IMPLEMENTATION OF GLOBAL SNAPSHOT ALGORITHM USING MPI

SUBMITTED BY:-

SAURA MANDAL
CLASS ROLL NUMBER: 001311001014
REGISTRATION NUMBER: 123600 OF 2013-14

TARUN NASKAR
CLASS ROLL NUMBER: 001311001027
REGISTRATION NUMBER: 123613 OF 2013-14

A PROJECT SUBMITTED TO
THE FACULTY OF ENGINEERING & TECHNOLOGY, JADAVPUR UNIVERSITY

UNDER THE SUPERVISION OF

MR. UTPAL KUMAR RAY
ASSISTANT PROFESSOR
DEPARTMENT OF INFORMATION TECHNOLOGY
JADAVPUR UNIVERSITY
2017

Acknowledgement

The satisfaction we have on the successful completion of the project will be incomplete without the mention of the people whose constant guidance and encouragement helped us to achieve success.

Firstly, we offer our sincere gratitude to the DEPARTMENT OF INFORMATION TECHNOLOGY, JADAVPUR UNIVERSITY, for giving us the opportunity to pursue this project during the degree course of Bachelor of Engineering in Information Technology and thereby helping us to improve our career.

We are also grateful to our guide, Prof. Utpal Kumar Ray for his support and invaluable help during the entire duration of the project. His inspiring presence, constant encouragement, meticulous guidance and painstaking efforts helped us to tide over the problems encountered during the various phases of the project.

It is also a pleasure to acknowledge the support received from Mr. Ryan Saptarshi Ray, PhD Scholar and Mr. Debendranath Das , M.E. Final Year student of the Information Technology department for their guidance and encouragement during the course of our project.

We also want to express our heartfelt gratitude to our friends and classmates for their support and assistance in many ways in completion of the project.

We take this opportunity to express our deep sense of gratitude with profound obeisance and veneration to our beloved parents, for their constant encouragement, support and blessings, which enabled us to complete the project in time.

(Saura Mandal)
B.E in Information Technology
JADAVPUR UNIVERSITY

(Tarun Naskar)
B.E in Information Technology
JADAVPUR UNIVERSITY

ABSTRACT

Recording on-the-fly global states of distributed executions is an important paradigm when one is interested in analyzing, testing, or verifying properties associated with these executions. This problem is called ***the snapshot problem.*** Unfortunately, the lack of both a globally shared memory and a global clock in a distributed system, added to the fact that transfer delays in these systems are finite but unpredictable, makes this problem non-trivial.

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be solved individually. Distributed computing deals with all forms of computing, information access, and information exchange across multiple processing platforms connected by computer networks. A multicomputer parallel system is a parallel system in which the multiple processors do not have direct access to shared memory. Such computers usually do not have a common clock. The different computers in the cluster communicate with each other through message passing. MPI provides a platform to perform tasks without the usage of shared memory. The component architecture of MPI provides both a stable platform for third-party research as well as enabling the run-time composition of independent software add-ons.

This project report discusses the issues which have to be addressed to compute distributed snapshots in a consistent way using MPI. It explains the underlying concepts of distributed system model and implements the Chandy Lamport's Global snapshot algorithm in MPI.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

The global state of a distributed system is a collection of the local states of its components. Recording the global state of a distributed system is an important paradigm and it finds applications in several aspects of distributed system design. A snapshot-recording method has been used in the distributed debugging facility of Estelle [1, 2], a distributed programming environment. Other applications include monitoring distributed events [3] such as in industrial process control, setting distributed breakpoints [4], protocol specification and verification, and discarding obsolete information. Therefore, it is important that there be efficient ways of recording the global state of a distributed system.

1.2 FOCUS

In this project report we discuss how to implement the global snapshot model using MPI in a multicomputing environment on a money transaction system which we have developed. We have also shown the performance results for end-to-end point implementation. We are using the OpenMPI software of MPI. OpenMPI is an all-new, production quality MPI-2 implementation that is fundamentally centered on component concepts. OpenMPI provides a unique combination of novel features previously unavailable in an open-source, production-quality implementation of MPI **[5]**.

In this project we have developed a money transaction system which is being implemented to transfer a random amount of money from any node to any other node in a distributed environment. Initially, all the nodes contain equal amount of money (Rs. 10000). Here, we have used two point-to-point MPI library routines between two nodes, i.e., MPI Isend () and MPI Irecv (). The transaction system should be such that even after many transactions, the total amount of money which the system possesses should always remain consistent. The system is continuously executing so that global snapshots can be taken.

1.3 **ORGANIZATION**

Chapter 2 gives overview of Multicomputing and MPI.
Chapter 3 describes the money transaction system we have developed and explains its relevance with the global snapshot algorithm.
Chapter 4 gives an overview of Chandy Lamport's Global snapshot algorithm.
Chapter 5 implements the Global snapshot algorithm on the money transaction system using MPI. The chapter also contains appropriate code snippets and observations.
Chapter 6 concludes the thesis and discusses about further work, which can be done. **References** and **Appendices** are also given.

