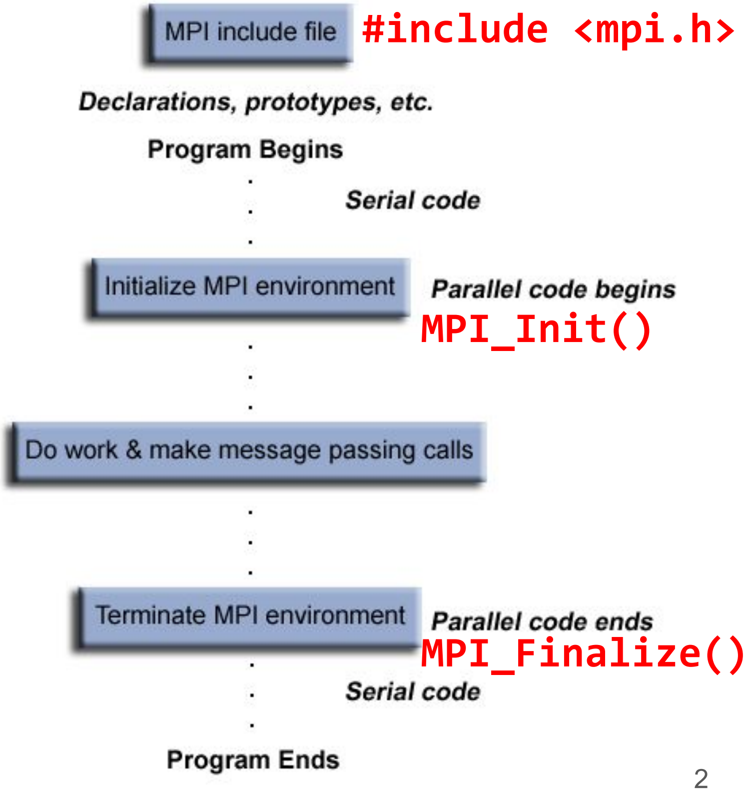
MPI Supplementary

Scott Cheng

MPI: Message Passing Interface

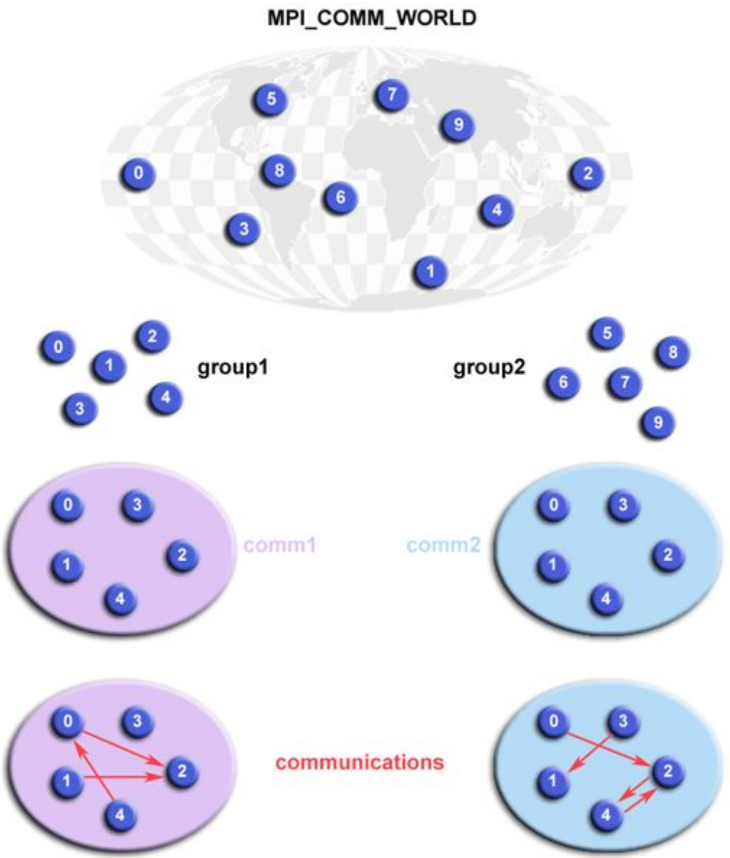
General MPI program structure:

- **Header file**: #include <mpi.h>
  - Required for making *MPI library* call
- **MPI calls**:
  - Format: rc = MPI_xxx(params, ...)
  - Example: rc = MPI_Bcast(&buffer,count,datatype,root,comm)
  - Error code: return as "rc"; rc=MPI_SUCCESS if successful



MPI: Message Passing Interface

- **Communicators and Groups**
  - **Groups** define which **collection of processes** may communicate with each other
  - Each group is associated with a **communicator** to perform its communication function calls
  - `MPI_COMM_WORLD` is the predefined communicator for all processors
- **Rank**
  - An **unique identifier** (task ID) for each process in a communicator
  - **Assigned by the system** when the process initializes
  - Contiguous and **begins at zero**



Environment Management Routines

- **`MPI_Init(int *argc, char **argv)`**
  - Initializes the MPI execution environment
  - Must be called before any other MPI functions
  - Must be called only once in an MPI program
- **`MPI_Finalize()`**
  - Terminates the MPI execution environment
  - No other MPI routines may be called after it
- **`MPI_Comm_size(comm, &size)`**
  - Determines the number of processes in the group associated with a
  - Communicator
- **`MPI_Comm_rank(comm, &rank)`**
  - Determines the rank of the calling process within the communicator

