

# MPI - Part 2

---

- Basic message passing
- Semantics
- Simple IO

# Introduction to MPI

---

- MPI is a standard for message passing interfaces
- MPI-1 covers point-to-point and collective communication
- MPI-2 covers connection based communication and I/O
- Typical implementations include MPICH (Used by Scyld), and LAMMPI

# MPI Basics

---

- Most used MPI commands

`MPI_Init` - start using MPI

`MPI_Comm_size` - get the number of tasks

`MPI_Comm_rank` - the unique index of this task

`MPI_Send` - send a message

`MPI_Recv` - receive a message

`MPI_Finalize` - stop using MPI

# Initialize and Finalize

---

```
int MPI_Init(int *argc, char ***argv);
```

- Must be called before any other MPI calls

```
int MPI_Finalize();
```

- No MPI calls may be made after this
- Program **MUST** call this (or it won't terminate)

# Initialize and Finalize

---

- First MPI call must be to `MPI_Init`.
- Last MPI call must be to `MPI_Finalize`.

```
#include <mpi.h>
main(int argc, int **argv)
{
    MPI_Init(&argc, &argv );
    // put program here
    MPI_Finalize();
}
```

# Size and Rank

---

- `MPI_Comm_size` returns the number of tasks in the job

```
int size;
```

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

- `MPI_Comm_rank` returns the number of the current task (0 .. size-1)

```
int rank;
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

# A Simple Example

---

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

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

    int rank, size;

    MPI_Init( &argc,&argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
    MPI_Comm_size( MPI_COMM_WORLD, &size);
    printf("Hello world from process %d of %d \n",
           rank,size);
    MPI_Finalize();
    return 0;
}
```

# Send and Recv

---

- MPI\_Send to send a message

```
char sbuf[COUNT];
```

```
MPI_Send(sbuf, COUNT, MPI_CHAR, 1,  
         99, MPI_COMM_WORLD);
```

- MPI\_Recv to receive a message

```
char rbuf[COUNT];
```

```
MPI_Status status;
```

```
MPI_Recv(rbuf, COUNT, MPI_CHAR, 1,  
         99, MPI_COMM_WORLD, &status);
```

# Anatomy of MPI\_Send

---

- `MPI_Send(sbuf, COUNT, MPI_CHAR, 1, 99, MPI_COMM_WORLD);`

`sbuf` : pointer to send buffer

`COUNT` : items in send buffer

`MPI_CHAR` : MPI datatype

`1` : destination task number (rank)

`99` : message tag

`MPI_COMM_WORLD` : communicator

# Anatomy of MPI\_Recv

---

- `MPI_Recv(rbuf, COUNT, MPI_CHAR, 1, 99, MPI_COMM_WORLD, &status);`

`rbuf` : pointer to receive buffer

`COUNT` : items in receive buffer

`MPI_CHAR` : MPI datatype

`1` : source task number (rank)

`99` : message tag

`MPI_COMM_WORLD` : communicator

`Status` : pointer to status struct

# MPI Datatypes

---

- Encodes type of data sent and received
- Built-in types

`MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`  
`MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG_DOUBLE`  
`MPI_BYTE`, `MPI_PACKED`

- User defined types (covered later)

`MPI_Type_contiguous`, `MPI_Type_vector`,  
`MPI_Type_indexed`, `MPI_Type_struct`  
`MPI_Pack`, `MPI_Unpack`

# MPI Communicators

---

- Abstract structure represents a group of MPI tasks that can communicate
- `MPI_COMM_WORLD` represents all of the tasks in a given job
- Programmer can create new communicators to subset `MPI_COMM_WORLD`
- `RANK` or task number is relative to a given communicator
- Messages from different communicators do not interfere

