COMP90025 Parallel and Multicore Computing

MPI Tutorial
with contributions from Xinyi Xu

Lachlan Andrew

School of Computing and Information Systems
The University of Melbourne

2019 Semester II

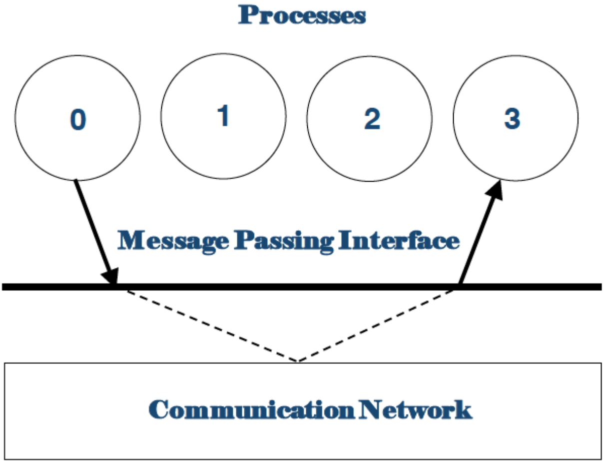
Overview

- MPI Introduction
  - Message Passing Model
  - Communication modes
  - MPI basics
- Point-to-Point Communications
  - Sending a message
  - Receiving a message
  - ping example
- Collective Communications
  - Barrier Synchronization
  - Broadcast, scatter, gather
  - Reduce - average example

Message Passing Model

- The message passing model is based on the notion of processes
- In the message passing model, parallelism is achieved by having many processes co-operate on the same task
- Each process has access only to its own data
- Processes communicate with each other by sending and receiving messages

Parallel Paradigm



Reference: `http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/`

Messages

- A message transfers a number of data items of a certain type from the memory of one process to the memory of another process
- A message typically contains
  - the ID of the sending processor
  - the ID of the receiving processor
  - the type of the data items
  - the number of data items
  - the data itself
  - a message type identifier

Communication modes

- Sending a message can either be synchronous or asynchronous
  - More options in the standard `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`
- A synchronous send is not completed until the message has started to be received
- An asynchronous send completes as soon as the message has gone
- Receives are usually synchronous - the receiving process must wait until the message arrives
  - Why?

What is MPI

- First message-passing interface standard.
- Message Passing Interface document produced in 1993
- MPI is a library of functions/subroutine calls
- MPI is not a language, there is no such thing as an MPI compiler
    - CUDA refers to an MPI compiler. It means something like `mpicxx`
    - This is just a wrapper for a C++ compiler that gives the paths to the MPI library and headers

Initializing and exiting MPI

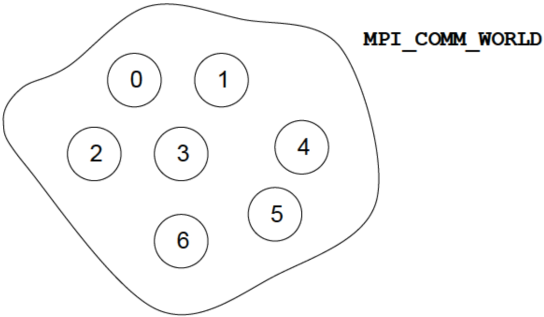
```
int MPI_Init(int *argc, char ***argv)
```

- Initializes the MPI execution environment.
- Must be the first MPI procedure called.
- Note the extra * in front of `argc` and `**argv`.

```
int MPI_Finalize()
```

- Terminates the MPI execution environment.
- Must be the last MPI procedure called, no other MPI routines may be called after it.

Communicators



- MPI uses *communicators* to define which collection of processes may communicate with each other.
- Most MPI routines require you to specify a communicator as an argument.
- `MPI_COMM_WORLD` is the predefined communicator that includes all of your MPI processes.

Reference:http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/

Rank and Size

How do you identify which process within a communicator you are?

