Netaji Subhas University of Technology

A STATE UNIVERSITY UNDER DELHI ACT 06 OF 2018, GOVT. OF NCT OF DELHI

Azad Hind Fauj Marg, Sector-3, Dwarka, New Delhi-110078

LABORATORY FILE

High Performance Computing

(COCSC18)

Submitted By:

Name: **Shobhit**

Roll Number: **2021UCS1618**

Branch: **CSE-3**

Index

PRACTICAL 1

AIM : Write a program in C to multiply two matrices of size 10000 x 10000 each and find it's execution-time using "time" command. Try to run this program on two or more machines having different configurations and compare execution-times obtained in each run. Comment on which factors affect the performance of the program

CODE :

```
#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>


#define SIZE 10

#define FROM_MASTER 1

#define FROM_WORKER 2

#define DEBUG 1


MPI_Status status;


static double a[SIZE][SIZE];

static double b[SIZE][SIZE];

static double c[SIZE][SIZE];

static void init_matrix(void)

{

   int i, j;

   for (i = 0; i < SIZE; i++)

   {

      for (j = 0; j < SIZE; j++) {

            a[i][j] = 1;

            b[i][j] = 1;

     } //end for i

   }  //end for j

} //end init_matrix()


static void print_matrix(void)

{

int i, j;

for(i = 0; i < SIZE; i++) {

   for(j = 0; j < SIZE; j++) {

      printf("%7.2f", c[i][j]);

   } //end for i

printf("\n");

  }    //end for j
```

```
  }       //end print_matrix

int main(int argc, char **argv)
{
int myrank, nproc;
int rows;
int mtype;
int dest, src, offseta, offsetb;
double start_time, end_time;
int i, j, k, l;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

rows = SIZE/nproc;  //compute the block size
mtype = FROM_MASTER; //  =1

   if (myrank == 0) {
    /*Initialization*/
    printf("SIZE = %d, number of nodes = %d\n", SIZE, nproc);
    init_matrix();

    start_time = MPI_Wtime();

    if(nproc == 1) {
        for(i = 0; i < SIZE; i++) {
            for(j = 0; j < SIZE; j++) {
                for(k = 0; k < SIZE; k++)
                    c[i][j] = c[i][j] + a[i][k]*b[j][k];
            } //end for i
        }  //end for j
        end_time = MPI_Wtime();
        print_matrix();//--------------------------------
        printf("Execution time on %2d nodes: %f\n", nproc, end_time-
                    start_time);
    } // end  if(nproc == 1)

    else {
```

```
    for(l = 0; l < nproc; l++){
      offsetb = rows*l;  //start from (block size * processor id)
      offseta = rows;
      mtype = FROM_MASTER; // tag =1

        for(dest = 1; dest < nproc; dest++){
          MPI_Send(&offseta, 1, MPI_INT, dest, mtype,
                    MPI_COMM_WORLD);
          MPI_Send(&offsetb, 1, MPI_INT, dest, mtype,
                    MPI_COMM_WORLD);
          MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
          MPI_Send(&a[offseta][0], rows*SIZE, MPI_DOUBLE, dest,
                     mtype, MPI_COMM_WORLD);
          MPI_Send(&b[0][offsetb], rows*SIZE, MPI_DOUBLE, dest,
                   mtype, MPI_COMM_WORLD);

          offseta += rows;
          offsetb = (offsetb+rows)%SIZE;

      } // end for dest

      offseta = rows;
      offsetb = rows*l;

//--mult the final local and print final global mult
      for(i = 0; i < offseta; i++) {
          for(j = offsetb; j < offsetb+rows; j++) {
                for(k = 0; k < SIZE; k++){
                    c[i][j] = c[i][j] + a[i][k]*b[k][j];
              }//end for k
          } //end for j
      }// end for i
        /*- wait for results from all worker tasks */
      mtype = FROM_WORKER;
      for(src = 1; src < nproc; src++){
          MPI_Recv(&offseta, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
                   &status);
          MPI_Recv(&offsetb, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
                    &status);
          MPI_Recv(&rows, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
```

```
                &status);
            for(i = 0; i < rows; i++) {
                MPI_Recv(&c[offseta+i][offsetb], offseta, MPI_DOUBLE,
        src, mtype, MPI_COMM_WORLD, &status);
            } //end for scr
        }//end for i
    } //end for l
    end_time = MPI_Wtime();
    print_matrix();
    printf("Execution time on %2d nodes: %f\n", nproc, end_time-
        start_time);
   }//end else
} //end if (myrank == 0)

else{
      /*---------------------------- worker----------------------*/
  if(nproc > 1) {
        for(l = 0; l < nproc; l++){
          mtype = FROM_MASTER;
          MPI_Recv(&offseta, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
                &status);
          MPI_Recv(&offsetb, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
                &status);
        MPI_Recv(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
                &status);

        MPI_Recv(&a[offseta][0], rows*SIZE, MPI_DOUBLE, 0, mtype,
        MPI_COMM_WORLD, &status);
        MPI_Recv(&b[0][offsetb], rows*SIZE, MPI_DOUBLE, 0, mtype,
        MPI_COMM_WORLD, &status);

        for(i = offseta; i < offseta+rows; i++) {
            for(j = offsetb; j < offsetb+rows; j++) {
                for(k = 0; k < SIZE; k++){
                    c[i][j] = c[i][j] + a[i][k]*b[k][j];
                } //end for j
            } //end for i

        } //end for l
```

```
        mtype = FROM_WORKER;
        MPI_Send(&offseta, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
        MPI_Send(&offsetb, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
          for(i = 0; i < rows; i++){
          MPI_Send(&c[offseta+i][offsetb], offseta, MPI_DOUBLE,   0,
           mtype, MPI_COMM_WORLD);


         } //end for i
        }//end for l
      } //end if (nproc > 1)
    } // end else
    MPI_Finalize();
    return 0;
} //end main()
```

