

# Distributed Computing System Lab Manual

Subject Code: CSE  
Class: VI Semester(CSE)

Prepared By Mr. Biswajit Sarma  
Assistant Professor



**Department of Computer Science & Engineering**  
**JORHAT ENGINEERING COLLEGE**  
**JORHAT : 785007, ASSAM**

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

**Vision of the Department**

To become a prominent department of Computer Science and Engineering for producing quality human resources to meet the needs of the industry and society

**Mission of the Department**

- 1: To impart quality education through well-designed curriculum and academic facilities to meet the computing needs of the industry and society
- 2: To inculcate the spirit of creativity, team work, innovation, entrepreneurship and professional ethics among the students
- 3: To facilitate effective interactions to foster networking with alumni, industries, institutions of learning and research and other stake-holders
- 4: To promote research and continuous learning in the field of Computer Science and Engineering

**OBJECTIVE:**This lab complements the Distributed Computing System course. Students will gain practical experience with basic techniques in the design and development of Distributed Systems and understanding solutions of the fundamental problems in distributed systems like mutual exclusion, deadlock detection, termination detection, RPC, RMI, OPENMP, MPI and CORBA.

**Program Outcomes**

PO1	Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

PO4	Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

**Program Specific Outcomes**

PSO1	Gain ability to employ modern computer languages, environments and platforms in creating innovative career paths
PSO2	Achieve an ability to implement, test and maintain computer based system that fulfils the desired needs

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

**Distributed Computing System Lab Syllabus**  
**(Practical Hours: 06, Credits: 00)**

**Implement the following programs using C Language in Linux Platform.**

Exp. No.	List of Experiments	Page No.
1.	To Simulate the functioning of Lamport's Logical clock in 'c'.	5-6
2.	To Simulate the functioning of Lamport's Vector clock in 'c'.	7
3.	To Simulate the Distributed Mutual exclusion in 'c'.	8
4.	To Simulate the Non Token/ Token based algorithm in Distributed system.	9
5.	To Simulate the Distributed Deadlock Detection algorithm-Edge chasing.	10
6.	To Implement 'RPC' mechanism for accessing methods of remote systems.	11-13
7.	To Implement 'Java RMI' mechanism for accessing methods of remote systems.	14-16
8.	To implement CORBA mechanism by using C++ program at one end and Java Program on the other.	17-35
9.	Experiment with the application programming interface OpenMP which supports multi-platform shared-memory and multiprocessing programming in C	36-68
10.	Experiment with Message Passing Interface Standard (MPI).	69-91

**Experiment No:1**

**AIM: To Simulate the functioning of Lamport's Logical clock in 'c'.**

**Prerequisites:** Global Clock Concept, C programming

**Outcome:** Program will exhibit the simulation of Lamport's logical clock behaviour based on two rules.

**Description:** The "time" concept in distributed systems -- used to order events in a distributed system.

Assumption:

- O The execution of a process is characterized by a sequence of events. An event can be the execution of one instruction or of one procedure.
- O Sending a message is one event, receiving a message is one event.
- O The events in a distributed system are not total chaos. Under some conditions, it is possible to ascertain the order of the events. Lamport's logical clocks try to catch this.

**Lamport's 'happened before' relation**

The "happened before" relation ( $\otimes$ ) is defined as follows:

O  $A \otimes B$  if A and B are within the same process (same sequential thread of control) and A occurred before B.

O  $A \otimes B$  if A is the event of sending a message M in one process and B is the event of receiving M by another process.

O if  $A \otimes B$  and  $B \otimes C$  then  $A \otimes C$ .

Event A causally affects event B iff  $A \otimes B$ .

Distinct events A and B are concurrent ( $A || B$ ) if we do not have  $A \otimes B$  or  $B \otimes A$ .

**Algorithm:**

$C_i$  is the local clock for process  $P_i$

O if a and b are two successive events in  $P_i$ , then  $C_i(b) = C_i(a) + d_1$ , where  $d_1 > 0$

O if a is the sending of message m by  $P_i$ , then m is assigned timestamp  $t_m = C_i(a)$

O if b is the receipt of m by  $P_j$ , then  $C_j(b) = \max\{C_j(b), t_m + d_2\}$ , where  $d_2 > 0$

**Expected Outcome:**

Enter the no. of physical clocks: 2

No. of nodes for physical clock 1: 2

Enter the name of process: a

Enter the name of process: b

No. of nodes for physical clock 2: 2

Enter the name of process: c

Enter the name of process: d

Press a key for watching timestamp of physical clocks

Physical Clock 1

Process a has P.T.: 6

Process b has P.T.: 7

Physical Clock 2

Process c has P.T.: 2

Process d has P.T.: 3

Press a key for watching timestamp of logical clocks

Logical Clock Timestamp for process a: 6

Logical Clock Timestamp for process b: 13

Logical Clock Timestamp for process c: 18

Logical Clock Timestamp for process d: 23

**Experiment No:2**

**AIM: To Simulate the functioning of Lamport's Vector clock in 'c'.**

**Prerequisites:** Vector clock functioning, C programming

**Outcome:** Causal ordering for transitive processes

**Description:** Vector Clocks are used in distributed systems to determine whether pairs of events are causally related. Using Vector Clocks, timestamps are generated for each event in the system, and their causal relationship is determined by comparing those timestamps.

The timestamp for an event is an n-tuple of integers, where n is the number of processes.

Each process assigns a timestamp to each event. The timestamp is composed of that process logical time and the last known time of every other process.

**Algorithm:**

$t_a < t_b$  If and only if they meet two conditions:

- 1.They are not equal timestamps ( there exists i,  $t_a[i] \neq t_b[i]$ ) and
- 2.each  $t_a[i]$  is less than or equal to  $t_b[i]$  (for all i,  $t_a[i] \leq t_b[i]$ )

**Expected Output:**

Process Vector

p1[13497767650101114]

p2[13497767650101114]

p3[13497767650101120]

**Experiment No:3**

**AIM: To Simulate the Distributed Mutual exclusion in 'c'.**

**Prerequisites:** Mutual exclusion concepts, C programming

**Description:** Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner. Only one process is allowed to execute the critical section (CS) at any given time. In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion.


**Algorithm:**

Process 1: Request resource:

Resource Allocated  No more requests process for this resource.

Process2: Request Resource  Denied

Process 1: Exit Resource:

Process2: Request Resource  Allocated

**Expected Output:**

Press a key (except q) to enter a process into critical section. Press q at any time to exit.

Process 0 entered critical section.

Error: Another process is currently executing critical section. Please wait till its execution is over.

Process 0 exits critical section.

Process 1 entered critical section.

Process 1 exits critical section.

Process 2 entered critical section.

Error: Another process is currently executing critical section. Please wait till its execution is over.

Process 2 exits critical section.



**Experiment No:4**

**AIM: To Simulate the Non Token/ Token based algorithm in Distributed system-Lamport's.**

**Prerequisites:** Understanding of Non-Token based algorithm, Java programming

**Description:** Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks. Every site  $S_i$  keeps a queue, request queue  $i$ , which contains mutual exclusion requests ordered by their timestamps. This algorithm requires communication channels to deliver messages the FIFO order.

**Algorithm:**

Requesting the critical section:

When a site  $S_i$  wants to enter the CS,

it broadcasts a REQUEST( $t_{si}, i$ ) message to all other sites and places the request on request queue  $i$ . ( $(t_{si}, i)$  denotes the timestamp of the request.)

When a site  $S_j$  receives the REQUEST( $t_{si}, i$ ) message from site  $S_i$ ,

places site  $S_i$ 's request on request queue  $j$  and it returns a timestamped REPLY message to  $S_i$ .

Executing the critical section: Site  $S_i$  enters the CS when the following two conditions hold:

L1:  $S_i$  has received a message with timestamp larger than  $(t_{si}, i)$  from all other sites.

L2:  $S_i$ 's request is at the top of request queue  $i$ .

**Expected Output:** The process which would have next to on ring would access resource.

**Experiment No:5**

**AIM: To Simulate the Distributed Deadlock Detection algorithm-Edge chasing.**

**Prerequisites:** Deadlock knowledge, C compiler,

**Description & Algorithm:**

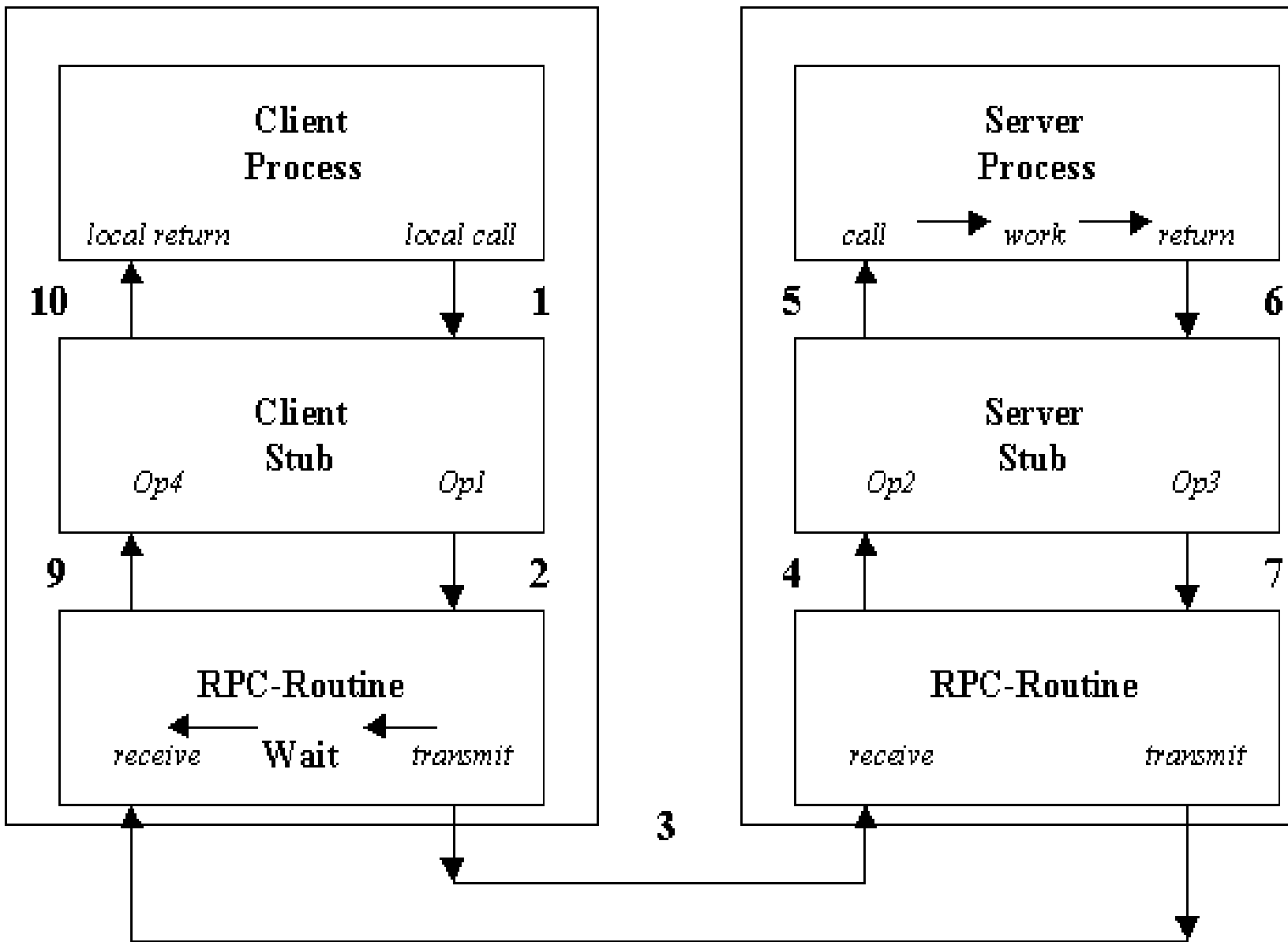
Whenever a process A is blocked for some resource, a probe message is sent to all processes A may depend on. The probe message contains the process id of A along with the path that the message has followed through the distributed system. If a blocked process receives the probe it will update the path information and forward the probe to all the processes it depends on. Non-blocked processes may discard the probe.

If eventually the probe returns to process A, there is a circular waiting loop of blocked processes, and a deadlock is detected. Efficiently detecting such cycles in the “wait-for graph” of blocked processes is an important implementation problem.

**Expected Output:** The status of the system as deadlocked or not.

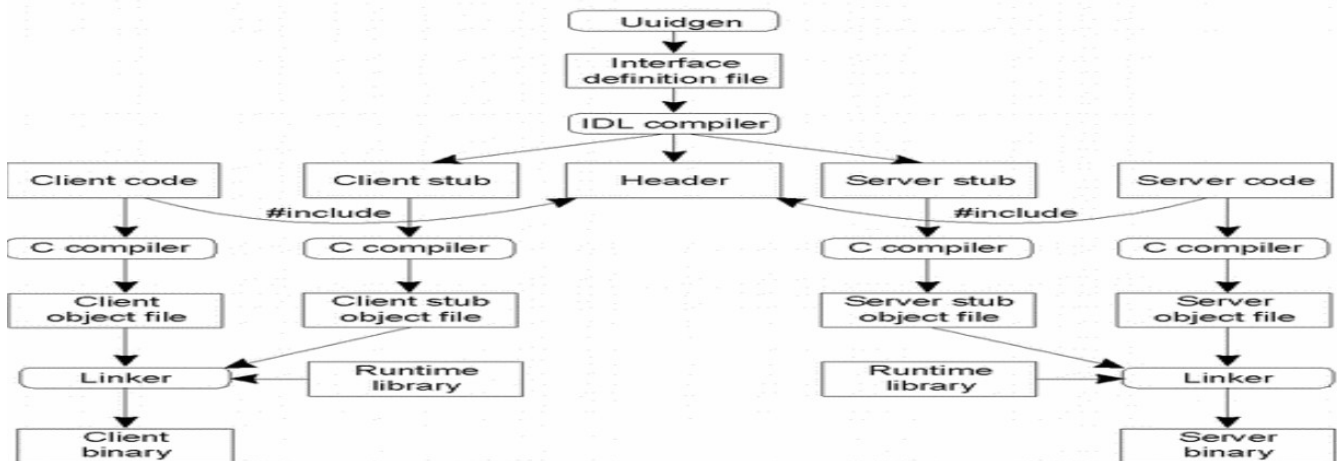
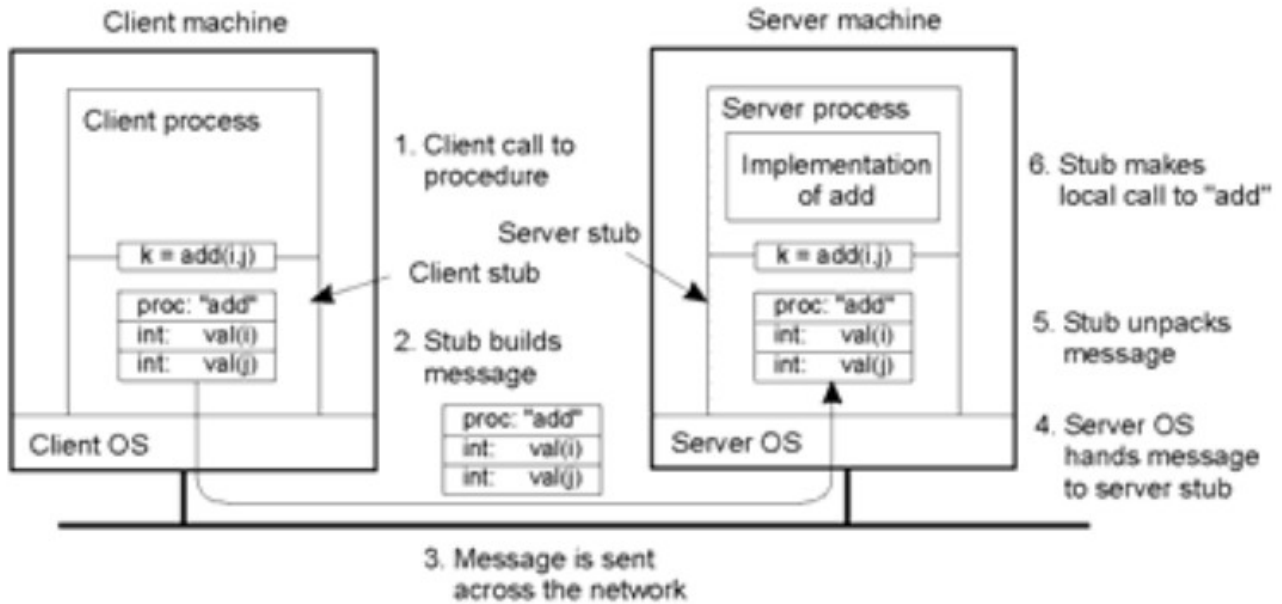
**Experiment No:6**

**AIM: To Implement 'RPC' mechanism for accessing methods of remote systems.**



**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---



### Example of rpcgen file

```
program DATEPROG { /* remote program name (not used)*/  
    version DATEVERS { /* declaration of program version number*/  
        long BINDATE(int) = 1; /*prototype of the function, procedure number = 1 */  
        } = 1; /* definition of program version =1*/  
    } = 0x3012225; /* remote program number (must be unique)*/
```

### How to compile:

```
$rpcgen -a test.x // -a          generate all files, including samples
```

Modify the test\_client.c and test\_server.c according to your requirement. Use the Makefile to build the executables. For Makfile use the following command

```
$make -f Makefile
```

Assignment: Implement Multiplication of two numbers in Remote Procedure Call.