CHAPTER 2

MULTICOMPUTING AND MPI BASICS

Chapter Gist: This chapter gives an overview of Multicomputing and MPI.

2.1 MULTICOMPUTING OVERVIEW

Distributed memory uses massively parallel multicomputers, which can provide high levels of performance required to solve the Grand Challenge computational science problems. To tackle such challenges we program in a message passing model. A distributed computing system consists of spatially separated processes that do not share a common memory and communicate asynchronously with each other by passing messages over communication channels. Each component of a distributed system has a local state. The state of a process is characterized by the state of its local memory and a history of its activity. The state of a channel is characterized by the set of messages sent along the channel less the messages received along the channel.

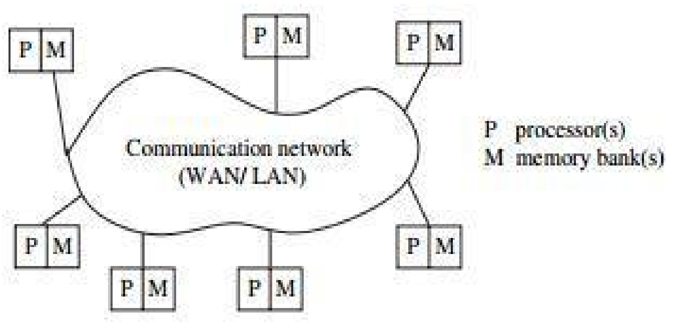


Fig. 2.1: A Distributed System

2.2 MPI OVERVIEW

MPI stands for Message Passing Interface and is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementers, and users.

MPI consists of:

- A header file **"mpi.h"**
- A library of routines and functions, and
- A runtime system.

MPI is for parallel computers, clusters, and heterogeneous networks. MPI is full-featured. MPI is designed to provide access to advanced parallel hardware for end users, library writers and tool developers. MPI can be used with C/C++, FORTRAN, and many other languages. MPI is actually just an Application Programming Interface (API). As such, MPI

- Specifies what a call to each routine should look like, and how each routine should behave.
- Does not specify how each routine should be implemented, and sometimes is intentionally vague about certain aspects of a routines behavior.
- Implementations are often platform vendor specific.
- Has several open-source implementations.

2.3 MPI ROUTINES

The following are some basic routines used in MPI programs:

- **MPI_Init()** starts the MPI runtime environment.
- **MPI_Comm_size()** gets the number of processes, $\mathbf{N_p}$.
- **MPI_Comm_rank()** gets the process ID of the current process which is between **0** and $\mathbf{N_p - 1}$, inclusive. (These last two routines are typically called right after MPI_Init()).
- **MPI_Send()** sends a message from the current process to another process (the destination).
- **MPI_Recv()** receives a message on the current process from another process (the source).
- **MPI_Finalize()** shuts down the MPI runtime environment.

MPI Send()

The calling sequence for MPI_Send() is:

```
int MPI_Send (
        void * buf ,                        /* pointer to send buffer */
         int count ,                        /* number of items to send */
        MPI_Datatype datatype ,         /* data type of buffer elements */
        int dest ,                          /* rank of destination process */
        int tag ,                           /* message type identifier */
        MPI_Comm comm                     /* MPI communicator to use */
)
```

The MPI_Send() function initiates a blocking send. Here "blocking" does not indicate that the sender waits for the message to be received, but rather that the sender waits for the message to be accepted by the MPI system. It does mean that once this function returns the send buffer may be changed without impacting the send operation.

MPI Recv()

The calling sequence for MPI_Recv() is:

```
int MPI_Recv (
        void * buf ,                          /* pointer to send buffer */
        int count ,                           /* number of items to send */
        MPI_Datatype datatype ,               /* datatype of buffer elements */
        int source ,                          /* rank of sending process */
        int tag ,                             /* message type identifier */
        MPI_Comm comm ,                       /* MPI communicator to use */
        MPI_Status * status )                /* MPI status object */
)
```

The **MPI_Recv()** function initiates a blocking receive. It will not return to its caller until a message with the specified tag is received from the specified source. **MPI_ANY_SOURCE** may be used to indicate that the message should be accepted from any source. **MPI_ANY_TAG** may be used to indicate the message should be accepted regardless of its tag.

MPI_Barrier(MPI_COMM_WORLD)

The name of the function is quite descriptive - the function forms a barrier, and no processes in the communicator can pass the barrier until all of them call the function.