OUTPUT :

```
~/Finalmpi$ cd 1matrix/
~/Finalmpi/1matrix$ mpicc -o matrixMult matrixMult.c
~/Finalmpi/1matrix$ mpirun -np 4 ./matrixMult
SIZE = 10, number of nodes = 4
     10     10     10     10     10     10     10     10      0      0
     10     10     10     10     10     10     10     10      0      0
      2      2      2      2      2      2      2      2      0      0
      2      2      2      2      2      2      2      2      0      0
      0      0      2      2      2      2      2      2      2      2
      0      0      2      2      2      2      2      2      2      2
      3      3      0      0      2      2      2      2      2      2
      3      3      0      0      2      2      2      2      2      2
      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0
Execution time on  4 nodes: 0.004758
~/Finalmpi/1matrix$
```

CONCLUSION: Matrix multiplication using MPI has been successfully implemented.

PRACTICAL 2

AIM - Write a parallel program to print “Hello World" using MPI

CODE -

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

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

  MPI_Init(&argc, &argv);              // Initialize MPI environment
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get rank of the current process
  MPI_Comm_size(MPI_COMM_WORLD, &size); // Get total number of processes

  printf("Hello World from process %d of %d\n", rank, size);

  MPI_Finalize(); // Finalize MPI environment
  return 0;
}
```

OUTPUT -

```
~/Finalmpi/7processSync$ cd ../2hello/
~/Finalmpi/2hello$ ls
hello  hello.c
~/Finalmpi/2hello$ mpicc -o hello hello.c
~/Finalmpi/2hello$ mpirun -np 5 ./hello
Hello World from process 0 of 5
Hello World from process 1 of 5
Hello World from process 3 of 5
Hello World from process 2 of 5
Hello World from process 4 of 5
~/Finalmpi/2hello$
```

CONCLUSION: “Hello World” program using MPI has been successfully implemented.

PRACTICAL 3

AIM - Write a parallel program to find sum of an array using MPI

CODE -

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

// size of array
#define n 10

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Temporary array for slave process
int a2[1000];

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

  int pid, np,
    elements_per_process,
    n_elements_recieved;
  // np -> no. of processes
  // pid -> process id

  MPI_Status status;

  // Creation of parallel processes
  MPI_Init(&argc, &argv);

  // find out process ID,
  // and how many processes were started
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  // master process
  if (pid == 0) {
    int index, i;
    elements_per_process = n / np;
```

```
// check if more than 1 processes are run
if (np > 1) {
  // distributes the portion of array
  // to child processes to calculate
  // their partial sums
  for (i = 1; i < np - 1; i++) {
    index = i * elements_per_process;

    MPI_Send(&elements_per_process,
        1, MPI_INT, i, 0,
        MPI_COMM_WORLD);
    MPI_Send(&a[index],
        elements_per_process,
        MPI_INT, i, 0,
        MPI_COMM_WORLD);
  }

  // last process adds remaining elements
  index = i * elements_per_process;
  int elements_left = n - index;

// master process add its own sub array
int sum = 0;
for (i = 0; i < elements_per_process; i++)
  sum += a[i];

// collects partial sums from other processes
int tmp;
for (i = 1; i < np; i++) {
  MPI_Recv(&tmp, 1, MPI_INT,
      MPI_ANY_SOURCE, 0,
      MPI_COMM_WORLD,
      &status);
  int sender = status.MPI_SOURCE;

  sum += tmp;
}

// prints the final sum of array
printf("Sum of array is : %d\n", sum);
```