# MPI Task Numbers

---

- Each task in a job has a unique rank or task number
- Numbers run from 0 to `size-1`, where `size` is the number of tasks
- `MPI_Comm_size(MPI_COMM_WORLD, &size)`
- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`
- `Send` and `Recv` specify destination or source task by rank
- `Recv` can specify source of `MPI_ANY_SOURCE` to receive from any task

# Message Tags

---

- All messages are sent with an integer message tag
- `MPI_Recv` will only receive a message with the tag specified
- `MPI_ANY_TAG` can be used to receive messages with any tag

# MPI Status Struct

---

- Allows user to query the return status of MPI call

```
status.MPI_SOURCE
```

```
status.MPI_TAG
```

```
status.MPI_ERROR
```

- Allows user to query number of items received

```
int count;
```

```
MPI_Get_count(&status, MPI_CHAR,  
             &count)
```

# Send and Receive Example

---

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

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

    int numprocs, myrank, namelen, i;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    char greeting[MPI_MAX_PROCESSOR_NAME + 80];
    MPI_Status status;

    MPI_Init( &argc,&argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
    MPI_Get_Processor_name( processor_name, &namelen);

    sprintf(greeting, "Hello world from process %d of %d on %s \n",
            myrank,numprocs, processor_name);
```

# Send and Receive Example

---

```
if (myrank == 0) {
    printf("%s\n", greeting);
    for(i=1;i<numprocs;i++) {
        MPI_Recv(greeting,sizeof(greeting), MPI_CHAR,
            i, 1, MPI_COMM_WORLD);
        printf("%s\n", greeting);
    }
}
else {
    MPI_Send(greeting, strlen(greeting) +1, MPI_CHAR,
        0,1,MPI_COMM_WORLD);
}

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

# Receive Buffer Size

---

- Receive buffer must be big enough for the message being received
- If message is smaller, only part of buffer filled in
- If message is too big, overflow error
- MPI\_Probe allows programmer to check the next message before receiving it

```
// src, tag, comm, stat
MPI_Probe(1, 99, MPI_COMM_WORLD,
          &status);
```

# Matching Send and Recv

---

- When `MPI_Send` called, send is "posted"
- When `MPI_Recv` called, receive is "posted"
- Posted `MPI_Recv` matches posted `MPI_Send` if
  - Destination of `MPI_Send` matches receiving task
  - Source of `MPI_Recv` matches sending task or source is `MPI_SOURCE_ANY`
  - Tag of `MPI_Send` matches tag of `MPI_Recv` or tag of `MPI_Recv` is `MPI_TAG_ANY`
  - Communicator of `MPI_Send` matches communicator of `MPI_Recv`

# Semantics

---

- Covered on next pages ...
  - Messages are non-overtaking
  - Progress is guaranteed
  - Fairness is not guaranteed
  - System resources may be limited

# Non-Overtaking Messages

---

- If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending.
- If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending.

Quoted from: MPI: A Message-Passing Interface Standard

(c) 1993, 1994, 1995 University of Tennessee, Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted, provided the University of Tennessee copyright notice and the title of this document appear, and notice is given that copying is by permission of the University of Tennessee.