Example

```
#include <mpi.h>

int main(int argc, char *argv[]) {
    int size, rank, rc;
    rc = MPI_Init(&argc, &argv);
    if (rc != MPI_SUCCESS) {
        printf("Error starting MPI program. Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Number of tasks= %d My rank= %d\n", size, rank);
    MPI_Finalize();
}
```

mpirun Examples

- **`mpirun -np 4 -hosts node1,node2 ./prog`**
    - **`np`**: number of processes
    - **`hosts`**: hostnames
- **`mpirun -np 4 -ppn 3 -hosts node1,node2 ./prog`**
    - **`ppn`**: process per node
- **`mpirun -np 4 -hostfile myHostFile ./prog`**
    - **`hostfile`**/**`machinefile`**: hostnames & slots

```
node1 slots=1
node2 slots=3
```

Outline

- **Point-to-Point Communication Routines**
- Collective Communication Routines
- Group and Communicator Management Routines
- MPI-IO

Point-to-Point Communication Routines

Blocking send	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking send	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

- **`buffer`**: address space that references the data to be sent or received
- **`type`**: `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_DOUBLE`, …
- **`count`**: indicates the **number of data elements** of a particular type to be sent or received
- **`comm`**: indicates the communication context
- **`source/dest`**: the **rank** of the sender/receiver
- **`tag`**: arbitrary non-negative integer **assigned by the programmer to uniquely identify a message**. Send and receive operations **must match** message tags. **`MPI_ANY_TAG`** is the wild card.
- **`status`**: status after operation
- **`request`**: used by **non-blocking** send and receive operations

Blocking Example

Blocking send	`MPI_Send(buffer,count,type,dest,tag,comm)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank); /* find process rank */
if (myRank == 0) {
    int x = 10;
    MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (myRank == 1) {
    int z;
    MPI_Recv(&z, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, status);
}
```

Non-Blocking Example

Non-blocking send	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);/* find process rank */
if (myRank == 0) {
    int x = 10;
    MPI_Isend(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, req1);
    compute();
} else if (myRank == 1) {
    int z;
    MPI_Irecv(&z, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, req1);
}
MPI_Wait(req1, status);
```

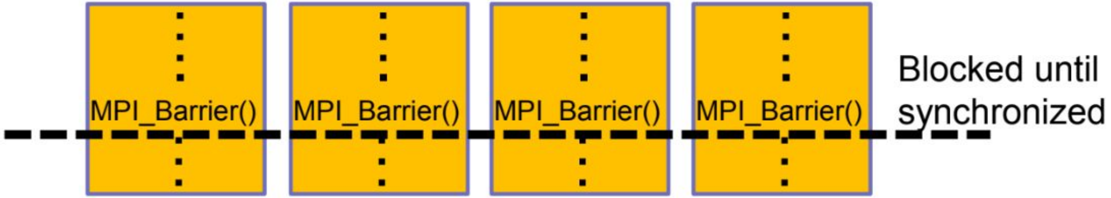
- `MPI_Wait()` **blocks** until the operation has actually **completed**
- `MPI_Test()` returns with a flag set indicating whether operation completed at that time.

Outline

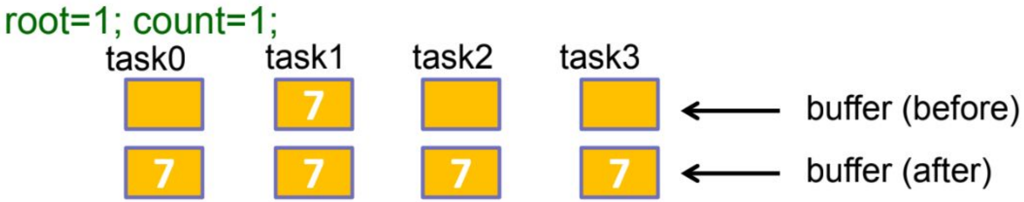
- Point-to-Point Communication Routines
- **Collective Communication Routines**
- Group and Communicator Management Routines
- MPI-IO

Collective Communication Routines

- **`MPI_Barrier(comm)`**
  - Creates a **barrier synchronization** in the group
  - **Blocks** until all tasks in the group reach the same `MPI_Barrier` call



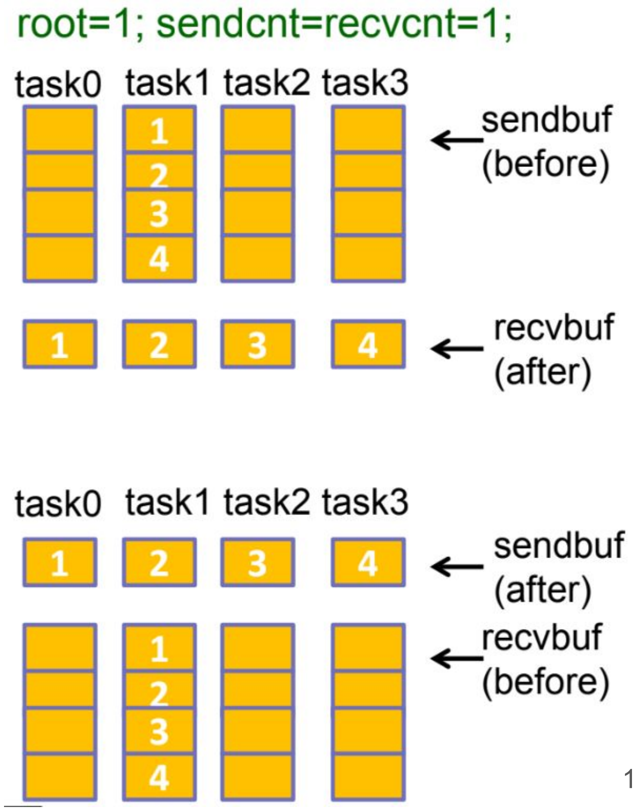
- **`MPI_Bcast(&buffer, count, datatype, root, comm)`**
  - Broadcasts (sends) a message from the process with rank ***`root`*** to all other processes in the group