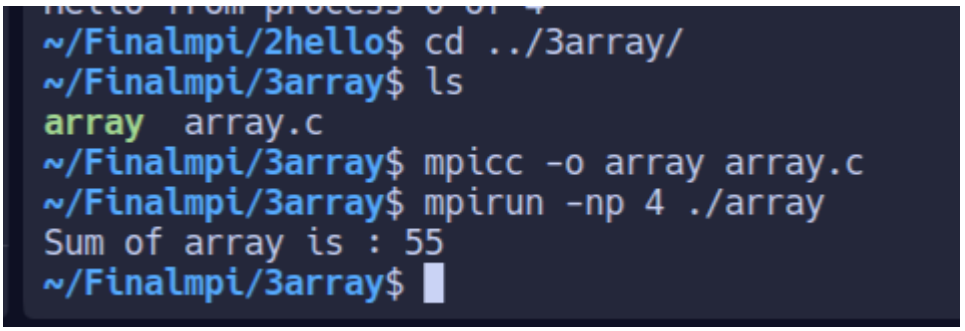
```
  }
  // slave processes
  else {
    MPI_Recv(&n_elements_recieved,
        1, MPI_INT, 0, 0,
        MPI_COMM_WORLD,
        &status);

    // stores the received array segment
    // in local array a2
    MPI_Recv(&a2, n_elements_recieved,
        MPI_INT, 0, 0,
        MPI_COMM_WORLD,
        &status);

    // calculates its partial sum
    int partial_sum = 0;
    for (int i = 0; i < n_elements_recieved; i++)
      partial_sum += a2[i];

    // sends the partial sum to the root process
    MPI_Send(&partial_sum, 1, MPI_INT,
        0, 0, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

OUTPUT -



CONCLUSION: Sum of array has been calculated using MPI has been successfully implemented.

PRACTICAL 4

AIM - Write a C program for parallel implementation of Matrix Multiplication using MPI

CODE -

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

#define SIZE 10
#define FROM_MASTER 1
#define FROM_WORKER 2
#define DEBUG 1

MPI_Status status;

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];
static void init_matrix(void)
{
   int i, j;
   for (i = 0; i < SIZE; i++)
   {
       for (j = 0; j < SIZE; j++) {
            a[i][j] = 1;
            b[i][j] = 1;
      } //end for i
   }  //end for j
} //end init_matrix()

static void print_matrix(void)
{
int i, j;
for(i = 0; i < SIZE; i++) {
   for(j = 0; j < SIZE; j++) {
       printf("%7.2f", c[i][j]);
   } //end for i
printf("\n");
  }    //end for j
 }       //end print_matrix
```

```
int main(int argc, char **argv)
{
int myrank, nproc;
int rows;
int mtype;
int dest, src, offseta, offsetb;
double start_time, end_time;
int i, j, k, l;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

rows = SIZE/nproc;  //compute the block size
mtype = FROM_MASTER; //  =1

  if (myrank == 0) {
   /*Initialization*/
   printf("SIZE = %d, number of nodes = %d\n", SIZE, nproc);
   init_matrix();

   start_time = MPI_Wtime();

   if(nproc == 1) {
      for(i = 0; i < SIZE; i++) {
         for(j = 0; j < SIZE; j++) {
            for(k = 0; k < SIZE; k++)
               c[i][j] = c[i][j] + a[i][k]*b[j][k];
         } //end for i
      }  //end for j
      end_time = MPI_Wtime();
      print_matrix();//--------------------------------
      printf("Execution time on %2d nodes: %f\n", nproc, end_time-
               start_time);
   } // end  if(nproc == 1)

   else {

        for(l = 0; l < nproc; l++){
          offsetb = rows*l;  //start from (block size * processor id)
```

```
offseta = rows;
mtype = FROM_MASTER; // tag =1

  for(dest = 1; dest < nproc; dest++){
    MPI_Send(&offseta, 1, MPI_INT, dest, mtype,
              MPI_COMM_WORLD);
    MPI_Send(&offsetb, 1, MPI_INT, dest, mtype,
              MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&a[offseta][0], rows*SIZE, MPI_DOUBLE, dest,
