Derived Data Types, Group and Communicator Management Routines

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

Derived Datatypes

- While your data may occasionally be that well behaved, it is likely that you will need to transmit collections of data of mixed types defined by the program, or data that are scattered in memory.
- Create your own types to suit your application
  - convenience
  - efficiency
- You can define new data structures based upon sequences of the MPI primitive data types, these are derived data types.
- Derived datatypes can be used in all send and receive operations including collective.

MPI Datatypes

- MPI Primitive Datatypes
  - **`MPI_Int, MPI_Float, MPI_INTEGER`**, etc.
- Derived Datatypes - can be constructed by four methods:
  - contiguous
  - vector
  - indexed
  - struct
- Must always call **`MPI_Type_commit`** before using a constructed datatype.

User Defined Datatypes

- **Methods for creating data types**

```
MPI_Type_contiguous();
MPI_Type_vector();
MPI_Type_indexed();
MPI_Type_struct();
MPI_Pack();
MPI_Unpack();
```

- **MPI allows datatypes to be defined in much the same way as modern programming languages (C,C++,F90)**
- **This allows your communication and I/O operations to operate using the same datatypes as the rest of your program**
- **Makes expressing the partitioning of datasets easier**

Contiguous Array

- **Creates an array of counts elements:**

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

Example

```
const int N = 10;

double A[N][N];
double B[N][N];

MPI_Datatype  matrix;

MPI_Type_contiguous(N*N, MPI_DOUBLE, &matrix);
MPI_Type_commit(&matrix);

if (rank == master_rank
    MPI_Send(A, 1, matrix, 1, 10, comm);
else if( rank == 1 )
    MPI_Recv(B, 1, matrix, 0, 10, comm, &status);
```

MPI_Type_vector

int MPI_Type_vector(int count,

int blocklength,

int stride,

MPI_Datatype old_type,

MPI_Datatype *newtype_p)

C Syntax

```
#include <mpi.h>
int MPI_Type_vector(int count, int blocklength, int stride,
     MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Fortran Syntax

```
INCLUDE 'mpif.h'
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
          IERROR)
     INTEGER   COUNT, BLOCKLENGTH, STRIDE, OLDTYPE
     INTEGER   NEWTYPE, IERROR
```

C++ Syntax

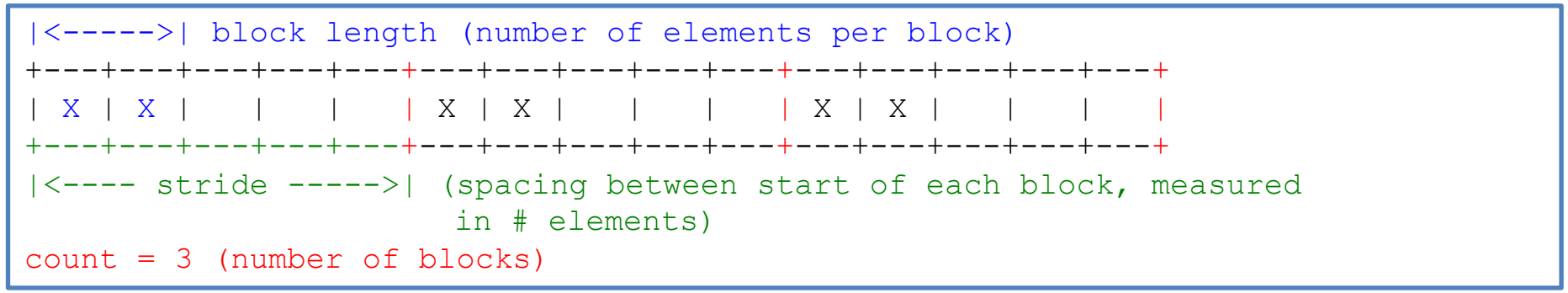
```
#include <mpi.h>
Datatype Datatype::Create_vector(int count, int blocklength,
     int stride) const
```

INPUT PARAMETERS

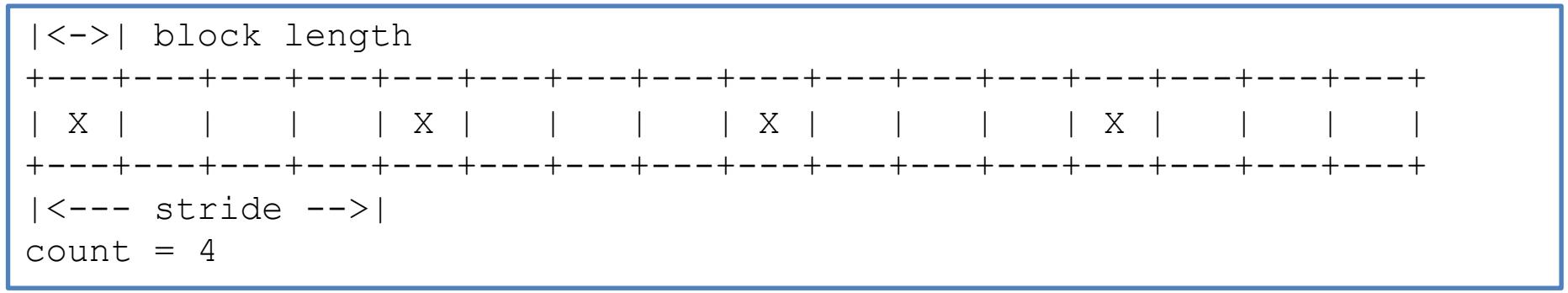
```
count     Number of blocks (nonnegative integer).

blocklength
          Number of elements in each block (nonnegative integer).

stride    Number of elements between start of each block (integer).

oldtype   Old datatype (handle).
```

MPI_Type_vector

MPI_Type_vector(3, 2, 5, MPI_INT, &newtype);



MPI_Type_vector(4, 1, 4, MPI_INT, &newtype);



Committing types

- **In order for a user-defined derived datatype to be used as an argument to other MPI calls, the type must be "committed".**

```
MPI_Type_commit(type);
MPI_Type_free(type)
```

- **Use commit after calling the type constructor, but before using the type anywhere else**
- **Call free after the type is no longer in use (no one actually does this, but it makes computer scientists happy...)**

Lab: MPI_Type_vector Example (mpi_type_vector.cpp)

```
/**
 * @brief Illustrates how to create a vector MPI datatype.
 * @details This program is meant to be run with 2 processes: a sender and a
 * receiver. These two MPI processes will exchange a message made of three
 * integers. On the sender, that message is in fact the middle column of an
 * array it holds, which will be represented by an MPI vector.
 *
 *
 *     Full array          What we want
 *                            to send
 * +-----+-----+-----+  +-----+-----+-----+
 * |  0  |  1  |  2  |  |  -  |  1  |  -  |
 * +-----+-----+-----+  +-----+-----+-----+
 * |  3  |  4  |  5  |  |  -  |  4  |  -  |
 * +-----+-----+-----+  +-----+-----+-----+
 * |  6  |  7  |  8  |  |  -  |  7  |  -  |
 * +-----+-----+-----+  +-----+-----+-----+
 *
 * How to extract a column with a vector type:
 *
 *                 distance between the
 *               start of each block: 3 elements
 *           <---------------> <---------------->
 *           |                 |                 |
 *       start of          start of          start of
 *       block 1           block 2           block 3
 *          |                 |                 |
 *          V                 V                 V
 *    +-----+-----+-----+-----+-----+-----+-----+-----+-----+
 *    |  -  |  1  |  -  |  -  |  4  |  -  |  -  |  7  |  -  |
 *    +-----+-----+-----+-----+-----+-----+-----+-----+-----+
 *           <--->             <--->             <--->
 *          block 1           block 2           block 3
 *
 * Block length: 1 element
 * Element: MPI_INT
 **/
```

Lab: MPI_Type_vector Example (mpi_type_vector.cpp)

