MATH 700: Introduction to Parallel Computing Using MPI

Madhu Nayakkankuppam

Author address:

Department of Mathematics & Statistics, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250–0001.

E-mail address: madhu@math.umbc.edu

Contents

Part 1

Overview of Parallel Computing

Part 2

Basic MPI

Part 3

Advanced MPI

LECTURE 1

Derived Types

Topics: In this lecture,

- General type constructor: `MPI_Type_struct` and `MPI_Type_commit`
- Special type constructors: `MPI_Type_vector`, `MPI_Type_contiguous`, `MPI_Type_indexed`
- Type matching
- Compacting data: `MPI_Pack`, `MPI_Unpack`

Reference: Textbook Ch.6.

1.1. Derived types

In order to facilitate the transfer of multiple pieces of data (such as structures), MPI provides a mechanism for user–defined types.

Both intrinsic (*e.g.* `MPI_FLOAT`, `MPI_CHAR` *etc.*) and derived datatypes in MPI can be viewed as a sequence of pairs

$$\{(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1})\}$$

where each t_i is an intrinsic MPI datatype and each d_i is a displacement (usually in bytes).

To build a derived type for broadcasting three integers, we could proceed as follows:

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char **argv) {
  int p, id;
  int a, b, c;
  int bl[3] = {1,1,1};
  MPI_Datatype tl[3] = {MPI_INT, MPI_INT, MPI_INT};
  MPI_Aint dl[3], saddr, addr;
  MPI_Datatype three_ints;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  a = id;
  b = 2*id;
  c = 3*id;

  MPI_Address(&a, &saddr);
  dl[0] = 0;

  MPI_Address(&b, &addr);
  dl[1] = addr - saddr;

  MPI_Address(&c, &addr);
  dl[2] = addr - saddr;
```

```
  MPI_Type_struct(3, bl, dl, tl, &three_ints);
  MPI_Type_commit(&three_ints);


  MPI_Bcast(&a, 1, three_ints, 1, MPI_COMM_WORLD);
  printf("p%d: a = %d\t b=%d\t c=%d\n", id, a, b, c);

  MPI_Finalize();
  return(0);
}
```

Although this example does not fully illustrate the power of derived types, it does introduce several new MPI features:

MPI_Aint: An MPI datatype for addresses, specially designed to handle cases where addresses don't fit in 4 bytes, which is the usual size in C. It is bad (but unfortunately common) practice in C to perform integer arithmetic on pointers.

MPI_Address: An MPI function whose prototype is

```
MPI_Address(
   void *var,         /* in: variable */
   MPI_Aint *address  /* out: address of variable */
);
```

Returns the address of a variable.

MPI_Type_struct: An MPI "type constructor" function whose prototype is

```
int MPI_Type_struct(
   int count,               /* in: number of blocks of elements */
   int *bl,                 /* in: array of block lengths */
   MPI_Aint *dl,            /* in: array of displacements */
   MPI_Datatype *tl,        /* in: array of types */
   MPI_Datatype *new_mpi_t  /* out: derived datatype */
);
```

Creates a derived datatype with the specified sequence of types, block lengths and displacements. The parameter `count` refers to the lengths of the `bl`, `dl` and `tl` arrays. Note that the actual starting address of the message is *not* a part of the datatype. This ensures that the datatype can be used for similar structures that occur frequently in a program. Note also that more complex derived datatypes can be built by calling this function with the `tl` array containing derived datatypes.

This function only creates a data structure representing the derived datatype, which is not ready for use until *committed* (see next).

MPI_Type_commit: MPI function whose prototype is

```
int MPI_Type_commit(
   MPI_Datatype *new_mpi_t   /* in/out: type to be installed */
);
```

This function has to be called before a derived datatype can be used to allow MPI to make internal optimizations to improve communication of objects of this type. In the case of a complex datatype built out of derived datatypes, note that only the final complex datatype needs to be committed, and not the intermediate datatypes which serve as building blocks.

When to use a derived datatype? When a fixed data structure that needs to be communicated occurs repeatedly in a proram, it is worthwhile to incur the overhead of building a dedicated datatype for it.

1.2. Special type constructors

The `MPI_Type_struct` is the most general type constructor in MPI. However, since vectors/matrices and data structures with multiple entries of same type occur so frequently in parallel applications, MPI provides specially tailored versions of `MPI_Type_struct`.

MPI_Type_vector: An MPI function whose prototype is