Here is an illustration of MPI_Barrier. In the figure below (Fig. 2.2) the horizontal axis represents execution of the program and the circles represent different processes **[6]**:

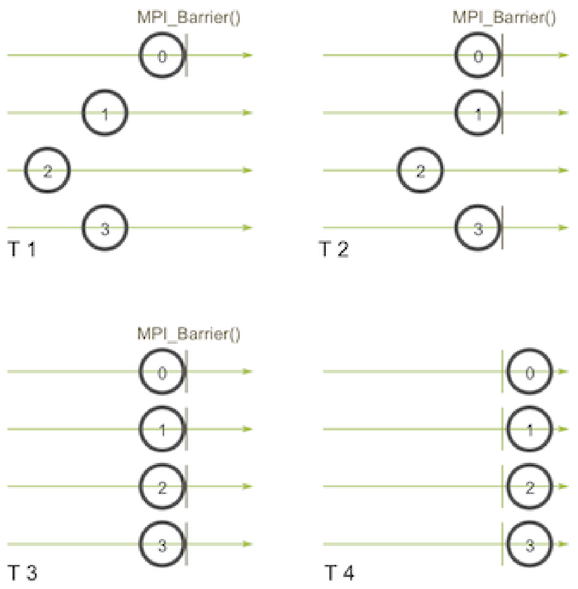


Fig. 2.2: MPI BARRIER

MPI_Isend()

The calling sequence for MPI_Isend() is:

int MPI_Isend(

const void *buf,	**/*Initial address of receive buffer (choice) */**
int count,	**/* Number of elements in receive buffer */**
MPI_Datatype datatype,	**/* Datatype of each receive buffer element*/**
int dest,	**/* Rank of destination (integer) */**
int tag,	**/* Message tag (integer) */**
MPI_Comm comm,	**/* Communicator (handle)*/**
MPI_Request *request)	**/* Communication request (handle)*/**

MPI_Isend starts a standard-mode, nonblocking send. Nonblocking calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

A send request can be considered to be complete by calling MPI_Wait, MPI_Waitany, MPI_Test, or MPI_Testany with request returned by this function.

MPI_Irecv()

The calling sequence for MPI_Irecv() is:

int MPI_Irecv(

```
void *buf,                  /*Initial address of receive buffer (choice) */
int count,                  /* Number of elements in receive buffer */
MPI_Datatype datatype,      /* Datatype of each receive buffer element*/
int source,                 /* Rank of source (integer) */
int tag,                    /* Message tag (integer) */
MPI_Comm comm,              /* Communicator (handle)*/
MPI_Request *request)       /* Communication request (handle)*/
```

Description: - Nonblocking calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

2.4 Compiling an MPI program

- Compiling a program for MPI is almost just like compiling a regular C or C++ program

- Compiler “front-ends” have been defined that know where the **mpi.h** file and the proper libraries are found.
- The C compiler is **mpicc** and the C++ compiler is **mpic++**.

For example, to compile hello.c one would use the command

```
mpicc hello.c -o hello
```

Here, hello is the name of the object file.

2.5 Running an MPI program

To run a program hello.c on a single machine we use the command:

```
mpirun -np n hello
```

Here, n denotes the number of processors and hello is the name of the object file.

To run a program hello.c on multiple machines we use the command:

```
mpirun -np n -pernode --hostfile my_host hello
```

Here, pernode ensures that one process executes in each node and hostfile contains the IP addresses of the nodes. The first IP address of the hostfile is the master processor’s IP Address.

CHAPTER 3

MONEY TRANSACTION SYSTEM

Chapter Gist: This chapter describes the money transaction system we have developed and explains its relevance with the global snapshot algorithm.

3.1 Problem Statement

A money transaction system is a system which is being implemented to transfer a random amount of money from any node to any other node in a distributed environment. Initially, all the nodes contain equal amount of money (Rs. 10000). Here, we have used two point-to-point MPI library functions between two nodes, i.e., MPI Isend () and MPI Irecv (). The transaction system should be such that even after many transactions, the total amount of money which the system possesses should always remain consistent. The system is continuously executing so that global snapshots can be taken.

3.2 Relevance with Global Snapshot Algorithm

A Global snapshot system can be described as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels, where nodes are indexed from 1 to ***n*** that are connected by channels. There is no globally shared memory and processes communicate solely by passing messages. There is no physical global clock in the system. Message send and receive is asynchronous. Messages are delivered reliably with finite but arbitrary time delay. For checking the global snapshot algorithm, we need a system of a number of nodes (n) in which the nodes are continuously passing values among themselves. The money transaction system which has been described in Section **3.1** is one such system.

CHAPTER 4

CHANDY LAMPORT'S GLOBAL SNAPSHOT ALGORITHM

Chapter Gist: This chapter gives an overview of Chandy Lamport's Global snapshot algorithm.

4.1 Principle

The Global State of a distributed system is a collection of local states of processes and channels.
A global state GS is defined as:

$$GS = \{\bigcup_i LS_i, \bigcup_{i,j} SC_{ij}\}$$

A global state GS is a consistent global state if it satisfies the following two conditions:

C1: $\text{send}(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus \text{rec}(m_{ij}) \in LS_j$. ($\oplus$ is Ex-OR operator.)
C2: $\text{send}(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge \text{rec}(m_{ij}) \notin LS_j$.

