Advanced Message Passing Interface (MPI)

Bruno C. Mundim

SciNet HPC Consortium

May 15, 2023

About This Workshop

What do you need for this workshop?

- A computer with browser and internet connection to attend the lectures.
- A Zoom client to connect to the lecture and office hours.
- An ssh client to connect to the SciNet Teach cluster.
  - Linux and MaxOS: Use the ssh command in the terminal.
  - Windows: Use MobaXTerm https://mobaxterm.mobatek.net.

Make sure you can login to the website https://scinet.courses/1269 !

Workshop structure

- MONDAY: A first online lecture over Zoom (you're here!).

  An assignment will be given during the course of the lecture.

  You can ask questions:

  - in the Zoom chat during and at the end of the lecture.
  - in the student forum on the course site.
  - and also during:

- WEDNESDAY: Zoom office hours.

  Submit a solution for the assignment on the course website (deadline is midnight Thursday)

  Assignment submission and lecture participation required to obtain credits towards the SciNet HPC Certificate.

- FRIDAY: A last online lecture on Zoom that will address the solution, common mistakes, and wrap-up.

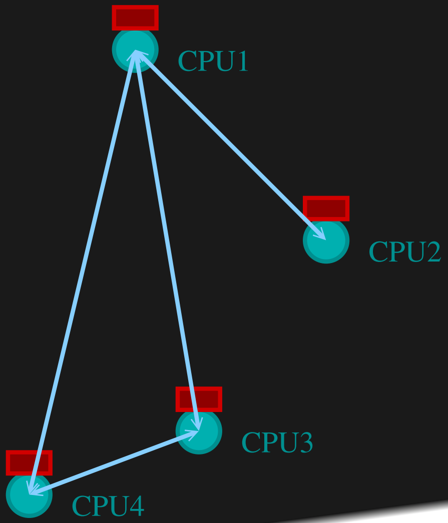
Today’s Lecture Outline

- MPI Basics Review
- Scientific MPI Example: 2D Diffusion Equation
- Teach Cluster Access and Assignment
- Derived Data Types
- Application Topology

MPI Basics Review

Distributed Memory: Clusters

- ***Machine Architecture:*** Clusters, or, distributed memory machines.
- ***Parallel code:*** run on separate computers and communicate with each other.
- ***Usual communication model:*** "message passing".
- ***Message Passing Interface (MPI):*** Open standard library interface for message passing, ratified by the MPI Forum.
- ***MPI Implementations:***
  - OpenMPI www.open-mpi.org
    - SciNet clusters (Niagara or Teach): `module load gcc openmpi`
  - MPICH2 www.mpich.org
    - Niagara: `module load intel intelmpi`



MPI is a Library for Message Passing

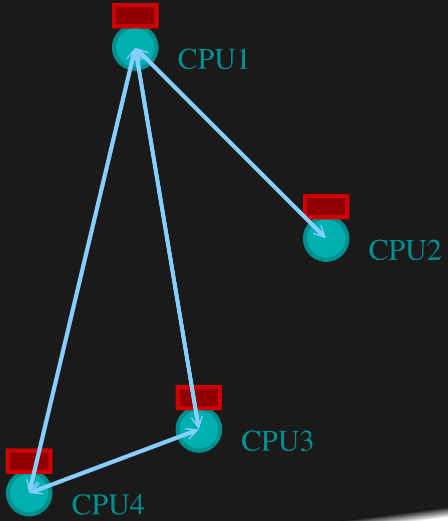
- Not built into the compiler.
- Function calls that can be made from any compiler, many languages.
- Just link to it.
- Wrappers: mpicc, mpif90, mpicxx

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size, err;
  err = MPI_Init(&argc, &argv);
  err = MPI_Comm_size(MPI_COMM_WORLD, &size);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from task %d of %d!\n",rank,
                                             size);
  err = MPI_Finalize();
}
```