# Non-overtaking Messages

---



# Progress

---

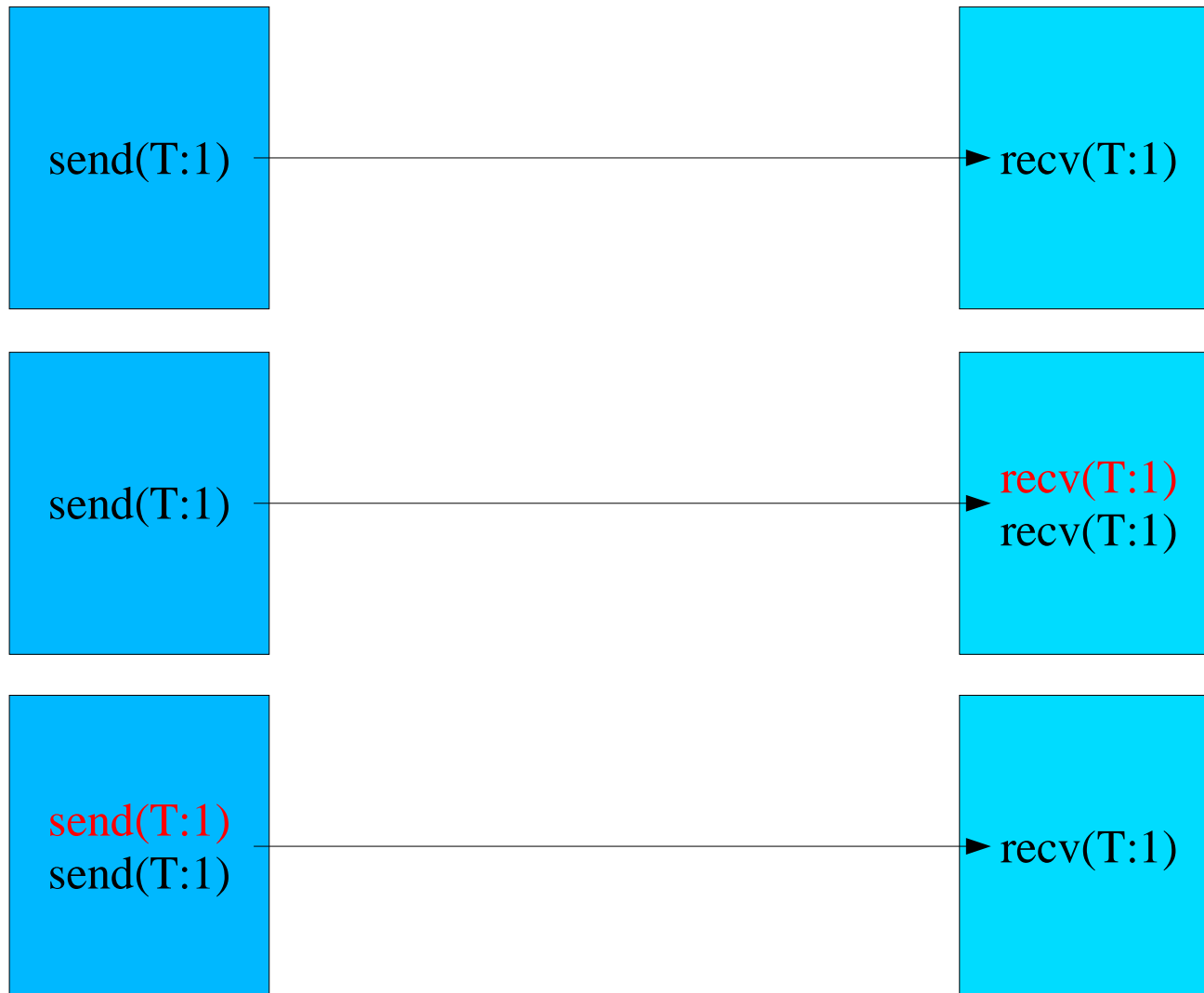
- If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system: the send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same destination process.

Quoted from: MPI: A Message-Passing Interface Standard

(c) 1993, 1994, 1995 University of Tennessee, Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted, provided the University of Tennessee copyright notice and the title of this document appear, and notice is given that copying is by permission of the University of Tennessee.

