Objectives:

To introduce you to the major concepts and ideas in parallel computing and its applications

To help you understand various models of parallelism (e.g., shared versus distributed memory models) and their strengths and limitations

To introduce you to basic “bottlenecks” encountered in parallel computing, e.g., I/O bottlenecks

To give you the basic knowledge to write simple MPI parallel programs
Part 1:
Basic Concepts and Terminology
What is Parallel Computing?

Traditionally software has been written for serial computations:

- To be run on a **single computer** having a **single Central Processing Unit (CPU)**
- A problem is broken into a discrete set of instructions
- Instructions are executed **one after another**
- **Only one instruction** can be executed at **any moment in time**
What is Parallel Computing?

In the simplest sense, parallel computing is the **simultaneous use of multiple compute resources** to solve a computational problem:

- To be run using **multiple CPUs**
- A problem is broken into discrete parts that can be **solved concurrently**
- Each part is further broken down to a series of instructions
- Instructions from each part **execute simultaneously on different CPUs**
What is Parallel Computing?

The compute resources can include:

- A single computer with multiple processors
- An arbitrary number of computers connected by a network
- A combination of both

The computational problem usually demonstrates characteristics such as the ability to be:

- Broken apart into discrete pieces of work that can be solved simultaneously
- Execute multiple program instructions at any moment in time
- Solved in less time with multiple compute resources than with a single compute resource
Why Use Parallel Computing?

Major reasons:

**Save time and/or money:** In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

**Solve larger problems:** Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

**Provide concurrency:** A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.

**Use of non-local resources:** Using compute resources on a wide area network, or even the Internet when local compute resources are scarce.
Applications of Parallel Computing:

Historically, parallel computing has been considered to be "the high end of computing", and has been used to model difficult scientific and engineering problems found in the real world. Some examples:

- Atmosphere, Earth, Environment, Space Weather
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Geology, Seismology
- Mechanical and Aerospace Engineering
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics
Applications of Parallel Computing:

Today, commercial applications provide an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. For example:

- Databases, data mining
- Oil exploration
- Web search engines, web based business services
- Medical imaging and diagnosis
- Pharmaceutical design
- Management of national and multi-national corporations
- Financial and economic modeling
- Advanced graphics and virtual reality, particularly in the entertainment industry
- Networked video and multi-media technologies
- Collaborative work environments
Classification of Parallel Computers:

Single Instruction, Single Data (SISD):
- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and even today, the most common type of computer
- Examples: older generation mainframes, minicomputers and workstations; most modern day PCs.
Classification of Parallel Computers:

Single Instruction, Multiple Data (SIMD):
- A type of parallel computer
- Single instruction: All processing units execute the same instruction at any given clock cycle
- Multiple data: Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units
Classification of Parallel Computers:

Multiple Instruction, Single Data (MISD):

- A single data stream is fed into multiple processing units
- Each processing unit operates on the data independently via independent instruction streams
- Some conceivable uses might be:
  multiple frequency filters operating on a single signal stream
  multiple cryptography algorithms attempting to crack a single coded message
Classification of Parallel Computers:

**Multiple Instruction, Multiple Data (MIMD):**
- **Currently, the most common type of parallel computer. Most modern computers fall into this category**
- Multiple Instruction: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs
**Some General Parallel Terminology:**

**Task**
A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

**Parallel Task**
A task that can be executed by multiple processors safely (yields correct results)

**Serial Execution**
Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.

**Parallel Execution**
Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.

**Pipelining**
Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing.

**Shared Memory**
From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

**Symmetric Multi-Processor (SMP)**
Hardware architecture where multiple processors share a single address space and access to all resources; shared memory computing.
Some General Parallel Terminology:

**Distributed Memory**
In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

**Communications**
Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

**Synchronization**
The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point. Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.
Some General Parallel Terminology:

Parallel Overhead
The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:

- Task start-up time
- Synchronizations
- Data communications
- Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
- Task termination time