```cpp
#include <iostream>
#include <mpi.h>        // MPI header file
#define MASTER 0
#define WORKER 1

/**
 * @brief Illustrates how to create a vector MPI datatype.
 * @details This program is meant to be run with 2 processes: a sender and a
 * receiver. These two MPI processes will exchange a message made of three
 * integers. On the sender, that message is in fact the middle column of an
 * array it holds, which will be represented by an MPI vector.
 *
 *
 *     Full array          What we want
 *                            to send
 * +-----+-----+-----+  +-----+-----+-----+
 * |  0  |  1  |  2  |  |  -  |  1  |  -  |
 * +-----+-----+-----+  +-----+-----+-----+
 * |  3  |  4  |  5  |  |  -  |  4  |  -  |
 * +-----+-----+-----+  +-----+-----+-----+
 * |  6  |  7  |  8  |  |  -  |  7  |  -  |
 * +-----+-----+-----+  +-----+-----+-----+
 *
 * How to extract a column with a vector type:
 *
 *                 distance between the
 *             start of each block: 3 elements
 *          <---------------> <--------------->
 *         |                 |                 |
 *      start of          start of          start of
 *      block 1           block 2           block 3
 *         |                 |                 |
 *         V                 V                 V
 *   +-----+-----+-----+-----+-----+-----+-----+-----+-----+
 *   |  -  |  1  |  -  |  -  |  4  |  -  |  -  |  7  |  -  |
 *   +-----+-----+-----+-----+-----+-----+-----+-----+-----+
 *          <--->             <--->             <--->
 *         block 1           block 2           block 3
 *
 * Block length: 1 element
 * Element: MPI_INT
 **/

using namespace std;

int main(int argc, char **argv) {
  int nprocs, rank;

  // initialize for MPI
  MPI_Init(&argc, &argv);
  // get number of processes
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  // get the rank = this process's number (ranges from 0 to nprocs - 1)
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // master send value to workers
  if (rank == MASTER)
  {
    // Create the datatype
    MPI_Datatype column_type;
    MPI_Type_vector(3, 1, 3, MPI_INT, &column_type);
    MPI_Type_commit(&column_type);

    // send the message
    int buffer[3][3] = { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
    MPI_Request request;
    cout << "MPI process " << rank << " sends values "
         << buffer[0][1] << ", " << buffer[1][1]
         << ", and " << buffer[2][1] << endl;
    MPI_Send(&buffer[0][1], 1, column_type, WORKER, 0, MPI_COMM_WORLD);
  }

  // workers receive the message
  if (rank == WORKER)
  {
    int received[3];
    MPI_Recv(&received, 3, MPI_INT, MASTER, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    cout << "MPI process " << rank << " received values "
         << received[0] << ", " << received[1]
         << ", and " << received[2] << endl;
  }

  // clean up for MPI
  MPI_Finalize();   // C style

  return 0;
}
```

Subarrarys

- Subarray type let's you divide a multi-dimensional array into smaller blocks

```
int MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
   array_of_starts, order, oldtype, newtype)
int          ndims;
int          *array_of_sizes;
int          *array_of_subsizes;
int          *array_of_starts;
int          order;
MPI_Datatype oldtype;
MPI_Datatype *newtype;
```

Structured Records

- **Allows different types to be combined**

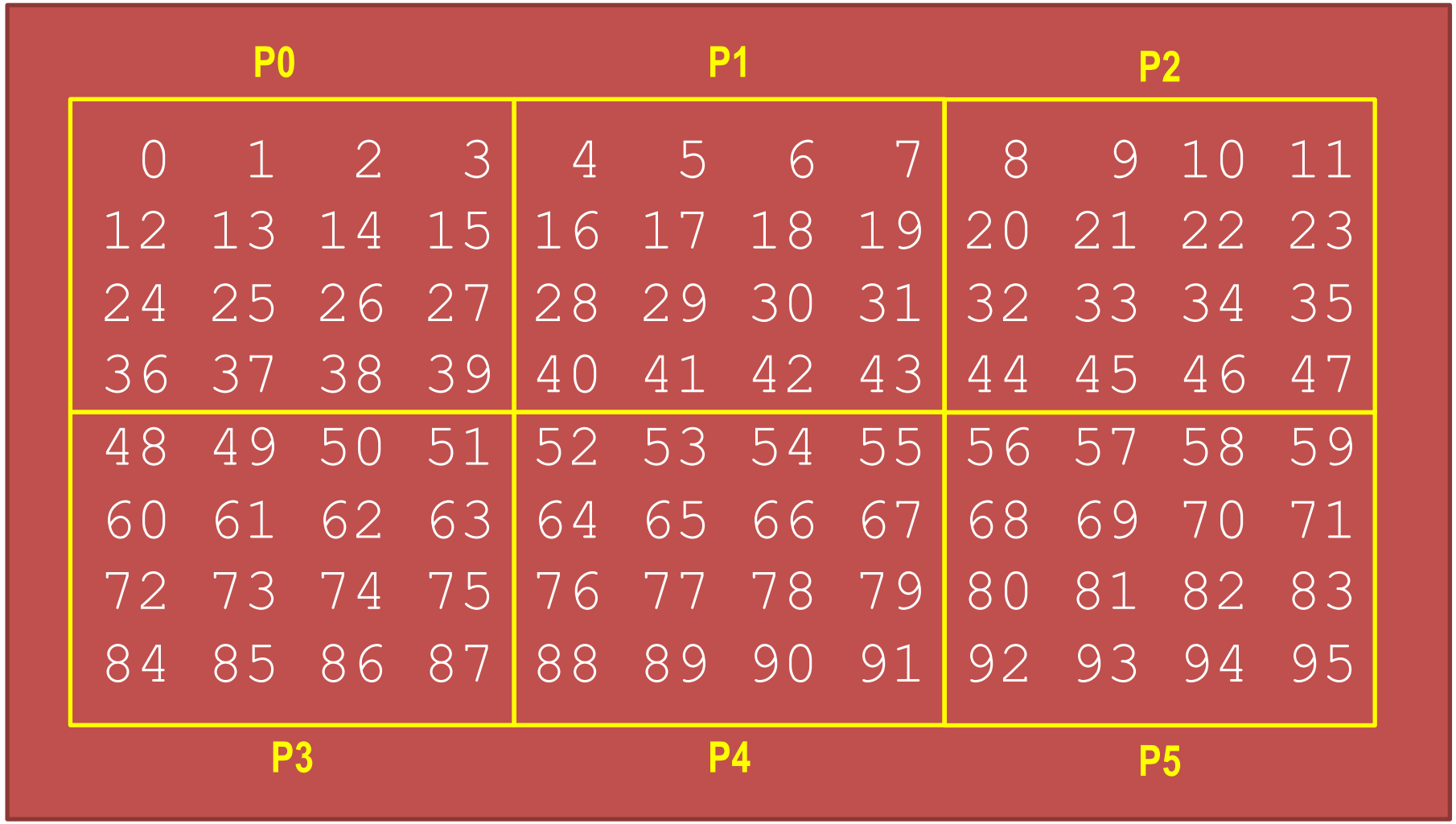
```
MPI_Type_struct(int count,
    int *array_of_blocklengths,
    MPI_Aint *array_of_displacements,
    MPI_Datatype *array_of_types,
    MPI_Datatype *newtype);
```

- **Blocklengths specified in number of elements**
- **Displacements specified in bytes**

Partition Matrix into Blocks