**Experiment No:7**

**AIM: To Implement 'Java RMI' mechanism for accessing methods of remote systems.**

**Prerequisites:** Knowledge of remoting, Java.

**Description:** The Java Remote Method Invocation (Java RMI) is a Java API that performs remote method invocation, the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage collection.

1. The original implementation depends on Java Virtual Machine (JVM) class representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Remote Method Protocol (JRMP).
2. In order to support code running in a non-JVM context, a CORBA version was later developed.

Usage of the term RMI may denote solely the programming interface or may signify both the API and JRMP, IIOP, or another implementation, whereas the term RMI-IIOP (read: RMI over IIOP) specifically denotes the RMI interface delegating most of the functionality to the supporting CORBA implementation.

The basic idea of Java RMI, the distributed garbage-collection (DGC) protocol, and much of the architecture underlying the original Sun implementation, come from the 'network objects' feature of Modula-3.

Sample code:

Code:

**RMI SERVER:**

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.*;
public class RmiServer extends UnicastRemoteObject implements
RmiServerIntf
{
    public static final String MESSAGE = "Hello World";
    public RmiServer() throws RemoteException
```

```
{
    super(0);
    // required to avoid the 'rmic' step, see below
}
public String getMessage()
{
    return MESSAGE;
}
public static void main(String args[]) throws Exception
{
    System.out.println("RMI server started");
    try
    { //special exception handler for registry creation
        LocateRegistry.createRegistry(1099);
        System.out.println("java RMI registry created.");
    }
    catch (RemoteException e)
    {
        //do nothing, error means registry already exists
        System.out.println("java RMI registry already exists.");
    }
    //Instantiate RmiServer
    RmiServer obj = new RmiServer();
    // Bind this object instance to the name "RmiServer"
    Naming.rebind("//localhost/RmiServer", obj);
    System.out.println("PeerServer bound in registry");
}
}
```

## **INTERFACE**

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface RmiServerIntf extends Remote
{
```

```
    public String getMessage() throws RemoteException;
}
```

**CLIENT**

```
import java.rmi.Naming;
public class RmiClient
{
    public static void main(String args[]) throws Exception
    {
        RmiServerIntf obj =(RmiServerIntf)Naming.lookup("//localhost/RmiServer");
        System.out.println(obj.getMessage());
    }
}
```

**Expected Output:** Client can access the method of given by server using interface.

**Assignment:** Implement Multiplication of two numbers in JAVA RMI.



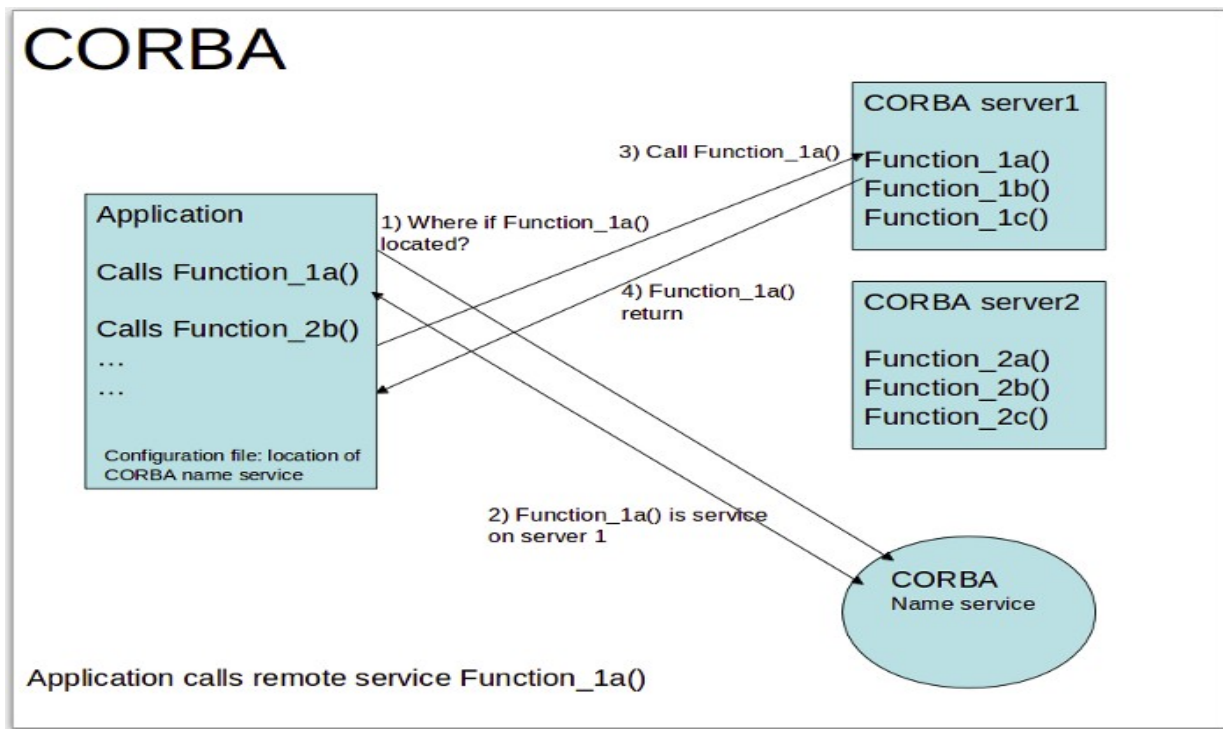
**Experiment No:8**

**AIM: To implement CORBA mechanism by using C++ program at one end and Java Program on the other.**

The Common Object Request Broker Architecture (CORBA) as defined by the [OMG](#) spec, allows clients to invoke operations on distributed objects (as defined by their IDL) without concern for their location, the programming language of the remote service, its OS, hardware platform (32 bit or 64 bit word size) or communications protocols (TCP/IP, IPX, FDDI, ATM,...).

CORBA is a support framework of applications, libraries and services for making distributed procedure calls. A program on computer "C" (CORBA client) calls a function which is computed and processed on computer node "S" (CORBA server) and passes to it function arguments. The function/method passes arguments and returns values as with any other C/C++ call except that it may be distributed across the network so that portions of the program may be executed on a remote machine.

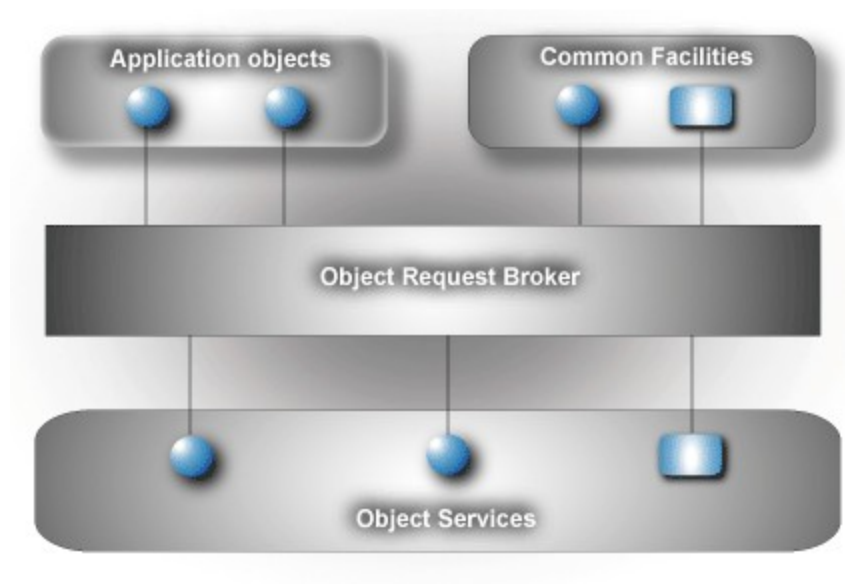
CORBA's strength is that it allows platform and programming language interoperability. The interface between the client and server is defined by the CORBA IDL language which can be processed to produce code to support a variety of languages and platforms. The CORBA communication protocol, the language mappings and object model are standardized to allow this general inter-operability.



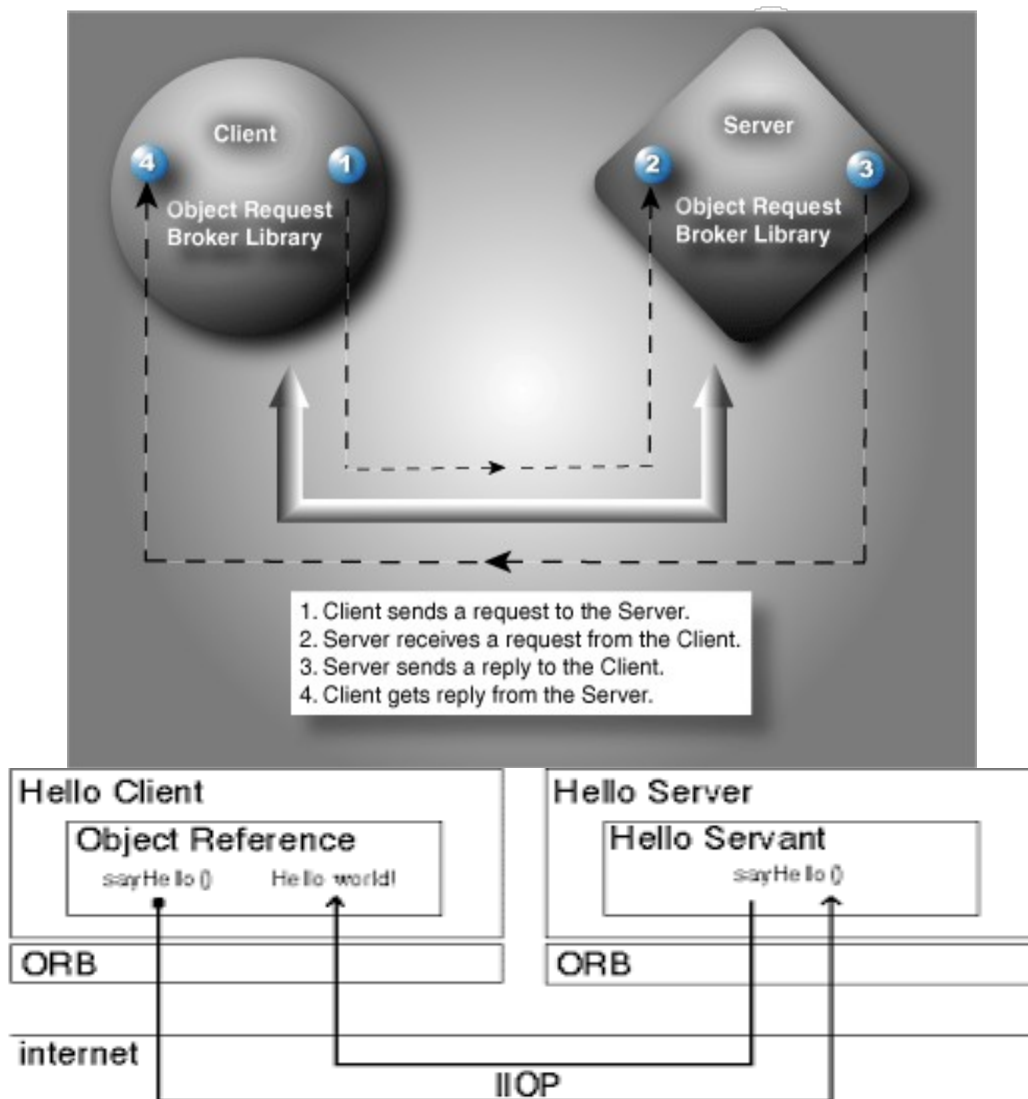
1. Application calls Function\_1a()
2. CORBA name service locates Function\_1a() on "server 1"
3. Application requests a call to Function\_1a() on "server 1"
4. Execution of Function\_1a() on "server 1" returns a function return value and returns an argument list.

What is CORBA (Executive summary)

- ORB: Object Request Broker = manages remote access to objects
- CORBA: Common ORB Architecture = software bus for distributed objects. CORBA specifies a system which provides interoperability between objects in a heterogeneous, distributed environment and in a way transparent to the programmer. Its design is based on OMG Object Model.
- CORBA provides a framework for distributed OO programming—remote objects are (nearly) transparently accessible from the local program—uses the client-server paradigm—platform and language independent.
- “an OO version of RPC”—but a framework rather than a technology .



Data communication from client to server is accomplished through a well-defined object-oriented interface. The Object Request Broker (ORB) determines the location of the target object, sends a request to that object, and returns any response back to the caller. Through this object-oriented technology, developers can take advantage of features such as inheritance, encapsulation, polymorphism, and runtime dynamic binding. These features allow applications to be changed, modified and re-used with minimal changes to the parent interface. The illustration below identifies how a client sends a request to a server through the ORB:



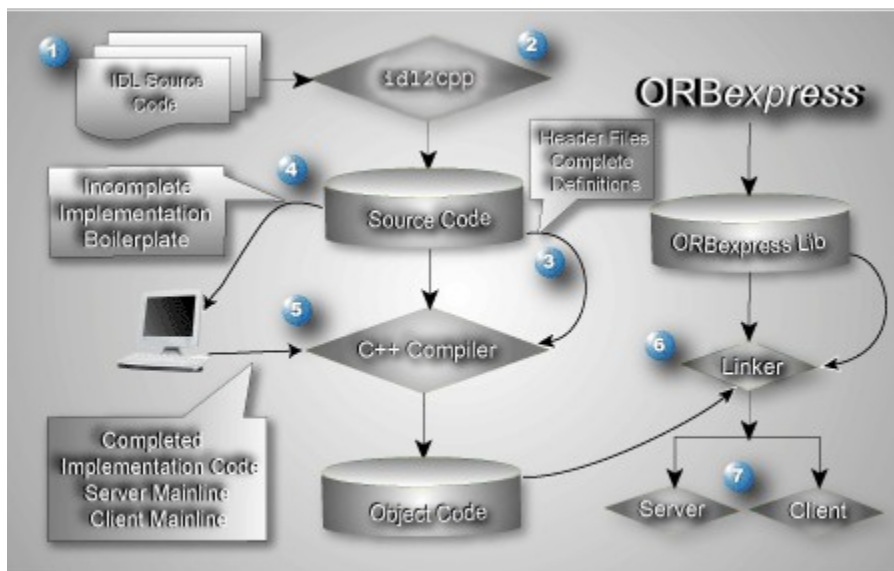
## **Interface Definition Language (IDL)**

A cornerstone of the CORBA standards is the Interface Definition Language. IDL is the OMG standard for defining language-neutral APIs and provides the platform-independent delineation of the interfaces of distributed objects. The ability of the CORBA environments to provide consistency between clients and servers in heterogeneous environments begins with a standardized definition of the data and operations constituting the client/server interface. This standardization mechanism is the IDL, and is used by CORBA to describe the interfaces of objects.

IDL defines the modules, interfaces and operations for the applications and is not considered a programming language. The various programming languages, such as Ada, C++, C#, or Java, supply the implementation of the interface via standardized IDL mappings.

### **Application Development Using ORBexpress**

The basic steps for CORBA development can be seen in the illustration below. This illustration provides an overview of how the IDL is translated to the corresponding language (in this example, C++), mapped to the source code, compiled, and then linked with the ORB library, resulting in the client and server implementation.



The basic steps for CORBA development include:

**1 Create the IDL to Define the Application Interfaces**

The IDL provides the operating system and programming language independent interfaces to all services and components that are linked to the ORB. The IDL specifies a description of any services a server component exposes to the client. The term "IDL Compiler" is often used, but the IDL is actually translated into a programming language.

**2 Translate the IDL**

An IDL translator typically generates two cooperative parts for the client and server implementation, stub code and skeleton code. The stub code generated for the interface classes is associated with a client application and provides the user with a well-defined Application Programming Interface (API). In this example, the IDL is translated into C++.

**3 Compile the Interface Files**

Once the IDL is translated into the appropriate language, C++ in this example, these interface files are compiled and prepared for the object implementation.

**4 Complete the Implementation**

If the implementation classes are incomplete, the spec and header files and complete bodies and definitions need to be modified before passing through to be compiled. The output is a complete client/server implementation.

**5 Compile the Implementation**

Once the implementation class is complete, the client interfaces are ready to be used in the client application and can be immediately incorporated into the client process. This client process is responsible for obtaining an object reference to a specific object, allowing the client to make requests to that object in the form of a method call on its generated API.

**6 Link the Application**

Once all the object code from steps three and five have been compiled, the object implementation classes need to be linked to the C++ linker. Once linked to the ORB library, in this example, *ORBexpress*, two executable operations are created, one for the client and one for the server.

**7 Run the Client and Server**

The development process is now complete and the client will now communicate with the server. The server uses the object implementation classes allowing it to communicate with the objects created by the client requests.

In its simplest form, the server must perform the following:

- Create the required objects.
- Notify the CORBA environment that it is ready to receive client requests.
- Process client requests by dispatching the appropriate servant.

### **Example:( I will use C++/JAVA for development of CORBA application here)**

Definition of the IDL for all remote methods. All distributed object have to defined in this file.