```
int MPI_Type_vector(
   int count,               /* in: # elements in new type */
   int bl,                  /* in: # entries in each element */
   int stride,              /* in: # elements between
                             *       successive elements
                             *       of new type */
   MPI_Datatype el_type,    /* in: type of each element */
   MPI_Datatype *new_mpi_t  /* out: new datatype */
);
```

Note the differences with `MPI_Type_struct`: `bl` and `el_type` are *not* arrays. Also, the `stride` parameter is a single number.

MPI_Type_contiguous: An MPI function whose prototype is

```
int MPI_Type_contiguous(
   int count,               /* in: # elements in new type */
   MPI_Datatype old_type,   /* in: type of each element */
   MPI_Datatype *new_mpi_t  /* out: new datatype */
);
```

Self–explanatory!

MPI_Type_indexed: An MPI function whose prototype is

```
int MPI_Type_indexed(
   int count,              /* in: # elements */
   int *bl,                /* in: array of block lengths */
   int *dl,                /* in: array of displacements */
   MPI_Datatype old_type,  /* in: old type */
   MPI_Datatype new_mpi_t  /* out: new datatype */
);
```

This is different from `MPI_Type_struct` in that all elements have to be of the *same type*, although they are are allowed to be located at arbitrary positions in memory.

1.3. Type matching

A typical `Send/Recv` of a derived datatype is as follows:

```
if (my_rank == 0) { /* root process */
   dest = 1;
   MPI_Send(mesg, send_count, send_mpi_t, dest, tag,
   MPI_COMM_WORLD);
} else if (my_rank == 1) {
   src = 0;
   MPI_Recv(mesg, recv_count, recv_mpi_t, src, tag,
   MPI_COMM_WORLD, &status);
}
```

What if `send_mpi_t` and `recv_mpi_t` are different? What if `send_count` and `recv_count` are different? These issues are resolved by *type matching*.

The *type signature* of a datatype $\{(t_0, d_0), \ldots, (t_{n-1}, d_{n-1})$ is simply the sequence $(t_0, \ldots, t_{n-1})$. The type signature of a *message* with `count` elements of a given datatype is simply the type signature of the given datatype concatenated `count` times.

Given these definitions for type signatures, the type matching rule in MPI for non–collective communication functions (such as `Send` and `Recv`) is as follows. If $(t_0, \ldots, t_{n-1})$ is the type signature of the sent message and $(u_0, \ldots, u_{m-1})$ is that of the received message, then we must ensure that

- $n \leq m$, and
- $u_i = t_i \quad (i = 0, \ldots, n-1)$.

Note, however, that for collective communication calls (such as `MPI_Bcast`), the type signatures specified by *all* processes must match *exactly*.

1.4. Packing/Unpacking data

Another mechanism to communicate multiple pieces of data is to pack data into one contiguous segment, bypassing the explicit construction of derived types. At the sending end, we have the MPI function

```
MPI_Pack(
void *pack_data,        /* in: data to be packed */
int in_count,           /* in: # elements to be packed */
MPI_Datatype datatype,  /* in: type of elements */
void *buffer,           /* out: packed data */
int buffer_size,        /* in: size of buffer in bytes */
int *position,          /* in: offset in buffer
                         * out: first free location in buffer */
MPI_Comm comm           /* in: communicator */
);
```

First, note that all elements must have the same type. The array `buffer` must be at least as large as the number of bytes to be packed. The first call to this function copies `in_count` elements from `pack_data` at the address starting at `buffer \ *position+`. On return, `position` contains the first location in `buffer` after the copied data.

At the receiving end, we have the MPI function

