Module 4: Communication

VIT®

Vellore Institute of Technology

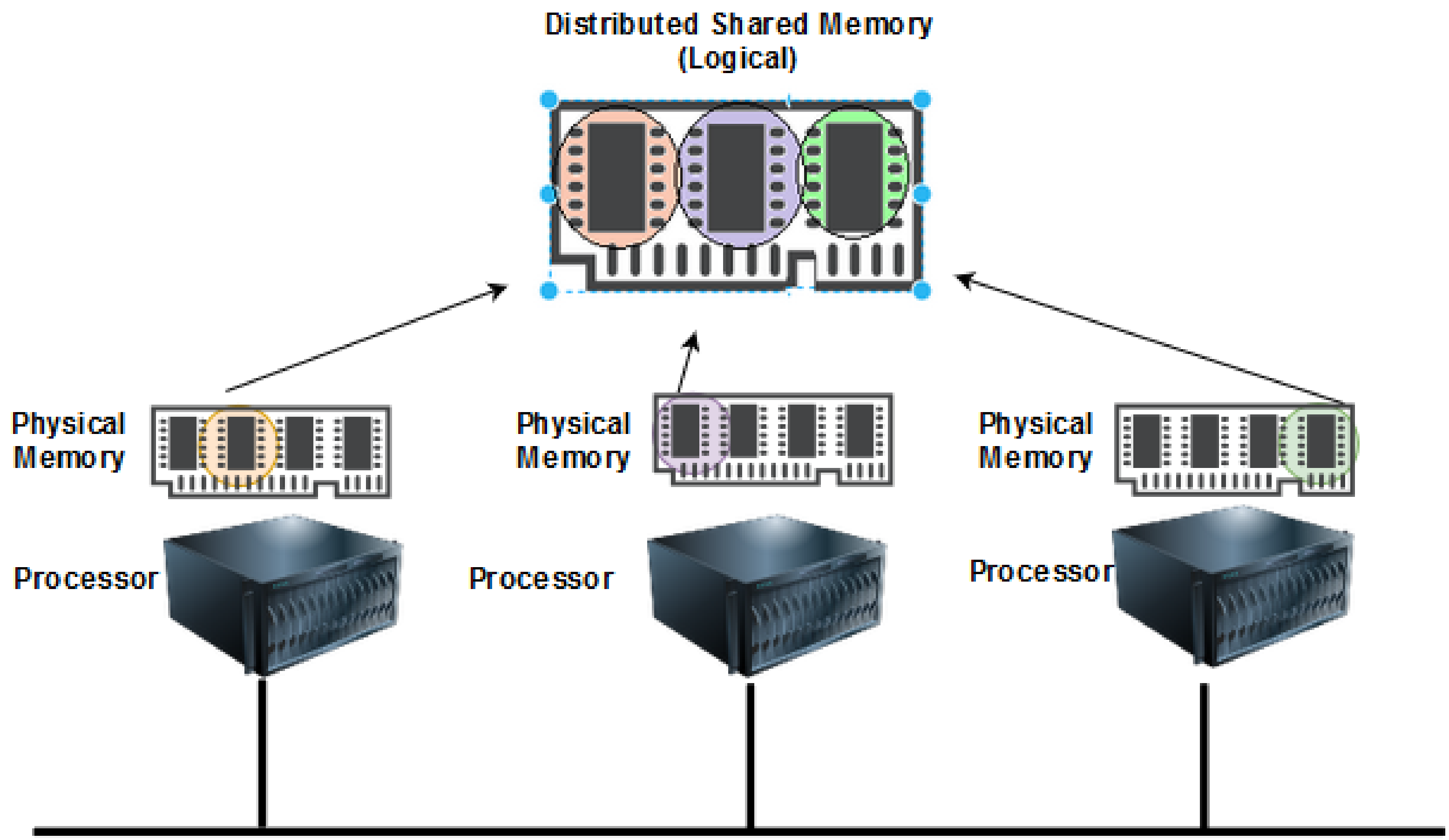
(Deemed to be University under section 3 of UGC Act, 1956)

Distributed Shared Memory (DSM) Approaches

How to share data between distributed processor nodes?

- Distributed Shared Memory
  - It's the abstraction to share data between distributed computers
- It enables access across multiple distributed physical memory to be accessed as single shared memory

DSM Abstraction



Working of DSM

- DSM enables easy access of individual computer shared data items
- DSM run time support sends updates as messages between computers
- Each computer has a local copy of recently accessed data items stored in DSM

…Continued…

- Shared memory request for a non-local piece of data is raised
- Single copy of data fetched and given to the requested system
- If multiple machines access the data at the same time, synchronization primitive like semaphore is used to handle the situation

…Continued

- Read(shared variable)
- Write (data, shared variable)

Issues related to DSM Semantics

- Structure and granularity – data shared at the bit, word or page level
- Consistency – If multiple requests for a single data and each machine tries to update it, consistency should be maintained. This involves cache coherence like solution
- Heterogeneity – Accommodating different data representations of different machines, languages and OS
- Scalability – Bus latency, Increased broadcast messages

DSM Versus Message Passing

- No marshaling of messages in DSM where as messages are marshalled and unmarshalled in message passing
- Synchronization in DSM is by constructs like semaphore/locks where as in message passing it is by message passing primitives
- DSM→ processes can communicate with non-overlapping lifetimes where as in message passing processes communicate at the same time
- Efficiency in DSM heavily depends on the pattern of Data access by multiple machines where as the same is not true for message passing

DSM features

- Space- uncoupled
- Time – uncoupled
- State based service
- Used for parallel and distributed computation
- Limited scalability
- Not associative

Distributed Resource Management: Distributed Shared Memory

Courtesy: CS-550: Distributed Shared Memory [SiS ’94]

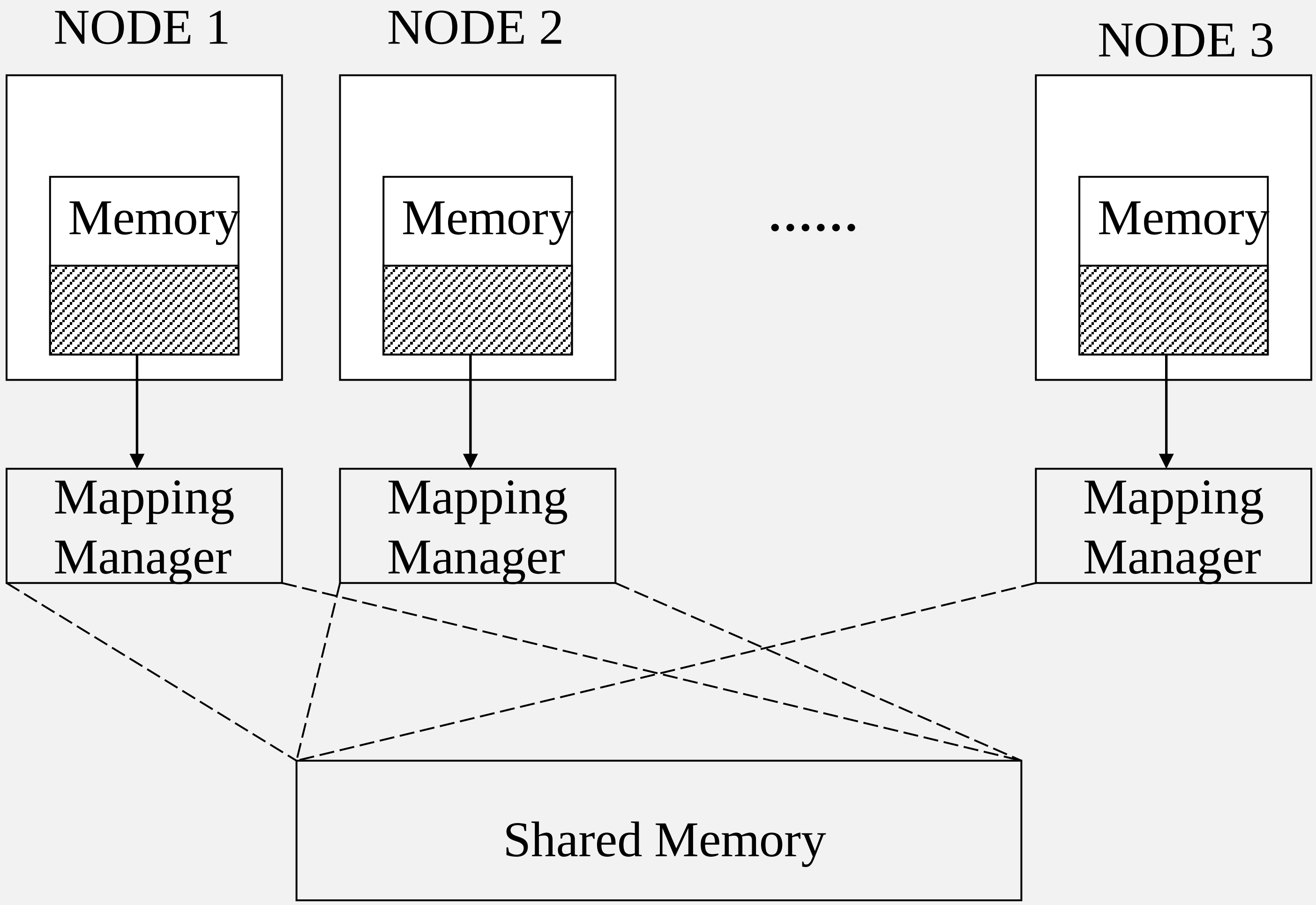
DSM

- What
  - The distributed shared memory (DSM) implements the shared memory model in distributed systems, which have no physical shared memory
  - The shared memory model provides a virtual address space shared between all nodes
  - To overcome the high cost of communication in distributed systems, DSM systems move data to the location of access

DSM

- How:
  - Data moves between main memory and secondary memory (within a node) and between main memories of different nodes
  - Each data object is owned by a node
    - Initial owner is the node that created object
    - Ownership can change as object moves from node to node
  - When a process accesses data in the shared address space, the mapping manager maps shared memory address to physical memory (local or remote)

DSM


NODE 1
NODE 2
NODE 3
Memory
Memory
Memory
……
Mapping Manager
Mapping Manager
Mapping Manager
Shared Memory