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

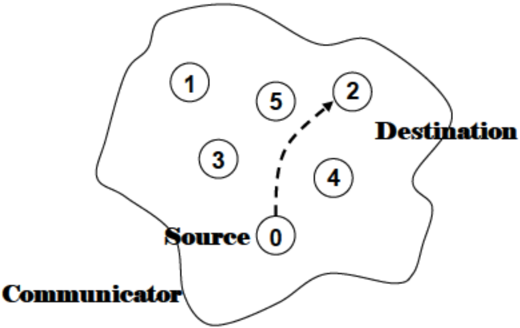
Numbering is always 0, 1, 2, ..., N-1.

- Can't compare with other communicators

How many processes are contained within a communicator?

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

Point-to-Point Communication



- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator.
- Destination process is identified by its rank in the communicator.

Point-to-Point Communication

- Sender calls a SEND routine
- Receiver calls a RECEIVE routine
- Data goes into the receive buffer
- Metadata describing message also transferred

Sending a message

Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse.

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

E.g. send data from rank 1 to rank 3

```
int x;
...
if (rank == 1)
MPI_Send(&x, 1, MPI_INT, /*dest=*/3, /*tag=*/0,
                 MPI_COMM_WORLD);
```

Receiving a message

Receive a message and block until the requested data is available in the application buffer in the receiving task.

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

Status indicates the source of the message, the tag of the message, and actual number of bytes received

E.g. Receive data from rank 1 on rank 3

```
int y;
...
if (rank == 3)
MPI_Recv(&y, 1, MPI_INT, /*src=*/1, /*tag=*/0,
                 MPI_COMM_WORLD, &status);
```

For a communication to succeed

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message types must match.
- Receiver's buffer must be large enough.
    - Need not hold the whole message, if the receiving process is extracting while the sender is sending

Collective Communications

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples
  - Barrier synchronisation.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

Barrier Synchronization

Synchronization operation. Creates a barrier synchronization in a group.

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

- When reaching the `MPI_Barrier` call, each task blocks until all tasks in the group reach the same `MPI_Barrier` call.
- Used less than in shared-memory synchronization, because we're not waiting for data structures in memory to become ready.
- Useful if we are sharing an OS resource that is not controlled by MPI
- e.g., wanting to write output in the "correct" order
  stackoverflow.com/questions/13305814/when-do-i-need-to-use-mpi-

Broadcast

Broadcasts (sends) a message from the process with rank “root” to all other processes in the group.

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

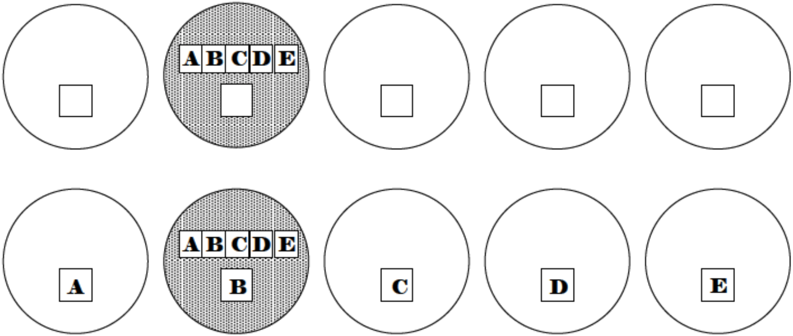
E.g.

```
MPI_Bcast(overallmin,2,MPI_INT,0, MPI_COMM_WORLD);
```

All the nodes in the group execute this line. The only difference is the action; most nodes participate by receiving, while node 0 participates by sending.

Scatter

It breaks long data into chunks which it parcels out to individual nodes (including itself).