After a site has recorded its snapshot, it sends a control message, called a ***marker,*** along all its outgoing channels before sending out any more messages. Since channels are **FIFO,** a marker separates the messages in the channel into those to be included in the snapshot (is. channel state or process state) from those not to be recorded in the snapshot. The role of markers in a **FIFO** system is to act as delimiters for the messages in the channels so that the channel state recorded by the process at the receiving end of the channel satisfies the condition C2. Since all messages that follow a marker on channel C_{ij} have been sent by process ***i*** after ***i*** has taken its snapshot, process ***j*** must record its snapshot not later than when it receives a marker on channel C_{ij}.

4.2 The Algorithm

A process initiates snapshot collection by executing the '**Marker Sending Rule**' by which it records its local state and sends a marker on each outgoing channel. A process executes the '**Marker Receiving Rule'** on receiving a marker. If the process has not yet recorded its local state, it executes the 'Marker Sending Rule' to record its local state. The state of the incoming channel on which the marker is received is recorded as being the set of computation messages received on that channel after recording the local state but before receiving the marker on that channel. The algorithm can be initiated by any process by executing the 'Marker Sending Rule'. To prove the correctness of the algorithm, we now show that a recorded snapshot satisfies conditions **C1** and **C2.** Since a process records its snapshot when it receives the first marker on any incoming channel, no messages that follow markers on the channels incoming to it are recorded in the process's snapshot. Moreover, a process stops recording the state of an incoming channel when a marker is received on that channel. Due to the **FIFO** property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition **C2** is satisfied.

When a process j receives message m_i, that precedes the marker on channel C_{ij}, it acts *as* follows: If process ***j*** has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records $\mathbf{m}_{ij}$ in the state of the channel $\mathbf{C}_{ij}$. Thus, condition C1 is satisfied.

The recorded local snapshots can be put together to create the global snapshot in several ways. One policy is to have each process send its local snapshot to the initiator of the algorithm.
Another policy is to have each process send the information it records along all outgoing channels, and to have each process receiving such information for the first time propagate it along its outgoing channels.
All the local snapshots get disseminated to all other processes and all the processes can determine the global state. The recording part of a single instance of the algorithm requires **O (e)** messages and **O (d)** time, where **e** is the number of edges in the graph and d is the diameter of the graph.

Algorithm:

```
Marker Sending Rule for process i:

begin
    (i)    Process i records its state.
    (ii)   For each outgoing channel C on which a marker has not been sent, i sends a marker along C
               before i sends further messages along C.
end

Marker Receiving Rule for process j:

On receiving a marker along channel C:
If j has not recorded its state then
    begin
        (i)    Record the state of C as the empty set.
        (ii)   Follow the 'Marker Sending Rule'.
    end

    else
               Record the state of C as the set of messages
               received along C after j's state was recorded
               and before j received the marker along C.
```

4.3 Proof:

- If A and B happen on the same process, then causal consistency is trivially true.
- Consider, when A is the send and B is the corresponding receive event on processes p and q, respectively **[9]**
  - Since B is pre snapshot, q has not received a marker and p has not sent a marker.
  - A must also happen pre snapshot.
- Similar logic for A happening post snapshot.

4.3.1 Proof: Part I

- In order for an application message m in the channel from process p to process q to be in the snapshot
  – Must happen after q has received its first marker
  – Before p has sent its marker to q

- A message m will only be in the snapshot if the sending process was pre snapshot and the receiving process was post snapshot.

4.3.2 Proof: Part 2

- How do we order concurrent events?
  – Remember, all processes communicate
- What if a process receives a marker in between sending a marker and some event?
  – These should happen atomically
- What if something happens on a process independently of messages after the wall-clock time of when the snapshot starts?
  – Snapshots are causally consistent

This we can conclude that for FIFO channels, Chandy Lamport's Global Snapshot always works. Hence, the system remains consistent.

4.4 Example

Let **S1** and **S2** be two distinct sites of a distributed system, which maintain bank accounts A, and B respectively. A site refers to a process in this example. Let the communication channels from site **S1** to site **S2** and from site **S2** to site **S1** be denoted by **C_{12}** and **C_{21}** respectively.