```
 0  1  2  3  4  5  6  7  8  9 10 11
12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95
```

Partition Matrix into Blocks



Dealing with Communicators

- Many MPI operations deal with all the processes in a communicator
- MPI_COMM_WORLD by default contains every task in your MPI job
- Other communicators can be defined for more complex operations; for different parts of the task, to add topology, to segregate different kinds of messaging
- MPI allows you to create subsets of communicators

Why Communicators?

- Isolate communication to a small number of processors
- Useful for creating libraries
- Different processors can work on different parts of the problem
- Useful for communicating with "nearest neighbors"

Communicators and Groups

- Many MPI users are only familiar with the communicator MPI_COMM_WORLD
- A communicator can be thought of a handle to a group
- A group is an ordered set of processes
  - Each process is associated with a rank
  - Ranks are contiguous and start from zero
- For many applications (dual level parallelism) maintaining different groups is appropriate
- Groups allow collective operations to work on a subset of processes
- Information can be added onto communicators to be passed into routines

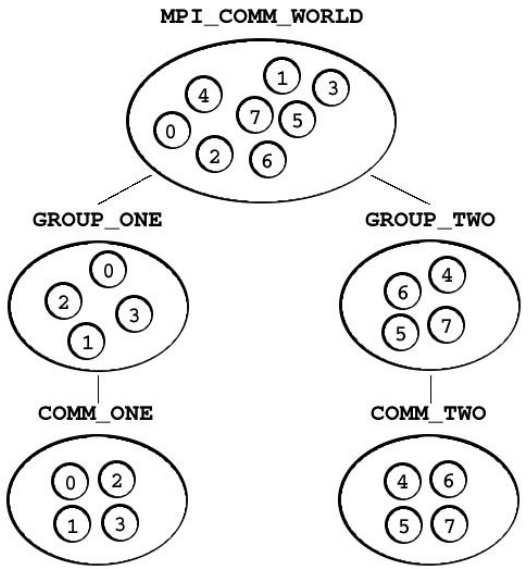
MPI Communication Features

- Communicators are divided into two kinds: intra-communicators for operations within a single group of processes and inter- communicators for operations between two groups of processes.
- Communicators (see below) provide a “caching” mechanism that allows one to associate new attributes with communicators, on a par with MPI built-in features. E.g., virtual topologies.
- Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving).
  - Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations.

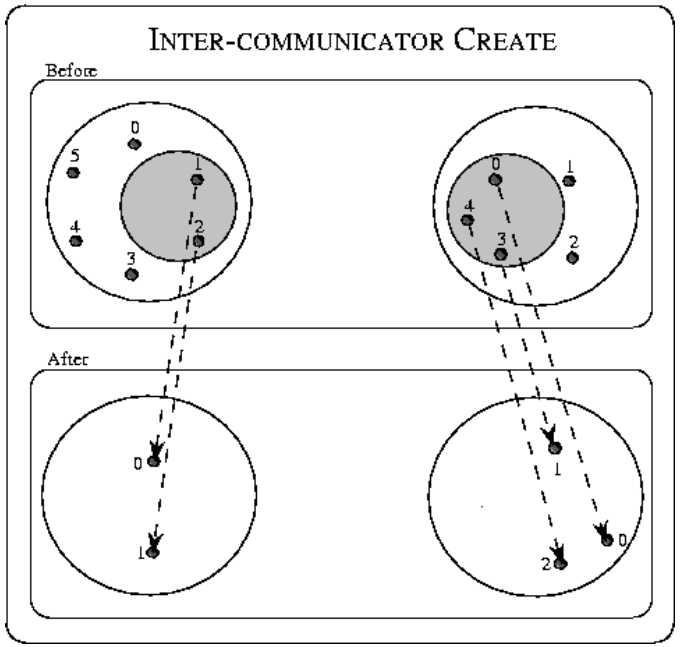
Communicators and Groups(cont)

- An intracommunicator is used for communication within a single group
- An intercommunicator is used for communication between 2 disjoint groups

intracommunicator



intercommunicator



MPI_Comm_create

- MPI_Comm_create creates a new communicator newcomm with group members defined by a group data structure

Name

MPI_Comm_create - Creates a new communicator.

Syntax

C Syntax

```
#include <mpi.h>
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

Fortran Syntax

```
INCLUDE 'mpif.h'
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER    COMM, GROUP, NEWCOMM, IERROR
```

C++ Syntax

```
#include <mpi.h>
MPI::Intercomm MPI::Intercomm::Create(const Group& group) const
MPI::Intracomm MPI::Intracomm::Create(const Group& group) const
```

How do you define a group?

MPI_Comm_group

- Given a communicator, MPI_Comm_group returns in group associated with the input communicator

Name

MPI_Comm_group - Returns the group associated with a communicator.

Syntax

C Syntax

```
#include <mpi.h>
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Fortran Syntax

```
INCLUDE 'mpif.h'
MPI_COMM_GROUP(COMM, GROUP, IERROR)
      INTEGER    COMM, GROUP, IERROR
```

C++ Syntax

```
#include <mpi.h>
Group Comm::Get_group() const
```

MPI_Group_incl

- Produces a group by reordering an existing group and taking only listed members.

Syntax

C Syntax

```
#include <mpi.h>
int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
    MPI_Group *newgroup)
```

Fortran Syntax

```
INCLUDE 'mpif.h'
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER    GROUP, N, RANKS(*), NEWGROUP, IERROR
```

C++ Syntax

```
#include <mpi.h>
Group Group::Incl(int n, const int ranks[]) const
```

MPI_Comm_split

- Creates new communicators based on colors and keys.

Syntax

C Syntax

```
#include <mpi.h>
int MPI_Comm_split(MPI_Comm comm, int color, int key,
    MPI_Comm *newcomm)
```

Fortran Syntax