See also: `MPI_Ibcast`

Collective Communication Routines

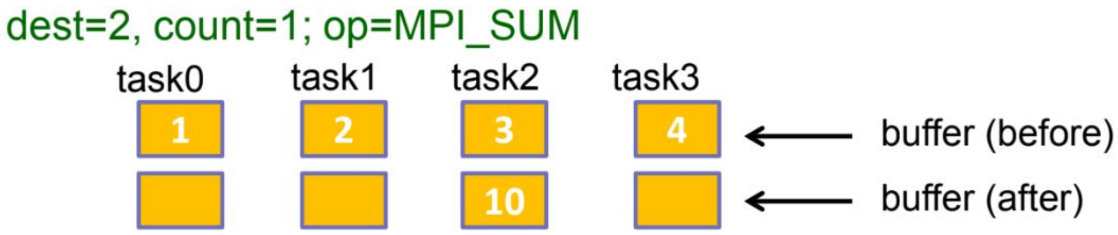
- **`MPI_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)`**
  - Distributes distinct messages from a source rank to all ranks
- **`MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)`**
  - Gathers distinct messages from each rank in the group to a single destination rank
  - This routine is the reverse operation of `MPI_Scatter`



See also: `MPI_Iscatter`, `MPI_Igather`

Collective Communication Routines

- **`MPI_Reduce(&sendbuf,&recvbuf,count,datatype,op,dest,comm)`**
  - Applies a reduction operation on all ranks in the group and places the result in one rank



See also: `MPI_Ireduce`

Predefined Reduction Operations

Operation	Meaning	Datatypes
`MPI_MAX`	Maximum	C integers and floating point
`MPI_MIN`	Minimum	C integers and floating point
`MPI_SUM`	Sum	C integers and floating point
`MPI_PROD`	Product	C integers and floating point
`MPI_LAND`	Logical AND	C integers
`MPI_BAND`	Bit-wise AND	C integers and byte
`MPI_LOR`	Logical OR	C integers
`MPI_BOR`	Bit-wise OR	C integers and byte
`MPI_LXOR`	Logical XOR	C integers
`MPI_BXOR`	Bit-wise XOR	C integers and byte
`MPI_MAXLOC`	max-min value-location	Data-pairs
`MPI_MINLOC`	min-min value-location	Data-pairs

Collective Communication Operations

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i 's).
- `MPI_MINLOC` does the same, except for minimum value of v_i.