Advantages of distributed shared memory (DSM)

- Data sharing is implicit, hiding data movement (as opposed to 'Send'/'Receive' in message passing model)
- Passing data structures containing pointers is easier (in message passing model data moves between different address spaces)
- Moving entire object to user takes advantage of locality difference
- Less expensive to build than tightly coupled multiprocessor system: off-the-shelf hardware, no expensive interface to shared physical memory
- Very large total physical memory for all nodes: Large programs can run more efficiently
- No serial access to common bus for shared physical memory like in multiprocessor systems
- Programs written for shared memory multiprocessors can be run on DSM systems with minimum changes

Algorithms for implementing DSM

- Issues
  - How to keep track of the location of remote data
  - How to minimize communication overhead when accessing remote data
  - How to access concurrently remote data at several nodes

The Central Server Algorithm

- Central server maintains all shared data
  - Read request: returns data item
  - Write request: updates data and returns acknowledgement message
- Implementation
  - A timeout is used to resend a request if acknowledgment fails
  - Associated sequence numbers can be used to detect duplicate write requests
  - If an application's request to access shared data fails repeatedly, a failure condition is sent to the application
- Issues: performance and reliability
- Possible solutions
  - Partition shared data between several servers
  - Use a mapping function to distribute/locate data

The Migration Algorithm

- Operation
  - Ship (migrate) entire data object (page, block) containing data item to requesting location
  - Allow only one node to access a shared data at a time
- Advantages
  - Takes advantage of the locality of reference
  - DSM can be integrated with VM at each node
    - Make DSM page multiple of VM page size
    - A locally held shared memory can be mapped into the VM page address space
    - If page not local, fault-handler migrates page and removes it from address space at remote node

Migration Algorithm

- To locate a remote data object:
  - Use a location server
  - Maintain hints at each node
  - Broadcast query
- Issues
  - Only one node can access a data object at a time
  - Thrashing can occur: to minimize it, set minimum time data object resides at a node

The Read-Replication Algorithm

- Replicates data objects to multiple nodes
- DSM keeps track of location of data objects
- Multiple nodes can have read access or one node write access (multiple readers-one writer protocol)
- After a write, all copies are invalidated or updated
- DSM has to keep track of locations of all copies of data objects. Examples of implementations:
  - IVY: owner node of data object knows all nodes that have copies
  - PLUS: distributed linked-list tracks all nodes that have copies
- Advantage
  - The read-replication can lead to substantial performance improvements if the ratio of reads to writes is large

The Full–Replication Algorithm

- Extension of read-replication algorithm: multiple nodes can read and multiple nodes can write (multiple-readers, multiple-writers protocol)
- Issue: consistency of data for multiple writers
- Solution: use of gap-free sequencer
  - All writes sent to sequencer
  - Sequencer assigns sequence number and sends write request to all sites that have copies
  - Each node performs writes according to sequence numbers
  - A gap in sequence numbers indicates a missing write request: node asks for retransmission of missing write requests

Memory coherence

- DSM are based on
  - Replicated shared data objects
  - Concurrent access of data objects at many nodes
- Coherent memory: when value returned by read operation is the expected value (e.g., value of most recent write)
- Mechanism that control/synchronizes accesses is needed to maintain memory coherence
- Sequential consistency: A system is sequentially consistent if
  - The result of any execution of operations of all processors is the same as if they were executed in sequential order, and
  - The operations of each processor appear in this sequence in the order specified by its program
- General consistency:
  - All copies of a memory location (replicas) eventually contain same data when all writes issued by every processor have completed

Memory coherence (Cont.)

- Processor consistency:
  - Operations issued by a processor are performed in the order they are issued
  - Operations issued by several processors may not be performed in the same order (e.g. simultaneous reads of same location by different processors may yields different results)
- Weak consistency:
  - Memory is consistent only (immediately) after a synchronization operation
  - A regular data access can be performed only after all previous synchronization accesses have completed

Memory coherence (Cont.)

- Release consistency:
  - Further relaxation of weak consistency
  - Synchronization operations must be consistent which each other only within a processor
  - Synchronization operations: Acquire (i.e. lock), Release (i.e. unlock)
  - Sequence: Acquire
Regular access
Release

Coherence Protocols

- Issues
  - How do we ensure that all replicas have the same information
  - How do we ensure that nodes do not access stale data

Write-invalidate protocol

- A write to shared data invalidates all copies except one before write executes
- Invalidated copies are no longer accessible
- Advantage: good performance for
  - Many updates between reads
  - Per node locality of reference
- Disadvantage
  - Invalidations sent to all nodes that have copies
  - Inefficient if many nodes access same object
- Examples: most DSM systems: IVY, Clouds, Dash, Memnet, Mermaid, and Mirage

Write-update protocol

- A write to shared data causes all copies to be updated (new value sent, instead of validation)
- More difficult to implement

Design issues

- Granularity: size of shared memory unit
  - If DSM page size is a multiple of the local virtual memory (VM) management page size (supported by hardware), then DSM can be integrated with VM, i.e. use the VM page handling
  - Advantages vs. disadvantages of using a large page size:
    - (+) Exploit locality of reference
    - (+) Less overhead in page transport
    - (-) More contention for page by many processes
  - Advantages vs. disadvantages of using a small page size
    - (+) Less contention
    - (+) Less false sharing (page contains two items, not shared but needed by two processes)
    - (-) More page traffic
  - Examples
    - PLUS: page size 4 Kbytes, unit of memory access is 32-bit word
    - Clouds, Munin: object is unit of shared data structure

Design issues (cont.)

- Page replacement
  - Replacement algorithm (e.g. LRU) must take into account page access modes: shared, private, read-only, writable
  - Example: LRU with access modes
    - Private (local) pages to be replaced before shared ones
    - Private pages swapped to disk
    - Shared pages sent over network to owner
    - Read-only pages may be discarded (owners have a copy)

Case studies: IVY

- IVY (Integrated shared Virtual memory at Yale) implemented in Apollo DOMAIN environment, i.e. Apollo workstations on a token ring
- Granularity: 1 Kbyte page
- Process address space: private space + shared VM space
  - Private space: local to process
  - Shared space: can be accesses by any process through the shared part of its address space

Case studies: IVY

- Node mapping manager: does mapping between local memory of that node and the shared virtual memory space
- Memory access operation
    - On page fault, block process
    - If page local, fetch from secondary memory
    - If not local, request a remote memory access, acquire page
- Page now available to all processes at the node

Case studies: IVY (Cont.)

- Coherence protocol
  - Page access modes: read only, write, nil (invalidate)
  - Multiple readers-single writer semantics
  - Protocol
    - Write invalidation: before a write to a page is allowed, all other read-only copies are invalidated
    - Strict consistency: a reader always sees the latest value written

Case studies: IVY (Cont.)

- Write sequence
  - Processor ‘i’ has write fault to page ‘p’
  - Processor ‘i’ finds owner of page ‘p’ and sends request
  - Owner of ‘p’ sends page and its copyset to ‘i’ and marks ‘p’ entry in its page table ‘nil’ (copyset = list of processors containing read-only copy of page)
  - Processor ‘i’ sends invalidation messages to all processors in copyset

Case studies: IVY (Cont.)

- Read sequence
  - Processor ‘i’ has read fault to page ‘p’
  - Processor ‘i’ finds owner of page ‘p’
  - Owner of ‘p’ sends copy of page to ‘i’ and adds ‘i’ to copyset of ‘p’. Processor ‘i’ has read-only access to ‘p’

Case studies: IVY (Cont.)

- Algorithms used for implementing actions for ‘Read’ and ‘Write’ actions
- Centralized manager scheme
  - Central manager resides on single processor: maintains all data ownership information
  - On page fault, processor ‘i’ requests copy of page from central manager
  - Central manager sends request to page owner. If ‘Write’ requested, updates owner information to indicate ‘i’ is the new owner
  - Owner sends copy of page to processor ‘i’ and
    - If ‘Write’, also sends <u>copyset</u> of page
    - If ‘Read’, adds ‘i’ to the <u>copyset</u> of page
  - On write, central manager sends invalidation messages to all processors in <u>copyset</u>
  - Performance issues
    - Two messages are required to locate page owner
    - On ‘Writes’, invalidation messages are sent to all processors in <u>copyset</u>
    - Centralized manager can become bottleneck

Algorithms used for implementing actions for ‘Read’ and ‘Write’ actions

- The fixed distributed manager scheme
  - Distributes the central manager’s role to every processor in the system
  - Every processor keeps track of the owners of a predetermined set of pages (determined by a mapping function *H*)
  - When a processor ‘i’ faults on page ‘p’, processor ‘i’ contacts processor *H(p)* for a copy of the page
  - The rest the protocol is the same as the one with the centralized manager

Note: In both the centralized and fixed distributed manager schemes, if two or more concurrent accesses to the same page are requested, the requests are serialized by the manager

Algorithms used for implementing actions for ‘Read’ and ‘Write’ actions

- The dynamic distributed manager scheme
  - Every host keeps track of the ownership of the pages that are in its local page table
    - Every page table has a field called *probowner* (probable owner)
    - Initially, *probowner* is set to a default processor
    - The field is modified as pages are requested from various processors
  - When a processor has a page fault, it sends a page request to processor ‘i’ indicated by the *probowner* field
  - If processor ‘i’ is the true owner of the page, fault handling proceeds like in centralized scheme
  - If ‘I’ is not the owner, it forwards the request to the processor indicated in its *probowner* field
  - This continues until the true owner of the page is found

Case studies: Mirage

- Developed at UCLA, kernel modified to support DSM operation
- Extends the coherence protocol of IVY system to control thrashing (in IVY, a page can move back and forth between multiple processors sharing the page)
- When a shared memory page is transferred to a processor, that processor will keep the page for ‘delta’ seconds
  - If a request for the page is made before ‘delta’ seconds expired, processor informs control manager of the amount of time left
  - ‘Delta’ can be a combination of real-time and service-time for that processor
- Advantages
  - Benefits locality of reference
  - Decreases thrashing