*example.idl*

```
interface ExampleInterface
{
    string send_message(in string message);
};
```

1. We want to write in [C++](#) and [Java](#). That is why, you need the C++ compiler *g++* and the Java JDK.
2. We need omniORB, omniidl and omniNames. Please install all this in your laptop.

### **C++ (omniORB)**

Files

- *server.cpp* server programm
- *client.cpp* client programm
- *MyExampleInterface\_impl* implementation of the class for distributed objects
- *MyExampleInterface\_impl.h* header for *Example\_impl.cpp*

### **Interface Implementation**

***MyExampleInterface\_impl.h***

```
#ifndef __MY_EXAMPLE_INTERFACE_IMPL_H__
#define __MY_EXAMPLE_INTERFACE_IMPL_H__

#include "example.hh"
```

```
class MyExampleInterface_impl : public POA_ExampleInterface
{
    public:
        virtual char * send_message(const char * message);
};

#endif // __MY_EXAMPLE_INTERFACE_IMPL_H__
```

### ***MyExampleInterface\_impl\_impl.cpp***

```
#include "MyExampleInterface_impl.h"
#include <iostream>

using namespace std;

char * MyExampleInterface_impl::send_message(const char * message)
{
    cout << "C++ (omniORB) server: " << message << endl;
    char * server = CORBA::string_alloc(42);
    strncpy(server, "Message from C++ (omniORB) server", 42);
    return server;
}
```

## **Server**

Implementation of the Server

### ***server.cpp***

```
#include "MyExampleInterface_impl.h"
#include <iostream>
#include <CORBA.h>
#include <Naming.hh>

/** Server name, clients needs to know this name */
#define SERVER_NAME "MyServerName"

using namespace std;

int main(int argc, char ** argv)
{
    try {

        -----
        //-----
        // Initialize CORBA ORB
```

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```

-----
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
-----
//-----
// Initialize POA: Get reference to root POA
//
client // Servant must register with POA in order to be made available for
// Get reference to the RootPOA.
//-----
CORBA::Object_var poa_obj = orb-
>resolve_initial_references("RootPOA");
PortableServer::POA_var poa =
PortableServer::POA::_narrow(poa_obj);
PortableServer::POAManager_var manager = poa->the_POAManager();
-----
//-----
// Create service
//-----
MyExampleInterface_impl * service = new MyExampleInterface_impl;
try {
-----
//-----
// Bind object to name service as defined by directive
InitRef // and identifier "NameService" in config file omniORB.cfg.
//-----
CORBA::Object_var ns_obj = orb-
>resolve_initial_references("NameService");
if (!CORBA::is_nil(ns_obj)) {
-----
//-----
// Narrow this to the naming context
//-----
CosNaming::NamingContext_ptr nc =
CosNaming::NamingContext::_narrow(ns_obj);
-----
//-----

```



**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```
requested by client.           // Bind to CORBA name service. Same name to be
                               //-----
                               CosNaming::Name name;
                               name.length(1);
                               name[0].id = CORBA::string_dup(SERVER_NAME);
                               name[0].kind = CORBA::string_dup("");
                               nc->rebind(name, service->_this());
                               //-----
                               // Intizialization ready, server runs
                               //-----
                               cout << argv[0] << " C++ (omniORB) server '" <<
SERVER_NAME << "' is running .." << endl;
                               }
                               } catch (CosNaming::NamingContext::NotFound &) {
                               cerr << "Caught CORBA exception: not found" << endl;
                               } catch (CosNaming::NamingContext::InvalidName &) {
                               cerr << "Caught CORBA exception: invalid name" << endl;
                               } catch (CosNaming::NamingContext::CannotProceed &) {
                               cerr << "Caught CORBA exception: cannot proceed" << endl;
                               }
                               //-----
                               // Activate the POA manager
                               //-----
                               manager->activate();
                               //-----
                               // Accept requests from clients
                               //-----
                               orb->run();
                               //-----
                               // Clean up
                               //-----
                               delete service;
```

```
----- //-----  
----- // Destroy ORB  
----- //-----  
-----  
orb->destroy();  
} catch (CORBA::UNKNOWN) {  
    cerr << "Caught CORBA exception: unknown exception" << endl;  
} catch (CORBA::SystemException &) {  
    cerr << "Caught CORBA exception: system exception" << endl;  
}  
}
```

## Client

Implementation of the client

### *client.cpp*

```
#include "example.hh"  
#include <iostream>  
#include <CORBA.h>  
#include <Naming.hh>  
  
/** Name is defined in the server.cpp */  
#define SERVER_NAME "MyServerName"  
  
using namespace std;  
  
int main(int argc, char ** argv)  
{  
    try {  
        //-----  
        // Initialize ORB object.  
        //-----  
        CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);  
        //-----  
        // Resolve service  
        //-----  
        ExampleInterface_ptr service_server = 0;
```

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```
try {
    //-----
    // Bind ORB object to name service object.
    // (Reference to Name service root context.)
    //-----
    CORBA::Object_var ns_obj = orb-
>resolve_initial_references("NameService");
    if (!CORBA::is_nil(ns_obj)) {
        //-----
        // Bind ORB object to name service object.
        // (Reference to Name service root context.)
        //-----
        CosNaming::NamingContext_ptr nc =
CosNaming::NamingContext::_narrow(ns_obj);
        //-----
        // The "name text" put forth by CORBA server in
name service.
        // This same name ("MyServerName") is used by the
CORBA server when
        // binding to the name server (CosNaming::Name).
        //-----
        CosNaming::Name name;
        name.length(1);
        name[0].id = CORBA::string_dup(SERVER_NAME);
        name[0].kind = CORBA::string_dup("");
        //-----
        // Resolve "name text" identifier to an object
reference.
        //-----
        CORBA::Object_ptr obj = nc->resolve(name);
        if (!CORBA::is_nil(obj)) {
            service_server =
ExampleInterface::_narrow(obj);
        }
    }
}
```

```
    }
} catch (CosNaming::NamingContext::NotFound &) {
    cerr << "Caught corba not found" << endl;
} catch (CosNaming::NamingContext::InvalidName &) {
    cerr << "Caught corba invalid name" << endl;
} catch (CosNaming::NamingContext::CannotProceed &) {
    cerr << "Caught corba cannot proceed" << endl;
}

//-----
-----
// Do stuff
//-----
-----

if (!CORBA::is_nil(service_server)) {
C++ (omniORB) client);
    char * server = service_server->send_message("Message from
    cout << "response from Server: " << server << endl;
    CORBA::string_free(server);
}

//-----
-----

// Destroy OBR
//-----
-----

orb->destroy();

} catch (CORBA::UNKNOWN) {
    cerr << "Caught CORBA exception: unknown exception" << endl;
}
}
```

## Build

Generate files from the interface description *example.idl*

```
omniidl -bcxx example.idl
```

*omniidl* will create files

- *example.hh*
- *exampleSK.cc*

The easiest way to build and compile all files is to use the *cpp/Makefile*[\[1\]](#)

**Maefile:**

# Makefile

```
OMNIORB_HOME=/usr
IDL=$(OMNIORB_HOME)/bin/omniidl
IDLFLAGS=-bcxx
INCLUDES=-I$(OMNIORB_HOME)/include -I$(OMNIORB_HOME)/include/
omniORB4
LIBS=-L$(OMNIORB_HOME)/lib -lomnithread -lomniORB4
```

```
.PHONY: all
all: server client
```

```
server: server.o MyExampleInterface_impl.o exampleSK.o
    $(CXX) -o $@ $^ $(LIBS)
```

```
server.o: server.cpp MyExampleInterface_impl.h
```

```
MyExampleInterface_impl.h: example.hh
client: client.o exampleSK.o
    $(CXX) -o $@ $^ $(LIBS)
```

```
MyExampleInterface_impl.o: MyExampleInterface_impl.cpp
MyExampleInterface_impl.h example.hh
```

```
exampleSK.o: exampleSK.cc example.hh
```

```
exampleSK.cc example.hh: $(IDL_FILE)
    $(IDL) $(IDLFLAGS) $<
```

```
.PHONY: clean
clean:
```

```
    find . -maxdepth 1 -type f -name "*.bak" -exec rm -f {} \;
    find . -maxdepth 1 -type f -name "*.o" -exec rm -f {} \;
    find . -maxdepth 1 -type f -name "*.stackdump" -exec rm -f {} \;
```

```
find . -maxdepth 1 -type f -name "*.exe" -exec rm -f {} \;  
find . -maxdepth 1 -type f -name "example.hh" -exec rm -f {} \;  
find . -maxdepth 1 -type f -name "exampleSK.cc" -exec rm -f {} \;
```

```
.cpp.o:  
$(CXX) $(CXXFLAGS) -o $@ -c $< $(INCLUDES)  
.cc.o:  
$(CXX) $(CXXFLAGS) -o $@ -c $< $(INCLUDES)
```

### **Configuration:**

#### **As a superuser**

1. include these lines in the file *etc/omniORB.cfg*

```
InitRef = NameService=corbaname::127.0.0.1:2809  
InitRef = EventService=corbaloc::localhost:4433/omniEvents  
supportBootstrapAgent = 1
```

2. service omniNames start

#### **Run your Program:**

```
./server -ORBInitRef EventService=corbaloc::localhost:4433/omniEvents
```

```
./client -ORBInitRef EventService=corbaloc::localhost:4433/omniEvents
```

### **For JAVA Implementation:**

#### **Files**

- *server.cpp* server programm
- *client.cpp* client programm
- *MyExampleInterface\_impl.java* class of the implementation for distributed objects

#### **Interface Implementation**

*MyExampleInterface\_impl.java*

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```
public class MyExampleInterface_impl extends ExampleInterfacePOA {  
    @Override  
    public String send_message(String message) {  
        System.out.println("Java Server: " + message);  
        return "Message from Java server";  
    }  
}
```

### Server

*server.java*

```
import java.util.*;  
  
import org.omg.CORBA.ORB;  
import org.omg.CosNaming.NameComponent;  
import org.omg.CosNaming.NamingContextExt;  
import org.omg.CosNaming.NamingContextExtHelper;  
import org.omg.PortableServer.POA;  
import org.omg.PortableServer.POAHelper;  
//import org.omg.PortableServer.POAManager;  
  
public class server {  
    /** Server name, clients needs to know this name */  
    public static final String SERVER_NAME = "MyServerName";  
  
    public static void main(String [] args) {  
        try {  
            // Create and initialize the CORBA ORB  
            ORB orb = ORB.init(args, null);  
  
            // Get reference to root POA and activate the POA Manager  
            POA rootpoa =  
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));  
            rootpoa.the_POAManager().activate();  
  
            // create servant and register it with the ORB  
            MyExampleInterface_impl service = new  
MyExampleInterface_impl();  
  
            // Get object reference from the servant  
            org.omg.CORBA.Object ref =  
rootpoa.servant_to_reference(service);
```

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```
ExampleInterface href = ExampleInterfaceHelper.narrow(ref);

// Get the root naming context
org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
// Use NamingContextExt which is part of the Interoperable
// Naming Service (INS) specification.
NamingContextExt ncRef =
NamingContextExtHelper.narrow(objRef);

// Bind the Object Reference in naming service
NameComponent path[] = ncRef.to_name( SERVER_NAME );
ncRef.rebind(path, href);

System.out.println("Java server '" + SERVER_NAME + "' is
running ...");

// Wait for remote invocations from clients
orb.run();

// destroy
orb.destroy();

} catch (org.omg.CORBA.UNKNOWN exception) {
exception.printStackTrace(System.out);
} catch (org.omg.CORBA.SystemException exception) {
exception.printStackTrace(System.out);
} catch (Exception exception) {
exception.printStackTrace(System.out);
}
}
}
```

## **Client**

*client.java*

```
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;

public class client {

    /** Name is defined in the server.cpp */
    public static final String SERVER_NAME = "MyServerName";

    public static void main(String [] args) {
```



**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```
try {
    // Create and initialize the CORBA ORB
    ORB orb = ORB.init(args, null);

    // Get the root naming context
    org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
    // Use NamingContextExt instead of NamingContext. This is
    // Part of the Interoperable naming Service.
    NamingContextExt ncRef =
NamingContextExtHelper.narrow(objRef);

    // Resolve the Object Reference in Naming
    ExampleInterface service_server =
ExampleInterfaceHelper.narrow(ncRef.resolve_str(SERVER_NAME));

    // Use service
    System.out.println("Java client is running ...");
    String server = service_server.send_message("Message from
Java client");
    System.out.println("> Response from Server: " + server);

    // Destroy
    orb.destroy();
} catch (org.omg.CORBA.ORBPackage.InvalidName exception) {
    exception.printStackTrace(System.out);
} catch (org.omg.CosNaming.NamingContextPackage.NotFound exception)
{
    exception.printStackTrace(System.out);
} catch (org.omg.CosNaming.NamingContextPackage.CannotProceed
exception) {
    exception.printStackTrace(System.out);
} catch (org.omg.CosNaming.NamingContextPackage.InvalidName
exception) {
    exception.printStackTrace(System.out);
} catch (org.omg.CORBA.COMM_FAILURE exception) {
    exception.printStackTrace(System.out);
} catch (Exception exception) {
    exception.printStackTrace(System.out);
}
}
```

## Build

Use the IDL compiler for JAVA *idlj* Generate files from the interface description *example.idl*

```
idlj -fall example.idl
```

*idlj* creates the files

- *ExampleInterface.java*
- *ExampleInterfaceHelper.java*
- *ExampleInterfaceHolder.java*
- *ExampleInterfaceOperations.java*
- *ExampleInterfacePOA.java*
- *\_ExampleInterfaceStub.java*

Just use the *java/Makefile*[\[1\]](#) to compile the code:

```
# Makefile
```

```
IDL=$(JAVA_HOME)/bin/idlj  
IDLFLAGS=-fall
```

```
JAVA=java  
JAVAC=javac
```

```
IDL_FILE = ../example.idl
```

```
.PHONY: all  
all: server.class client.class
```

```
server.class: \  
    server.java \  
    MyExampleInterface_impl.java \  
    ExampleInterfacePOA.java \  
    ExampleInterfaceOperations.java \  
    ExampleInterfaceHelper.java \  
    ExampleInterface.java \  
    _ExampleInterfaceStub.java  
    $(JAVAC) $^
```

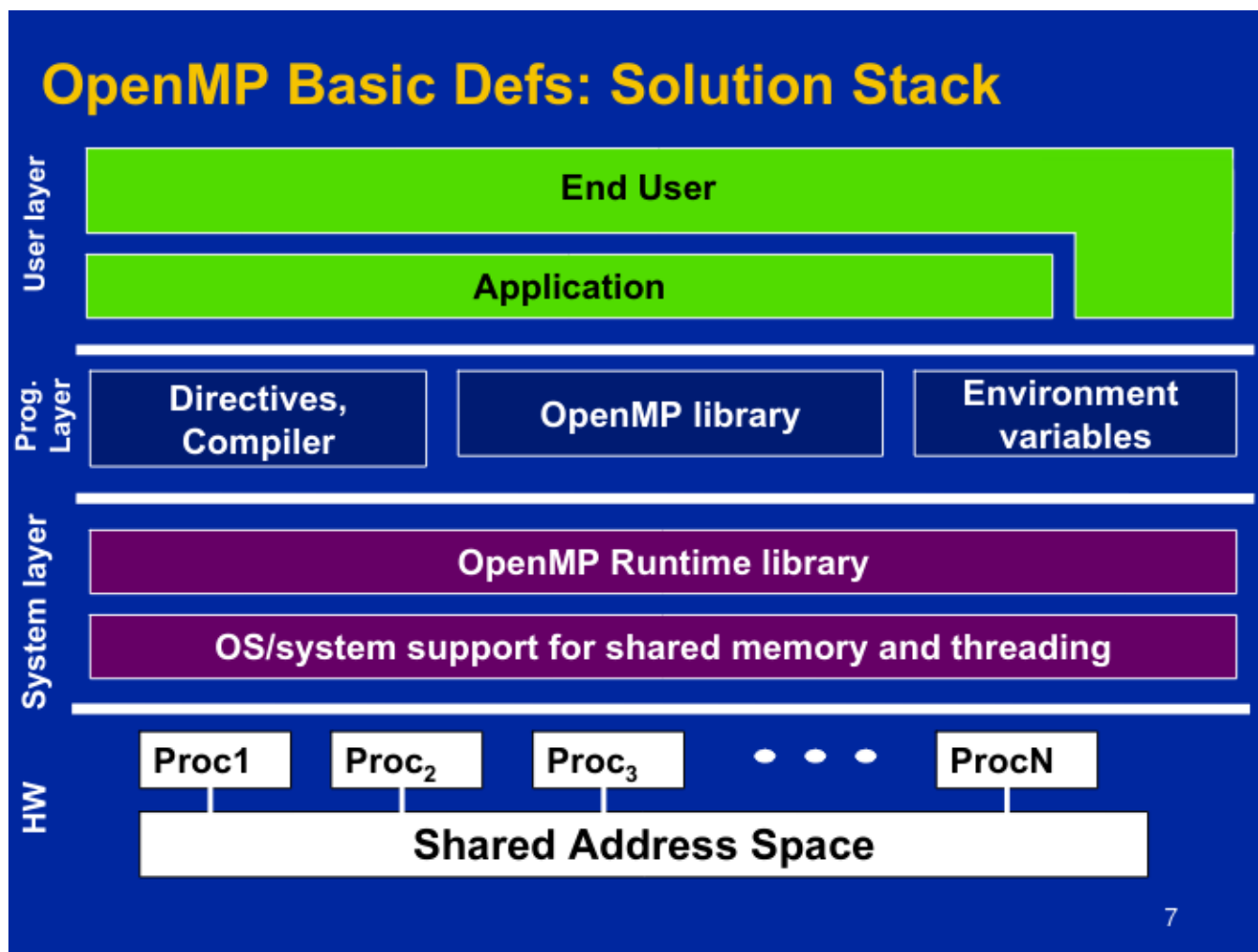
```
client.class: \  
    client.java \  
    ExampleInterface.java \  
    ExampleInterfaceOperations.java \  
    _ExampleInterfaceStub.java
```



**Experiment No:9**

**AIM:** Experiment with the application programming interface OpenMP which supports multi-platform shared-memory and multiprocessing programming in C.

