Topologies in MPI

Instructor: Dr. M. Taufer

Topology

- We can associate additional information (beyond the group and the context) to a communicator.
- A linear ranking of processes may not adequately reflect the logical communication pattern of the processes:
  - usually a optimal ranking is determined by the underlying problem geometry and the numerical algorithm used
- A topology:
  - can provide a convenient naming mechanism for the processes of a group (within a communicator)
  - may assist the runtime system in mapping the processes onto hardware
Topology

- By creating a topology, you create a new communicator
- Two main types of topologies supported by MPI:
  - Cartesian (grid)
  - Graph

Types of Topologies

- Cartesian
  - 1-D
  - 2-D
  - 3-D
  - Tori
  - Hypercube
- General
  - Trees
  - General graphs
**MPI_Cart_create**

- Creates a new communicator to which Cartesian topology information has been attached.

- **C**
  ```c
  MPI_Cart_create (comm_old, ndims, *dims[], *periods, reorder, *comm_cart)
  ```

- **FORTRAN**
  ```fortran
  MPI_CART_CREATE (comm_old, ndims, dims(), periods, reorder, comm_cart, ierr)
  ```

**Input parameters**
- `comm_old` - input communicator (handle)
- `ndims` - number of dimensions of cartesian grid (integer)
- `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
- `reorder` - ranking may be reordered (true) or not (false) (logical)

**Output parameters**
- `comm_cart` - communicator with new cartesian topology (handle)
Example

- Map (or rename) 6 processes from a linear ordering (0,1,2,3,4,5) into a two-dimensional matrix ordering of 3 rows by 2 columns (i.e., (0,0), (0,1), ..., (2,1)

<table>
<thead>
<tr>
<th>(0,0)</th>
<th>(1,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2,0)</td>
<td>(3,1)</td>
</tr>
<tr>
<td>(4,0)</td>
<td>(5,1)</td>
</tr>
</tbody>
</table>

- With processes renamed in a 2D grid topology, we will be able to assign or distribute work, or distinguish among the processes by their grid topology rather than by their linear process ranks.

```c
include "mpif.h"
integer old_comm, new_comm, ndims, reorder, ierr
integer dim_size(2)
logical periods(0:1)
old_comm = MPI_COMM_WORLD
ndims = 2   ! 2D matrix/grid
dim_size(1) = 3   ! rows
dim_size(2) = 2   ! columns
periods(0) = .true.
! row-periodic (each column wraps around)
periods(1) = .false.
! column-nonperiodic
reorder = 1
call MPI_Cart_create(old_comm, ndims, dim_size, &
                     periods, reorder, new_comm, ierr)
```
Reordering

- If reorder is true, then the MPI runtime environment is supposed to reorder processes while creating the new communicator
- Initial order of processes (0, 1, ..., m-1) as preferred by the OS
- Reordering of processes may improve the communication performance
Reordering Example: Cube

Hardware topology

Communication pattern as in the code

Initial topology

Optimized topology

Communication pattern as in the code
Periodicity (I)

\[
\begin{array}{ccc}
0(-1,0) & 5(-1,1) \\
0(0,0) & 1(0,1) & -1(0,2) \\
2(1,0) & 3(1,1) & -1(1,2) \\
4(2,0) & 5(2,1) & -1(2,2) \\
0(3,0) & 1(3,1) \\
\end{array}
\]

\[\text{periods}(0) = \text{true.} \quad \text{periods}(1) = \text{false.}\]

Periodicity (II)

\[
\begin{array}{ccc}
-1(-1,0) & -1(-1,1) \\
1(0,-1) & 0(0,0) & 1(0,1) & 0(0,2) \\
3(1,-1) & 2(1,0) & 3(1,1) & 2(1,2) \\
4(2,-1) & 4(2,0) & 5(2,1) & 4(2,2) \\
-1(3,0) & -1(3,1) \\
\end{array}
\]

\[\text{periods}(0) = \text{false.} \quad \text{periods}(1) = \text{true.}\]
**MPI_Cart_create: Parameters**