```
INCLUDE 'mpif.h'
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER    COMM, COLOR, KEY, NEWCOMM, IERROR
```

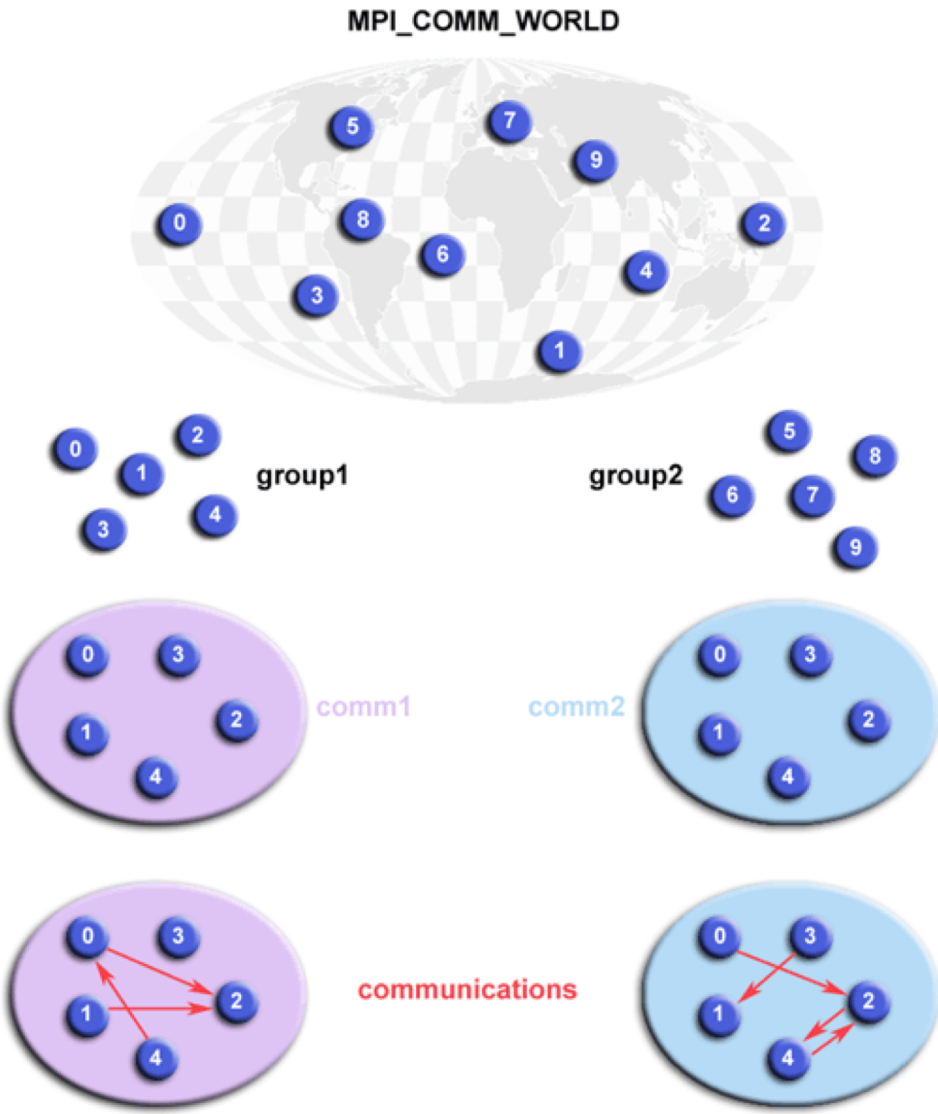
C++ Syntax

```
#include <mpi.h>
MPI::Intercomm MPI::Intercomm::Split(int color, int key) const
MPI::Intracomm MPI::Intracomm::Split(int color, int key) const
```

```
my_row = my_rank/q;
MPI_Comm_split(MPI_COMM_WORLD,my_row,my_rank,&my_row_comm);
```

Typical usage:

- Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
- Form new group as a subset of global group using MPI_Group_incl
- Create new communicator for new group using MPI_Comm_create
- Determine new rank in new communicator using MPI_Comm_rank
- Conduct communications using any MPI message passing routine
- When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free



Lab: MPI_Comm_split

```cpp
# include <iostream>
# include <cstdlib>
# include "mpi.h"

using namespace std;

int main(int argc, char **argv)
{
    // Get the rank and size in the original communicator
    int rank, nprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    // Split the communicator based on the color (global rank/4)
    // Your Code
    int color = rank / 4;
    MPI_Comm group_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &group_comm);

    // get new rank and size
    int group_rank, group_nprocs;
    MPI_Comm_rank(group_comm, &group_rank);
    MPI_Comm_size(group_comm, &group_nprocs);

    printf("World_Rank/World_Nprocs: %d/%d \t Group_Rank/Group_Nprocs: %d/%d\n",
            rank, nprocs, group_rank, group_nprocs);

    // MPI Comm Free
    MPI_Comm_free(&group_comm);

    MPI_Finalize();
    return 0;
}
```

Virtual Topologies

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

Topology

- A topology is a mechanism for associating different addressing schemes with processes.
- A topology can provide a convenient naming mechanism for processes.
- A topology can allow MPI to optimize communications.
- Simplifies writing of code.
- There are virtual process topology and topology of the underlying hardware.
- The virtual topology can be exploited by the system in assigning of processes to processors.

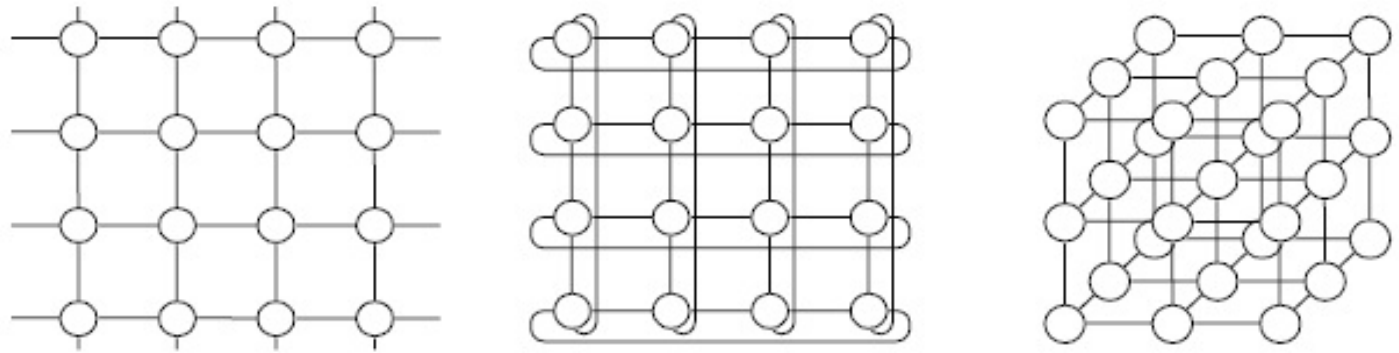
Introduction

- Any process topology can be represented by graphs.
- MPI provides defaults for ring, mesh, torus and other common structures

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

How to Use a Virtual Topology

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



2D mesh 2D torus 3D mesh

Topology Types

- Cartesian topologies
  - Each process is connected to its neighbors in a virtual grid
  - Boundaries can be cyclic
  - Processes can be identified by cartesian coordinates

- Graph topologies
  - Any process topology can be represented by graphs
  - General graphs
  - Will not be covered here

Cartesian Topology

- *Cartesian topologies* assume the presentation of a set of processes as a rectangular grid and the use of Cartesian coordinate system for pointing to the processes,
- Cartesian structures of arbitrary dimensions
- Can be periodic along any number of dimensions
- Popular Cartesian structures – linear array, ring, rectangular mesh, cylinder, torus (hypercubes)

Cartesian Topology

- Process coordinates begin with 0
- Row-major numbering
- Example: 12 processes arranged on a 3 x 4 grid

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

Virtual Topology MPI Routines

- Some of the MPI topology routines are
  - MPI_CART_CREATE
  - MPI_CART_COORDS
  - MPI_CART_RANK
  - MPI_CART_SUB
  - MPI_CARTDIM_GET
  - MPI_CART_GET
  - MPI_CART_SHIFT

MPI_Cart_create

- Used to create Cartesian coordinate structures, of arbitrary dimensions, on the processes. The new communicator receives no cached information.

C Syntax

```
#include <mpi.h>
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
    const int periods[], int reorder, MPI_Comm *comm_cart)
```

Fortran Syntax

```
INCLUDE 'mpif.h'
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER,
        COMM_CART, IERROR)
    INTEGER    COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL    PERIODS(*), REORDER
```

C++ Syntax

```
#include <mpi.h>
Cartcomm Intracomm.Create_cart(int[] ndims, int[] dims[],
    const bool periods[], bool reorder) const
```

- ndims – number of dimensions of Cartesian grid
- dims – number of processes along each dimension
- periods – whether a dimension is cyclic (true) or not
- reorder – ranking of initial processes may be reordered (true) or not (rank preserved)

MPI_Cart_coords & MPI_Cart_rank

- Rank-to-coordinates translator

C Syntax

