Parallel & Distributed Computing [Y1]

LAB MANUAL/REPORT

HAMAD SALEEM
F2021376065
MIAN MUHAMMAD BILAL
[F2021408054]
MUHAMMAD TALHA
[F2021408057]

LAB MANUAL

Matrix Operations:

The parallelized matrix multiplication code uses OpenMP to distribute the computation of matrix elements across multiple threads. Each thread computes individual elements of the result matrix by summing the products of corresponding elements from the input matrices.

When race conditions occur, the results of these operations can become unpredictable due to unsynchronized access to shared data.

With the Race condition

Code:

```c
#include <stdio.h>
#include <omp.h>
#define N 3
void printMatrix(int mat[N][N]){
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}
void addMatrices(int A[N][N], int B[N][N], int C[N][N]){
    int sum = 0;
#pragma omp parallel for
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            C[i][j] = A[i][j] + B[i][j];
            // Race condition: multiple threads update 'sum' concurrently
            sum += C[i][j];
        }
    }
    printf("Sum of all elements in C: %d\n", sum);
}

int main(){
    int A[N][N] = {{1, 1, 1}, {1, 1, 1}, {1, 1, 1}};
    int B[N][N] = {{2, 2, 2}, {2, 2, 2}, {2, 2, 2}};
    int C[N][N];
    addMatrices(A, B, C);
    printf("Matrix A:\n");
    printMatrix(A);
    printf("\nMatrix B:\n");
    printMatrix(B);
    printf("\nMatrix C (A + B):\n");
    printMatrix(C);
    return 0;
}
```

Output:

```
2 2 2
2 2 2
2 2 2

Matrix C (A + B):
3 3 3
3 3 3
3 3 3

┌──(biologist㉿kali)-[~/…/Parallel & Distr
ibuted Computing/Lab/Report/1.matrix_opera
tions]
└─$ cd "/home/biologist/Documents/2UMT/sem
ester6/Parallel & Distributed Computing/La
b/Report/1.matrix_operations/" && gcc with
_race_cond.c -o with_race_cond && "/home/b
iologist/Documents/2UMT/semester6/Parallel
 & Distributed Computing/Lab/Report/1.matr
ix_operations/"with_race_cond
Sum of all elements in C: 27
Matrix A:
1 1 1
1 1 1
1 1 1

Matrix B:
2 2 2
2 2 2
2 2 2

Matrix C (A + B):
3 3 3
3 3 3
3 3 3
```

Without the Racing condition:

Code:

```c
#include <stdio.h>
#include <omp.h>
#define N 3
// No race condition in this code as each thread handles a unique element
void printMatrix(int mat[N][N]){
#pragma omp parallel for
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}
void addMatrices(int A[N][N], int B[N][N], int C[N][N]){
#pragma omp parallel for
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
int main(){
    int A[N][N] = {{1, 1, 1}, {1, 1, 1}, {1, 1, 1}};
    int B[N][N] = {{2, 2, 2}, {2, 2, 2}, {2, 2, 2}};
    int C[N][N];
    addMatrices(A, B, C);
    printf("Matrix A:\n");
    printMatrix(A);
    printf("\nMatrix B:\n");
    printMatrix(B);
    printf("\nMatrix C (A + B):\n");
    printMatrix(C);

    return 0;
}
```

Output:

```
Matrix A:
1 1 1
1 1 1
1 1 1

Matrix B:
2 2 2
2 2 2
2 2 2

Matrix C (A + B):
3 3 3
3 3 3
3 3 3
```

Prefix Sum: The parallelized prefix sum code leverages OpenMP to divide the array into segments processed by different threads. Each thread calculates partial sums for its segment, and synchronization mechanisms are used to ensure that the cumulative sums from previous segments are correctly included.