Case studies: Clouds

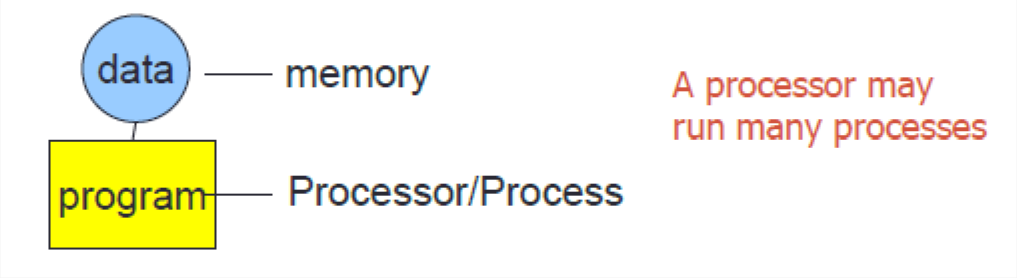
- Developed at Georgia Institute of Technology
- The virtual address space of all objects is viewed as a global distributed shared memory
  - The objects are composed of segments which are mapped into virtual memory by the kernel using the memory management hardware
  - A segment is a multiple of the physical page size
- For remote object invocations, the DSM mechanism transfers the required segments to the requesting host
  - On a segment fault, a *location system object* is consulted to locate the object
  - The *location system object* broadcasts a query for each locate operation
  - The actual data transfer is done by the distributed shared memory controller (DSMC)

Message Passing
&
Programming Using the Message-Passing Paradigm

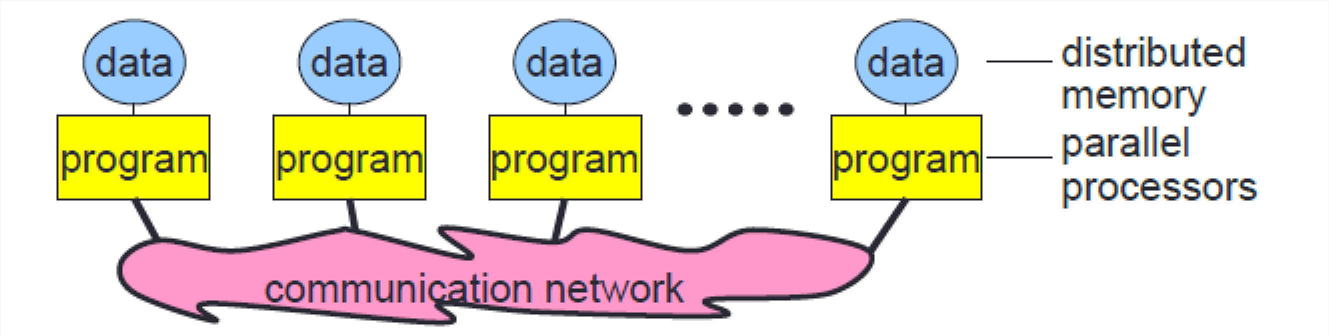
- Slides Taken from Hanjun Kim, Princeton University

The Message-Passing Programming Paradigm

- Sequential Programming Paradigm



- Message-Passing Programming Paradigm

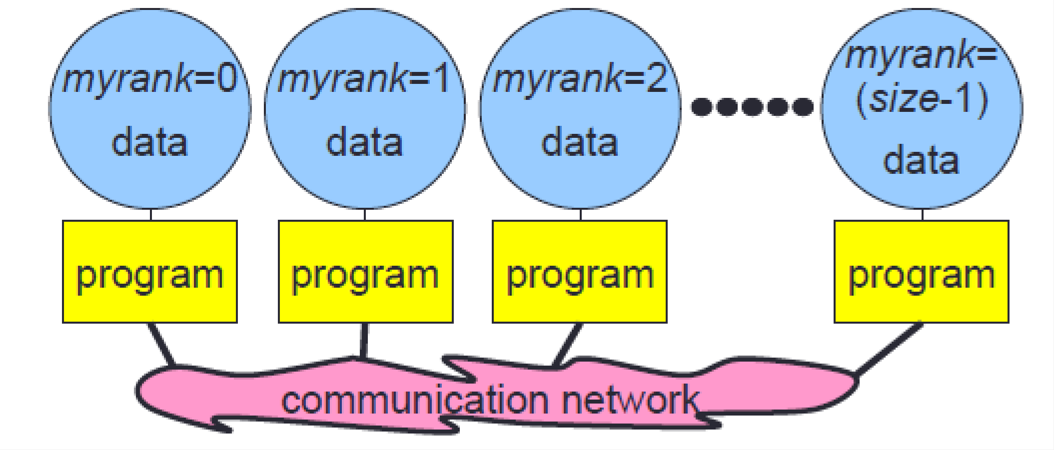


MPI Operation

- A **process** is a program performing a task on a **processor**
- Each processor/process in a message passing program runs a instance/copy of a program
- Typically a single program operating of multiple dataset
- The variables of each sub-program have
  - The same name
  - But different locations (distributed memory) and different data
  - i.e., all variables are local to a process
- Communicate via special send and receive routines (message passing)

Data and Work Distribution

- To communicate together MPI processes need identifiers: **rank = identifier number**
- All distribution decision are based on the **rank**
- i.e., which process works on which data



SPMD example

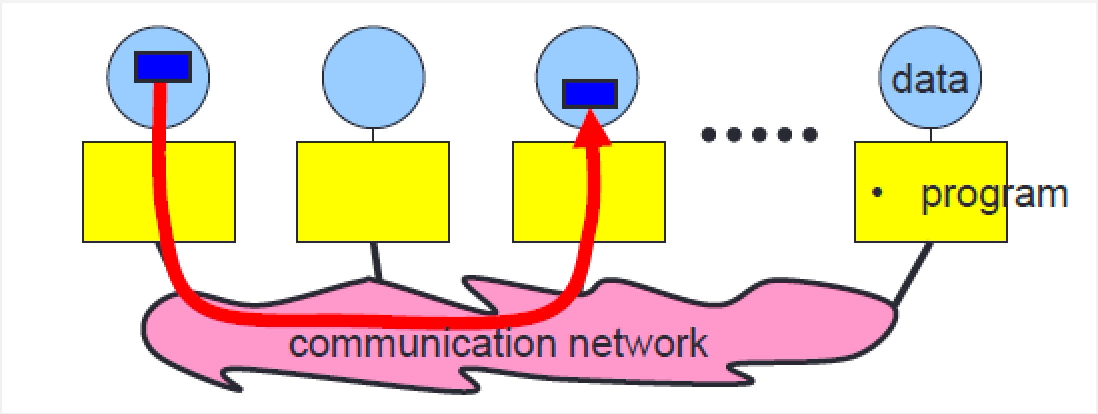
```
main(int argc, char **argv){
 if(process is assigned Master role){
        /* Assign work and coordinate workers
 and collect results */
        MasterRoutine(/*arguments*/);
      } else {  /* it is worker process */
        /* interact with master and other
 workers. Do the work and send results to
 the master*/
        WorkerRoutine(/*arguments*/);
      }
}
```

Why MPI?

- Small:
  - Many programs can be written with only 6 basic functions
- Large:
  - MPI's extensive functionality from many functions
- Scalable:
  - Point-to-point communication
- Flexible:
  - Don't need to rewrite parallel programs across platforms

Message Passing

- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:
- Sending Process, Receiving process -->i.e., the ranks
- Source location, Destination Location
- Source Data type, Destination Data type
- Source Data Size, Destination buffer size



Access

- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
  - phone line
  - mail box
  - fax machine
  - etc.
- MPI:
  - program must be linked with an MPI library
  - program must be started with the MPI startup tool

Basic functions

FUNCTION	DESCRIPTION
`int MPI_Init(int *argc, char **argv)`	Initialize MPI
`int MPI_Finalize()`	Exit MPI
`int MPI_Comm_size(MPI_Comm comm, int *size)`	Determine number of processes within a comm
`int MPI_Comm_rank(MPI_Comm comm, int *rank)`	Determine process rank within a comm
`int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`	Send a message
`int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status *status)`	Receive a message

Communicator

- An identifier associated with a group of processes
  - Each process has a unique rank within a specific communicator from 0 to (nprocesses-1)
  - Always required when initiating a communication by calling an MPI function
- Default: **MPI_COMM_WORLD**
  - contains all processes
- Several communicators can co-exist
  - A process can belong to different communicators at the same time

Hello World

- #include "mpi.h”
- intmain(intargc, char *argv[]) {
- intnproc, rank;

- MPI_Init(&argc,&argv); /* Initialize MPI*/
- MPI_Comm_size(MPI_COMM_WORLD,&nproc); /* Get CommSize*/
- MPI_Comm_rank(MPI_COMM_WORLD,&rank); /* Get rank*/

- printf(“Hello World from process %d\n”, rank);

- MPI_Finalize(); /* Finalize*/
- return 0;
- }

How to compile

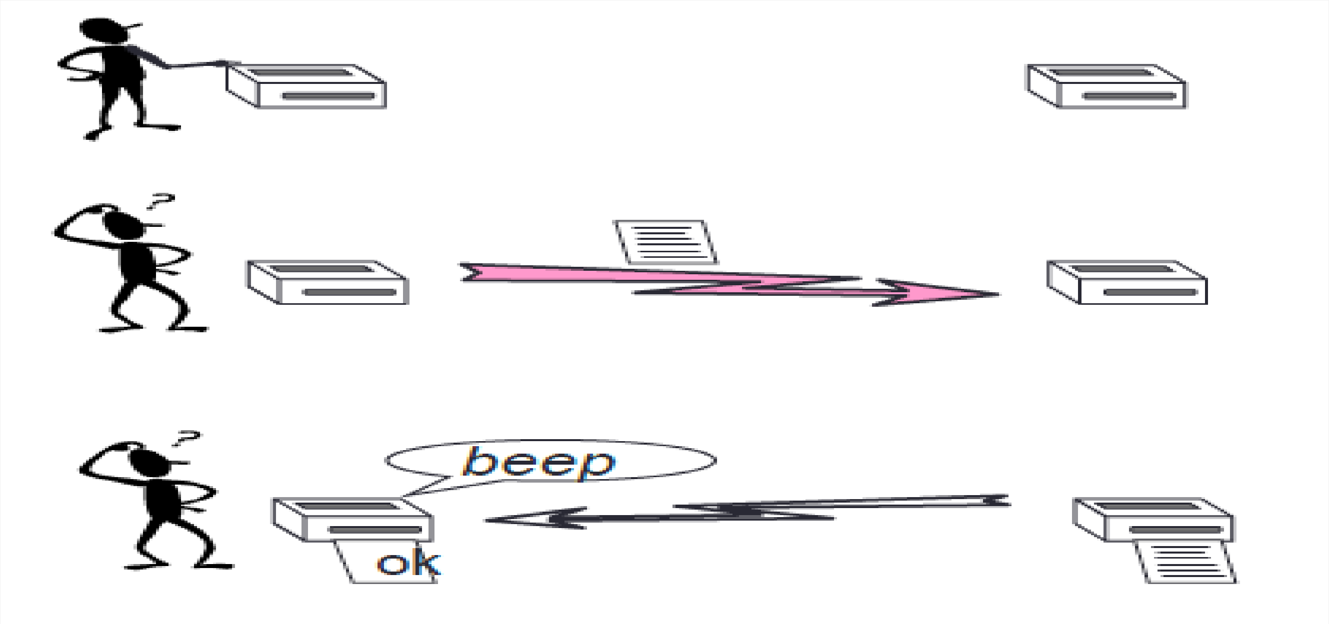
- Fortunately, most MPI implementations come with scripts that take care of these issues:
  - mpicc mpi_code.c –o a.out
- Two widely used (and free) MPI implementations
  - MPICH (http://www-unix.mcs.anl.gov/mpi/mpich)
  - OPENMPI (http://www.openmpi.org)

Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communication:
  - synchronous send
  - buffered = asynchronous send

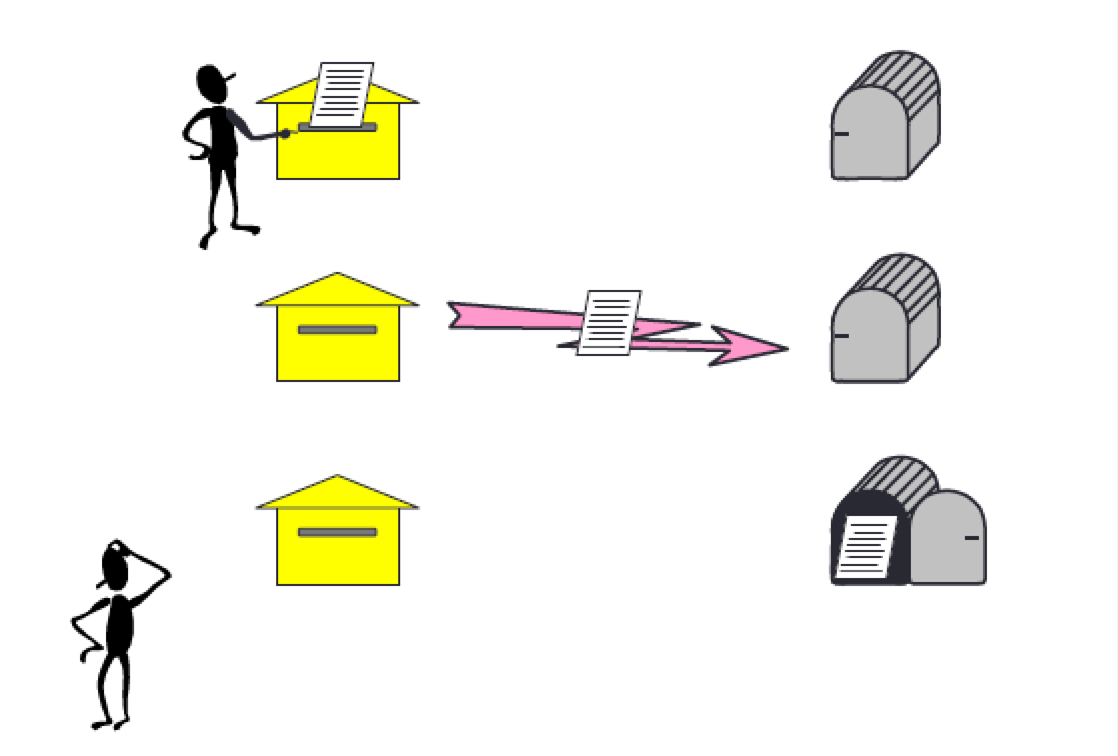
Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.



Buffered = Asynchronous Sends

- Only know when the message has left.



Blocking Operations

- **Some sends/receives may block until another process acts**:
- Synchronous send operation **blocks until** receive is issued;
- Receive operation **blocks until** message is sent.
- Blocking subroutine returns only when the operation has completed.

Blocking Message Passing

- The call waits until the data transfer is done:
- The sending process waits until all data are transferred to the system buffer
- The receiving process waits until all data are transferred from the system buffer to the receive buffer
- Buffers can be freely reused

Blocking Message Send

- **MPI_Send(void *buf, intcount, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);**

• buf	Specifies the starting address of the buffer.
• count	Indicates the number of buffer elements
• dtype	Denotes the datatype of the buffer elements
• dest	Specifies the rank of the destination process in the group associated with the communicator comm
• tag	Denotes the message label
• comm	Designates the communication context that identifies a group of processes

Blocking Message Send

`Standard(MPI_Send)`	The sending process returns when the system can buffer the message or when the message is received and the buffer is ready for reuse.
`Buffered (MPI_Bsend)`	The sending process returns when the message is buffered in an application-supplied buffer.
`Synchronous(MPI_Ssend)`	The sending process returns only if a matching receive is posted and the receiving process has started to receive the message.
`Ready (MPI_Rsend)`	The message is sent as soon as possible.

Blocking Message Receive

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

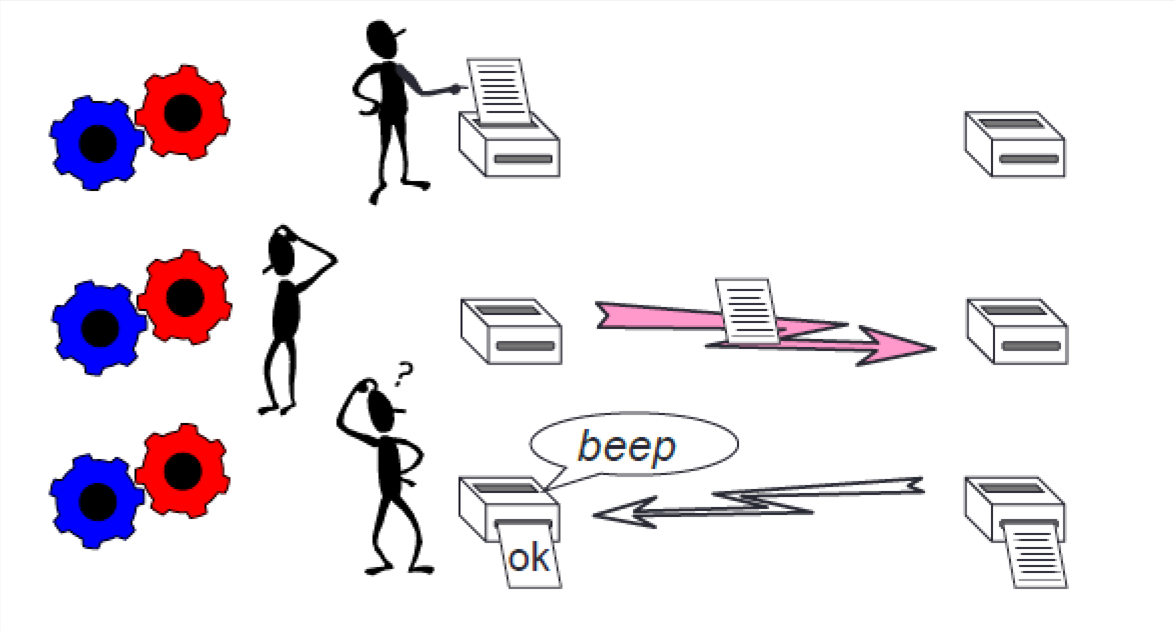
• `buf`	Specifies the starting address of the buffer.
• `count`	Indicates the number of buffer elements
• `dtype`	Denotes the datatype of the buffer elements
• `source`	Specifies the rank of the source process in the group associated with the communicator comm
• `tag`	Denotes the message label
• `comm`	Designates the communication context that identifies a group of processes
• `status`	Returns information about the received message

Example

- …
- if (rank == 0)
- {
- for (i=0; i<10; i++) buffer[i] = i;
- MPI_Send(buffer, 10, MPI_INT, 1, 123, MPI_COMM_WORLD);
- }
- else if (rank == 1)
- {
- for (i=0; i<10; i++)buffer[i] = -1;
- MPI_Recv(buffer, 10, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
- for (i=0; i<10; i++)
- if (buffer[i] != i)
- printf("Error: buffer[%d] = %d but is expected to be %d\n", i, buffer[i], i);
- }
- …

Non-Blocking Operations

- Non-blocking operations return immediately and allow the sub-program to perform other work.



Non-blocking Message Passing

- Returns immediately after the data transferred is initiated
- Allows to overlap computation with communication
- Need to be careful though
  - When send and receive buffers are updated before the transfer is over, the result will be wrong

Non-blocking Message Passing

- MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *req);
- MPI_Recv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Request *req);
- MPI_Wait(MPI_Request *req, MPI_Status *status);

• req	Specifies the request used by a completion routine when called by the application to complete the send operation.

Blocking	MPI_Send	MPI_Bsend	MPI_Ssend	MPI_Rsend	MPI_Recv
Non-blocking	MPI_Isend	MPI_Ibsend	MPI_Issend	MPI_Irsend	MPI_Irecv

Non-blocking Message Passing

```
…
right = (rank + 1) % nproc;
left = rank - 1;
if (left < 0)          left = nproc - 1;
MPI_Irecv(buffer, 10, MPI_INT, left, 123,
 MPI_COMM_WORLD, &request);
MPI_Isend(buffer2, 10, MPI_INT, right, 123,
 MPI_COMM_WORLD, &request2);
MPI_Wait(&request, &status);
MPI_Wait(&request2, &status);
…
```

How to execute MPI codes?