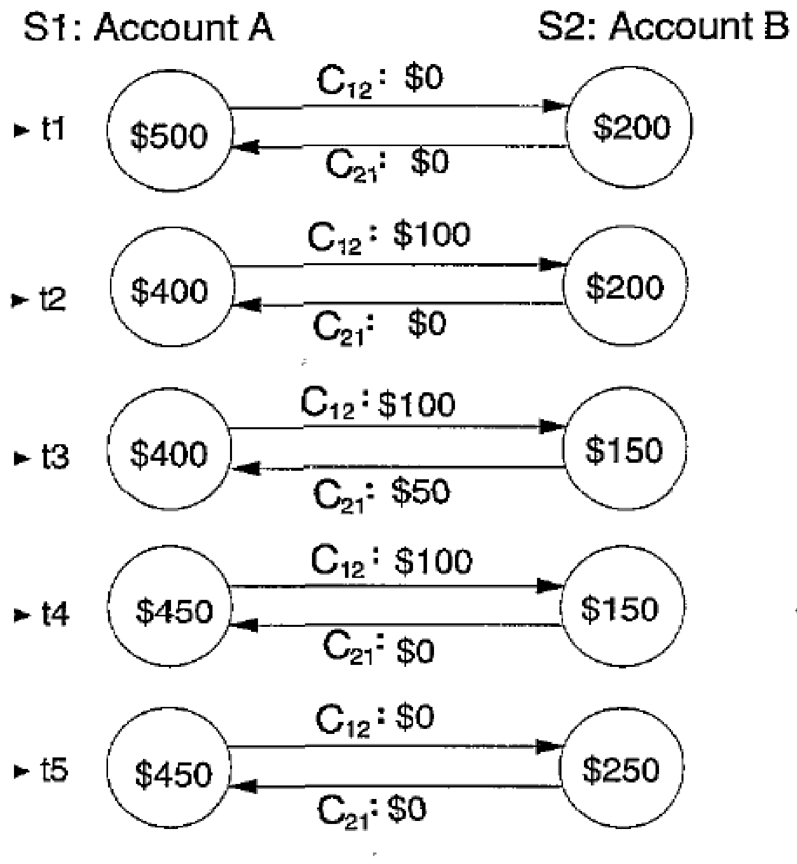


Fig. 4.1: An example for demonstrating Chandy Lamport's Algorithm

Let us consider a sequence of actions:-

a) Initially, Account A= **\$500**, Account B= **\$200**, C_{12} = **\$0**, C_{21} = **\$0**.

b) Site ***S1*** initiates a transfer of **\$100** from Account A to Account B. Account A is decremented by **\$100** to **\$400** and a request for **\$100** credit to Account **B** is sent on Channel C_{12} to ***site S2***. Account **A** = \$400, Account B = **\$200**, C_{12} = **\$100**, C_{21} = **\$0**.

c) Site ***S2*** initiates a transfer of **\$50** from Account B to Account A. Account B is decremented by **\$50** to **\$150** and a request for **\$50** credit to Account A is sent on Channel C_{21} to site S1.
Account A = \$400, Account B **= \$150**, C_{12} = **\$100**, C_{21} = **\$50**.

d) Site ***S1*** receives the message for a \$50 credit to Account A and updates Account A.
Account A = **\$450**, Account B = **\$150**, C_{12} = **\$100**, C_{21} = **\$0**.

e) Site ***S2*** receives the message for a **\$100** credit to Account B and updates Account B**.** Account A = **\$450**,
Account B = **\$250***,* C_{12} = **\$0***,* C_{21} = **\$0**.

Let us suppose the local state of Account **A** is recorded at the end of step 1 to show \$500 and the local state of Account B and channels C_{12} and C_{21} are recorded at the end of step 3 to show \$150, \$100, and \$50, respectively. Then the recorded global state shows \$800 in the system. An extra of \$100 appears in the system. The reason for the inconsistency is that Account A‘s state was recorded before the \$100 transfer to Account **B** using channel C_{21} was initiated, whereas channel C_{12} 's state was recorded after the \$100 transfer was initiated. **[10]**

The above example shows that recording a consistent global state of a distributed system is not a trivial task.

CHAPTER 5

IMPLEMENTATION OF CHANDY LAMPORT'S ALGORITHM USING MPI

Chapter Gist: This chapter implements the Global snapshot algorithm on the money transaction system using MPI. The chapter also contains appropriate code snippets and observations.

5.1 CHANDY LAMPORT’S GLOBAL SNAPSHOT ALGORITHM IN MONEY TRANSACTION SYSTEM

Let us consider a distributed system consisting of n nodes.
Assumptions*:* ***Recording State*** and ***Sending Marker*** are ***atomic*** operations.

For each processor node, we send a random amount of money from initiator node to all other nodes and deduct initiator processor’s money. At the same time, we keep the record of the incoming channel. We receive the random amount of money from any processor i and add it to current money.

The below code snippet does the money_transaction_system (). Each processor is initialized with $10,000.

```
for (int i=0;i<size;i++) {
            if(rank==i)
                  continue;
        int d=rand()%10+1;
        MPI_Send(&d,1,MPI_INT,i,0,MPI_COMM_WORLD);
        data-=d;
         g_node.channel_record[i]=data;
        g_node.channel_trans[i]=d;
        MPI_Recv(&d,1,MPI_INT,i,0,MPI_COMM_WORLD,&status);
        data+=d;
}
```

5.2 IMPLEMENTATION USING MPI

The following code snippet ensures the initialization of state, channel buffers and status of the channels.

```
//initialize the channel buffer, state and status of channel
for(int i=0;i<size;i++) {
        g_node.channel_status[i]=RED;
        g_node.channel_record[i]=null;
}
g_node.state=data;
srand(time(NULL));
```