Value	15	17	11	12	17	11
Process	0	1	2	3	4	5

```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.

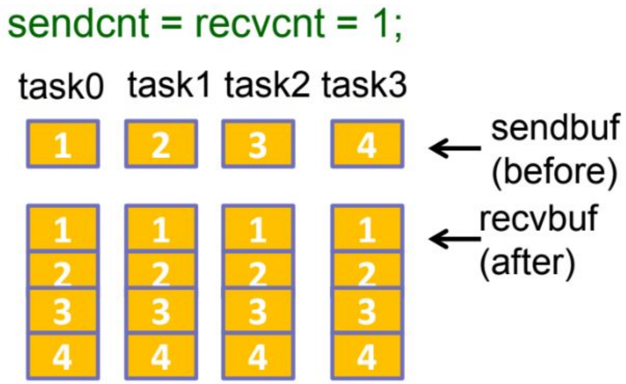
Collective Communication Operations

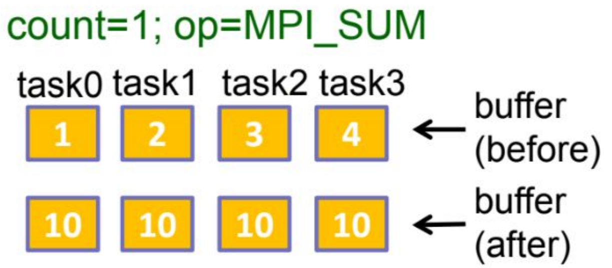
MPI datatypes for data-pairs used with the `MPI_MAXLOC` and `MPI_MINLOC` reduction operations.

MPI Datatype	C Datatype
`MPI_2INT`	pair of ints
`MPI_SHORT_INT`	short and int
`MPI_LONG_INT`	long and int
`MPI_LONG_DOUBLE_INT`	long double and int
`MPI_FLOAT_INT`	float and int
`MPI_DOUBLE_INT`	double and int

Collective Communication Routines

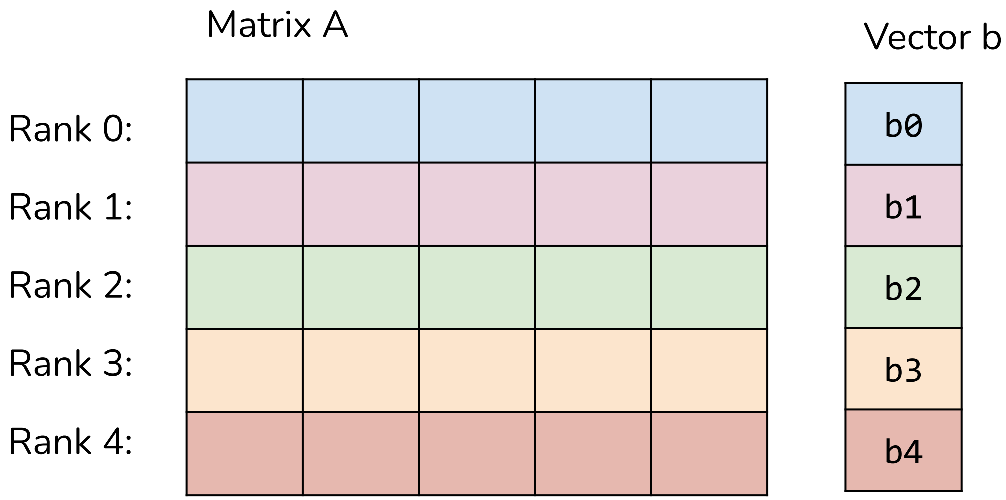
- **`MPI_Allgather(&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)`**
  - Concatenation of data to all ranks
  - This is equivalent to an `MPI_Gather` followed by an `MPI_Bcast`
- **`MPI_Allreduce(&sendbuf, &recvbuf, count, datatype, op, comm)`**
  - Applies a reduction operation and places the result in all ranks
  - This is equivalent to an `MPI_Reduce` followed by an `MPI_Bcast`



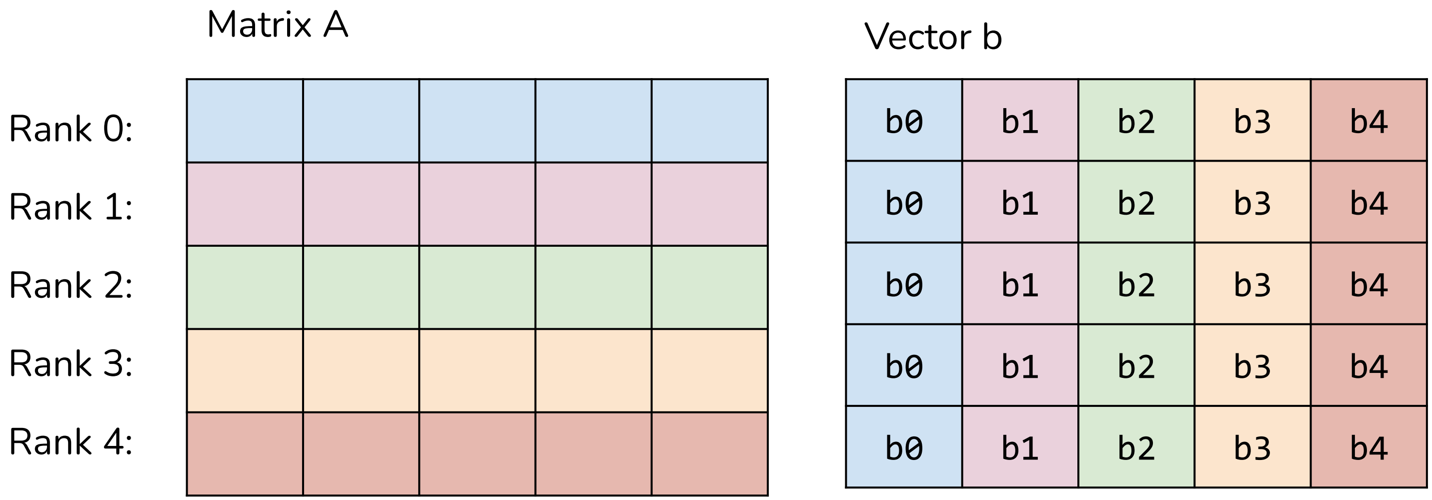


See also: `MPI_Iallgather`, `MPI_Iallreduce`

Example1: Row-major Matrix-Vector Multiplication



Example1: Row-major Matrix-Vector Multiplication



```c
RowMatrixVectorMultiply(int n, double *a, double *b, double *x, MPI_Comm comm) {
    int i, j;
    int nlocal; /* Number of locally stored rows of A */
    double *fb; /* Will point to a buffer that stores the entire vector b */
    int npes, myrank;
    MPI_Status status;

    /* Get information about the communicator */
    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    /* Allocate the memory that will store the entire vector b */
    fb = (double *)malloc(n*sizeof(double));

    nlocal = n/npes;

    /* Gather the entire vector b on each processor using MPI's ALLGATHER operation */
    MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE, comm);

    /* Perform the matrix-vector multiplication involving the locally stored submatrix */
    for (i=0; i<nlocal; i++) {
        x[i] = 0.0;
        for (j=0; j<n; j++)
            x[i] += a[i*n+j]*fb[j];
    }

    free(fb);
}
```

See Also:

- MPI_Scatterv
- MPI_Gatherv
- MPI_Allgatherv
- MPI_Alltoallv

Example2: Odd-Even Sort

Algorithm:

- comparing & switch in order between all (odd, even)-indexed pairs of adjacent elements in the list
- comparing & switch in order between all (even,odd)-indexed pairs of adjacent elements in the list
- Repeat until the list is sorted

1. [Even-phase]
even/odd indexed adjacent elements are grouped into pairs.

Index	0	1	2	3	4	5	6	7
Value	6	1	4	8	2	5	9	3

2. [Even-phase]
elements in a pair are switched if they are in the wrong order.

Index	0	1	2	3	4	5	6	7
Value	1	6	4	8	2	5	3	9

3. [Odd-phase]
odd/even indexed adjacent elements are grouped into pairs.

Index	0	1	2	3	4	5	6	7
Value	1	6	4	8	2	5	3	9

4. [Odd-phase]
elements in a pair are switched if they are in the wrong order.

Index	0	1	2	3	4	5	6	7
Value	1	4	6	2	8	3	5	9

Sequential version:

```
bool sorted = false;
while(!sorted) {
  sorted=true;
  for(int i=0; i<N-1; i+=2) {
    if(a[i] > a[i+1]) {
      swap(a, i, i+1);
      sorted = false;
    }
  }

  for(int i=1; i<N-1; i+=2) {
    if(a[i] > a[i+1]) {
      swap(a, i, i+1);
      sorted = false;
    }
  }
}
```