# Progress Guaranteed

---



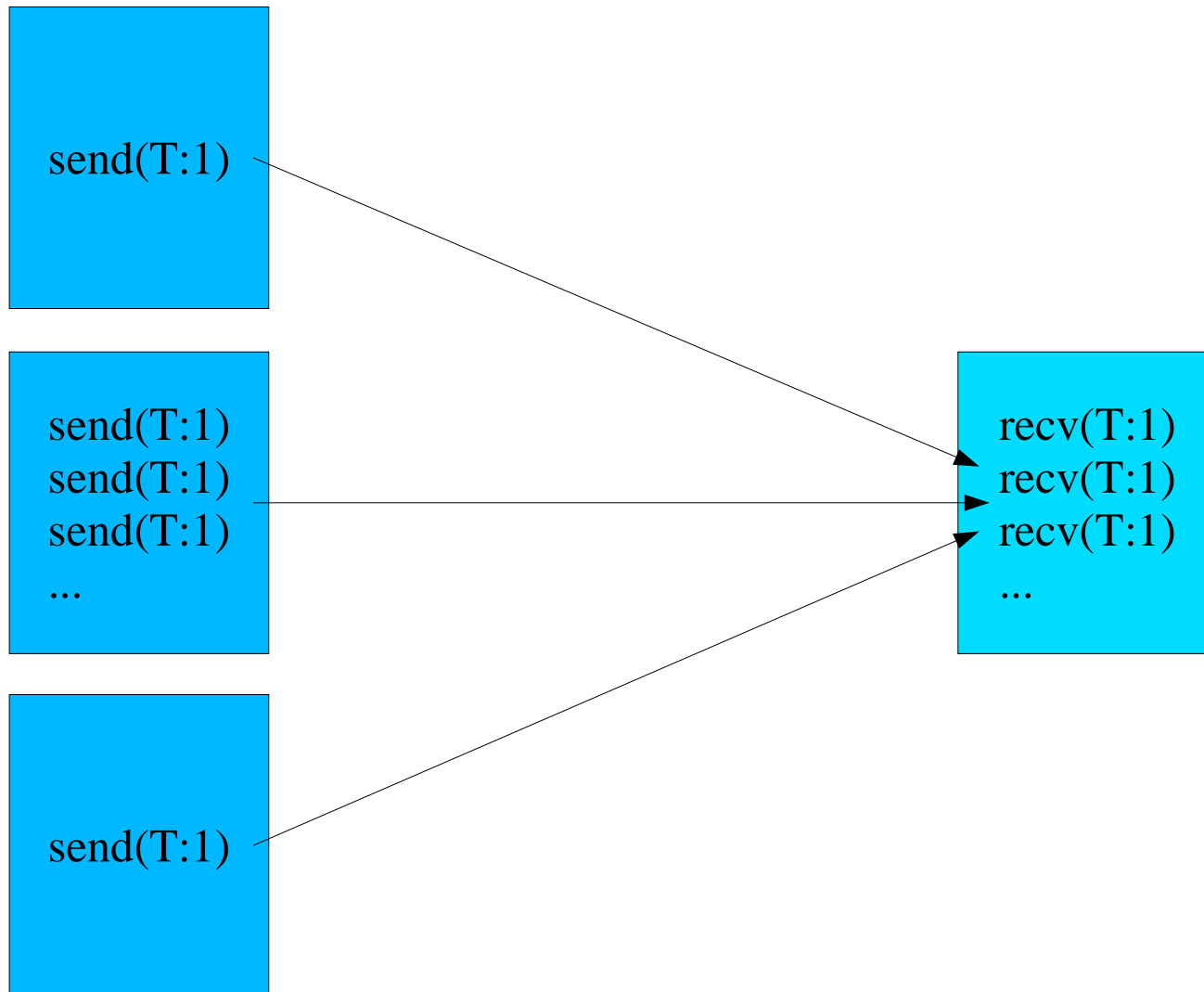
# Fairness

---

- MPI does not guarantee fairness in matching messages
- Messages from different sources may overtake one another
- Programmer's responsibility to prevent starvation

# Fairness NOT Guaranteed

---



# Limited System Resources

---

- MPI does not guarantee system resources exist for buffering messages
- Programs that assume system resources available can deadlock if resources become busy
- Properly coded programs usually exist that will complete regardless of available buffer space
- Buffered mode allows programmer to provide adequate buffer space

