request *request );
```

- The variable `request` is used to identify the operation.

Non-blocking Receive

```
int MPI_Irecv( void *buffer,
               int count,
               MPI_Datatype type,
               int source,
               int tag,
               MPI_Comm comm,
               MPI_Request *request );
```

- The variable `request` is used to identify the operation.

Testing for Completion

- Non-blocking operations need to be tested for completion using:

```
int MPI_Test( MPI_Request *request,
     int *flag, MPI_Status *status );
```

- If `flag` = 1, operation has completed;
- Following function blocks/waits until operation identified by `request` has completed:

```
int MPI_Wait( MPI_Request *request,
              MPI_Status *status );
```

Collective Communication

- When more than two processes are involved in communication;
- This can be one-to-many (broadcast), many-to-one (reduction) and many-to-many;
- All participating processes must execute the same collective subroutine with the same parameters;
- All collective operations are blocking.

MPI Broadcast

- A single process broadcasts data to other participating processes:

```
int MPI_Bcast( void *buffer,
               int count,
               MPI_Datatype type,
               int root,
               MPI_Comm comm );
```

- No tag facility is provided;
- All arguments except `buffer` have to be identical.

MPI Reduction

- A single process receives data from other participating processes:

```
int MPI_Reduce( void *sendbuf,
                void *recvbuf,
                int count,
                MPI_Datatype type,
                MPI_Op operation
                int root,
                MPI_Comm comm );
```

- The `operation` indicates which binary operation to execute.

MPI Reduction Operations

MPI Representation	Mathematical Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum value and index
MPI_MINLOC	Minimum value and index

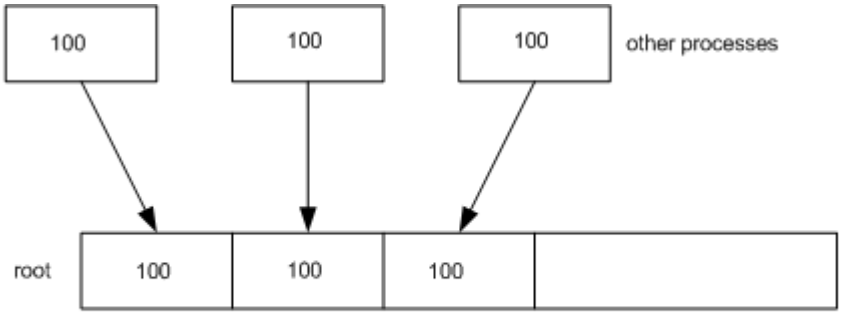
MPI Gather (1)

- A root process collects blocks of data from all other participating processes:

```
int MPI_Gather( void *sendbuf,
 int sendcount,
 MPI_Datatype sendtype,
 void *recvbuf,
 int recvcount,
 MPI_Datatype recvtype,
 int root,
 MPI_Comm comm );
```

MPI Gather (2)

- All processes must execute this with the same `root`, `sendtype` and `sendcount`.



MPI_Gather(sendbuf, 100, MPI_INT, recvbuf, 100, MPI_INT, root, comm);

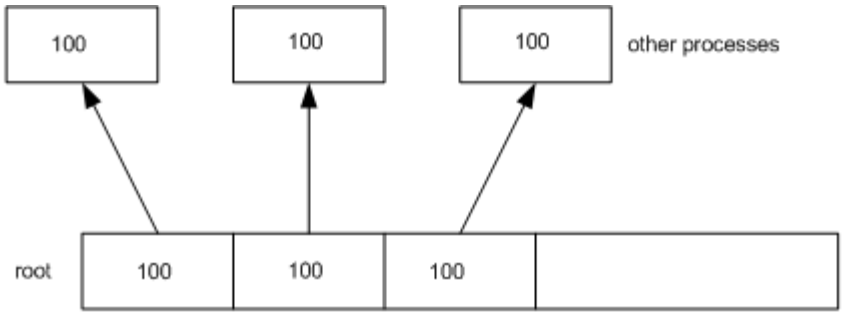
MPI Scatter (1)

- A root process sends different blocks of data to different processes:

```
int MPI_Scatter( void *sendbuf,
 int sendcount,
 MPI_Datatype sendtype,
 void *recvbuf,
 int recvcount,
 MPI_Datatype recvtype,
 int root,
 MPI_Comm comm );
```

MPI Scatter (1)

- All processes must execute this with the same `root`, `sendtype` and `sendcount`;
- The receiving process *`i`* receives `sendbuff[`*`i*sendcount`*`, …, (i+1)*sendcount`*`]`



MPI_Scatter(sendbuf, 100, MPI_INT, recvbuf, 100, MPI_INT, root, comm);

MPI AllGather (1)

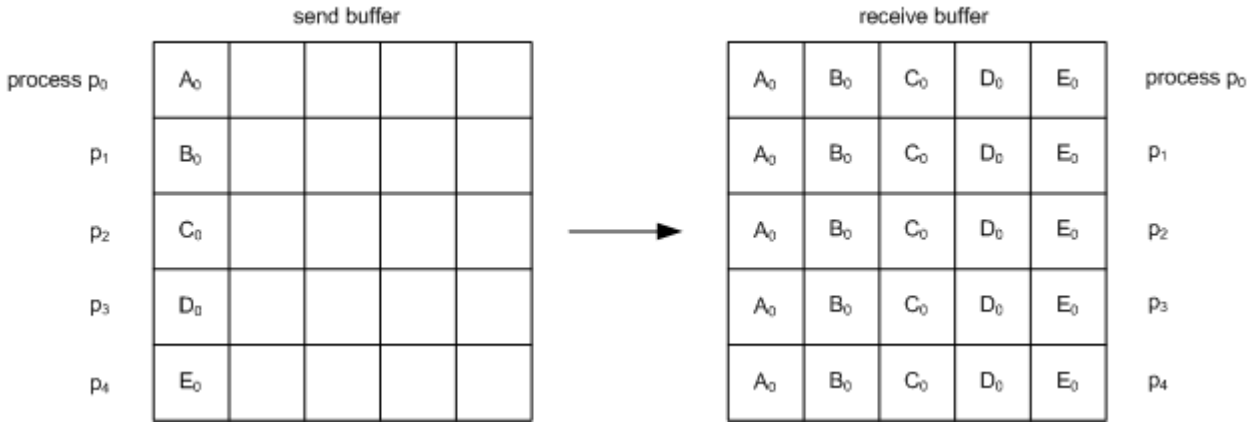
- AllGather sends local blocks of data to all other participating processes;

```
int MPI_Allgather( void *sendbuf,
  int sendcount, MPI_Datatype sendtype,
  void *recvbuf, int recvcount,
  MPI_Datatype recvtype,
  MPI_Comm comm );
```

- For p processes $recvount = p * sendcount$

MPI AllGather (2)

- Can be viewed pictorially:



MPI AllToAll (1)

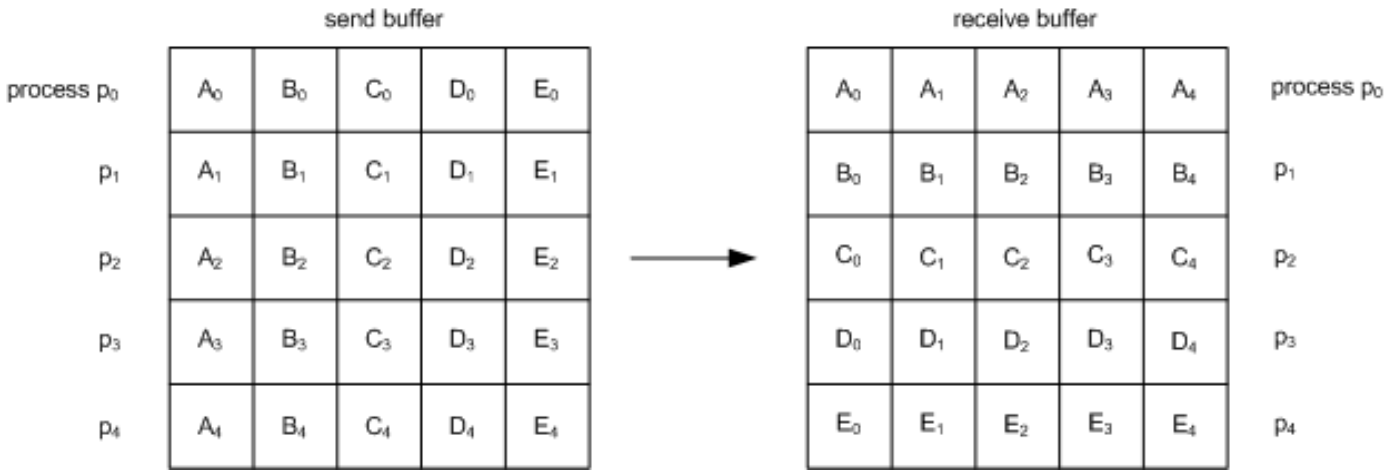
- All processes send their local block to all other participating processes;