```
MPI_Unpack(
void *buffer,           /* in: packed data */
int size,               /* in: size of buffer in bytes */
int *position,          /* in: offset into buffer
                         * out: first location after copied data */
void *unpack_data,      /* out: unpacked data */
int count,              /* in: # elements to be unpacked */
MPI_Datatype datatype,  /* in: type of elements */
MPI_Comm comm           /* in: communicator */
);
```

On the first call, the data starting from `buffer \ *position+` is copied into the memory segment with starting address `unpack_data`. On return, `*position` contains the first location in `buffer` after the copied data.

When communicating packed data, use the type `MPI_PACKED`.

In general, packed data should be used for communicating a data structure

- that occurs so infrequently that the overhead in the construction of a derived type (for instance, via `MPI_Type_indexed`) is not justified, or
- that has variable length, so a single derived type cannot be used for communicating all instances of the data structure (*e.g.* sparse matrices of fixed dimension, but variable number of nonzeros).

LECTURE 2

Communicators

Topics: In this lecture,

- Constructing new communicators
- `MPI_Comm_create` and `MPI_Comm_split`

Reference: Textbook Ch.7.

2.1. Communicators

So far, we have always used `MPI_COMM_WORLD` as our universe of communicating processes. In many applications, it's natural for communication to repeatedly occur within a subgroup of the processes in `MPI_COMM_WORLD`. For example, consider the following numerical approximations typical in gradient calculations for $f : \mathbb{R}^2 \to \mathbb{R}$:

$$\nabla_x f(x_0, y_0) \approx \frac{f(x_0 + h, y_0) - f(x_0, y_0)}{h} \qquad \text{(right–sided)}$$

$$\nabla_x f(x_0, y_0) \approx \frac{f(x_0, y_0) - f(x_0 - h, y_0)}{h} \qquad \text{(left–sided)}$$

$$\nabla_x f(x_0, y_0) \approx \frac{f(x_0 + h, y_0) - f(x_0 - h, y_0)}{2h} \qquad \text{(two–sided)}$$

If we map a rectangular mesh of grid points in the domain on to a (virtual) grid of processes, we see that communication occurs between processes of the same row. Similar expressions for $\nabla_y f(x_0, y_0)$ will show that communication occurs within processes of each column. Although it is possible to implement these communications naively with the usual MPI functions we have used so far, it is definitely convenient to be able to define subgroups of communicating processes which can take part in both point–to–point as well as collective communication.

MPI provides a mechanism to create new communicators. New user–defined communicators may be created in one of two broad classes:

Intra–communicators: Communication (point–to–point and collective) occurs between processes of a newly created communicating class.

Inter–communicators: Communication can occur between processes of two different communicating classes.

The second category is quite a complicated issue; we will focus on the first kind. Hereafter, our usage of *communicator* refers to *intra–communicator*.

2.2. Creating New Communicators

Note that a communicator is merely a class of communicating processes, so a particular process can participate in different communicators. Abstractly, a communicator is indentified by a

Group: An array of process ranks. (In `MPI_COMM_WORLD`, these ranks are $0, \ldots, p-1$, where p is number of processes.)

Context: This is a unique ID that MPI uses internally for identifying communicators. Two distinct communicators will have distinct ID's even if they comprise the same group of processes. The same process could have different ranks in different communicators. Since we are dealing with intra–communicators, all processes in a given communicator must use the same communicator in their MPI function calls. The context is a quick way for MPI to verify that this is the case.

The actual implementation of groups and contexts is opaque to the user, who has no need to explicitly manipulate them.

Let us try to construct a new communicator consisting of the first row of processes in our virtual grid.

```
/* comm_create.c -- builds a communicator from the first q processes
 *     in a communicator containing p = q^2 processes.
 *
 * Input: none
 * Output: q -- program tests correct creation of new communicator
 *     by broadcasting the value 1 to its members -- all other
 *     processes have the value 0 -- global sum computed across
 *     all the processes.
 *
 * Note:  Assumes that MPI_COMM_WORLD contains p = q^2 processes
 *
 * See Chap 7, pp. 117 & ff in PPMPI
 */
#include <stdio.h>
#include "mpi.h"
#include <math.h>
#include <stdlib.h>

main(int argc, char* argv[]) {
    int        p;
    int        q; /* = sqrt(p) */
    int        my_rank;
    MPI_Group  group_world;
    MPI_Group  first_row_group;
    MPI_Comm   first_row_comm;
    int*       process_ranks;
    int        proc;
    int        test = 0;
    int        sum;
    int        my_rank_in_first_row;


    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    q = (int) sqrt((double) p);

    /* Make a list of the processes in the new
     * communicator */
    process_ranks = (int*) malloc(q*sizeof(int));
    for (proc = 0; proc < q; proc++)
        process_ranks[proc] = proc;

    /* Get the group underlying MPI_COMM_WORLD */
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);

    /* Create the new group */
    MPI_Group_incl(group_world, q, process_ranks,
        &first_row_group);
```