When user gives input ‘m’ and initiator node, the initiator node triggers the marker and the global snapshot begins. The money_system_transaction runs in background. The following code snippet performs the global snapshot algorithm for any processor.

```
while (true) {
        perform_money_system_transaction();     //running indefinitely
        scanf("%c %d",&MARKER,&initiator);
        if(initiator > size-1) {
                        printf("Enter a valid node!\n");
                        MPI_Abort(MPI_COMM_WORLD,1);
        }
        //SENDING MARKER
        for(int i=0;i<size;i++) {
                if(rank==i)
                        continue;
                else if(rank==initiator) {
                //start recording on incoming channels  and make incoming channels wide open
                        for(int j=1;j<size;j++)
                                g_node.channel_status[j]=RED;
                }
```

The following code snippet sends the **MARKER** using MPI_Isend() and then records its own state.

```
                    //send markers to other processors
          ierr=MPI_Isend(&MARKER,1,MPI_CHAR,i,0,MPI_COMM_WORLD,&send_request);
                    g_node.state=data;    //record own state
                    ierr=MPI_Wait(&send_request,&status);
     }
```

The following code snippet receives the MARKER using MPI_Irecv(). If **MARKER** is received for the first time:

```
          //RECEIVING MARKER
          for(int i=0;i<size;i++) {
                    if(rank==i)
                              continue;

                    //received marker for the first time


ierr=MPI_Irecv(&MARKER,1,MPI_CHAR,i,0,MPI_COMM_WORLD,&recv_request);
     ierr=MPI_Wait(&recv_request,&status);
                    //if marker is received for the first time, record own state
                    if(g_node.channel_status[i]==RED) {
                              g_node.state=data;         //record state
                              g_node.channel_record[rank]=0;   //intiate NULL
                              g_node.channel_status[i]=GREEN;  //close the channel
                    }
```

The code snippet below ensures that if MARKER was received earlier, then the following cases are considered:

i) If it is an initiator processor
ii) Else, other processor

```
                    else { //if received earlier

                              if(rank==initiator) {
                                        if(g_node.channel_status[i]==GREEN)
                                                  ;
                                        //check if all the incoming channels are recorded GREEN
                                        for(int j=0;j<size;j++) {
                                                  if(j==initiator)
                                                            continue;
                                                  if(g_node.channel_status[j]==RED) {
                                                            g_node.allgreen=0;
                                                            break;
                                                  }
```

```
                                    }
```

The code snippet below ensures if status of channels are GREEN for the initiator, then the loop is broken.

```
                                    //if all are green we are done
                                    if(g_node.allgreen)
                                            break;
                                    else {
                                            g_node.allgreen=true;
                                            break;
                                    }
                            }
                            else {
                                    //for non-initiator processes check if all are green
                                    for(int j=0;j<size;j++) {
                                            if(rank==j)
                                                    continue;
                                            if(g_node.channel_status[j]==RED) {
                                                    g_node.allgreen=0;
                                                    break;
                                            }
                                    }
                                    if(!g_node.allgreen)
                                            continue;
                                    else {
                                            g_node.allgreen=true;
                                            break;
                                    }
                            }
                    }

            }
```

The code below sends and receives channel states, records and ranks of the processors such that we get the entire snapshot in proper order and hence prints the sum of the all the channels with their states.

```
            sum=0;
            if(rank!=0) {
                    //calculate sum
                    for(int i=0;i<size;i++) {
                            sum+=g_node.channel_trans[i];
                    }
```

The code below sends an array which contains the channel state, channel records and total sum of a particular processor to processor 0.

```
                    for(int i=0;i<size;i++)
                            g_node.state-=(g_node.channel_trans[i]);
                    sum+=g_node.state;

                    array[0]=g_node.state;
                    for(int i=1;i<(size+2-1);i++)
                            array[i]=g_node.channel_trans[i-1];
                    array[size+2-1]=sum;
                    MPI_Send(&array,size+2,MPI_INT,0,0,MPI_COMM_WORLD);

            }
```

The code below ensures that processor 0 has received all the information and calculates the desired result.