Scalability
Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:

- Hardware - particularly memory-cpu bandwidths and network communications
- Application algorithm
- Parallel overhead related
- Characteristics of your specific application and coding

Supercomputing / High Performance Computing
Use of the world's fastest, largest machines to solve large problems.
Parallel Computer Memory Architectures:

Shared Memory:

General Characteristics:
- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: UMA and NUMA.
Parallel Computer Memory Architectures:

Shared Memory:

Advantages:
- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Disadvantages:
- Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
- Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.
Parallel Computer Memory Architectures:

Distributed Memory:

General Characteristics:
- Distributed memory systems require a communication network to connect interprocessor memory.
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.
Parallel Computer Memory Architectures:

Distributed Memory:

Advantages:
- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages:
- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access (NUMA) times
Parallel Computer Memory Architectures:

**Hybrid Distributed-Shared Memory:**

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.
Parallel Programming Models:

Shared Memory Model
- In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously.
- Various mechanisms may be used to control access to the shared memory.
- Advantage: There is no need to specify explicitly the communication of data between tasks. Program development can often be simplified.
- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality.

- OpenMP
- POSIX Threads (Pthreads)
Parallel Programming Models:

**Distributed memory**

- A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

- **Message Passing Interface (MPI)**
- **Parallel Virtual Machine (PVM)**
Elements of a “parallel” program

- High-level language
  - Fortran
  - C++

- Communication library
  - OpenMP
  - Pthreads
  - Shared Memory
  - MPI
  - PVM
  - Distributed Memory

- System Scripts
  - Commands for program execution
Part 2:

Going Parallel with MPI
What is Message Passing Interface (MPI)? Yet Another Library!

MPI is a library, not a language. It specifies the names, calling sequences and results of functions or subroutines to be called from C or Fortran programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran, C or C++ are compiled with ordinary compilers and linked with the MPI library. MPI is a specification, not a particular implementation. MPI programs should be able to run on all possible machines and run all MPI implementations without change. An MPI computation is a collection of processes communicating with messages.
MPI

• Pros:
  – Very portable
  – Requires no special compiler
  – Requires no special hardware but can make use of high performance hardware
  – Very flexible -- can handle just about any model of parallelism
  – No shared data!
  – Can download free libraries for your Linux PC!
  – Forces you to do things the "right way" in terms of decomposing your problem.

• Cons:
  – All-or-nothing parallelism (difficult to incrementally parallelize existing serial codes)
  – No shared data! Requires distributed data structures
  – Generally have to write more code
  – Partitioning operations on distributed arrays can be messy.
MPI

- Used to create parallel SPMD programs based on message passing
  - Normally the same program is running on several different processors
  - Processors communicate using message passing

- Typical methodology:

```plaintext
start job on n processors
  If rank=0 then
    do some work for process 0
  else
    do work for other processes
  end if
end job
```
Typical method: flowchart

- MPI initialization
  - rank = 0
    - Task 0
  - Other ranks
    - Task > 0
- MPI finish
**MPI Communicator:**

- A group of MPI processes with a name (context).
- Any process is identified by its rank. The rank is only meaningful within a particular communicator.
- By default communicator MPI_COMM_WORLD contains all the MPI processes.
- Mechanism to identify subset of processes.
- Promotes modular design of parallel libraries.

- Can create subsets of MPI_COMM_WORLD
- Processors within a communicator are assigned numbers (ranks) 0 to n-1
MPI: communicator

- MPI_INIT – defines "communicator" MPI_COMM_WORLD
- MPI_COMM_WORLD – defines participating processes
- RANK – labels processes
The 6 most important MPI routines:

- MPI_Init - initiate an MPI computation
- MPI_Finalize - terminate the MPI computation and clean up
- MPI_Comm_size - how many processes participate in a given MPI communicator?
- MPI_Comm_rank - which one am I? (A number between 0 and size-1.)
- MPI_Send - send a message to a particular process within an MPI communicator
- MPI_Recv - receive a message from a particular process within an MPI communicator
Startup and end up