OpenMP: An API for Writing Multithreaded Applications. A set of compiler directives and library routines for parallel application programmers. Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++.



**Ex. – Your program takes 20 days to run**

**95% can be parallelized**

**5% cannot (serial)**

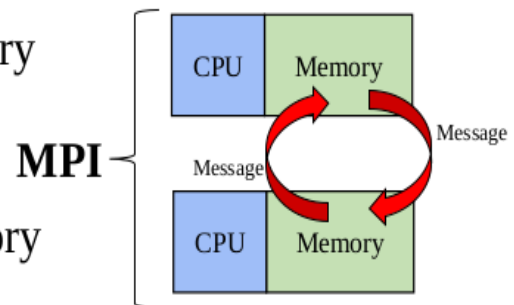
**What is the fastest this code can run?**

- **As many CPU's as you want!**
- **As you consider parallel programming understanding the underlying architecture is important**
- **Performance is affected by hardware configuration**
  - **Memory or CPU architecture**
  - **Numbers of cores/processor**
  - **Network speed and architecture**

## MPI and OpenMP

- **MPI – Designed for distributed memory**

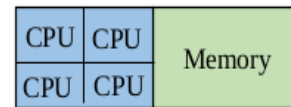
- Multiple systems
- Send/receive messages



- **OpenMP – Designed for shared memory**

- Single system with multiple cores
- One thread/core sharing memory

**OpenMP**



- **C, C++, and Fortran**

- **There are other options**

- **Interpreted languages with multithreading**
  - Python, R, matlab (have OpenMP & MPI underneath)
- **CUDA, OpenACC (GPUs)**
- **Pthreads, Intel Cilk Plus (multithreading)**
- **OpenCL, Chapel, Co-array Fortran, Unified Parallel C (UPC)**

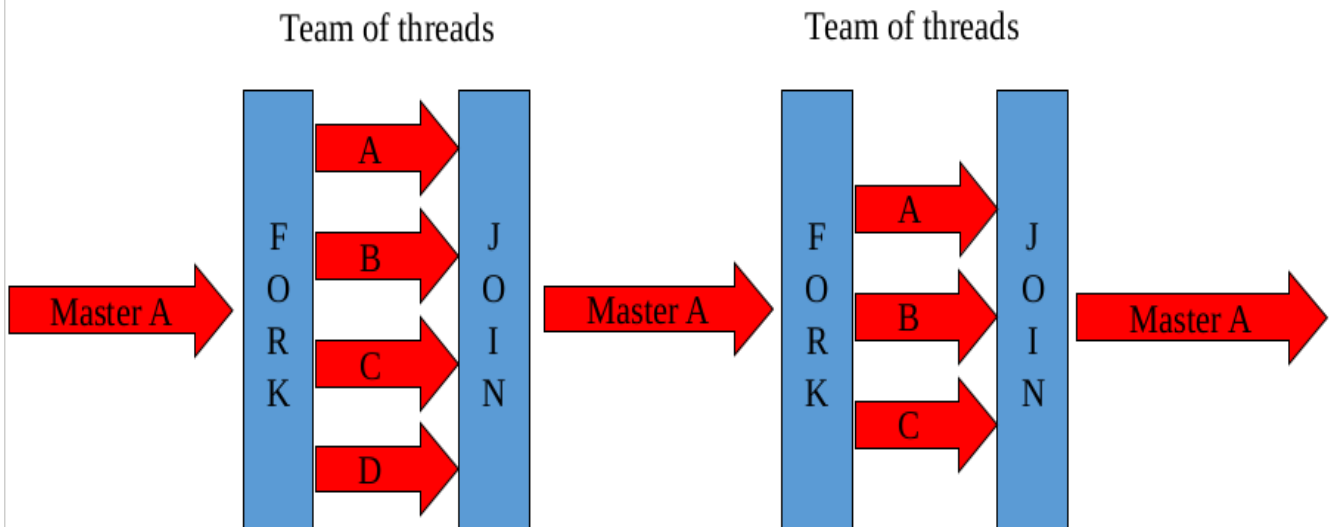
## OPENMP

- What is it?
  - Open Multi-Processing
  - Completely independent from MPI
  - Multi-threaded parallelism
- Standard since 1997
  - Defined and endorsed by the major players
- Fortran, C, C++
- Requires compiler to support OpenMP
  - Nearly all do
- For shared memory machines
  - Limited by available memory
  - Some compilers support GPUs.
- Preprocessor Directives
  - Preprocessor directives tell the compiler what to do
    - Always start with #
    - You've already seen one:  
`#include <stdio.h>`
  - OpenMP directives tell the compiler to add machine code for parallel execution of the following block  
`#pragma omp parallel`
    - "Run this next set of instructions in parallel"
- Some OpenMP Subroutines
  - `int omp_get_max_threads()`
    - Returns max possible (generally set by OMP\_NUM\_THREADS)
  - `int omp_get_num_threads()`
    - Returns number of threads in current team\\
  - `int omp_get_thread_num()`
    - Returns thread id of calling thread
    - Between 0 and `omp_get_num_threads-1`
- Process vs. Thread
  - MPI = Process, OpenMP = Thread
  - Program starts with a single process
  - Processes have their own (private) memory space

- A process can create one or more threads
- Threads created by a process share its memory space
  - Read and write to same memory addresses
  - Share same process ids and file descriptors
- Each thread has a unique instruction counter and stack pointer
  - A thread can have private storage on the stack

## OpenMP Fork-Join Model

- Automatically distributes work
- Fork-Join Model



# OpenMP Hello World

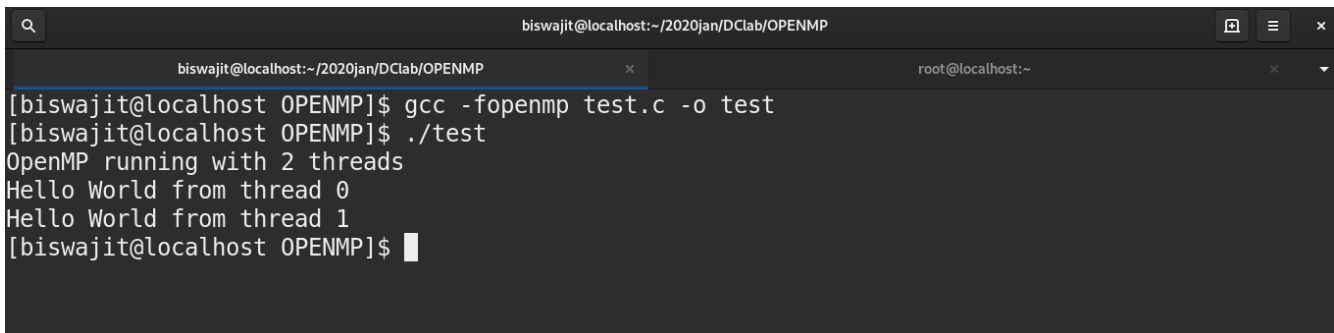
```
#include <omp.h> //<-- necessary header file for OpenMP API
#include <stdio.h>

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

    printf("OpenMP running with %d threads\n", omp_get_max_threads());

#pragma omp parallel
    {
        //Code here will be executed by all threads
        printf("Hello World from thread %d\n", omp_get_thread_num());
    }

    return 0;
}
```



The screenshot shows a terminal window with the following output:

```
biswajit@localhost:~/2020jan/DClab/OPENMP
biswajit@localhost:~/2020jan/DClab/OPENMP$ gcc -fopenmp test.c -o test
biswajit@localhost OPENMP$ ./test
OpenMP running with 2 threads
Hello World from thread 0
Hello World from thread 1
biswajit@localhost OPENMP$
```

- OMP\_NUM\_THREADS defines run time number of threads can be set in code as well using: `omp_set_num_threads()`
- OpenMP may try to use all available cpus if not set (On cluster–Always set it!)



**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

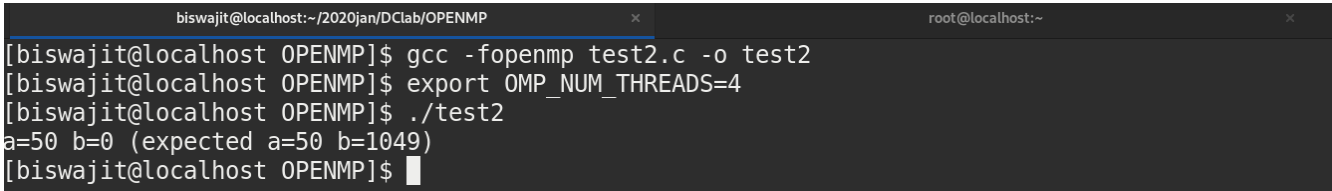
```
[biswajit@localhost OPENMP]$ export OMP_NUM_THREADS=4
[biswajit@localhost OPENMP]$ ./test
OpenMP running with 4 threads
Hello World from thread 1
Hello World from thread 0
Hello World from thread 3
Hello World from thread 2
[biswajit@localhost OPENMP]$
```

```
Private Variables 1:#include <omp.h>
#include <stdio.h>
int main()
{
    int i;
    const int N = 1000;
    int a = 50;
    int b = 0;
    #pragma omp parallel for default(shared)
    for (i=0; i<N; i++)
    {
        b = a + i;
    }
    printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

```
[biswajit@localhost OPENMP]$ gcc -fopenmp test1.c -o test1
[biswajit@localhost OPENMP]$ export OMP_NUM_THREADS=1
[biswajit@localhost OPENMP]$ ./test1
a=50 b=1049 (expected a=50 b=1049)
[biswajit@localhost OPENMP]$ export OMP_NUM_THREADS=4
[biswajit@localhost OPENMP]$ ./test1
a=50 b=549 (expected a=50 b=1049)
[biswajit@localhost OPENMP]$ █
```

### Private Variables 2:

```
#include <omp.h>
#include <stdio.h>
int main()
{
    int i;
    int a = 50;
    int b = 0;
#pragma omp parallel for default(none) private(i) private(a) private(b)
    for (i=0; i<1000; i++)
    {
        b = a + i;
    }
    printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```



```
biswajit@localhost:~/2020jan/DClab/OPENMP x root@localhost:~ x
[biswajit@localhost OPENMP]$ gcc -fopenmp test2.c -o test2
[biswajit@localhost OPENMP]$ export OMP_NUM_THREADS=4
[biswajit@localhost OPENMP]$ ./test2
a=50 b=0 (expected a=50 b=1049)
[biswajit@localhost OPENMP]$
```

```
Private Variables 3:#include <omp.h>
#include <stdio.h>
int main()
{
    int i;
    int a = 50;
    int b = 0;
#pragma omp parallel for default(none) private(i) private(a) lastprivate(b)
    for (i=0; i<1000; i++)
    {
        b = a + i;
    }
}
```

```
printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
```

```
}
```

```
[biswajit@localhost OPENMP]$ gcc -fopenmp test3.c -o test3
[biswajit@localhost OPENMP]$ export OMP_NUM_THREADS=4
[biswajit@localhost OPENMP]$ ./test3
a=50 b=1002 (expected a=50 b=1049)
[biswajit@localhost OPENMP]$ █
```

#### Private Variables 4:

```
#include <omp.h>
```

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

```
int main()
```

```
{
```

```
    int i;
```

```
    int a = 50;
```

```
    int b = 0;
```

```
#pragma omp parallel for default(none) private(i) firstprivate(a) lastprivate(b)
```

```
    for (i=0; i<1000; i++)
```

```
    {
```

```
        b = a + i;
```

```
    }
```

```
    printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
```

```
}
```

```
[biswajit@localhost OPENMP]$ gcc -fopenmp test4.c -o test4
[biswajit@localhost OPENMP]$ export OMP_NUM_THREADS=4
[biswajit@localhost OPENMP]$ ./test4
a=50 b=1049 (expected a=50 b=1049)
[biswajit@localhost OPENMP]$ █
```

#### OpenMP Constructs:

- Parallel region
    - Thread creates team, and becomes master (id 0)
    - All threads run code after
    - Barrier at end of parallel section
- ```
#pragma omp parallel [clause ...]
    if (scalar_expression)
```

private (list)  
shared (list)  
default (shared | none)  
firstprivate (list)  
lastprivate (list)  
reduction (operator: list)  
num\_threads (integer)

#### **OMP Parallel Clauses 1:**

#pragma omp parallel if (scalar\_expression)

- Only execute in parallel if true
- Otherwise serial

#pragma omp parallel private (list)

- Data local to thread
- Values are not guaranteed to be defined on exit (even if defined before)
- No storage associated with original object
  - Use firstprivate and/or lastprivate clause to override

#### **OMP Parallel Clauses 2:**

#pragma omp parallel firstprivate (list)

- Variables in list are private
- Initialized with the value the variable had before entering the construct

#pragma omp parallel for lastprivate (list)

- Only in for loops
- Variables in list are private
- The thread that executes the sequentially last iteration updates the value of the variables in the list.

#### **OMP Parallel Clause 3:**

#pragma omp shared (list)

- Data is accessible by all threads in team
- All threads access same address space
- Improperly scoped variables are big source of OMP bugs
  - Shared when should be private

- Race condition

`#pragma omp default (shared | none)`

- Tip: Safest is to use default(none) and declare by hand

### **Shared and Private Variables:**

- Take home message:
  - Be careful with the scope of your variables
  - Results must be independent of thread count
  - Test & debug thoroughly!
- Important note about compilers
  - C (before C99) does not allow variables declared in for loop syntax
    - Compiler will make loop variables private
    - Still recommend explicit

```
#pragma omp parallel private(i)
```

```
for (i=0; i<1000; i++)
```

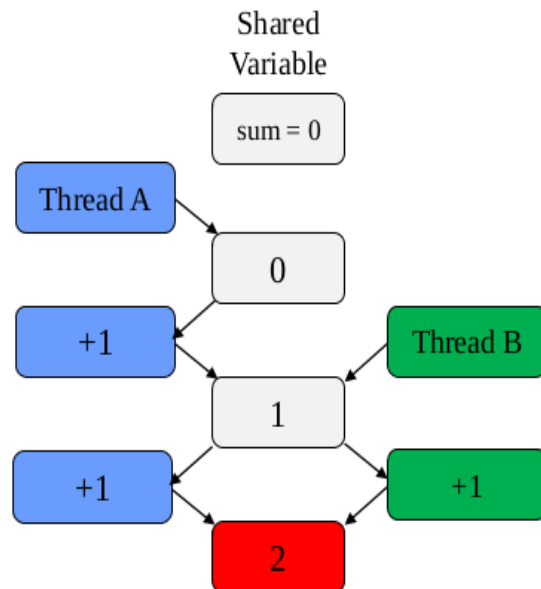
```
{
```

```
    b = a + i;
```

```
}
```

## Caution: Race Condition

- When multiple threads simultaneously read/write shared variable
- Multiple OMP solutions
  - Reduction
  - Atomic
  - Critical



Should be 3!

