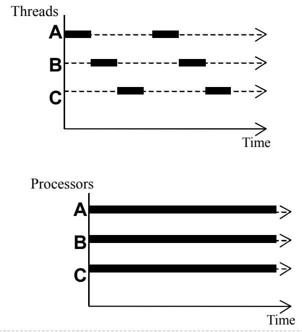
Parallel Programming and MPI- Lecture 1

Abhik Roychoudhury
CS 3211
National University of Singapore

Sample material: Parallel Programming by Lin and Snyder, Chapter 7.
Made available via IVLE reading list, accessible from Lesson Plan.

Concurrency and Parallelism



Why parallel programming?

- Performance, performance, performance!
- Increasing advent of multi-core machines!!
  - Homogeneous multi-processing architectures.
  - Discussed further in a later lecture.
- Parallelizing compilers never worked!
  - Automatically extracting parallelism from app. is very hard
- Better for the programmer to indicate which parts of the program to execute in parallel and how.

How to program for parallel machines?

- Use a parallelizing compiler
  - Programmer does nothing, too ambitious !
- Extend a sequential programming language
  - Libraries for creation, termination, synchronization and communication between parallel processes.
  - The base language and its compiler can be used.
  - Message Passing Interface (MPI) is one example.
- Design a parallel programming language
  - Develop a new language – Occam.
    - Or add parallel constructs to a base language – High Perf. Fortran.
  - Must beat programmer resistance, and develop new compilers.

Parallel Programming Models

- Message Passing
  - MPI: Message Passing Interface
  - PVM: Parallel Virtual Machine
  - HPF: High Performance Fortran
- Shared Memory
  - Automatic Parallelization
  - POSIX Threads (Pthreads)
  - OpenMP: Compiler directives

The Message-Passing Model

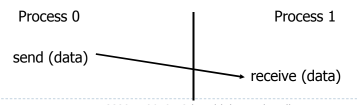
- A process is (traditionally) a program counter and address space
- Processes may have multiple threads (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's

The programming model in MPI

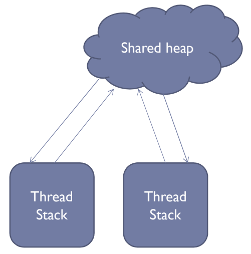
- Communicating Sequential Processes
  - Each process runs in its local address space.
  - Processes exchange data and synchronize by message passing
  - Typically, but not always, the same code may be executed by all processes.

Cooperative Operations for Communication

- Message-passing approach makes the exchange of data cooperative
- Data is explicitly sent by one process and received by another
- Advantage:
  - Any change in the receiving process's memory is made with the receiver's active participation.
- Communication and synchronization are combined.



Shared Memory communication in Java



Java program compiled into bytecodes. Bytecodes are interpreted by the Java Virtual Machine.

Bytecodes are the assembly language of the Java Virtual Machine (a machine implemented in software).

Bytecode execution returns in movements between thread local stack and the shared heap (which is shared across threads).

Program to Bytecode

```
3: public int foo(int j){
4: int ret;
5: if ( j % 2 == 1 )
6: ret= 2;
7: else
8: ret= 5;
9: return ret;
10: }
```

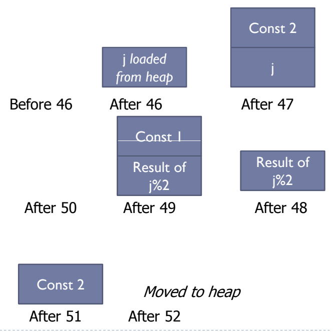
```
public int foo(int);
46: iload_1
47: iconst_2
48: irem
49: iconst_1
50: if_icmpne 54
51: iconst_2
52: istore_2
53: goto 56
54: iconst_5
55: istore_2
56: iload_2
57: ireturn
```