- **MPI_INIT**
  - The first MPI call in any MPI process
  - Establishes MPI environment
  - One and only one call to MPI_INIT per process

- **MPI_FINALIZE**
  - Exiting from MPI
  - Cleans up state of MPI
  - The last call of an MPI process
MPI basic commands

- **C++:**
  1. `MPI_INIT(&argc, &argv)`
  2. `MPI_COMM_SIZE(MPI_COMM_WORLD,&nprocs)`
  3. `MPI_COMM_RANK(MPI_COMM_WORLD,&my_rank)`
  4. `MPI_FINALIZE()`

- **Fortran**
  1. `MPI_INIT(ierr)`
  2. `MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)`
  3. `MPI_COMM_RANK(MPI_COMM_WORLD,my_rank,ierr)`
  4. `MPI_FINALIZE(ierr)`
Using namespace std;
#include <mpi.h>
#include <iostream>
int main (int nargs, char* args[])
{
    int numprocs, my_rank;
    // MPI initializations
    MPI_Init (&nargs, &args);
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    cout << "Hello world, I have rank " << my_rank <<
    numprocs << endl;
    // End MPI
    MPI_Finalize ();
Parallel “Hello World” in Fortran

PROGRAM main

INCLUDE "mpif.h"
INTEGER ierr, my_rank, nprocs

! Initialize MPI
CALL MPI_INIT(ierr)
! Find this processor number
CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
! Find the number of processors
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

write(*,*) 'Hello from processor ', my_rank, ' out of ', nprocs

! Shut down MPI
CALL MPI_FINALIZE(ierr)

STOP
END

• Output:
  Hello from processor 0 out of 3
  Hello from processor 1 out of 3
  Hello from processor 2 out of 3
  Hello from processor 3 out of 3
MPI: Point-to-point communication

- Identification of source and destination by rank
- MPI call returns after completion of send/receive
MPI: command parameters

- Message = data + envelope

```
MPI_Send(startbuf, count, datatype, dest, tag, comm)
```

DATA

COMMAND

ENVELOPE
MPI Data types

- **MPI + Fortran 90**
  - MPI_CHAR
  - MPI_INT
  - MPI_REAL
  - MPI_DOUBLE_PRECISION
  - MPI_COMPLEX
  - MPI_LOGICAL
  - MPI_BYTE
  - MPI_PACKED

- **MPI + C++**
  - MPI_CHAR
  - MPI_INT
  - MPI_SHORT
  - MPI_LONG
  - MPI_UNSIGNED
  - MPI_UNSIGNED_SHORT
  - MPI_UNSIGNED_LONG
  - MPI_FLOAT
  - MPI_DOUBLE
  - MPI_LONG_DOUBLE
  - MPI_BYTE
  - MPI_PACKED
MPI send/receive messages

- Call `MPI_Send(buffer, count, datatype, destination, tag, communicator, ierr)`

  **Buffer**: The data array to be sent  
  **Count**: Length of data array  
  **Datatype**: Type of data, for example:  
    - `MPI_DOUBLE_PRECISION`, `MPI_INT`, etc  
  **Destination**: Destination processor number (within given communicator)  
  **Tag**: Message type (arbitrary integer)  
  **Communicator**: Your set of processors  
  **ierr**: Error return (Fortran only)
MPI send/receive messages

- **C**
  
  ```c
  MPI_Recv(&buffer, count, datatype, source, tag, communicator, &status);
  ```