               mtype, MPI_COMM_WORLD);
    MPI_Send(&b[0][offsetb], rows*SIZE, MPI_DOUBLE, dest,
             mtype, MPI_COMM_WORLD);

    offseta += rows;
    offsetb = (offsetb+rows)%SIZE;

  } // end for dest

  offseta = rows;
  offsetb = rows*I;

//--mult the final local and print final global mult
  for(i = 0; i < offseta; i++) {
    for(j = offsetb; j < offsetb+rows; j++) {
          for(k = 0; k < SIZE; k++){
              c[i][j] = c[i][j] + a[i][k]*b[k][j];
        }//end for k
    } //end for j
  }// end for i
    /*- wait for results from all worker tasks */
  mtype = FROM_WORKER;
  for(src = 1; src < nproc; src++){
    MPI_Recv(&offseta, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
             &status);
    MPI_Recv(&offsetb, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
              &status);
    MPI_Recv(&rows, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
             &status);
    for(i = 0; i < rows; i++) {
```

```
            MPI_Recv(&c[offseta+i][offsetb], offseta, MPI_DOUBLE,
        src, mtype, MPI_COMM_WORLD, &status);
            } //end for scr
        }//end for i
    } //end for l
    end_time = MPI_Wtime();
    print_matrix();
    printf("Execution time on %2d nodes: %f\n", nproc, end_time-
        start_time);
   }//end else
} //end if (myrank == 0)

else{
      /*---------------------------- worker----------------------*/
  if(nproc > 1) {
        for(l = 0; l < nproc; l++){
          mtype = FROM_MASTER;
          MPI_Recv(&offseta, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
                &status);
          MPI_Recv(&offsetb, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
                &status);
        MPI_Recv(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
               &status);

        MPI_Recv(&a[offseta][0], rows*SIZE, MPI_DOUBLE, 0, mtype,
        MPI_COMM_WORLD, &status);
        MPI_Recv(&b[0][offsetb], rows*SIZE, MPI_DOUBLE, 0, mtype,
        MPI_COMM_WORLD, &status);

        for(i = offseta; i < offseta+rows; i++) {
            for(j = offsetb; j < offsetb+rows; j++) {
                for(k = 0; k < SIZE; k++){
                    c[i][j] = c[i][j] + a[i][k]*b[k][j];
                } //end for j
            } //end for i

        } //end for l

        mtype = FROM_WORKER;
        MPI_Send(&offseta, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
```

```
        MPI_Send(&offsetb, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
          for(i = 0; i < rows; i++){
          MPI_Send(&c[offseta+i][offsetb], offseta, MPI_DOUBLE,   0,
           mtype, MPI_COMM_WORLD);


         } //end for i
        }//end for l
      } //end if (nproc > 1)
    } // end else
    MPI_Finalize();
    return 0;
} //end main()
```