Simplified Bytecode format

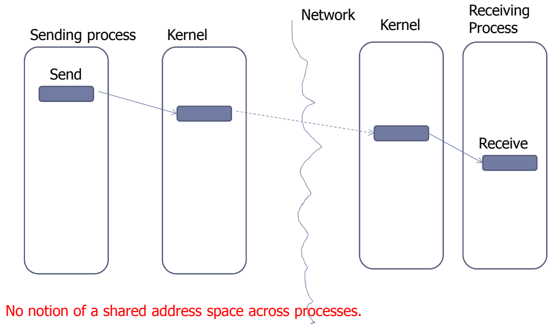
Stack ↔ Heap movements in Java

```
public int foo(int);
46: iload_1
47: iconst_2
48: irem
49: iconst_1
50: if_icmpne 54
51: iconst_2
52: istore_2
53: goto 56
54: iconst_5
55: istore_2
56: iload_2
57: ireturn
```

j%2 == 1
ret = 2

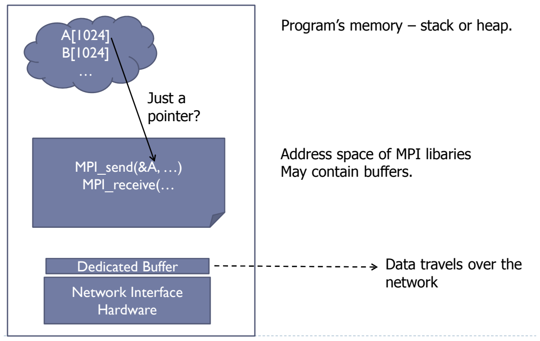


In comparison, communication in MPI is:



No notion of a shared address space across processes.

More elaborate view of a MPI process


A[1024]
B[1024]
…
Just a pointer?
MPI_send(&A, …)
MPI_receive(…
Dedicated Buffer
Network Interface Hardware
Program's memory – stack or heap.
Address space of MPI libaries
May contain buffers.
Data travels over the network

Message Passing Interface (MPI)

- A message-passing library specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Designed to provide access to parallel hardware for
  - End users
  - Library writes
  - Tool developers
- Provides a powerful, efficient, and portable way to express parallel programs

MPI (Contd.)

- The processes in a parallel program are written in a sequential language (e.g., C or Fortran)
- Processes communicate and synchronize by calling functions in MPI library
- Single Program, Multiple Data (SPMD) style
  - Processors execute copies of the same program
  - Each instance determines its identity and takes different actions

MPI History

- Message Passing Interface Forum
  - Representative from over 40 organizations
- Goal
  - Develop a single library that could be implemented efficiently on the variety of multiprocessors
- MPI-1 accepted in 1994
- MPI-2 accepted in 1997
- MPI is a standard
- Several implementations exist

Some Basic Concepts

- Processes can be collected into groups
  - An ordered set of processes.
- A group and context together form a communicator
  - A scoping mechanism to define a group of processes.
  - For example define separate communicators for application level and library level routines.
- A process is identified by its rank in the group associated with a communicator
- There exists a default communicator whose group contains all initial processes, called MPI_COMM_WORLD

MPI Datatypes

- Data in a message is described by a triple
  - <address, count, datatype> where
- MPI datatype is recursively defined as
  - Predefined corresponding to a data type from the language (MPI_INT, MPI_DOUBLE)
  - A contiguous array of MPI datatypes
  - A strided block of datatypes
  - An indexed array of blocks of datatypes
  - An arbitrary structure of datatypes
- MPI functions can be used to construct custom datatypes

Why datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication)
- Specifying application-oriented layout of data in memory
  - Reduces memory-to-memory copies in the implementation
  - Allows the use of special hardware (scatter/gather) when available

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a tag or not screened by specifying MPI_ANY_TAG as the tag in a receive

Basic MPI Functions

- MPI_Init(int *argc, char ***argv)
  - Initializes MPI
  - Must be called before any other MPI functions
- MPI_Comm_rank(MPI_Comm comm,
  int *rank)
  - Find my rank within specified communicator
- MPI_Comm_size (MPI_Comm comm,
  int *size)
  - Find number of group members within specified communicator
- MPI_Finalize ()
  - Called at the end to clean up

Getting started

```
#include "mpi.h"
#include <stdio.h>
int main( argc, argv )
int argc;
char **argv; {
   MPI_Init( &argc, &argv );
   printf( "Hello world\n" );   /* run on each process */
   MPI_Finalize();
   return 0;
}
```

MPI_Comm_size and MPI_comm_rank

- Two of the first questions asked in a parallel program are:
  - How many processes are there? and
  - Who am I?
- How many is answered with
  - MPI_Comm_size
- Who am I is answered with
  - MPI_Comm_rank.
  - The rank is a number between zero and size-1.