- **Fortran**
  
  ```fortran
  Call MPI_RECV(buffer, count, datatype, source, tag, communicator, status, ierr)
  ```
MPI send/receive messages

rank 0

MPI_SEND → Post send → Wait for Recv → transfer → handshake → Do other work

rank > 0

Do some work → MPI_RECV → Post receive → transfer → handshake → Do other work
MPI implementation of trapezoidal rule:

\[ I = \int_a^b f(x)dx = h(f(a)/2 + f(a + h) + f(a + 2h) + \cdots + f(b - h) + f_b/2) \]

```c
C serial.f -- calculate definite integral using trapezoidal rule.
C
C The function f(x) is hardwired.
C Input: a, b, n.
C Output: estimate of integral from a to b of f(x)
C using n trapezoids.
C
PROGRAM serial
INCLUDE 'mpif.h'
real integral
real a
real b
integer n
real h
real x
integer i

C
real f
C
print *, 'Enter a, b, and n'
read *, a, b, n
C
h = (b-a)/n
integral = (f(a) + f(b))/2.0
x = a
```
do 100 i = 1, n-1  
  x = x + h  
  integral = integral + f(x)  
100 continue  
  integral = integral*h
C
print *, 'With n =', n, ' trapezoids, our estimate'  
print *, 'of the integral from ', a, ' to ', b, ' = ', integral
end
C
C*******************************************************************************
real function f(x)
real x
C Calculate f(x). Store calculation in return_val.
C
  f = x**x
  return
end
One approach to parallelizing this program is to simply split the interval \([a,b]\) up among the processes, and each process can estimate the integral of \(f(x)\) over its subinterval. In order to estimate the total integral, the processes’ local calculations are added.

Suppose there are \(p\) processes and \(n\) trapezoids, and, in order to simplify the discussion, also suppose that \(n\) is evenly divisible by \(p\). Then it is natural for the first process to calculate the area of the first \(n/p\) trapezoids, the second process to calculate the area of the next \(n/p\), etc. So process \(q\) will estimate the integral over the interval

\[
\left[ a + q \frac{nh}{p}, a + (q+1) \frac{nh}{p} \right]
\]

Thus each process needs the following information.

- The number of processes, \(p\).
- Its rank.
- The entire interval of integration, \([a,b]\).
- The number of subintervals, \(n\).
MPI: Collective communication

Source

0 → 1
0 → 2
0 → 3
0 → ...
n-1

MPI_COMM_WORLD

MPI message
Collective communication

- A more complex sequence of point-to-point calls
- Involve all the processes in communicator
- Called by all processes in communicator
- All routines block until they are locally complete
- Receive buffers must be exactly the right size
- No message tags are needed
Collective communication

- **Broadcast**

- **scatter**

- **gather**

- **all-to-all**

- **all gather**

- **Reduce**

- **Scan**
Broadcast

• MPI_Bcast(buffer, count, datatype, root, comm)
  - Broadcast the message of length count in buffer from the process root to all other processes in the group. All processes must call with same arguments.
Reduce

- MPI_Reduce(sbuf, rbuf, count, stype, op, root, comm)
  - Apply the reduction function op to the data of each process in the group (type stype in sbuf) and store the result in rbuf on the root process. op can be a pre-defined function, or defined by the user.
Predefined Reduce Operations

<table>
<thead>
<tr>
<th>MPI_NAME</th>
<th>FUNCTION</th>
<th>MPI_NAME</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
<td>MPI_MAXLOC</td>
<td>Maximum and location</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
<td>MPI_MINLOC</td>
<td>Minimum and Location</td>
</tr>
</tbody>
</table>
program bcast

include 'mpif.h'

integer :: ibuf, nprocs, my_rank, ierr
integer :: isend, irecv

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)

if (my_rank == 0) then
  ibuf = 1000
else
  ibuf = 0
end if

call MPI_BCAST(ibuf, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
write(*,*) 'ibuf = ', ibuf

ibuf = ibuf + my_rank
write(*,*) 'ibuf = ', ibuf

call MPI_REDUCE(ibuf, irecv, 1, MPI_INTEGER, MPI_SUM, 0, MPI_COMM_WORLD, ierr)

if (my_rank == 0) write(*,*) 'irecv = ', irecv

call MPI_FINALIZE(ierr)
end
reduce.f -- Parallel Trapezoidal Rule. Uses 3 calls to MPI_Bcast to distribute input. Also uses MPI_Reduce to compute final sum.