OUTPUT -

```
~/Finalmpi/4matrix$ mpicc -o matrix matrix.c
~/Finalmpi/4matrix$ mpirun -np 4 ./matrix
SIZE = 10, number of nodes = 4
  10.00  10.00  10.00  10.00  10.00  10.00  10.00  10.00   0.00   0.00
  10.00  10.00  10.00  10.00  10.00  10.00  10.00  10.00   0.00   0.00
   2.00   2.00   2.00   2.00   2.00   2.00   2.00   2.00   0.00   0.00
   2.00   2.00   2.00   2.00   2.00   2.00   2.00   2.00   0.00   0.00
   0.00   0.00   2.00   2.00   2.00   2.00   2.00   2.00   2.00   2.00
   0.00   0.00   2.00   2.00   2.00   2.00   2.00   2.00   2.00   2.00
   3.00   3.00   0.00   0.00   2.00   2.00   2.00   2.00   2.00   2.00
   3.00   3.00   0.00   0.00   2.00   2.00   2.00   2.00   2.00   2.00
   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
Execution time on  4 nodes: 0.001753
```

CONCLUSION: Matrix multiplication using MPI has been successfully implemented.

PRACTICAL 5

AIM - Write a C program to implement the Quick Sort Algorithm using MPI.

CODE -

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

void swap(int* arr, int i, int j)
{
  int t = arr[i];
  arr[i] = arr[j];
  arr[j] = t;
}
void quicksort(int* arr, int start, int end)
{
  int pivot, index;

  // Base Case
  if (end <= 1)
    return;
  pivot = arr[start + end / 2];
  swap(arr, start, start + end / 2);
  index = start;

  for (int i = start + 1; i < start + end; i++) {

    // Swap if the element is less
    // than the pivot element
    if (arr[i] < pivot) {
      index++;
      swap(arr, i, index);
    }
  }

  // Swap the pivot into place
  swap(arr, start, index);

  // Recursive Call for sorting
```

```
  // of quick sort function
  quicksort(arr, start, index - start);
  quicksort(arr, index + 1, start + end - index - 1);
}

// Function that merges the two arrays
int* merge(int* arr1, int n1, int* arr2, int n2)
{
  int* result = (int*)malloc((n1 + n2) * sizeof(int));
  int i = 0;
  int j = 0;
  int k;

  for (k = 0; k < n1 + n2; k++) {
    if (i >= n1) {
      result[k] = arr2[j];
      j++;
    }
    else if (j >= n2) {
      result[k] = arr1[i];
      i++;
    }

    // Indices in bounds as i < n1
    // && j < n2
    else if (arr1[i] < arr2[j]) {
      result[k] = arr1[i];
      i++;
    }

    // v2[j] <= v1[i]
    else {
      result[k] = arr2[j];
      j++;
    }
  }
  return result;
}
// Driver Code
int main(int argc, char* argv[])
```

```
{
  int number_of_elements;
  int* data = NULL;
  int chunk_size, own_chunk_size;
  int* chunk;
  FILE* file = NULL;
  double time_taken;
  MPI_Status status;

  if (argc != 3) {
    printf("Desired number of arguments are not their "
      "in argv....\n");
    printf("2 files required first one input and "
      "second one output....\n");
    exit(-1);
  }


    // Printing total number of elements
    // in the file
    fprintf(
      file,
      "Total number of Elements in the array : %d\n",
      own_chunk_size);

    // Printing the value of array in the file
    for (int i = 0; i < own_chunk_size; i++) {
      fprintf(file, "%d ", chunk[i]);
    }

    // Closing the file
    fclose(file);

    printf("\n\n\n\nResult printed in output.txt file "
      "and shown below: \n");

    // For Printing in the terminal
    printf("Total number of Elements given as input : "
      "%d\n",
      number_of_elements);
```

```
    printf("Sorted array is: \n");

    for (int i = 0; i < number_of_elements; i++) {
      printf("%d ", chunk[i]);
    }

    printf(
      "\n\nQuicksort %d ints on %d procs: %f secs\n",
      number_of_elements, number_of_process,
      time_taken);
  }
  MPI_Finalize();
  return 0;
}
```

OUTPUT -

```
~/Finalmpi/4matrix$ cd ../5quick/
~/Finalmpi/5quick$ ls
input.txt  output.txt  quick  quick.c  to_run.txt
~/Finalmpi/5quick$ mpicc -o quick quick.c
quick.c: In function 'main':
quick.c:136:5: warning: ignoring return value of 'fscanf' declared with
 attribute 'warn_unused_result' [-Wunused-result]
  136 |     fscanf(file, "%d", &number_of_elements);
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
quick.c:154:7: warning: ignoring return value of 'fscanf' declared with
 attribute 'warn_unused_result' [-Wunused-result]
  154 |       fscanf(file, "%d", &data[i]);
      |       ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
~/Finalmpi/5quick$ mpirun -np 1 ./quick input.txt output.txt
Reading number of Elements From file ....
Number of Elements in the file is 9
Reading the array from the file.......
Elements in the array is :
9 6 7 8 4 2 3 1 3




Result printed in output.txt file and shown below:
Total number of Elements given as input : 9
Sorted array is:
1 2 3 3 4 6 7 8 9