Reference: http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/

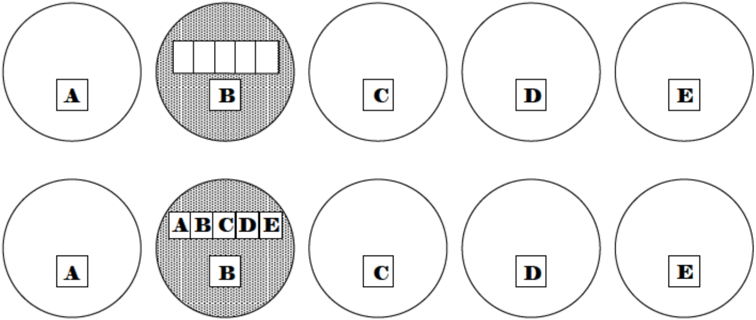
Scatter

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

Examples: `scatter.c`
(found at `.../mpi/scatter.c`)

Gather

Stringing everything together in node order and depositing it all in the program running at Node root.



Reference:`http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/`

Gather

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

- all nodes participate in a gather operation
- each node (including Node root) contributes `sendcount` MPI integers
- from a location `sendbuf`
- Node root then receives `sendcount` items sent from each node

```
MPI_Allgather (&sendbuf,sendcount,sendtype,
        &recvbuf,recvcount,recvtype,comm)
```

- places the result at all nodes, not just one.

Reduce

Applies a reduction operation on all tasks in the group and places the result in one task.

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
        int count, MPI_Datatype datatype,
        MPI_Op op, int root, MPI_Comm comm)
```

E.g.

```
MPI_Reduce(mysum,overallsum,1,MPI_INT,MPI_SUM,
        0,MPI_COMM_WORLD);
```

- type of reduce operation is `MPI_SUM` (sum value)
- Each node contributes a value to be checked, from a location mysum
- type of the pair is `MPI_INT`
- The overall sum value will be computed by combining all of these values at node 0, where they will be placed at a location overallsum

Example found at .../mpi/reduce_avg.c

```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
           MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
         global_sum / (world_size * num_elements_per_proc));
}
```

Overview

- MPI modes (Ssend, Bsend and Send)
- Meaning and use of communicator
- Multithreading

Modes

- `MPI_Send` (standard Send)
  - may be implemented as synchronous or asynchronous send
  - this may cause a lot of confusion
  - also the most efficient, since it gives the most flexibility to the system
- `MPI_Ssend` (Synchronous Send)
  - guaranteed to be synchronous
  - routine will not return until message has been delivered
- `MPI_Bsend` (Buffered Send)
  - guaranteed to be asynchronous
  - routine returns before the message is delivered
  - system copies data into a (user-supplied) buffer and sends it later on

Synchronous send

Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.

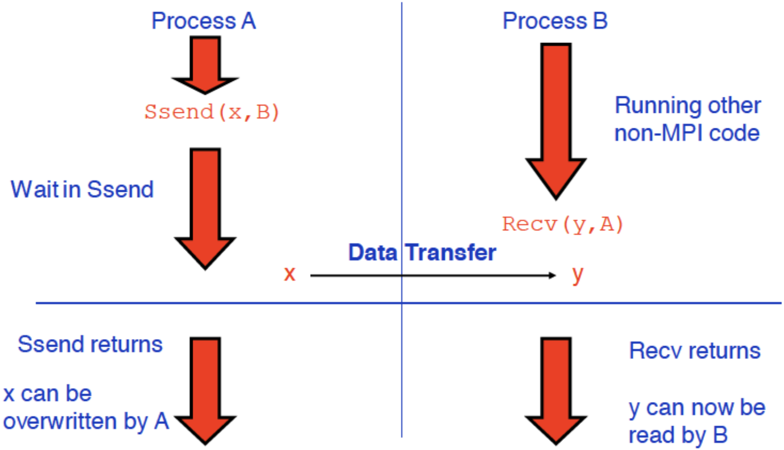
```
int MPI_Ssend(void *buf, int count,
              MPI_Datatype datatype,
              int dest, int tag,
              MPI_Comm comm)
```

Asynchronous send

This routine is a buffered mode send, routine returns before the message is delivered.

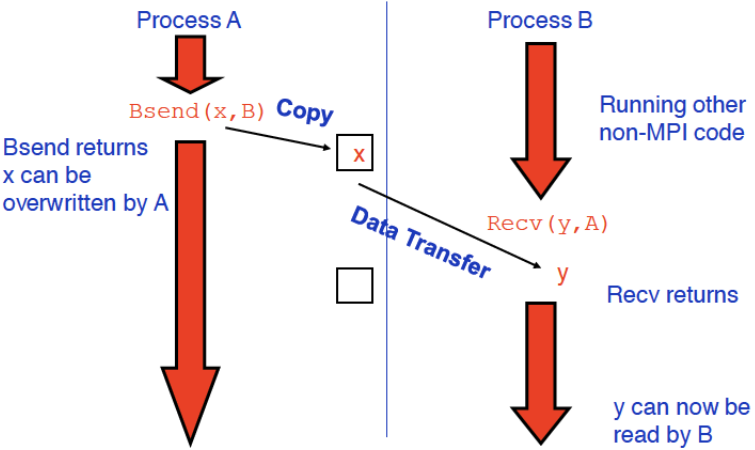
```
int MPI_Bsend(
              void *buf, int count,
              MPI_Datatype datatype,
              int dest, int tag,
              MPI_Comm comm )
```

MPI_Ssend



reference:http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/

MPI_Bsend



reference: `http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/`

MPI_Send

`MPI_Send` tries to solve these problems

- buffer space is provided by the system
- Send will normally be asynchronous (like Bsend)
- if buffer is full, Send becomes synchronous (like Ssend)

Message Matching(I)