```c
#include <stdlib.h>
#include <mpi.h> /* Include MPI's header file */

int main(int argc, char *argv[]) {
    int n; /* The total number of 1elements to be sorted */
    int npes; /* The total number of processes */
    int myrank; /* The rank of the calling process */
    int nlocal; /* The local number of elements, and the array that stores them */
    int *elmnts; /* The array that stores the local elements */
    int *relmnts; /* The array that stores the received elements */
    int oddrank; /* The rank of the process during odd-phase communication */
    int evenrank; /* The rank of the process during even-phase communication */
    int *wspace; /* Working space during the compare-split operation */
    int i;
    MPI_Status status;
    /* Initialize MPI and get system information */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    n = atoi(argv[1]);
    nlocal = n/npes; /* Compute the number of elements to be stored locally. */
    /* Allocate memory for the various arrays */
    elmnts = (int *)malloc(nlocal*sizeof(int));
    relmnts = (int *)malloc(nlocal*sizeof(int));
    wspace = (int *)malloc(nlocal*sizeof(int));
    /* Fill-in the elmnts array with random elements */
    srandom(myrank);
    for (i=0; i<nlocal; i++)
        elmnts[i] = random();
```

```
    /* Sort the local elements using the built-in quicksort routine */
    qsort(elmnts, nlocal, sizeof(int), IncOrder);
    /* Determine the rank of the processors that myrank needs to communicate during the odd and even phases */
    if (myrank%2 == 0) {
        oddrank = myrank-1;
        evenrank = myrank+1;
    }
    else {
        oddrank = myrank+1;
        evenrank = myrank-1;
    }
    /* Set the ranks of the processors at the end of the linear */
    if (oddrank == -1 || oddrank == npes)
        oddrank = MPI_PROC_NULL;
    if (evenrank == -1 || evenrank == npes)
        evenrank = MPI_PROC_NULL;
    /* Get into the main loop of the odd-even sorting algorithm */
    for (i=0; i<npes-1; i++) {
        if (i%2 == 1) /* Odd phase */
            MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
                nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
        else /* Even phase */
            MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
                nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);

        CompareSplit(nlocal, elmnts, relmnts, wspace, myrank < status.MPI_SOURCE);
    }

    free(elmnts); free(relmnts); free(wspace);
    MPI_Finalize();
}
```

Outline

- Point-to-Point Communication Routines
- Collective Communication Routines
- **Group and Communicator Management Routines**
- MPI-IO

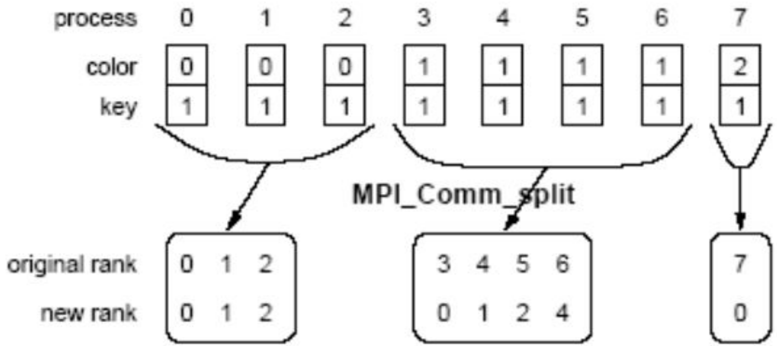
Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

- This operation groups processors by color and sorts resulting groups on the key.

Groups and Communicators



Using `MPI_Comm_split` to split a group of processes in a communicator into subgroups.

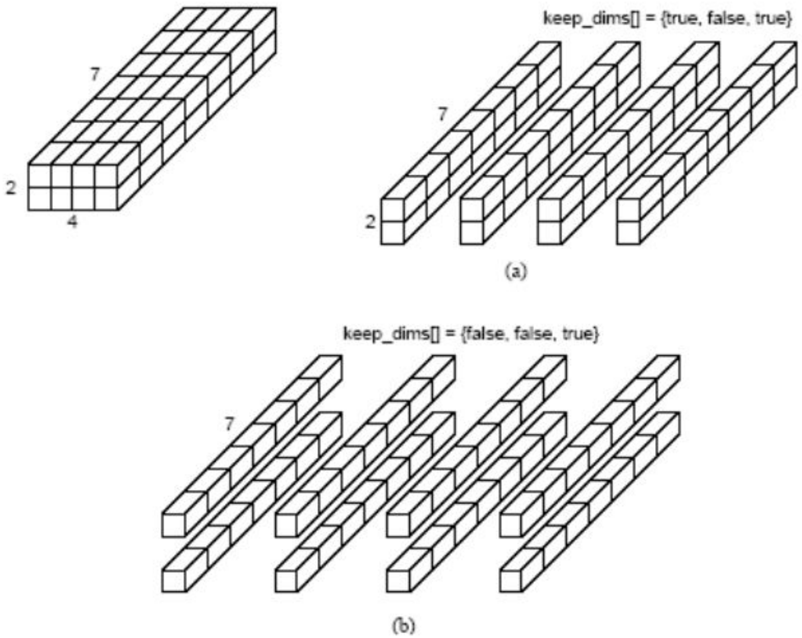
Groups and Communicators

- In many parallel algorithms, processes are arranged in a virtual grid, and in different steps of the algorithm, communication needs to be restricted to a different subset of the grid.
- MPI provides a convenient way to partition a Cartesian topology to form lower-dimensional grids:

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
                 MPI_Comm *comm_subcart)
```

- If `keep_dims[i]` is true (non-zero value in C) then the `i`th dimension is retained in the new sub-topology.
- The coordinate of a process in a sub-topology created by `MPI_Cart_sub` can be obtained from its coordinate in the original topology by disregarding the coordinates that correspond to the dimensions that were not retained.