```
program helloworld
use mpi
implicit none
integer :: rank, commsize, err
call MPI_Init(err)
call MPI_Comm_size(MPI_COMM_WORLD, commsize, err)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, err)
print *,'Hello world from task',rank,'of',commsize
call MPI_Finalize(err)
end program helloworld
```

MPI is a Library for Message Passing

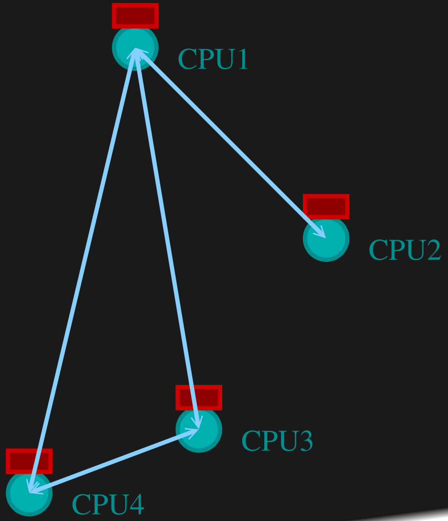
- Communication/coordination between tasks done by sending and receiving messages.
- Each message involves a function call from each of the programs.



MPI is a Library for Message Passing

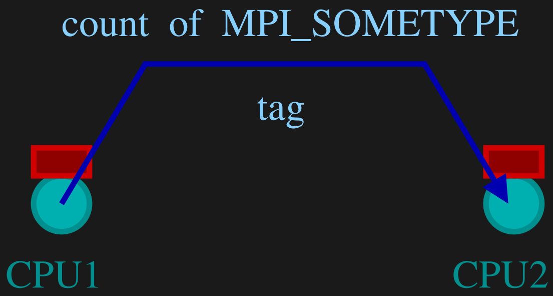
Three basic sets of functionality:

- Pairwise communications via messages
- Collective operations via messages
- Efficient routines for getting data from memory into messages and vice versa



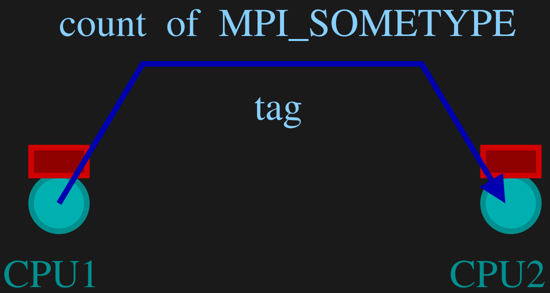
Messages

- Messages have a **sender** and a **receiver**.
- When you are sending a message, don't need to specify sender (it's the current processor).
- A sent message has to be actively received by the receiving process.



Messages

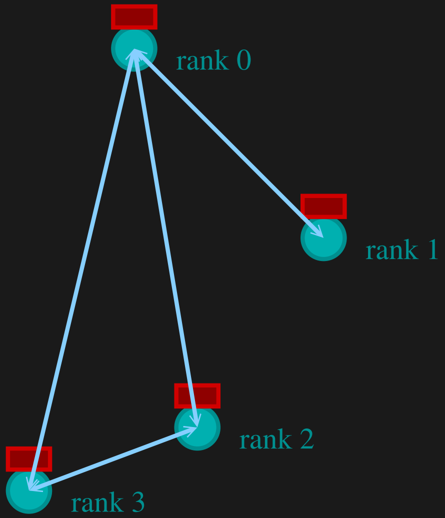
- MPI messages are a string of length **count** all of some fixed MPI **type**.
- MPI types exist for characters, integers, floating point numbers, etc.
- An arbitrary non-negative integer **tag** is also included – it helps keep things straight if lots of messages are sent.



Communicators

- MPI groups processes into communicators.
- Each communicator has some size – number of tasks.
- Every task has a rank 0..size-1
- Every task in your program belongs to `MPI_COMM_WORLD`.

`MPI_COMM_WORLD`:
size = 4, ranks = 0..3



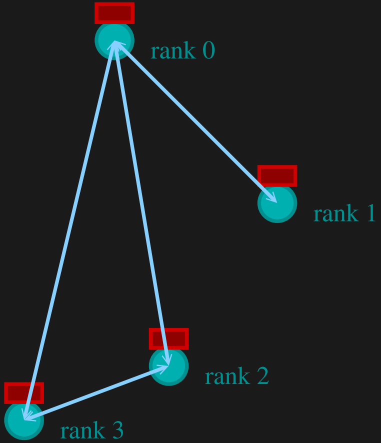
Communicators

- One can create one's own communicators over the same tasks.
- May break the tasks up into subgroups.
- May just re-order them for some reason

Communicators

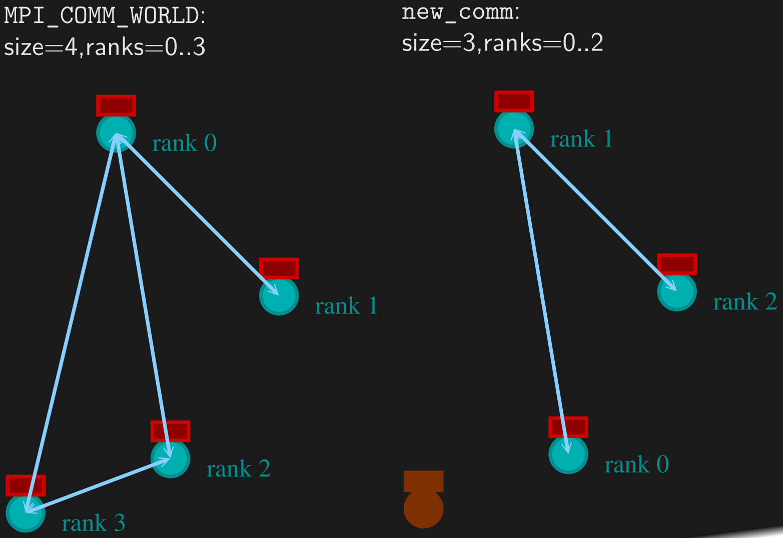
MPI_COMM_WORLD:
size=4,ranks=0..3

- One can create one's own communicators over the same tasks.
- May break the tasks up into subgroups.
- May just re-order them for some reason



Communicators

- One can create one's own communicators over the same tasks.
- May break the tasks up into subgroups.
- May just re-order them for some reason



MPI Communicator Basics

Communicator Components

- `MPI_COMM_WORLD`:
  Global Communicator
- `MPI_Comm_rank(MPI_COMM_WORLD,&rank)`
  Get current task's rank
- `MPI_Comm_size(MPI_COMM_WORLD,&size)`
  Get communicator size

Different versions of SEND

MPI_Ssend: Standard synchronous send

- guaranteed to be synchronous.
- routine will not return until the receiver has “picked up”.

MPI_Bsend: Buffered Send

- guaranteed to be asynchronous.
- routine returns before the message is delivered.
- system copies data into a buffer and sends it in due course.
- can fail if buffer is full.

MPI_Send (standard Send)

- may be implemented as synchronous or asynchronous send.
- causes a lot of confusion.

In this class, stick with MPI_Ssend for clarity and robustness

Send and Receive

C

```c
MPI_Status status;
err = MPI_Ssend(sendptr, count, MPI_TYPE, destination, tag, Communicator);
err = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag, Communicator, status);
```

Fortran

```fortran
integer status(MPI_STATUS_SIZE)
call MPI_SSEND(sendarr, count, MPI_TYPE, destination, tag, Communicator, err)
call MPI_RECV(rcvarr, count, MPI_TYPE, source, tag, Communicator, status, err)
```

MPI: Sendrecv

```
err = MPI_Sendrecv(sendptr, count, MPI_TYPE, destination, tag,
                   recvptr, count, MPI_TYPE, source, tag, Communicator, MPI_Status)
```

- A blocking send and receive built together
- Let them happen simultaneously
- Can automatically pair send/recvs
- Why 2 sets of tags/types/counts?

MPI Non-Blocking Functions: `MPI_Isend`, `MPI_Irecv`

- Returns immediately, posting request to system to initiate communication.
- However, communication is not completed yet.
- Cannot tamper with the memory provided in these calls until the communication is completed.

Nonblocking Sends

- Allows you to get work done while message is *in flight*.
- Must not alter send buffer until send has completed.
- C:

```
MPI_Isend(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request)
```

- FORTRAN:

```
MPI_ISEND(BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG, INTEGER COMM, INTEGER REQUEST,
                                                                               INTEGER ERROR)
```

MPI: Non-Blocking Isend & Irecv

```
err = MPI_Isend(sendptr, count, MPI_TYPE, destination, tag, Communicator, MPI_Request)
err = MPI_Irecv(rcvptr, count, MPI_TYPE, source, tag, Communicator, MPI_Request)
```

- sendptr/rcvptr: pointer to message
- count: number of elements in ptr
- MPI_TYPE: one of MPI_DOUBLE, MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- destination/source: rank of sender/receiver
- tag: unique id for message pair
- Communicator: MPI_COMM WORLD or user created
- MPI_Request: Identify comm operations

MPI Collectives

- ***Reduction:***
  - Works for a variety of operations (+,*,min,max)
  - For example, to calculate the min/mean/max of numbers accross the cluster.