- The implementation supplies scripts to launch the MPI parallel calculation
- **Mpirun –np #proc a.out**
- **Mpiexec –n #proc a.out**

- A copy of the same program runs on each processor core within its own process (private address space)

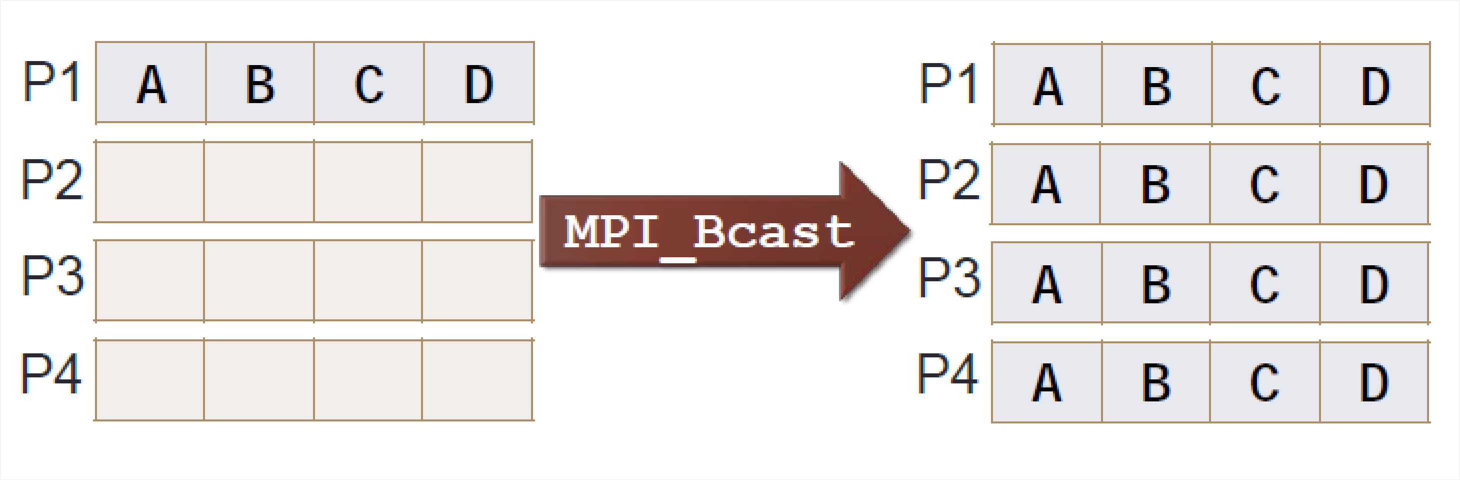
Collective Communications

- A single call handles the communication between all the processes in a communicator
- There are 3 types of collective communications
  - Data movement (e.g. MPI_Bcast)
  - Reduction (e.g. MPI_Reduce)
  - Synchronization (e.g. MPI_Barrier)

Broadcast

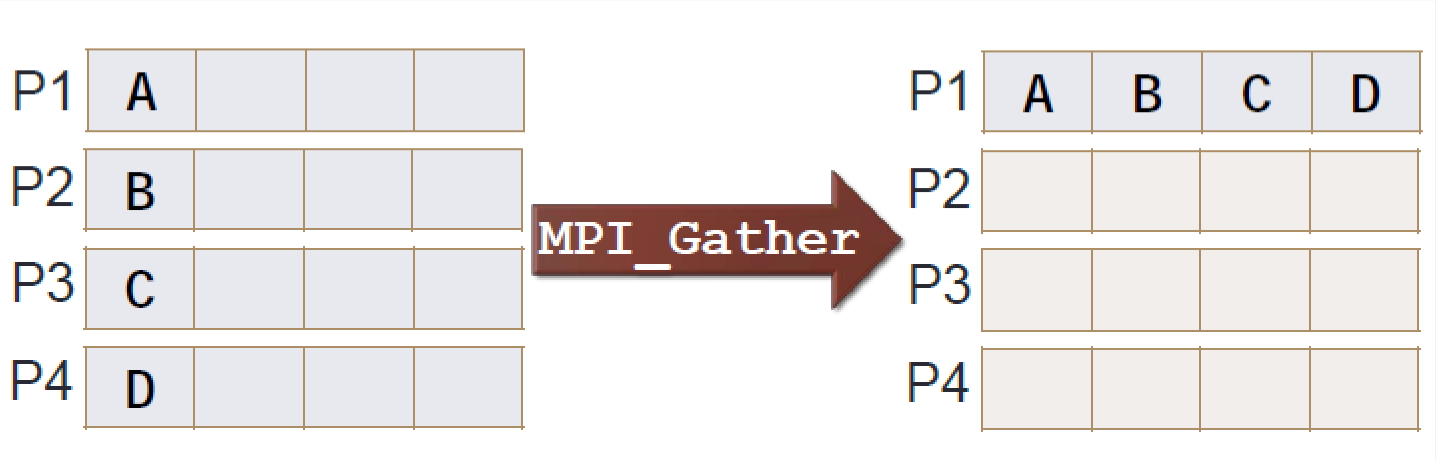
- A one-to-many communication.

- **Int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);**
- One process (root) sends data to all the other processes in the same communicator
- Must be called by all the processes with the same arguments



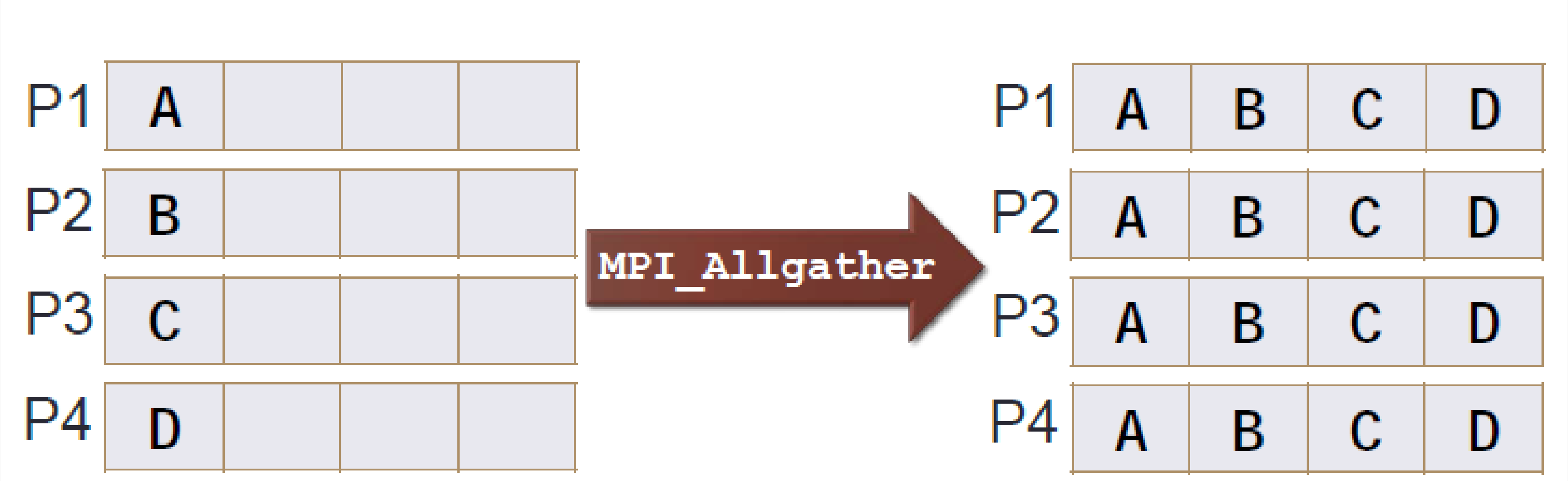
Gather

- **Int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)**
- One process (root) collects data to all the other processes in the same communicator
- Must be called by all the processes with the same arguments



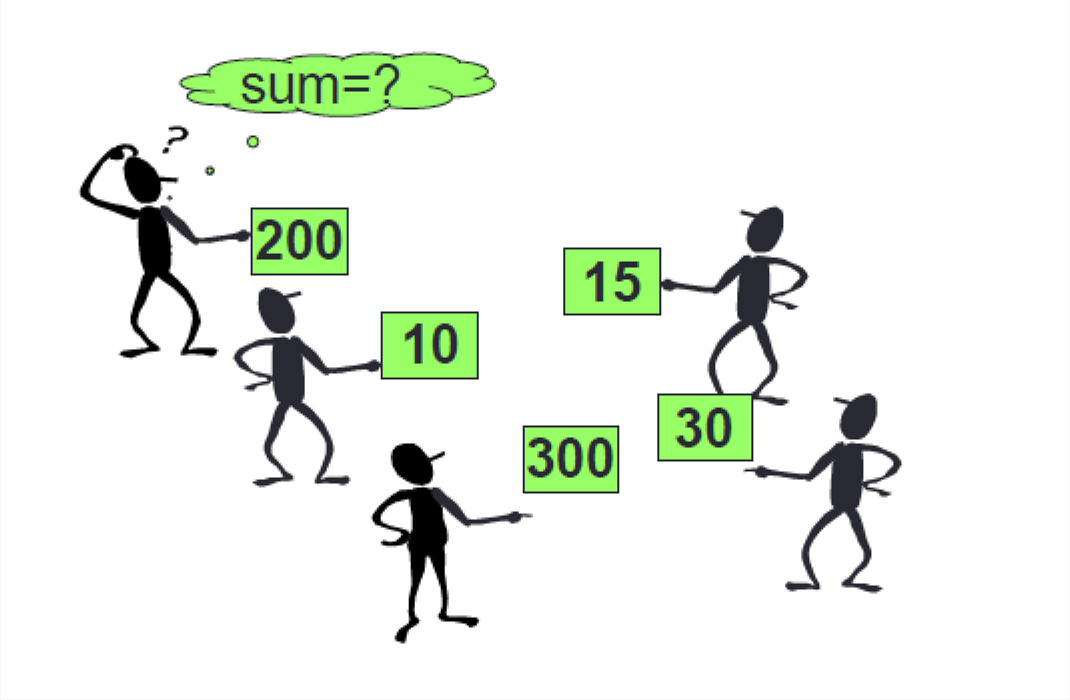
Gather to All

- **int MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)**
- All the processes collects data to all the other processes in the same communicator
- Must be called by all the processes with the same arguments