# Combined Send/Recv

---

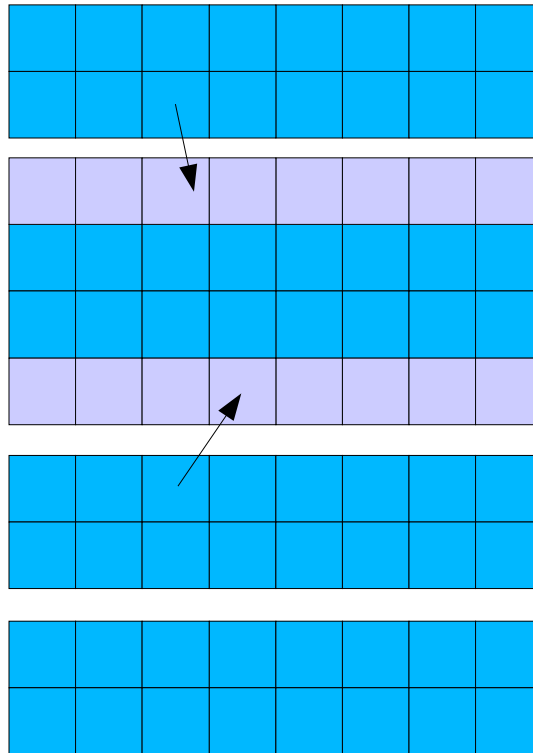
- Single call both sends and receives a message
- Can send and receive to same task, or different tasks
- Guarantees that buffering and blocking semantics will not result in deadlock

```
MPI_Sendrecv(sbuf, scount, stype,  
             dest, stag, rbuf, rcount, rtype,  
             source, rtag, comm, &status);
```

```
MPI_Sendrecv_replace(buf, count,  
                     type, dest, stag, source, rtag,  
                     comm, &status);
```

# Smoothing Example

---



# Back to our example

---

```
#include <mpi.h>
#define n 1000;
main(int argc, char **argv)
{
    int n, SIZE, RANK;
    int *input, *output;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &SIZE);
    MPI_Comm_rank(MPI_COMM_WORLD, &RANK);

    input = (int *)malloc((n/SIZE+2) * n * sizeof(int));
    output = (int *)malloc(n/SIZE * n * sizeof(int));

    read_image(n, SIZE, RANK, input);
    parallel_smooth(n, SIZE, RANK, input, output);
    write_image(n, SIZE, RANK, output);

    MPI_Finalize();
}
```

# Smoothing Function

---

```
parallel_smooth(int n, int SIZE, int RANK,
               int input[][n], int output[][n])
{
    int r, c, rm, cm, sum, cnt;
    exchange_borders(n, SIZE, RANK, input);
    for (r = 1; r < (n/SIZE)+1; r++)
        for (c = 0; c < n; c++)
            {
                cnt = 0;
                sum = 0;
                for (rm = -1; rm < 2; rm++)
                    {
                        if (RANK == 0 && r+rm < 1 ||
                            RANK == SIZE-1 && r+rm > n/SIZE)
                            continue;
                        for (cm = -1; cm < 2; cm++)
                            {
                                if (c+cm < 0 || c+cm >= n)
                                    continue;
                                sum += input[r+rm][c+cm];
                                cnt++;
                            }
                    }
                output[r][c] = sum / cnt;
            }
}
```

# Exchange Function

---

```
exchange_borders(int n; int SIZE, int RANK, int input[][n])
{
    MPI_Status status;
    if (RANK < SIZE-1)
    {
        MPI_Send(&input[1][0], n, MPI_INT, RANK+1, 1,
                MPI_COMM_WORLD);
        MPI_Recv(&input[0][0], n, MPI_INT, RANK+1, 1,
                MPI_COMM_WORLD, &status);
    }
    if (RANK > 0)
    {
        MPI_Send(&input[n/SIZE][0], n, MPI_INT, RANK-1, 1,
                MPI_COMM_WORLD);
        MPI_Recv(&input[n/SIZE+1][0], n, MPI_INT, RANK-1, 1,
                MPI_COMM_WORLD, &status);
    }
}
```