```
int MPI_Alltoall( void *sendbuf,
 int sendcount, MPI_Datatype sendtype,
 void *recvbuf, int recvcount,
 MPI_Datatype recvtype,
 MPI_Comm comm );
```

- For p processes $recvount = p * sendcount$
- Equivalent to p sends and p receives for all p processes.

MPI AllToAll (2)

- The j-th block from process i is received by process j and stored the i-th block;



MPI Synchronisation

- To synchronise a group of processes, use:

```
int MPI_Barrier( MPI_Comm comm );
```

- This needs to be invoked by all participating processes.

Groups and Communicators (1)

- MPI allows the creation of groups of processes and subsequent communication handles for such groups;
- This is useful if collective communication should be confined within process groups;
- To obtain a new group from a current communicator, use:

```
int MPI_Comm_group( MPI_Comm comm
                    MPI_Group *group);
```

where `comm = MPI_COMM_WORLD`

Groups and Communicators (2)

- Then select the ranks that should be included in the new group:

```
int MPI_Group_incl( MPI_Group group,
                    int p, int *ranks
                    MPI_Group *new_group );
```

- Creates processes 0, …, p - 1 with process i has rank `ranks[i]`
- Ranks can also be excluded:

```
int MPI_Group_excl( MPI_Group group,
                    int p, int *ranks
                    MPI_Group *new_group );
```

Groups and Communicators (3)

- To get the size of a group:

```
int MPI_Group_size( MPI_Group group,
                    int *size );
```

- To get the rank of a calling process:

```
int MPI_Group_rank( MPI_Group group,
                    int *rank );
```

- To de-allocate a group:

```
int MPI_Group_free( MPI_Group *group );
```

Groups and Communicators (4)

- To create a new communication handle:

```
int MPI_Comm_create( MPI_Comm comm,
                     MPI_Group group,
                     MPI_Comm *new_comm );
```

- To get the size and rank of the calling process:

```
int MPI_Comm_size( MPI_Comm comm,
                   int *size );
int MPI_Comm_rank( MPI_Comm comm,
                   int *rank );
```

- To de-allocate a communicator:

```
int MPI_Comm_free( MPI_Comm *comm );
```

Process Topologies

- MPI allows the creation of a virtual topology that conveniently fits in with a computational domain;
- This can simplify code development;
- Can allow MPI to optimise communication between processes;
- The new topology creates a new communicator;
- Examples include operating on two- and three-dimensional computational grids.

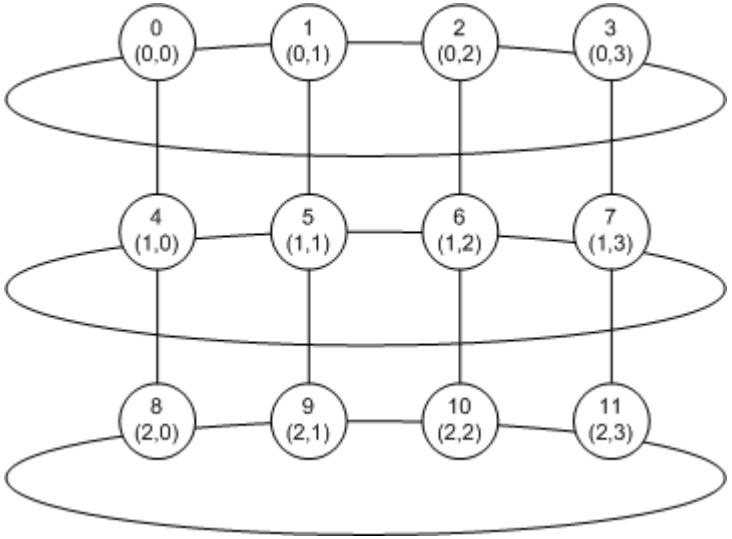
MPI Cartesian Create (1)