Input:
  a, b: limits of integration.
  n: number of trapezoids.
Output: Estimate of the integral from a to b of f(x) using the trapezoidal rule and n trapezoids.

Notes:
  1. f(x) is hardwired.
  2. Assumes number of processes (p) evenly divides

PROGRAM getdata
INCLUDE 'mpif.h'
integer my_rank
integer p
real a
real b
integer n
real h
real local_a
real local_b
integer local_n

my calculation
real integral
real total
integer source
integer dest
integer tag
integer status(MPI_STATUS_SIZE)
integer ierr
data dest, tag /0, 0/

real Trap
Let the system do what it needs to start up MPI
   call MPI_INIT( ierr )

Get my process rank
   call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr )

Find out how many processes are being used
   call MPI_COMM_SIZE(MPI_COMM_WORLD, p, ierr )

call Get_data2( a, b, n, my_rank)

   h = (b-a)/n
   local_n = n/p

Length of each process' interval of
integration = local_n*h. So my interval
starts at:
   local_a = a + my_rank*local_n*h
   local_b = local_a + local_n*h
   integral = Trap(local_a, local_b, local_n, h)

Add up the integrals calculated by each process
   call MPI_REDUCE(integral, total, 1, MPI_REAL, +
                      MPI_SUM, 0, MPI_COMM_WORLD, ierr)

Print the result
   if (my_rank .EQ. 0) then
      print *, 'With n = ', n, ' trapezoids, our estimate'
      print *, 'of the integral from ', a, ' to ', b, ' = ', total
   endif

Shut down MPI
   call MPI_FINALIZE(ierr)
end
C

subroutine Get_data2(a, b, n, my_rank)
real a
real b
integer n
integer my_rank
integer ierr
include 'mpif.h'

C
C
if (my_rank .EQ. 0) then
   print *, 'Enter a, b, and n'
   read *, a, b, n
endif

C
C
call MPI_BCAST(a, 1, MPI_REAL , 0, MPI_COMM_WORLD, ierr )
call MPI_BCAST(b, 1, MPI_REAL , 0, MPI_COMM_WORLD, ierr )
call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr )
return
end

C
C
C *************************************************************************
C ******************************************************************************
    real function Trap(local_a, local_b, local_n, h)
    real   local_a
    real   local_b
    integer  local_n
    real   h

    real integral
    real x
    integer  i

    real f

    integral = (f(local_a) + f(local_b))/2.0
    x = local_a
    do 100 i = 1 ,local_n-1
       x = x + h
       integral = integral + f(x)
    100   continue
    Trap = integral*h
    return
end

C C

C ******************************************************************************
    real function f( x)
    real x

C Calculate f(x).
C Store calculation in return_val.
    f = x*x
    return
end

C C
MPI Barrier

- Objective: synchronize all processes
- A node calling it will be blocked until all nodes have called it
- C:
  - ierr = MPI_BARRIER(comm)
- Fortran:
  - call MPI_BARRIER(comm, ierr)
MPI Scatter

- `MPI_Scatter(sbuf, scount, stype, rbuf, rcount, rtype, root, comm, ierror)`
  - The number of elements sent to each process is the same (scount). The first scount elements are sent to process 0, the next to process 1, and so on.
MPI Gather

- MPI_Gather(sbuf, scount, stype, rbuf, rcount, rtype, root, comm, ierror)
  - Collects individual messages from each process to the root process and stores them in rank order.
Acknowledgements

The material presented here is based on the following (major) sources:

(1) [http://www.llnl.gov/computing/hpc/training](http://www.llnl.gov/computing/hpc/training)

(2) [http://www-rohan.sdsu.edu/~terane/](http://www-rohan.sdsu.edu/~terane/) Thanks to Edgar Teran for some of the slides