# Exchange Using Sendrecv

---

```
exchange_borders(int n; int SIZE, int RANK, int input[][n])
{
    MPI_Status status;
    if (RANK < SIZE-1)
    {
        MPI_Sendrecv(&input[1][0], n, MPI_INT, RANK+1, 1,
                    &input[0][0], n, MPI_INT, RANK+1, 1,
                    MPI_COMM_WORLD, &status);
    }
    if (RANK > 0)
    {
        MPI_Sendrecv(&input[n/SIZE][0], n, MPI_INT, RANK-1, 1,
                    &input[n/SIZE+1][0], n, MPI_INT, RANK-1, 1,
                    MPI_COMM_WORLD, &status);
    }
}
```

# Working Exchange

---

```
exchange_borders(int n; int SIZE, int RANK, int input[][n])
{
    MPI_Status status;
    if (RANK == 0)
        MPI_Send(&input[n/SIZE][0], n, MPI_INT, 1, 1,
                MPI_COMM_WORLD);
    else if (RANK < SIZE-1)
        MPI_Sendrecv(&input[n/SIZE][0], n, MPI_INT, RANK+1, 1,
                    &input[0][0], n, MPI_INT, RANK-1, 1,
                    MPI_COMM_WORLD, &status);
    else
        MPI_Recv(&input[0][0], n, MPI_INT, RANK-1, 1,
                MPI_COMM_WORLD, &status);
    if (RANK == 0)
        MPI_Recv(&input[n/SIZE+1][0], n, MPI_INT, 1, 1,
                MPI_COMM_WORLD);
    else if (RANK < SIZE-1)
        MPI_Sendrecv(&input[1][0], n, MPI_INT, RANK-1, 1,
                    &input[n/SIZE+1][0], n, MPI_INT, RANK+1, 1,
                    MPI_COMM_WORLD, &status);
    else
        MPI_Send(&input[1][0], n, MPI_INT, RANK-1, 1,
                MPI_COMM_WORLD, &status);
}
```