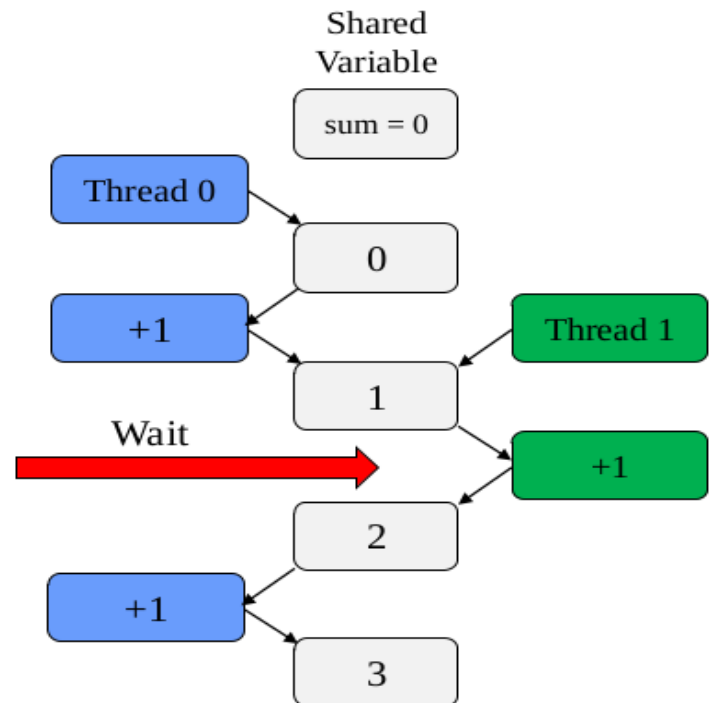
```
#pragma omp parallel for private(i) shared(sum)
for (i=0; i<N; i++) {
    sum += i;
}
```

# Critical Section

- One solution: use critical
- Only one thread at a time can execute a critical section

```
#pragma omp critical  
{  
    sum += i;  
}
```

- Downside?
  - SLOOOOOWWW
  - Overhead & serialization

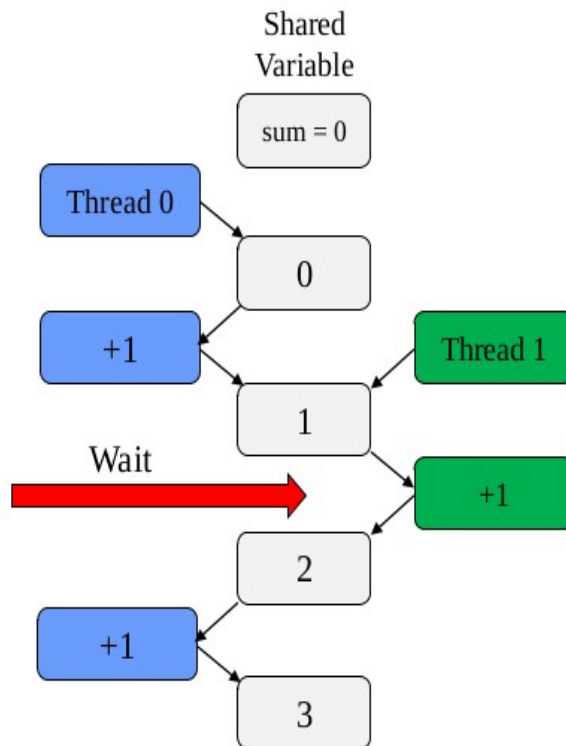


# OMP Atomic

- Atomic like “mini” critical
- Only one line
  - Certain limitations

```
#pragma omp atomic  
sum += i;
```

- Hardware controlled
  - Less overhead than critical



## OMP Reduction:

```
#pragma omp reduction (operator:variable)
```

- Avoids race condition
- Reduction variable must be shared
- Makes variable private, then performs operator at end of loop
- Operator cannot be overloaded (c++)
  - One of: +, \*, -, / (and &, ^, |, &&, ||)
  - OpenMP 3.1: added min and max for c/c++



**Reduction Example:**

```
#include <omp.h>
#include <stdio.h>
int main()
{
    int i;
    const int N = 1000;
    int sum = 0;
#pragma omp parallel for private(i) reduction(+: sum)
    for (i=0; i<N; i++)
    {
        sum += i;
    }
    printf("reduction sum=%d (expected %d)\n", sum, ((N-1)*N)/2);
}
```

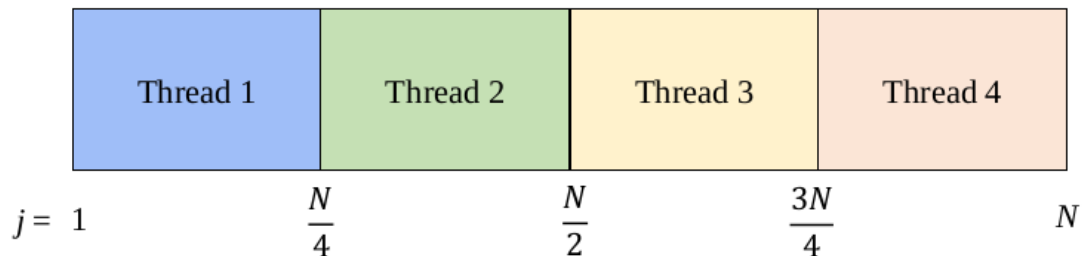
```
[biswajit@localhost OPENMP]$ gcc -fopenmp test5.c -o test5
[biswajit@localhost OPENMP]$ export OMP_NUM_THREADS=4
[biswajit@localhost OPENMP]$ ./test5
reduction sum=499500 (expected 499500)
[biswajit@localhost OPENMP]$
```

## Scheduling omp for

- How does a loop get split up?
  - In MPI, we have to do it manually
- If you don't tell it what to do, the compiler decides
- Usually compiler chooses "static" – chunks of  $N/p$

```
#pragma omp parallel for default(shared) private(j)
for (j=0; j<N; j++) {
    ... // some work here
}
```

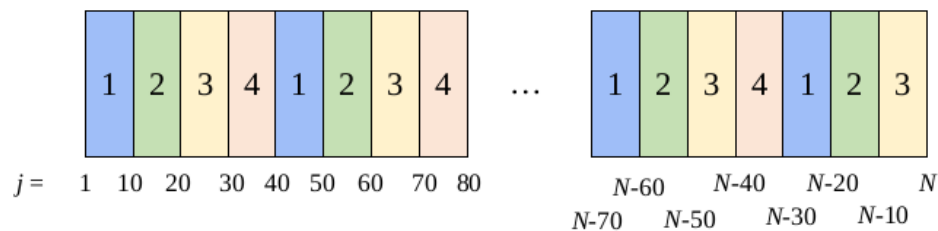
Unspecified schedule



# Static Scheduling

- You can tell the compiler what size chunks to take

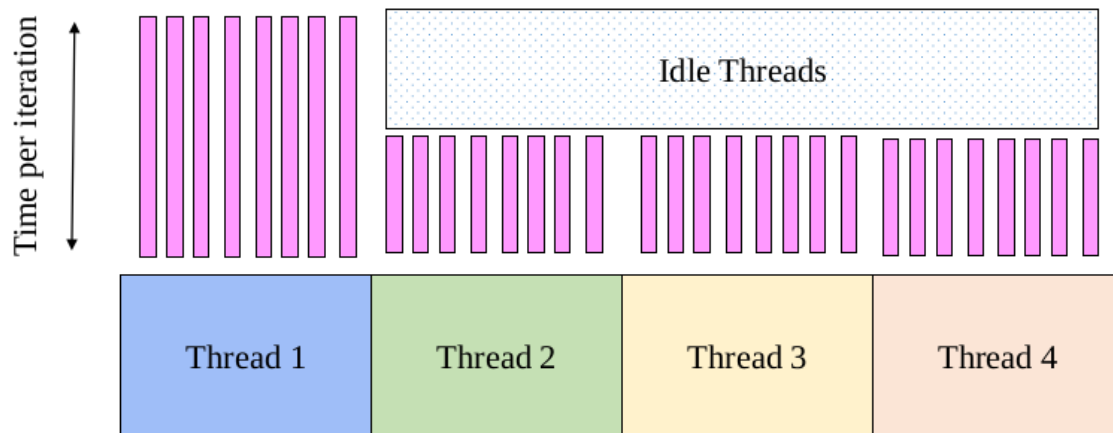
```
#pragma omp parallel for default(shared) private(j) schedule(static,10)  
for (j=0; j<N; j++) {  
    ... // some work here  
}
```



- Keeps assigning chunks until done
- Chunk size that isn't a multiple of the loop will result in threads with uneven numbers

## Problem with Static Scheduling

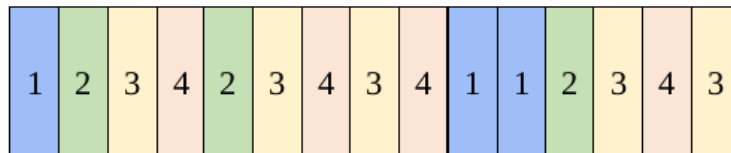
- What happens if loop iterations do not take the same amount of time?
  - Load imbalance



## Dynamic Scheduling

- Chunks are assigned on the fly, as threads become available
  - When a thread finishes one chunk, it is assigned another

```
#pragma omp parallel for default(shared) private(j) schedule(dynamic, 10)  
  for (j=0; j<N; j++) {  
    ... // some work here  
  }
```



- Caveat Emptor: higher overhead than static!

## omp for Scheduling Recap

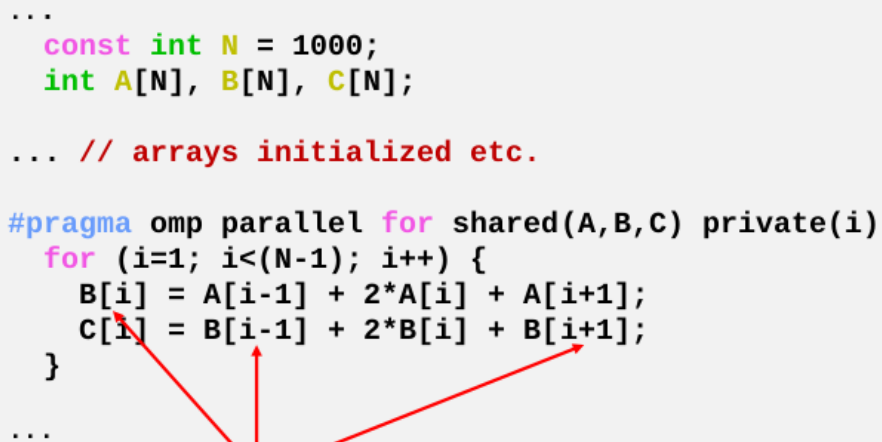
```
#pragma omp parallel for schedule(type [,size])
```

- Scheduling types
  - Static
    - Chunks of specified size assigned round-robin
  - Dynamic
    - Chunks of specified size are assigned when thread finishes previous chunk
  - Guided
    - Like dynamic, but chunks are exponentially decreasing
    - Chunk will not be smaller than specified size
  - Runtime
    - Type and chunk determined at runtime via environment variables

## Where not to use OpenMP

What could go wrong here?

```
...  
const int N = 1000;  
int A[N], B[N], C[N];  
  
... // arrays initialized etc.  
  
#pragma omp parallel for shared(A,B,C) private(i)  
for (i=1; i<(N-1); i++) {  
    B[i] = A[i-1] + 2*A[i] + A[i+1];  
    C[i] = B[i-1] + 2*B[i] + B[i+1];  
}  
...
```



B[i-1] and B[i+1] are not  
guaranteed to be available/correct

SOME SIMPLE EXAMPLES:  
`#include<stdio.h>`

`#include<omp.h>`

`void main()`

`{`

`int ID = 0;`

`printf(" hello(%d) ", ID);`

`printf(" world(%d) \n", ID);`

`}`

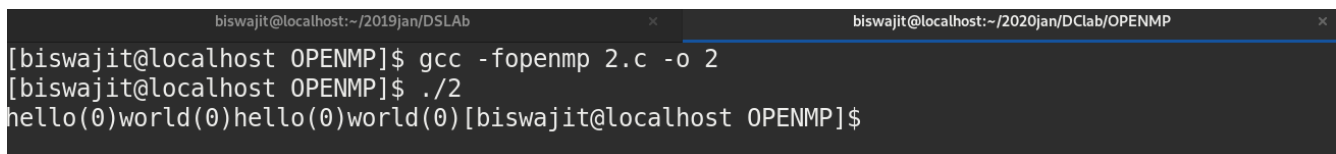
`[biswajit@localhost OPENMP]$ gcc -fopenmp 1.c -o 1`

`[biswajit@localhost OPENMP]$ ./1`

`hello(0) world(0)`

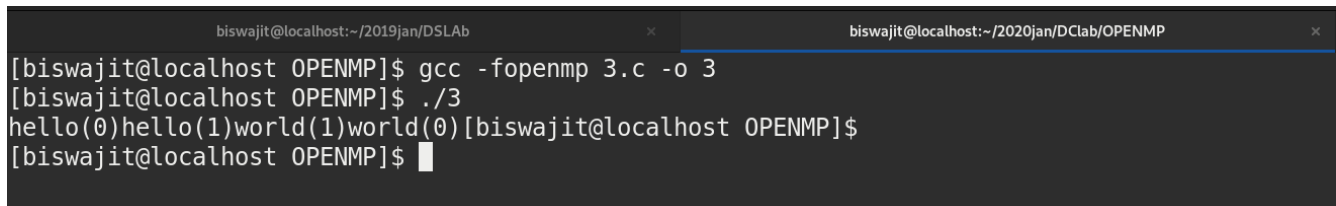
`[biswajit@localhost OPENMP]$`

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int id=0;
    #pragma omp parallel
    {
        printf("hello(%d)",id);
        printf("world(%d)",id);
    }
}
```



```
biswajit@localhost:~/2019jan/DSLAb x biswajit@localhost:~/2020jan/DClab/OPENMP x
[biswajit@localhost OPENMP]$ gcc -fopenmp 2.c -o 2
[biswajit@localhost OPENMP]$ ./2
hello(0)world(0)hello(0)world(0)[biswajit@localhost OPENMP]$
```

```
#include<stdio.h>
#include<omp.h>
int main()
{
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("hello(%d)",id);
        printf("world(%d)",id);
    }
}
```



```
biswajit@localhost:~/2019jan/DSLAb x biswajit@localhost:~/2020jan/DClab/OPENMP x
[biswajit@localhost OPENMP]$ gcc -fopenmp 3.c -o 3
[biswajit@localhost OPENMP]$ ./3
hello(0)hello(1)world(1)world(0)[biswajit@localhost OPENMP]$
[biswajit@localhost OPENMP]$
```



```
#include<stdio.h>
#include<omp.h>
void pooh(int, double[]);
int main()
{
    double A[1000];
    omp_set_num_threads(4);//Runtime function to request a certain number of threads
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();//Runtime function returning a thread ID
        pooh(ID,A);
    }
}
void pooh(int x, double a[])
{
    printf("\n %d",x);
}
}
```

```
[biswajit@localhost OPENMP]$ gcc -fopenmp 4.c -o 4
[biswajit@localhost OPENMP]$ ./4
2
0
3
1[biswajit@localhost OPENMP]$
```

```
#include<stdio.h>
#include<omp.h>
void pooh(int, double[]);
int main()
{
    double A[1000];
    #pragma omp parallel num_threads(4)//clause to request a certain number of threads
    {
        int ID = omp_get_thread_num();//Runtime function returning a thread ID
        pooh(ID,A);
    }
}
```

```
    }  
}  
void pooh(int x, double a[])  
{  
    printf("\n %d",x);  
}
```

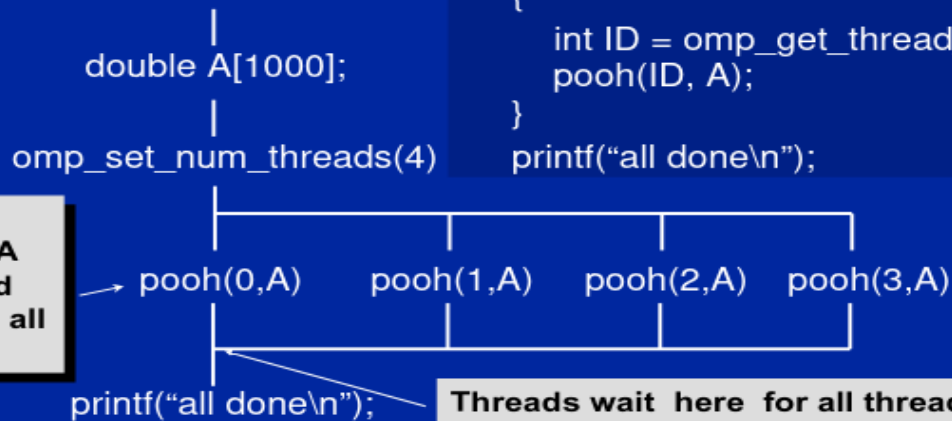
```
[biswajit@localhost OPENMP]$ gcc -fopenmp 5.c -o 5  
[biswajit@localhost OPENMP]$ ./5  
3  
0  
2  
1[biswajit@localhost OPENMP]$
```

## Thread Creation: Parallel Regions example

- Each thread executes the same code redundantly.