```
#include <mpi.h>
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
    int coords[])
```

Fortran Syntax

```
INCLUDE 'mpif.h'
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER    COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

C++ Syntax

```
#include <mpi.h>
void Cartcomm::Get_coords(int rank, int maxdims,
    int coords[]) const
```

- Coordinates-to-rank translator

C Syntax

```
#include <mpi.h>
int MPI_Cart_rank(MPI_Comm comm, int coords[], int *rank)
```

Fortran Syntax

```
INCLUDE 'mpif.h'
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER    COMM, COORDS(*), RANK, IERROR
```

C++ Syntax

```
#include <mpi.h>
int Cartcomm::Get_cart_rank(const int coords[]) const
```

MPI_Cart_coords & MPI_Cart_rank

```
int MPI_Cart_coords(MPI_Comm communicator,
                    int rank,
                    int dimension_number,
                    int* coords);
```

- rank – rank of a process within group of comm
- maxdims – length of vector coords in the calling program
- coords – array containing the Cartesian coordinates of specified process

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

- coords – array containing the Cartesian coordinates of the process

Lab: MPI_Cart_create Example

```cpp
# include <iostream>
# include <cstdlib>
# include "mpi.h"

using namespace std;

int main(int argc, char *argv[])
{
    // MPI Init and get rank
    int rank, nprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    // Variables required by Create_cart
    int ndims, dims[2], periods[2], reorder;
    ndims = 2;          // 2D matrix/grid
    dims[0] = 3;        // rows
    dims[1] = 4;        // columns
    periods[0] = true;  // row periodic (each column forms a ring)
    periods[1] = true;  // columns periodic (each row forms a ring)
    reorder = true;     // allows processes reordered for efficiency

    // Create a communicator given the 2D torus topology.
    MPI_Comm new_comm;
    MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &new_comm);

    // Torus rank in the new communicator
    int torus_rank;
    MPI_Comm_rank(new_comm, &torus_rank);

    // Get my coordinates in the new communicator
    int torus_coords[2];
    MPI_Cart_coords(new_comm, torus_rank, ndims, torus_coords);

    // Print my location in the 2D torus
    cout << "rank:" << rank
         << "> torus_rank=" << torus_rank
         << ", coords=(" << torus_coords[0] << "," << torus_coords[1] << ")" << endl;

    // MPI finalize and exit
    MPI_Finalize();
    return 0;
}
```

Lab: MPI_Cart_create Example

```
ai@ubuntu-20-04:~/Lab/MPI$ mpic++ mpi_cart_create.cpp -o mpi_cart_create
ai@ubuntu-20-04:~/Lab/MPI$ mpirun -np 12 ./mpi_cart_create
rank:6> torus_rank=6, coords=(1,2)
rank:8> torus_rank=8, coords=(2,0)
rank:10> torus_rank=10, coords=(2,2)
rank:0> torus_rank=0, coords=(0,0)
rank:2> torus_rank=2, coords=(0,2)
rank:4> torus_rank=4, coords=(1,0)
rank:5> torus_rank=5, coords=(1,1)
rank:7> torus_rank=7, coords=(1,3)
rank:9> torus_rank=9, coords=(2,1)
rank:1> torus_rank=1, coords=(0,1)
rank:3> torus_rank=3, coords=(0,3)
rank:11> torus_rank=11, coords=(2,3)
```

Sending and receiving in Cartesian topology

- There is no MPI_Cart_send or MPI_Cart_recv which would allow you to send a message to process (1,0) in your Cartesian topology, for example.
- You must use standard communication functions.
- There is a convenient way to obtain the rank of the desired destination/source process from you Cartesian coordinate grid.
- Usually one needs to determine which are the adjacent processes in the grid and obtain their ranks in order to communicate.

MPI_Cart_shift

- Returns the shifted source and destination ranks, given a shift direction and amount.

C Syntax

```
#include <mpi.h>
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
    int *rank_source, int *rank_dest)
```

Fortran Syntax

```
INCLUDE 'mpif.h'
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE,
        RANK_DEST, IERROR)
    INTEGER    COMM, DIRECTION, DISP, RANK_SOURCE
    INTEGER    RANK_DEST, IERROR
```

C++ Syntax

```
#include <mpi.h>
void Cartcomm::Shift(int direction, int disp, int& rank_source,
    int& rank_dest) const
```

- direction – the cartesian grid index, has range (0, 1, …, ndim-1). For a 2D grid, the 2 choices for direction are 0 and 1.
- displ: – Amount and sense of shift (<0: downward shifts, >0: upwards shifts, or 0)
- source – rank of process to receive data from
- dest – rank of process to send data to

Lab: MPI_Cart_shift Example

- Append below code into MPI_Cart_create Lab.

```
40
41      // Declare our neighbours
42      enum DIRECTIONS {DOWN, UP, LEFT, RIGHT};
43      const char* neighbours_names[4] = {"DOWN", "UP", "LEFT", "RIGHT"};
44      int neighbours_ranks[4];
45
46      // Let consider dims[0] = X, so the shift tells us our left and right neighbours
47      MPI_Cart_shift(new_comm, 0, 1, &neighbours_ranks[UP], &neighbours_ranks[DOWN]);
48      MPI_Cart_shift(new_comm, 1, 1, &neighbours_ranks[LEFT], &neighbours_ranks[RIGHT]);
49
50      // Get my rank
51      int my_rank;
52      MPI_Comm_rank(new_comm, &my_rank);
53
54      // Print my neighbors
55      for(int i = 0; i < 4; i++)
56      {
57         cout << "MPI rank=" << my_rank << " I have a " << neighbours_names[i]
58              << " neighbour rank=" << neighbours_ranks[i] << endl;
59      }
60
61      // MPI finalize and exit
62      MPI_Finalize();
63      return 0;
64  }
```

Lab: MPI_Cart_shift Example

```
ai@ubuntu-20-04:~/Lab/MPI$ mpic++ mpi_cart_shift.cpp -o mpi_cart_shift
ai@ubuntu-20-04:~/Lab/MPI$ mpirun -np 12 mpi_cart_shift
rank:6> torus_rank=6, coords=(1,2)
rank:8> torus_rank=8, coords=(2,0)
rank:10> torus_rank=10, coords=(2,2)
rank:0> torus_rank=0, coords=(0,0)
rank:2> torus_rank=2, coords=(0,2)
rank:4> torus_rank=4, coords=(1,0)
MPI rank=10 I have a DOWN neighbour rank=2
MPI rank=10 I have a UP neighbour rank=6
MPI rank=10 I have a LEFT neighbour rank=9
MPI rank=10 I have a RIGHT neighbour rank=11
MPI rank=0 I have a DOWN neighbour rank=4
MPI rank=0 I have a UP neighbour rank=8
MPI rank=0 I have a LEFT neighbour rank=3
MPI rank=0 I have a RIGHT neighbour rank=1
MPI rank=8 I have a DOWN neighbour rank=0
MPI rank=8 I have a UP neighbour rank=4
MPI rank=8 I have a LEFT neighbour rank=11
MPI rank=8 I have a RIGHT neighbour rank=9
MPI rank=4 I have a DOWN neighbour rank=8
MPI rank=4 I have a UP neighbour rank=0
MPI rank=4 I have a LEFT neighbour rank=7
MPI rank=4 I have a RIGHT neighbour rank=5
rank:3> torus_rank=3, coords=(0,3)
MPI rank=3 I have a DOWN neighbour rank=7
MPI rank=3 I have a UP neighbour rank=11
MPI rank=3 I have a LEFT neighbour rank=2
MPI rank=3 I have a RIGHT neighbour rank=0
```