```
err = MPI_Allreduce(sendptr, rcvptr, count, MPI_TYPE, MPI_Op, Communicator);
err = MPI_Reduce(sendbuf, recvbuf, count, MPI_TYPE, MPI_Op, root, Communicator);
```

- `sendptr/rcvptr`: pointers to buffers
- `count`: number of elements in ptrs
- `MPI_TYPE`: one of `MPI_DOUBLE`, `MPI_FLOAT`, `MPI_INT`, `MPI_CHAR`, etc.
- `MPI_Op`: one of `MPI_SUM`, `MPI_PROD`, `MPI_MIN`, `MPI_MAX`.
- `Communicator`: MPI_COMM_WORLD or user created.
- `All` variants send result back to all processes; non-`All` sends to process `root`.

Collective Operations

Collective

- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Collective Operations

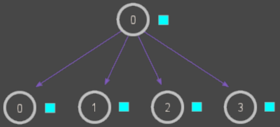
Collective

- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

Broadcast

MPI_Bcast



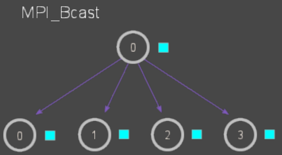
Collective Operations

Collective

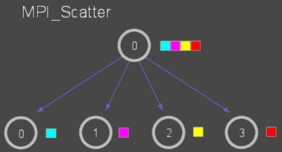
- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

Broadcast



Scatter



Collective Operations

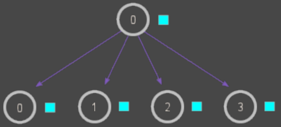
Collective

- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

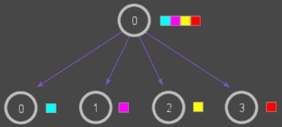
Broadcast

MPI_Bcast



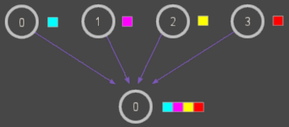
Scatter

MPI_Scatter



Gather

MPI_Gather



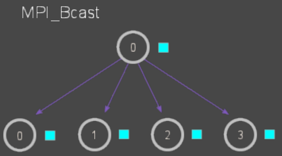
Collective Operations

Collective

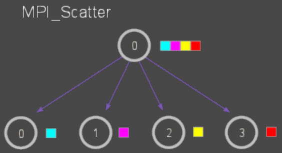
- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

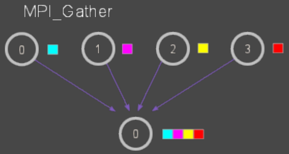
Broadcast



Scatter



Gather



- File I/O
- Barriers (don't!)
- All-to-all . . .

Scientific MPI Example

Scientific MPI Example

Consider a diffusion equation with an explicit **finite-difference**, **time-marching** method.

Imagine the problem is too large to fit in the memory of one node, so we need to do **domain decomposition**, and use **MPI**.

Discretizing Derivatives

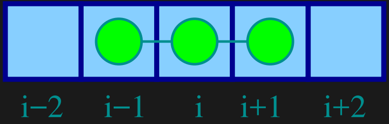
- Partial Differential Equations like the diffusion equation

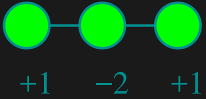
$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

  are usually numerically solved by finite differencing the discretized values.

- Implicitly or explicitly involves interpolating data and taking the derivative of the interpolant.
- Larger ‘stencils’ → More accuracy.

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$





Diffusion equation in higher dimensions

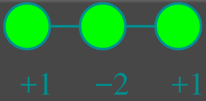
Spatial grid separation: Δx. Time step Δt.

Grid indices: i, j. Time step index: (n)

1D

$$\left.\frac{\partial T}{\partial t}\right|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$



Diffusion equation in higher dimensions

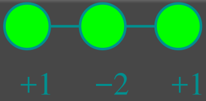
Spatial grid separation: Δx. Time step Δt.

Grid indices: i, j. Time step index: (n)

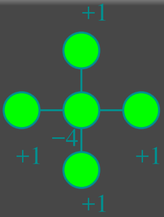
1D

$$\left.\frac{\partial T}{\partial t}\right|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$



2D



Diffusion equation in higher dimensions

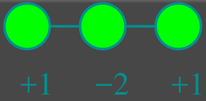
Spatial grid separation: Δx. Time step Δt.

Grid indices: i, j. Time step index: (n)

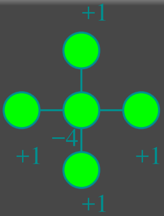
1D

$$\left.\frac{\partial T}{\partial t}\right|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$



2D



$$\left.\frac{\partial T}{\partial t}\right|_{i,j} \approx \frac{T_{i,j}^{(n)} - T_{i,j}^{(n-1)}}{\Delta t}$$

$$\left.\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)\right|_{i,j} \approx \frac{T_{i-1,j}^{(n)} + T_{i,j-1}^{(n)} - 4T_{i,j}^{(n)} + T_{i+1,j}^{(n)} + T_{i,j+1}^{(n)}}{\Delta x^2}$$

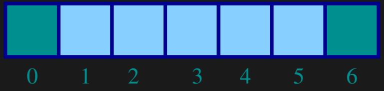
Stencils and Boundaries

- How do you deal with boundaries?
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution:

Guard cells:

- Pad domain with these guard cells so that stencil works even for the first point in domain.
- Fill guard cells with values such that the required boundary conditions are met.

1D



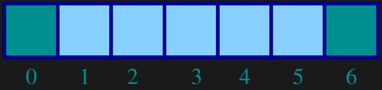
- Number of guard cells $n_g = 1$
- Loop from $i = n_g \ldots N - 2n_g$.

Stencils and Boundaries

- How do you deal with boundaries?
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution:

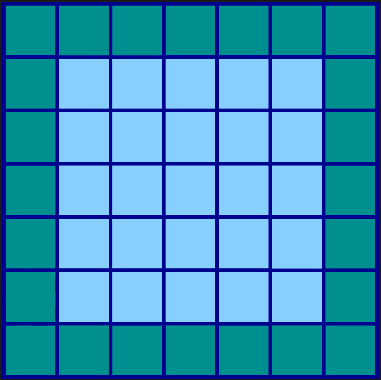
  Guard cells:
  - Pad domain with these guard cells so that stencil works even for the first point in domain.