# Program IO

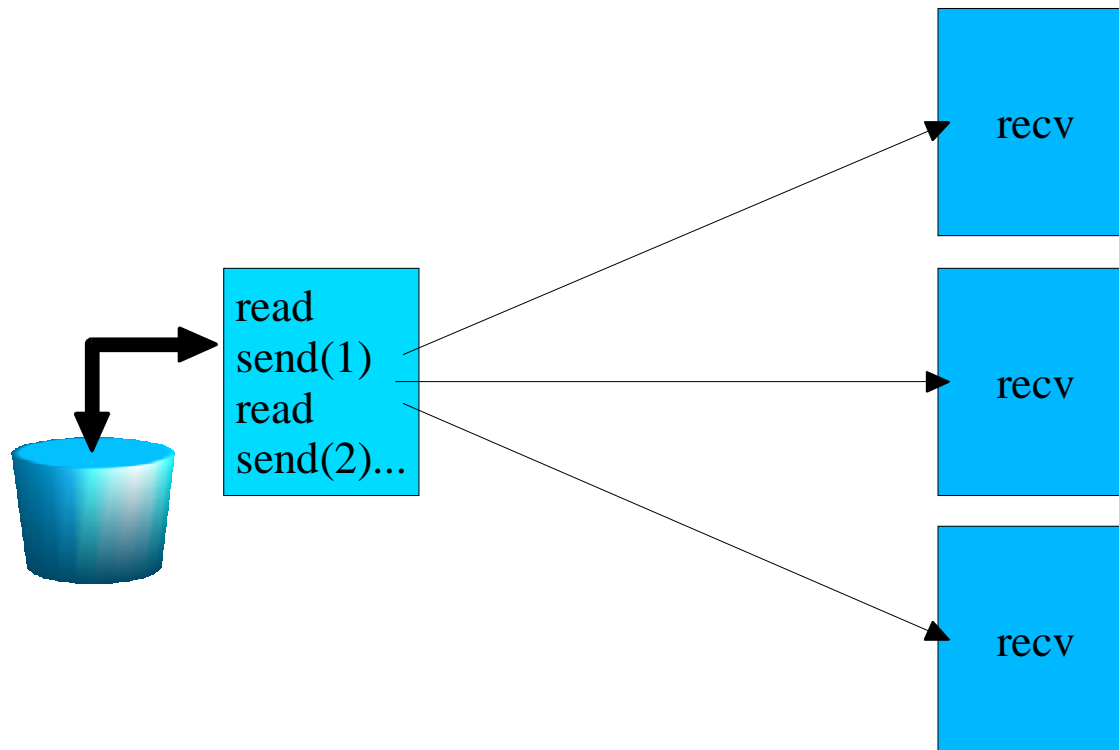
---

- Master task IO model
  - One task (task 0?) does all IO
  - Sends/recvs from other tasks
  - A must if data not available on all nodes
- Independent IO model
  - Each task does its own IO
  - Each node must have data available
- Hybrid models
  - Subset of nodes have access to data
- Parallel IO model
  - System software supports IO by parallel tasks

# Master Task IO

---

- One task reads data, then sends to other tasks
- Other tasks receive data from IO task



# Master task IO

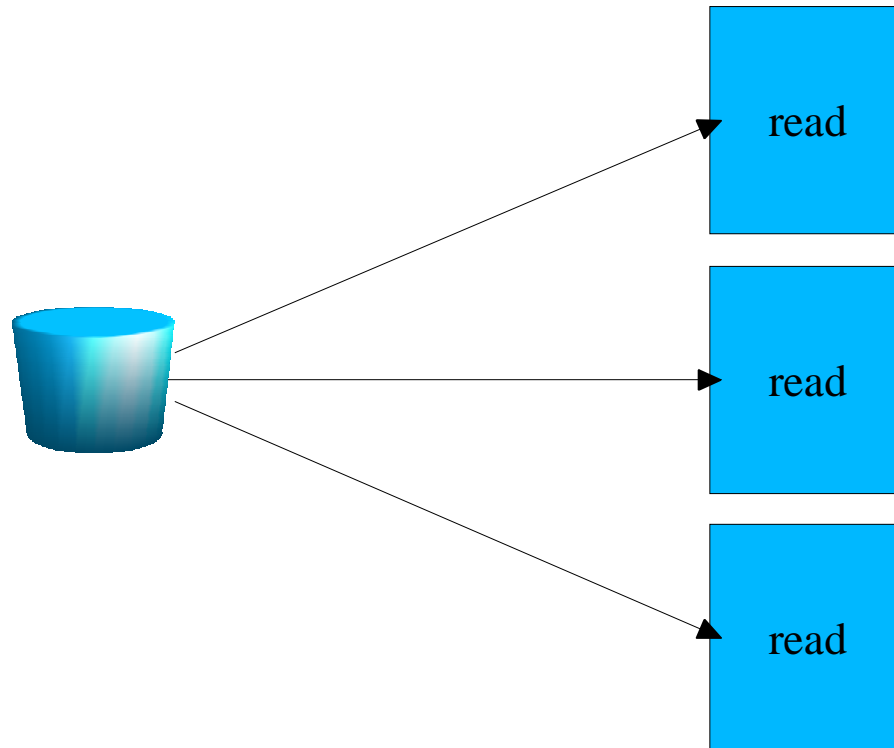
---

```
parallel_read_buffer (int fd, char *buffer, int count)
{
    if (RANK == 0)
    {
        int t;
        lseek (fd, SIZE, SEEK_SET); /* skip task 0's data */
        for (t = 1; t < SIZE; t++)
        {
            read(fd, count, buffer); /* assume task data is sequential */
            MPI_Send(buffer, count, MPI_BYTE, t, 1, MPI_COMM_WORLD);
        }
        lseek (fd, 0, SEEK_SET); /* now read task 0's data */
        read(fd, count, buffer);
    }
    else
    {
        MPI_Status status;
        MPI_Recv(buffer, count, MPI_BYTE, 0, 1,
                MPI_COMM_WORLD, &status);
    }
}
```