Matrix-Matrix Multiplication

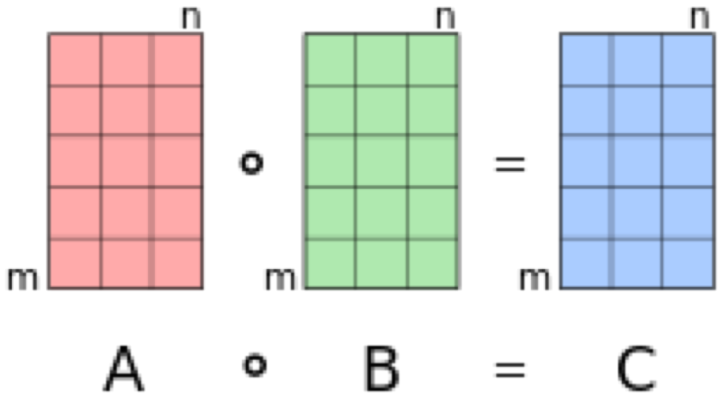
There are two types of matrix multiplication operations:

- Hadamard (element-wise) multiplication $C = A. * B$
- Matrix-Matrix Multiplication

Hadamard (element-wise) Multiplication

- For two matrices *A* and B of the same dimension $m \times n$, the Hadamard product $A \circ B$ is a matrix of the same dimension as the operands, with elements given by

$$(A \circ B)_{i,j} = (A)_{i,j}(B)_{i,j}$$



Application: The Hadamard product appears in lossy compression algorithms such as JPEG. The decoding step involves an entry-for-entry product, in other words the Hadamard product.

https://en.wikipedia.org/wiki/Hadamard_product_(matrices)

Introduction

Definition A matrix is a rectangular array of numbers.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & a_{ij} & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

element in ith row, jth column

Also written as $A=[a_{ij}]$

m rows

$m \times n$ matrix

n columns

When $m=n$, A is called a square matrix.

http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Introduction

Definition A matrix is a rectangular array of numbers.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & a_{ij} & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

element in ith row, jth column

Also written as $A=[a_{ij}]$

m rows

$m \times n$ matrix

n columns

When $m=n$, A is called a square matrix.

http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Matrix Multiplication

Let A be a $m \times k$ matrix, and B be a $k \times n$ matrix,

$$AB = \lfloor c_{ij} \rfloor$$

$$c_{ij} = \sum_{t=1}^{k} a_{it} b_{tj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \ldots + a_{ik} b_{kj}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \end{bmatrix}$$

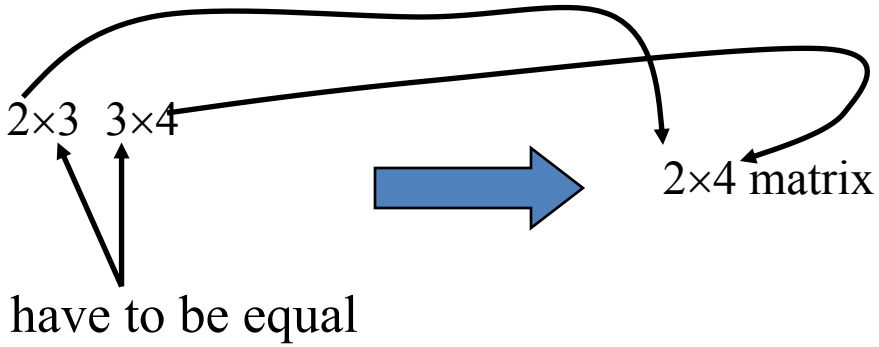
$$a_{11} b_{12} + a_{12} b_{22} + a_{13} b_{32} = c_{12}$$

http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Matching Dimensions

To multiply two matrices, inner numbers must match:

Otherwise,
not defined.



$$\underset{2\times 3}{\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}} \underset{3\times 4}{\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix}} = \underset{2\times 4}{\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \end{bmatrix}}$$

http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Multiplicative Properties

Note that just because AB is defined, BA may not be.

Example If A is 3×4, B is 4×6, then AB=3×6, but BA is not defined (4×6 . 3×4).

Even if both AB and BA are defined, they may not have the same size. Even if they do, matrices do not commute.

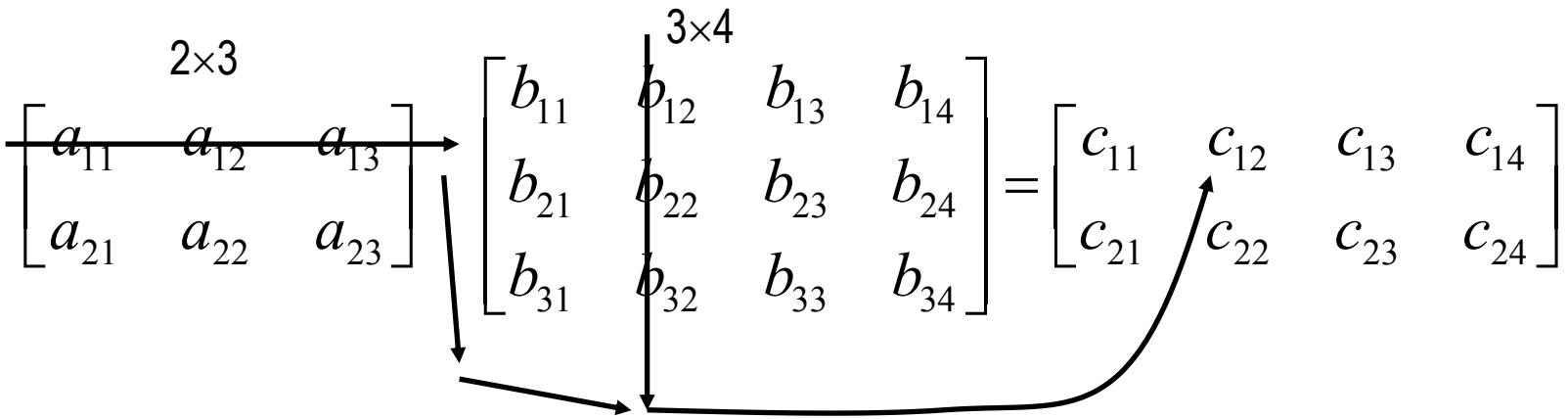
$$A = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

$$AB = \begin{bmatrix} 3 & 2 \\ 5 & 3 \end{bmatrix} \quad BA = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}$$

however $(AB)C = A(BC)$, assuming dimensions match

http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Efficiency of Multiplication



$a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} = c_{12}$

Takes 3 multiplications, and 2 additions for each element.

This has to be done 2×4 (=8) times (since product matrix is 2×4). So 2×4×3 multiplications are needed.

•$(m \times k)$ $(k \times n)$ matrix product requires $m.k.n$ multiplications.

http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Best Order and Efficiency?

Let A be a 20×30 matrix, B 30×40, C 40×10. $(AB)C$ or $A(BC)$?

$(20\times30 \;\; 30\times40) \;\; 40\times10$

$20 \cdot 40 \cdot 10 = 8000$ operations

$20 \cdot 30 \cdot 40 = 24000$ operations

} 32000

$20\times30 \;\; (30\times40 \;\; 40\times10)$

$30 \cdot 40 \cdot 10 = 12000$ operations

$20 \cdot 30 \cdot 10 = 6000$ operations

} 18000

So, $A(BC)$ is best in this case.

http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Identity Matrix

The identity matrix has 1's down the diagonal, e.g.:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For a $m \times n$ matrix A, $I_m A = A I_n$

$m \times m \; m \times n = m \times n \; n \times n$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} = \begin{bmatrix} a+0+0 & b+0+0 \\ 0+c+0 & 0+d+0 \\ 0+0+e & 0+0+f \end{bmatrix}$$

Inverse Matrix

Let A and B be $n \times n$ matrices.
If $AB=BA=I_n$ then B is called the inverse of A, denoted $B=A^{-1}$.