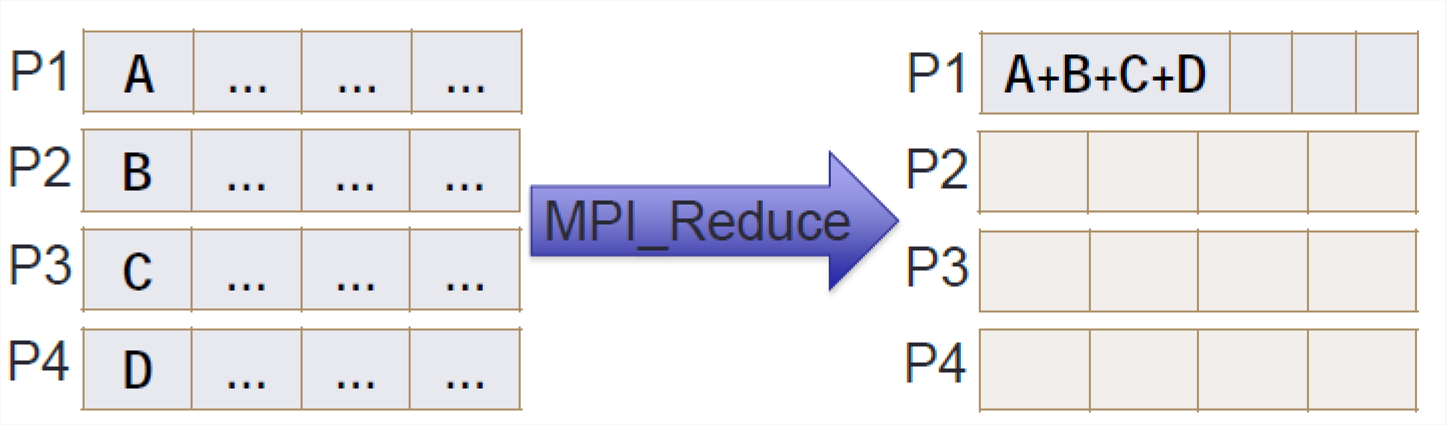
Reduction Operations

- Combine data from several processes to produce a single result.

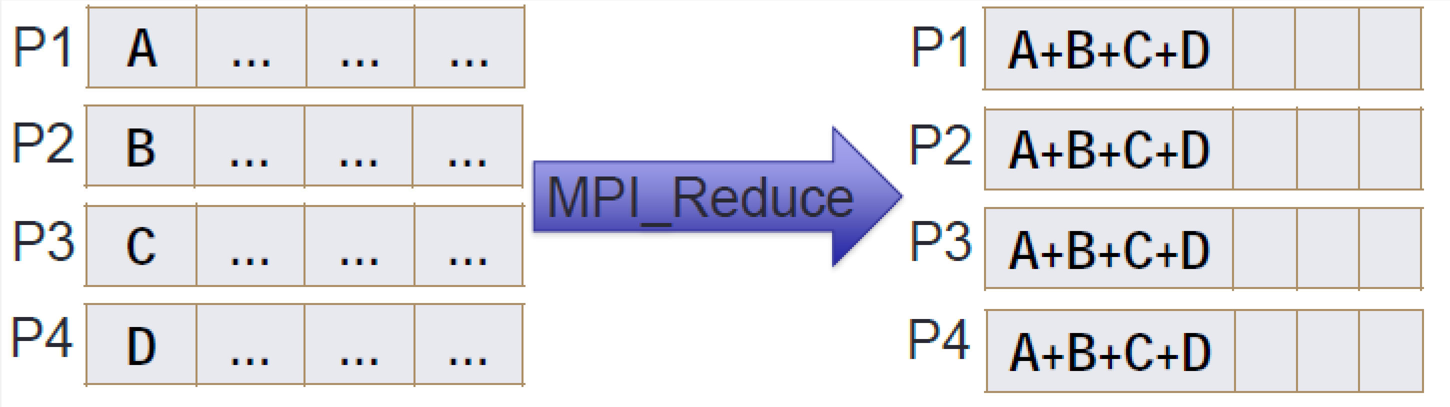


Reduction

- **int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)**
- One process (root) collects data to all the other processes in the same communicator, and performs an operation on the data
- MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
- **MPI_Op_create()**: User defined operator

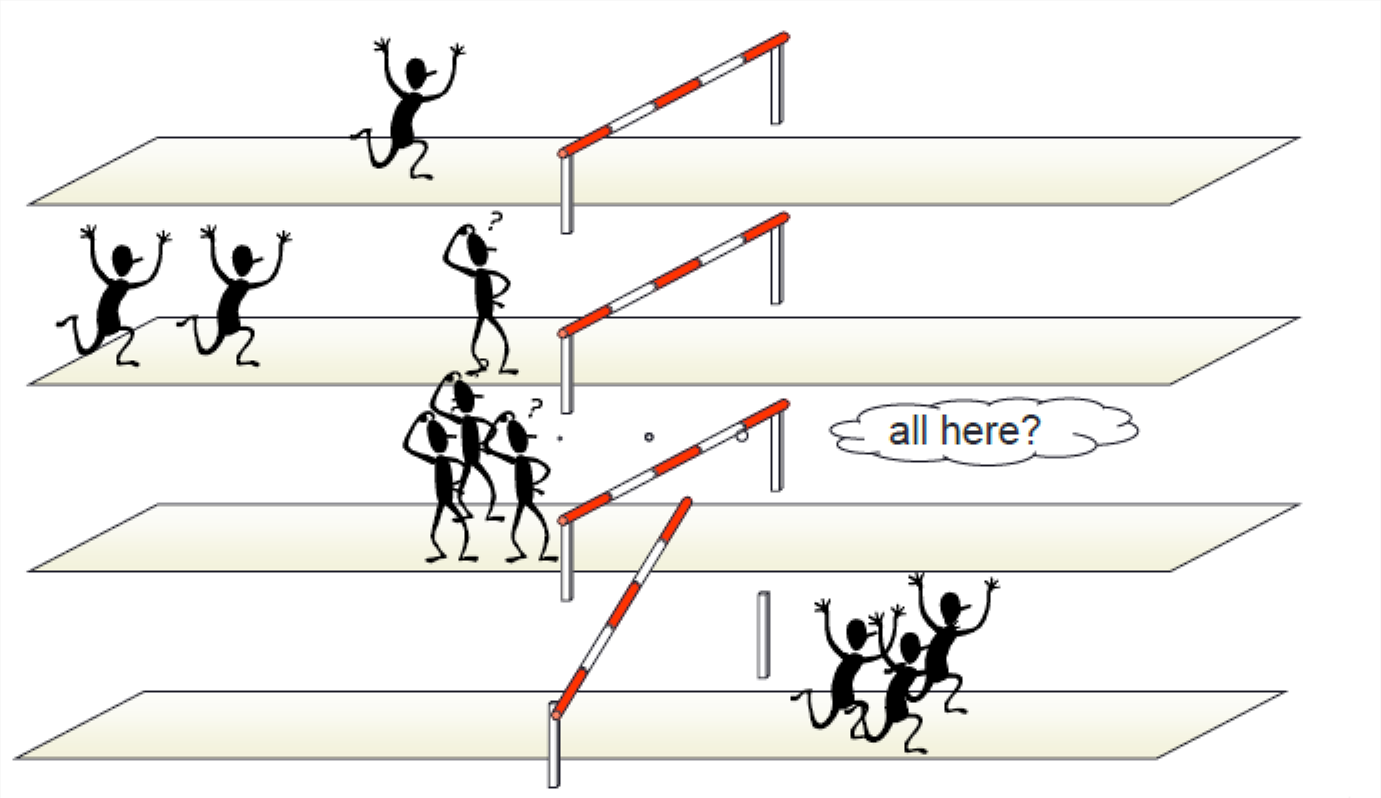


- **int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)**
- All the processes collect data to all the other processes in the same communicator, and perform an operation on the data
- MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
- **MPI_Op_create()**: User defined operator



Barriers

- Synchronize processes.



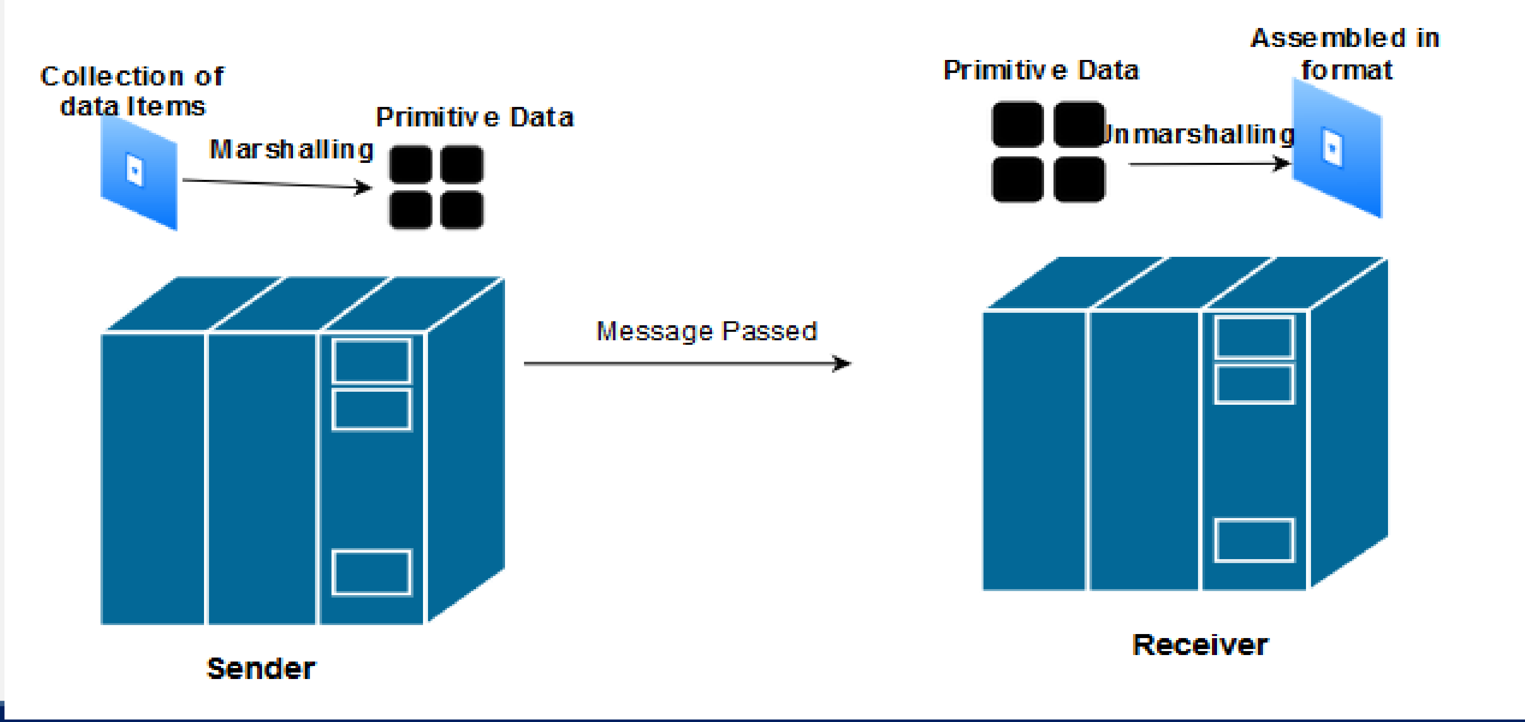
Synchronization

```
int MPI_Barrier(MPI_Comm comm)
```