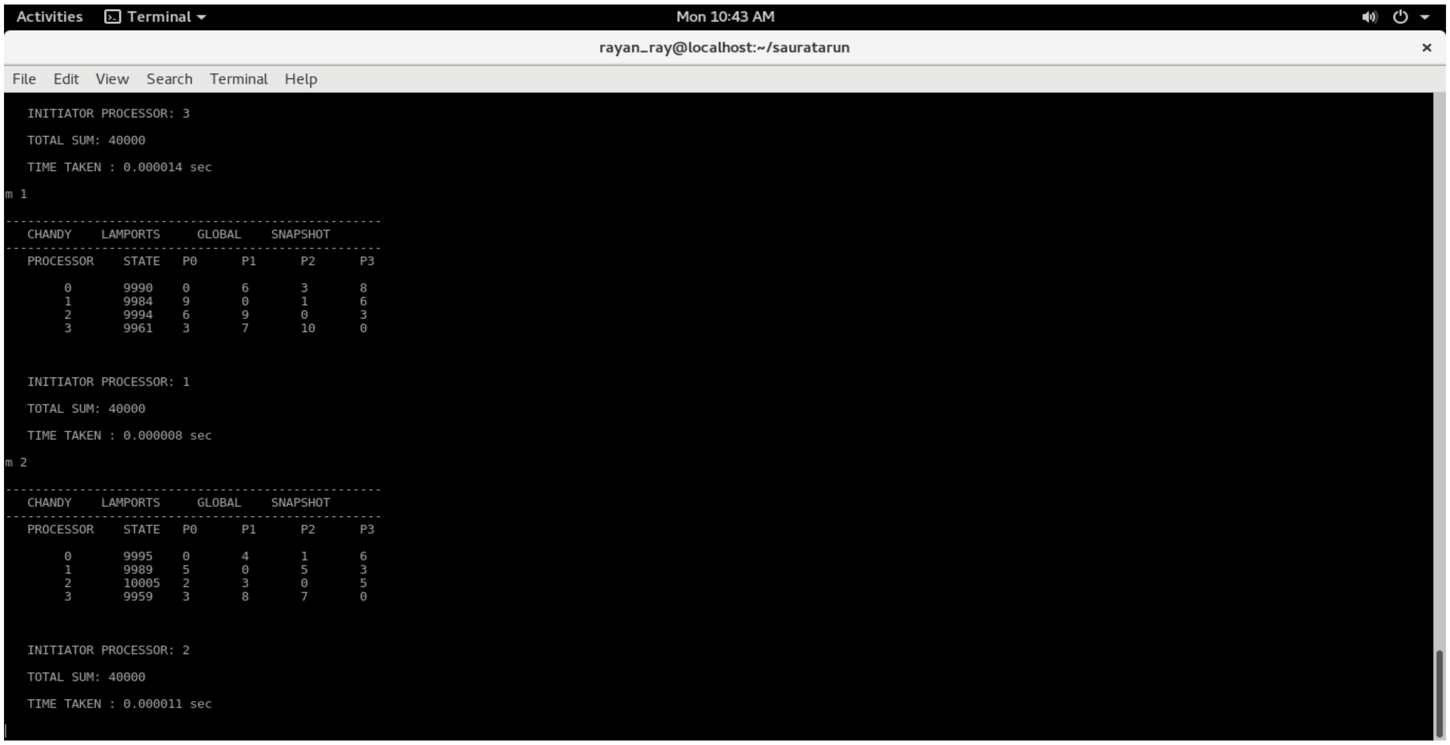
```
                    if(rank==0) {

                            for(int i=0;i<size;i++) {
                                    sum+=g_node.channel_trans[i];
                            }

                            for(int i=0;i<size;i++)
                                    g_node.state-=(g_node.channel_trans[i]);
                            sum+=g_node.state;

                            printf("\n--------------------------------------------------");
                            printf("\n  CHANDY   LAMPORTS   GLOBAL   SNAPSHOT");
                            printf("\n--------------------------------------------------\n");

                            printf("  PROCESSOR   STATE  ");
                            for(int i=0;i<size;i++)
                                    printf("\tP%d ",i);
                            printf("\n\n");
                            printf("\t%d",rank);

                            //store own info
                            printf("\t%d",g_node.state);
                            for(int i=1;i<(size+2-1);i++) {
                                    printf("\t%d",g_node.channel_trans[i-1]);
```

```
				}
				printf("\n");
```

```
			int f=0;
			f+=sum;
			int i;

			for(i=1;i<size;i++) {
				MPI_Recv(&array,size+2,MPI_INT,i,0,MPI_COMM_WORLD,&status);
				printf("\t%d",i);
				for(int j=1;j<(size+2);j++)
				{
					//info[i][j-1]=tarray[j-1];
					printf("\t%d",array[j-1]);
				}

				f+=array[size+2-1];
				if(i==size-1) {
					printf("\n\n\n\n   INITIATOR PROCESSOR: %d",initiator);
					printf("\n\n   TOTAL SUM: %d",f);
					totaltime = ((double) (final_time - initial_time));
					printf("\n\n   TIME TAKEN : %f sec\n\n",totaltime);
				}
				else
					printf("\n");
			}

		}

		scanf("%c %d",&MARKER,&initiator);
		if(initiator > size-1)
			MPI_Abort(MPI_COMM_WORLD,0);
	}
	printf("\n");
	MPI_Finalize();
```