```
Rank 0:
        Ssend(msg1, dest=1, tag=1)
        Ssend(msg2, dest=1, tag=2)
Rank 1:
        Recv(buf1, src=0, tag=1)
        Recv(buf2, src=0, tag=2)
```

- buf1 = msg1; buf2 = msg2
- Sends and receives correctly matched

Message Matching(II)

```
Rank 0:
        Ssend(msg1, dest=1, tag=1)
        Ssend(msg2, dest=1, tag=2)
Rank 1:
        Recv(buf2, src=0, tag=2)
        Recv(buf1, src=0, tag=1)
```

- Deadlock (due to synchronous send)
- Sends and receives incorrectly matched

Message Matching(III)

```
Rank 0:
        Bsend(msg1, dest=1, tag=1)
        Bsend(msg2, dest=1, tag=1)
Rank 1:
        Recv(buf1, src=0, tag=1)
        Recv(buf2, src=0, tag=1)
```

- buf1 = msg1; buf2 = msg2
- Messages have same tags but matched in order

Message Matching(IV)

```
Rank 0:
        Bsend(msg1, dest=1, tag=1)
        Bsend(msg2, dest=1, tag=2)
Rank 1:
        Recv(buf2, src=0, tag=2)
        Recv(buf1, src=0, tag=1)
```

- buf1 = msg1; buf2 = msg2
- Do not have to receive messages in order!

Message Matching(V)

```
Rank 0:
        Bsend(msg1, dest=1, tag=1)
        Bsend(msg2, dest=1, tag=2)
Rank 1:
        Recv(buf1, src=0, tag=MPI_ANY_TAG)
        Recv(buf2, src=0, tag=MPI_ANY_TAG)
```

- buf1 = msg1; buf2 = msg2
- Messages guaranteed to match in send order
  examine status to find out the actual tag values

Uses of Communicator

- Can split MPI_COMM_WORLD into pieces
  - each process has a new rank within each sub-communicator
  - guarantees messages from the different pieces do not interact

```
MPI_Comm_split(
        MPI_Comm comm,
        int color,
        int key,
        MPI_Comm* newcomm)
```

Example of using multiple communicators

```
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
       world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

Uses of Communicator

- Can make a copy of `MPI_COMM_WORLD`
  e.g. call the `MPI_Comm_dup` routine
  containing all the same processes but in a new communicator
- Enables processes to communicate with each other safely within a piece of code
  guaranteed that messages cannot be received by other code

Multithreading

MPI libraries vary in their level of thread support:

- `MPI_THREAD_SINGLE` - Level 0: Only one thread will execute.
- `MPI_THREAD_FUNNELED` - Level 1: The process may be multi-threaded, but only the main thread will make MPI calls - all MPI calls are funneled to the main thread.
- `MPI_THREAD_SERIALIZED` - Level 2: The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time. That is, calls are not made concurrently from two distinct threads as all MPI calls are serialized.
- `MPI_THREAD_MULTIPLE` - Level 3: Multiple threads may call MPI with no restrictions.

SINGLE

There are no threads in the system
E.g., there are no OpenMP parallel regions

```
int main(int argc, char ** argv)
{
 int buf[100];
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 for (i = 0; i < 100; i++)
  compute(buf[i]);
 /* Do MPI stuff */
 MPI_Finalize();
 return 0;
}
```

FUNNELED

All MPI calls are made by the master thread * Outside the OpenMP parallel regions * In OpenMP master regions

```
int main(int argc, char ** argv)
{
 int buf[100], provided;
 MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED,
        &provided);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 #pragma omp parallel for
  for (i = 0; i < 100; i++) {
   compute(buf[i]);
  }
 /* Do MPI stuff */
 MPI_Finalize();
 return 0;
}
```

SERIALIZED

Only one thread can make MPI calls at a time
Protected by OpenMP critical regions

```
int main(int argc, char ** argv)
{
 int buf[100], provided;
 MPI_Init_thread(&argc, &argv,
        MPI_THREAD_SERIALIZED, &provided);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 #pragma omp parallel for
  for (i = 0; i < 100; i++) {
   compute(buf[i]);
   #pragma omp critical
     /* Do MPI stuff */
 }
 MPI_Finalize();
 return 0;
}
```

MULTIPLE

Any thread can make MPI calls any time

```
int main(int argc, char ** argv)
{
 int buf[100], provided;
 MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE,
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 #pragma omp parallel for
  for (i = 0; i < 100; i++) {
   compute(buf[i]);
   /* Do MPI stuff */
 }
 MPI_Finalize();
 return 0;
}
```

Compile and run

- Use `mpicc -fopenmp [myprogram.c] -o myprogram` to compile
- Parallel jobs using MPI can be run using mpiexec or mpirun
- Mpiexec uses the task manager library of PBS to spawn copies of the executable on the nodes in a PBS allocation.

```
mpiexec [-n X] [-pernode]
      [-cpus-per-proc #perproc] [program]
```

where `X` is number of copies of `program`,
`[-pernode]` launches one process each node or `[-npernode #pernode]`
`[-cpu-per-proc #perproc]` controls the number of cores per process.

Overview

- Topologies
  1. Cartesian
- MPI Derived Types
  - Vectors
  - Structs
  - Others

The what and why of topologies

- Specifying which processes are “neighbours” of each other
- New communicators representing neighbours

- Convenient process naming
- Naming scheme to fit the communication pattern
- Simplifies writing of code.
- Can allow MPI to optimise communications.

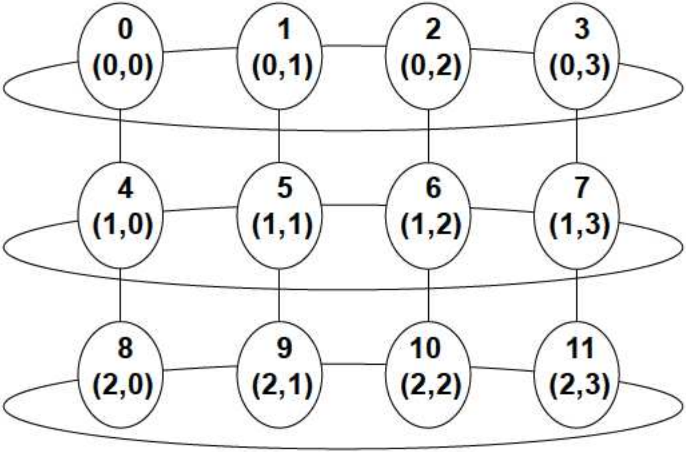
http://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-topo.html

How to use topologies

- Creating a topology produces a new communicator.
- MPI provides "mapping functions".
- Mapping functions compute processor ranks, based on the topology naming scheme.

Example

A 2-dimensional Cylinder



Topology types

- Cartesian topologies
  - each process is connected to its neighbours in a virtual grid.
  - boundaries can be cyclic, or not.
  - optionally re-order ranks to allow MPI implementation to optimise for underlying network interconnectivity.
  - processes are identified by cartesian coordinates.

Creating a Cartesian Virtual Topology

```
int MPI_Cart_create(MPI_Comm comm_old,
        int ndims, int *dims, int *periods,
        int reorder, MPI_Comm *comm_cart)
```

Balanced Processor Distribution

```
int MPI_Dims_create(int nnodes, int ndims,
        int *dims)
```

Cartesian Mapping Functions

- Call tries to set dimensions as close to each other as possible

dims before the call	function call	dims on return
(0, 0)	MPI_DIMS_CREATE(6, 2, dims)	(3, 2)
(0, 0)	MPI_DIMS_CREATE(7, 2, dims)	(7, 1)
(0, 3, 0)	MPI_DIMS_CREATE(6, 3, dims)	(2, 3, 1)
(0, 3, 0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

- Non zero values in dims sets the number of processors required in that direction.

Cartesian Mapping Functions

Mapping process grid coordinates to ranks

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

Mapping ranks to process grid coordinates

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
        int maxdims, int *coords)
```

Cartesian Mapping Functions

Computing ranks of my neighbouring processes

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

```
int MPI_Sendrecv_replace(void* buf, int count,
        MPI_Datatype datatype, int dest, int sendtag
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

```
e.g.,
MPI_SendRecv_replace(A, 1, MPI_REAL, dest, 0,
        source, 0, comm, status, ierr)
```

Non-existent ranks

Computing ranks of my neighbouring processes

- What if you ask for the rank of a non-existent process? or look off the edge of a non-periodic grid?
- MPI returns a NULL processor - rank is `MPI_PROC_NULL`

Derived Types

- MPI Derived Types
  - Vectors
  - Structs
  - Others

Motivation

- Send / Recv calls need a datatype argument
- What about types defined by a program?
  e.g., structures (in C)
- Send / Recv calls take a count parameter
  what about data that isnt contiguous in memory?
  e.g., subsections of 2D arrays

Approach

- Can define new types in MPI
  - User calls setup routines to describe new datatype to MPI
  - MPI returns a new datatype handle
  - Store this value in a variable, eg `MPI_MY_NEWTYPE`
- Derived types have same status as pre-defined
  - Can use in any message-passing call
- Some care needed for reduction operations
  - User must also define a new `MPI_Op` appropriate to the new datatype to tell MPI how to combine them

Defining types

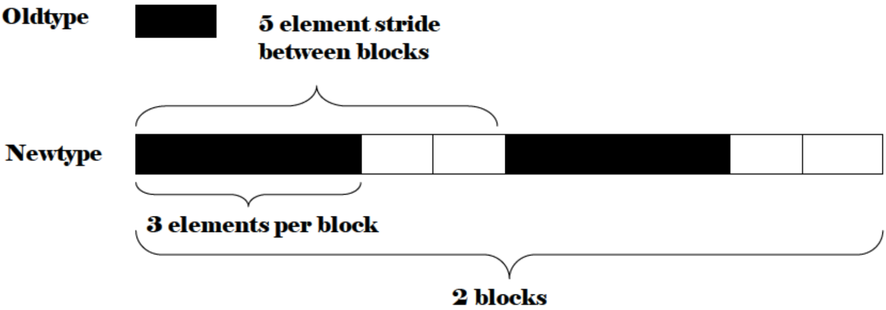
- All derived types stored by MPI as a list of basic types and displacements (in bytes)
  - for a structure, types may be different
  - for an array subsection, types will be the same
- User can define new derived types in terms of both basic types and other derived types

Contiguous Data

The simplest derived datatype consists of a number of contiguous items of the same datatype.

```
int MPI_Type_contiguous(int count,
        MPI_Datatype oldtype,
        MPI_Datatype *newtype)
```

Vector Datatype Example



- count = 2
- stride = 5
- blocklength = 3

What is a vector type?