What does this program do?

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

int main( argc, argv )
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! I'm %d of %d\n", rank, size );
   MPI_Finalize();
   return 0;
}
```

Embarrassingly simple MPI program

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i, id, p;
    void unit_task( int, int); // no return value

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    for (i=id; i < 65536; i+=p) unit_task(id, i);
    printf("Process %d is done\n", id);
    fflush(stdout); MPI_Finalize();
    return 0;
}
```

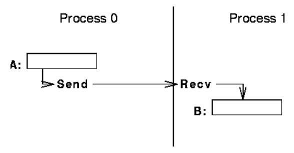
Compile: mpicc –o simple simple.c

Run: mpirun –np 2 simple (creating 2 processes)

Organization

- So Far
  - What is MPI
  - Entering and Exiting MPI
  - Creating multiple processes
- Now
  - Message Passing

Inter-process comunication



- Via point-to-point message passing.
- Messages are stored in message buffers.

Basic Blocking Communication

- int MPI_Send (void *buff, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - Send contents of a variable (single or array) to specified PE within specified communicator
- When this function returns, the data has been delivered and the buffer can be reused. The message may not have been received by the target process

- [Blocking here means]
  - Sender blocks until the send action is completed, not recv.
  - Receiver blocks until the recv. is completed.

More on blocking send

- int MPI_Send (void *buff, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - The address of data to be sent
  - # of data elements to be sent
  - Type of data elements to be sent
  - ID of process that should receive the message
  - A message tag to distinguish the message from other messages which may be sent to the same process.
    - Wild cards allowed, we can say MPI_ANY_TAG
  - A communication context capturing groups of processes working on the same sub-problem
    - By default MPI_COMM_WORLD captures the group of all processes.

Basic Blocking Communication (contd.)

- int MPI_Recv(void *buff, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - Receive contents of a variable (single or array) from specified PE within specified communicator
- Waits until a matching (on source and tag) message is received
- Source is rank in communicator specified by comm or MPI_ANY_SOURCE
- Receiving fewer than count occurrences of datatype is OK, but receiving more is an error
- The status field captures information about
  - Source , Tag, How many elements were actually received

Simple Sample Program

```
#include <mpi.h>
main( int argc, char *argv[]) {
  ........
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid == 0)
    { otherid = 1; myvalue = 14;}
  else
    { otherid = 0; myvalue = 25;}
  MPI_Send (&myvalue, 1,MPI_INT,otherid, 1, tag, MPI_COMM_WORLD);
  MPI_Recv (&othervalue, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
    MPI_COMM_WORLD, &status);
  printf(" process %d received %d\n", myid, othervalue);
  MPI_Finalize();
}
```

Another example

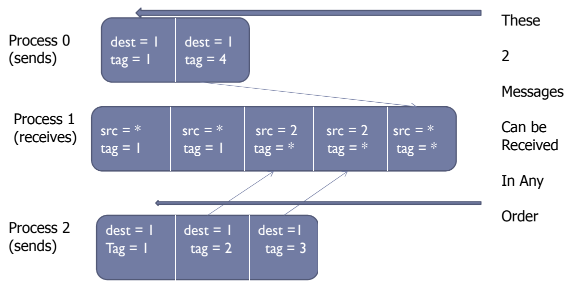
```
char msg[20]; int myrank, tag =99;
MPI_status status;
…
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0){
  strcpy(msg, "Hello there");
  MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else if (myrank == 1){
  MPI_Recv(msg, 20, MPI_CHAR,0, tag, MPI_COMM_WORLD, status);
}

…
```

status tells us how many elements were actually received!