5.3 OBSERVATIONS

When user initiates the MARKER at any given processor, the algorithm starts working and continues its execution until the initiator node receives all the MARKERS along its incoming channel.
The screenshot below shows the output of Chandy Lamport's global snapshot when the initiator nodes are nodes 1 and 2.



```
   INITIATOR PROCESSOR: 3

   TOTAL SUM: 40000

   TIME TAKEN : 0.000014 sec

m 1

---------------------------------------------------
   CHANDY    LAMPORTS     GLOBAL    SNAPSHOT
---------------------------------------------------
   PROCESSOR    STATE   P0      P1      P2      P3

        0       9990    0       6       3       8
        1       9984    9       0       1       6
        2       9994    6       9       0       3
        3       9961    3       7       10      0


   INITIATOR PROCESSOR: 1

   TOTAL SUM: 40000

   TIME TAKEN : 0.000008 sec

m 2

---------------------------------------------------
   CHANDY    LAMPORTS     GLOBAL    SNAPSHOT
---------------------------------------------------
   PROCESSOR    STATE   P0      P1      P2      P3

        0       9995    0       4       1       6
        1       9989    5       0       5       3
        2       10005   2       3       0       5
        3       9959    3       8       7       0


   INITIATOR PROCESSOR: 2

   TOTAL SUM: 40000

   TIME TAKEN : 0.000011 sec

```

From the above output, we can see that, each processor has the following records:
- PROCESSOR :
- STATE
- CHANNEL RECORDS

Finally, each state, their corresponding channel records with the overall total sum are shown. The overall total sum was always consistent.

CHAPTER 6

CONCLUSION

Chapter Gist: This chapter concludes the thesis and discusses about further work which can be done.

6.1 CONCLUSION

We have developed a money transaction system which runs indefinitely and whose global snapshots are taken. The basic idea is to use two colors viz. **RED** and **GREEN** that indicate whether a process has already taken its local snapshot and whether a message was sent before or after the local snapshot of a process. Thus, messages, which would make a snapshot inconsistent, can easily be recognized and avoided, and the receiving process can catch messages, which are in transit. We have observed that, the total amount of money was always consistent.

6.2 FURTHER WORK

There are several variants of the Chandy Lamport's snapshot algorithm. These variants refine and optimize the basic algorithm. For example, **Spezialetti** and **Kearns** algorithm **[12]** optimizes concurrent initiation of snapshot collection and efficiently distributes the recorded snapshot. **Venkatesan's** algorithm **[11]** optimizes the basic snapshot algorithm to efficiently record repeated snapshots of a distributed system that are required in recovery algorithms with synchronous check pointing. More distributed systems like the money transaction system proposed in this project can be developed to check these algorithms.

References

[1] Gerstel **O,** Hurfin **M,** Plouzeau **N,** Raynal **M** and Zaks **S** On-the-fly replay: **A** practical paradigm and its implementation for distributed debugging ***Proc. 6th IEEE Int. Symp.*** *on* ***Parallel and Distributed Debugging (Dallas, TX, Oct.*** 1995) pp 266-272

[2] Hurfin M, Plouzeau N and Raynal M. “A debugging tool for distributed Estelle programs” pp 328-33

[3] Spezialetti M and Kearns P “Simultaneous regions: a framework for the consistent monitoring of distributed systems” ***Proc. 9th Int. Conf. on Distributed Computing Systems,*** 1989, pp. 61-80

[4] Miller B and Choi **J** “Breakpoints and halting in distributed programs” ***Proc.*** **8th Int.** ***Conf. on Distributed Computing Systems,*** 1988, pp. 316-323

[5] D. E. Berthold “A component architecture for high-performance scientific computing”, Intl. J. High-Performance Computing Applications, 2004

[6] ONLINE RESOURCE: MPI-1 standard, http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[7] Birman K and Joseph T “Reliable communication in presence of failures” ***ACM Trans. Computer Systems 3,*** 1987, pp. 47-76

[8] Babaoglu **O** and Marzullo K Consistent global states of distributed systems: fundamental concepts and mechanisms ***Distributed Systems S*** J Mullender (ACM Press) Ch 4, 1993

[9] Acharya **A** and Badrinath B R “Recording distributed snapshots based **on** causal order of message delivery ***Information Processing Lett***”***,*** 1992, pp 317-21

[10] Chandy K M and Lamport L “Distributed snapshots: determining global states of distributed systems” ***ACM Trans. Computer Systems 3,*** 1985, pp. 63-75

[11] Venkatesan S Message-optimal incremental snapshots ***J. Computer Software Engineering*** **1,** 1993, pp. 211-231

[12] Spezialetti M and Kearns P Efficient distributed snapshots ***Proc.*** 6th ***Int. Conf. on Distributed Computing Systems,*** 1986, pp. 382

APPENDICES

APPENDIX A

System Description:

1. Hardware Configuration

Processor: Intel® Core i5-6500 CPU @3.20 GHz
Number of CPU Cores: 4 cores
Total Memory Space (RAM): 4 GB

2. Operating System

Fedora 24 (64 bit)

3. Software Configuration

1) Programming Language Used: C
2) gcc compiler version: gcc 6.1.1
3) openmpi compiler version: 2.0.1

N.B.:- All the machines have identical configurations.

APPENDIX B

Source File Description

This project has one source file called “GSRA.c” which implements the Chandy Lamport’s global snapshot algorithm using MPI.