```
    /* Create the new communicator */
    MPI_Comm_create(MPI_COMM_WORLD, first_row_group,
        &first_row_comm);

    /* Now check whether we can do collective ops in first_row_comm */
    if (my_rank < q) {
        MPI_Comm_rank(first_row_comm, &my_rank_in_first_row);
        if (my_rank_in_first_row == 0) test = 1;
        MPI_Bcast(&test, 1, MPI_INT, 0, first_row_comm);
    }
    MPI_Reduce(&test, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (my_rank == 0) {
        printf("q = %d, sum = %d\n", q, sum);
    }

    MPI_Finalize();
}  /* main */
```

`MPI_Group` is an opaque, internal type and

```
MPI_Group(
  MPI_Comm *comm,    /* in */
  MPI_Group *group   /* out */
);
```

returns the group associated with `comm` in `group`. A call to

```
MPI_Group_incl(
  MPI_Group old_group,  /* in */
  int q,                /* in */
  int *process_ranks,   /* in */
  MPI_Group *new_group  /* out */
);
```

constructs a new group called `new_group` for the `q` processes whose ranks are in the array `process_ranks`, whose context in the parent communicator (`MPI_COMM_WORLD` in this case) is in `old_group`. Finally, the new communicator is actually created by calling

```
MPI_Comm_create(
  MPI_Comm old_comm,    /* in */
  MPI_Group new_group,  /* in */
  MPI_Comm *new_comm    /* out */
);
```

which returns a handle on the new communicator in `new_comm`. Note that this function is a collective call made by all processes in `old_comm` (with the same argument passed for `old_comm`), inluding those not being included in the new communicator! This is necessary for MPI to internally update the context.

If we want to simultaneously create a new communicator for each row, modifying the above code is somewhat tedious. MPI provides the collectively–called function

```
MPI_Comm_split(
  MPI_Comm old_comm,   /* in */
  int split_key,       /* in */
  int rank_key,        /* in */
  MPI_Comm *new_comm   /* out */
);
```

which, *for each calling process*, provides a handle `new_comm` into a new communicator containing all processes in `old_comm` that executed this function with the same value of `split_key`. For two processes `p1` and `p2` belonging to the same `new_comm` that called `MPI_Comm_split` with arguments `rank_key1 > rank_key2`, MPI will assign new ranks of these processes in `new_comm` such that the rank of `p1` is greater than the rank of `p2`. If a particular process in `old_comm` is to be excluded from `new_comm`, it should call `MPI_Comm_split` with `split_key = MPI_UNDEFINED`. In this case, the predefined type `MPI_COMM_NULL` will be returned in `new_comm` for this process. Note that although different processes in `new_comm` may receive different literal pointers `new_comm` which point their own copies of a representation of `new_comm`, MPI maintains the context of `new_comm`, so they will all be identified as the same communicator in any subsequent point–to–point or collective communcation.

The following example illustrates how to use `MPI_Comm_split` to simultaneously create a new communicator for each row of our virtual grid.

```
/* comm_split.c -- build a collection of q communicators using MPI_Comm_split
 *
 * Input: none
 * Output:  Results of doing a broadcast across each of the q communicators.
 *
 * Note:  Assumes the number of processes, p = q^2
 *
 * See Chap. 7, pp. 120 & ff in PPMPI
 */
#include <stdio.h>
#include "mpi.h"
#include <math.h>

main(int argc, char* argv[]) {
    int       p;
    int       my_rank;
    MPI_Comm  my_row_comm;
    int       my_row;
    int       q;
    int       test;
    int       my_rank_in_row;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    q = (int) sqrt((double) p);

    /* my_rank is rank in MPI_COMM_WORLD.
     * q*q = p */
    my_row = my_rank/q;
    MPI_Comm_split(MPI_COMM_WORLD, my_row, my_rank,
        &my_row_comm);
```

```
    /* Test the new communicators */
    MPI_Comm_rank(my_row_comm, &my_rank_in_row);
    if (my_rank_in_row == 0)
        test = my_row;
    else
        test = 0;

    MPI_Bcast(&test, 1, MPI_INT, 0, my_row_comm);

    printf("Process %d > my_row = %d, my_rank_in_row = %d, test = %d\n",
        my_rank, my_row, my_rank_in_row, test);