  - Fill guard cells with values such that the required boundary conditions are met.

1D



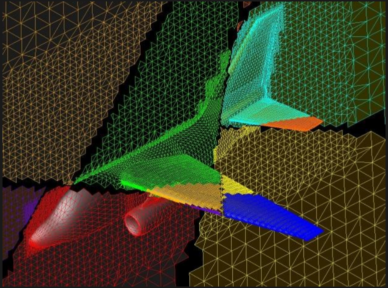
- Number of guard cells $n_g = 1$
- Loop from $i = n_g \ldots N - 2n_g$.

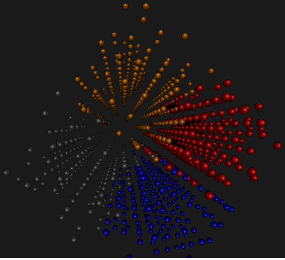
2D

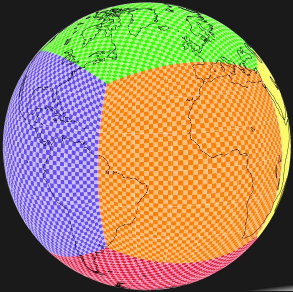


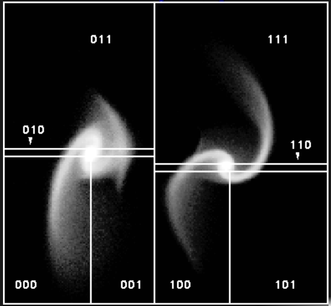
Domain decomposition

- A very common approach to parallelizing on distributed memory computers.
- Subdivide the domain into contiguous subdomains.
- Give each subdomain to a different MPI process.
- No process contains the full data!
- Maintains locality.
- Need mostly local data, ie., only data at the boundary of each subdomain will need to be sent between processes.



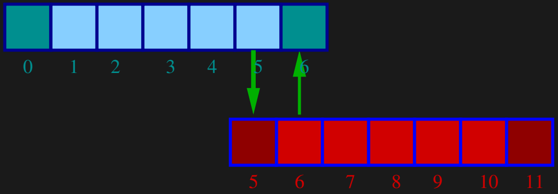






Guard cell exchange

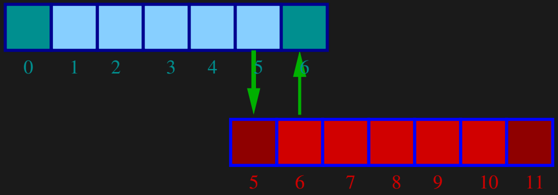
- In the domain decomposition, the stencils will jut out into a neighbouring subdomain.
- Much like the boundary condition.
- One uses guard cells for domain decomposition too.
- If we managed to fill the guard cell with values from neighbouring domains, we can treat each coupled subdomain as an isolated domain with changing boundary conditions.



- Could use even/odd trick, or sendrecv.

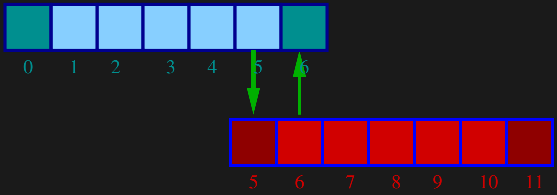
Diffusion: Had to wait for communications to compute

- Could not compute end points without guardcell data
- All work halted while all communications occurred
- Significant parallel overhead.

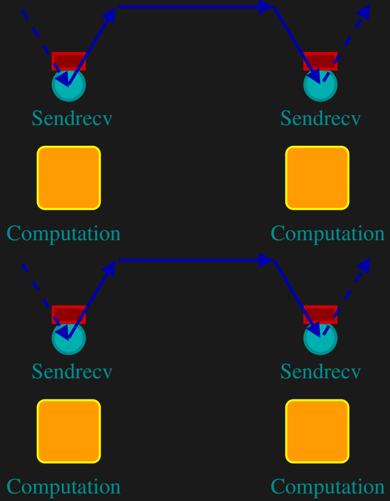


Diffusion: *Had* to wait?

- But inner zones could have been computed just fine.
- Ideally, would do inner zones work while communications is being done; then go back and do end points.



Blocking Communication/Computation Pattern

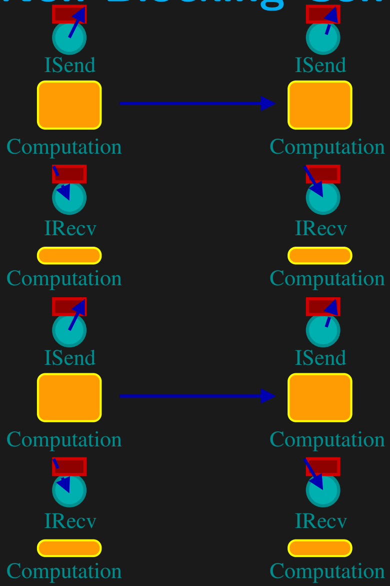


We have the following sequence of communication and computation:

- The code exchanges guard cells using `Sendrecv`
- The code **then** computes the next step.
- The code exchanges guard cells using `Sendrecv` again.
- etc.

We can do better.

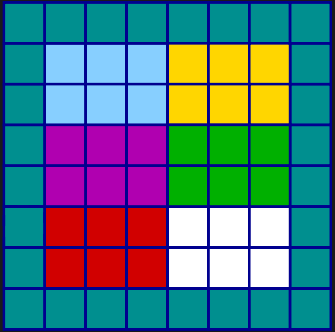
Non-Blocking Communication/Computation Pattern

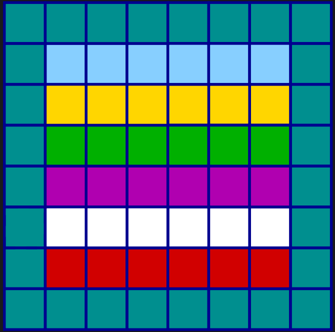


- The code start a send of its guard cells using `ISend`.
- Without waiting for that send's completion, the code computes the next step for the inner cells (while the guard cell message is *in flight*)
- The code then receives the guard cells using `IRecv`.
- Afterwards, it computes the outer cell's new values.
- Repeat.

2D diffusion with MPI

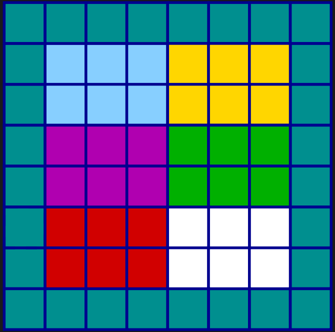
How to divide the work in a 2D grid?

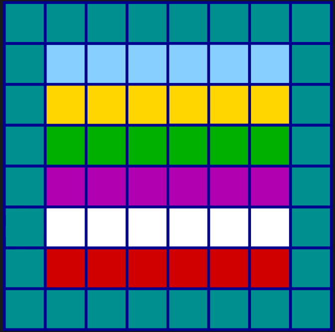




2D diffusion with MPI

How to divide the work in a 2D grid?

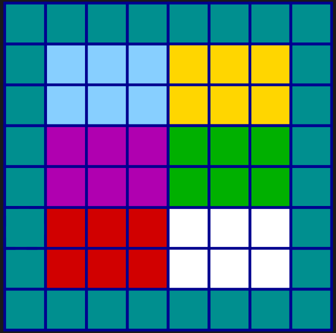


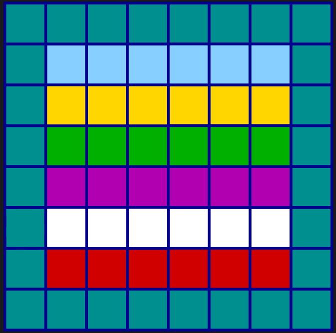


- Less communication (18 edges).
- Harder to program, non-contiguous data to send, left, right, up and down.

2D diffusion with MPI

How to divide the work in a 2D grid?





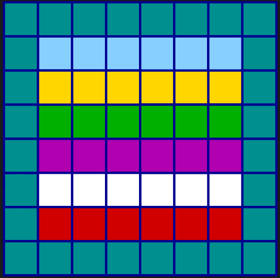
- Less communication (18 edges).
- Harder to program, non-contiguous data to send, left, right, up and down.

- Easier to code, similar to 1d, but with contiguous guard cells to send up and down.
- More communication (30 edges).

Let’s look at the easiest domain decomposition.

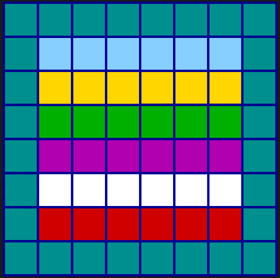
Let's look at the easiest domain decomposition.

Serial:

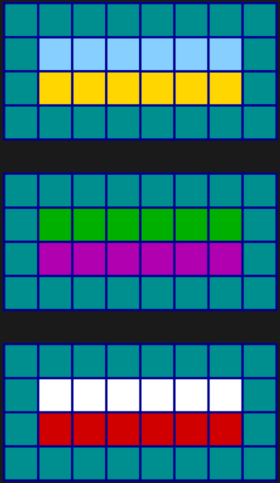


Let’s look at the easiest domain decomposition.

Serial:

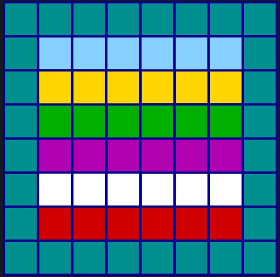


Parallel $(P = 3)$:

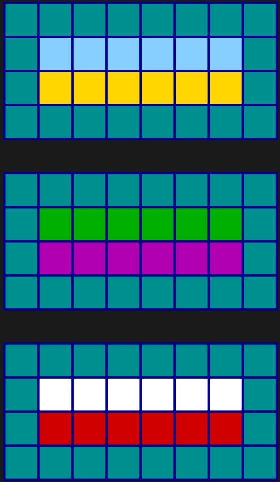


Let's look at the easiest domain decomposition.

Serial:



Parallel $(P = 3)$*:*



Communication pattern:

- Copy upper stripe to upper neighbour bottom guard cell.
- Copy lower stripe to lower neighbout top guard cell.
- Contiguous cells: can use `count` in `MPI_Sendrecv`.
- Similar to 1d diffusion.

Teach Cluster Access and Assignment

Access to SciNet’s Teach supercomputer

Access to SciNet’s Teach supercomputer

- SciNet’s Teach supercomputer is part of the old GPC system (42 nodes) that has been repurposed for education and training in general, and in particular for many of summer school sessions.
- Log into Teach login node, **teach01**, with your *Compute Canada* account credentials or your *lcl_uothpc383sNNNN* temporary account.