Not all square matrices are invertible.

Use of Inverse to Solve Equations

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$\begin{bmatrix} a_1x + a_2y + a_3z \\ b_1x + b_2y + b_3z \\ c_1x + c_2y + c_3z \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a^{-1}_1 & a^{-1}_2 & a^{-1}_3 \\ b^{-1}_1 & b^{-1}_2 & b^{-1}_3 \\ c^{-1}_1 & c^{-1}_2 & c^{-1}_3 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

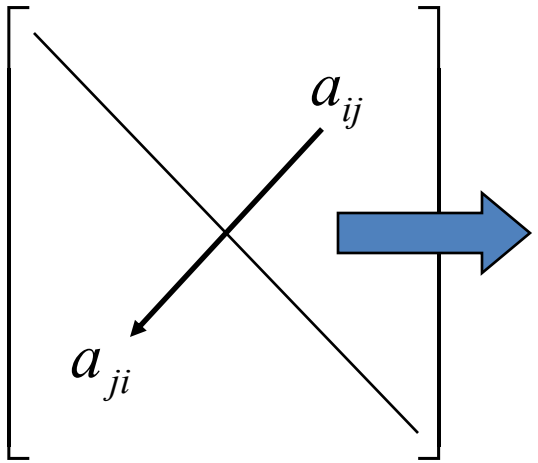
$$AX = K$$

$$A^{-1}AX = A^{-1}K$$

$$IX = A^{-1}K$$

Please note that a^{-1}_j is NOT necessarily $(a_j)^{-1}$.

Transposes of Matrices



Flip across diagonal

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

written A^t

Transposes are used frequently in various algorithms.

Symmetric Matrix

If $A^t = A$ A is called symmetric.

$$\begin{bmatrix} 1 & 4 & -1 \\ 4 & 3 & 0 \\ -1 & 0 & 2 \end{bmatrix}$$

is symmetric. Note, for A to be symmetric, is has to be square.

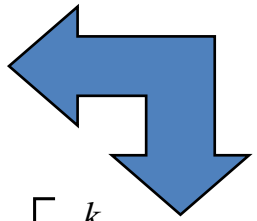
I_n is trivially symmetric...

Examples

$$(AB)^t = B^t A^t$$

$$AB = [c_{ij}] = \left[\sum_{x=1}^{k} a_{ix} b_{xj}\right]$$

$$(AB)^t = [c_{ij}]^t = [c_{ji}] = \left[\sum_{x=}^{k} a_{jx} b_{xi}\right]$$



$$B^t A^t = \left[\sum_{x=1}^{k} b^t{}_{ix} a^t{}_{xj}\right] = \left[\sum_{x=1}^{k} b_{xi} a_{jx}\right] = \left[\sum_{x=1}^{k} a_{jx} b_{xi}\right]$$

Power Matrix

- For a $n \times n$ square matrix A, the power matrix is defined as:

$$A^r = \underbrace{A \cdot A \cdot \ldots \cdot A}_{r \text{ times}}$$

- A^0 is defined as I_n.

Zero-one Matrices

- All entries are 0 or 1.
- Operations are $\wedge$ and $\vee$.
- Boolean product is defined using:

$\wedge$ for multiplication, and

$\vee$ for addition.

Boolean Operations

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

$A \wedge B$ called "meet" $A \wedge B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$

$A \vee B$ called "join" $A \vee B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$

Terminology is from Boolean Algebra.Think
“join” is “put together”, like union, and
“meet” is “where they meet”, or intersect.

Boolean Product

$A = \lfloor a_{ij} \rfloor$ be $m \times k$, and $B = \lfloor b_{ij} \rfloor$ be $k \times n$

$A \otimes B = \lfloor c_{ij} \rfloor, \; c_{ij} = (a_{i1} \wedge b_{1j}) \vee (a_{i2} \wedge b_{2j}) \vee \cdots \vee (a_{ik} \wedge b_{kj})$

(Should be a 'dot')

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}, \; B = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

Since this is "or'd", you can stop when you find a '1'

$$A \otimes B = \begin{bmatrix} (1 \wedge 1) \vee (0 \wedge 0) & 1 & 0 \\ (0 \wedge 1) \vee (1 \wedge 0) & 1 & 1 \\ (1 \wedge 1) \vee (0 \vee 0) & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Boolean Product Properties

- In general, $A \otimes B \neq B \otimes A$
- Example

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$A \otimes B = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad B \otimes A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Matrix-Matrix multiplication

Multiplying two square matrices:
$C \leftarrow A \times B, \qquad A, B, C \in \Re^{n \times n}$
can be achieved by calculating each element of the result matrix as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj},$$

where a_{ij}, b_{ij} and c_{ij} are the element on i^{th} row and j^{th} column of A, B and C respectively.

Pseudocode:

```
1:  Procedure MAT-MULT (A, B, C)
2:  begin
3:      for i ← 0 to n − 1 do
4:          for j ← 0 to n − 1 do
5:              begin
6:                  c_ij = 0
6:                  for k ← 0 to n − 1 do
7:                      c_ij += a_ik · b_kj
8:              endfor
9:  end MAT-MULT
```

Since we assume that floating point multiplications take one time unit, and that floating point additions are very fast (they take zero time units), the time needed for the serial algorithm to complete is expressed as $\Theta(n^3)$.

Traditional Matrix Multiplication (C format)

```
double sequentialMultiply(TYPE** matrixA, TYPE** matrixB, TYPE** matrixC, int dimension){

        struct timeval t0, t1;
        gettimeofday(&t0, 0);

        for(int i=0; i<dimension; i++){
                for(int j=0; j<dimension; j++){
                        for(int k=0; k<dimension; k++){
                                matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
                        }
                }
        }


        double elapsed = (t1.tv_sec-t0.tv_sec) * 1.0f + (t1.tv_usec - t0.tv_usec) / 1000000.0f;

        return elapsed;
}
```

https://medium.com/tech-vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27

Parallel Matrix Multiplication (C format)

```
double sequentialMultiply(TYPE** matrixA, TYPE** matrixB, TYPE** matrixC, int dimension){

        struct timeval t0, t1;
        gettimeofday(&t0, 0);

        #pragma omp parallel for

        for(int i=0; i<dimension; i++){
                for(int j=0; j<dimension; j++){
                        for(int k=0; k<dimension; k++){
                                matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
                        }
                }
        }


        double elapsed = (t1.tv_sec-t0.tv_sec) * 1.0f + (t1.tv_usec - t0.tv_usec) / 1000000.0f;

        return elapsed;
}
```

https://medium.com/tech-vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27

Optimized Parallel Matrix Multiplication (C format)

Steps of optimized matrix multiplication implementation is given below.