Groups and Communicators



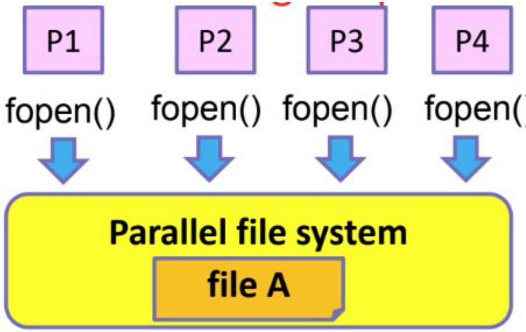
Splitting a Cartesian topology of size 2 x 4 x 7 into (a) four subgroups of size 2 x 1 x 7, and (b) eight subgroups of size 1 x 1 x 7.

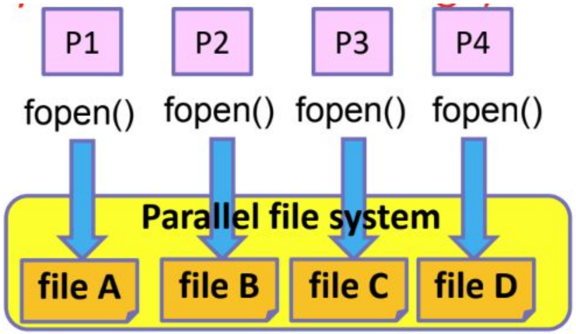
Outline

- Point-to-Point Communication Routines
- Collective Communication Routines
- Group and Communicator Management Routines
- **MPI-IO**

POSIX File Access Operations

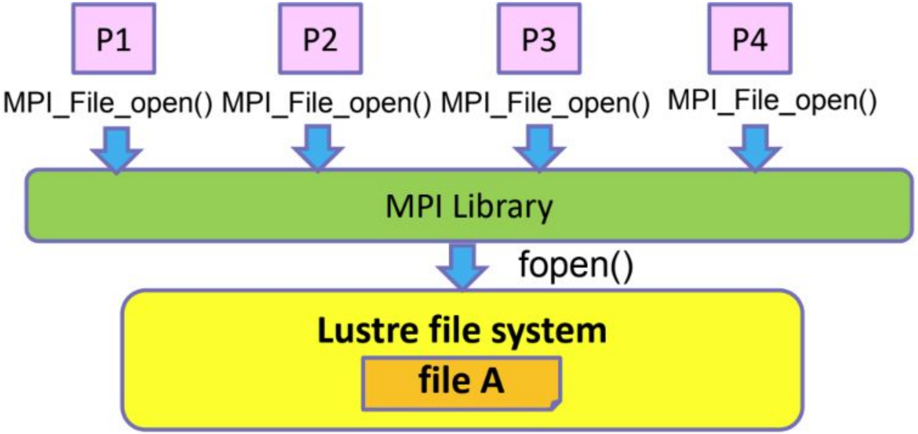
- POSIX file system call: **`fopen()`**
  - The same file is opened by each processes => multiple file handlers across your MPI processes
  - Open the same file with read permission is OK
  - But can't open with write permission together due file system locking mechanism => data inconsistency
  - To write simultaneously must create multiple files





MPI-IO File Access Operations

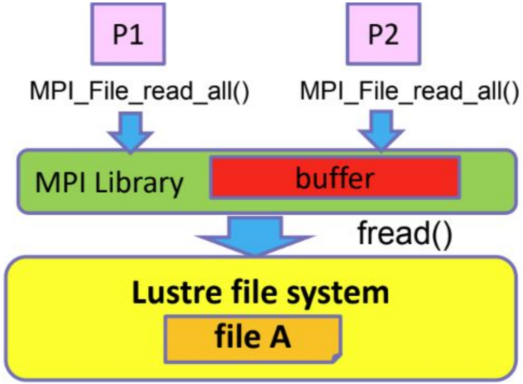
- MPI-IO call: **`MPI_File_open()`**
  - File is **opened only once** in a **collective** manner
  - MPI library will share and synchronize with each other to use the same file handler
  - Can handle both read and write together



MPI-IO Independent/Collective I/O

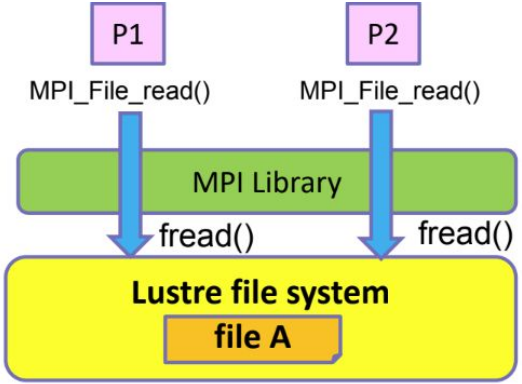
Collective I/O

- Read/write to a shared memory buffer, then issue **ONE** file request
- Reduce #I/O request => Good for small I/O
- Require **synchronization**



Independent I/O

- Read/write individually
- Prevent synchronization
- One request per process
- Request is **serialized if access the same OST** => Good for large I/O



MPI-IO API

- `MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)`
- `MPI_File_close(MPI_File *fh)`
- `MPI_File_read/write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
  - **Independent** read/write using individual file pointer
- `MPI_File_read/write_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
  - **Collectively** read/write using individual file pointer
- `MPI_File_sync(MPI_File fh)`
  - Flush all previous writes to the storage device