Message ordering

- MPI_Send and MPI_Recv are blocking
  - MPI_Send blocks until send buffer can be reclaimed.
  - MPI_Recv blocks until receive is completed.
  - When MPI_Send returns we cannot guarantee that the receive has even started.
- If the sender sends 2 messages to same destination which match the same receive, the receive cannot match the 2nd msg, if the 1st msg is still pending.
- If a receiver posts 2 receives, and both match the same msg, the 2nd receive cannot get the msg, if the 1st receive is still pending.

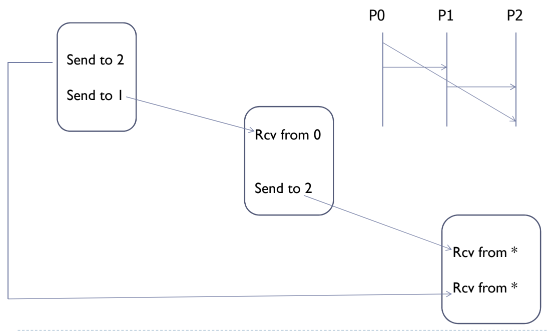
Order preservation in messages



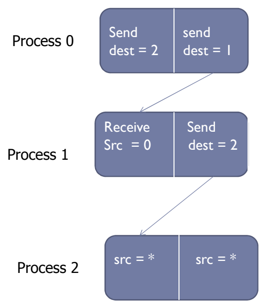
Order preservation in messages

- Messages are non-overtaking
  - Successive messages sent by a process p to another process q are ordered in sequence.
- Receives posted by a process are also ordered.
  - Each incoming message matches the first matching receive.
  - Matching defined by tags and source/destination.

Order preservation is not transitive



Order preservation is not transitive



Between any pair of processes, messages flow in order. However, across pairs of processes we cannot guarantee a consistent total order on the comm. events.

Communication delays can be arbitrary.

Wrapping up

- Blocking sends and receives
  - A blocking send completes when the send buffer can be re-used
  - A blocking receive completes, when the data is available in the receive buffer.
  - Each incoming message matches the first matching receive.
  - Order is preserved between any pair of processes.
  - Order preservation is however, not transitive.

Organization

- So Far
  - What is MPI
  - Entering and Exiting MPI
  - Creating multiple processes
  - Blocking Message Passing (point-to-point)
- Now
  - Non-blocking point to point communication
  - Collective communication

Non-blocking Communication

- MPI_Recv is blocking
  - It does not return until the message is received AND it is safe to modify the function arguments
- MPI_Send is blocking
  - It does not return until the message is buffered OR received by the destination processor, i.e., until it is safe to modify the function arguments
- Non-blocking primitives allows useful computation while waiting for send/receive to complete

Non-blocking Communication (Contd.)

- Non-blocking send or receive simply starts the operation
- A different function call will be required to complete the operation
- An additional request parameter is needed in non-blocking calls
- The parameter is used in subsequent operation to reference this message in order to complete the call

Nonblocking Functions

- int MPI_Isend(void *buff, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *req)
  - Begins a standard non-blocking message send
  - Returns before msg. is copied out of send buffer of sender process.
- int MPI_Irecv(void *buff, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *req)
  - Begins a standard non-blocking message receive.
  - Returns before message is received.

Nonblocking Functions (Contd.)

- int MPI_Wait (MPI_Request *request, MPI_Status *status)
  - Blocking call that completes MPI_Isend or MPI_Irecv function call
- int MPI_Test (MPI_Request *request,
  int *flag, MPI_Status *status)
  - Nonblocking call that tests the completion of MPI_Isend or MPI_Irecv function call
  - flag is TRUE if operation is complete

Request objects

- Allocated by MPI and reside in the MPI "system" memory
- It is opaque at the program levels
  - Structure of the object cannot be accessed.
- Only "system" may use the request object for identifying various properties of a "communication" operation
  - e.g. communication buffer associated with it.
  - e.g. to store information about the status of pending communication operations.