- Put common calculation at one place
- Cache friendly algorithm implementation
- Using Stack Vs Heap Memory efficiently

- Very well described in https://medium.com/tech-vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27

OpenMP (40 threads)

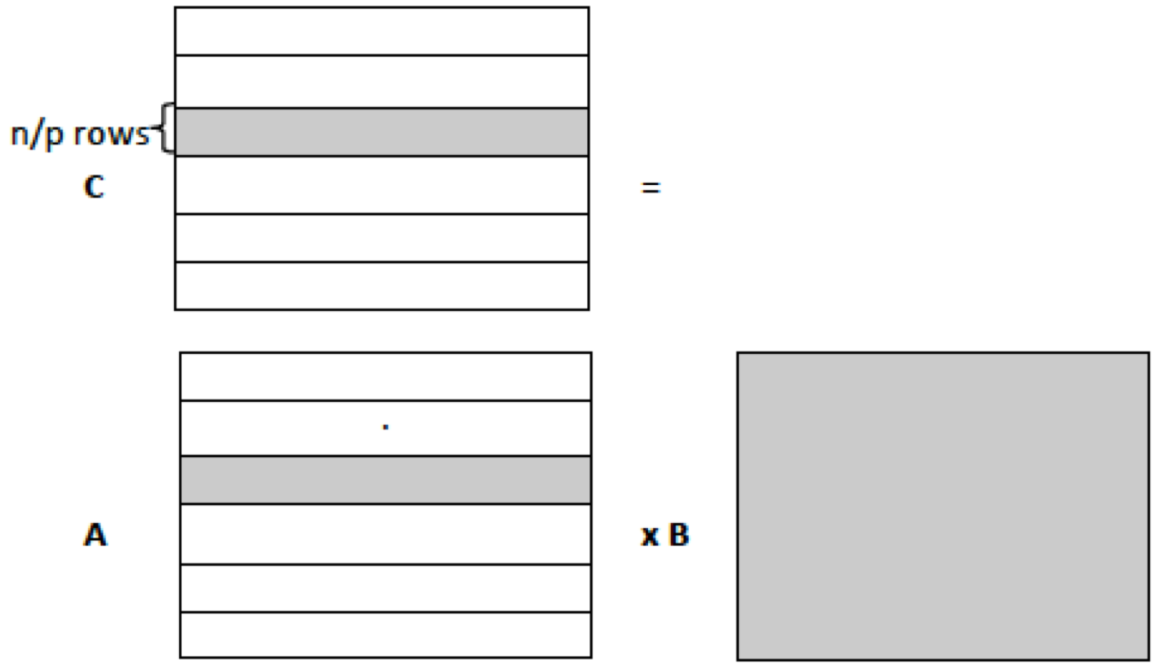
Dimension of an input square matrix	Traditional matrix multiplication	Matrix multiplication using parallel for loops	Optimized matrix multiplication using parallel for loops
200	0.063983	0.008793	0.020379
400	0.6364	0.075818	0.051928
600	3.091816	0.345842	0.120652
800	8.04193	0.911936	0.265509
1000	16.078837	1.818936	0.501389
1200	28.435798	3.102427	0.820317
1400	48.318835	5.01389	1.164099
1600	64.474781	7.91622	1.747289
1800	136.135144	12.260045	2.451863
2000	204.029016	21.258355	3.465287

https://medium.com/tech-vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27

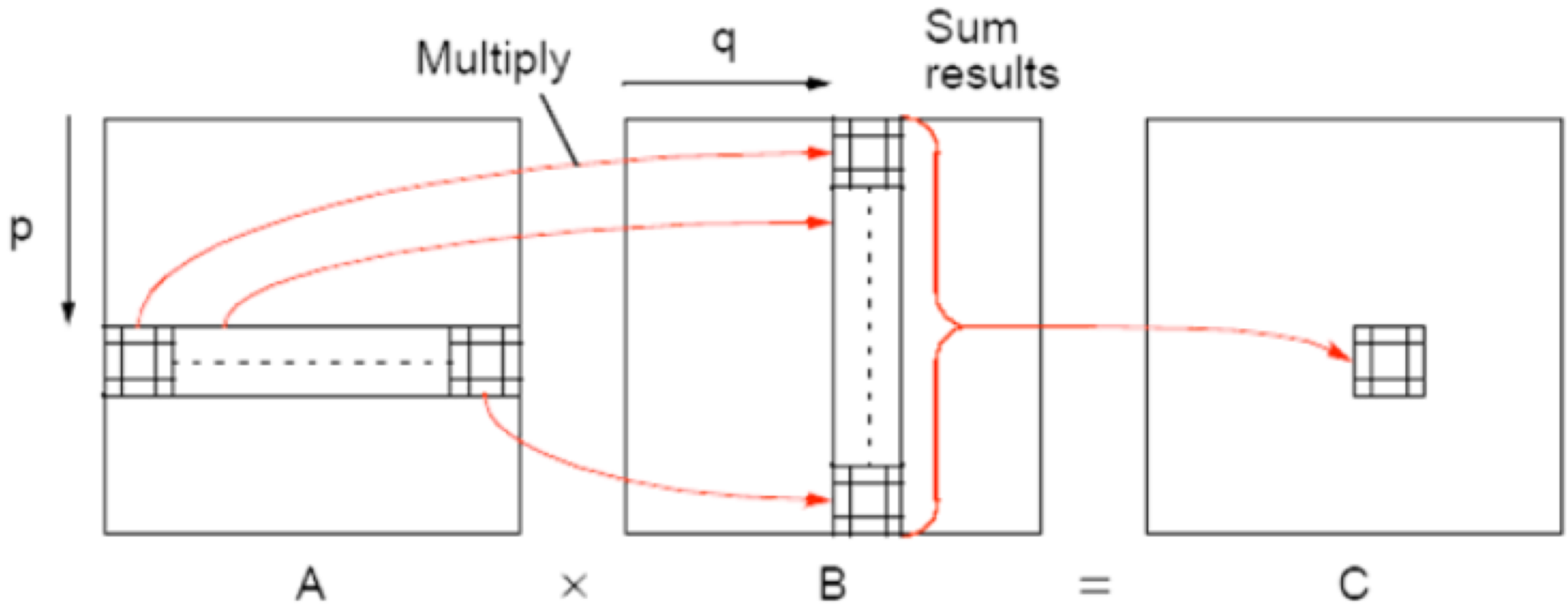
MPI Parallel Matrix Multiplication

Write the parallel matrix multiplication using 1-D partitioning.

- Consider a homogeneous parallel machine with the number of processors p a perfect (i.e., $\sqrt{p}$ is an integer). $P_{i,j}$ is the processor row i and column j where $0 \leq i,j \leq \sqrt{p}-1$.
- We consider only the simple case where the matrix dimension n is a multiple of $\sqrt{p}$ (i.e., $n/\sqrt{p}$ is an integer).
- Implement the program using collective communications such as MPI_Scatter, MPI_Bcast, and MPI_Gather. For example, scatter A matrix and broadcast B matrix to the processes. Compute partial multiplication and gather partial C into C.



Cannon's Algorithm



Cannon's Algorithm

- https://iq.opengenus.org/cannon-algorithm-distributed-matrix-multiplication/
- https://slideplayer.com/slide/9869572/

1: Set $c_{ij} = 0$, for $\forall\ i$ and j
2: **Skew** A: for $i = 0 : (\sqrt{p} - 1)$
3: left circular shift i^{th} sub-block row of A by i
4: **Skew** B: for $j = 0 : (\sqrt{p} - 1)$
5: up circular shift j^{th} sub-block column of B by j
6: for $i = 0 : (\sqrt{p} - 1)$
7: Each processor multuplies the current sub-block of A by the current sub-block of B
8: and adds the result to the sub-block of C of the processor
9: **Roll** A: left circular shift sub-blocks of A by 1
10: **Roll** B: up circular shift sub-blocks of B by 1
11: end

Example: