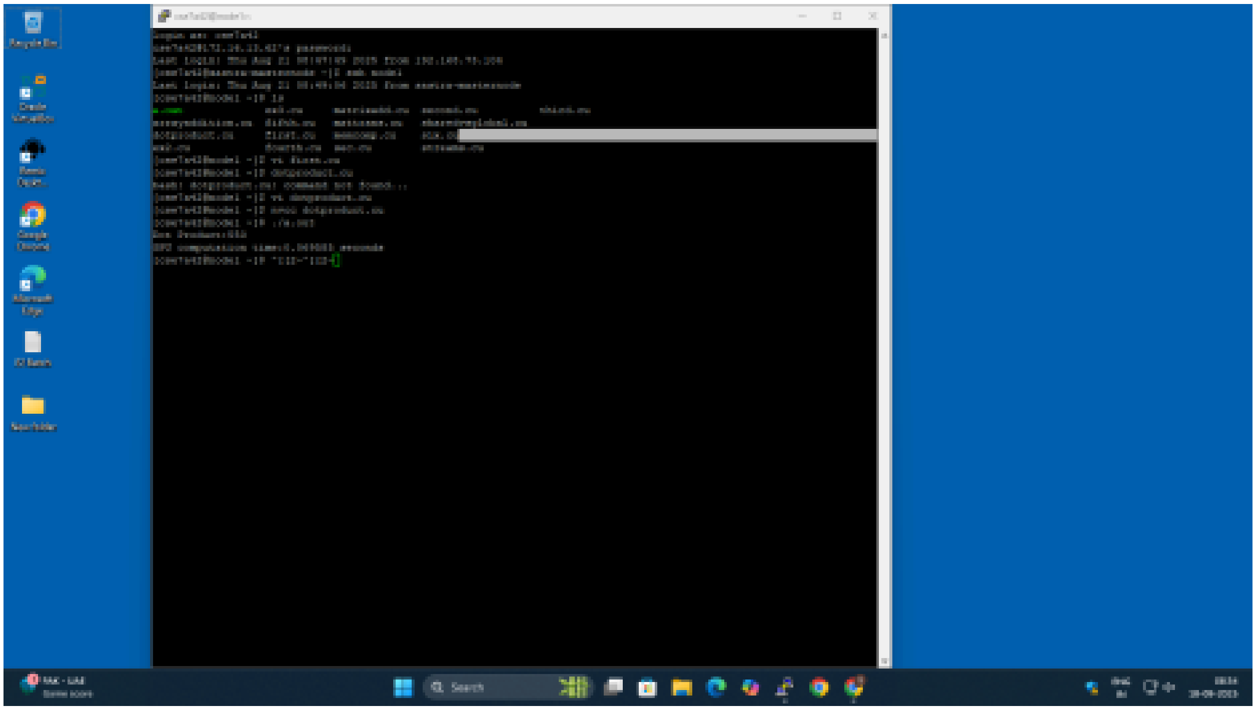
1)COMPUTE DOT PRODUCT OF TWO VECTORS USING PARALLEL REDUCTION

```
#include<stdio.h>

#include<sys/time.h>

double cpusecond() {

struct timeval tp;

gettimeofday(&tp,NULL);

return ((double)tp.tv_sec+(double)tp.tv_usec*1.e-6); }

__global__ void dotproductkernel(int* a,int* b,int* result,int n)

{

int tid=threadIdx.x+blockIdx.x*blockDim.x;

int index=tid;

if(index<n){

a[index]=a[index]*b[index];

}

__syncthreads();

for(int stride=1;stride<n;stride*=2){

if(index%(2*stride)==0 &&(index+stride)<n){

a[index]+=a[index+stride];

}

}

__syncthreads();

if(index==0){

*result=a[0];

}
}
```

```
int main(){
const int N=5;
int h_a[N]={1,2,3,4,5};
int h_b[N]={10,20,30,40,50};
int h_result=0;
int*d_a,*d_b,*d_result;
cudaMalloc((void**)&d_a,N*sizeof(int));
cudaMalloc((void**)&d_b,N*sizeof(int));
cudaMalloc((void**)&d_result,sizeof(int));
cudaMemcpy(d_a,h_a,N*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_b,h_b,N*sizeof(int),cudaMemcpyHostToDevice);
double gpu_start=cpusecond();
int threadsperblock=5;
int blocks=(N+threadsperblock-1)/threadsperblock;
dotproductkernel<<<blocks,threadsperblock>>>(d_a,d_b,d_result,N);
cudaDeviceSynchronize();
double gpu_end=cpusecond();
cudaMemcpy(&h_result,d_result,sizeof(int),cudaMemcpyDeviceToHost);
printf("dot product:%d\n",h_result);
printf("GPU computation time:%f seconds\n",gpu_end-gpu_start);
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_result);
return 0;
```

```
}
```

OUTPUT:



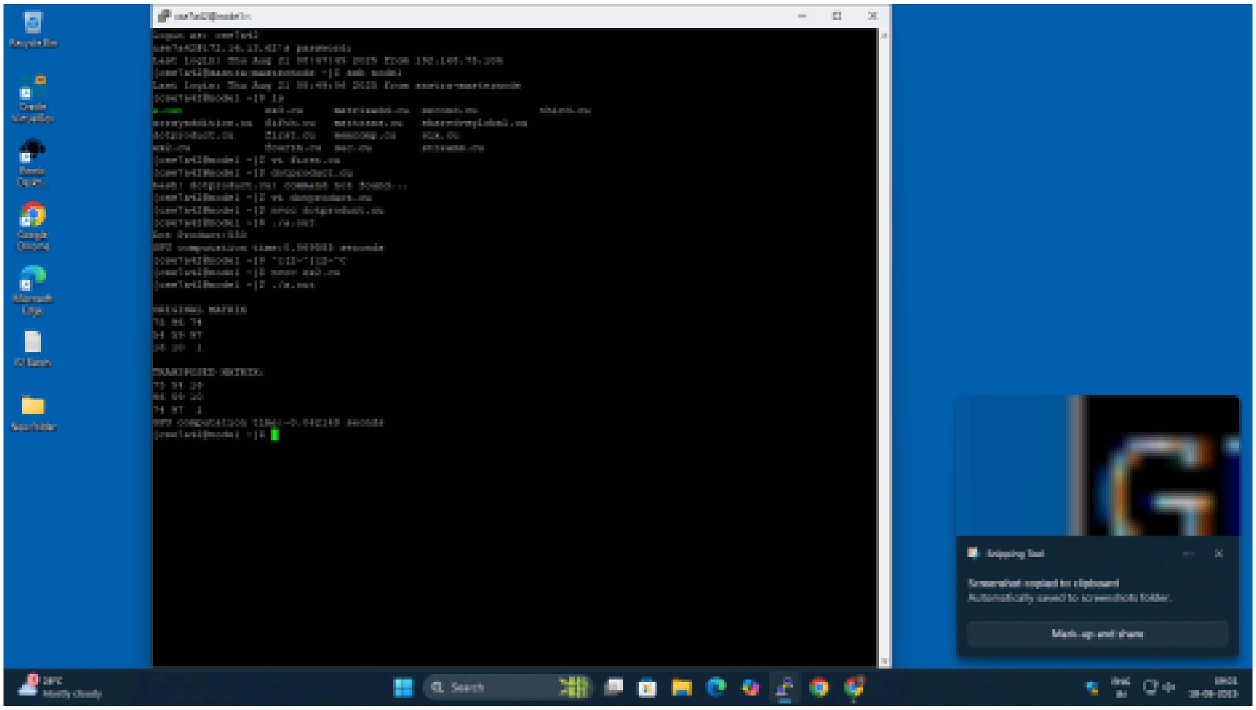
2)Matrix Transpose with Random Number Generation using Shared Memory

```
#include <stdio.h>
#define N 3 // 3x3 matrix

__global__ void transposer(int *input, int *output, int width) {
 __shared__ int tile[N][N];
```

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;

if (idx < width * width) {
int row = idx / width;
int col = idx % width;

tile[col][row]= input[row * width + col];

__syncthreads();



output[idx] = tile[row][col];
}
}
int main() {
int size = N * N * sizeof(int);
int h_input[N * N], h_output[N * N];
srand(time(NULL));
for (int i = 0; i < N * N; i++) {
h_input[i] = rand() % 100;
}

int *d_input, *d_output;
cudaMalloc(&d_input, size);
cudaMalloc(&d_output, size);

cudaMemcpy(d_input, h_input, size, cudaMemcpyHostToDevice);

int threadsPerBlock = 256;
```

```
 int blocksPerGrid = (N * N + threadsPerBlock - 1) / threadsPerBlock;

transposer<<<blocksPerGrid, threadsPerBlock>>>(d_input, d_output, N);

cudaMemcpy(h_output, d_output, size, cudaMemcpyDeviceToHost);

printf("Original Matrix:\n");

 for (int i = 0; i < N; i++) {
 for (int j = 0; j < N; j++) {
 printf("%2d ", h_input[i * N + j]);  }
 printf("\n");
 }

 printf("\nTransposed Matrix:\n");
for (int i = 0; i < N; i++) {
 for (int j = 0; j < N; j++) {
 printf("%2d ", h_output[i * N + j]);  }
 printf("\n"); }

 cudaFree(d_input);
 cudaFree(d_output);
 return 0;
}
```

OUTPUT:



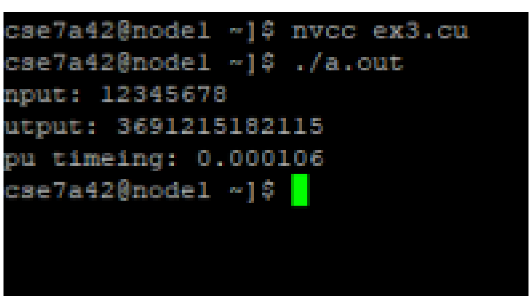
3)1D STENCIL USING CONSTANT MEMORY

```
#include <stdio.h>

#define N 8
#define RADIUS 1

// Constant memory for stencil weights
__constant__ int d_weights[2 * RADIUS + 1];

// Kernel using constant memory
__global__ void stencil1D(int *in, int *out)
 int i = threadIdx.x;

 int result = 0;
```

```
 for (int j = -RADIUS; j <= RADIUS; j++) {
 int idx = i + j;
 // Handle boundary
 if (idx >= 0 && idx < N) {
 result += d_weights[j + RADIUS] * in[idx];
 }
 }

 out[i] = result;
}

int main() {
 int h_in[N] = {1, 2, 3, 4, 5, 6, 7, 8};
 int h_out[N];

 int *d_in, *d_out;

 int h_weights[2 * RADIUS + 1] = {1, 1, 1};



 cudaMemcpyToSymbol(d_weights, h_weights, sizeof(h_weights));

 // Allocate device memory
 cudaMalloc(&d_in, N * sizeof(int));
 cudaMalloc(&d_out, N * sizeof(int));

 // Copy input to device
 cudaMemcpy(d_in, h_in, N * sizeof(int), cudaMemcpyHostToDevice);

 // Launch kernel
```

```
stencil1D<<<1, N>>>(d_in, d_out);

// Copy result back
cudaMemcpy(h_out, d_out, N * sizeof(int), cudaMemcpyDeviceToHost);

// Print result
printf("Input : ");
for (int i = 0; i < N; i++) printf("%d ", h_in[i]);
printf("\nOutput: ");
for (int i = 0; i < N; i++) printf("%d ", h_out[i]);
printf("\n");

// Free memory
cudaFree(d_in);
cudaFree(d_out);

return 0; }
```

OUTPUT:



4)PREORDER TREE TRAVERSAL USING PRAM ALGORITHM

```
#include<stdio.h>

#define N 9
```

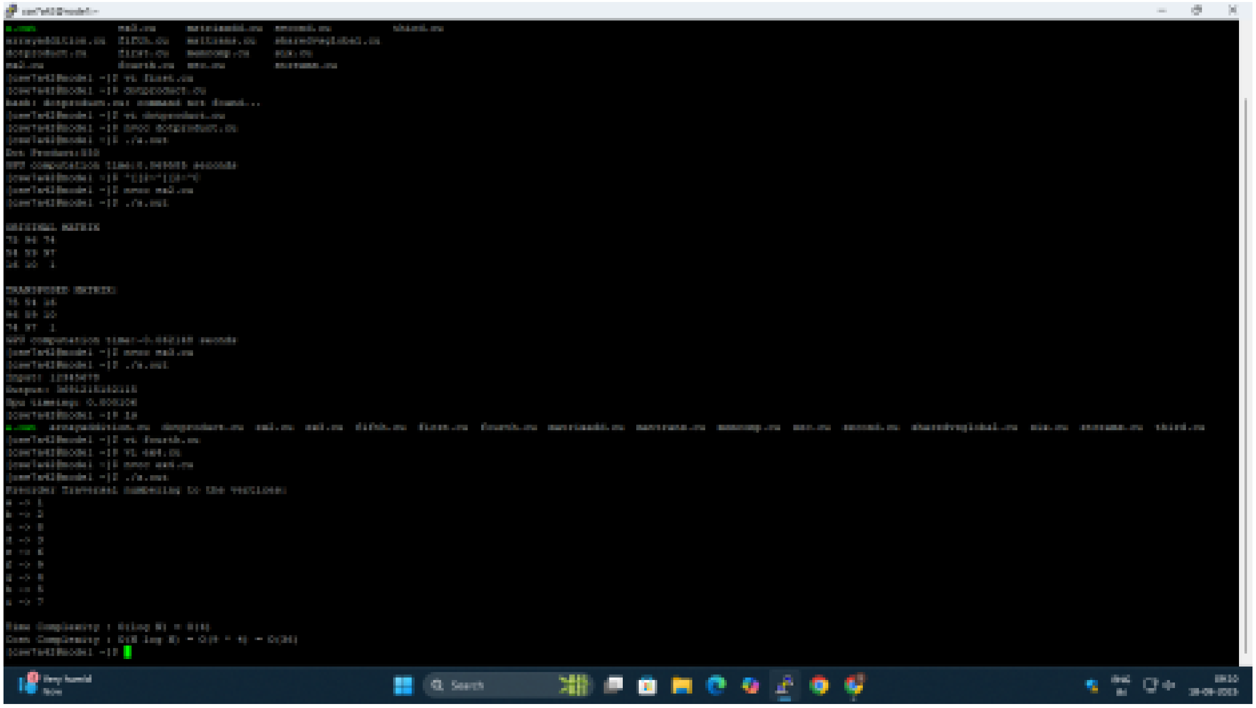
```
__device__ int temp[9][9];
__global__ void traverse(int *parent,int *child,int *sibling,int *edge0
,int *edge1,int *succ0,int *succ1,int *position,int *preorder) {
int i=threadIdx.x;
if(parent[edge0[i]]==edge1[i]){
 if(sibling[edge0[i]]!=-1){
 succ0[i]=edge1[i];
 succ1[i]=sibling[edge0[i]];
 }
 else if(parent[edge1[i]]!=-1){
 succ0[i]=edge1[i];
 succ1[i]=parent[edge1[i]];
 }
 else{
 succ0[i]=edge0[i];
 succ1[i]=edge1[i];
 preorder[edge1[i]]=1;
}
}
else{
if(child[edge1[i]]!=-1){
succ0[i]=edge1[i];
 succ1[i]=child[edge1[i]];
 }
 else{
 succ0[i]=edge1[i];
 succ1[i]=edge0[i];
 }
}
```

```
if(parent[edge0[i]]==edge1[i]){
 position[i]=0;
}
else{
 position[i]=1;
}
int x;
for(int k=0;k<4;k++){
 x=temp[succ0[i]][succ1[i]];
 position[i]=position[i]+position[x];
 succ0[i]=succ0[x];
 succ1[i]=succ1[x];
}
if(edge0[i]==parent[edge1[i]]){
 preorder[edge1[i]]=9+1-position[i];
}
}
__global__ void initialize(int *edge0,int *edge1){
 for(int i=0;i<16;i++){
 temp[edge0[i]][edge1[i]]=i;
 }
}
int main()
{
char vertices[9]={'a','b','c','d','e','f','g','h','i'};
int parent[9]={-1,0,0,1,1,2,3,3,4};
int child[9]={1,3,5,6,8,-1,-1,-1,-1};
int sibling[9]={-1,2,-1,4,-1,-1,7,-1,-1};
int edge0[16]={0,1,1,3,3,6,3,7,1,4,0,2,4,8,2,5};
int edge1[16]={1,0,3,1,6,3,7,3,4,1,2,0,8,4,5,2};
```

```
int succ0[16]; int succ1[16]; int position[16]; int preorder[9]; int
*dparent,*dchild,*dsibling,*dedge0,*dedge1,*dsucc0,*dsucc1; int
*dposition,*dpreorder;
cudaMalloc((void**)&dparent,9*sizeof(int));
cudaMalloc((void**)&dchild,9*sizeof(int));
cudaMalloc((void**)&dsibling,9*sizeof(int));
cudaMalloc((void**)&dedge0,16*sizeof(int));
cudaMalloc((void**)&dedge1,16*sizeof(int));
cudaMalloc((void**)&dsucc0,16*sizeof(int));
cudaMalloc((void**)&dsucc1,16*sizeof(int));
cudaMalloc((void**)&dposition,16*sizeof(int));
cudaMalloc((void**)&dpreorder,9*sizeof(int));
cudaMemcpy(dparent,&parent,9*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dchild,&child,9*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dsibling,&sibling,9*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dedge0,&edge0,16*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dedge1,&edge1,16*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dsucc0,&succ0,16*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dsucc1,&succ1,16*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dposition,&position,16*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dpreorder,&preorder,9*sizeof(int),cudaMemcpyHostToDevice);
initialize<<<1,1>>>(dedge0,dedge1);
traverse<<<1,16>>>(dparent,dchild,dsibling,dedge0,dedge1,dsucc0,dsucc1,
dposition,dpreorder);
cudaMemcpy(&succ0,dsucc0,16*sizeof(int),cudaMemcpyDeviceToHost);
cudaMemcpy(&succ1,dsucc1,16*sizeof(int),cudaMemcpyDeviceToHost);
cudaMemcpy(&preorder,dpreorder,9*sizeof(int),cudaMemcpyDeviceToHost);
printf("Preorder Traversal numbering to the vertices: \n");
for(int i=0;i<9;i++){
 printf("%c -> %d\n",vertices[i],preorder[i]);
```

```
}
int logN = (int)ceil(log2((double)N));
 printf("\nTime Complexity : O(log N) = O(%d)\n", logN);
 printf("Cost Complexity : O(N log N) = O(%d * %d) = O(%d)\n", N, logN, N*logN);
cudaFree(dparent);
cudaFree(dchild);
cudaFree(dsibling);
cudaFree(dedge0);
cudaFree(dedge1);
cudaFree(dsucc0);
cudaFree(dsucc1);
cudaFree(dposition);
cudaFree(dpreorder);
return 0;
}
```

OUTPUT:



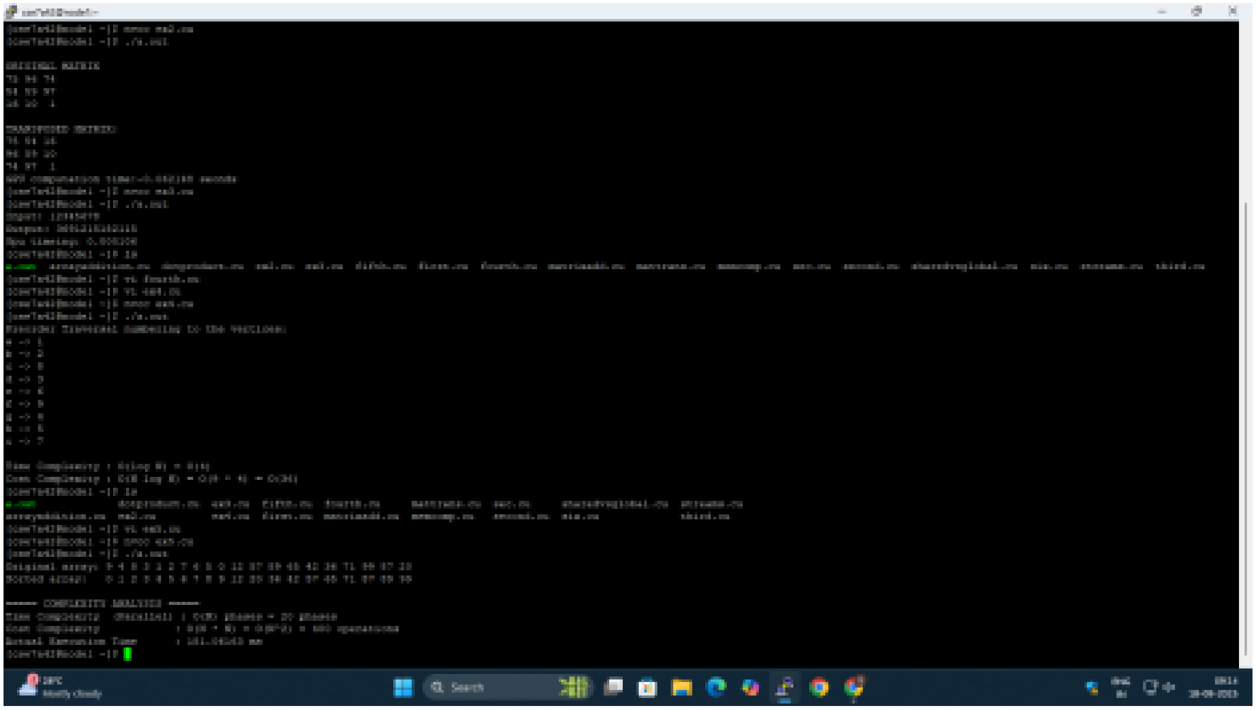
5)ODD EVEN TRANSPOSITION SORT

```
#include <stdio.h>
#define N 20
__global__ void oddEvenTS(int *data, int n, int phase)
{
 int tid = threadIdx.x + blockIdx.x * blockDim.x;
 if (tid < n - 1) {
 if (phase % 2 == 0 && (tid % 2 == 0)) {
 if (data[tid] > data[tid + 1]) {
 int temp = data[tid];
 data[tid] = data[tid + 1];
 data[tid + 1] = temp;
 } }
 if (phase % 2 == 1 && (tid % 2 == 1)) {
```

```
if (data[tid] > data[tid + 1]) {
int temp = data[tid];
data[tid] = data[tid + 1];
data[tid + 1] = temp;
} } }
}

int main()
{
int h_data[N] = {9,4,8,3,1,2,7,6,5,0,12,57,89,65,42,36,71,99,87,20};
int *d_data;
printf("Original array: ");
for (int i = 0; i < N; i++)
printf("%d ", h_data[i]); printf("\n");

cudaMalloc((void **)&d_data, N * sizeof(int));

cudaMemcpy(d_data, h_data, N * sizeof(int), cudaMemcpyHostToDevice);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
int threadsPerBlock =10;
int blocksPerGrid = (N * N + threadsPerBlock - 1) / threadsPerBlock;
cudaEventRecord(start);

for (int phase = 0; phase < N; phase++) {
oddEvenTS<<<blocksPerGrid, threadsPerBlock>>>(d_data, N, phase);
cudaDeviceSynchronize(); }
```

```
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float elapsedTime = 0;
cudaEventElapsedTime(&elapsedTime, start, stop);

cudaMemcpy(h_data, d_data, N * sizeof(int), cudaMemcpyDeviceToHost);

printf("Sorted array: ");
for (int i = 0; i < N; i++)
printf("%d ", h_data[i]);
printf("\n");

printf("\n===== COMPLEXITY ANALYSIS =====\n");
printf("Time Complexity (Parallel) : O(N) phases ≈ %d phases\n", N);
printf("Cost Complexity : O(N * N) = O(N^2) ≈ %d operations\n", N * N);
printf("Actual Execution Time : %.5f ms\n", elapsedTime);

cudaFree(d_data);
cudaEventDestroy(start);
cudaEventDestroy(stop);
return 0;
}
```

OUTPUT:



6)PARALLEL QUICK SORT

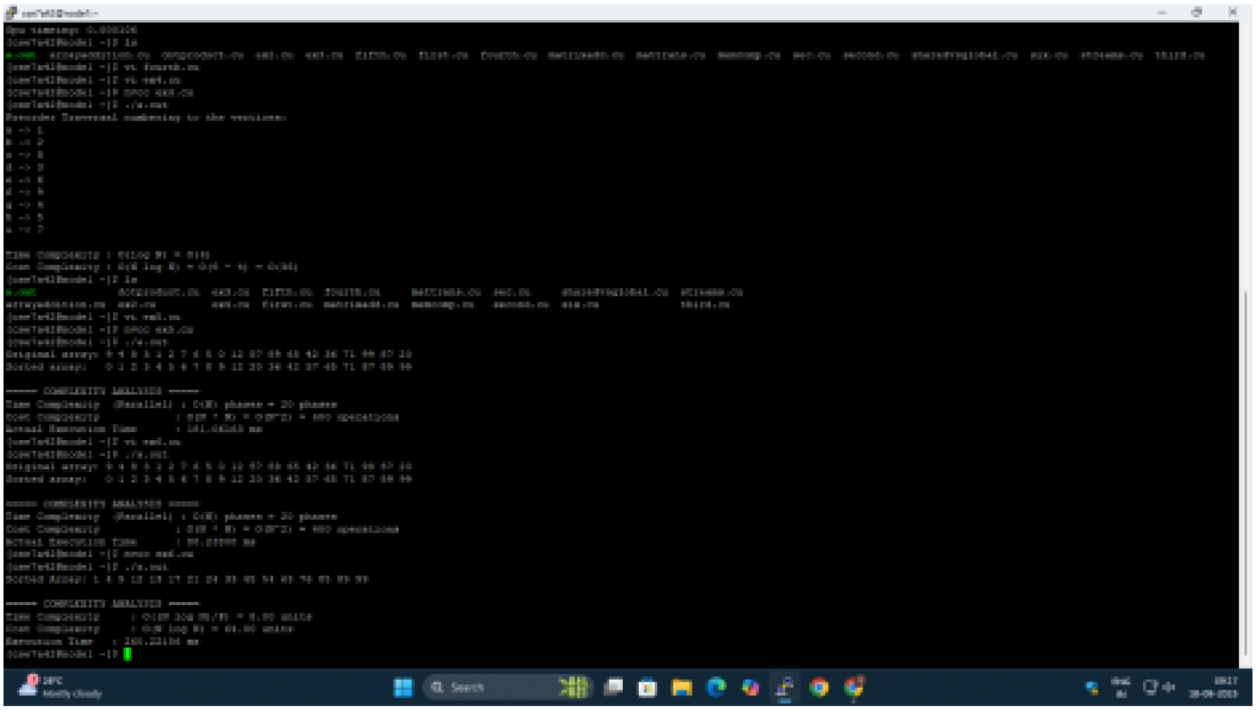
```
#include <stdio.h>
#define N 16
#define TPB 8
#define MIN_PART 16

__device__ void insertionSort(int *a, int l, int r) {
    for (int i = l + 1; i <= r; i++) {
        int key = a[i], j = i - 1;
        while (j >= l && a[j] > key) { a[j + 1] = a[j]; j--; }
        a[j + 1] = key;
    }
}

__device__ int partition(int *a, int l, int r) {
```

```
    int p = a[r], i = l - 1;
    for (int j = l; j < r; j++)
        if (a[j] < p) { i++; int t = a[i]; a[i] = a[j]; a[j] = t; }
    int t = a[i + 1]; a[i + 1] = a[r]; a[r] = t;
    return i + 1;
}

__global__ void quicksortKernel(int *a, int *l, int *r, int *nl, int *nr, int nTasks, int *next) {
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id >= nTasks) return;

    int low = l[id], high = r[id];
    if (low < high) {
        if (high - low + 1 <= MIN_PART) { insertionSort(a, low, high); return; }

        int p = partition(a, low, high);
        if (p - 1 > low) { int idx = atomicAdd(next, 1); nl[idx] = low; nr[idx] = p - 1; } if
    (p + 1 < high){ int idx = atomicAdd(next, 1); nl[idx] = p + 1; nr[idx] = high; } } }
int main() {
    int h_a[N] = {24,17,85,13,9,54,76,45,4,63,21,33,89,12,99,1};
    int *d_a; cudaMalloc(&d_a, N*sizeof(int));
    cudaMemcpy(d_a, h_a, N*sizeof(int), cudaMemcpyHostToDevice);

    int *l, *r, *nl, *nr, *next;
    cudaMalloc(&l, N*sizeof(int)); cudaMalloc(&r, N*sizeof(int));
    cudaMalloc(&nl, N*sizeof(int)); cudaMalloc(&nr, N*sizeof(int));
    cudaMalloc(&next, sizeof(int));

    int h_l[1] = {0}, h_r[1] = {N-1};
    cudaMemcpy(l, h_l, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(r, h_r, sizeof(int), cudaMemcpyHostToDevice);

    cudaEvent_t start, stop; cudaEventCreate(&start); cudaEventCreate(&stop);
```

```
cudaEventRecord(start);
int nTasks = 1;
while (nTasks > 0) {
    cudaMemset(next, 0, sizeof(int));
    int blocks = (nTasks + TPB - 1) / TPB;
    quicksortKernel<<<blocks, TPB>>>(d_a, l, r, nl, nr, nTasks, next);
    cudaDeviceSynchronize();
    cudaMemcpy(&nTasks, next, sizeof(int), cudaMemcpyDeviceToHost);


    int *tmpL = l; l = nl; nl = tmpL;
    int *tmpR = r; r = nr; nr = tmpR;
}
cudaEventRecord(stop); cudaEventSynchronize(stop);
float ms; cudaEventElapsedTime(&ms, start, stop);

cudaMemcpy(h_a, d_a, N*sizeof(int), cudaMemcpyDeviceToHost);

double log2n = log2((double)N);
double timec = ((double)N * log2n) / TPB;
double costc = (double)N * log2n;

printf("Sorted Array: ");
for (int i = 0; i < N; i++) printf("%d ", h_a[i]);
printf("\n\n===== COMPLEXITY ANALYSIS =====\n");
printf("Time Complexity : O((N log N)/P) = %.2f units\n", timec);
printf("Cost Complexity : O(N log N) = %.2f units\n", costc);
printf("Execution Time : %.5f ms\n", ms);

cudaFree(d_a); cudaFree(l); cudaFree(r);
cudaFree(nl); cudaFree(nr); cudaFree(next);
cudaEventDestroy(start); cudaEventDestroy(stop);
return 0;
```

```
}
```

OUTPUT:



7)CHAT SERVER APPLICATION WITH MULTIPLE CLIENTS:

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]) {
int rank, size;
char message[100];
char input_buffer[100];
int i;
double start_time, end_time, elapsed_time;
```

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Barrier(MPI_COMM_WORLD);
start_time = MPI_Wtime();
if (rank == 0) {
printf("Enter message to broadcast: ");
fflush(stdout);
if (fgets(input_buffer, 100, stdin) != NULL) {
input_buffer[strcspn(input_buffer, "\n")] = '\0'; // Remove newline
} else {
strcpy(input_buffer, "Hello Clients, this is a broadcast from Server!");
}
strcpy(message, input_buffer);
MPI_Bcast(message, 100, MPI_CHAR, 0, MPI_COMM_WORLD);
printf("Server (rank %d) broadcasted: %s\n", rank, message);
for (i = 1; i < size; i++) {
char response[100];
MPI_Recv(response, 100, MPI_CHAR, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("Server received response from client %d: %s\n", i, response);
}
} else {
MPI_Bcast(message, 100, MPI_CHAR, 0, MPI_COMM_WORLD);
printf("Client (rank %d) received: %s\n", rank, message);
char response[100];
sprintf(response, "Hello Server, client %d received your message!", rank);
MPI_Send(response, strlen(response) + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
}

MPI_Barrier(MPI_COMM_WORLD);
end_time = MPI_Wtime();
if (rank == 0) {
```

```
elapsed_time = end_time-start_time;
printf("\n--All processes completed ---\n");
printf("Total execution time: %f seconds\n", elapsed_time);
}
MPI_Finalize();
return 0;
}
```

```
[cse7a59@sastra-masternode ~]$ mpirun -np 10 out
Enter message to broadcast: hi
Server (rank 0) broadcasted: hi
Server received response from client 1: Hello Server, client 1 received your mes
sage!
Client (rank 1) received: hi
Client (rank 2) received: hi
Client (rank 3) received: hi
Client (rank 4) received: hi
Client (rank 5) received: hi
Client (rank 6) received: hi
Client (rank 7) received: hi
Client (rank 8) received: hi
Client (rank 9) received: hi
Server received response from client 2: Hello Server, client 2 received your mes
sage!
Server received response from client 3: Hello Server, client 3 received your mes
sage!
Server received response from client 4: Hello Server, client 4 received your mes
sage!
Server received response from client 5: Hello Server, client 5 received your mes
sage!
Server received response from client 6: Hello Server, client 6 received your mes
sage!
Server received response from client 7: Hello Server, client 7 received your mes
sage!
Server received response from client 8: Hello Server, client 8 received your mes
sage!
Server received response from client 9: Hello Server, client 9 received your mes
sage!

--All processes completed ---
Total execution time: 3.487687 seconds
[cse7a59@sastra-masternode ~]$
```

8)MUTUAL EXCLUSION:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define REQUEST 1
#define REPLY 2
#define RELEASE 3
typedef enum {IDLE, WANTED, HELD, RELEASED} State;
void handle_error(int errcode, const char* msg, int rank) {
if(errcode != MPI_SUCCESS) {
char err_string[MPI_MAX_ERROR_STRING];
int resultlen;
MPI_Error_string(errcode, err_string, &resultlen);
fprintf(stderr, "Process %d: MPI error at %s: %s\n", rank, msg, err_string);
MPI_Abort(MPI_COMM_WORLD, errcode);
}
}
int main(int argc, char **argv) {
int rank, size, i;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
State state = IDLE;
int logical_clock = 0;
int request_ts = 0;
int release_flag = 0;
int err;
double start_time, end_time, elapsed_time;
```

```
start_time = MPI_Wtime();
if(rank == 0) {
state = WANTED;
logical_clock++;
request_ts = logical_clock;
printf("Process %d requesting critical section\n", rank); fflush(stdout);
err = MPI_Bcast(&request_ts, 1, MPI_INT, 0, MPI_COMM_WORLD);
handle_error(err, "MPI_Bcast REQUEST", rank);
printf("Process %d sent REQUEST to all other processes\n", rank); fflush(stdout);

int replies = 0;
MPI_Status status;
while(replies < size-1){
int recv_ts;
err = MPI_Recv(&recv_ts, 1, MPI_INT, MPI_ANY_SOURCE, REPLY, MPI_COMM_WORLD,
&status);
handle_error(err, "MPI_Recv REPLY", rank);
logical_clock = (logical_clock > recv_ts ? logical_clock : recv_ts) + 1;
printf("Process %d received REPLY from process %d\n", rank, status.MPI_SOURCE);
fflush(stdout);
replies++;
}
state = HELD;
printf("Process %d in critical section\n", rank); fflush(stdout);
int sum = 0;
for(i=1;i<=5;i++) sum += i*i;
printf("Process %d computed sum of squares 1..5 = %d\n", rank, sum); fflush(stdout);
sleep(1);
state = RELEASED;
printf("Process %d releasing critical section\n", rank); fflush(stdout);
logical_clock++;
release_flag = 1;
```

```
err = MPI_Bcast(&release_flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
handle_error(err, "MPI_Bcast RELEASE", rank);
state = IDLE;
}
else {
err = MPI_Bcast(&request_ts, 1, MPI_INT, 0, MPI_COMM_WORLD);
handle_error(err, "MPI_Bcast REQUEST", rank);
logical_clock = (logical_clock > request_ts ? logical_clock : request_ts) + 1;
err = MPI_Send(&logical_clock, 1, MPI_INT, 0, REPLY, MPI_COMM_WORLD);
handle_error(err, "MPI_Send REPLY", rank);
err = MPI_Bcast(&release_flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
handle_error(err, "MPI_Bcast RELEASE", rank);
if(release_flag)
printf("Process %d received RELEASE from process 0\n", rank); fflush(stdout);
}
MPI_Barrier(MPI_COMM_WORLD);
end_time = MPI_Wtime();
if (rank == 0) {

elapsed_time = end_time-start_time;
printf("Total execution time: %f seconds\n", elapsed_time);
}
MPI_Finalize();
return 0;
}
```

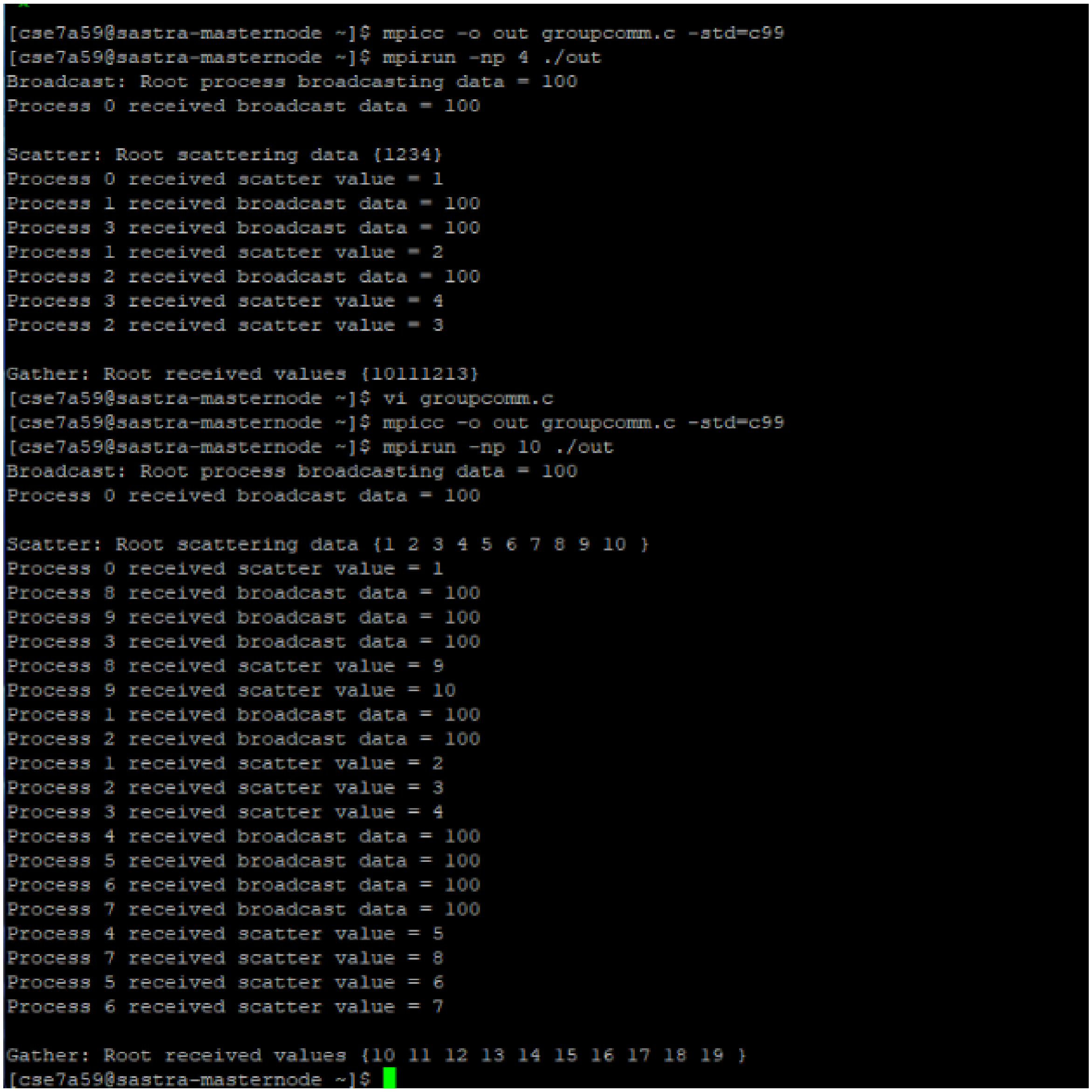
```
[cse7a42@sastra-masternode ~]$ vi ex8.c
[cse7a42@sastra-masternode ~]$ mpicc -o out2 ex8.c
[cse7a42@sastra-masternode ~]$ mpirun -np 4 out2
Process 0 requesting critical section
Process 0 sent REQUEST to all other processes
Process 0 received REPLY from process 2
Process 0 received REPLY from process 1
Process 0 received REPLY from process 3
Process 0 in critical section
Process 0 computed sum of squares 1..5 = 55
Process 0 releasing critical section
Process 2 received RELEASE from process 0
Process 1 received RELEASE from process 0
Process 3 received RELEASE from process 0
Total execution time: 1.059549 seconds
[cse7a42@sastra-masternode ~]$
```

9)GROUP COMMUNICATION:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char** argv) {
int world_size, world_rank;
const int root_process = 0;
double start_time, end_time, local_elapsed_time, max_elapsed_time;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

MPI_Barrier(MPI_COMM_WORLD);
start_time = MPI_Wtime();
int bcast_data;
if (world_rank == root_process) {
bcast_data = 100;
```

```
printf("Process %d broadcasts %d\n", root_process, bcast_data);
}
MPI_Bcast(&bcast_data, 1, MPI_INT, root_process, MPI_COMM_WORLD);
if (world_rank != root_process) {
printf("Process %d received %d from bcast\n", world_rank, bcast_data);
}
MPI_Barrier(MPI_COMM_WORLD);
printf("\n");
int* send_data = NULL;
int recv_data;
const int num_elements = 4;
if (world_rank == root_process) {
send_data = (int*)malloc(world_size * sizeof(int));
for (int i = 0; i < world_size; i++) {
send_data[i] = (i + 1) * 10;
}
}
MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, root_process,
MPI_COMM_WORLD);
printf("Process %d received %d from scatter\n", world_rank, recv_data);
int* gathered_data = NULL;
if (world_rank == root_process) {
gathered_data = (int*)malloc(world_size * sizeof(int));
}
MPI_Gather(&recv_data, 1, MPI_INT, gathered_data, 1, MPI_INT,
root_process,
MPI_COMM_WORLD);
if (world_rank == root_process) {
printf("Root process gathered data:");
for (int i = 0; i < world_size; i++) {
printf(" %d", gathered_data[i]);
}
```

```
printf("\n");
free(send_data);
free(gathered_data);
}
MPI_Barrier(MPI_COMM_WORLD);
printf("\n"); // Add a newline for separation

int local_value = recv_data;
int reduction_result = 0;
MPI_Reduce(&local_value, &reduction_result, 1, MPI_INT, MPI_SUM,
root_process,
MPI_COMM_WORLD);
if (world_rank == root_process) {
printf("Reduction result: %d\n", reduction_result);
}
MPI_Barrier(MPI_COMM_WORLD);
end_time = MPI_Wtime();
local_elapsed_time = end_time - start_time;
MPI_Reduce(&local_elapsed_time, &max_elapsed_time, 1, MPI_DOUBLE,
MPI_MAX, root_process, MPI_COMM_WORLD);
if (world_rank == root_process) {
printf("\nTotal Execution Time: %f seconds\n", max_elapsed_time);
}
MPI_Finalize();
return 0;
}
```



```
[cse7a59@sastra-masternode ~]$ mpicc -o out groupcomm.c -std=c99
[cse7a59@sastra-masternode ~]$ mpirun -np 4 ./out
Broadcast: Root process broadcasting data = 100
Process 0 received broadcast data = 100

Scatter: Root scattering data {1234}
Process 0 received scatter value = 1
Process 1 received broadcast data = 100
Process 3 received broadcast data = 100
Process 1 received scatter value = 2
Process 2 received broadcast data = 100
Process 3 received scatter value = 4
Process 2 received scatter value = 3

Gather: Root received values {10111213}
[cse7a59@sastra-masternode ~]$ vi groupcomm.c
[cse7a59@sastra-masternode ~]$ mpicc -o out groupcomm.c -std=c99
[cse7a59@sastra-masternode ~]$ mpirun -np 10 ./out
Broadcast: Root process broadcasting data = 100
Process 0 received broadcast data = 100

Scatter: Root scattering data {1 2 3 4 5 6 7 8 9 10 }
Process 0 received scatter value = 1
Process 8 received broadcast data = 100
Process 9 received broadcast data = 100
Process 3 received broadcast data = 100
Process 8 received scatter value = 9
Process 9 received scatter value = 10
Process 1 received broadcast data = 100
Process 2 received broadcast data = 100
Process 1 received scatter value = 2
Process 2 received scatter value = 3
Process 3 received scatter value = 4
Process 4 received broadcast data = 100
Process 5 received broadcast data = 100
Process 6 received broadcast data = 100
Process 7 received broadcast data = 100
Process 4 received scatter value = 5
Process 7 received scatter value = 8
Process 5 received scatter value = 6
Process 6 received scatter value = 7

Gather: Root received values {10 11 12 13 14 15 16 17 18 19 }
[cse7a59@sastra-masternode ~]$
```

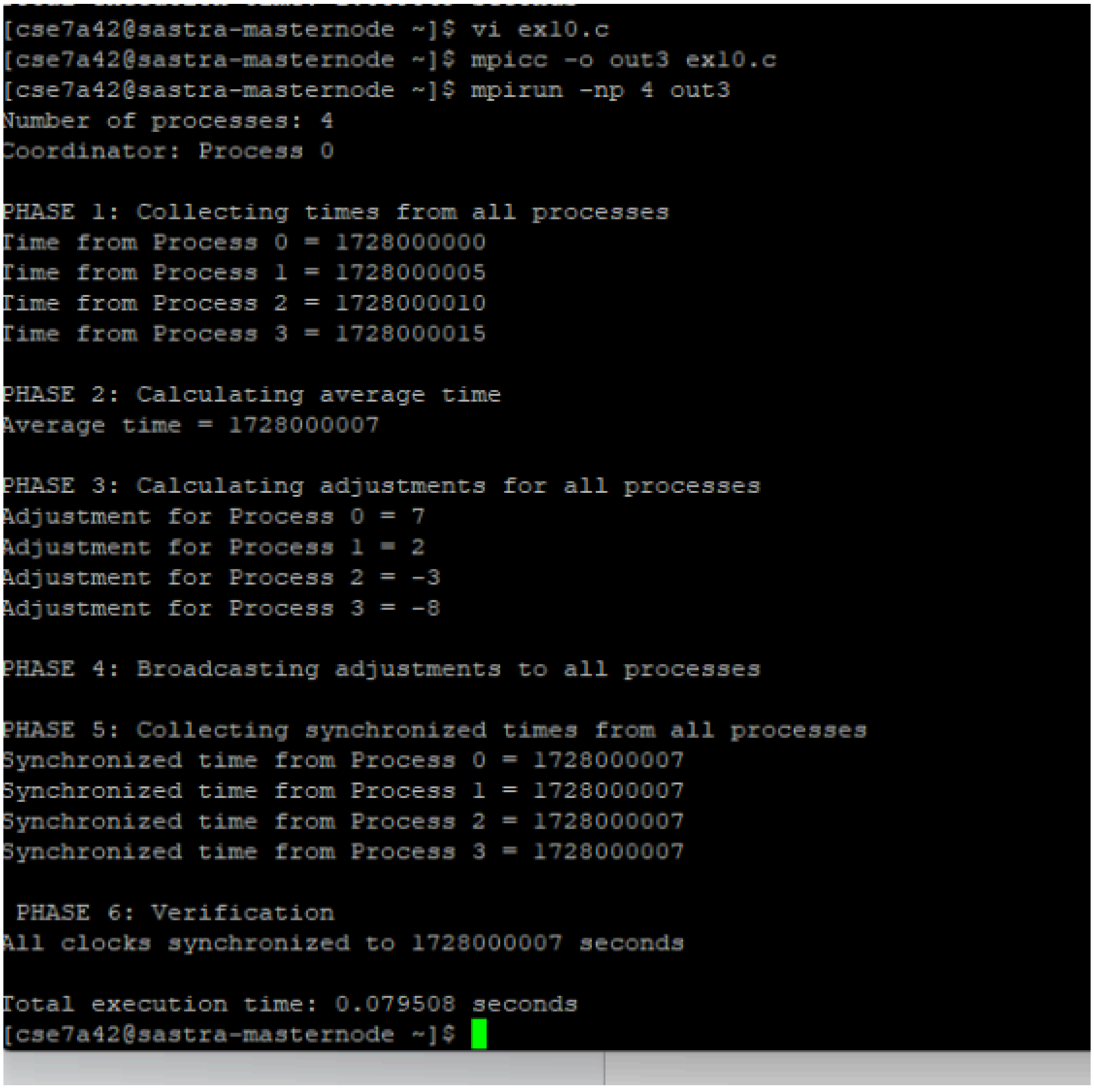
<u>10)CLOCK SYNCHRONIZATION:</u>

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define MASTER 0
long get_initial_time(int rank) {
long base_time = 1728000000L;
```

```
return base_time + (rank * 5);
}
int main(int argc, char** argv) {
int rank, size, i;
long local_time, average_time;
long *all_times = NULL;
long *adjustments = NULL;
long *synced_times = NULL;
double start_time, end_time;
MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
start_time = MPI_Wtime();
local_time = get_initial_time(rank);
MPI_Barrier(MPI_COMM_WORLD);
if (rank == MASTER) {
printf("Number of processes: %d\n", size);
printf("Coordinator: Process %d\n\n", MASTER);
all_times = malloc(size * sizeof(long));
adjustments = malloc(size * sizeof(long));
synced_times = malloc(size * sizeof(long));
printf("PHASE 1: Collecting times from all processes\n");
all_times[0] = local_time;
MPI_Gather(MPI_IN_PLACE, 1, MPI_LONG, all_times, 1, MPI_LONG, MASTER,
MPI_COMM_WORLD);
for (i = 0; i < size; i++)
printf("Time from Process %d = %ld\n", i, all_times[i]);
printf("\nPHASE 2: Calculating average time\n");
long sum = 0;
for (i = 0; i < size; i++) sum += all_times[i];
average_time = sum / size;
```

```
printf("Average time = %ld\n", average_time);
printf("\nPHASE 3: Calculating adjustments for all processes\n");
for (i = 0; i < size; i++) {
adjustments[i] = average_time - all_times[i];
printf("Adjustment for Process %d = %ld\n", i, adjustments[i]);
}
printf("\nPHASE 4: Broadcasting adjustments to all processes\n");
MPI_Bcast(adjustments, size, MPI_LONG, MASTER, MPI_COMM_WORLD);
local_time += adjustments[0];
printf("\nPHASE 5: Collecting synchronized times from all processes\n");
synced_times[0] = local_time;
MPI_Gather(MPI_IN_PLACE, 1, MPI_LONG, synced_times, 1, MPI_LONG, MASTER,
MPI_COMM_WORLD);
for (i = 0; i < size; i++)
printf("Synchronized time from Process %d = %ld\n", i, synced_times[i]);
printf("\n PHASE 6: Verification\n");
int ok = 1;

for (i = 1; i < size; i++)
if (synced_times[i] != synced_times[0]) ok = 0;
if (ok)
printf("All clocks synchronized to %ld seconds\n", synced_times[0]);
else
printf("Synchronization failed\n");
free(all_times);
free(adjustments);
free(synced_times);
} else {
MPI_Gather(&local_time, 1, MPI_LONG, NULL, 0, MPI_LONG, MASTER,
MPI_COMM_WORLD);
adjustments = malloc(size * sizeof(long));
MPI_Bcast(adjustments, size, MPI_LONG, MASTER, MPI_COMM_WORLD);
local_time += adjustments[rank];
```

```
MPI_Gather(&local_time, 1, MPI_LONG, NULL, 0, MPI_LONG, MASTER,
MPI_COMM_WORLD);
free(adjustments);
}
MPI_Barrier(MPI_COMM_WORLD);
if (rank == MASTER) {
end_time = MPI_Wtime();
printf("\nTotal execution time: %f seconds\n", end_time - start_time);
}
MPI_Finalize();
return 0;
}
```



```
[cse7a42@sastra-masternode ~]$ vi ex10.c
[cse7a42@sastra-masternode ~]$ mpicc -o out3 ex10.c
[cse7a42@sastra-masternode ~]$ mpirun -np 4 out3
Number of processes: 4
Coordinator: Process 0

PHASE 1: Collecting times from all processes
Time from Process 0 = 1728000000
Time from Process 1 = 1728000005
Time from Process 2 = 1728000010
Time from Process 3 = 1728000015

PHASE 2: Calculating average time
Average time = 1728000007

PHASE 3: Calculating adjustments for all processes
Adjustment for Process 0 = 7
Adjustment for Process 1 = 2
Adjustment for Process 2 = -3
Adjustment for Process 3 = -8

PHASE 4: Broadcasting adjustments to all processes

PHASE 5: Collecting synchronized times from all processes
Synchronized time from Process 0 = 1728000007
Synchronized time from Process 1 = 1728000007
Synchronized time from Process 2 = 1728000007
Synchronized time from Process 3 = 1728000007

 PHASE 6: Verification
All clocks synchronized to 1728000007 seconds

Total execution time: 0.079508 seconds
[cse7a42@sastra-masternode ~]$
```

11)LEADER ELECTION:

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

int main(int argc, char** argv) {
double start_time, end_time;
```

```
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (size < 2) {
if (rank == 0) {
printf("Need at least 2 processes.\n");
}
MPI_Finalize();
return 0;
}

int initiator = 3,i,failed = size - 2,leader = -1;
int successor = (rank + 1) % size,predecessor = (rank + size - 1) % size;
int *ranks_array = (int *)malloc(size * sizeof(int));
int array_size = 0;
start_time = MPI_Wtime();
if (rank == initiator) {
printf("No. of processes = %d\n",size);
printf("Process %d Detected failure of coordinator process %d and initiated an
election.\n", rank, failed);
if (rank != failed) {

ranks_array[0] = rank;
array_size = 1;
}
printf("Rank array at process %d: {",rank);
for(i = 0; i < array_size-1; i++){
printf("%d, ",ranks_array[i]);
}
printf("%d}\n",ranks_array[array_size-1]);
```

```
MPI_Send(ranks_array, size, MPI_INT, successor, 0, MPI_COMM_WORLD);
MPI_Send(&array_size, 1, MPI_INT, successor, 1, MPI_COMM_WORLD);
MPI_Recv(ranks_array, size, MPI_INT, predecessor, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
MPI_Recv(&array_size, 1, MPI_INT, predecessor, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
printf("Final Rank array: {",rank);
for(i = 0; i < array_size-1; i++){
printf("%d, ",ranks_array[i]);
}
printf("%d}\n",ranks_array[array_size-1]);

leader = -1;
for (i = 0; i < array_size; i++) {
if (ranks_array[i] > leader) {
leader = ranks_array[i];
}
}
printf("Process %d elected as coordinator. Announcing coordinator to all other
processes.\n",leader);
printf("Acknowledgement by all processes:\n");

} else {
MPI_Recv(ranks_array, size, MPI_INT, predecessor, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
MPI_Recv(&array_size, 1, MPI_INT, predecessor, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
if (rank != failed) {
ranks_array[array_size] = rank;
array_size++;
printf("Rank array at process %d: {",rank);
```

```
for(i = 0; i < array_size-1; i++){
printf("%d, ",ranks_array[i]);
}
printf("%d}\n",ranks_array[array_size-1]);
}
MPI_Send(ranks_array, size, MPI_INT, successor, 0, MPI_COMM_WORLD);
MPI_Send(&array_size, 1, MPI_INT, successor, 1, MPI_COMM_WORLD);
}
MPI_Bcast(&leader, 1, MPI_INT, initiator, MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);

for (i = 0; i < size; i++) {
if (rank == i && rank != failed) {
if(rank != leader)
printf("Process %d acknowledges process %d as coordinator\n", rank,

leader);
else
printf("Process %d is the coordinator\n", rank);
}
MPI_Barrier(MPI_COMM_WORLD);
}

end_time = MPI_Wtime();
if (rank == 0) {
printf("Execution time: %f seconds\n", end_time - start_time);
}

free(ranks_array);
MPI_Finalize();
return 0;
```

}

```
[cse7a42@sastra-masternode ~]$ vi ex11.c
[cse7a42@sastra-masternode ~]$ mpicc -o out4 ex11.c
[cse7a42@sastra-masternode ~]$ mpirun -np 4 out4
No. of processes = 4
Process 3 Detected failure of coordinator process 2 and initiated an election.
Rank array at process 3: {3}
Rank array at process 0: {3, 0}
Rank array at process 1: {3, 0, 1}
Final Rank array: {3, 0, 1}
Process 3 elected as coordinator. Announcing coordinator to all other processes.
Acknowledgement by all processes:
Process 0 acknowledges process 3 as coordinator
Process 1 acknowledges process 3 as coordinator
Process 3 is the coordinator
Execution time: 0.156027 seconds
[cse7a42@sastra-masternode ~]$
```

int *ranks_array = (int *)malloc(si

12)BYZANTINE AGREEMENT PROBLEM:

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

#define N 4

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

int received[N];
int received_matrix[N][N]; // stores what each process received from each other
int faulty_rank = 3; // process 3 acts Byzantine

MPI_Init(&argc, &argv);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (size != N) {
if (rank == 0)
printf("Run with %d processes only!\n", N);
MPI_Finalize();
return 0;
}

srand(time(NULL) + rank);

value = rand() % 2;
printf("Process %d initial value: %d\n", rank, value);

MPI_Barrier(MPI_COMM_WORLD);

for (int i = 0; i < size; i++) {
if (i == rank) continue; // skip self
int send_value = value;

if (rank == faulty_rank) {
send_value = rand() % 2;
printf("Process %d (faulty) sends %d to Process %d\n", rank, send_value, i);
} else {

printf("Process %d sends %d to Process %d\n", rank, send_value, i);
}

MPI_Send(&send_value, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
}
```

```
for (int i = 0; i < size; i++) {
if (i == rank) {
received[i] = value;
continue;
}
MPI_Recv(&received[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}

printf("\nProcess %d received values: ", rank);
for (int i = 0; i < size; i++)
printf("%d ", received[i]);
printf("\n");

MPI_Allgather(received, N, MPI_INT, received_matrix, N, MPI_INT,
MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);

if (rank == 0) {
printf("\n--- Fault Detection Report ---\n");
for (int sender = 0; sender < N; sender++) {
int consistent = 1;

for (int i = 1; i < N; i++) {
if (received_matrix[i][sender] != received_matrix[0][sender])
consistent = 0;
}

if (!consistent)
printf(" Process %d is detected as FAULTY (sent inconsistent values)\n",
sender);
```

```
else
printf(" Process %d is consistent across all receivers\n", sender);
}
}

MPI_Finalize();
return 0;
}
```

```
[cse7a42@sastra-masternode ~]$ mpirun -np 4 out5
Process 2 initial value: 1
Process 3 initial value: 1
Process 0 initial value: 0
Process 1 initial value: 0
Process 0 sends 0 to Process 1
Process 1 sends 0 to Process 0
Process 0 sends 0 to Process 2
Process 0 sends 0 to Process 3
Process 2 sends 1 to Process 0
Process 2 sends 1 to Process 1
Process 2 sends 1 to Process 3
Process 1 sends 0 to Process 2
Process 1 sends 0 to Process 3
Process 3 (faulty) sends 0 to Process 0
Process 3 (faulty) sends 0 to Process 1
Process 3 (faulty) sends 0 to Process 2

Process 1 received values: 0 0 1 0

Process 3 received values: 0 0 1 1

Process 0 received values: 0 0 1 0

Process 2 received values: 0 0 1 0

--- Fault Detection Report ---
 Process 0 is consistent across all receivers
 Process 1 is consistent across all receivers
 Process 2 is consistent across all receivers
 Process 3 is detected as FAULTY (sent inconsistent values)
[cse7a42@sastra-masternode ~]$
```