Multiple producers, one consumer

```
typedef struct{
   char data[MAXSIZE];
   int datasize;
   MPI_Request req;
} Buffer;
Buffer *buffer;
MPI_Status status;
…
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
/* producer code … */
/* consumer code … */
```

Producer code

```
if (rank != size – 1){
   /* producer allocates one buffer */
   buffer = (Buffer *) malloc(sizeof(Buffer));
   while(1) {
     /* fill buffer, and return # of bytes stored in the buffer */
        produce(buffer->data, &buffer->datasize);
        /* send the data*/
        MPI_Send(buffer->data,buffer->datasize,MPI_CHAR, size-1,
                      tag, comm)
     }
}
```

Consumer code

```
else{    /* rank == size – 1 */
   buffer = (Buffer*)malloc(sizeof(Buffer)*size-1));
   for (i=0; i<size-1;i++)   /* post a nonblocking receive from each producer */
       MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm, &(buffer[i].req));
   for (i=0;  ;  i=(i+1)%(size-1)) {
       MPI_Wait(&(buffer[i].req), &status);
       /* find number of bytes actually received */
       MPI_Get_count(&status, MPI_CHAR, &(buffer[i].datasize));
       /*consumer empties data buffer */
       consume(buffer[i].data,  buffer[i].datasize);
      /*  post new receive */
      MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm, &(buffer[i].req));
   }
}
```

More on the consumer

- Employs a strict round-robin discipline among the producers for receiving messages from them.
- Can we do first-come first-serve?
  - Consume a message from a producer who has produced its message.
  - Note that these messages may not be received at the consumer's end in exactly the same order in which they are produced !!

Alternative Consumer code

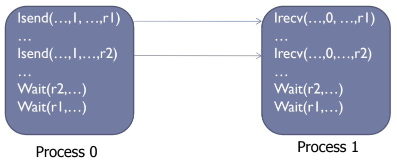
```
else{    /* rank == size – 1 */
    buffer = (Buffer*)malloc(sizeof(Buffer)*size-1));
    for (i=0; i<size-1;i++)    /* post a nonblocking receive from each producer */
        MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm, &(buffer[i].req));
    i= 0;
    while(1){
        for (flag=0; !flag;  i = (i+1)%(size-1)){
            MPI_Test(&(buffer[i].req),  & flag, &status);
        }
        MPI_Get_count(&status, MPI_CHAR, &(buffer[i].datasize));
        consume(buffer[i].data,  buffer[i].datasize);
        MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm, &(buffer[i].req));
    }
}
```

What is happening in this program?

Difference in the two consumers

- MPI_Wait is blocking
- MPI_Test is not
  - Usually employed for busy-waiting.
- Using MPI_Wait, the consumer employs a round-robin schedule among the producers.
- Using MPI_Test, the consumer employs a first-come-first-serve discipline among the producers.
  - HOW?

Message ordering in non-blocking communication



- Both Isend can complete before either receive.
- Still, first Isend matches with first Irecv
  - Second Isend matches with second Irecv

Message ordering

- The first Isend matches with first Irecv
- However, this does not fix the order of completion of operations.
- In non-blocking communication
  - Each send or recv is split into two parts
    - Start of the operation: Isend / Irecv
    - Completion of the operation.

Order preservation in non-blocking communication

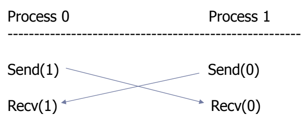
- Process 0   Process 1
- ---
- Isend(dest=1) $\longrightarrow$ Irecv(src=0)
- Isend(dest=1) $\longrightarrow$ Irecv(src=0)
- Waitall   Waitall

Suppose – (i) all the send/recv operations have the same tag
(ii) all sends happen before any receive.

Even then, order preservation rules ensure that the first send matches with the first receive (each incoming message matches the first matching receive).

Deadlocks in MPI

- An MPI implementation is not required to implement message buffering.



For blocking communication, the above communication pattern may cause the system to "hang".

Use non-blocking communication

- The deadlock can be avoided by a programming level solution instead
  - Use non-blocking communication primitives

```
Process 0                    Process 1
-------------------------------------------------------
ISend(1)                     ISend(0)
IRecv(1)                     IRecv(0)
Waitall                      Waitall
```

Multiple Completions

- The MPI_Wait completes one specific communication.
- We may want to complete some or all communications, rather than a specific communication.
  - MPI_Waitany(count, array_of_req, index, status)
    - Count is the list length
    - Array of request handled
    - Index of the request handle that completed
    - Status
  - MPI_Waitall(count, array_of_req, array_of_status)
    - All of the communication events should complete.