```
double A[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int ID = omp_get_thread_num();  
    pooh(ID, A);  
}  
printf("all done\n");
```

A single copy of A is shared between all threads.



Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

```
#include<stdio.h>
#include<omp.h>
int big_callc1(int);
void big_callc2(int[],int);
int main()
{
    int id,A[10],i;
    #pragma omp parallel shared (A) private(id)
    {
        id=omp_get_thread_num();
        A[id]=big_callc1(id);
        #pragma omp for
        for(i=0;i<4;i++)//implicit barrier at the end of a for worksharing construct
        {
            big_callc2(A,i);
        }
    }
}

int big_callc1(int x)
{
    printf("\n in call1 %d",x);
    return x;
}

void big_callc2(int C[],int i)
{
    printf("\n in call2 %d",C[i]);
}
```

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```
biswajit@localhost:~/2019jan/DSLAb x biswajit@localhost:~/2020jan/DClab/OPENMP x
[biswajit@localhost OPENMP]$ gcc -fopenmp 6.c -o 6
[biswajit@localhost OPENMP]$ ./6

in call1 1
in call2 1
in call1 2
in call2 2
in call1 0
in call2 0
in call1 3
in call2 3[biswajit@localhost OPENMP]$
```

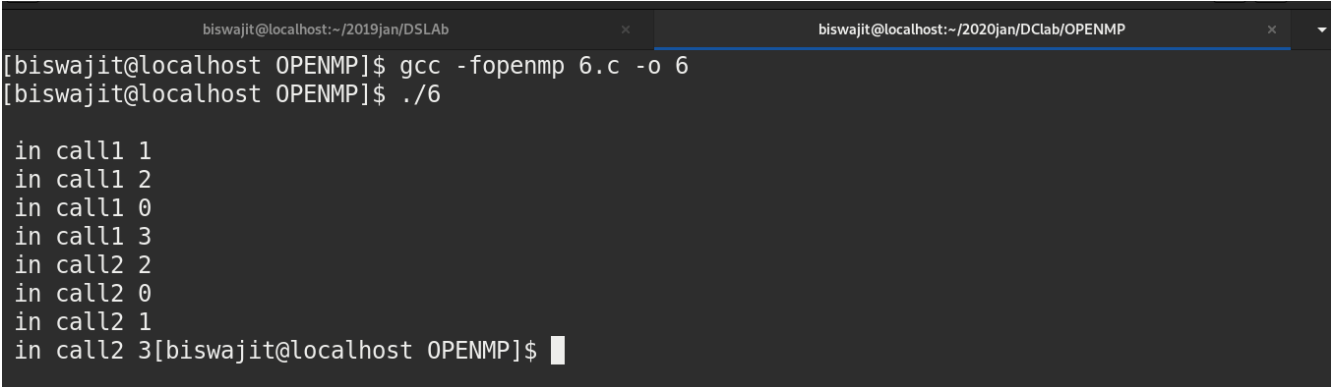
Call1 and call2 both are mixing to synchronization we use #pragma omp barrier

```
#include<stdio.h>
#include<omp.h>
int big_callc1(int);
void big_callc2(int[],int);
int main()
{
    int id,A[10],i;
    #pragma omp parallel shared (A) private(id)
    {
        id=omp_get_thread_num();
        A[id]=big_callc1(id);
        #pragma omp barrier
        #pragma omp for
        for(i=0;i<4;i++)//implicit barrier at the end of a for worksharing construct
        {
            big_callc2(A,i);
        }
    }
}

int big_callc1(int x)
{
    printf("\n in call1 %d",x);
```

```
    return x;
}

void big_callc2(int C[],int i)
{
    printf("\n in call2 %d",C[i]);
}
```



```
biswajit@localhost:~/2019jan/DSLAb
biswajit@localhost:~/2020jan/DClab/OPENMP
[biswajit@localhost OPENMP]$ gcc -fopenmp 6.c -o 6
[biswajit@localhost OPENMP]$ ./6

in call1 1
in call1 2
in call1 0
in call1 3
in call2 2
in call2 0
in call2 1
in call2 3[biswajit@localhost OPENMP]$
```

```
#pragma omp for nowait
#include<stdio.h>
#include<omp.h>
int big_callc1(int);
void big_callc2(int[],int);
void big_callc3(int[],int);
void big_callc4(int);
int main()
{
    int id,A[10],i;
    #pragma omp parallel shared (A) private(id)
    {
        id=omp_get_thread_num();
        A[id]=big_callc1(id);
        #pragma omp barrier
        #pragma omp for
        for(i=0;i<6;i++)//implicit barrier at the end of a for worksharing construct
```

```
    {
        big_callc2(A,i);
    }
    #pragma omp for nowait
    for(i=0;i<6;i++)//no implicit barrier due to nowait
    {
        big_callc3(A,i);
    }
    big_callc4(id);
}
}
```

```
int big_callc1(int x)
```

```
{
    printf("\n in call1 %d",x);
    return x;
}
```

```
void big_callc2(int C[],int i)
```

```
{
    printf("\n in call2 %d",C[i]);
}
```

```
void big_callc3(int C[],int i)
```

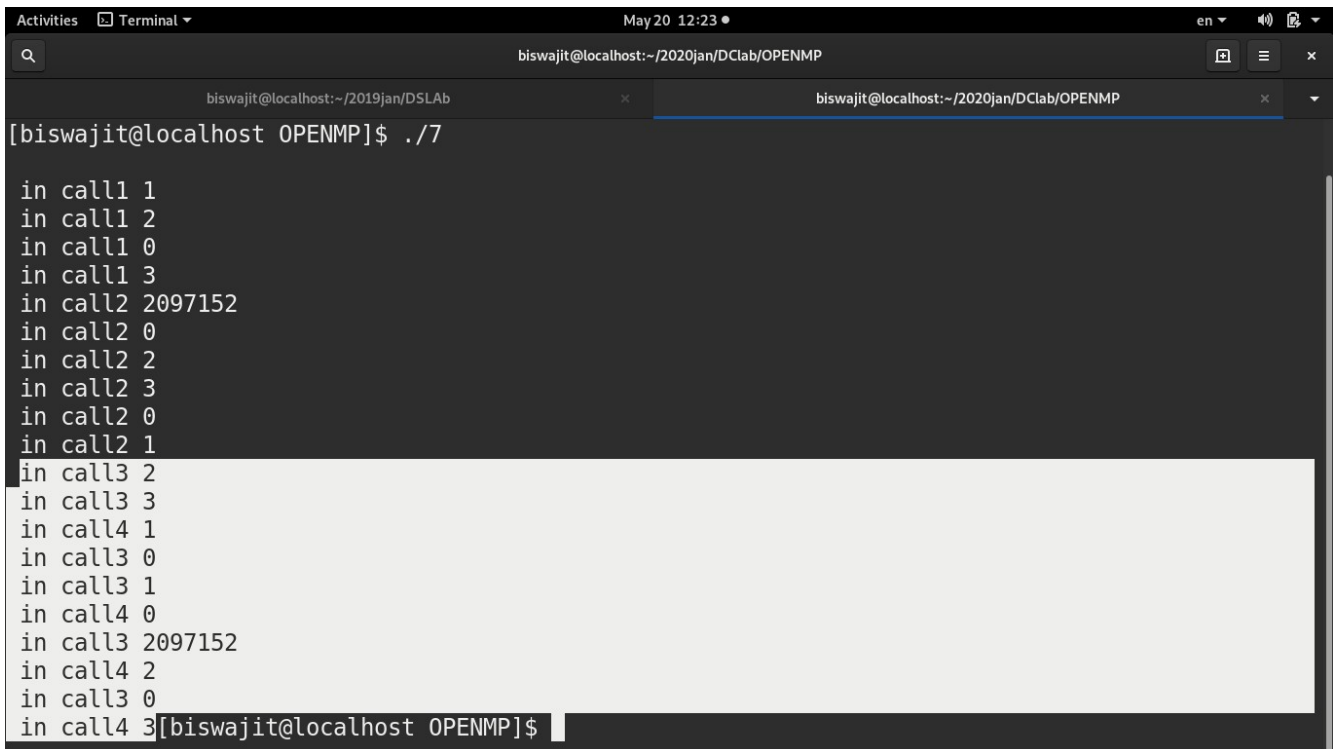
```
{
    printf("\n in call3 %d",C[i]);
}
```

```
void big_callc4(int x)
```

```
{
    printf("\n in call4 %d",x);
}
```

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---



```
Activities Terminal May 20 12:23 en
biswajit@localhost:~/2020jan/DClab/OPENMP
biswajit@localhost:~/2019jan/DSLAb
biswajit@localhost:~/2020jan/DClab/OPENMP
[biswajit@localhost OPENMP]$ ./7
in call1 1
in call1 2
in call1 0
in call1 3
in call2 2097152
in call2 0
in call2 2
in call2 3
in call2 0
in call2 1
in call3 2
in call3 3
in call4 1
in call3 0
in call3 1
in call4 0
in call3 2097152
in call4 2
in call3 0
in call4 3[biswajit@localhost OPENMP]$
```

Sequence is broken call3 and call4 both are executing .

**Master Construct:**

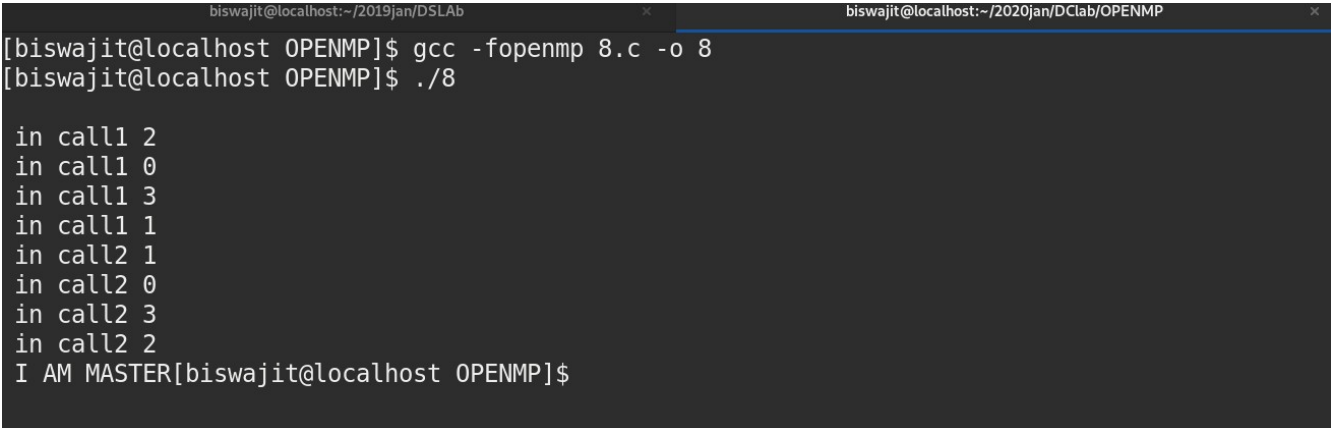
The masterconstruct denotes a structured block that is only executed by the master thread. The other threads just skip it (no synchronization is implied).

```
#include<stdio.h>
#include<omp.h>
int big_callc1(int);
void big_callc2(int[],int);
int main()
{
    int id,A[10],i;
    #pragma omp parallel shared (A) private(id)
    {
        id=omp_get_thread_num();
```

```
A[id]=big_callc1(id);
#pragma omp barrier
#pragma omp for
for(i=0;i<4;i++)//implicit barrier at the end of a for worksharing construct
{
    big_callc2(A,i);
}
#pragma omp master
{
    printf("\n I AM MASTER");
}
}
}

int big_callc1(int x)
{
    printf("\n in call1 %d",x);
    return x;
}

void big_callc2(int C[],int i)
{
    printf("\n in call2 %d",C[i]);
}
```



The image shows a terminal window with two tabs. The left tab is titled 'biswajit@localhost:~/2019jan/DSLAb' and the right tab is 'biswajit@localhost:~/2020jan/DCLab/OPENMP'. The terminal output shows the compilation of a C program named '8.c' using 'gcc -fopenmp 8.c -o 8' and its execution with './8'. The output of the program is as follows:

```
[biswajit@localhost OPENMP]$ gcc -fopenmp 8.c -o 8
[biswajit@localhost OPENMP]$ ./8

in call1 2
in call1 0
in call1 3
in call1 1
in call2 1
in call2 0
in call2 3
in call2 2
I AM MASTER[biswajit@localhost OPENMP]$
```



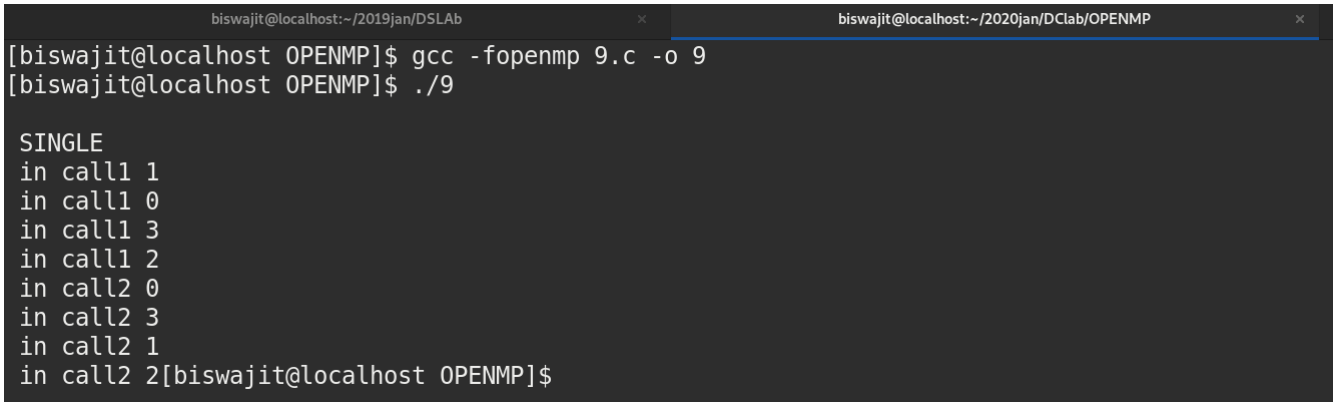
**Single worksharing Construct:**

The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread). A barrier is implied at the end of the single block (can remove the barrier with a nowait clause).

```
#include<stdio.h>
#include<omp.h>
int big_callc1(int);
void big_callc2(int[],int);
int main()
{
    int id,A[10],i;
    #pragma omp parallel shared (A) private(id)
    {
        #pragma omp single
        {
            printf("\n SINGLE");
        }
        id=omp_get_thread_num();
        A[id]=big_callc1(id);
        #pragma omp barrier
        #pragma omp for
        for(i=0;i<4;i++)//implicit barrier at the end of a for worksharing construct
        {
            big_callc2(A,i);
        }
    }
}

int big_callc1(int x)
{
    printf("\n in call1 %d",x);
    return x;
}
```

```
void big_callc2(int C[],int i)
{
    printf("\n in call2 %d",C[i]);
}
```



```
biswajit@localhost:~/2019jan/DSLAb
biswajit@localhost:~/2020jan/DClab/OPENMP
[biswajit@localhost OPENMP]$ gcc -fopenmp 9.c -o 9
[biswajit@localhost OPENMP]$ ./9

SINGLE
in call1 1
in call1 0
in call1 3
in call1 2
in call2 0
in call2 3
in call2 1
in call2 2[biswajit@localhost OPENMP]$
```

### Synchronization:

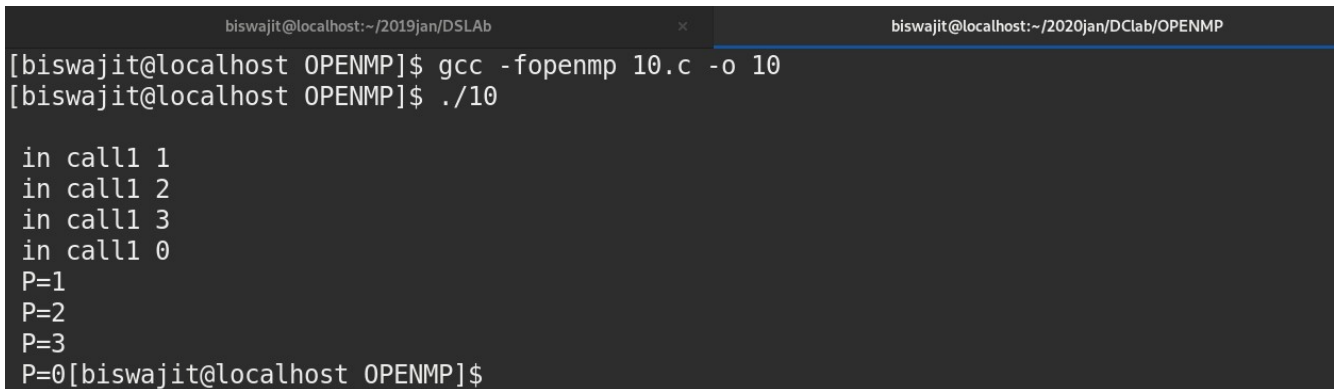
The ordered region executes in the sequential order.

```
#include<stdio.h>
#include<omp.h>
int big_callc1(int);
int big_callc2(int[],int);
int main()
{
    int id,A[10],i,p=0;
    #pragma omp parallel private(A,id)
    {
        id=omp_get_thread_num();
        A[id]=big_callc1(id);
        #pragma omp barrier
        #pragma omp for ordered reduction(+:p)
        for(i=0;i<4;i++)//implicit barrier at the end of a for worksharing construct
        {
            #pragma ordered
```

```
        {
            p+=big_callc2(A,id);
            printf("\n P=%d",p);
        }
    }
}
```

```
int big_callc1(int x)
{
    printf("\n in call1 %d",x);
    return x;
}
```

```
int big_callc2(int C[],int i)
{
    return C[i];
}
```



```
biswajit@localhost:~/2019jan/DSLAb x biswajit@localhost:~/2020jan/DClab/OPENMP
[biswajit@localhost OPENMP]$ gcc -fopenmp 10.c -o 10
[biswajit@localhost OPENMP]$ ./10

in call1 1
in call1 2
in call1 3
in call1 0
P=1
P=2
P=3
P=0[biswajit@localhost OPENMP]$
```

### Synchronization:

Lock routines

Simple Lock routines: A simple lock is available if it is unset.– `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`, `omp_destroy_lock()`

Nested Locks: A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function– `omp_init_nest_lock()`, `omp_set_nest_lock()`, `omp_unset_nest_lock()`, `omp_test_nest_lock()`, `omp_destroy_nest_lock()`

**Note:**

1. a thread always accesses the most recent copy of the lock, so you don't need to use a flush on the lock variable.
2. A lock implies a memory fence (a “flush”) of all thread visible variables.

