**Virtual Topologies**

In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric “shape”.

The two main types of topologies supported by MPI are Cartesian (grid) and Graph.

MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.

Virtual topologies are built upon MPI communicators and groups.

They must be “programmed” by the application developer.

**Why Use Them?**

**Convenience**

Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure. For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.

**Communication Efficiency**

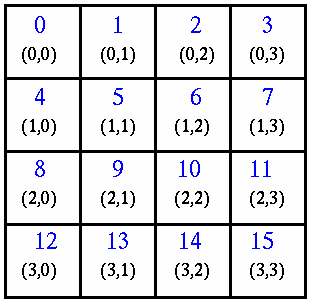
Some hardware architectures may impose penalties for communications between successively distant “nodes”.

A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.

The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

**Example:**

A simplified mapping of processes into a Cartesian virtual topology appears below:



**C Language - Cartesian Virtual Topology Example**

Create a 4 x 4 Cartesian topology from 16 processors and have each process exchange its rank with four neighbors

#include "mpi.h"

#include <stdio.h>

#define SIZE 16

#define UP 0

#define DOWN 1

#define LEFT 2

#define RIGHT 3

int main(int argc, char \*argv[]) {

int numtasks, rank, source, dest, outbuf, i, tag=1,

inbuf[4]={MPI\_PROC\_NULL,MPI\_PROC\_NULL,MPI\_PROC\_NULL,MPI\_PROC\_NULL,},

nbrs[4], dims[2]={4,4},

periods[2]={0,0}, reorder=0, coords[2];

MPI\_Request reqs[8];

MPI\_Status stats[8];

MPI\_Comm cartcomm; // required variable

MPI\_Init(&argc,&argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numtasks);

if (numtasks == SIZE) {

// create cartesian virtual topology, get rank, coordinates, neighbor ranks

MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dims, periods, reorder, &cartcomm);

MPI\_Comm\_rank(cartcomm, &rank);

MPI\_Cart\_coords(cartcomm, rank, 2, coords);

MPI\_Cart\_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);

MPI\_Cart\_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

printf("rank= %d coords= %d %d neighbors(u,d,l,r)= %d %d %d %d\n",

rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],

nbrs[RIGHT]);

outbuf = rank;

// exchange data (rank) with 4 neighbors

for (i=0; i<4; i++) {

dest = nbrs[i];

source = nbrs[i];

MPI\_Isend(&outbuf, 1, MPI\_INT, dest, tag,

MPI\_COMM\_WORLD, &reqs[i]);

MPI\_Irecv(&inbuf[i], 1, MPI\_INT, source, tag,

MPI\_COMM\_WORLD, &reqs[i+4]);

}

MPI\_Waitall(8, reqs, stats);

printf("rank= %d inbuf(u,d,l,r)= %d %d %d %d\n",

rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]); }

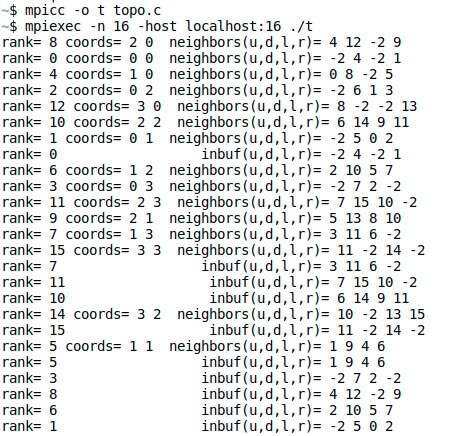
else

printf("Must specify %d processors. Terminating.\n",SIZE);

MPI\_Finalize();

}

**program output:-**

****