Consumer code with MPI_Wait

```
else{    /* rank == size – 1 */
    buffer = (Buffer*)malloc(sizeof(Buffer)*size-1));
    for (i=0; i<size-1;i++)    /* post a nonblocking receive from each producer */
        MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm, &(buffer[i].req));
    for (i=0;  ;  i=(i+1)%(size-1)) {
        MPI_Wait(&(buffer[i].req), &status);
        /* find number of bytes actually received */
        MPI_Get_count(&status, MPI_CHAR, &(buffer[i].datasize));
        /*consumer empties data buffer */
        consume(buffer[i].data,  buffer[i].datasize);
       /*  post new receive */
       MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm, &(buffer[i].req));
    }
}
```

Consumer code with MPI_Waitany

```
MPI_Request *req;
….
else{    /* rank == size – 1 */
    buffer = (Buffer*)malloc(sizeof(Buffer)*size-1));
    for (i=0; i<size-1;i++)    /* post a nonblocking receive from each producer */
        MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm, &(buffer[i].req));
    while (1){
        MPI_Waitany(size-1, req,  &i,  &status);
        MPI_Get_count(&status, MPI_CHAR, &(buffer[i].datasize));
        consume(buffer[i].data,  buffer[i].datasize);
       /*  post new receive */
        MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm, &req[i]);
    }
}
```

Consumer can repeatedly consume from 1 process, starving other processes.

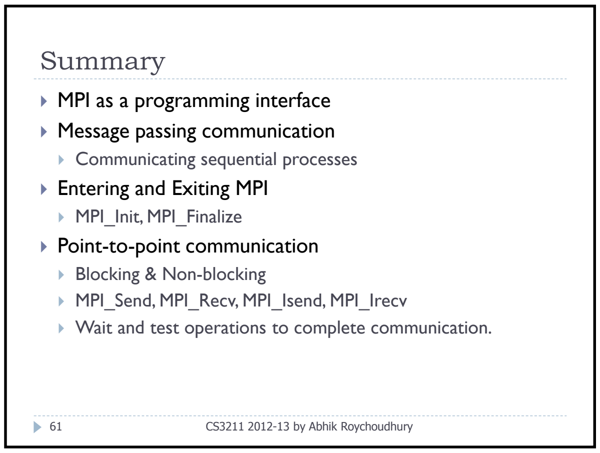
Multiple completions

- The most general version is MPI_Waitsome
  - MPI_Waitsome(incount, array_of_req, outcount,
  - array_of_indices, array_of_statuses)
  - Outcount is the number of completed communications.
  - array_of_indices are the indices of the completed operations
  - Similarly for array_of_statuses.
  - Waits until at least one of the pending communications is completed.
  - More flexible than MPI_Waitany and MPI_Waitall.

Consumer code with MPI_Waitsome

```
else{
    buffer = (Buffer*)malloc(sizeof(Buffer)*size-1));
    for (i=0; i<size-1;i++)
        MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm, &(buffer[i].req));
    while (1){
        MPI_Waitsome(size-1, req,  &count,  index, &status);
        for (i=0; i < count; i++){
            j= index[i];
            MPI_Get_count(&status[i], MPI_CHAR, &(buffer[j].datasize));
            consume(buffer[j].data,  buffer[j].datasize);
            MPI_Irecv(buffer[j].data, MAXSIZE, MPI_CHAR, j, tag, comm, &req[j]);
        }
    }
}
```

Starvation is avoided, receives all posted sends. Less comm. calls too.



Summary

- MPI as a programming interface
- Message passing communication
  - Communicating sequential processes
- Entering and Exiting MPI
  - MPI_Init, MPI_Finalize
- Point-to-point communication
  - Blocking & Non-blocking
  - MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv
  - Wait and test operations to complete communication.