    MPI_Finalize();
}  /* main */
```

LECTURE 3

Topologies

Topics: In this lecture,

- Cartesian grids and tori
- `MPI_Cart_create, MPI_Cart_coords, MPI_Cart_rank` and `MPI_Cart_sub`.
- Fox's algorithm for matrix multiplication.

Reference: Textbook Ch.7.

3.1. Cartesian Grids and Tori

A *topology* is a mapping of processes in a communicator to an addressing scheme. The addressing scheme is usually chosen for convenience; there may be no relation between the addressing scheme and the physial hardware layout of a parallel computer. While any graph topology can be implemented on a set of MPI processes, a special set of functions is available to construct and manipulate Cartesian grids which occur frequently in applications. In the next section, we will see an application of Cartesian grids, but first let us see how to construct a communicator with this topology.

The following code snippet shows how to map q^2 processes onto a $q \times q$ Cartesian grid:

```
MPI_Comm grid_comm;
int dim_sizes[2] = {q, q};
int wrap_around[2] = {1, 1};
int reorder = 1;

MPI_Cart_create(MPI_COMM_WORLD, 2, dim_sizes, wrap_around,
reorder, &grid_comm);
```

The second argument in `MPI_Cart_create` indicates the dimension of the grid (2–D in this case). The array `dim_sizes` contains the size along each dimension. The `wrap_around` array has entries 1 (first process is considered adjacent to the last process along that dimension) or 1 (first process is *not* considered to be adjacent to the last process in that dimension). Setting the parameter `reorder` to 1 allows MPI to relabel the process ranks in `grid_comm` to optimize communication.

MPI provides two addressing functions for Cartesian grids:

```
MPI_Cart_coords(
  MPI_Comm grid_comm,      /* in */
  int my_grid_rank,        /* in */
  int dim,                 /* in */
  int *coords              /* out */
);
```

returns in the array `coords`, the grid location associated with process rank `my_grid_rank` in the `dim`–dimensional grid communicator `grid_comm`.

The inverse of this function is

```
MPI_Cart_rank(
  MPI_Comm grid_comm,    /* in */
  int *coords            /* in */
```

```
  int *grid_rank         /* out */
);
```

which returns the rank of the process at grid location given by `coords` in the address `grid_rank`.

(Note the annoying asymmetry between the argument lists for these two functions. The `dim` parameter is not needed for `MPI_Cart_rank`, presumably because MPI keeps track of it in the context for `grid_comm` at the time the grid was created. Yet, the `dim` parameter has to be passed explicitly for `MPI_Cart_coords`!)

Once a grid has been created, we can subsequently partition it into subgrids of communicating processes using