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
    int rank, nprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Hello, world.  I am %d of %d\n", rank,
 nprocs);
    MPI_Finalize();
    return 0;
}
```

Marshalling

- Process of taking collection of data items and assembling them in order to transmit
- Unmarshalling is the reverse of marshalling



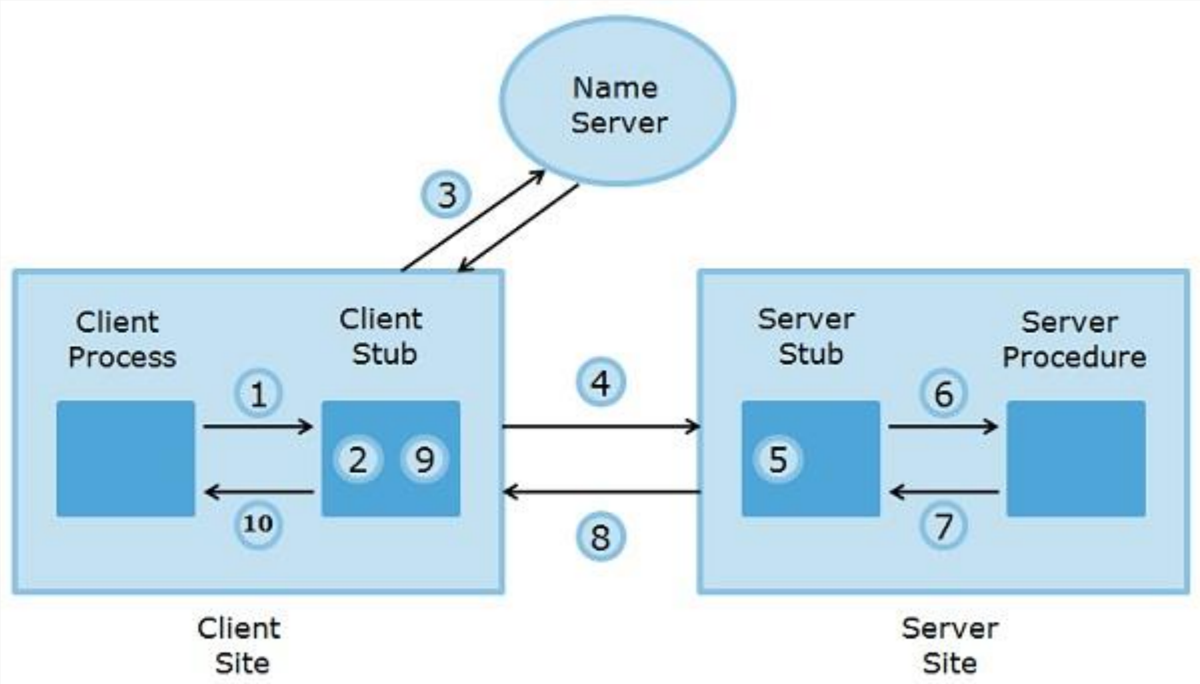
References

- http://www.mpi-forum.org
- http://www.llnl.gov/computing/tutorials/mpi/
- http://www.nersc.gov/nusers/help/tutorials/mpi/intro/
- http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html
- http://www-unix.mcs.anl.gov/mpi/tutorial/
- MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/)
- Open MPI (http://www.open-mpi.org/)

- MPI descriptions and examples are referred from
  – http://mpi.deino.net/mpi_functions/index.htm
  – *Stéphane Ethier (PPPL)'s PICSciE/PICASso Mini-Course Slides*

Case Study (RPC and Java RMI)

RPC



Name
Server
3
Client
Process
Client
Stub
1
2
9
10
4
8
Server
Stub
5
Server
Procedure
6
7
Client
Site
Server
Site

RMI

- The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java.
- The RMI allows an object to invoke methods on an object running in another JVM.
- The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

…Continued…

- A **remote object** is an object whose method can be invoked from another JVM.
- stub
  - The stub is an object, acts as a gateway for the client side.
  - All the outgoing requests are routed through it.
  - It resides at the client side and represents the remote object.

What happens when the caller invokes method on the stub object?

- It initiates a connection with remote Virtual Machine (JVM),
- It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
- It waits for the result
- It reads (unmarshals) the return value or exception, and
- It finally, returns the value to the caller.

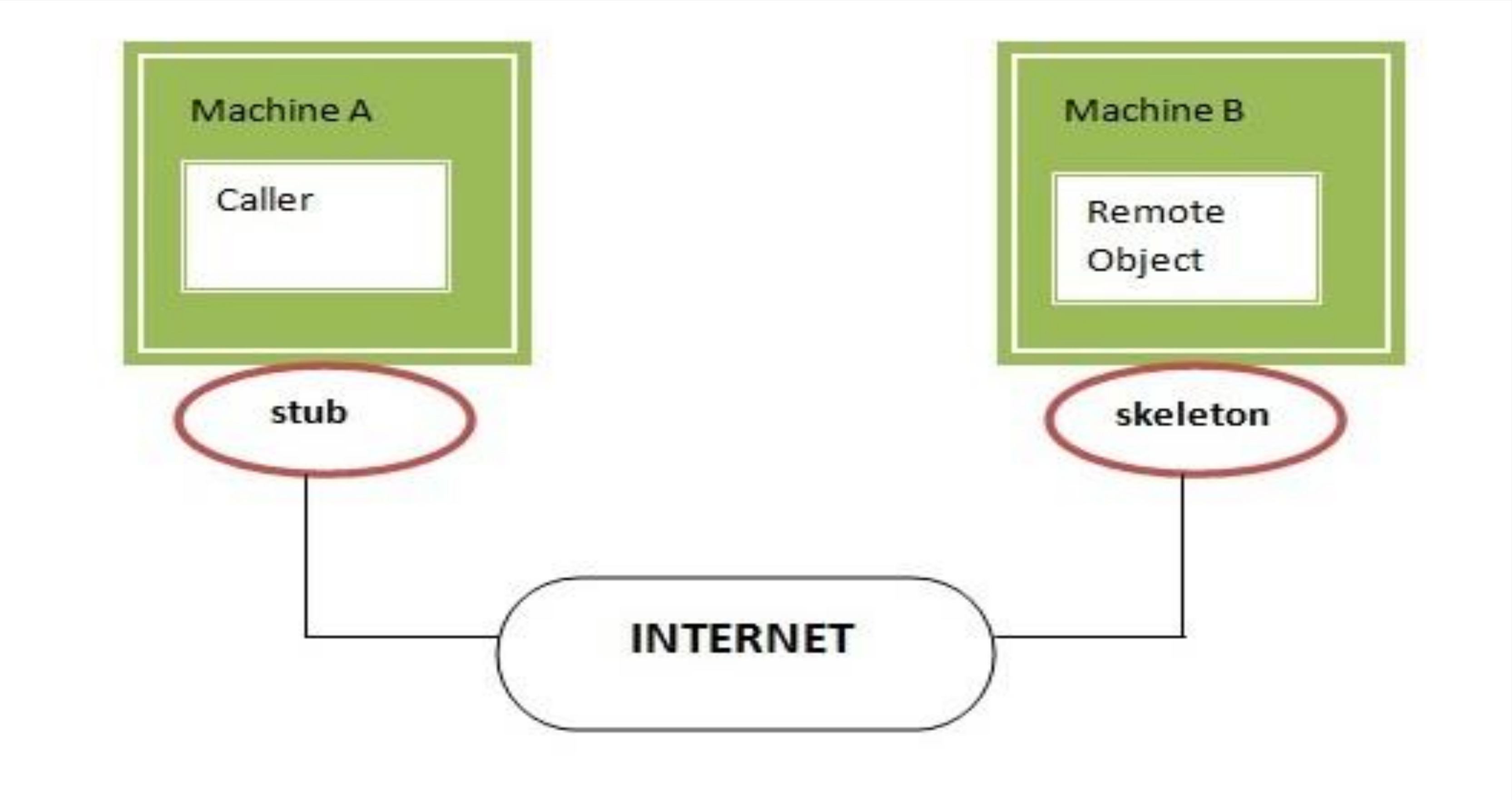
skeleton

- The skeleton is an object, acts as a gateway for the server side object.
- All the incoming requests are routed through it.

Upon receiving the incoming request

- It reads the parameter for the remote method
- It invokes the method on the actual remote object, and
- It writes and transmits (marshals) the result to the caller.

...Continued...



Machine A
Caller
Machine B
Remote
Object
stub
skeleton
INTERNET

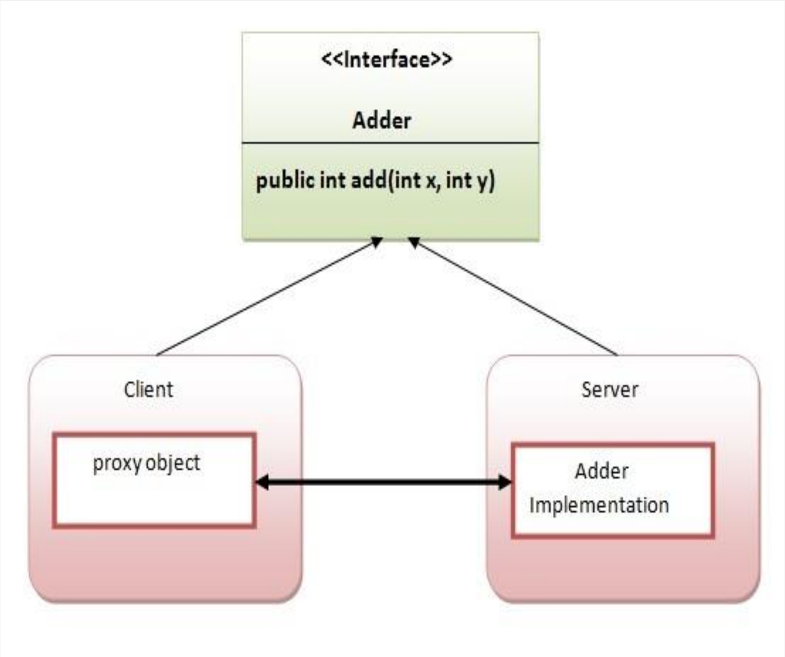
Distributed applications Requirements

- If any application performs these tasks, it can be distributed application.
  - The application need to locate the remote method
  - It need to provide the communication with the remote objects, and
  - The application need to load the class definitions for the objects.
- The RMI application have all these features, so it is called the distributed application.

Java RMI Example

- 6 steps to write the RMI program
  - Create the remote interface
  - Provide the implementation of the remote interface
  - Compile the implementation class and create the stub and skeleton objects using the rmic tool
  - Start the registry service by rmiregistry tool
  - Create and start the remote application
  - Create and start the client application

- The client application need only two files, remote interface and client application.
- In the rmi application, both client and server interacts with the remote interface.
- The client application invokes methods on the proxy object, RMI sends the request to the remote JVM.
- The return value is sent back to the proxy object and then to the client application.



1) create the remote interface