<table>
<thead>
<tr>
<th></th>
<th>2-D grid</th>
<th>cube</th>
<th>2-D torus</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm_old</td>
<td>MPI_COMM_WORLD</td>
<td>MPI_COMM_WORLD</td>
<td>MPI_COMM_WORLD</td>
</tr>
<tr>
<td>ndims</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>dims</td>
<td>(2, 2)</td>
<td>(2, 2, 2)</td>
<td>(3, 4)</td>
</tr>
<tr>
<td>periods</td>
<td>(false, false)</td>
<td>(false, false)</td>
<td>(true, true)</td>
</tr>
<tr>
<td>reorder</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>comm_cart</td>
<td>twodim</td>
<td>cube</td>
<td>twotorus</td>
</tr>
</tbody>
</table>

**Topology Calls**

- MPI_Cart_create
- MPI_Cart_get
- MPI_Cart_rank
- MPI_Cart_sub
- MPI_Cart_coords
- MPI_Cart_map
- MPI_Cart_shift
- MPI_Cartdim_get
- MPI_Dims_create
- MPI_Topo_test
- MPI_Graphdims_get
- MPI_Graph_get
- MPI_Graph_map
- MPI_Graph_neighbors
- MPI_Graph_create
- MPI_Graph_neighbors_count
Optimum Splitting

- How do we split processes into a Cartesian grid? Given 192 processes to be split in a three-dimensional grid:
  - 2*2*48, 2*4*24, 2*8*12, ...
- MPI_Dims_create creates a division of processors in a Cartesian grid © find an optimum splitting
  int MPI_Dims_create(nnodes, ndims, dims)
  int nnodes;
  int ndims;
  int *dims;
- INPUT PARAMETERS
  nnodes - number of nodes in a grid (integer)
  ndims - number of cartesian dimensions (integer)
- IN/OUT PARAMETER
  dims - integer array of size ndims specifying the number of nodes in each dimension

MPI_Cart_rank

- MPI_Cart_rank retrieves the rank from a Cartesian communicator
- C
  ierr= MPI_Cart_rank( comm_cart, coords, rank)
- FORTRAN
  MPI_CART_RANK(comm_cart, coords, rank, ierr)

where rank is the rank of process specified by coords
**MPI_Cart_get**

- **MPI_Cart_get** retrieves the Cartesian coordinate information from the Cartesian communicator.
- C
  
  ```c
  ierr = MPI_Cart_get( comm_cart, maxdims, &dims, &iperiod, &coords)
  ```
- FORTRAN
  
  ```fortran
  MPI_CART_GET(comm_cart, maxdims, dims, iperiod, coords, ierr)
  ```

  - maxdims is the maximum dimensions of vectors dims, periods, coords
  - dims is the array of dimensions of Cartesian topology
  - periods expresses the periodicity (true or false) for each dimensions
  - coords is the coordinates of calling process

**MPI_Cart_coords**

- **MPI_Cart_coords** maps from rank to Cartesian coordinates
- C
  
  ```c
  ierr = MPI_Cart_coords( comm_cart, rank, maxdims, coords)
  ```
- FORTRAN
  
  ```fortran
  MPI_CART_COORDS( comm_cart, rank, maxdims, coords, ierr)
  ```

  where coords is the coordinates of the process specify by rank
**MPI_Cart_shift**

- **MPI_Cart_shift** determines the source rank and destination rank for shifting data to neighbors removed by a known number of steps.
- C
  
  ```c
  ierr = MPI_Cart_shift (comm_cart, direction, disp, rank_source, rank_dest)
  ```

- **FORTRAN**
  
  ```fortran
  MPI_CART_SHIFT(comm_cart, direction, disp, rank_source, rank_dest, ierr)
  ```
  
  - directions: 1 < direction < ndims
  - disp: > 0 upward; < 0 downwards

---

**Column Shift**

A(i,j) a local matrix labeled by the initial processor grid. A periodic (wrap around) shifting is assumed.