```
MPI_Cart_sub(
  MPI_Comm grid_comm,  /* in */
  int *free_coords,    /* in */
  MPI_Comm *new_comm   /* out */
);
```

The processes in `grid_comm` are partitioned into *disjoint* communicators with associated Cartesian topologies. Here, the length of the `free_coords` array must equal the number of dimensions of the grid corresponding to `grid_comm`. If `free_coords[i] = 0`, then the resulting `new_comm` contains all those processes of `grid_comm` whose i^{th} coordinate is the same. Thus, if the grid associated with `grid_comm` has dimensions $d_1 \times \ldots d_n$, and if `free_coords[i] = 0` only for $i = i_1, i_2, \ldots, i_k$, then $d_{i_1} d_{i_2} \ldots d_{i_k}$ communicators are simultaneously created.

Like `MPI_Comm_split`, this function is also a collective operation.

3.2. Fox's algorithm

In this section, we will discuss Fox's algorithm for matrix multiplication. Although this may not be the best way to multiply to matrices in parallel, it serves to illustrate the usefulness of Cartesian grid topologies.

Suppose we want to multiply two $n \times n$ matrices A and B. For the sake of illustration, let us assume that we have n^2 processes in a Cartesian grip topology, so we can index a process with a pair (i, j). We assume that arithemtic on indices are modulo n, *e.g.* $i + k$ stands for $(i + k)$ mod n. In the beginning, each processor (i, j) contains a_{ij} and b_{ij}. The multiplication proceeds in n stages as described below.

- Stage 0 on process (i, j): This process multiplies the diagonal entry of A in its process row i with its entry of B, namely b_{ij}:
$$c_{ij} = a_{ii} b_{ij}.$$
- Stage 1 on process (i, j): This process multiplies the element to the right of the diagonal of A in its process row i by the element of B directly beneath its own element of B, namely $b_{i+1,j}$:
$$c_{ij} = c_{ij} + a_{i,i+1} b_{i+1,j}.$$
- $\ldots$ and so on.
- Stage k on process (i, j):
$$c_{ij} = c_{ij} + a_{i,i+k} b_{i+k,j}.$$

Over the course of the n stages, we compute c_{ij} as

$$c_{ij} = a_{ii} b_{ij} + a_{i,i+1} b_{i+1,j} + \ldots + a_{i,n-1} b_{n-1,j} + a_{i0} b_{0j} + \ldots + a_{i,i-1} b_{i-1,j}.$$

At each stage, each of the n^2 processes performs one operation (a multiplication and an addition) amounting to a total of n^3 operations, but in n time steps. To understand how to move data around from Stage k to Stage (k+1), see Figure 7.3 of the textbook. An implementation of the algorithm, along with other code appearing in the book, can be downloaded from the web page for the textbook.

LECTURE 4

Rings and Hypercubes

Topics: In this lecture,

- Special topologies: rings and hypercubes.
- Implementing `MPI_Allgather` on rings and hypercubes.

Reference: Textbook Ch.13.1–13.4.

In the previous lecture, we saw how MPI can partition the universe of processes `MPI_COMM_WORLD` into virtual Cartesian grids, irrespective of the underlying physical connections between the processors. In this lecture, we will see how to implement MPI's communication functions on rings and hypercubes. The specific communication function we use to illustrate the idea is `MPI_Allgather`.

4.1. Rings

Suppose we have p processes $0, \ldots, p-1$ configured as a ring. Perhaps the simplest way to implement `Allgather` is to use a *ring pass*. This proceeds in $p-1$ stages. During Stage k, process i passes to process $i+1$ the data it received from process $i-1$ in Stage $k-1$. (All arithmetic here is modulo p.) This operation clearly takes $O(p)$ time to complete, and is accomplished by the following code snippet. We assume that, in the beginning, each process contains one unit of data, which is a block of floats.

```
void Allgather_ring(
         float     x[]        /* in  */,
         int       blocksize  /* in  */,
         float     y[]        /* out */,
         MPI_Comm  ring_comm  /* in  */) {

    int         i, p, my_rank;
    int         successor, predecessor;
    int         send_offset, recv_offset;
    MPI_Status  status;

    MPI_Comm_size(ring_comm, &p);
    MPI_Comm_rank(ring_comm, &my_rank);

    /* Copy x into correct location in y */
    for (i = 0; i < blocksize; i++)
        y[i + my_rank*blocksize] = x[i];

    successor = (my_rank + 1) % p;
    predecessor = (my_rank - 1 + p) % p;

    for (i = 0; i < p - 1; i++) {
        send_offset = ((my_rank - i + p) % p)*blocksize;
        recv_offset =
            ((my_rank - i - 1 + p) % p)*blocksize;
        MPI_Send(y + send_offset, blocksize, MPI_FLOAT,
            successor, 0, ring_comm);
```

```
        MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
            predecessor, 0, ring_comm, &status);
    }
} /* Allgather_ring */
```

Note that every processor first sends, then receives. Consequently, this function would be unsafe in systems without buffering.

4.2. Hypercubes

A classic way to propagate data in a hypercube is the so–called *hypercube exchange*, which we will now describe. Note that the d–dimensional hypercube consists of 2^d processes, and can be defined recursively by connecting corresponding nodes in two $(n-1)$–dimensional hypercubes. We can address each of these nodes using a bit vector of length d. The addresses are chosen such that two nodes in a hypercube are considered adjacent if the Hamming distance between their addresses is unity. In the beginning, each node has one unit of data. During stage k, adjacent pairs of nodes in the two $(d-1)$–dimensional hypercubes exchange 2^k units of data (see Fig. 13.6 of the textbook).

This operation is completed in $O(d)$ stages, or equivalenty, $O(\log p)$ stages, where $p = 2^d$ is the number of processes. However, note that the amount of data transferred in each stage is not constant. Suppose we want to `Allgather` a single precision number on each node. Initially (Stage 0), each node exchanges one float with a neighbor. In the first stage, each node exchanges two $(= 2^1)$ floats, but spaced 2^{p-1} units apart, with another neighbor. In the second stage, each node exchanges four $(= 2^2)$ floats, but spaced 2^{p-2} units apart, and so on. In general, for transferring arrays, we will have to deal with blocks of floats, for which we can build a derived datatype using, for example, `MPI_Type_vector`.

Addressing nodes in a hypercube is best done with bit operators in the C language. The following code snippet illustrates the basic idea.