```
import java.rmi.*;
public interface Adder extends Remote
{
public int add(int x,int y)throws RemoteException;
}
```

2) Provide the implementation of the remote interface

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderRemote extends UnicastRemot
eObject implements Adder{
AdderRemote()throws RemoteException{
super();
}
public int add(int x,int y){return x+y;}
}
```

3) create the stub and skeleton objects using the rmic tool.

- The rmic tool invokes the RMI compiler and creates stub and skeleton objects
- rmic AdderRemote

4) Start the registry service by the rmiregistry tool

- start the registry service by using the rmiregistry tool.

rmiregistry 5000

5000 – port number

5) Create and run the server application

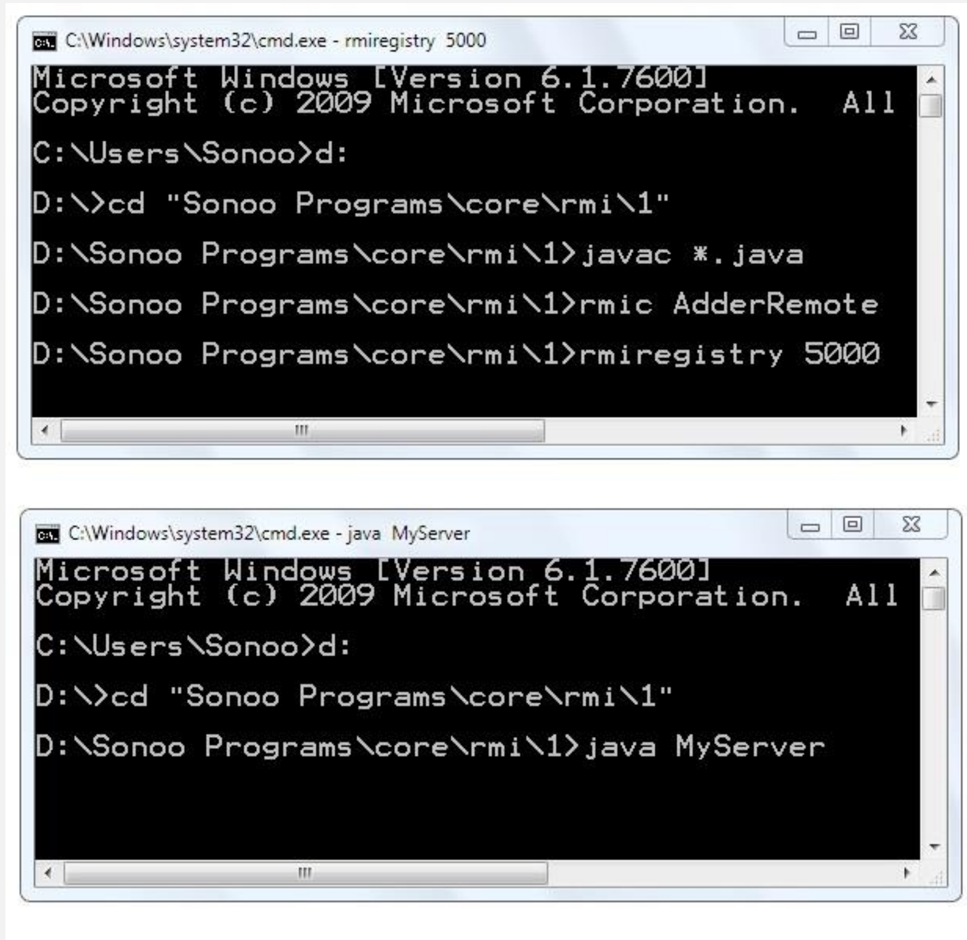
```java
import java.rmi.*;
import java.rmi.registry.*;
public class MyServer{
public static void main(String args[]){
try{
Adder stub=new AdderRemote();
Naming.rebind("rmi://localhost:5000/sonoo",stub);
}catch(Exception e){System.out.println(e);}
}
}
```

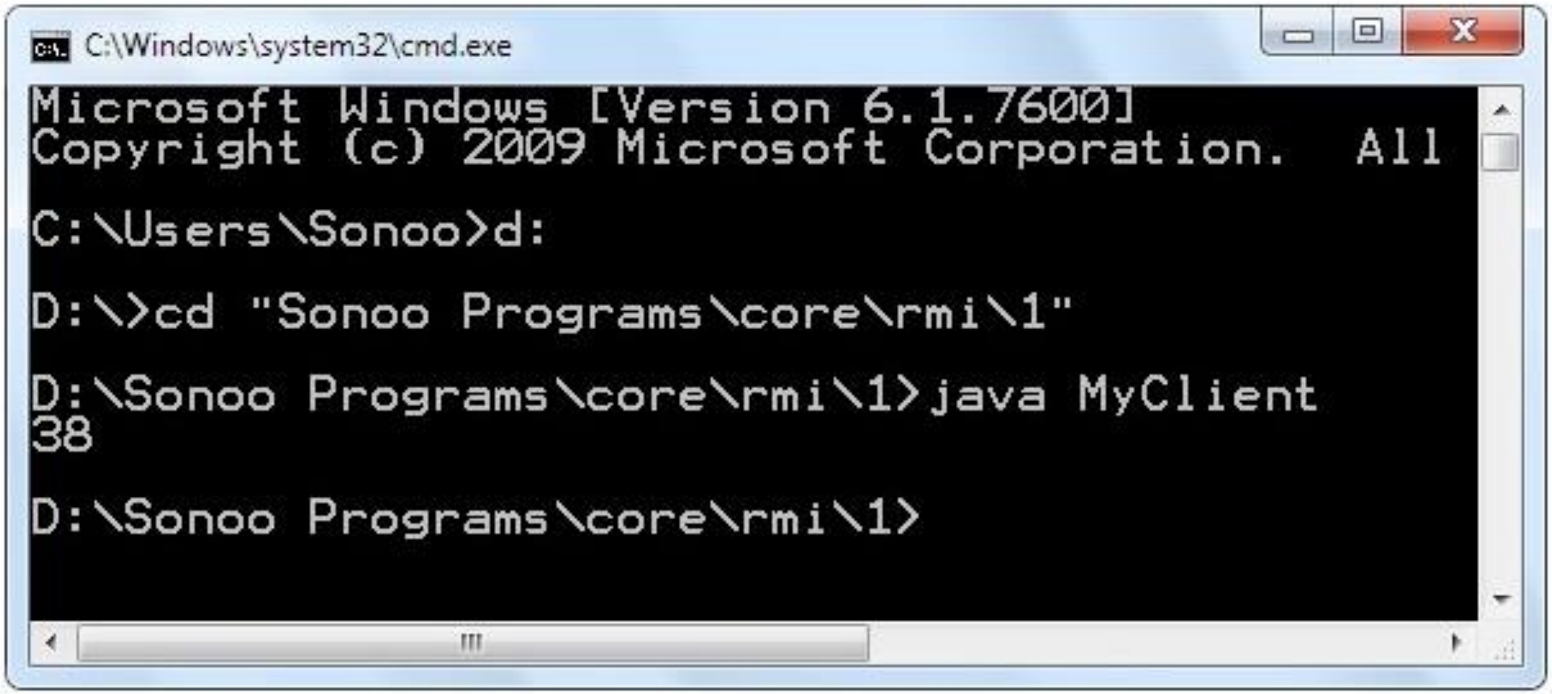
6) Create and run the client application

```
import java.rmi.*;
public class MyClient{
public static void main(String args[]){
try{
Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
System.out.println(stub.add(34,4));
}catch(Exception e){}
}
}
```

- For running **this** rmi example,
- 1) compile all the java files : javac *.java
- 2)create stub and skeleton object by rmic tool : rmic AdderRemote
- 3)start rmi registry in one command prompt :

  rmiregistry 5000
- 4)start the server in another command prompt :
- java MyServer
- 5)start the client application in another command prompt : java MyClient


C:\Windows\system32\cmd.exe - rmiregistry 5000
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All
C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>javac *.java
D:\Sonoo Programs\core\rmi\1>rmic AdderRemote
D:\Sonoo Programs\core\rmi\1>rmiregistry 5000
C:\Windows\system32\cmd.exe - java MyServer
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All
C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyServer


C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All
C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyClient
38
D:\Sonoo Programs\core\rmi\1>