```
int MPI_Cart_create( MPI_Comm comm,
                     int ndims, int *dims,
                     int *periods, int reorder,
                     MPI_Comm *new_comm );
```

- `ndims` - number of dimensions;
- `dims` - array of size `ndims` where `dims[i]` indicates number of processes in dimension `i`;
- `periods` - Boolean array indicating whether direction `i` is cyclic;
- `reorder` - internal optimisations;
- `new_comm` - the new communicator.

MPI Cartesian Create (2)

- For `ndims = 2, dims[0] = 3, dims[1] = 4`
- `periods[0] = 1, periods[1] = 0` and `reorder = 0`



MPI Cartesian Mapping (1)

- To get process rank from coordinate:

```
int MPI_Cart_rank( MPI_Comm comm,
                   int *coords, int *rank );
```

- To get coordinates from rank:

```
int MPI_Cart_coords( MPI_Comm comm,
                     int rank, int ndims, *coords );
```

MPI Cartesian Mapping (2)

- To get rank of neighbouring processes:

```
int MPI_Cart_shift( MPI_Comm comm,
    int dir, int disp,
    int *rank_source, int *rank_dest );
```

- `dir` - direction of shift;
- `disp` - length of shift in processor coordinates (+ or -);
- `rank_source` - where calling process should receive data from;
- `rank_dest` - where calling process should send data to.

MPI Cartesian Mapping (3)

- For example two-dimensional operation:

```
MPI_Cart_shift( comm, 0, 1, &west,
                            &east );
MPI_Cart_shift( comm, 1, 1, &north,
                            &south );
```

- Neighbours are `west`, `east`, `north` and `south`;
- Values can be used in an `MPI_Send` and `MPI_Recv`.

MPI Timing Routines

- Time since a fixed point:

```
double MPI_Wtime( void );
```

- Clock resolution:

```
double MPI_Wtick( void );
```

- For example:

```
t1 = MPI_Wtime( );
// lots of work here
t2 = MPI_Wtime( );
dt = t2 - t1; // time in seconds
```

MPI Abort

- To abort an MPI program due to the discovery of a catastrophe, use:

```
int MPI_Abort( MPI_Comm comm,
               int code );
```

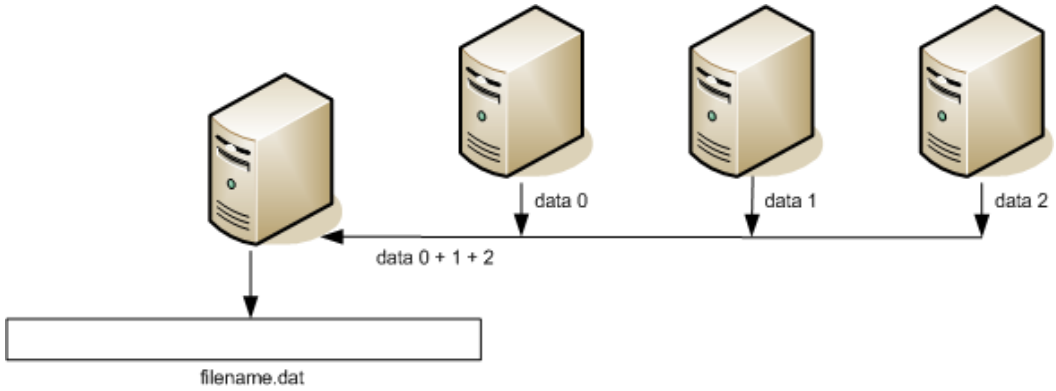
The code used will be the code returned by the program.

Parallel Input/Output

- Files are generally handled by a single process;
- This has resulted in codes with a single process tasked to do IO whilst other processes are sat idle;
- This is insufficient for parallel programs, so a parallel IO library has been created for MPI programs called MPI-IO;
- This allows multiple processes to manipulate distinct parts of a single file in parallel;
- This results in greater bandwidth, speed and code scalability.

Traditional IO methods (1)

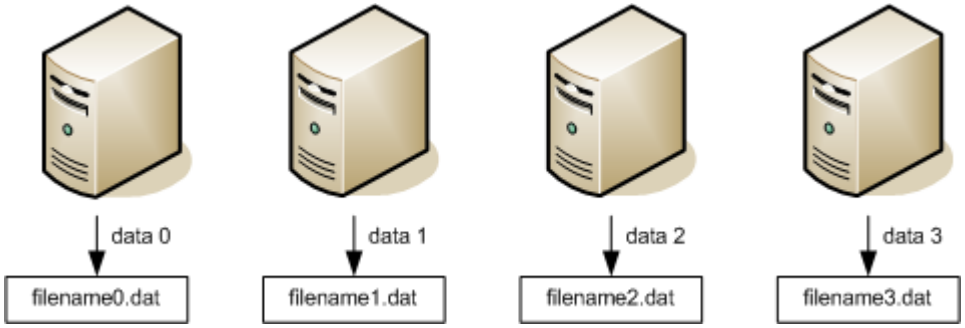
- Single node is responsible for IO:



- Bottleneck exists at the writing process;
- Reduced scalability;
- All memory may not even fit into a single node.

Traditional IO methods (2)

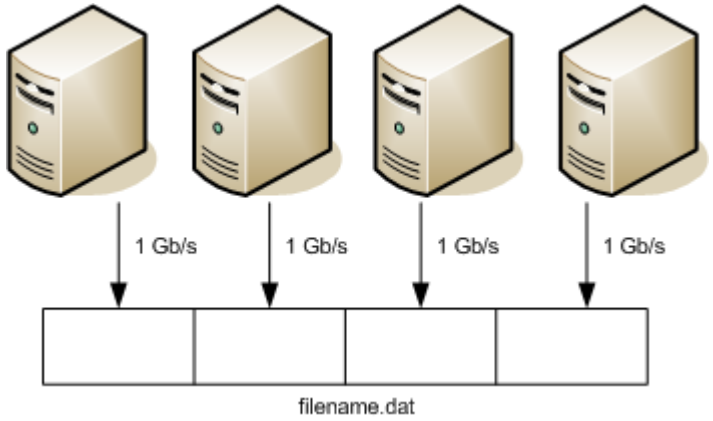
- All nodes write to their own file



- Resulting in a large number of files;
- This creates file management complications.

True Parallel Input/Output

- Each compute node's network port is used in the IO which is 1 Gb/s;
- This useful for codes that checkpoint their data for very long runs;



- The MPI-IO library has internal optimisations for better performance than traditional methods.

Parallel File Operations

- Firstly, the file has to be opened and a file handle is created. All operations refer to this file handle;
- The process' position in the file is set. This states that a process can access a certain part of the file;
- More than one process writes to the same position in the file → race condition;
- Data is read from or written to a file in parallel;
- The file is finally closed;
- Data is written in binary format - faster reading and writing.

Parallel File Open in C

- To open a file for parallel operation:

```
int MPI_file_open( MPI_Comm comm,
   char *fn, int mode, MPI_Info info,
   MPI_File *fh );
```

- `comm` - communicator;
- `fn` - the name of the file;
- `mode` - file open mode: `MPI_MODE_CREATE`, `MPI_MODE_WRONLY`, `MPI_MODE_RDONLY`;
- `info` - stores information regarding the call;
- `fh` - MPI file handle.

Parallel File Open for Fortran

- To open a file for parallel operation:

```
MPI_FILE_OPEN( comm, fn, mode, info,
               fh, ierr );
character *(*) fn
integer comm, mode, info, fh, ierr
```

- `ierr` - error status;
- Other parameters are the same as in the C function.

Parallel File Set View in C

- To set a process' position in a file:

```
int MPI_file_set_view( MPI_File fh,
   MPI_Offset disp, MPI_Datatype etype,
   MPI_Datatype filetype, char *datarep,
   MPI_Info *info );
```

- `disp` - where in the file the process should position itself;
- `etype` - unit of data, e.g. `MPI_INT`, `MPI_FLOAT`;
- `filetype` - type of data in the file, e.g. `MPI_INT`;
- `datarep` - data representation, e.g. `native` implies store data as it is represented in memory.

Parallel File Set View in Fortran

- To set a process' position in a file :