Quicksort 9 ints on 1 procs: 0.001357 secs
```

CONCLUSION: Quick Sort using MPI has been successfully implemented.

PRACTICAL 6

AIM - Write a multithreaded program to generate Fibonacci series using pThreads.

CODE -

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

#define MAX_TERM 93
#define int unsigned long long
// Global variables to store Fibonacci terms
int fib[MAX_TERM];

// Mutex to synchronize access to fib array
pthread_mutex_t mutex;

// Function to generate Fibonacci series
void *generate_fibonacci(void *arg) {
    int n = *((int *)arg);
    if (n == 0) {
        fib[0] = 0;
    } else if (n == 1) {
        fib[1] = 1;
    } else {
        fib[0] = 0;
        fib[1] = 1;
        for (int i = 2; i <= n; ++i) {
            pthread_mutex_lock(&mutex);
            fib[i] = fib[i - 1] + fib[i - 2];
            pthread_mutex_unlock(&mutex);
        }
    }
    pthread_exit(NULL);
}
signed main() {
    pthread_t tid;
    int n;

    // Initialize mutex
    pthread_mutex_init(&mutex, NULL);

    printf("Enter the number of terms for Fibonacci series : ");
```

```
    scanf("%lld", &n);

    if (n > MAX_TERM) {
        printf("Number of terms exceeds the limit. Exiting...\n");
        return 1;
    }
    // Create a thread to generate Fibonacci series
    pthread_create(&tid, NULL, generate_fibonacci, (void *)&n);

    // Wait for the thread to finish
    pthread_join(tid, NULL);

    // Print the Fibonacci series
    printf("Fibonacci Series:\n");
    for (int i = 0; i < n; ++i) {
        printf("%lld ", fib[i]);
    }
    printf("\n");
    // Destroy mutex
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

OUTPUT -

```
~/Finalmpi/6fibon$ cd ../6fibon/
~/Finalmpi/6fibon$ ls
fib  fib.c  to_run.txt
~/Finalmpi/6fibon$ mpicc -o fib fib.c
fib.c: In function 'main':
fib.c:39:5: warning: ignoring return value of 'scanf' declared with att
ribute 'warn_unused_result' [-Wunused-result]
   39 |     scanf("%lld", &n);
      |     ^~~~~~~~~~~~~~~~~
~/Finalmpi/6fibon$ mpirun -np 4 ./fib
50
Enter the number of terms for Fibonacci series (maximum 93): Fibonacci
Series:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10
946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269
2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102
334155 165580141 267914296 433494437 701408733 1134903170 1836311903 29
71215073 4807526976 7778742049
```

CONCLUSION: Multithreaded program to generate Fibonacci series using pThreads has been successfully implemented

PRACTICAL 7

AIM - Write a program to implement Process Synchronization by mutex locks using pThreads.

CODE -

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define jobs 10

pthread_t tid[jobs];

void *trythis(void *arg) {
  int counter = *((int *)arg);
  printf("\n Job %d has started\n", counter);

  // Introduce some computation to delay the thread
  for (unsigned long long i = 0; i < 90000000; i++) {
    // Introduce a side effect to prevent optimization
    volatile unsigned long long temp = i * 2; // Example computation
  }

  printf("\n Job %d has finished\n", counter);
  return NULL;
}

int main(void) {
  // Allocate memory for thread arguments
  int *thread_args = malloc(jobs * sizeof(int));
  if (thread_args == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    return 1;
  }

  // Create threads
  for (int i = 0; i < jobs; i++) {
    thread_args[i] = i;
    int error = pthread_create(&(tid[i]), NULL, &trythis, &(thread_args[i]));
```

```
    if (error != 0)
      printf("\nThread can't be created : [%s]", strerror(error));
  }
  // Join threads
  for (int i = 0; i < jobs; i++) {
    pthread_join(tid[i], NULL);
  }
  // Free allocated memory
  free(thread_args);
  return 0;
}
```

OUTPUT -

```
~/Finalmpi/6fibon$ cd ../7processSync/
~/Finalmpi/7processSync$ ls
process  process.c  to_run.txt
~/Finalmpi/7processSync$ ls
process  process.c  to_run.txt
~/Finalmpi/7processSync$ cat to_run.txt
gcc -O3 -flto -o pthread_example pthread_example.c -pthread
./pthread_example
~/Finalmpi/7processSync$ gcc -O3 -flto -o process process.c
~/Finalmpi/7processSync$ ./process

 Job 3 has started

 Job 4 has started

 Job 9 has started

 Job 6 has started

 Job 0 has started

 Job 1 has started

 Job 2 has started

 Job 5 has started

 Job 7 has started

 Job 8 has started

 Job 5 has finished

 Job 1 has finished

 Job 9 has finished

 Job 7 has finished

 Job 8 has finished
```

CONCLUSION: Process Synchronization by mutex locks using pThreads has been successfully implemented.