![Column Shift Diagram](image.png)
Row Shift

B(i,j) a local matrix labeled by the initial processor grid. A periodic (wrap around) shifting is assumed.

Master-Slave Model

Instructor Dr. Taufer
Master/Slave Algorithm

- One master node sends out the task(s) to $p$ slave nodes
- Many parallel coarse-grained applications are implemented using this algorithm
- The tasks are independent

Program Analysis

- For the programs in Hoffman article:
  - Sketch the graph of the communication between master and slave. For each communication report the size. Keep your graph simple considering only one master and one slave
  - Which program is more efficient? Motivate.
  - Does the program enclose a dynamic load balancing to ensure maximum utilization of the resources?
- Code 2 can be used to test network performance:
  - By altering the size of RESULT_SIZE and NUM_CATEGORIES you can measure the impact of a varying number of messages with different size
  - Implement a simple case study and summarize the results
Background

- In March 1995, the MPI Forum began discussing enhancements to the MPI standard. Following this:
  - December 1995: Supercomputing 95 conference - Birds of a Feather meeting to discuss MPI-2 extensions.
- These enhancements have now become part of the MPI-2 standard.
What was New in MPI-2

• Dynamic processes
  • extensions which remove the static process model of MPI and
to provide routines to create new processes.
• One sided communications
  • provides routines for one directional communications. Include
shared memory operations (put/get) and remote accumulate
operations.
• Extended collective operations
  • allows for non-blocking collective operations and application of
collective operations to inter-communicators
• External interfaces
  • defines routines which allow developers to layer on top of MPI,
such as for debuggers and profilers.
• Additional language bindings
  • describes C++ bindings and discusses Fortran-90 issues.
• Parallel I/O (MPI-IO)
  • discusses MPI support for parallel I/O.

Process Management

• MPI-1 applications are static
  • The number of processes cannot change as the application
runs.
• MPI-2 allows process creation and management of
processes after the start of the application
• PVM has offered this feature for years
MPICH2 and LAM/MPI

- MPICH2 and LAM/MPI has been extended to support subset of the MPI-2 features
  - Both provide support for MPI-IO through ROMIO
  - Offer mpiexec command line startup
  - Provide dynamic process management
  - Offer C++ binding
  - Implement basic one-sided communication

- If you want to see more about how to install LAM/MPI and MPICH2 look at:

MPI_Comm_spawn (I)

- A group of processes can create another group of processes with MPI_Comm_spawn:

```
#include <mpi.h>
int MPI_Comm_spawn(char* command, char** argv, int maxprocs,
                    MPI_Info info, int root, MPI_Comm comm, MPI_Comm *
                    intercomm, int *errcodes)
```

**INPUT PARAMETERS**
- command: name of program to spawn (only significant at root)
- argv: arguments to command (only significant at root)
- maxprocs: max number of processes to start (only significant at root)
- info: startup hints
- root: rank of process to perform the spawn
- comm: parent intracommunicator

**OUTPUT PARAMETERS**
- intercomm: child intercommunicator containing spawned processes
- errcodes: one code per process
**MPI_Comm_spawn and MPI_Comm_spawn_multiple**

**MPI_Comm_spawn:**
- MPI_Comm_spawn is a collective operation over the parent communicator.
- The child group starts up like any MPI application. The processes must begin by calling `MPI_Init`, after which the pre-defined communicator, `MPI_COMM_WORLD`, may be used.
- This world communicator contains only the child processes. It is distinct from the `MPI_COMM_WORLD` of the parent processes.

**MPI_Comm_spawn_multiple:**
- `MPI_Comm_spawn_multiple` is used to manually specify a group of different executables and arguments to spawn.
- `MPI_Comm_spawn` is used to specify one executable and set of arguments

---

**MPI_Attr_get()**

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
    int *usize;
    int flag;
    MPI_Init(&argc, &argv);
    MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, &usize, &flag);
    if (flag == 1)
        printf("Got universe size: %d\n", *usize);
    else
        printf("Didn't get universe size\n");
    MPI_Finalize();
    return 0;
}
```