```
$ ssh -Y USER@teach.scinet.utoronto.ca
$ cd $SCRATCH
$ cp -r /scinet/course/mpi/advanced-mpi .
$ cd advanced-mpi
$ source setup
```

Access to SciNet's Teach supercomputer

Access to SciNet's Teach supercomputer

- SciNet's Teach supercomputer is part of the old GPC system (42 nodes) that has been repurposed for education and training in general, and in particular for many of summer school sessions.

- Log into Teach login node, **teach01**, with your *Compute Canada* account credentials or your *lcl_uothpc383sNNNN* temporary account.

```
$ ssh -Y USER@teach.scinet.utoronto.ca
$ cd $SCRATCH
$ cp -r /scinet/course/mpi/advanced-mpi .
$ cd advanced-mpi
$ source setup
```

Running computations

- On most supercomputer, a scheduler governs the allocation of resources.

- This means submitting a job with a jobscript.

- `srun`: a command that is a resource request + job running command all in one, and will run the command on one (or more) of the available resources.

- We have set aside 34 nodes with 16 cores for this class, so occasionally, only in very busy sessions, you may have to wait for someone else's `srun` command to finish.

Assignment: 2D Diffusion

- 2D diffusion equation serial code:

```
$ cd $SCRATCH/advanced-mpi/diffusion2d
$ # source ../setup
$ make diffusion2dc
$ ./diffusion2dc
```

Assignment: 2D Diffusion

- 2D diffusion equation serial code:

```
$ cd $SCRATCH/advanced-mpi/diffusion2d
$ # source ../setup
$ make diffusion2dc
$ ./diffusion2dc
```

- 2D diffusion equation parallel code:

```
$ make diffusion2dc-mpi-nonblocking
$ # or srun
$ mpirun -np 4 ./diffusion2dc-mpi-nonblocking
```

- Part I: Use MPI derived datatypes instead of packing and unpacking the data manually.
- `cp diffusion2dc-mpi-nonblocking.c diffusion2dc-mpi-nonblocking-datatype.c`
- Build with `make diffusion2dc-mpi-nonblocking-datatype`
- Test on 4..9 processors

Assignment: 2D Diffusion

- 2D diffusion equation serial code:

```
$ cd $SCRATCH/advanced-mpi/diffusion2d
$ # source ../setup
$ make diffusion2dc
$ ./diffusion2dc
```

- 2D diffusion equation parallel code:

```
$ make diffusion2dc-mpi-nonblocking
$ # or srun
$ mpirun -np 4 ./diffusion2dc-mpi-nonblocking
```

- Part I: Use MPI derived datatypes instead of packing and unpacking the data manually.
- `cp diffusion2dc-mpi-nonblocking.c diffusion2dc-mpi-nonblocking-datatype.c`
- Build with `make diffusion2dc-mpi-nonblocking-datatype`
- Test on 4..9 processors

- Part II: Use MPI Cartesian topology routines to map the 2D cartesian grid of the diffusion equation domain into a 2D layout of processes. Get rid of the manually done mapping.
- `cp diffusion2dc-mpi-nonblocking-datatype.c diffusion2dc-mpi-nonblocking-carttopo.c`
- Build with `make diffusion2dc-mpi-nonblocking-carttopo`

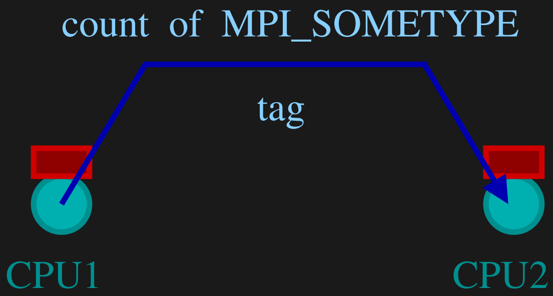
Tips

- Switch off graphics (in Makefile, change `USEPGPLOT=-DPGPLOT` to `USEPGPLOT=`);
- Get familiar with the serial code in 2D and review the 1D one, if needed.
- If you get stuck debugging, try to decrease the problem size and the number of steps.

Derived Datatypes

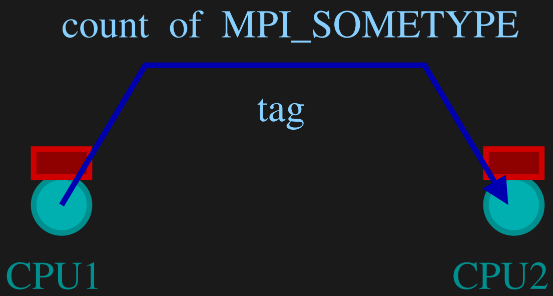
Motivation

- Every message is associated with a datatype.
- All MPI data movement functions move data in some ***count*** units of some datatype.
- Portability: specifying the length of a message as a given count of occurrences of a given datatype is more portable than using length in bytes, since lengths of given types may vary from one machine to another.
- So far our messages correspond to contiguous regions of memory: a count of the the basic MPI datatypes such as MPI_INT or MPI_DOUBLE was sufficient to describe our messages.



Motivation

- Derived datatypes allow us to specify noncontiguous areas of memory, such as a column of an array stored rowwise.
- A new datatype might describe, for example, a group of elements that are separated by a constant amount in memory, a stride.
- Derived datatypes allow arbitrary data layouts to be serialized into message streams



Basic Datatypes for Fortran

- MPI provides a rich set of predefined datatypes.
- All basic datatypes in C and Fortran.
- Two datatypes specific to MPI:
  - `MPI_BYTE`: Refers to a byte defined as eight binary digits.
  - `MPI_PACKED`: Rather than create a new datatype, just assemble a contiguous buffer to be sent.
- Why not use char as bytes?
  - Usually represented by implementations but not required. For example C for Japanese might choose 16-bit chars.
  - Machines might have different character sets in heterogeneous environment.

MPI Datatype	**Fortran Datatype**
MPI_BYTE	
MPI_CHARACTER	CHARACTER
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_INTEGER	INTEGER
MPI_LOGICAL	LOGICAL
MPI_PACKED	
MPI_REAL	REAL

Basic Datatypes for C

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_LONG_LONG_INT	long long
MPI_SIGNED_CHAR	signed char

MPI Datatype	C Datatype
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_LONG_LONG	unsigned long long
MPI_C_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_PACKED	
MPI_BYTE	

Datatype Concepts

- Basic definitions:
  - Datatype is an object consisting of a sequence of the basic datatypes and displacements, in bytes, of each of these datatypes.
  - Displacements in bytes are relative to the buffer the datatype describes.
- How does MPI describe a general datatype?
  - MPI represents a datatype as a sequence of pairs of basic types and displacements, a typemap.
- Typemap:

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}$$

  - For example, type MPI_INT represented by $(int, 0)$.
  - Displacements tell MPI where to find the bits.
- Type signature: list of the basic datatypes in a datatype.

$$Typesignature = \{type_0, ..., type_{n-1}\}$$

  - It controls how data items are interpreted when sent or received.

Datatype Concepts (Cont.)

- Component Displacement Lower bound (lb) is the location of the first byte described by the datatype:

$$lb(Typemap) = \min_j (disp_j)$$

- Component Displacement Upper bound (ub) is the location of the last byte described by the datatype:

$$ub(Typemap) = \max_j (disp_j + sizeof(type_j)) + pad$$

  - Where $sizeof$ operator returns the size of the basic datatype in bytes.
- Extent is the difference between these two bounds:

$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

  - ub is possibly increased by pad to satisfy alignment requirements.

Data Alignment

- Both C and Fortran require that the basic datatypes be properly aligned:
  - The locations of an integer or a double-precision value occur only where allowed.
  - Each implementation of these languages defines what is allowed.
  - Most common: the address of an item in bytes is a multiple of the length of that item in bytes.
  - For example, if an int takes four bytes, then the address of an int must be a multiple of the length of that item in bytes: evenly divisible by four.
- Data aligment requirement reflects in the definition of extent of a MPI datatype.

- Example of a typemap on a computer that requires int's to be aligned on 4-byte boundaries:

$$\{(int, 0), (char, 4)\}$$

$$lb = min(0, 4) = 0$$

$$ub = max(0 + 4, 4 + 1) = 5$$

  - Next int can only be placed with displacement eight from the int in the typemap. Pad in this case is three.
  - Therefore, this typemap's extend on this computer is eight.

Datatype Information

MPI routines to retrieve information about MPI datatypes:

MPI_Type_get_extent

```
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)
```

- Get the lower bound and extent for a datatype:
  - `datatype`: handle on datatype to get information on.
  - `lb`: lower bound returned and stored as MPI_Aint, an integer type that can hold an arbitrary address.
  - `extent`: the returned extent of the datatype. Previous example extent was 8 bytes.

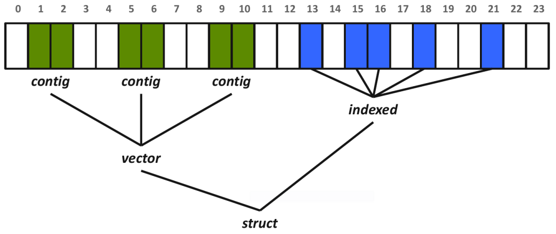
MPI_Type_size

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

- Get the number of bytes or the size of a datatype:
  - `datatype`: handle on datatype to get information on.
  - `size`: datatype size in bytes. Previous example size was 5 bytes.

Datatype Constructors

- **Problem**: Typemap is a general way of describing an arbitrary datatype, but not convenient for a large number of entries.
- **Solution**: MPI provides different ways to create datatypes without explicitly constructing the typemap:
  - Contiguous: It produces a new datatype by making count copies of an old one. Displacements incremented by the extent of the oldtype.
  - Vector: Like contiguous but allows for regular gaps in displacements. Elements separated by multiples of the extent of the input datatype.
  - Hvector: Like vector, but elements are separated by a number of bytes.
- More sophisticated constructors:
  - Indexed: Array of displacements provided. Displacements measured in terms of the extent of the input datatype.
  - Hindexed: Like indexed, but displacements measured in bytes.
  - Struct: Fully general. Input is the typemap, if input are basic MPI datatypes.



Datatype Constructors (Cont.)

MPI_Type_contiguous

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

- Simplest datatype constructor, which allows replication of a oldtype datatype into contiguous locations:
  - count: replication count (nonnegative integer).
  - oldtype: old datatype handle.
  - newtype: new datatype handle.
- Example: if original datatype (oldtype) has typemap:

$$\{(int, 0), (double, 8)\}$$

then:

```
MPI_Type_contiguous(2, oldtype, &newtype);
```

produces a datatype newtype with typemap:

$$\{(int, 0), (double, 8), (int, 16), (double, 24)\}$$

Datatype Constructors (Cont.)

MPI_Type_vector

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

- Allows replication of a oldtype datatype into locations of equally spaced blocks. Each block is obtained by concatenating blocklength copies of the old datatype:
  - `count`: number of blocks (nonnegative integer).
  - `blocklength`: number of elements in each block (nonnegative integer).
  - `stride`: number of elements between start of each block.
- Very useful for Cartesian arrays.
- Example: if original datatype (oldtype) has typemap: $(double, 0)$ with extent 8, then:

```
MPI_Type_vector(3, 2, 4, oldtype, &newtype);
```

produces a datatype newtype with extension 3 x 4 x 8 = 96 bytes and typemap:

$$\{(double, 0), (double, 8), (double, 32), (double, 40), (double, 64), (double, 72)\}$$

Datatype Constructors (Cont.)

MPI_Type_create_hvector

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

- Creates a vector (strided) data type with offset in bytes.
- Useful for composition, for example, vector of structs.

MPI_Type_create_indexed_block

```
int MPI_Type_create_indexed_block(int count, int blocklength, const int array_of_displacements[],
                                  MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Creates an indexed data type with the same block length for all blocks.
- Useful for retrieving irregular subsets of data from a single array.
  - blocklength $= 2$
  - array_of_displacements $= \{0, 5, 8, 13, 18\}$

Datatype Constructors (Cont.)