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
    id = omp_get_thread_num();
    tmp = do_lots_of_work(id);
    omp_set_lock(&lck); //Wait here for your turn
    printf(“%d %d”, id, tmp);
    omp_unset_lock(&lck); //Wait here for your turn
}
omp_destroy_lock(&lck); //Free-up storage when done
```

### **Experiment No:10**

#### **AIM: Experiment with Message Passing Interface Standard (MPI).**

MPI is a library of routines that can be used to create parallel programs in C or Fortran77. Standard C and Fortran include no constructs supporting parallelism so vendors have developed a variety of extensions to allow users of those languages to build parallel applications. The result has been a spate of non-portable applications, and a need to retrain programmers for each platform upon which they work.

The MPI standard was developed to ameliorate these problems. It is a library that runs with standard C or Fortran programs, using commonly-available operating system services to create parallel processes and exchange information among these processes.

MPI is designed to allow users to create programs that can run efficiently on most parallel architectures. The design process included vendors (such as IBM, Intel, TMC, Cray, Convex, etc.), parallel library authors (involved in the development of PVM, Linda, etc.), and applications specialists. The final version for the draft standard became available in May of 1994.

MPI can also support distributed program execution on heterogenous hardware. That is, you may run a program that starts processes on multiple computer systems to work on the same problem. This is useful with a workstation farm.

Here is the basic Hello world program in C using MPI:

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

main(int argc, char **argv)
{
    int ierr;

    ierr = MPI_Init(&argc, &argv);
    printf("Hello world\n");

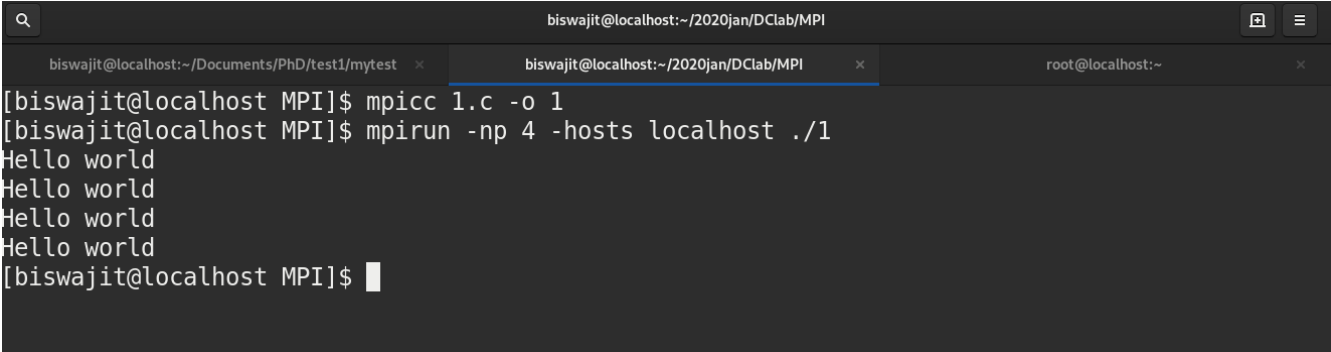
    ierr = MPI_Finalize();
}
```

If you compile hello.c with a command like

```
mpicc 1.c -o 1
```

you will create an executable file called `hello`, which you can execute by using the `mpirun` command as in the following session segment:

```
mpirun -np 4 -hosts localhost ./1
```



```
biswajit@localhost MPI$ mpicc 1.c -o 1
biswajit@localhost MPI$ mpirun -np 4 -hosts localhost ./1
Hello world
Hello world
Hello world
Hello world
biswajit@localhost MPI$
```

When the program starts, it consists of only one process, sometimes called the "parent", "root", or "master" process. When the routine `MPI_Init` executes within the root process, it causes the creation of 3 additional processes (to reach the number of processes (`np`) specified on the `mpirun` command line), sometimes called "child" processes.

Each of the processes then continues executing separate versions of the hello world program. The next statement in every program is the `printf` statement, and each process prints "Hello world" as directed. Since terminal output from every program will be directed to the same terminal, we see four lines saying "Hello world".

## Identifying the separate processes

As written, we cannot tell which "Hello world" line was printed by which process. To identify a process we need some sort of process ID and a routine that lets a process find its own process ID. MPI assigns an integer to each process beginning with 0 for the parent process and incrementing each time a new process is created. A process ID is also called its "rank".

MPI also provides routines that let the process determine its process ID, as well as the number of processes that are have been created.

Here is an enhanced version of the Hello world program that identifies the process that writes each line of output:

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv)
{
    int ierr, num_procs, my_id;
    ierr = MPI_Init(&argc, &argv);
    /* find out MY process ID, and how many processes were started. */
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    printf("Hello world! I'm process %i out of %i processes\n",my_id, num_procs);
    ierr = MPI_Finalize();
    return 0;
}
```

When we run this program, each process identifies itself:

```
$ mpicc 2.c -o 2
```

```
$ mpirun -np 4 -hosts localhost ./2
```

```
Hello world! I'm process 0 out of 4 processes.
```

```
Hello world! I'm process 2 out of 4 processes.
```

```
Hello world! I'm process 1 out of 4 processes.
```

```
Hello world! I'm process 3 out of 4 processes.
```

```
$
```

**Note that the process numbers are not printed in ascending order. That is because the processes execute independently and execution order was not controlled in any way. The programs may print their results in different orders each time they are run.**

(To find out which Origin processors and memories are used to run a program you can turn on the `MPI_DSM_VERBOSE` environment variable with "export `MPI_DSM_VERBOSE=ON`", or equivalent.)

To let each process perform a different task, you can use a program structure like:

```
#include <mpi.h>

int main(int argc, char **argv)
{
    int my_id, root_process, ierr, num_procs;
    MPI_Status status;

    /* Create child processes, each of which has its own variables.
```

```
* From this point on, every process executes a separate copy
* of this program. Each process has a different process ID,
* ranging from 0 to num_procs minus 1, and COPIES of all
* variables defined in the program. No variables are shared.
**/

ierr = MPI_Init(&argc, &argv);

/* find out MY process ID, and how many processes were started. */

ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

if( my_id == 0 ) {

    /* do some work as process 0 */
}
else if( my_id == 1 ) {

    /* do some work as process 1 */
}
else if( my_id == 2 ) {

    /* do some work as process 2 */
}
else {

    /* do this work in any remaining processes */
}
/* Stop this process */

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

## Basic MPI communication routines

It is important to realize that separate processes share no memory variables. They appear to be using the same variables, but they are really using COPIES of any variable defined in the program.

As a result, these programs cannot communicate with each other by exchanging information in memory variables. Instead they may use any of a large number of MPI communication routines. The two basic routines are:

- MPI\_Send, to send a message to another process, and



- `MPI_Recv`, to receive a message from another process.

The syntax of `MPI_Send` is:

```
int MPI_Send(void *data_to_send, int send_count, MPI_Datatype send_type,  
            int destination_ID, int tag, MPI_Comm comm);
```

- `data_to_send`: variable of a C type that corresponds to the `send_type` supplied below
- `send_count`: number of data elements to be sent (nonnegative int)
- `send_type`: datatype of the data to be sent (one of the MPI datatype handles)
- `destination_ID`: process ID of destination (int)
- `tag`: message tag (int)
- `comm`: communicator (handle)

Once a program calls `MPI_Send`, it blocks until the data transfer has taken place and the `data_to_send` variable can be safely reused. As a result, these routines provide a simple synchronization service along with data exchange.

The syntax of `MPI_Recv` is:

```
int MPI_Recv(void *received_data, int receive_count, MPI_Datatype receive_type,  
            int sender_ID, int tag, MPI_Comm comm, MPI_Status *status);
```

- `received_data`: variable of a C type that corresponds to the `receive_type` supplied below
- `receive_count`: number of data elements expected (int)
- `receive_type`: datatype of the data to be received (one of the MPI datatype handles)
- `sender_ID`: process ID of the sending process (int)
- `tag`: message tag (int)
- `comm`: communicator (handle)
- `status`: status struct (`MPI_Status`)

The `receive_count`, `sender_ID`, and `tag` values may be specified so as to allow messages of unknown length, from several sources (`MPI_ANY_SOURCE`), or with various tag values (`MPI_ANY_TAG`).

The amount of information actually received can then be retrieved from the status variable, as with:

```
count MPI_Get_count(&status, MPI_FLOAT, &true_received_count);
```

```
received_source = status.MPI_SOURCE;  
received_tag = status.MPI_TAG;
```

MPI\_Recv blocks until the data transfer is complete and the received\_data variable is available for use.

The basic datatypes recognized by MPI are:

| <b>MPI datatype handle</b> | <b>C datatype</b> |
|----------------------------|-------------------|
| MPI_INT                    | int               |
| MPI_SHORT                  | short             |
| MPI_LONG                   | long              |
| MPI_FLOAT                  | float             |
| MPI_DOUBLE                 | double            |
| MPI_CHAR                   | char              |
| MPI_BYTE                   | unsigned char     |
| MPI_PACKED                 |                   |

**There also exist other types like: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, and MPI\_LONG\_DOUBLE.**

## **A common pattern of process interaction**

A common pattern of interaction among parallel processes is for one, the master, to allocate work to a set of slave processes and collect results from the slaves to synthesize a final result.

The master process will execute program statements like:

```
/* distribute portions of array1 to slaves. */  
for(an_id = 1; an_id < num_procs; an_id++) {  
    start_row = an_id*num_rows_per_process;  
    ierr = MPI_Send( &num_rows_to_send, 1, MPI_INT,  
                    an_id, send_data_tag, MPI_COMM_WORLD);  
    ierr = MPI_Send( &array1[start_row], num_rows_per_process,
```

```
        MPI_FLOAT, an_id, send_data_tag, MPI_COMM_WORLD);
    }

    /* and, then collect the results from the slave processes,
     * here in a variable called array2, and do something with them. */
    for(an_id = 1; an_id < num_procs; an_id++) {

        ierr = MPI_Recv( &array2, num_rows_returned, MPI_FLOAT,
                        MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        /* do something with array2 here */

    }

    /* and then print out some final result using the
     * information collected from the slaves. */
```

In this fragment, the master program sends a contiguous portion of array1 to each slave using MPI\_Send and then receives a response from each slave via MPI\_Recv. In practice, the master does not have to send an array; it could send a scalar or some other MPI data type, and it could construct array1 from any components to which it has access.

Here the returned information is put in array2, which will be written over every time a different message is received. Therefore, it will probably be copied to some other variable within the receiving loop.

Note the use of the MPI constant MPI\_ANY\_SOURCE to allow this MPI\_Recv call to receive messages from any process. In some cases, a program would need to determine exactly which process sent a message received using MPI\_ANY\_SOURCE. status.MPI\_SOURCE will hold that information, immediately following the call to MPI\_Recv.

The slave program to work with this master would resemble:

```
/* Receive an array segment, here called array2 */.

ierr = MPI_Recv( &num_rows_to_receive, 1, MPI_INT,
                root_process, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

ierr = MPI_Recv( &array2, num_rows_to_receive, MPI_FLOAT,
                root_process, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

/* Do something with array2 here, placing the result in array3,
 * and send array3 to the root process. */
```

```
ierr = MPI_Send( &array3, num_rows_to_return, MPI_FLOAT,  
                root_process, return_data_tag, MPI_COMM_WORLD);
```

There could be many slave programs running at the same time. Each one would receive data in array2 from the master via MPI\_Recv and work on its own copy of that data. Each slave would construct its own copy of array3, which it would then send to the master using MPI\_Send.

## A non-parallel program that sums the values in an array

The following program calculates the sum of the elements of a array. It will be followed by a parallel version of the same program using MPI calls.

```
#include <stdio.h>  
#define max_rows 10000000  
  
int array[max_rows];  
  
int main(int argc, char **argv)  
{  
    int i, num_rows;  
    long int sum;  
  
    printf("please enter the number of numbers to sum: ");  
    scanf("%i", &num_rows);  
  
    if(num_rows > max_rows) {  
        printf("Too many numbers.\n");  
        exit(1);  
    }  
  
    /* initialize an array */  
  
    for(i = 0; i < num_rows; i++) {  
        array[i] = i;  
    }  
  
    /* compute sum */  
  
    sum = 0;  
    for(i = 0; i < num_rows; i++) {  
        sum += array[i];  
    }  
  
    printf("The grand total is: %i\n", sum);  
}
```

```
    return 0;

}
```

## Design for a parallel program to sum an array

The code below shows a common Fortran structure for including both master and slave segments in the parallel version of the example program just presented. It is composed of a short set-up section followed by a single `if . . . else` loop where the master process executes the statements between the brackets after the `if` statement, and the slave processes execute the statements between the brackets after the `else` statement.

```
/* This program sums all rows in an array using MPI parallelism.
 * The root process acts as a master and sends a portion of the
 * array to each child process. Master and child processes then
 * all calculate a partial sum of the portion of the array assigned
 * to them, and the child processes send their partial sums to
 * the master, who calculates a grand total.
 */

#include <stdio.h>
#include <mpi.h>
int main()
{
int my_id, root_process, ierr, num_procs, an_id;
MPI_Status status;

root_process = 0;

/* Now replicate this process to create parallel processes.

ierr = MPI_Init(&argc, &argv);

/* find out MY process ID, and how many processes were started */

ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

if(my_id == root_process) {
/* I must be the root process, so I will query the user
 * to determine how many numbers to sum.

 * initialize an array,
```

```
    * distribute a portion of the array to each child process,  
    * and calculate the sum of the values in the segment assigned  
    * to the root process,  
  
    * and, finally, I collect the partial sums from slave processes,  
    * print them, and add them to the grand sum, and print it */  
}  
  
else {  
    /* I must be slave process, so I must receive my array segment,  
    * calculate the sum of my portion of the array,  
    * and, finally, send my portion of the sum to the root process. */  
  
}  
  
/* Stop this process */  
  
ierr = MPI_Finalize();  
return 0;  
}
```

## The complete parallel program to sum a array

Here is the expanded parallel version of the same program using MPI calls.

```
/* This program sums all rows in an array using MPI parallelism.  
 * The root process acts as a master and sends a portion of the  
 * array to each child process. Master and child processes then  
 * all calculate a partial sum of the portion of the array assigned  
 * to them, and the child processes send their partial sums to  
 * the master, who calculates a grand total.  
 **/  
  
#include <stdio.h>  
#include <mpi.h>  
  
#define max_rows 100000  
#define send_data_tag 2001  
#define return_data_tag 2002  
  
int array[max_rows];  
int array2[max_rows];
```

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```
int main(int argc, char **argv)
{
    long int sum, partial_sum;
    MPI_Status status;
    int my_id, root_process, ierr, i, num_rows, num_procs,
        an_id, num_rows_to_receive, avg_rows_per_process,
        sender, num_rows_received, start_row, end_row, num_rows_to_send;

    /* Now replicate this process to create parallel processes.
     * From this point on, every process executes a separate copy
     * of this program */

    ierr = MPI_Init(&argc, &argv);

    root_process = 0;

    /* find out MY process ID, and how many processes were started. */

    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    if(my_id == root_process) {

        /* I must be the root process, so I will query the user
         * to determine how many numbers to sum. */

        printf("please enter the number of numbers to sum: ");
        scanf("%i", &num_rows);

        if(num_rows > max_rows) {
            printf("Too many numbers.\n");
            exit(1);
        }

        avg_rows_per_process = num_rows / num_procs;

        /* initialize an array */

        for(i = 0; i < num_rows; i++) {
            array[i] = i + 1;
        }

        /* distribute a portion of the vector to each child process */

        for(an_id = 1; an_id < num_procs; an_id++) {
            start_row = an_id*avg_rows_per_process + 1;
```

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```
end_row = (an_id + 1)*avg_rows_per_process;

if((num_rows - end_row) < avg_rows_per_process)
    end_row = num_rows - 1;

num_rows_to_send = end_row - start_row + 1;

ierr = MPI_Send( &num_rows_to_send, 1 , MPI_INT,
                an_id, send_data_tag, MPI_COMM_WORLD);

ierr = MPI_Send( &array[start_row], num_rows_to_send, MPI_INT,
                an_id, send_data_tag, MPI_COMM_WORLD);
}

/* and calculate the sum of the values in the segment assigned
 * to the root process */

sum = 0;
for(i = 0; i < avg_rows_per_process + 1; i++) {
    sum += array[i];
}

printf("sum %i calculated by root process\n", sum);

/* and, finally, I collect the partial sums from the slave processes,
 * print them, and add them to the grand sum, and print it */

for(an_id = 1; an_id < num_procs; an_id++) {

    ierr = MPI_Recv( &partial_sum, 1, MPI_LONG, MPI_ANY_SOURCE,
                    return_data_tag, MPI_COMM_WORLD, &status);

    sender = status.MPI_SOURCE;

    printf("Partial sum %i returned from process %i\n", partial_sum,
sender);

    sum += partial_sum;
}

printf("The grand total is: %i\n", sum);
}

else {

    /* I must be a slave process, so I must receive my array segment,
     * storing it in a "local" array, array1. */
```