```
int log_base2(int p) {
/* Just counts number of bits to right of most significant
 * bit.  So for p not a power of 2, it returns the floor
 * of log_2(p).
 */
    int return_val = 0;
    unsigned q;

    q = (unsigned) p;
    while(q != 1) {
        q = q >> 1;
        return_val++;
    }
    return return_val;
}  /* log_base2 */

/*--------------------------------------------------------------------*/

void  Allgather_cube(
         float    x[]        /* in  */,
         int      blocksize  /* in  */,
         float    y[]        /* out */,
         MPI_Comm comm       /* in  */) {

    int           i, d, p, my_rank;
    unsigned      eor_bit;
    unsigned      and_bits;
```

```
    int           stage, partner;
    MPI_Datatype  hole_type;
    int           send_offset, recv_offset;
    MPI_Status    status;

    int log_base2(int p);

    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &my_rank);

    /* Copy x into correct location in y */
    for (i = 0; i < blocksize; i++)
        y[i + my_rank*blocksize] = x[i];

    /* Set up */
    d = log_base2(p);
    eor_bit = 1 << (d-1);
    and_bits = (1 << d) - 1;

    for (stage = 0; stage < d; stage++) {
        partner = my_rank ^ eor_bit;
        send_offset = (my_rank & and_bits)*blocksize;
        recv_offset = (partner & and_bits)*blocksize;

        MPI_Type_vector(1 << stage, blocksize,
            (1 << (d-stage))*blocksize, MPI_FLOAT,
            &hole_type);
        MPI_Type_commit(&hole_type);

        MPI_Send(y + send_offset, 1, hole_type,
            partner, 0, comm);
        MPI_Recv(y + recv_offset, 1, hole_type,
            partner, 0, comm, &status);

        MPI_Type_free(&hole_type);  /* Free type so we  */
                 /* can build new type during next pass */
        eor_bit = eor_bit >> 1;
        and_bits = and_bits >> 1;
    }
} /* Allgather_cube */
```

Once again, notice that the function is unsafe as it may fail in the absence of buffering. Also, the behavior of `MPI_Type_vector` is not well–defined when `blocksize` is zero, resulting in an invalid datatype.

4.3. Avoiding Deadlocks

In the absence of buffering, an `MPI_Send` will not return until the corresponding `MPI_Recv` has been executed. If all processes are busy in a send operation, we have a *deadlock*.

One way to avoid a deadlock is to re–organize the sends and the receives. For instance, the odd numbered processes could send, while the even numbered processes could receive. (Convince yourself that this scheme works even when there are an odd number of processes.) Alternatively, MPI provides the function

```
int MPI_Sendrecv(
  void *send_buf,            /* in */
```

```
  int send_count,            /* in */
  MPI_Datatype send_type,    /* in */
  int dest,                  /* in */
  int send_tag,              /* in */
  void *recv_buf,            /* out */
  int recv_count,            /* in */
  MPI_Datatype recv_type,    /* in */
  int src,                   /* in */
  int recv_tag,              /* in */
  MPI_Comm comm,             /* in */
  MPI_Status *status         /* out */
);
```

which is a safe implementation of a simultaneous send and receive. The parameters `dest` and `src` can be the same. Moreover, this function call by the sender (receiver) can be matched by a call to `MPI_Recv` (`MPI_Send`) by the receiver (sender). However, `send_buf` and `recv_buf` must be distinct. (If it is convenient for data to be received in the same buffer whose data was sent, recall the function `MPI_Sendrecv_replace` which we encountered earlier. However, a little thought immediately shows that this function implicitly assumes some buffering.)

4.4. Null Processes

To exclude a specific process from a communication, we can test the rank of the process. A more elegant solution that works cleanly even in collective communication is the use of `MPI_PROC_NULL`, a predefined constant. A process executing a send to `MPI_PROC_NULL` will return immediately without any effect. A process executing a receive from `MPI_PROC_NULL` will return immediately without any change to its output buffer. The `status` argument in the receive will have the `count` parameter set to zero and the `MPI_Tag` parameter set to `MPI_ANY_TAG`.