MPI_Type_indexed

```
int MPI_Type_indexed(int count, const int array_of_blocklengths[],
            const int array_of_displacements[], MPI_Datatype oldtype,
            MPI_Datatype *newtype)
```

- Creates an indexed datatype, where each block can contain a different number of oldtype copies.
  - array_of_blocklengths $= \{1, 1, 2, 1, 2, 1\}$
  - array_of_displacements $= \{0, 3, 5, 9, 13, 17\}$

MPI_Type_create_struct

```
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
            const MPI_Aint array_of_displacements[], const MPI_Datatype array_of_types[],
            MPI_Datatype *newtype)
```

- Creates a structured data type.
- Useful for retrieving virtually any data layout in memory.
  - array_of_blocklengths $= \{1, 1, 2, 1, 2, 1\}$
  - array_of_displacements $= \{0, 3, 5, 9, 13, 17\}$
  - array_of_types = { MPI_INT, MPI_DOUBLE, MPI_INT, MPI_INT, MPI_INT }

Datatype Constructors (Cont.)

MPI_Type_create_subarray

```
int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],
                             const int array_of_subsizes[], const int array_of_starts[],
                             int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Creates a data type describing an n-dimensional subarray of an n-dimensional array.

Create, Commit, Duplicate, and Free

- Once a type is created with one of the constructors above, it must be committed to the system before use.
  - Use MPI_Type_commit.
  - This allows the system to hopefully perform heavy optimizations.
- MPI_Type_dup
  - Duplicates a type.
- MPI_Type_free
  - Free MPI resources for datatypes.

Application Topology

Introduction to Application Topology

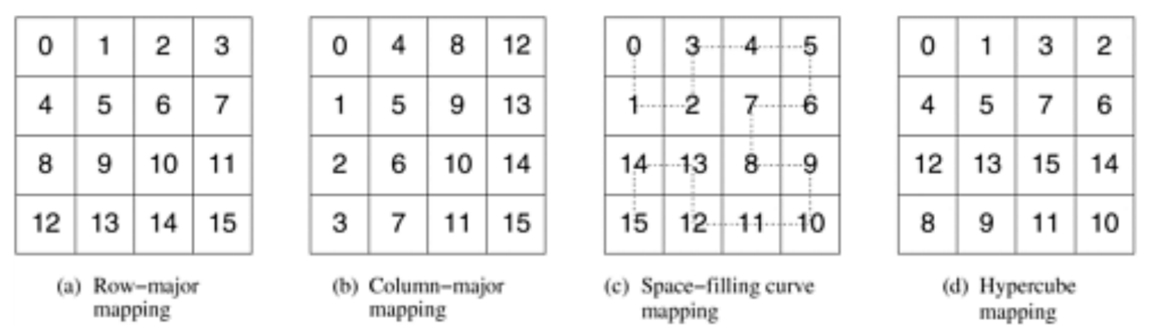
- MPI offers a facility, called ***process topology***, to attach information about the communication relationships between processes to a communicator.
- Programmer specifies the topology once during setup and then reuse it in different parts of the code.
- User-specified topology matches application communication patterns.

Definitions

- Topology of the computer, or interconnection network, is the description of how the processes in a parallel computer are connected to one another.
- Virtual or application topology is the pattern of communication amongst the processes.

Process Mapping

- Good choice of mapping depends on the details of the underlying hardware.
- Only the vendors knows the best way to fit the application topologies into the machine topology. They optimize through the implementation of MPI topology functions.
- MPI does not provide the programmer any control over these mappings.



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(a) Row–major mapping

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

(b) Column–major mapping

0	3	4	5
1	2	7	6
14	13	8	9
15	12	11	10

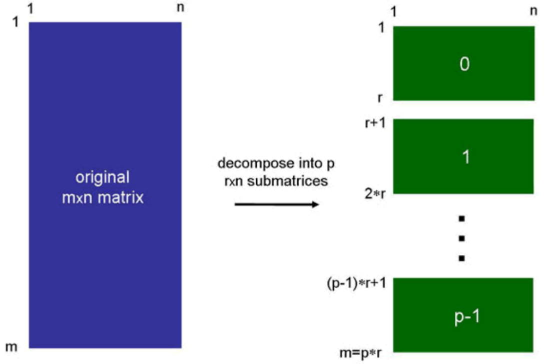
(c) Space–filling curve mapping

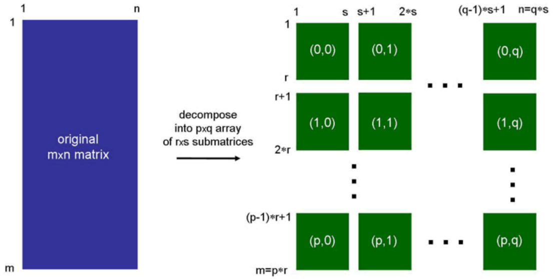
0	1	3	2
4	5	7	6
12	13	15	14
8	9	11	10

(d) Hypercube mapping

Different ways to map a set of processes to a 2-dimensional grid.

Example: Matrix Partitioning





2-dimensional rank numbering system provides a clearer representation of submatrices relationships.

Graph and Cartesian Topologies

- MPI has the task of deciding how to assign processes to each part of the decomposed domain.
- MPI provides the service of handling assignment of processes to regions. It provides two types of topology routines to address the needs of different data topological layouts:

Cartesian Topology

- It is a decomposition of the application processes in the natural coordinate directions, for example, along x and y directions.

Graph Topology

- It is the type of virtual topology that allows general relationships between processes, where processes are represented by nodes of a graph.

MPI Cartesian Topology Functions

MPI provides routines for dealing with cartesian topologies:

- MPI_Cart_create: Create a Cartesian topology.
- MPI_Cart_coords: Determine process coordinates in Cartesian topology given rank in group.
- MPI_Cart_rank: Determines process rank in communicator given Cartesian location.
- MPI_Cart_sub: Partitions a communicator into subgroups, which form lower-dimensional Cartesian subgrids.
- MPI_Cart_get: Retrieves Cartesian topology information associated with a communicator.
- MPI_Cartdim_get: Retrieves Cartesian topology information associated with a communicator: number of dimensions.
- MPI_Cart_shift: Returns the shifted source and destination ranks, given a shift direction and amount.

MPI Cartesian Topology Functions (Cont.)

MPI_Cart_create

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

- It returns a a handle to a new communicator to which the Cartesian topology information is attached.
- If reorder = false then the rank of each process in the new group is identical to its rank in the old group.
- Otherwise it may reorder to choose a good embedding of the virtual topology onto the physical machine.
  - `comm_old`: handle to input communicator.
  - `ndims`: number of dimensions of Cartesian grid.
  - `dims`: integer array of size ndims specifying the number of processes in each dimension.
  - `periods`: logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension.
- If the total size of the Cartesian grid is smaller than the size of the group of comm, then some processes are returned MPI_COMM_NULL.
- The call is erroneous if it specifies a grid that is larger than the group size.

MPI Cartesian Topology Functions (Cont.)

- This code snippet uses MPI_Cart_create to remap the process ranks from a linear ordering (0, 1,. . . ,5) to 2-dimensional array of 3 rows by 2 columns ((0,0),(0,1),. . . ,(2,1)).
- We are able to assign work to the processes by their grid topology instead of their linear process rank.
- We imposed periodicity on the first dimension. This means any reference beyond the first or last entry of the columns it cycles back to the last and first entry, respectively.
- Any reference to column index outside the range returns MPI_PROC_NULL.

MPI_Cart_create (code snippet)

```
#include "mpi.h"
MPI_Comm old_comm, new_comm;
int ndims, reorder, periods[2], dim_size[2];
old_comm = MPI_COMM_WORLD;
ndims = 2; /* 2-D matrix/grid */
dim_size[0] = 3; /* rows */
dim_size[1] = 2; /* columns */
periods[0] = 1; /* row periodic
                   (each column forms a ring) */
periods[1] = 0; /* columns nonperiodic */
reorder = 1; /* allows processes reordered
                for efficiency */
MPI_Cart_create(old_comm, ndims, dim_size,
                periods, reorder, &new_comm);
```

MPI Cartesian Topology Functions (Cont.)

- Messages are still sent to and received from process's ranks.
- MPI provides routines to map or convert ranks to cartesian coordinates and vice-versa:

MPI_Cart_coords

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

- It provides a mapping of ranks to Cartesian coordinates.
  - `rank`: rank of a process within group of comm.
  - `maxdims`: length of vector coords in the calling program.
  - `coords`: Integer array of size ndims (defined by MPI_Cart_create call) containing the Cartesian coordinates of specified process.

MPI_Cart_rank

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

- It translates the logical process coordinates to process ranks.

MPI Cartesian Topology Functions (Cont.)

MPI_Cart_coords (code snippet)

```c
MPI_Cart_create(old_comm, ndims, dim_size, periods, reorder, &new_comm);
if(rank == 0) { /* only want to do this on one process */
  for (rank=0; rank<p; rank++) {
    MPI_Cart_coords(new_comm, rank, ndims, &coords);
    printf("%d, %d, %d\n ",rank, coords[0], coords[1]);
  }
}
```

MPI_Cart_rank (code snippet)

```c
MPI_Cart_create(old_comm, ndims, dim_size, periods, reorder, &new_comm);
if(rank == 0) { /* only want to do this on one process */
  for (i=0; i<nv; i++) {
    for (j=0; j<mv; j++) {
      coords[0] = i;
      coords[1] = j;
      MPI_Cart_rank(new_comm, coords, &rank);
      printf("%d, %d, %d\n",coords[0],coords[1],rank);
    }
  }
}
```

MPI Cartesian Topology Functions (Cont.)

MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *comm_new)
```

- It partitions a communicator into subgroups, which form lower-dimensional Cartesian subgrids.
- It builds for each subgroup a communicator with the associated subgrid Cartesian topology.
  - `remain_dims`: logical vector indicating if the ith dimension corresponding to the ith entry of remain_dims, is kept in the subgrid (true) or is dropped (false).

MPI_Cart_sub (code snippet)

```
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder, &comm2D);
MPI_Comm_rank(comm2D, &id2D);
MPI_Cart_coords(comm2D, id2D, ndim, coords2D);
/* Create 1D row subgrids */
belongs[0] = 0;
belongs[1] = 1; /* this dimension belongs to subgrid */
MPI_Cart_sub(comm2D, belongs, &commrow);
/* Create 1D column subgrids */
belongs[0] = 1; /* this dimension belongs to subgrid */
belongs[1] = 0;
MPI_Cart_sub(comm2D, belongs, &commcol);
```

MPI Cartesian Topology Functions (Cont.)

- It is common on large programs to create the cartesian topology with associated communicator in one routine while being used in another.
- The follow two functions help retrieving information about these communicators:

MPI_Cartdim_get

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

- It retrieves the number of dimensions from a communicator with Cartesian structure.

MPI_Cart_get

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
                 int coords[])
```

- It retrieves information from a communicator with Cartesian topology:
  - maxdims: Length of vectors dims, periods, and coords in the calling program.
  - dims: Number of processes for each Cartesian dimension.
  - periods: Periodicity for each Cartesian dimension.
  - coords: Coordinates of the calling process in Cartesian structure.

MPI Cartesian Topology Functions (Cont.)

MPI_Cart_shift

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

- It returns the shifted source and destination ranks, given a shift direction and amount.
  - `direction`: Coordinate dimension of shift, i.e., the coordinate whose value is modified by the shift.
  - `disp`: Displacement (> 0: upward shift, < 0: downward shift).
  - `rank_source`: Rank of source process.
  - `rank_dest`: Rank of destination process.
- A MPI_Sendrecv operation is likely to be used along a coordinate direction to perform a shift of data.
  - As input, it takes the rank of a source process for the receive, and the rank of a destination process for the send.
  - MPI_Cart_shift provides MPI_Sendrecv with the above identifiers.

MPI Cartesian Topology Functions (Cont.)

MPI_Cart_shift (code snippet)

```
/* Create Cartesian topology for processes */
ndim = 2; /* number of dimensions */
dims[0] = nrow; /* number of rows */
dims[1] = mcol; /* number of columns */
period[0] = 1; /* cyclic in this direction */
period[1] = 0; /* not cyclic in this direction */
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder,
&comm2D);
MPI_Comm_rank(comm2D, &me);
MPI_Cart_coords(comm2D, me, ndim, &coords);
displ = 1; /* shift by 1 */
index = 0; /* shift along the 1st index (out of 2) */
MPI_Cart_shift(comm2D, index, displ, &source0, &dest0);
index = 1; /* shift along the 2nd index (out of 2) */
MPI_Cart_shift(comm2D, index, displ, &source1, &dest1);
```

- MPI_Cart_shift is used to obtain the source and destination rank numbers of the calling process. There are two calls to MPI_Cart_shift, the first shifting along columns, and the second along rows.

Conclusion

Recap

- MPI Basics Review
- Scientific MPI Example: 2D Diffusion Equation
- Derived Data Types
- Application Topology

Good References

- W. Gropp, E. Lusk, and A. Skjellun, Using MPI: Portable Parallel Programming with the Message-Passing Interface. Third Edition. (MIT Press, 2014).
- W. Gropp, T. Hoefler, R. Thakur, E. Lusk, Using Advanced MPI: Modern Features of the Message-Passing Interface. (MIT Press, 2014).
- A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Second Edition. (Addison-Wesley, 2003) (A bit old but still reasonable)
- The man pages for various MPI commands.
- http://www.mpi-forum.org/docs/ for MPI standard specification.