Message-Passing Environments & Systems

- Origins of Cluster Computing or Commodity Supercomputing:
  - Limitations of Homogeneous Supercomputing Systems
  - Heterogeneous Computing (HC)
  - Broad Issues in Heterogeneous Computing:
    - Analytical Benchmarking
    - Code-type Profiling
    - HC Task Matching and Scheduling (Mapping)
    - Heterogeneous Computing System Interconnect Requirements
    - Development of Portable Message Passing Environments for HC

HC Reference Papers on course web page

- Message-Passing Parallel Systems:
    - Commercial Massively Parallel Processor Systems (MPPs) Vs. Clusters
- Message-Passing Programming & Environments
    - Process Creation In Message Passing
    - Message-Passing Modes
- Overview of Message Passing Interface (MPI)
    - History
    - Major Features: Static Process Creation, Rank ...
    - Compiling and Running MPI Programs
    - MPI Indispensable Functions
    - MPI Examples

Version 1.2

Parallel Programming: chapter 2, Appendix A.
MPI References/resources in course web page and mps.ce.rit.edu
(Including a link to html version of the book: MPI: The Complete Reference)

Origins of Cluster Computing:

Limitations of Homogeneous Supercomputing Systems

- Traditional homogeneous parallel supercomputing system architectures were characterized by:

  1. Support of a single homogeneous mode of parallelism exploited at a specific suitable task grain size: Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), and data parallel/vector processing.
  2. No separation between parallel programming model/architecture:
     - Parallel programming environments were developed specifically for a given target architecture and not compatible with other parallel architectures.

e.g no communication abstraction

- Such systems perform well when the application contains a single mode of parallelism that matches the mode (and grain size) supported by the system.

- In reality, many supercomputing/HPC applications have tasks with different and diverse modes of parallelism/grain size requirements.
  – When such applications execute on a homogeneous system, the machine spends most of the time executing subtasks for which it is not well suited.
  – The outcome is that only a small fraction of the peak performance of the machine is achieved for such applications.

Example application with diverse computational mode/grain size requirements:

- Image understanding is an example application that requires different types of parallelism that can be exploited at drastically different task grain sizes.

Origins of Cluster Computing:

Heterogeneous Computing (HC)

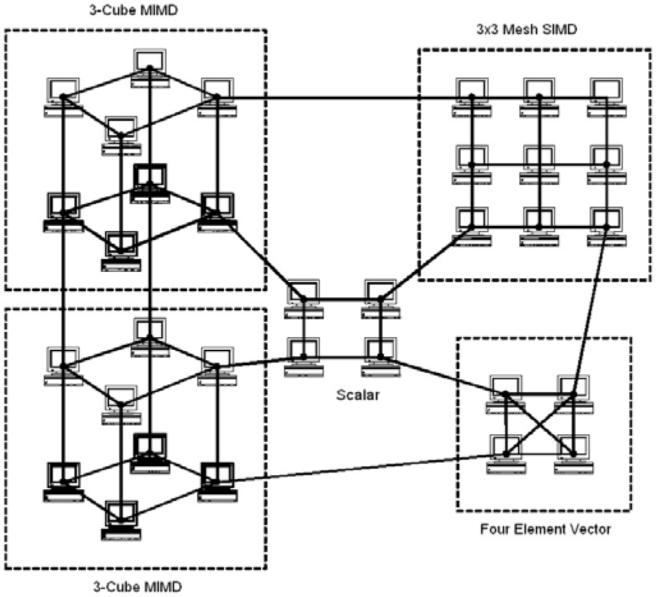
- Heterogeneous Computing (HC), addressees the issue of computational mode homogeneity in supercomputers by:
  - Effectively utilizing a heterogeneous suite of high-performance autonomous machines that differ in both speed and modes of parallelism supported to optimally meet the demands of large tasks with diverse computational requirements.

HC Definition

HC Requirements

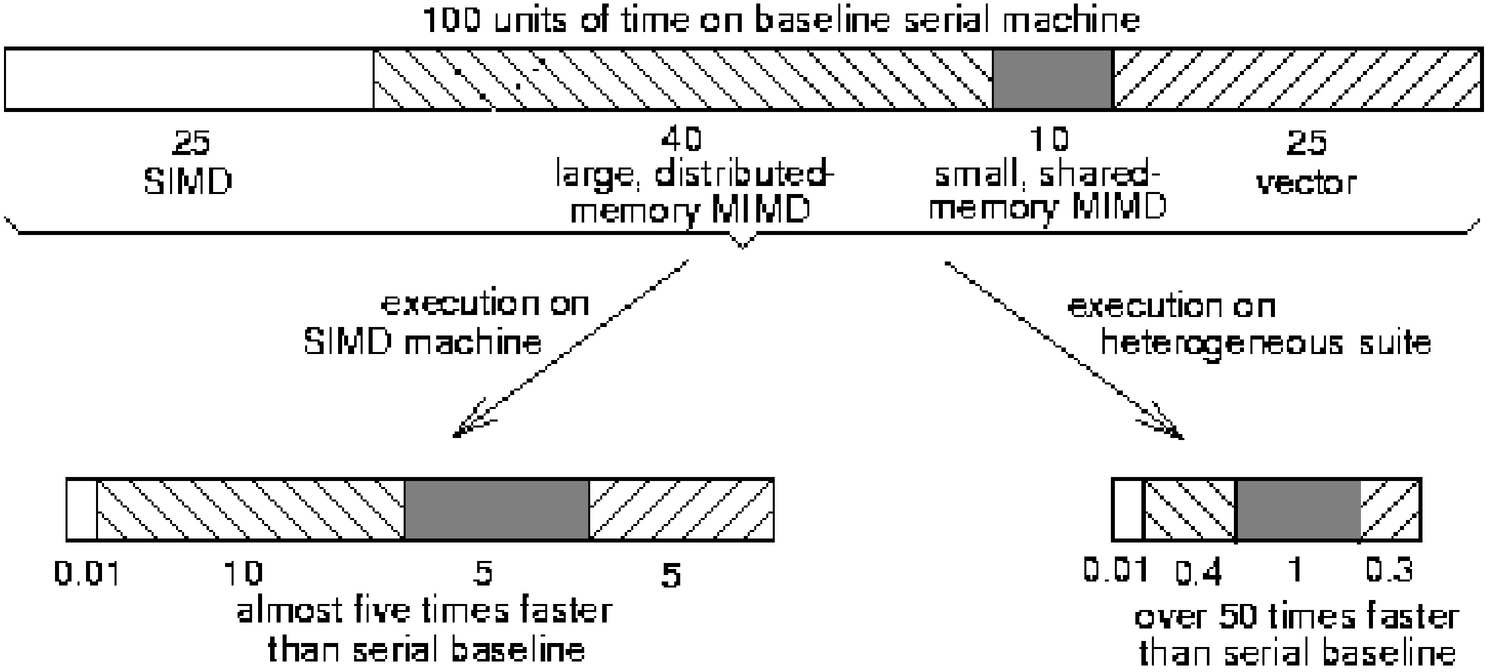
1- A network with high-bandwidth low-latency interconnects + 2- platform-independent (portable) message passing programming environments handle the intercommunication between each of the machines.

  - Heterogeneous computing is often referred to as Heterogeneous SuperComputing (HSC) reflecting the fact that the collection of machines used are usually supercomputers.



AKA: Network Heterogeneous SuperComputing (NHSC)

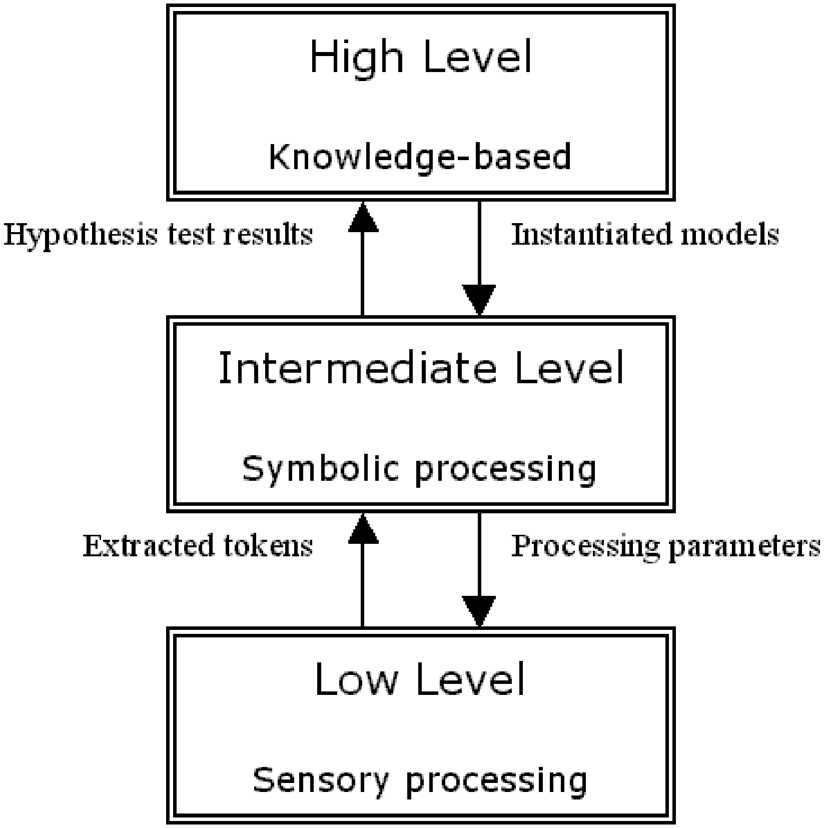
Motivation For Heterogeneous Computing



- **Hypothetical example of the advantage of using a heterogeneous suite of machines, where the heterogeneous suite time includes inter-machine communication overhead. Not drawn to scale.**

A Heterogeneous Computing Driving Application Example: Image Understanding Processing Levels

Coarse-grain Tasks

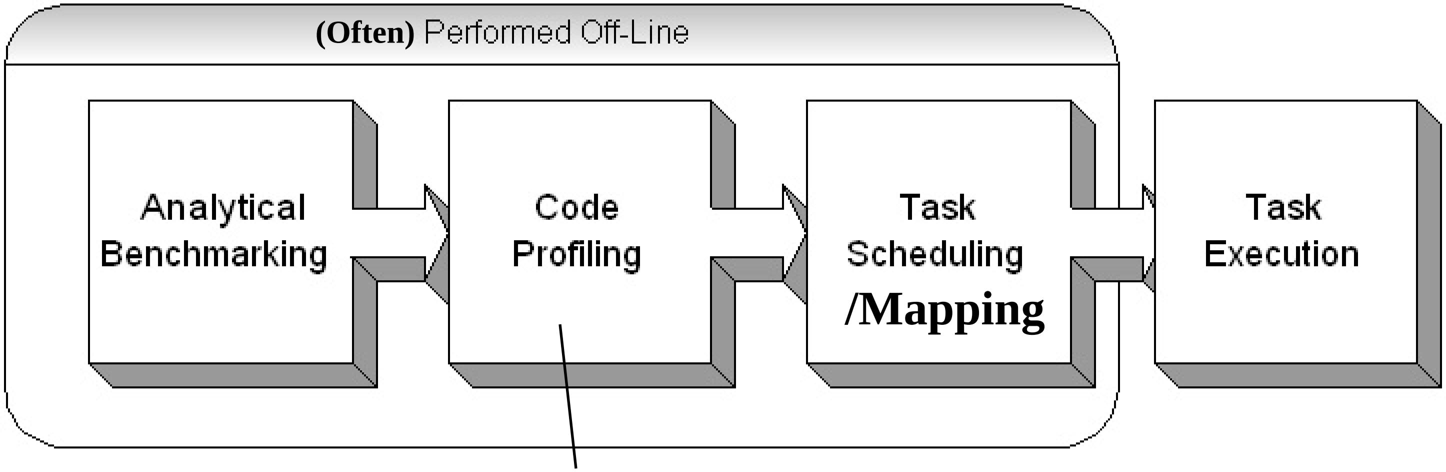


Fine-grain Tasks:
Highest degree of parallelism (DOP)
Type of parallelism: Data Parallelism

- Highest Level (Knowledge Processing):
  - Uses the data from the previous levels to infer semantic attributes of an image
  - Requires coarse-grained loosely coupled MIMD machines.
- Intermediate Level (Symbolic Processing):
  - Grouping and organization of features previously extracted
  - Communication is irregular, parallelism decreases as features are grouped
  - Best suited to medium-grained MIMD machines
- Lowest Level (Sensory Processing):
  - Consists of pixel-based operators and pixel subset operators such as edge detection
  - Highest amount of data parallelism
  - Best suited to mesh connected SIMD machines/vector architectures.

e.g Feature Extraction

Steps of Application Processing in Heterogeneous Computing

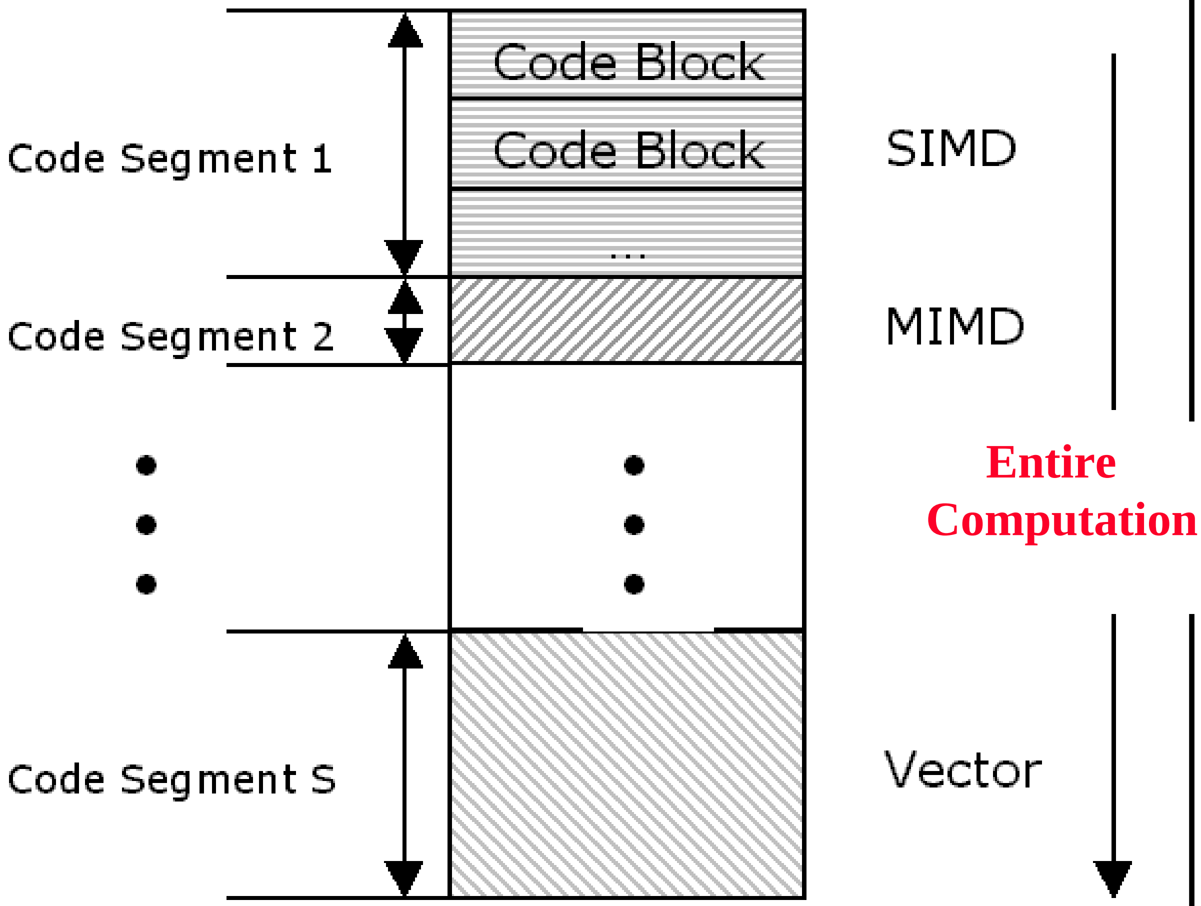


Includes decomposition/assignment (partitioning)

- **Analytical Benchmarking:**
  - This step provides a measure of how well a given machine is able to perform when running a certain type of code. e.g of a given type of parallelism/task grain size
  - Why? This is required in HC to determine which types of code should be mapped to which machines.
- **Code-type Profiling:** *includes decomposition/assignment (partitioning)*
  - Used to determine the type and fraction of processing modes of parallelism that exist in each program segment (task).
  - Why? Needed so that an attempt can be made to match each code segment (task) with the most efficient machine. in mapping step

Code-Type Profiling Example

What types/modes of parallelism are present in the computation?



- Example results from the code-type profiling of a program.
- The program is broken into S segments, each of which contains embedded homogeneous parallelism.

HC Task Matching and Scheduling (Mapping)

Tasks/processes → Machines/processors

- Task matching involves assigning a task to a suitable machine.
- Task scheduling on the assigned machine determines the order of execution of that task.
- Goal of mapping is to maximize performance by assigning code-types to the best suited machine while taking into account the costs of the mapping including computation and communication costs based on information obtained from analytical benchmarking, code-type profiling and possibly system workload.
- The problem of finding optimal mapping has been shown in general to be NP-complete even for homogeneous environments. (Non-Polynomial complexity) Thus
- For this reason, the development of heuristic mapping and scheduling techniques that aim to achieve "good" sub-optimal mappings is an active area of research resulting in a large number of heuristic mapping algorithms for HC.
- Two different types of mapping heuristics for HC have been proposed, static or dynamic.

Matching + Scheduling = Mapping

HC Task Matching and Scheduling (Mapping)

Tasks/processes → Machines/processors

Static Mapping Heuristics:

At compilation time (off-line)

- **Most such heuristic algorithms developed for HC are static and assume the ETC (expected time to compute) for every task on every machine to be known from code-type profiling and analytical benchmarking and not change at runtime.**
- **In addition many such heuristics assume large independent or meta-tasks that have no (or few) data dependencies .**
- **Even with these assumptions static heuristics have proven to be effective for many HC applications.**

Dynamic Mapping Heuristics:

At runtime (on-line)

- **Mapping is performed on-line (at runtime) taking into account current system workload (loading).**
- **Research on this type of heuristics for HC is fairly recent and is motivated by utilizing the heterogeneous computing system for real-time applications.**

Origins of Cluster Computing:

Heterogeneous Computing System Interconnect Requirements

- In order to realize the performance improvements offered by heterogeneous computing, communication costs must be minimized.
- The interconnection medium must be able to provide high bandwidth (multiple gigabits per second per link) at a very low latency.
- The system interconnect performance must scale well as the the number machines connected is increased:

  e.g scalable network/system interconnects

  - Proportional increase of total bandwidth and slow increase in latency
- It must also overcome current deficiencies such as the high overheads incurred during context switches, executing high-level protocols on each machine, or managing large amounts of packets.
- While the use of Ethernet-based LANs has become commonplace, these types of network interconnects are not well suited to heterogeneous supercomputers (high latency).
- This requirement of HC led to the development of cost-effective scalable system area networks (SANS) that provide the required high bandwidth, low latency, and low protocol overheads including Myrinet and Dolphin SCI interconnects.

1

- These system interconnects developed originally for HC, currently form the main interconnects in high performance cluster computing.

An important factor that helped give rise to cluster computing

Origins of Cluster Computing:

Development of Portable Message Passing Environments for HC

Platform-Independent

Message Passing APIs
(Application Programming Interfaces)

- Originally parallel programming environments were developed specifically for a given target architecture and thus were not compatible with other parallel architectures (e.g . Suite of heterogeneous machines for HC)
- Since the suite of machines in a heterogeneous computing system are loosely coupled and do not share memory, communication between the cooperating tasks must be achieved by exchanging messages over the network (message passing) using message passing environments that must be compatible with all the machines/platforms present in the heterogeneous suite of machines.
- This requirement led to the development of a number of portable, platform-independent message-passing programming environments or APIs that provide source-code portability across platforms.
- This also contributed to the separation of parallel programming models and parallel architectures (Communication Architecture/Abstraction).
- Parallel Virtual Machine (PVM), and Message Passing Interface (MPI) are the most widely-used of these environments.

2

- This also played a major role in making cluster computing a reality.

Commodity Supercomputing:

Cluster Computing

- The research in heterogeneous supercomputing led to the development of: 1- high-speed system area networks and 2- portable message passing APIs, and environments.
- These developments in conjunction with the impressive performance improvements and low cost of commercial general-purpose microprocessors led to the current trend in high-performance parallel computing of moving away from expensive specialized traditional supercomputing platforms to cluster computing that utilizes cheaper, general purpose systems consisting of loosely coupled commodity of-the-shelf (COTS) components.
- Such clusters are commonly known as Beowulf clusters (Linux used) and are comprised of three components:
  - 1 ▪ Computing Nodes: Each low-cost computing node is usually a small Symmetric Multi-Processor (SMP) system that utilizes COTS components including commercial General-Purpose Processors (GPPs) with no custom components.
  - 2 ▪ System Interconnect: Utilize COTS Ethernet-based or system area interconnects including Myrinet and Dolphin SCI interconnects originally developed for HSC.
  - 3 ▪ System Software and Programming Environments: Such clusters usually run an open-source royalty-free version of UNIX (Linux being the de-facto standard). The message-passing environments (PVM , MPI), developed originally for heterogeneous supercomputing systems, provide portable message-passing communication primitives.

Non-Dedicated Cluster = Network of Workstations (NOW)

APIs = Application Programming Interfaces

Message-Passing Parallel Systems:
Commercial Massively Parallel Processor Systems (MPPs) Vs. Clusters

Message Passing

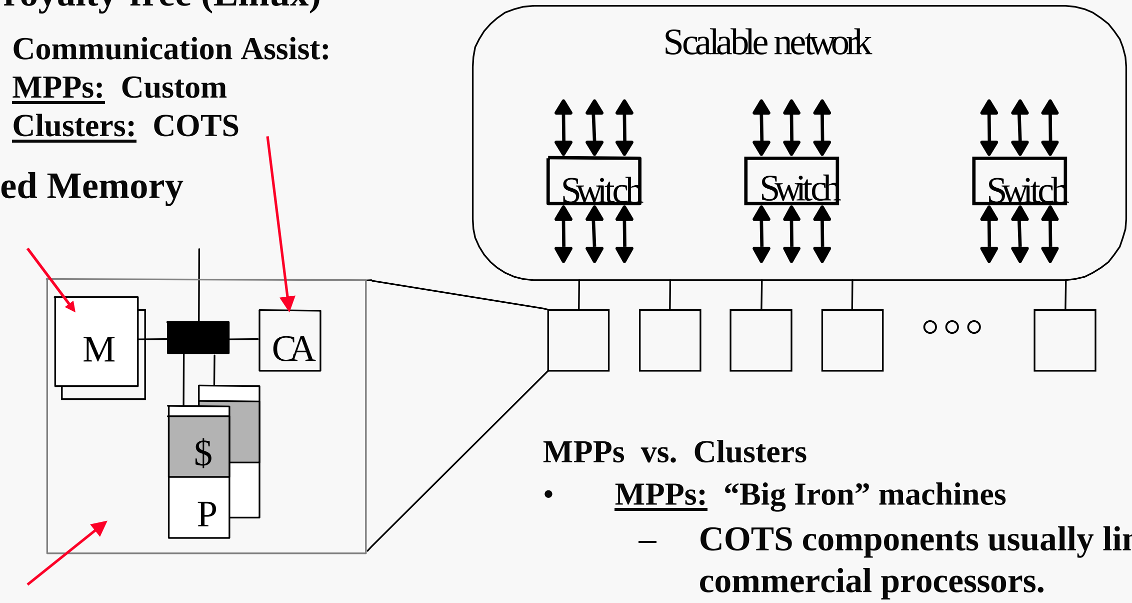
Operating system?
<u>MPPs:</u> Proprietary
<u>Clusters:</u> royalty-free (Linux)

Communication Assist:
<u>MPPs:</u> Custom
<u>Clusters:</u> COTS

Distributed Memory

Parallel Programming:

Between nodes: Message passing using PVM, MPI
In SMP nodes: Multithreading using Pthreads, OpenMP (SAS)



<u>Scalable Network:</u>
Low latency
High bandwidth
<u>MPPs:</u> Custom
<u>Clusters:</u> COTS
- •Gigabit Ethernet
- •System Area Networks (SANS)
  - • ATM
  - • Myrinet
  - • SCI
  - • FDDI
  - • HIPPI ...

MPPs vs. Clusters

- <u>MPPs:</u> "Big Iron" machines
  - COTS components usually limited to using commercial processors.
  - High system cost
- <u>Clusters:</u> Commodity Supercomputing
  - COTS components used for all system components.
  - Lower cost than MPP solutions

Node: O(10) SMP
<u>MPPs:</u> Custom node
<u>Clusters:</u> COTS node (workstations or PCs system boards)

Custom-designed CPU?
<u>MPPs:</u> Custom or commodity (mostly)
<u>Clusters:</u> commodity General Purpose Processors (GPPs)

Message-Passing Programming

- Deals with parallel programming by passing messages among processing nodes and processes.
- A number of message passing environments have been created for specific parallel architectures with most of the ideas developed merged into the portable (platform-independent) PVM and MPI standards.
- **Parallel Virtual Machine(PVM): (development started 1989, ended 1998?)**
  - Originally developed to enable a heterogeneous network of UNIX computers to be used as a large-scale message-passing parallel computer.
  - Using PVM, a virtual machine, a set of fully connected nodes is constructed with dynamic process creation and management. Each node can be a uniprocessor or a parallel computer.
  - PVM provides a portable self-contained software system with library routines (API) to support interprocess communication and other functions.
- **Message-Passing Interface (MPI): 1994**

Development still active

  - A standard specification for a library of message-passing functions (API) developed by the MPI Forum.

Implementation of API not specified by standard

  - Achieves portability using public-domain platform-independent message-passing library.
  - Not self-contained; relies on underlying platform for process management.

Both: Portable (Platform-independent) Message-Passing APIs (above communication abstraction)

Process Creation In Message Passing

Possible methods of generating (creating) processes:

1. Static process creation PVM, MPI 1.2 (used in this course)

- In static process creation, all the processes are specified before the program is executed (or at start of execution, MPI 1.2), and the system will execute a fixed number of processes.
- The programmer usually explicitly identifies the number of processes.

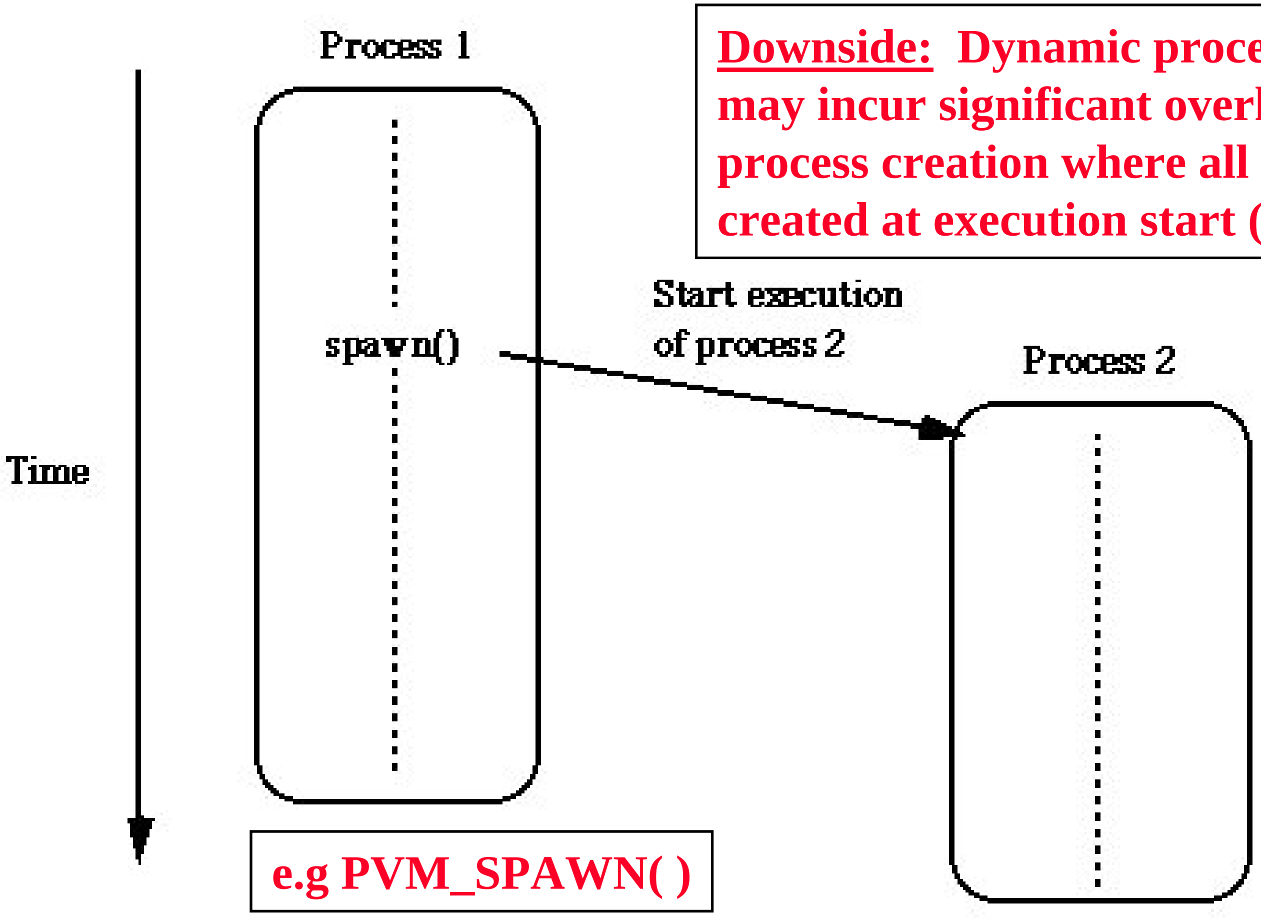
2. Dynamic process creation. PVM, MPI 2.0

- In dynamic process creation, processes can be created and started for execution during the execution of the main program using process creation constructs or system calls; processes can also be destroyed.
- Process creation and destruction might be done conditionally.
- The number of processes may vary during execution.
- Clearly dynamic process creation (and destruction) is a more powerful technique than static process creation, but it does incur very significant process-creation overheads when the processes are created dynamically.

Disadvantage

Dynamic Process Creation In Message Passing

Downside: Dynamic process creation may incur significant overheads vs. static process creation where all processes are created at execution start (e.g as in MPI 1.2)



e.g PVM_SPAWN( )

Figure 2.1 Spawning a process

Not supported by MPI 1.2 (used in this course)
Supported by PVM, MPI 2.0

i.e No MPI_Spawn ( ) in MPI 1.2

Generic Message-Passing Routines

(Point-to-Point Communication)

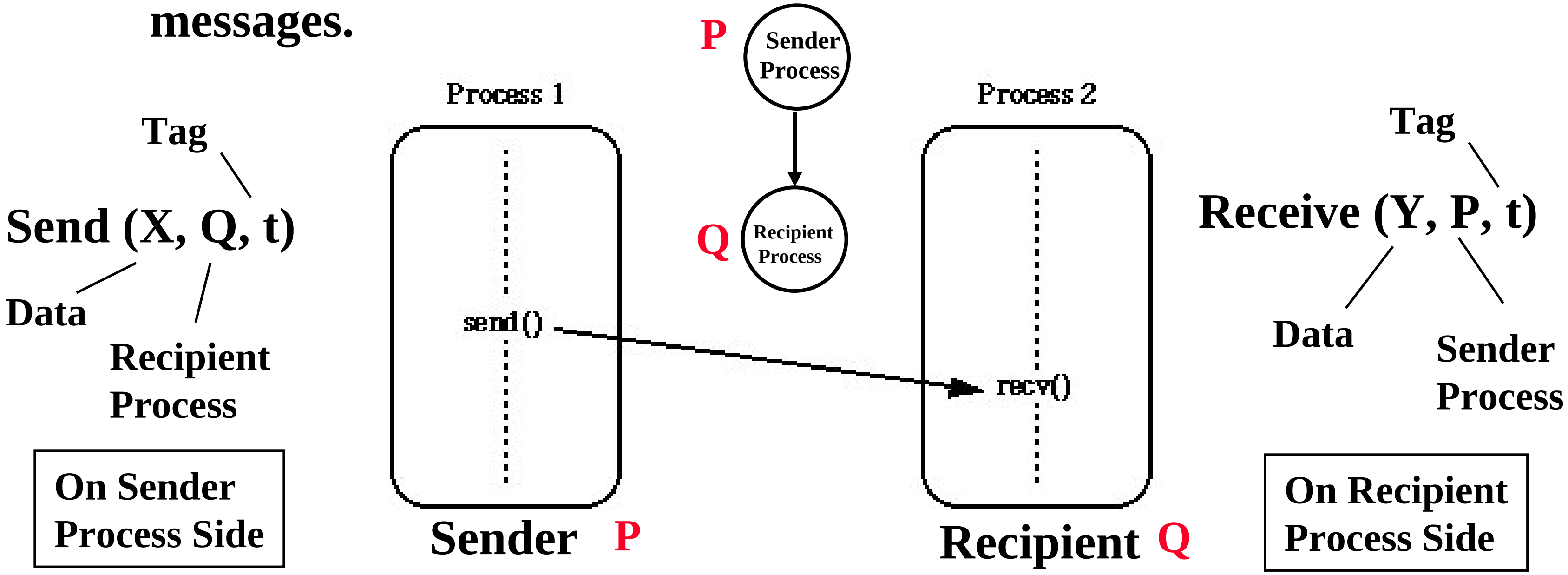
- Send and receive message-passing procedure/system calls often have the form:

  send(parameters)

  recv(parameters)

  Communication Explicit
  Point-to-point Synchronization/ordering Implicit

  – where the parameters identify the <u>source</u> and <u>destination</u> processes, and the <u>data</u> sent/received and a <u>message tag</u> to match messages.



Send (X, Q, t)

Tag

Data

Recipient Process

On Sender Process Side

Receive (Y, P, t)

Tag

Data

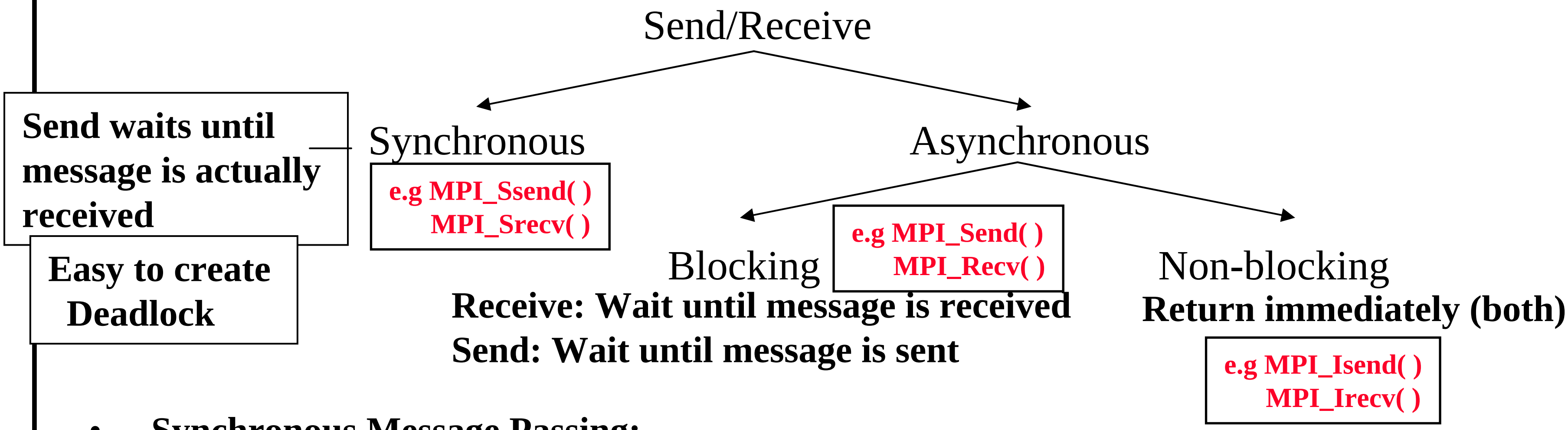
Sender Process

On Recipient Process Side

Figure 2.2 Passing a message between processes using send() and recv() system calls

Send/Receive Matched Pair

Message-Passing Modes: **Send and Receive Alternatives**



Send/Receive

Synchronous

e.g MPI_Ssend( )
MPI_Srecv( )

Send waits until message is actually received

Easy to create Deadlock

Asynchronous

Blocking

e.g MPI_Send( )
MPI_Recv( )

Receive: Wait until message is received
Send: Wait until message is sent

Non-blocking

Return immediately (both)

e.g MPI_Isend( )
MPI_Irecv( )

- **Synchronous Message Passing:**

  Process X executing a synchronous send to process Y has to wait until process Y has executed a synchronous receive from X. Easy to deadlock.

- **Asynchronous Message Passing:**
- **Blocking Send/Receive:**

  Blocking send/receive: Return when locally complete (both)

  A blocking send is executed when a process reaches it without waiting for a corresponding receive. Blocking send returns when the message is sent. A blocking receive is executed when a process reaches it and only returns after the message has been received.

  Most commonly used mode

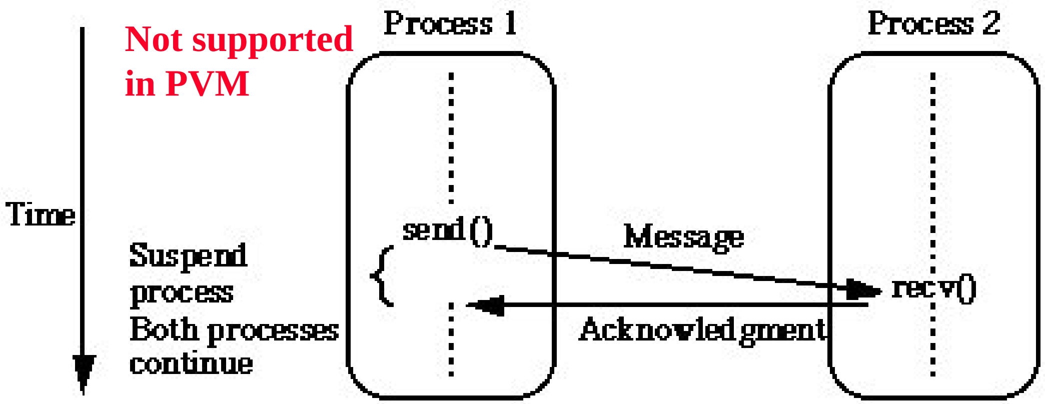
- **Non-Blocking Send/Receive:**

  A non-blocking send is executed when reached by the process without waiting for a corresponding receive. A non-blocking receive is executed when a process reaches it without waiting for a corresponding send. Both return immediately.

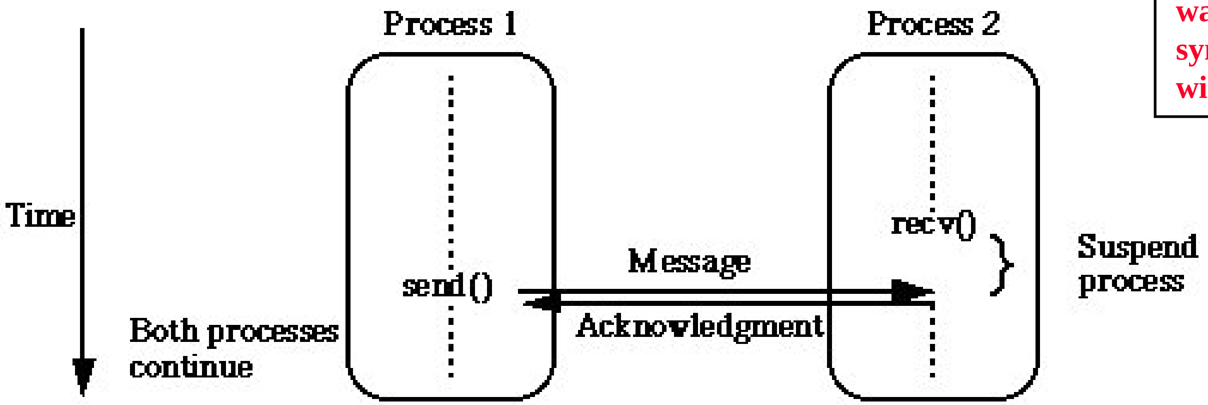
Repeated here from last lecture #4

Synchronous send( ) and recv( ) Calls

Not supported in PVM



(a) When send() occurs before recv()



(b) When rev() occurs before send()

Problem:
Easy to deadlock

P1 Send → Wait

P2 ← Send Wait

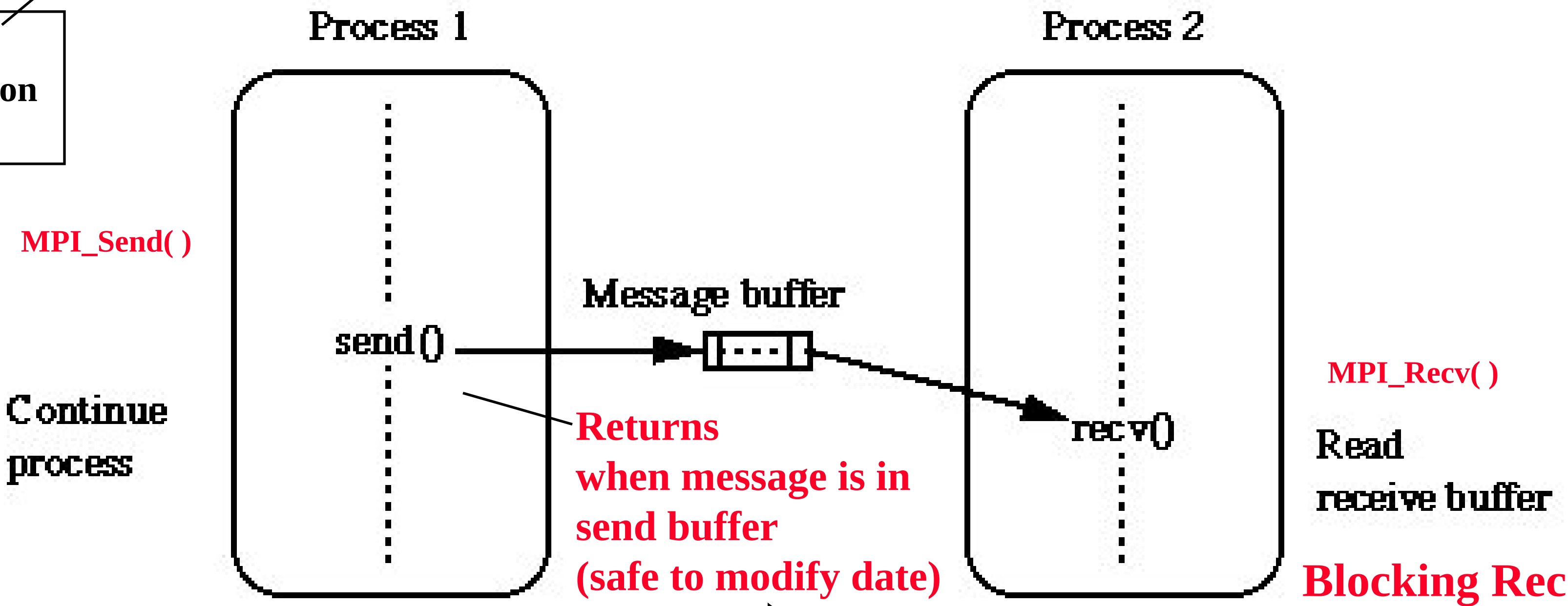
Using synchronous send both P1 and P2 will deadlock since both are waiting for corresponding synchronous receives that will not be reached

Synchronous Send/Receive in MPI: MPI_Ssend(…) MPI_Srecv(…)

Asynchronous Message Passing:

Blocking send( ) and recv( ) System Calls

Most Common Type



Returns when message is in send buffer (safe to modify date)

Blocking Receive: Returns when message is received

Blocking send/receive: Return when locally complete (both)

In MPI:

Blocking

Blocking or Basic Send: Return when it's safe for sending process to modify sent data.
Blocking or Basic Receive: Return when message is received

e.g MPI_Send( )
MPI_Recv( )

Non-Blocking or Immediate

Non-Blocking or Immediate Send: Return immediately. It may not be safe to modify data by sending process.
Non-Blocking or Immediate Receive: Return even if message is not received.

e.g MPI_Isend( )
MPI_Irecv( )

Message Passing Interface (MPI)

- **MPI, the *Message Passing Interface*, is a portable library of message passing functions, and a software standard developed by the MPI Forum to make use of the most attractive features of existing message passing systems for parallel programming.**
- **The public release of version 1.0 of MPI was made in June 1994 followed by version 1.1 in June 1995 and and shortly after with version 1.2 which mainly included corrections and specification clarifications.**
- **An MPI 1.0 process consists of a C or Fortran 77 program which communicates with other MPI processes by calling MPI routines. The MPI routines provide the programmer with a consistent interface across a wide variety of different platforms.**
- **MPI Version 2.0, a major update of MPI, was released July 1997 adding among other features support for dynamic process creation, one-sided communication and bindings for Fortran 90 and C++. MPI 2.0 features are not covered here.**

e.g MPI_Get, MPI_Put

- **Several commercial and free implementations of MPI 1.2 exist Most widely used free implementations of MPI 1.2 :**
  - **LAM-MPI : Developed at University of Notre Dame , http://www.lam-mpi.org/**
  - **MPICH-1: Developed at Argonne National Laboratory**

    http://www-unix.mcs.anl.gov/mpi/mpich/

    - **(MPICH 1.2.7 is the MPI implementation installed on the CE cluster).**

MPICH-2
is an MPI 2 implementation

rocks.ce.rit.edu

(PP Chapter 2, Appendix A, MPI References in course web page and mps.ce.rit.edu)
(Including html version of the book: MPI: The Complete Reference)

Major Features of MPI 1.2

Standard includes 125 functions to provide:

- Point-to-point message passing
- Group/Collective communication
- Support for process groups
- Support for communication contexts   Communicators
- Support for application topologies
- Environmental inquiry routines
- Profiling interface

MPICH 1.2.7 is the MPI 1.2 implementation installed on the CE cluster

rocks.ce.rit.edu

Compiling and Running MPI Programs

- **To compile MPI C programs use:**

  mpicc [linking flags] program_name.c -o program_name

  Ex: mpicc hello.c -o hello

- **To run a compiled MPI C program use:**

  mpirun -np <number of processes> [mpirun_options] -machinefile < machinefile> <program name and arguments>

  optional

  The machinefile contains a list of the machines on which you want your MPI programs to run.

  EX: **mpirun -np 4 -machinefile .rhosts hello**

  starts four processes on the top four machines from machinefile .rhosts all running the program hello

On CE Cluster a default machine file is already set up

MPI Static Process Creation & Process Rank

- MPI 1.2 ***does not support dynamic process creation***, i.e. one cannot spawn a process from within a process as can be done with pvm:
  - All processes must be started together at the beginning of the computation. Ex: mpirun -np 4 -machinefile .rhosts hello
  - There is no equivalent to the pvm **pvm_spawn( )** call.
  - This restriction leads to the direct support of MPI for single program-multiple data (**SPMD**) model of computation where each process has the same executable code.

i.e execution

i.e No MPI_Spawn()

MPI and SPMD

- **<u>MPI process Rank:</u>** A number between <u>0 to N-1</u> identifying the process where N is the total number of MPI processes involved in the computation.
  - The MPI function **MPI_Comm_rank** reports the rank of the calling process (e.g rank from 0 to N-1).
  - The MPI function **MPI_Comm_size** reports the total number of MPI processes (e.g N).

My rank?

How many MPI processes?

MPI Process Initialization & Clean-Up

MPI_INIT **MPI_FINALIZE**

- The first MPI routine called in any MPI program *must* be the initialization routine **MPI_INIT**.

  Process → MPI Process

  - **Every MPI program must call this routine *once*, before any other MPI routines.**

  The process becomes an MPI process

- An MPI program should call the MPI routine **MPI_FINALIZE** when all communications have completed. This routine cleans up all MPI data structures etc.
- **MPI_FINALIZE** does NOT cancel outstanding communications, so it is the responsibility of the programmer to make sure all communications have completed.
  - Once this routine is called, no other calls can be made to MPI routines, not even MPI_INIT, so a process **cannot later re-enroll in MPI.**

  The process is no longer an MPI process

MPI Communicators, Handles

Communicator ***or*** **Name of a group of MPI Processes**

- **MPI_INIT** defines a default communicator called **MPI_COMM_WORLD** for each process that calls it.
- All MPI communication calls require **a communicator argument** and MPI processes can only communicate if they share a communicator.

i.e same group name

- Every **communicator** contains **a *group*** which is **a list of processes**. The processes are ordered and numbered consecutively from zero, the number of each process being its ***rank***. The rank identifies each process within the communicator.

e.g rank = 0 …. N-1

Default Communicator

- The group of **MPI_COMM_WORLD** is the **set of all MPI processes**.
- MPI maintains internal data structures related to communications etc. and these are referenced by the user through *handles*.
- Handles are returned to the user from some MPI calls and can be used in other MPI calls.

MPI Datatypes

- **The data in a message to sent or receive is described by a triplet:**

  (address, count, datatype)

where:

- **An MPI *datatype* is recursively defined as:**
  - **Predefined: corresponding to a data type from the language**
    - **(e.g., MPI_INT, MPI_DOUBLE_PRECISION)**
  - **A contiguous array of MPI datatypes**
  - **A strided block of datatypes**
  - **An indexed array of blocks of datatypes**
  - **An arbitrary structure of datatypes**

- **There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.**

MPI Indispensable Functions

- MPI_Init - Initialize MPI
- MPI_Finalize - Terminate MPI
- MPI_Comm_size - Find out how many processes there are
- MPI_Comm_rank - Find out my rank
- MPI_Send - Send a message
- MPI_Recv - Receive a message

} Basic or Blocking Send/Receive

- MPI_Bcast - Broadcasts data
- MPI_Reduce - Combines values into a single value

MPI_Init ,   MPI_Finalize

MPI_Init

Process Becomes an MPI Process

- The call to MPI_Init is required in every MPI program and must be the first MPI call. It establishes the MPI execution environment.

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

  Input Parameters:

  argc - Pointer to the number of arguments from main ()
  argv - Pointer to the argument vector from main()

MPI_Finalize

Process is no longer an MPI Process

- This routine terminates the MPI execution environment; all processes must call this routine before exiting.

  int MPI_Finalize( )

MPI_Comm_size

How many MPI processes in a group?

i.e. Communicator

- This routine determines the size (i.e., number of processes) of the group associated with the communicator given as an argument.

  int MPI_Comm_size(MPI_Comm comm, int *size)

  Input:

  comm - communicator (handle)

  Output:

  size - number of processes in the group of comm (returned)

EX: MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

Communicator (group) handle
Here:
Group of all MPI processes

How many MPI processes?

MPI_Comm_rank

What is my rank?

- The routine determines the rank (i.e., which process number am I?) of the calling process in the communicator.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Input:

comm - communicator (handle)

Output:

rank - rank of the calling process in the group of comm (integer, returned)

EX: MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

Communicator (group) handle
Here:
Group of all MPI processes

Rank of calling process

Simple “Minimal” MPI C Program Hello.c

```
#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( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

To Compile: mpicc Hello.c -o Hello

To Run: mpirun -np 4 -machinefile .rhosts Hello

Multiple Program Multiple Data (MPMD) in MPI

- Static process creation in MPI where each process has the same executable code leads to the direct support of MPI for single program-multiple data (**SPMD**).
- In general **MPMD** and **master-slave** models are supported by **utilizing the rank of each process to execute different portions of the code.**

Example:

```
main (int argc, char **argv)
{
    MPI_Init(&argc, &argc);                          /* initialize MPI */
       .  .  .
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);      /* find process rank */
    if (myrank==0)
   master();
    else
   slave();
       .  .  .
    MPI_Finalize();
}
```

Process with rank = 0 is the master

Other processes are slaves/workers

where **master( )** and **slave( )** are procedures in the main program to be executed by the master process and slave process respectively.

SPMD = Single Program Multiple Data
MPMD = Multiple Program Multiple Data

Variables Within Individual MPI Processes

SPMD = single program-multiple data

- Given the **SPMD** model, any global declarations of variables will be duplicated in each process. Variables and data that is not to be duplicated will need to be declared locally, that is, declared within code only executed by a process, for example as in:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find process rank */
if(myrank==0) {
    int x,y, data[100];
   .  .
} else {
    int x,y;
   .   .
}
```

Variables declared locally only by MPI process with rank = 0

Variables declared by other MPI processes (rank ≠ 0)

- Here **x** and **y** in process 0 are different local variables than **x** and **y** in process 1. The array, **data[100]**, might be for example some data which is to be sent from the master to the slaves.

Rank ≠ 0

Rank = 0

Basic or Blocking Send/Receive Routines

(Point-to-Point Communication)

- **Basic or Blocking routines** (send or receive) return when they are **locally complete**.

Most common MPI point-to-point communication type

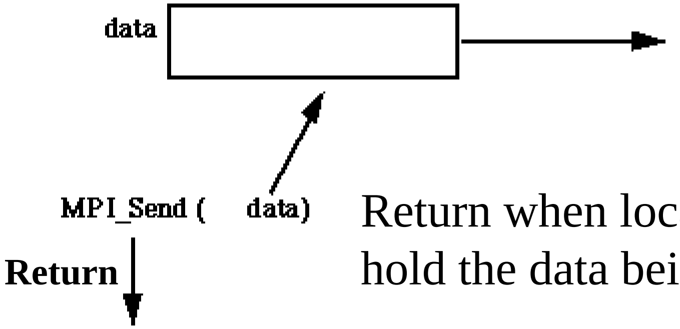
- In the context of a blocking send routine **MPI_Send( )** the local conditions are that the location being used to hold the data being sent **can be safely altered**, and blocking send will send the message and return then.
- This does not mean that the message has been received, just that the process is free to move on without adversely affecting the message being sent.
- A blocking receive routine **MPI_Recv( )** will also return when it is locally complete, which in this case means that the **message has been received** into the destination location and the destination location **can be read**.

Basic or Blocking Send/Receive Routines

MPI_Send(buf, count, datatype, dest, tag, comm)

- **buf**: address of send buffer
- **count**: number of data elements in send buffer
- **datatype**: MPI datatype of each send buffer element
- **dest**: rank of destination process
- **tag**: message tag
- **comm**: Communicator (group)

MPI_Send

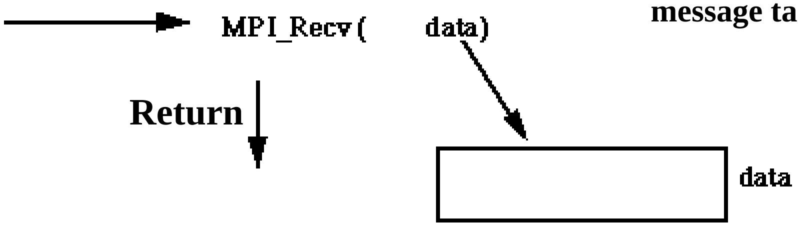


Return when location being used to hold the data being sent can be safely altered

MPI_Recv(buf, count, datatype, source, tag, comm, *status)

- **buf**: address of receive buffer
- **count**: maximum number of data elements in receive buffer
- **datatype**: MPI datatype of each receive buffer element
- **source**: rank of source process
- **tag**: message tag
- **comm**: Communicator (group)
- ***status**: Status info on received message

MPI_Recv



Return when the message has been received into the destination location and the destination location can be read.

Syntax is similar for synchronous MPI Send/Receive:
MPI_Ssend ( ), MPI_Srecv ( )

MPI_Send (Basic or Blocking Send)

- This routine performs a basic (blocking) send; this routine may block and returns when the send is locally complete

  int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

  Input:

  buf - address of send buffer

  count - number of elements in send buffer (nonnegative integer)

  datatype - datatype of each send buffer element (handle)

  dest - rank of destination (integer)

  tag - message tag (integer)

  comm - communicator (handle)

- **MPI_Recv** (Basic or Blocking Receive)
- This routine performs a basic blocking receive.

int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Output:

buf - address of receive buffer

status - status object, provides information about message received; status is a structure of type MPI_Status. The elements of status: MPI_SOURCE is the source of the message received, and the element status. MPI_TAG is the tag value. MPI_ERROR potential errors.

Input:

count - maximum number of elements in receive buffer (integer)

datatype - datatype of each receive buffer element (handle)

source - rank of source (integer)

tag - message tag (integer)

comm - communicator (handle)

MPI Tags/Source Matching

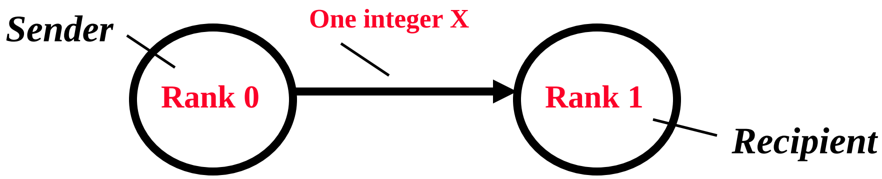
To match or not to match?

- Messages are sent with an accompanying **user-defined integer *tag***, to assist the receiving process in identifying/matching the message.
- Messages can be screened at the receiving end by specifying a specific tag, or **not screened (no tag matching) by specifying**:
  - **MPI_ANY_TAG** as the tag in a receive. (e.g. it matches with any tag).
- Similarly, messages can be screened at the receiving end by specifying **a specific source**, or not screened by specifying:
  - **MPI_ANY_SOURCE** as the source in a receive. (**e.g. it matches with any source process**)

No Source matching done

Blocking Send/Receive Code Example

To send an integer x from MPI process with rank 0 to MPI process with rank 1:



```
   .
   .
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find process
   rank */
if (myrank==0) {
    int x;
    MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank==1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD,
   status);
}
   .
   .
```

Count

MPI Datatype

Rank of Destination Process

Message Tag

Communicator

Address of
receive buffer

Count

MPI Datatype

Rank of source process

Non-blocking Send/Receive Routines

- Non-blocking or **immediate** MPI send/receive routines return immediately whether or not they are locally (or globally) complete.
- The non-blocking send, **MPI_Isend()**, where **I** refers to the word "immediate", will return even before the source location is safe to be altered.
- The non-blocking receive, **MPI_Irecv()**, will return even if there is no message to accept.
- The non-blocking send, **MPI_Isend()**, corresponds to the pvm routine **pvm_send()** and the non-blocking receive, **MPI_Irecv()**, corresponds to the pvm routine **pvm_nrecv()**. The formats are:

  MPI_Isend(buf, count, datatype, dest, tag, comm, request)

  MPI_Irecv(buf, count, datatype, source, tag, comm, request)

- The additional parameter, **request**, is used when it is necessary to know that the operation has actually completed.
- Completion can be detected by separate routines, **MPI_Wait( )** and **MPI_Test( )**. **MPI_Wait( )** returns if the operation has actually completed and will wait until completion occurs. **MPI_Test( )** returns immediately with a flag set indicating whether the operation has completed at this time. These routines need the identity of the operation, in this context the non-blocking message passing routines which is obtained from request.

Why?

Non-blocking routines provide the facility to **overlap communication and computation** which is essential when communication delays are high.

Non-blocking Send Code Example

(immediate)

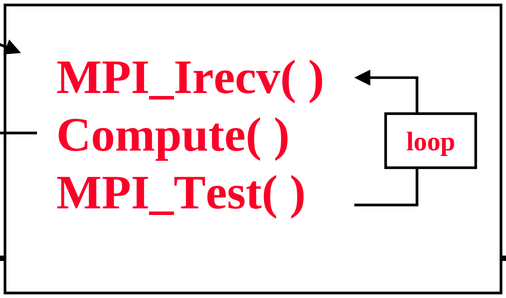
- To send an integer x from process 0 to process 1 and allow process 0 to continue immediately:

```
   .
   .
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find process rank */
if (myrank==0) {
    int x;
    MPI_Isend(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
    compute();
    MPI_Wait(req1, status);
} else if (myrank==1) {
    int x;
    MPI_Recv(&x, 0, MPI_INT, 1, msgtag, MPI_COMM_WORLD, status);
   .
```

Overlap computation with communication

Sent?

or



MPI_Irecv( )
Compute( )
MPI_Test( )

Overlap with communication

MPI Group/Collective Communications Functions

The principal collective operations are:

- **MPI_Barrier( )** **/* Blocks process until all processes have called it */**

  EX. `MPI_Barrier( MPI_COMM_WORLD )`

- → **MPI_Bcast( )** **/* Broadcast from root to all other processes */**
- → **MPI_Reduce( )** **/* Combine values on all processes to single value */**
- → **MPI_Gather( )** **/* Gather values for group of processes */**
- → **MPI_Scatter( )** **/* Scatters a buffer in parts to group of processes */**
- **MPI_Alltoall( )** **/* Sends data from all processes to all processes */**
- **MPI_Reduce_scatter( )/* Combine values and scatter results */**
- **MPI_Scan( )** **/* Compute prefix reductions of data on processes */**

MPI_Bcast (Data Broadcast)

- This routine broadcasts data from the process with rank "root" to all other processes of the group.

(i.e same communicator)

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Input/Output:

buffer - address of buffer
count - number of entries in buffer (integer)
datatype - data type of buffer (handle)
root - rank of broadcast root (integer)
comm - communicator (handle)

```
EX.  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Address of send buffer — Count — Datatype — Root — Communicator

MPI_Bcast (Data Broadcast)

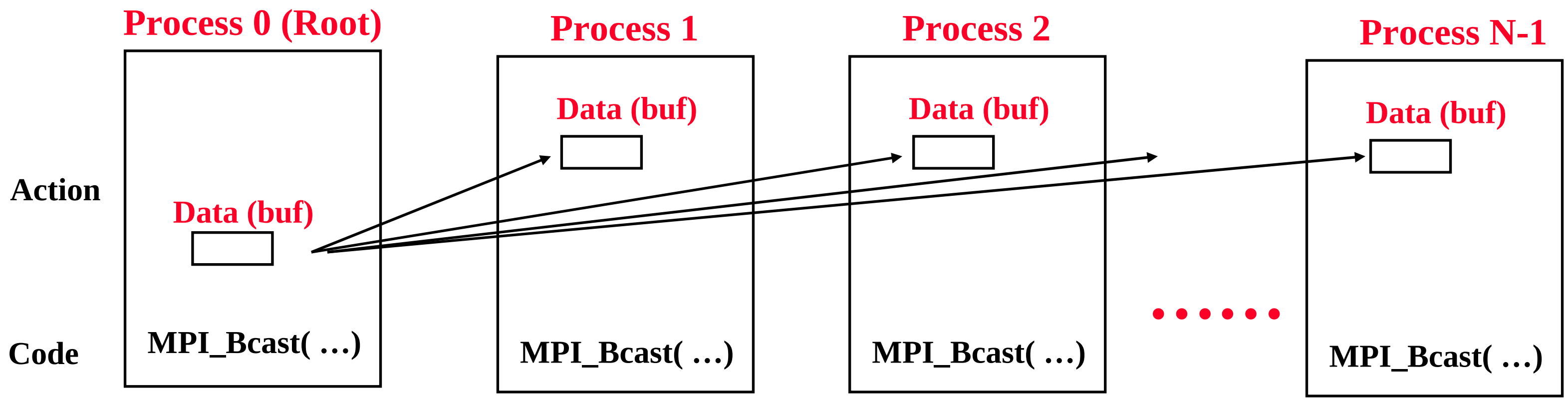
MPI_Bcast( ) /* Broadcast from root to all other processes */

MPI_Bcast(buf, count, datatype, root, comm)

- buf: **address of send/receive buffer**
- count: **number of data elements in send buffer**
- datatype: **MPI datatype of each send buffer element**
- root: **rank of root process** **i.e rank of broadcasting process**
- comm: **Communicator (group)**

```
EX.  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- &n: **Address of send/receive buffer**
- 1: **Count**
- MPI_INT: **Datatype**
- 0: **Root**
- MPI_COMM_WORLD: **Communicator** **i.e group name**



Assuming MPI_Comm_size = N MPI Processes

MPI_Bcast is an example MPI group/collective communication Function

MPI_Reduce

- This routine combines values on all processes into a single value using the operation defined by the parameter op.

int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Input:
sendbuf - address of send buffer
count - number of elements in send buffer (integer)
datatype - data type of elements of send buffer (handle)
op - reduce operation (handle) (user can create using MPI_Op_create or use predefined operations MPI_MAX, MPI_MIN, MPI_PROD, MPI_SUM, MPI_LAND, MPI_LOR, MPI_LXOR, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC in place of MPI_Op op.
root - rank of root process (integer)
comm - communicator (handle)

Reduce Operations

Output:
recvbuf - address of receive buffer (significant only at root)

MPI_Reduce is an example MPI group/collective communication Function

MPI_Reduce (Data Reduction)

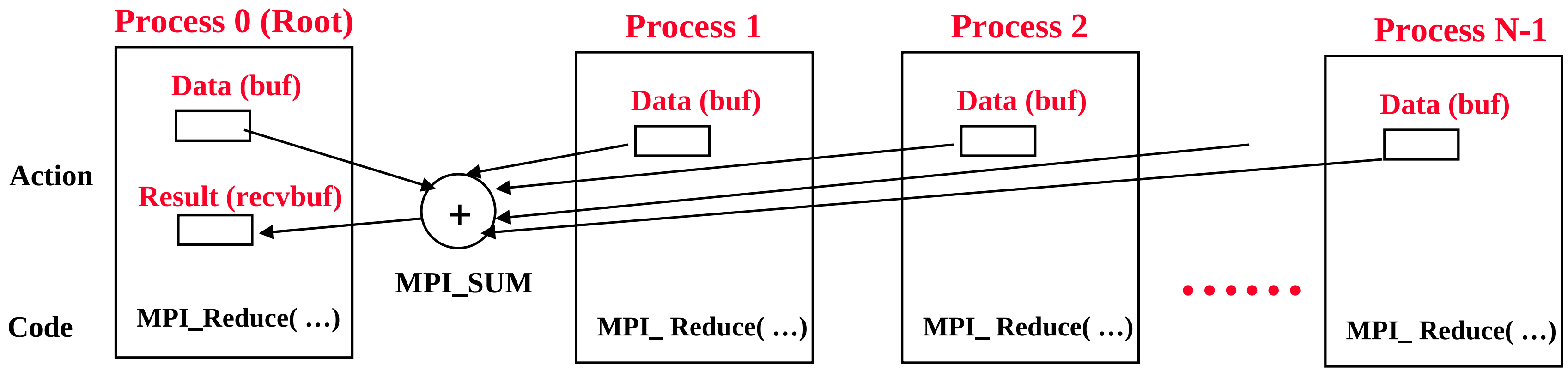
MPI_Reduce( ) /* Combine values on all processes to single value */

```
MPI_Reduce(buf, recvbuf, count, datatype, op, root, comm)
```

- **buf**: address of send buffer
- **recvbuf**: address of receive buffer (at root)
- **count**: number of data elements in send buffer
- **datatype**: MPI datatype of each send buffer element
- **op**: Reduce operation
- **root**: Rank of root process
- **comm**: Communicator (group)

```
EX. MPI_Reduce(&n,&m, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

- **&n**: Address of send buffer
- **&m**: address of receive buffer (at root)
- **1**: Count
- **MPI_INT**: Datatype
- **MPI_SUM**: Reduce operation
- **0**: Root
- **MPI_COMM_WORLD**: Communicator i.e group name



Assuming MPI_Comm_size = N MPI Processes

MPI_Reduce is an example MPI group/collective communication Function

MPI Program Example To Compute Pi

Part 1 of 2

The program computes Pi (Π) by computing the area under the curve $f(x) = 4/(1+x^2)$ between 0 and 1.

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```

Input n number of intervals

Communication
O(p)
p number of processes

MPI process with rank 0 (i.e master process) broadcasts number of intervals n

MPI Program Example To Compute Pi

Part 2 of 2

```
 h   = 1.0 / (double) n;
  sum = 0.0;
  for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  mypi = h * sum;
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);
  if (myid == 0)
    printf("pi is approximately %.16f, Error is %.16f\n",
           pi, fabs(pi - PI25DT));
}
MPI_Finalize();
return 0;
}
```

Computation
O(n/p)
n number of intervals
p number of processes

Reduce
operation

Communication
O(p)

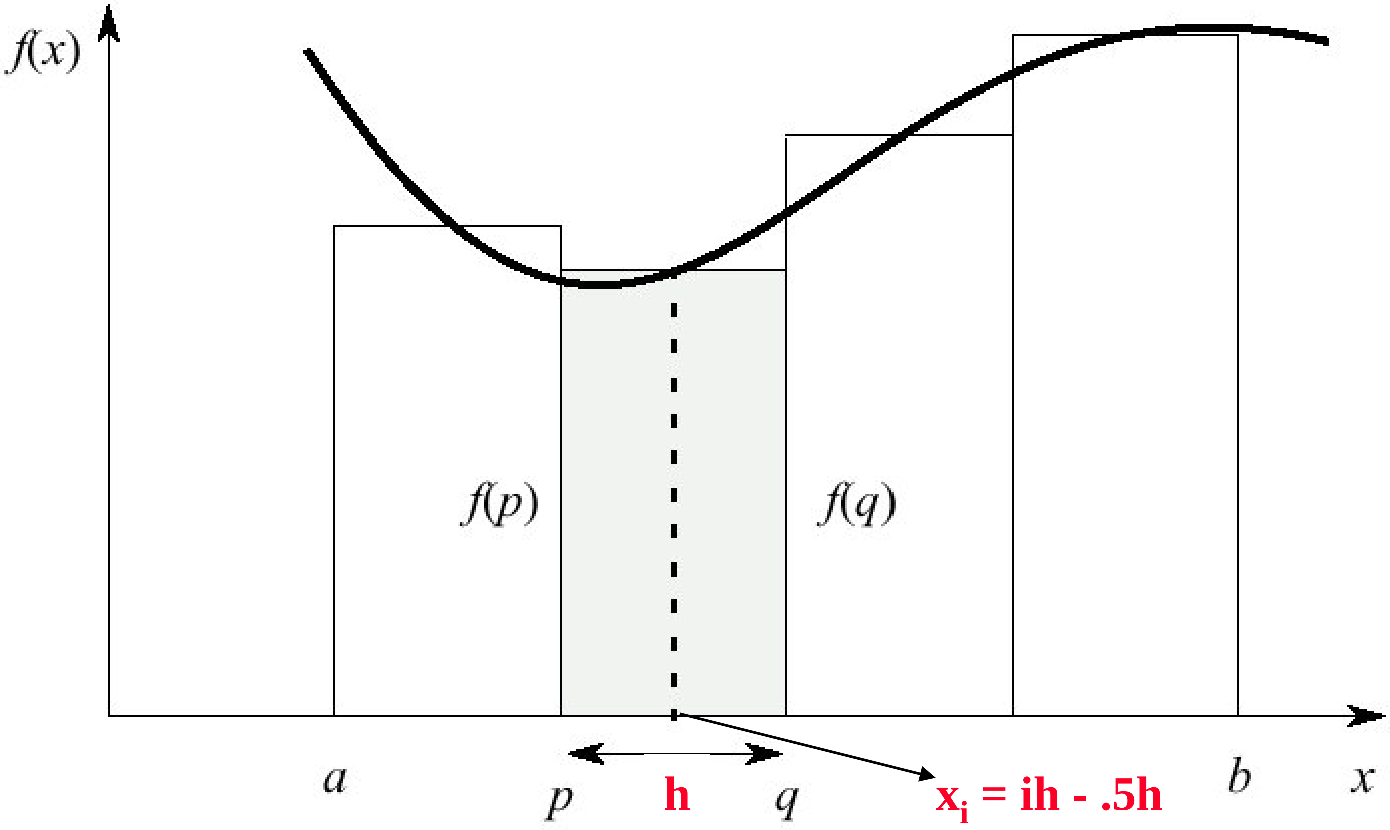
root

Computation = O(n/p)
Communication ~ O(p)
Communication to Computation ratio
C-to-C ratio = O(p / (n/p) = $O(p^2 /n)$
Ex. n = 1000 p = 8
C-to-C = 8 x 8 / 1000 = 64/1000 = 0.064

n = total number of intervals
p = number of processes or processors
n/p = intervals per process or processor

Numerical Integration Using Rectangles

To Compute Pi integrate $f(x) = 4/(1+x^2)$ between $x = 0$ and 1



h

x_i = ih - .5h

rectangle area = h x $f(x_i)$

h = 1/ n

n = number of intervals

n/p intervals per process or processor

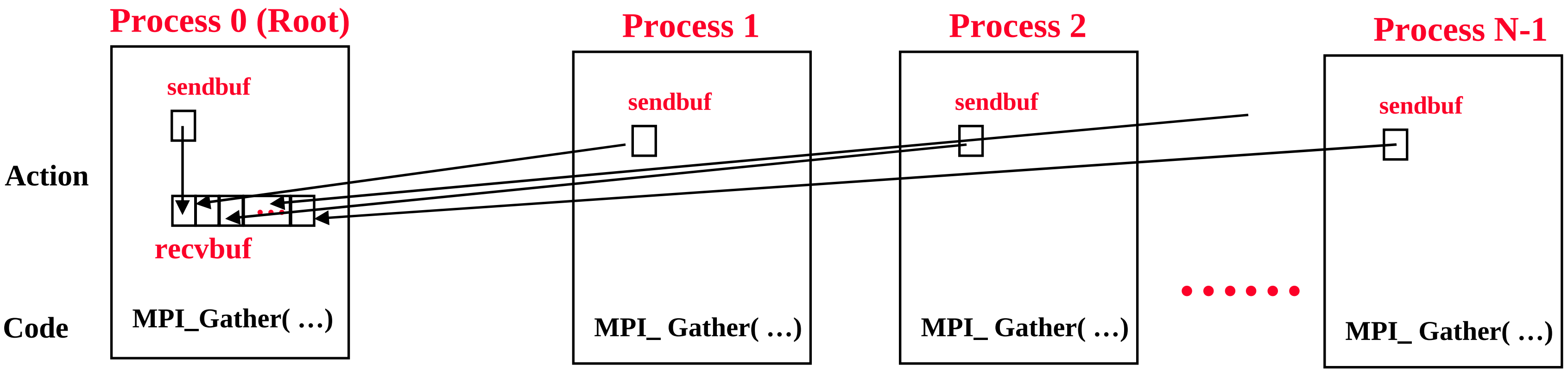
MPI_Gather (Data Gather Operation)

MPI_Gather( ) /* Gather values for group of processes */

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

- **sendbuf**: address of send buffer
- **sendcount**: number of data elements in send buffer
- **sendtype**: MPI datatype of each send buffer element
- **recvbuf**: address of receive buffer (at root)
- **recvcount**: number of data elements in receive buffer
- **recvtype**: MPI datatype of each receive buffer element
- **root**: Rank of root (receiving process)
- **comm**: Communicator (group)

```
EX. MPI_Gather(&s,1, MPI_INT, &r, 1, MPI_INT, 0, MPI_COMM_WORLD);
```



Assuming MPI_Comm_size = N MPI Processes

MPI_Scatter (Data Scatter Operation)

MPI_Scatter( ) /* Scatters a buffer in parts to group of processes */

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

address of send buffer (at root)

number of data elements in send buffer

MPI datatype of each send buffer element

address of receive buffer

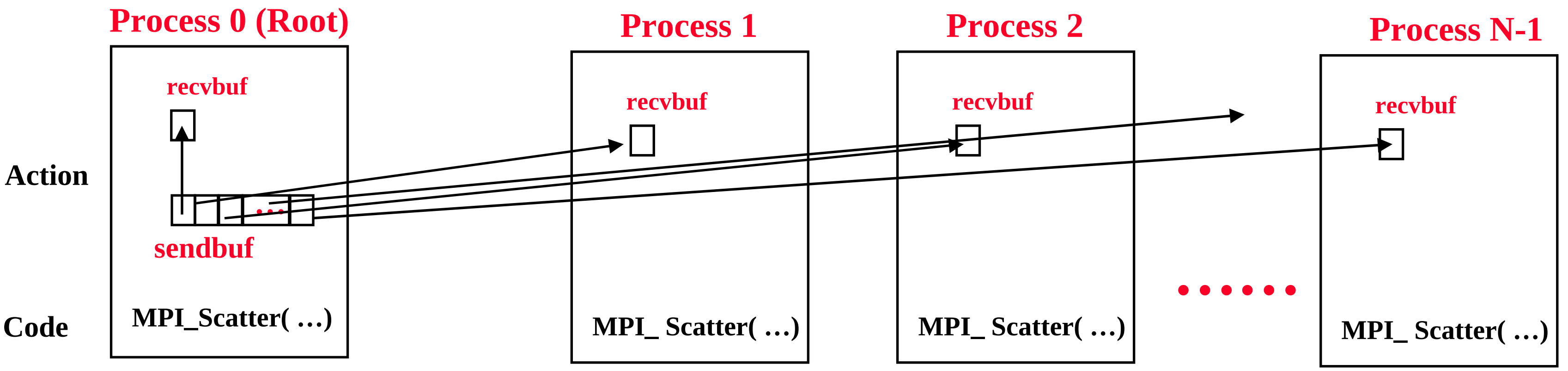
number of data elements in receive buffer

MPI datatype of each receive buffer element

Rank of root (receiving process)

Communicator (group)

```
EX. MPI_Scatter(&s,1, MPI_INT, &r, 1, MPI_INT, 0, MPI_COMM_WORLD);
```



Assuming MPI_Comm_size = N MPI Processes

MPI Program Example:

Partitioned "Divide And Conquer" Sum

- This program adds N = 1000 numbers stored in a file "rand_data.txt".
- The master process (rank 0) reads the data from a file into an array "data" and broadcasts the numbers to all MPI processes.
- Each MPI process adds its portion of the data forming a local partial sum.
- The partial sums are added using "MPI_reduce" with the master/root process (rank 0) ending up with the overall sum.
- Due to the even distribution of the data, the overall speed of the program is limited by the speed of the slowest processor in the virtual system.
- To avoid this, the work load should be distributed so that the faster processors perform more of the work.

Partitioned Sum MPI Program Example: sum.c

In Parallel Programming book Figure 2.12 Page 60

Divide And Conquer Addition Example

Communication

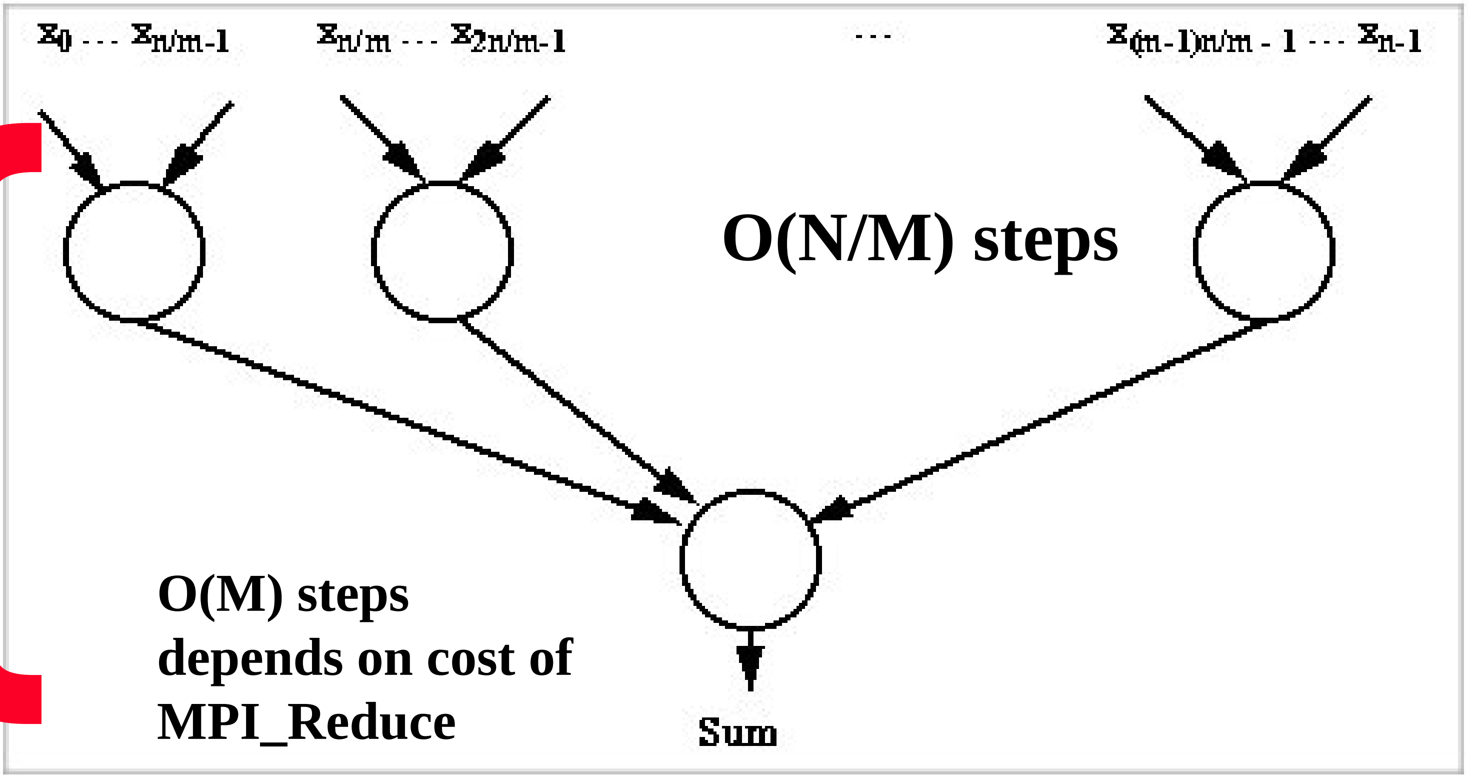
$t_{com} = O(N)$

Broadcast

Computation

t_{comp}

Reduce



O(N/M) steps

O(M) steps
depends on cost of
MPI_Reduce

$$t_p = t_{com} + t_{comp} = O(N + N/M + M)$$

Communication to computation ratio (C-to-C) = N / (N/M + M)

p = M = number or processors

Performance of Divide And Conquer Addition Example

- The communication time t_{com} required for the master process to transmit the numbers to slave processes is proportional to N.
- The M processes each add N/M numbers together which requires N/M - 1 additions.
- Since all M processes are operating together we can consider all the partial sums will be obtained in the N/M - 1 steps.
- The master process has to add the m partial sums which requires M - 1 steps.
- Hence the parallel computation time, t_{comp} is:

  $$t_{comp} = N/M - 1 + M - 1$$

  or a parallel time complexity of:

  $$t_{comp} = O(N/M + M)$$

  Hence:

  $$t_p = t_{com} + t_{comp\,=}\; N + N/M + M - 2 = O(N + N/M + M)$$

  where the first N term is the communication aspect and the remaining terms are the computation aspects.
- Communication to computation ratio (C-to-C) = N / (N/M + M)
- This is worse than the sequential time of $t_s = N - 1$ and complexity of O(N).

p = M = number or processors

CE Cluster Configuration/MPI Usage Resources/Guides

http://mps.ce.rit.edu/

- Cluster configuration:

  http://mps.ce.rit.edu/guide/configuration.php

- Using/configuring MPI on the cluster:

  http://mps.ce.rit.edu/guide/mpi.php

  http://mps.ce.rit.edu/guide/tutorial.php

- Cluster status:

  http://rocks.ce.rit.edu/ganglia/