Backup Slides

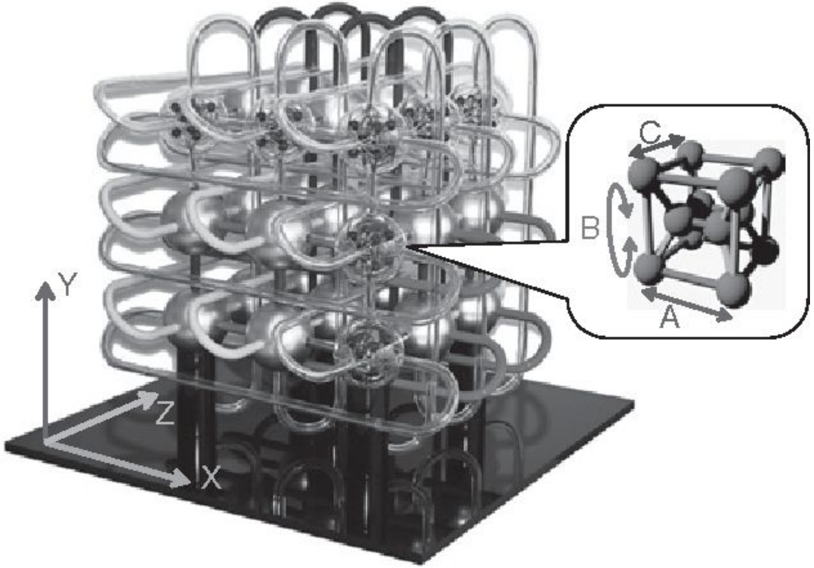
Transports

- TCP/IP
- Infiniband
- Omni-Path
- Cray Aries

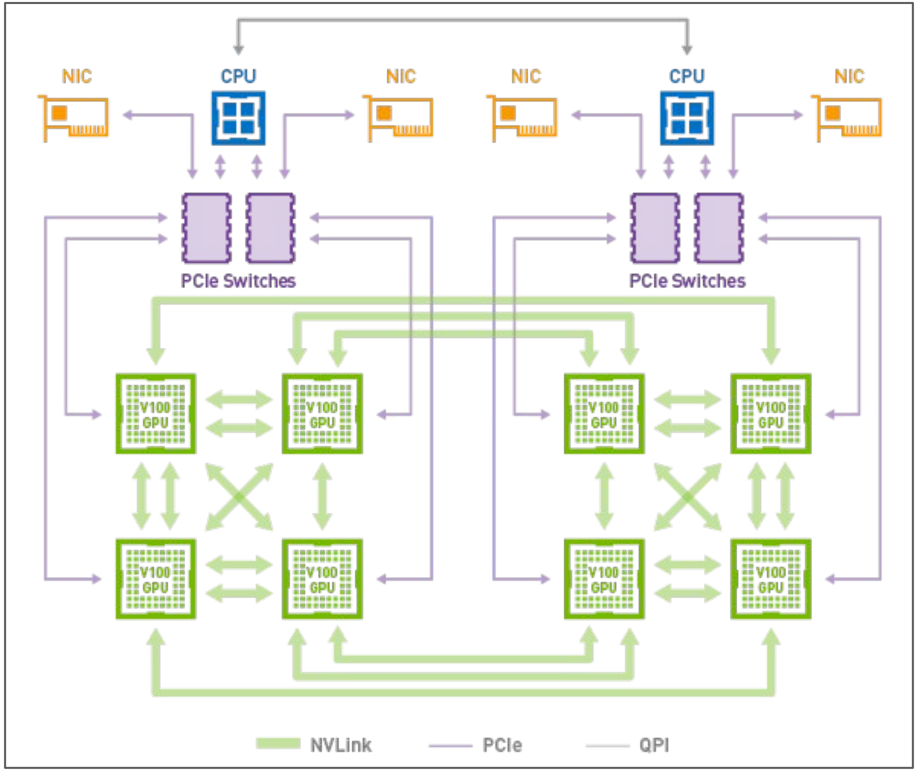
...

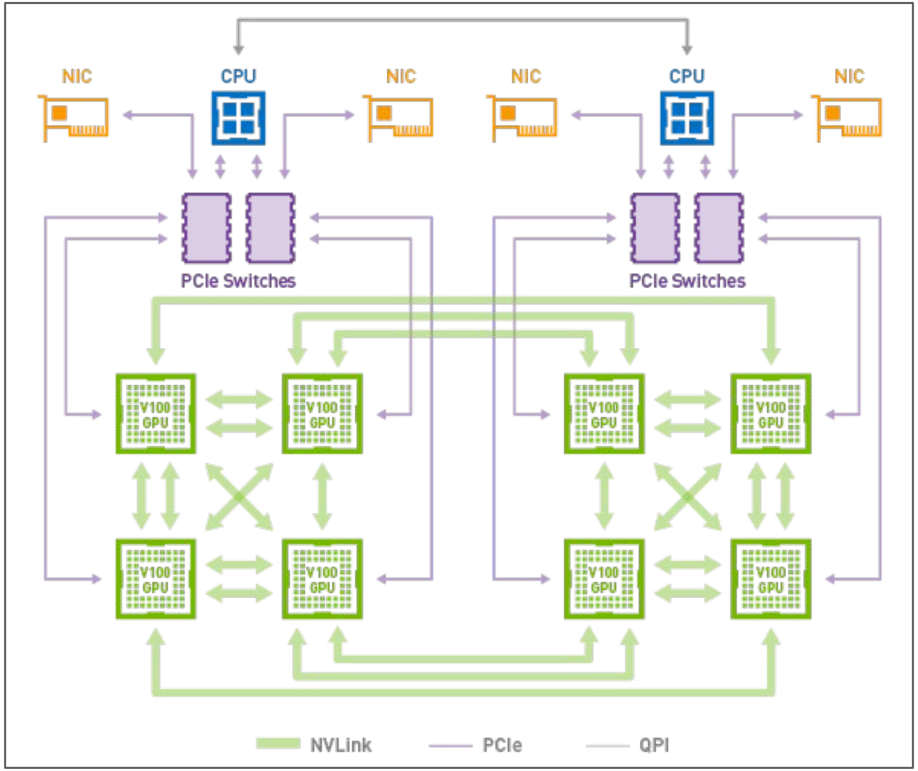
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