With the Race condition:

Code:

```c
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int prefix_sum[10];

    prefix_sum[0] = arr[0]; // First element remains the same

#pragma omp parallel
    {
        int thread_id = omp_get_thread_num();

        // Compute local prefix sum for this thread
        int local_sum = arr[0]; // Initialize with the first element
        for (int i = 1; i < 10; i++)
        {
            local_sum += arr[i];
            // Race condition: multiple threads update the same index of prefix_sum simultaneously
            prefix_sum[i] = local_sum; // Store the local prefix sum
        }
    }

    printf("Prefix sum: ");
    for (int i = 0; i < 10; i++)
    {
        printf("%d ", prefix_sum[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Prefix sum: 1 3 6 10 15 21 28 36 45 55

┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/2.prefix_sum]
└─$ gcc -fopenmp race.c -o race

┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/2.prefix_sum]
└─$ ./race
Prefix sum: 1 3 6 10 15 21 28 36 45 55

┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/2.prefix_sum]
└─$ gcc -fopenmp race.c -o race

┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/2.prefix_sum]
└─$ ./race
Prefix sum: 1 3 6 10 15 21 28 36 45 56

┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/2.prefix_sum]
└─$ gcc -fopenmp race.c -o race

┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/2.prefix_sum]
└─$ ./race
Prefix sum: 1 3 6 10 15 21 28 36 45 55

┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/2.prefix_sum]
└─$ gcc -fopenmp race.c -o race

┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/2.prefix_sum]
└─$ ./race
Prefix sum: 1 3 6 10 15 21 28 36 45 54
```

Without the Racing condition

Code:

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

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int prefix_sum[10];

    prefix_sum[0] = arr[0]; // First element remains the same

#pragma omp parallel for
    for (int i = 1; i < 10; i++)
    {
        prefix_sum[i] = prefix_sum[i - 1] + arr[i];
    }

    printf("Prefix sum: ");
    for (int i = 0; i < 10; i++)
    {
        printf("%d ", prefix_sum[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
┌──(biologist㉿kali)-[~/…/semester6/Parallel &
 Distributed Computing/Lab/Report]
└─$ cd "/home/biologist/Documents/2UMT/semeste
r6/Parallel & Distributed Computing/Lab/Report
/2.prefix_sum/" && gcc without.c -o without &&
 "/home/biologist/Documents/2UMT/semester6/Par
allel & Distributed Computing/Lab/Report/2.pre
fix_sum/"without
Prefix sum: 1 3 6 10 15 21 28 36 45 55
```

Recurrence Sum:

In the parallelized recurrence sum code, each thread computes its partial sum of elements based on the recurrence relation. To prevent race conditions, critical sections or other synchronization mechanisms are employed to ensure that concurrent updates to shared variables are properly coordinated. This coordination guarantees the correctness of the computed sum.

With the Race condition

Code:

```c
#include <stdio.h>
#include <omp.h>
int main(){
    int n;
    printf("Enter the number of terms in the sequence: ");
    scanf("%d", &n);
    // Initialize array to store sequence terms
    int sequence[n];
    // Initialize the first term of the sequence
    int start_value;
    printf("Enter the starting value of the sequence: ");
    scanf("%d", &start_value);
    sequence[0] = start_value;
#pragma omp parallel for
    for (int i = 1; i < n; i++){
        sequence[i] = sequence[i - 1] + 3;
    // Race condition: multiple threads update the same index of sequence simultaneously
    }
    printf("Sequence: ");
    for (int i = 0; i < n; i++){
        printf("%d ", sequence[i]);
    }
    printf("\n");
    return 0;
}
```

Output:

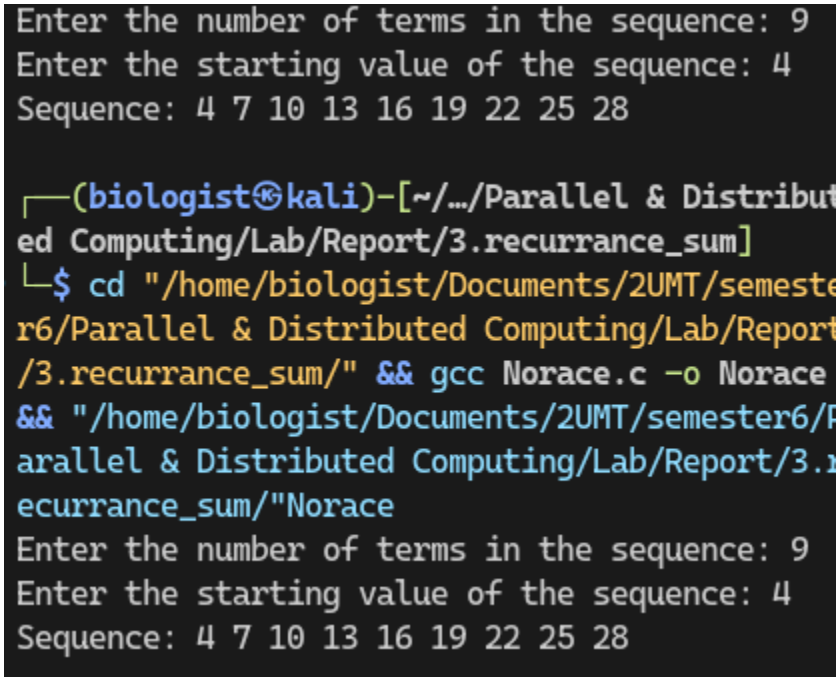
```
ecurrance_sum/"Norace
Enter the number of terms in the sequence: 9
Enter the starting value of the sequence: 4
Sequence: 4 7 10 13 16 19 22 25 28

┌──(biologist㉿kali)-[~/…/Parallel & Distribut
ed Computing/Lab/Report/3.recurrance_sum]
└─$ cd "/home/biologist/Documents/2UMT/semeste
r6/Parallel & Distributed Computing/Lab/Report
/3.recurrance_sum/" && gcc Norace.c -o Norace
&& "/home/biologist/Documents/2UMT/semester6/P
arallel & Distributed Computing/Lab/Report/3.r
ecurrance_sum/"Norace
Enter the number of terms in the sequence: 9
Enter the starting value of the sequence: 4
Sequence: 4 8 12 16 20 24 28 32 36
```

Without the Racing condition

Code:

```c
#include <stdio.h>
#include <omp.h>
int main(){
    int n;
    printf("Enter the number of terms in the sequence: ");
    scanf("%d", &n);
    // Initialize array to store sequence terms
    int sequence[n];
    // Initialize the first term of the sequence
    int start_value;
    printf("Enter the starting value of the sequence: ");
    scanf("%d", &start_value);
    sequence[0] = start_value;
// Parallel computation of local sequence terms
#pragma omp parallel
    {
        // Each thread computes a local sequence segment
        int thread_id = omp_get_thread_num();
        int local_start = (n * thread_id) / omp_get_num_threads();
        int local_end = (n * (thread_id + 1)) / omp_get_num_threads();
        for (int i = local_start + 1; i < local_end; i++){
// Use a critical section to ensure serialized updates
#pragma omp critical
            {
                sequence[i] = sequence[i - 1] + 3;
            }
        }
    }
    // Print the sequence
    printf("Sequence: ");
    for (int i = 0; i < n; i++){
        printf("%d ", sequence[i]);
    }
    printf("\n");
    return 0;
}
```

Output:



Depth First Search (DFS):

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It's used to traverse or search for nodes in a graph or tree structure.

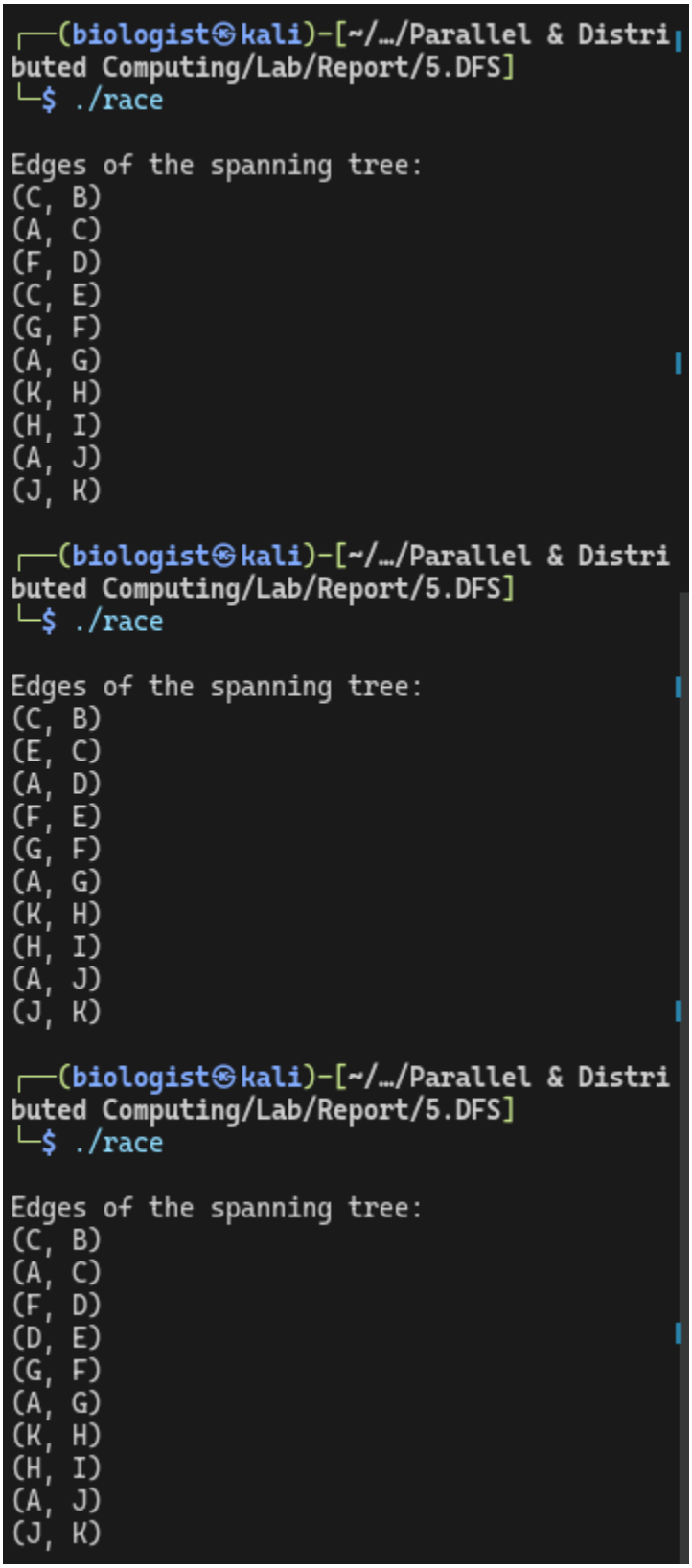
When race conditions occur, the results of these operations can become unpredictable due to unsynchronized access to shared data.

With the Race condition

Code:

```c
#include <stdio.h>
#include <omp.h>
#define V 11          // Number of vertices in the graph
#define NUM_THREADS 4 // Number of threads to use
char vertexLabels[V] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K'};
int visited[V] = {0}; // Array to track visited vertices
int parent[V];        // Array to store the parent of each vertex during traversal
int graph[V][V] = {
    {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0},
    {0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0},
    {0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0},
};

void DFS(int v){
    visited[v] = 1;
    for (int i = 0; i < V; i++){
        if (graph[v][i] && !visited[i]){
            parent[i] = v;
            DFS(i);
        }
    }
}
int main(){
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for
    for (int v = 0; v < V; v++){
        if (!visited[v]){
            DFS(v);
        }
    }
    printf("\nEdges of the spanning tree:\n");
    for (int i = 1; i < V; i++){
        printf("(%c, %c)\n", vertexLabels[parent[i]], vertexLabels[i]);
    }
    return 0;
}
```

Output:

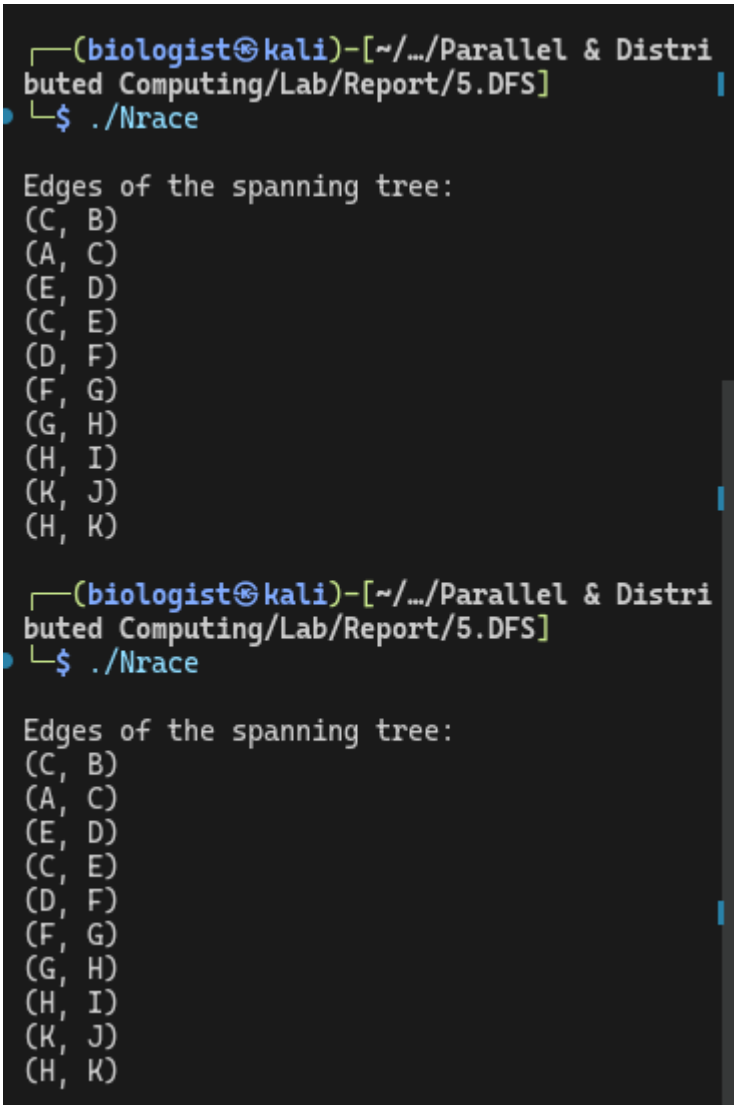

┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/5.DFS]
└─$ ./race
Edges of the spanning tree:
(C, B)
(A, C)
(F, D)
(C, E)
(G, F)
(A, G)
(K, H)
(H, I)
(A, J)
(J, K)
┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/5.DFS]
└─$ ./race
Edges of the spanning tree:
(C, B)
(E, C)
(A, D)
(F, E)
(G, F)
(A, G)
(K, H)
(H, I)
(A, J)
(J, K)
┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/5.DFS]
└─$ ./race
Edges of the spanning tree:
(C, B)
(A, C)
(F, D)
(D, E)
(G, F)
(A, G)
(K, H)
(H, I)
(A, J)
(J, K)

Without the Racing condition

Code:

```c
#include <stdio.h>
#include <omp.h>
#define V 11          // Number of vertices in the graph
#define NUM_THREADS 4 // Number of threads to use
char vertexLabels[V] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K'};
int visited[V] = {0}; // Array to track visited vertices
int parent[V];        // Array to store the parent of each vertex during traversa
int graph[V][V] = {
    {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0},
    {0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0},
    {0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0},
};
void DFS(int v){
    visited[v] = 1;
    for (int i = 0; i < V; i++){
        if (graph[v][i] && !visited[i]){
            parent[i] = v;
            DFS(i);
        }
    }
}
int main(){
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for
    for (int v = 0; v < V; v++){
#pragma omp critical
        if (!visited[v]){
            DFS(v);
        }
    }
    printf("\nEdges of the spanning tree:\n");
    for (int i = 1; i < V; i++){
        printf("(%c, %c)\n", vertexLabels[parent[i]], vertexLabels[i]);
    }
    return 0;
```

Output:



Breadth First Search (BFS):

With the Race condition

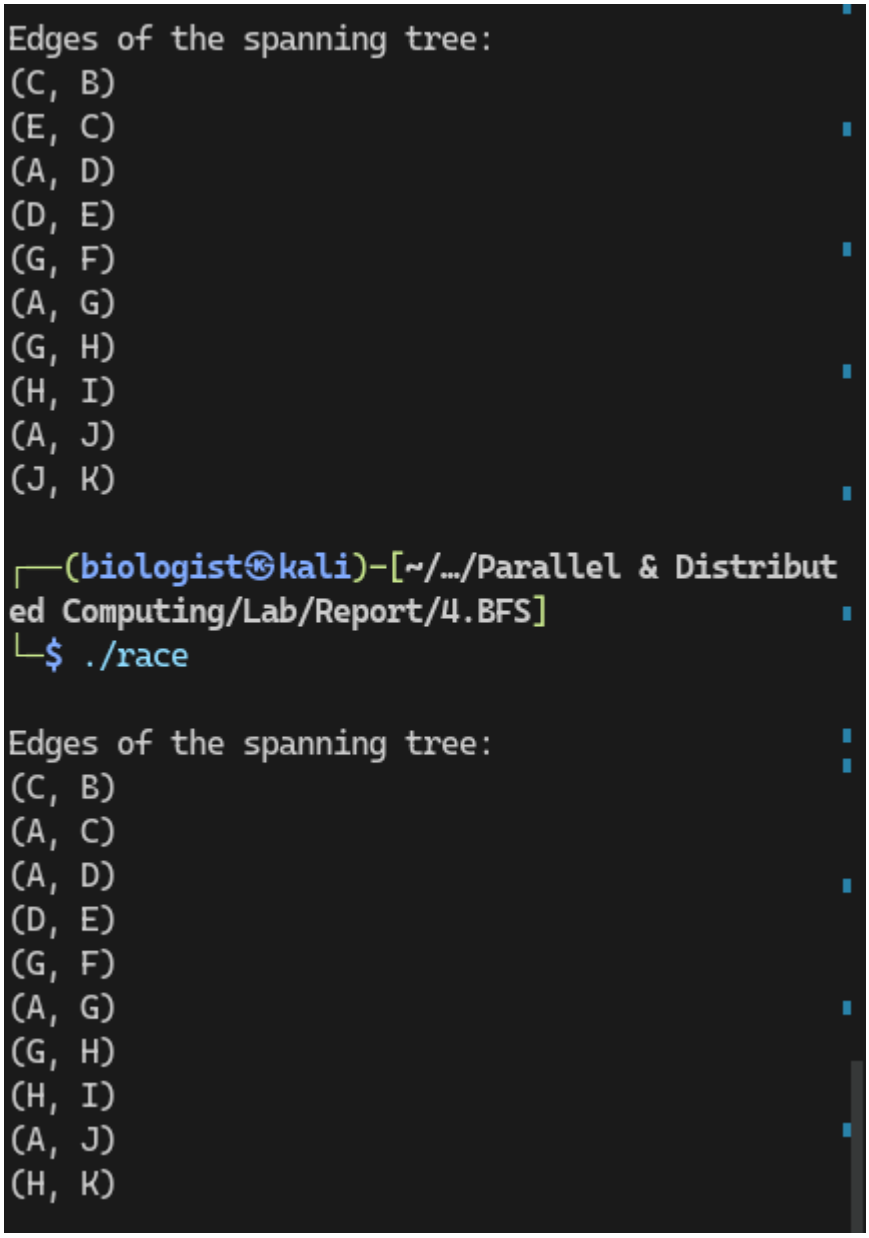
DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It's used to traverse or search for nodes in a graph or tree structure.

When race conditions occur, the results of these operations can become unpredictable due to unsynchronized access to shared data.

Code:

```c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#define V 11          // Number of vertices in the graph
#define NUM_THREADS 4 // Number of threads to use
char vertexLabels[V] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K'};
int visited[V] = {0}; // Array to track visited vertices
int parent[V];        // Array to store the parent of each vertex during traversal
int graph[V][V] = {
    {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0},
    {0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0},
    {0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0},
};
void BFS(int start_vertex)
{
    int queue[V];
    int front = 0, rear = 0;
    queue[rear++] = start_vertex;
    visited[start_vertex] = 1;
    while (front ≠ rear){
        int current_vertex = queue[front++];
        for (int i = 0; i < V; i++){
            if (graph[current_vertex][i] && !visited[i]){
                queue[rear++] = i;
                visited[i] = 1;
                parent[i] = current_vertex;
            }
        }
    }
}
int main(){
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for
    for (int v = 0; v < V; v++)
    {
        if (!visited[v]){
            BFS(v);
        }
    }
    printf("\nEdges of the spanning tree:\n");
    for (int i = 1; i < V; i++){
        printf("(%c, %c)\n", vertexLabels[parent[i]], vertexLabels[i]);
    }
    return 0;
}
```

output:



Without the Racing condition

Code:

```c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#define V 11           // Number of vertices in the graph
#define NUM_THREADS 4 // Number of threads to use
char vertexLabels[V] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K'};
int visited[V] = {0}; // Array to track visited vertices
int parent[V];        // Array to store the parent of each vertex during traversa
int graph[V][V] = {
    {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0},
    {0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0},
    {0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0},
};
void BFS(int start_vertex){
    int queue[V];
    int front = 0, rear = 0;
    queue[rear++] = start_vertex;
    visited[start_vertex] = 1;
    while (front != rear){
        int current_vertex = queue[front++];
        for (int i = 0; i < V; i++){
            if (graph[current_vertex][i] && !visited[i]){
                queue[rear++] = i;
                visited[i] = 1;
                parent[i] = current_vertex;
            }
        }
    }
}
int main(){
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for
    for (int v = 0; v < V; v++){
#pragma omp critical
        if (!visited[v]){
            BFS(v);
        }
    }
    printf("\nEdges of the spanning tree:\n");
    for (int i = 1; i < V; i++){
        printf("(%c, %c)\n", vertexLabels[parent[i]], vertexLabels[i]);
    }
    return 0;
}
```

Output:

```
Edges of the spanning tree:
(C, B)
(A, C)
(E, D)
(C, E)
(E, F)
(F, G)
(F, H)
(H, I)
(K, J)
(H, K)

┌──(biologist㉿kali)-[~/…/Parallel & Distributed Computing/Lab/Report/4.BFS]
└─$ ./Nrace

Edges of the spanning tree:
(C, B)
(A, C)
(E, D)
(C, E)
(E, F)
(F, G)
(F, H)
(H, I)
(K, J)
(H, K)
```