# Independent IO

---

- Each task reads its own data
- Each task must determine which data to read



# Independent IO

---

```
parallel_read_buffer (int fd, char *buffer, int count)
{
    int start;
    start = RANK * count;
    lseek (fd, start, SEEK_SET);
    read(fd, count, buffer);
}
```

```
ppart(int total, int *first, int *count, int *max)
{
    *max = *count = total / SIZE;
    *first = RANK * *count;
    *max += total - (*count * SIZE);
    if (RANK == SIZE-1)
        *count = *max;
}
```

# Partitioning Items

---

```
ppart(int total, int *first, int *count, int *max)
{
    int tcount = total / SIZE;
    int leftover = total -
        (tcount * SIZE);
    if (max && leftover > 0)
        *max = tcount + 1;
    if (RANK < leftover)
        tcount++;
    if (first)
        if (RANK < leftover)
            *first = RANK * tcount;
        else
            *first = RANK * tcount + leftover
    if (count)
        *count = tcount;
}
```

# Independent IO - Redux

---

```
parallel_read_from (int fd, int offset, char *buffer,  
                    int tcount, int size)  
{  
    int first, count;  
    ppart(tcount, &first, &count, NULL);  
    start = first * size + offset;  
    lseek (fd, start, SEEK_SET);  
    read(fd, count * size, buffer);  
}
```

# Independent Writes

---

- Work just like reads except ...
  - Local cache may cause unexpected behavior
  - File creation can be tricky
  - Extending a file can cause data loss
- When using NFS, usually best to ...
  - Have one task create file
  - Have one task pre-allocate file to final length
- Use of MPI-IO (covered later)
  - Should fix task creation/extending problems
  - May fix cache related issues
  - Depends on file system used for implementation

# Parallel Open

---

```
int parallel_open (char *fname, int flags,
                  int mode, int size)
{
    int fd;
    char data = 0;
    if (RANK == 0)
    {
        fd = open(fname, flags, mode);
        lseek(fd, size-1, SEEK_SET);
        write(fd, &data, 1);
        lseek (fd, 0, SEEK_SET);
        MPI_Barrier(MPI_COMM_WORLD);
    }
    else
    {
        MPI_Barrier(MPI_COMM_WORLD);
        fd = open (fname, flags, mode);
    }
    return fd;
}
```

# Barrier

---

- Barrier synchronizes all tasks of a communicator

```
MPI_Barrier(MPI_COMM_WORLD);
```

- Each task calling `MPI_Barrier` will stop until all tasks in the communicator have called `MPI_Barrier`

# Running MPI programs with LAM

---

- LAMMPI already installed on ullab machines
- Compile your programs with `mpicc`
  - Automatically handles includes and libraries
  - Otherwise just like `cc` or `gcc`
- Run `lamboot` to start local area machine
  - All tasks run on local machine
  - `bhost` file specifies additional machines
  - Shut down machine with `wipe`
- Run programs with `mpirun`

# Mpirun

---

- Runs one or more tasks on nodes:

```
mpirun [options] prog-name arg1 arg2 ...
```

```
-np <int>          run <int> tasks
```

```
--all_cpus        run a task on each cpu
```

```
--all_local      run all tasks on master node
```

```
--no_local       run no tasks on master node
```

```
--exclude <int>[:<int>]
```

```
run on any nodes not listed
```

```
--map <int>[:<int>]
```

```
run nodes listed
```