Tofu interconnection - Fujitsu



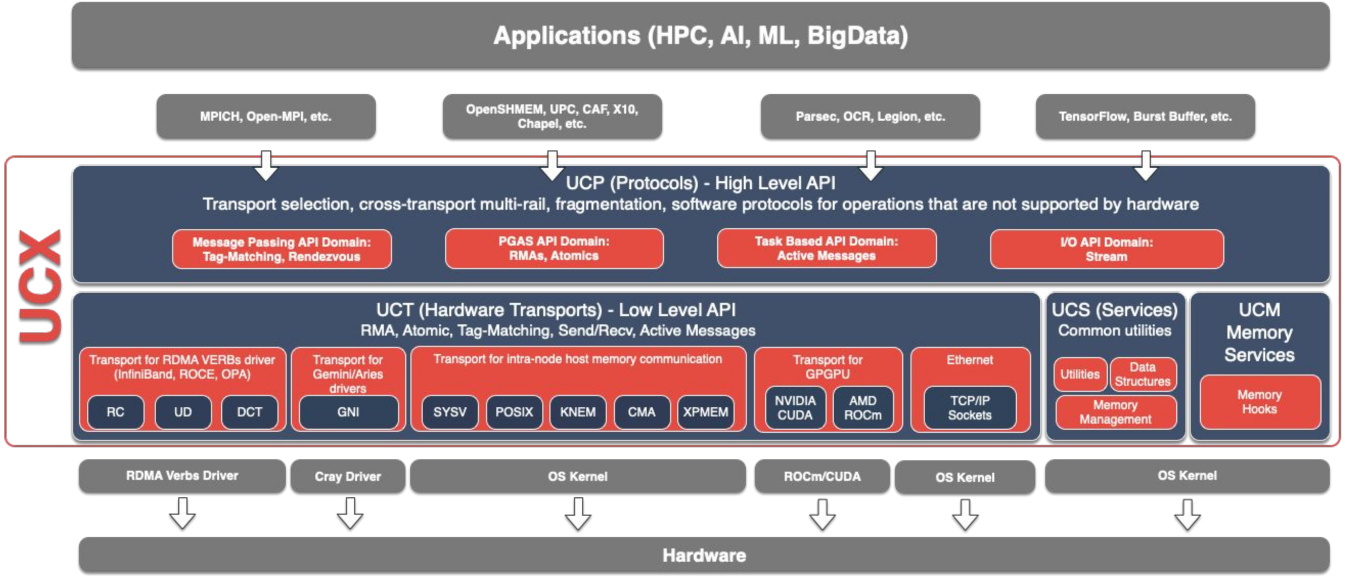
GPUs


NIC
CPU
NIC
NIC
CPU
NIC
PCIe Switches
PCIe Switches
V100 GPU
NVLink
PCIe
QPI


NIC
CPU
NIC
NIC
CPU
NIC
PCIe Switches
PCIe Switches
V100 GPU
NVLink
PCIe
QPI

MPI Implementation

- Intel MPI
- MPICH2
- MVAPICH2
- OpenMPI
- PlatformMPI



Q: Divide N elements to M groups

Each set number must be continuous to increase the locality of the data and each size is similar.

1. Intuitive
2. Lazy programmer
3. Floor / Ceil
4. Residual

1. Intuitive

```
void method0(int n, int m) {
    for(int i = 0, L = 0, R; i < m; i++) {
        int size_i = n/m + (i < n%m);
        R = L + size_i - 1;
        printf("%d len([%d, %d]) = %d\n", i, L, R, (R - L + 1));
        L = R + 1;
    }
}
```

2. Lazy programmer

```
void method1(int n, int m) {
    for (int i = 0, L, R; i < m; i++) {
        L = i*n/m, R = (i+1)*n/m - 1;
        printf("%d len([%d, %d]) = %d\n", i, L, R, (R - L + 1));
    }
}
```

3. Floor / Ceil

$$n = \left\lceil \frac{n}{m} \right\rceil + \left\lceil \frac{n-1}{m} \right\rceil + \cdots + \left\lceil \frac{n-m-1}{m} \right\rceil$$

$$n = \left\lfloor \frac{n}{m} \right\rfloor + \left\lfloor \frac{n+1}{m} \right\rfloor + \cdots + \left\lfloor \frac{n+m-1}{m} \right\rfloor$$

Chapter 3, Concrete Mathematics, Donald Knuth

4. Residual

```
void method4(int n, int m) {
    int base = (n+m-1)/m;
    for (int i = 0, L = 0, R; i < m; i++) {
        L = i*base, R = min(L+base-1, n-1);
        printf("%d len([%d, %d]) = %d\n", i, L, R, (R - L + 1));
    }
}
```

Reference

- LLNL MPI Tutorial https://hpc-tutorials.llnl.gov/mpi/
- OpenMPI Doc https://www.open-mpi.org/doc/v4.1/