```
MPI_file_set_view( fh, disp, etype,
filetype, datarep, info, ierr )
integer fh, etype, filetype, info, ierr
character *(*) datarep
integer( kind=MPI_OFFSET_KIND ) disp
```

- `disp` - where in the file the process should position itself must of kind `MPI_OFFSET_KIND` to ensure portability and correctness;
- Other parameters are the same as in the C function.

Parallel File Write in C

- To write data to a file:

```
int MPI_file_write( MPI_File fh,
 void *buf, int count,
 MPI_Datatype datatype,
 MPI_Status *status );
```

- `buf` - data that is required to be written;
- `count` - the number of elements in `buf` that need to be written;
- `datatype` - the type of data the needs to be written.

Parallel File Write in Fortran

- To write data to a file:

```
MPI_FILE_WRITE( fh, buf, count,
datatype, status, ierr )
<type> buf(*)
integer fh, count, datatype
integer status(MPI_STATUS_SIZE), ierr
```

- `buf` - data that is required to be written;
- `count` - the number of elements in `buf` that need to be written;
- `datatype` - the type of data the needs to be written.

Parallel File Read in C

- To read data from a file:

```
int MPI_file_read( MPI_File fh,
 void *buf, int count,
 MPI_Datatype datatype,
 MPI_Status *status );
```

- `buf` - address of buffer that will store the data;
- `count` - the number of elements in `buf` that need to be read;
- `datatype` - the type of data the needs to be read.

Parallel File Read in Fortran

- To read data from a file:

```
MPI_FILE_READ( fh, buf, count,
datatype, status, ierr )
<type> buf(*)
integer fh, count, datatype
integer status(MPI_STATUS_SIZE), ierr
```

- `buf` - address of buffer that will store the data;
- `count` - the number of elements in `buf` that need to be read;
- `datatype` - the type of data the needs to be read.

Getting File Size

- In C:

```
int MPI_File_get_size( MPI_File fh,
  MPI_Offset *size );
```

- In Fortran:

```
MPI_FILE_GET_SIZE( fh, size, ierr )
  integer fh, size, ierr
```

File Close

- In C:

```
int MPI_File_close( MPI_File *fh );
```

- In Fortran:

```
MPI_FILE_CLOSE( fh, ierr )
   integer fh, ierr
```

Parallel NetCDF

- The parallel NetCDF completes reads/writes in parallel using the MPI-IO library;
- The Grace module files are:

```
pnetcdf/gcc/1.2.0
pnetcdf/intel/1.2.0
pnetcdf/pgi/1.2.0
```

- The static library file name is:

```
libpnetcdf.a
```

Parallel HDF5

- The parallel HDF5 completes reads/writes in parallel using the MPI-IO library;
- The Grace module files are:

  `phdf/5-1.8.5`
- The static libraries file names are:

  `libhdf5.a`
  `libhdf5_fortran.a`
  `libhdf5_hl.a`
  `libhdf5hl_fortran.a`

Conclusion

- MPI allows developers to write parallel programs for distributed memory systems;
- It is important to develop scalable parallel programs which depends on the parallel algorithm and the underlying numerical algorithm;
- A challenge in writing parallel programs is thinking in parallel - once you understand the concepts, implementation should be easier;
- The future of computational science is in programming in many-core environments.

Practical Exercises

- Load the correct module files:

```
module load mpi/platform/intel/8.2.1
module load icc/intel/12.1
```

- To compile, use the commands:

```
mpicc program.c -o program
mpiCC program.cc -o program
mpif90 program.f90 -o program
mpif77 program.f -o program
```

- To submit a job to LSF:

```
bsub < mpi_job.bsub
```

References

1. *Parallel Programming with MPI*. P. Pacheco;
2. *Parallel Programming for Multicore and Cluster Systems*. T. Rauber and G. Runger;
3. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. W. Gropp, *et al*;
4. *Using MPI-2: Advanced Features of the Message Passing Interface*. W. Gropp, *et al*;
5. Parallel NetCDF
   http://trac.mcs.anl.gov/projects/parallel-netcdf/
6. Parallel HDF5
   http://www.hdfgroup.org/HDF5/PHDF5/