**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```

ierr = MPI_Recv( &num_rows_to_receive, 1, MPI_INT,
                root_process, send_data_tag, MPI_COMM_WORLD, &status);

ierr = MPI_Recv( &array2, num_rows_to_receive, MPI_INT,
                root_process, send_data_tag, MPI_COMM_WORLD, &status);

num_rows_received = num_rows_to_receive;

/* Calculate the sum of my portion of the array */

partial_sum = 0;
for(i = 0; i < num_rows_received; i++) {
    partial_sum += array2[i];
}

/* and finally, send my partial sum to the root process */

ierr = MPI_Send( &partial_sum, 1, MPI_LONG, root_process,
                return_data_tag, MPI_COMM_WORLD);
}
ierr = MPI_Finalize();
return 0;
}

```

The following table shows the values of several variables during the execution of `sumarray_mpi`. The information comes from a two-processor parallel run, and the values of program variables are shown in **both** processor memory spaces. Note that there is only one process active prior to the call to `MPI_Init`.

**Value histories of selected variables  
within the master and slave processes  
during a 2-process execution of program `sumarray_mpi`**

| Program location          | Before MPI_Init | After MPI_Init |        | Before MPI_Send to slave |        | After MPI_Recv by slave |        | After MPI_Recv by master |        |
|---------------------------|-----------------|----------------|--------|--------------------------|--------|-------------------------|--------|--------------------------|--------|
|                           | Proc 0          | Proc 0         | Proc 1 | Proc 0                   | Proc 1 | Proc 0                  | Proc 1 | Proc 0                   | Proc 1 |
| <code>root_process</code> | 0               | 0              | 0      | 0                        | 0      | 0                       | 0      | 0                        | 0      |
| <code>my_id</code>        | .               | 0              | 1      | 0                        | 1      | 0                       | 1      | 0                        | 1      |
| <code>num_procs</code>    | .               | 2              | 2      | 2                        | 2      | 2                       | 2      | 2                        | 2      |
| <code>num_rows</code>     | .               | .              | .      | 6                        | .      | 6                       | .      | 6                        | .      |

|                      |   |   |   |     |   |     |     |      |      |
|----------------------|---|---|---|-----|---|-----|-----|------|------|
| avg_rows_per_process | . | . | . | 3   | . | 3   | .   | 3    | .    |
| num_rows_received    | . | . | . | .   | . | .   | 3   | .    | 3    |
| array[0]             | . | . | . | 1.0 | . | 1.0 | .   | 1.0  | .    |
| array[1]             | . | . | . | 2.0 | . | 2.0 | .   | 2.0  | .    |
| array[2]             | . | . | . | 3.0 | . | 3.0 | .   | 3.0  | .    |
| array[3]             | . | . | . | 4.0 | . | 4.0 | .   | 4.0  | .    |
| array[4]             | . | . | . | 5.0 | . | 5.0 | .   | 5.0  | .    |
| array[5]             | . | . | . | 6.0 | . | 6.0 | .   | 6.0  | .    |
| array2[0]            | . | . | . | .   | . | .   | 4.0 | .    | 4.0  |
| array2[1]            | . | . | . | .   | . | .   | 5.0 | .    | 5.0  |
| array2[2]            | . | . | . | .   | . | .   | 6.0 | .    | 6.0  |
| array2[3]            | . | . | . | .   | . | .   | .   | .    | .    |
| array2[4]            | . | . | . | .   | . | .   | .   | .    | .    |
| array2[5]            | . | . | . | .   | . | .   | .   | .    | .    |
| partial_sum          | . | . | . | .   | . | .   | .   | 6.0  | 15.0 |
| sum                  | . | . | . | .   | . | .   | .   | 21.0 | .    |

## Logging and tracing MPI activity

It is possible to use `mpirun` to record MPI activity, by using the options `-mpilog` and `-mpitrace`. For more information about this facility see `man mpirun`.

## Collective operations

MPI\_Send and MPI\_Recv are "point-to-point" communications functions. That is, they involve one sender and one receiver. MPI includes a large number of subroutines for performing "collective" operations. Collective operation are performed by MPI routines that are called by each member of a group of processes that want some operation to be performed for them as a group. A collective function may specify one-to-many, many-to-one, or many-to-many message transmission.

MPI supports three classes of collective operations:

- synchronization,

- data movement, and
- collective computation

These classes are not mutually exclusive, of course, since blocking data movement functions also serve to synchronize process activity, and some MPI routines perform both data movement and computation.

## Synchronization

The MPI\_Barrier function can be used to synchronize a group of processes. To synchronize a group of processes, each one must call MPI\_Barrier when it has reached a point where it can go no further until it knows that all its cohorts have reached the same point. Once a process has called MPI\_Barrier, it will be blocked until all processes in the group have also called MPI\_Barrier.

## Collective data movement

There are several routines for performing collective data distribution tasks:

MPI\_Bcast

Broadcast data to other processes

MPI\_Gather, MPI\_Gatherv

Gather data from participating processes into a single structure

MPI\_Scatter, MPI\_Scatterv

Break a structure into portions and distribute those portions to other processes

MPI\_Allgather, MPI\_Allgatherv

Gather data from different processes into a single structure that is then sent to all participants (Gather-to-all)

MPI\_Alltoall, MPI\_Alltoallv

Gather data and then scatter it to all participants (All-to-all scatter/gather)

The routines with "V" suffixes move variable-sized blocks of data.

The subroutine MPI\_Bcast sends a message from one process to all processes in a communicator.

```
int MPI_Bcast(void *data_to_be_sent, int send_count, MPI_Datatype send_type,  
             int broadcasting_process_ID, MPI_Comm comm);
```

When processes are ready to share information with other processes as part of a broadcast, **ALL of them must execute a call to MPI\_BCAST**. There is no separate MPI call to receive a broadcast.

MPI\_Bcast could have been used in the program `sumarray_mpi` presented earlier, in place of the MPI\_Send loop that distributed data to each process. Doing so would have resulted in excessive data movement, of course. A better solution would be MPI\_Scatter or MPI\_Scatterv.

The subroutines MPI\_Scatter and MPI\_Scatterv take an input array, break the input data into separate portions and send a portion to each one of the processes in a communicating group.

```
int MPI_Scatter(void *send_data, int send_count, MPI_Datatype send_type,  
              void *receive_data, int receive_count, MPI_Datatype receive_type,  
              int sending_process_ID, MPI_Comm comm);
```

or

```
int MPI_Scatterv(void *send_data, int *send_count_array, int *send_start_array,  
                MPI_Datatype send_type, void *receive_data, int receive_count,  
                MPI_Datatype receive_type, int sender_process_ID, MPI_Comm comm);
```

- `data_to_send`: variable of a C type that corresponds to the MPI `send_type` supplied below
- `send_count`: number of data elements to send (int)
- `send_count_array`: array with an entry for each participating process containing the number of data elements to send to that process (int)
- `send_start_array`: array with an entry for each participating process containing the displacement relative to the start of `data_to_send` for each data segment to send (int)
- `send_type`: datatype of elements to send (one of the MPI datatype handles)
  
- `receive_data`: variable of a C type that corresponds to the MPI `receive_type` supplied below
- `receive_count`: number of data elements to receive (int)
- `receive_type`: datatype of elements to receive (one of the MPI datatype handles)
- `sender_ID`: process ID of the sender (int)
- `receive_tag`: receive tag (int)
- `comm`: communicator (handle)
- `status`: status object (MPI\_Status)

The routine MPI\_Scatterv could have been used in the program `sumarray_mpi` presented earlier, in place of the MPI\_Send loop that distributed data to each process.

MPI\_Bcast, MPI\_Scatter, and other collective routines build a communication tree among the participating processes to minimize message traffic. If there are N processes involved, there would normally be N-1 transmissions during a broadcast operation, but if a tree is built so that the broadcasting process sends the broadcast to 2 processes, and they each send it on to 2 other processes, the total number of messages transferred is only  $O(\ln N)$ .

## Collective computation routines

Collective computation is similar to collective data movement with the additional feature that data may be modified as it is moved. The following routines can be used for collective computation.

### MPI\_Reduce

Perform a reduction operation. That is, apply some operation to some operand in every participating process. For example, add an integer residing in every process together and put the result in a process specified in the MPI\_Reduce argument list.

### MPI\_Allreduce

Perform a reduction leaving the result in all participating processes

### MPI\_Reduce\_scatter

Perform a reduction and then scatter the result

### MPI\_Scan

Perform a reduction leaving partial results (computed up to the point of a process's involvement in the reduction tree traversal) in each participating process. (parallel prefix)

The subroutine MPI\_Reduce combines data from all processes in a communicator using one of several reduction operations to produce a single result that appears in a specified target process.

```
int MPI_Reduce(void *data_to_be_sent, void *result_to_be_received_by_target,  
              int send_count, MPI_Datatype send_type, MPI_Op operation,  
              int target_process_ID, MPI_Comm comm);
```

When processes are ready to share information with other processes as part of a data reduction, all of the participating processes execute a call to MPI\_Reduce, which uses local data to calculate each process's portion of the reduction operation and communicates the local result to other processes as necessary. Only the target\_process\_ID receives the final result.

MPI\_Reduce could have been used in the program sumarray\_mpi presented earlier, in place of the MPI\_Recv loop that collected partial sums from each process.

## Collective computation built-in operations

Many of the MPI collective computation routines take both built-in and user-defined combination functions. The built-in functions are:

| Operation handle | Operation                  |
|------------------|----------------------------|
| MPI_MAX          | Maximum                    |
| MPI_MIN          | Minimum                    |
| MPI_PROD         | Product                    |
| MPI_SUM          | Sum                        |
| MPI_LAND         | Logical AND                |
| MPI_LOR          | Logical OR                 |
| MPI_LXOR         | Logical Exclusive OR       |
| MPI_BAND         | Bitwise AND                |
| MPI_BOR          | Bitwise OR                 |
| MPI_BXOR         | Bitwise Exclusive OR       |
| MPI_MAXLOC       | Maximum value and location |
| MPI_MINLOC       | Minimum value and location |

## A collective operation example

The following program integrates the function  $\sin(X)$  over the range 0 to 2 pi. It will be followed by a parallel version of the same program that uses the MPI library.

```
/* program to integrate sin(x) between 0 and pi by computing
 * the area of a number of rectangles chosen so as to approximate
 * the shape under the curve of the function.
 *
 * 1) ask the user to choose the number of intervals,
 * 2) compute the interval width (rect_width),
 * 3) for each interval:
 *
 * a) find the middle of the interval (x_middle),
 * b) compute the height of the rectangle, sin(x_middle),
 * c) find the area of the rectangle as the product of
 *    the interval width and its height sin(x_middle), and
 * d) increment a running total.
 */
#include <stdio.h>
```

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

```
#include <math.h>

#define PI 3.1415926535

int main(int argc, char **argv)
{
    int i, num_intervals;
    double rect_width, area, sum, x_middle;

    printf("Please enter the number of intervals to interpolate: ");
    scanf("%i", &num_intervals);

    rect_width = PI / num_intervals;

    sum = 0;
    for(i = 1; i < num_intervals + 1; i++) {

        /* find the middle of the interval on the X-axis. */

        x_middle = (i - 0.5) * rect_width;
        area = sin(x_middle) * rect_width;
        sum = sum + area;
    }

    printf("The total area is: %f\n", (float)sum);
    return 0;
}
```

The next program is an MPI version of the program above. It uses MPI\_Bcast to send information to each participating process and MPI\_Reduce to get a grand total of the areas computed by each participating process.

```
/* This program integrates sin(x) between 0 and pi by computing
 * the area of a number of rectangles chosen so as to approximate
 * the shape under the curve of the function using MPI.
 *
 * The root process acts as a master to a group of child process
 * that act as slaves. The master prompts for the number of
 * interpolations and broadcasts that value to each slave.
 *
 * There are num_procs processes all together, and a process
 * computes the area defined by every num_procs-th interval,
 * collects a partial sum of those areas, and sends its partial
 * sum to the root process.
 */
```

**Jorhat Engineering College**  
**Department of Computer Science and Engineering**  
**Jorhat 785007**

---

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

#define PI 3.1415926535

int main(int argc, char **argv)
{
    int my_id, root_process, num_procs, ierr, num_intervals, i;
    double rect_width, area, sum, x_middle, partial_sum;
    MPI_Status status;

    /* Let process 0 be the root process. */
    root_process = 0;

    /* Now replicate this process to create parallel processes. */
    ierr = MPI_Init(&argc, &argv);

    /* Find out MY process ID, and how many processes were started. */
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    if(my_id == root_process) {
        /* I must be the root process, so I will query the user
           to determine how many interpolation intervals to use. */
        printf("Please enter the number of intervals to interpolate: ");
        scanf("%i", &num_intervals);
    }

    /* Then...no matter which process I am:
     *
     * I engage in a broadcast so that the number of intervals is
     * sent from the root process to the other processes, and ...
     */
    ierr = MPI_Bcast(&num_intervals, 1, MPI_INT, root_process,
                    MPI_COMM_WORLD);

    /* calculate the width of a rectangle, and */
    rect_width = PI / num_intervals;

    /* then calculate the sum of the areas of the rectangles for
```



```

    * which I am responsible. Start with the (my_id +1)th
    * interval and process every num_procs-th interval thereafter.
    **/
    partial_sum = 0;
    for(i = my_id + 1; i < num_intervals + 1; i += num_procs) {

        /* Find the middle of the interval on the X-axis. */
        x_middle = (i - 0.5) * rect_width;
        area = sin(x_middle) * rect_width;
        partial_sum = partial_sum + area;
    }
    printf("proc %i computes: %f\n", my_id, (float)partial_sum);

    /* and finally, engage in a reduction in which all partial sums
    * are combined, and the grand sum appears in variable "sum" in
    * the root process,
    **/
    ierr = MPI_Reduce(&partial_sum, &sum, 1, MPI_DOUBLE,
        MPI_SUM, root_process, MPI_COMM_WORLD);

    /* and, if I am the root process, print the result. */

    if(my_id == root_process) {
        printf("The integral is %f\n", (float)sum);

        /* (yes, we could have summed just the heights, and
        * postponed the multiplication by rect_width til now.) */
    }

    /* Close down this processes. */

    ierr = MPI_Finalize();
    return 0;
}

```

## Simultaneous send and receive

The subroutine `MPI_Sendrecv` exchanges messages with another process. A send-receive operation is useful for avoiding some kinds of unsafe interaction patterns and for implementing remote procedure calls.

A message sent by a send-receive operation can be received by `MPI_Recv` and a send-receive operation can receive a message sent by an `MPI_Send`.

`MPI_Sendrecv(&data_to_send, send_count, send_type, destination_ID, send_tag,`

```
&received_data, receive_count, receive_type, sender_ID, receive_tag,  
comm, &status)
```

- `data_to_send`: variable of a C type that corresponds to the MPI `send_type` supplied below
- `send_count`: number of data elements to send (int)
- `send_type`: datatype of elements to send (one of the MPI datatype handles)
- `destination_ID`: process ID of the destination (int)
- `send_tag`: send tag (int)
- `received_data`: variable of a C type that corresponds to the MPI `receive_type` supplied below
- `receive_count`: number of data elements to receive (int)
- `receive_type`: datatype of elements to receive (one of the MPI datatype handles)
- `sender_ID`: process ID of the sender (int)
- `receive_tag`: receive tag (int)
- `comm`: communicator (handle)
- `status`: status object (MPI\_Status)

## **MPI tags**

MPI\_Send and MPI\_Recv, as well as other MPI routines, allow the user to specify a tag value with each transmission. These tag values may be used to specify the message type, or "context," in a situation where a program may receive messages of several types during the same program. The receiver simply checks the tag value to decide what kind of message it has received.

## **MPI communicators**

Every MPI communication operation involves a "communicator." Communicators identify the group of processes involved in a communication operation and/or the context in which it occurs. The source and destination processes specified in point-to-point routines like MPI\_Send and MPI\_Recv must be members of the specified communicator and the two calls must reference the same communicator.

Collective operations include just those processes identified by the communicator specified in the calls.

The communicator `MPI_COMM_WORLD` is defined by default for all MPI runs, and includes all processes defined by `MPI_Init` during that run. Additional communicators can be defined that include all or part of those processes. For example, suppose a group of processes needs to engage in two different reductions involving disjoint sets of processes. A communicator can be defined for each subset of `MPI_COMM_WORLD` and specified in the two reduction calls to manage message transmission.

`MPI_Comm_split` can be used to create a new communicator composed of a subset of another communicator. `MPI_Comm_dup` can be used to create a new communicator composed of all of the members of another communicator. This may be useful for managing interactions within a set of processes in place of message tags.

## Exercises

Here are some exercises for continuing your investigation of MPI:

- Convert the hello world program to print its messages in rank order.
- Convert the example program `sumarray_mpi` to use `MPI_Scatter` and/or `MPI_Reduce`.
- Write a program to find all positive primes up to some maximum value, using `MPI_Recv` to receive requests for integers to test. The master will loop from 2 to the maximum value on
  1. issue `MPI_Recv` and wait for a message from any slave (`MPI_ANY_SOURCE`),
  2. if the message is zero, the process is just starting,  
if the message is negative, it is a non-prime,  
if the message is positive, it is a prime.
  3. use `MPI_Send` to send a number to test.and each slave will send a request for a number to the master, receive an integer to test, test it, and return that integer if it is prime, but its negative value if it is not prime.
- Write a program to send a token from processor to processor in a loop.