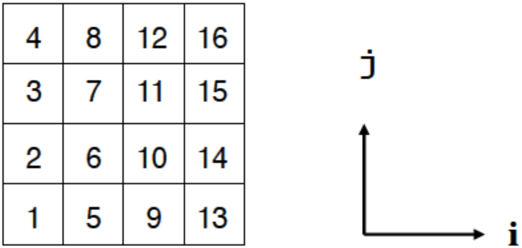
Why is a pattern with blocks and gaps useful?
A vector type corresponds to a subsection of a 2D array
Think about how arrays are stored in memory

Array Layout In Memory

C: x[16]

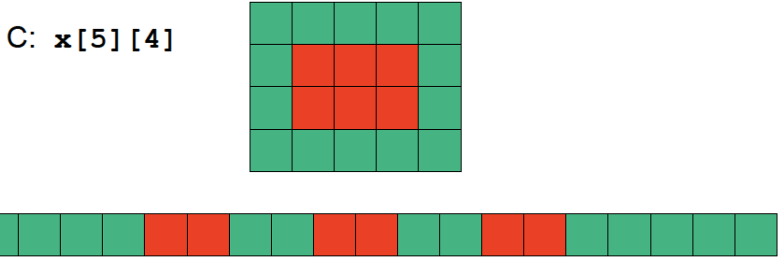
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

C: x[4][4]



Data is contiguous in memory

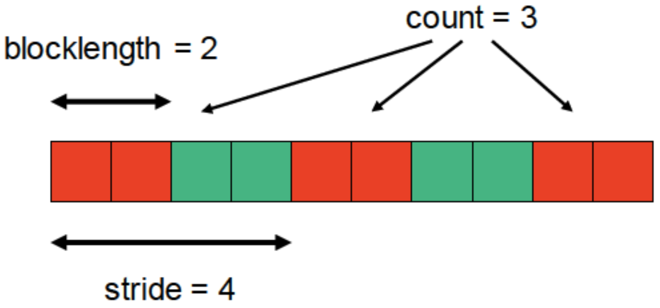
C example



A 3 X 2 subsection of a 5 x 4 array

- three blocks of two elements separated by gaps of two

Equivalent Vector Datatypes



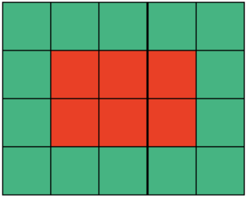
Constructing a Vector Datatype

```
int MPI_Type_vector (int count,
        int blocklength, int stride,
        MPI_Datatype oldtype,
        MPI_Datatype *newtype)
```

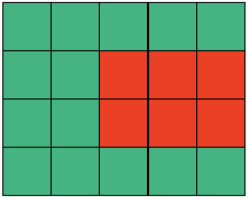
- Have defined a 3x2 subsection of a 5x4 array
  - but not defined WHICH subsection
  - is it the bottom left-hand corner? top-right?
- Data that is sent depends on what buffer you pass to the send routines
  - pass the address of the first element that should be sent

Constructing a Vector Datatype

```
MPI_Ssend(&x[1][1], 1, vector3x2, ...);
```



```
MPI_Ssend(&x[2][1], 1, vector3x2, ...);
```



Committing a Datatype

Once a datatype has been constructed, it needs to be committed before it is used in a message-passing call.
This is done using

```
MPI_TYPE_COMMIT
C:
int MPI_Type_commit (MPI_Datatype *datatype)
```