BFS traversal

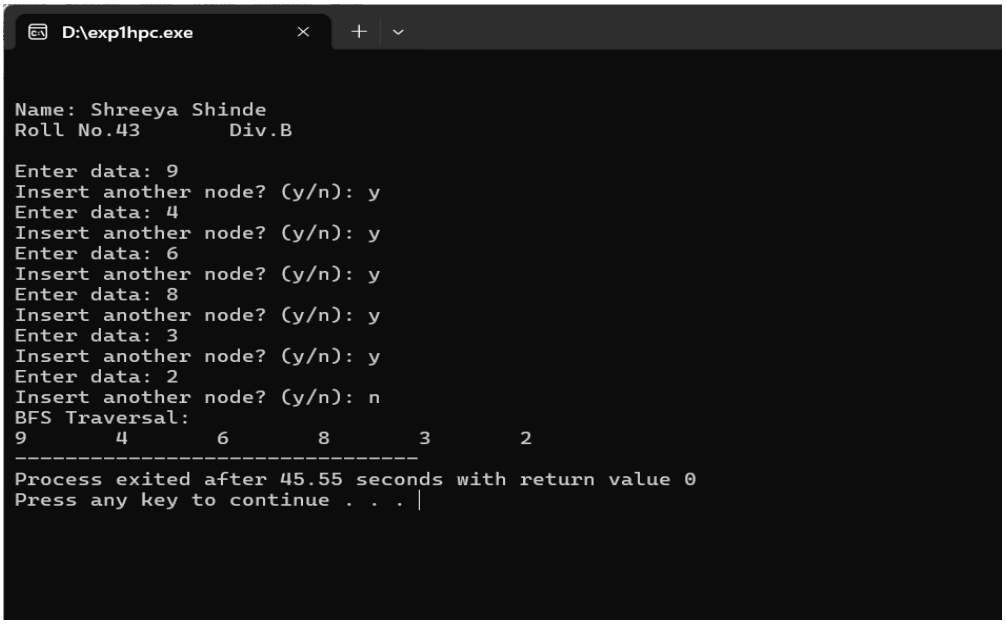
```
#include <iostream>
#include <queue>
#include <omp.h> // Required for OpenMP directives
using namespace std;

class Node
{
public:
   Node *left, *right;
   int data;
};

class BreadthFS
{
public:
   Node *insert(Node *root, int data); // Insert a node into the tree
   void bfs(Node *root);              // Perform parallel BFS
};

// Insert a new node using level-order insertion
Node *BreadthFS::insert(Node *root, int data)
{
   if (!root)
   {
      root = new Node;
```

```
    root->left = NULL;
    root->right = NULL;
    root->data = data;
    return root;
}

std::queue<Node *> q;
q.push(root);

while (!q.empty())
{
    Node *current = q.front();
    q.pop();

    if (!current->left)
    {
        current->left = new Node;
        current->left->left = NULL;
        current->left->right = NULL;
        current->left->data = data;
        return root;
    }
    else
    {
        q.push(current->left);
    }
```

```
        if (!current->right)
        {
            current->right = new Node;
            current->right->left = NULL;
            current->right->right = NULL;
            current->right->data = data;
            return root;
        }
        else
        {
            q.push(current->right);
        }
    }
    return root; // Ensures all control paths return a value
}

// Parallel BFS using OpenMP
void BreadthFS::bfs(Node *root)
{
    if (!root)
        return;

    queue<Node *> q;
    q.push(root);
```

```
   while (!q.empty())
   {
      int level_size = q.size();

#pragma omp parallel for // Parallelize processing of nodes at the current level
      for (int i = 0; i < level_size; i++)
      {
         Node *current = NULL;

#pragma omp critical // Thread-safe access to the queue
         {
            current = q.front();
            q.pop();
            cout << current->data << "\t";
         }

#pragma omp critical // Thread-safe insertion of children
         {
            if (current->left)
               q.push(current->left);
            if (current->right)
               q.push(current->right);
         }
      }
   }
}
```

```
int main()
{
    BreadthFS bfs;
    Node *root = NULL;
    int data;
    char choice;

    cout << "\n\nName: Shreeya Shinde\nRoll No.43 \t Div.B\n\n";

    do
    {
        cout << "Enter data: ";
        cin >> data;
        root = bfs.insert(root, data);
        cout << "Insert another node? (y/n): ";
        cin >> choice;
    } while (choice == 'y' || choice == 'Y');

    cout << "BFS Traversal:\n";
    bfs.bfs(root);

    return 0;
}
```


D:\exp1hpc.exe
Name: Shreeya Shinde
Roll No.43 Div.B
Enter data: 9
Insert another node? (y/n): y
Enter data: 4
Insert another node? (y/n): y
Enter data: 6
Insert another node? (y/n): y
Enter data: 8
Insert another node? (y/n): y
Enter data: 3
Insert another node? (y/n): y
Enter data: 2
Insert another node? (y/n): n
BFS Traversal:
9 4 6 8 3 2
Process exited after 45.55 seconds with return value 0
Press any key to continue . . .

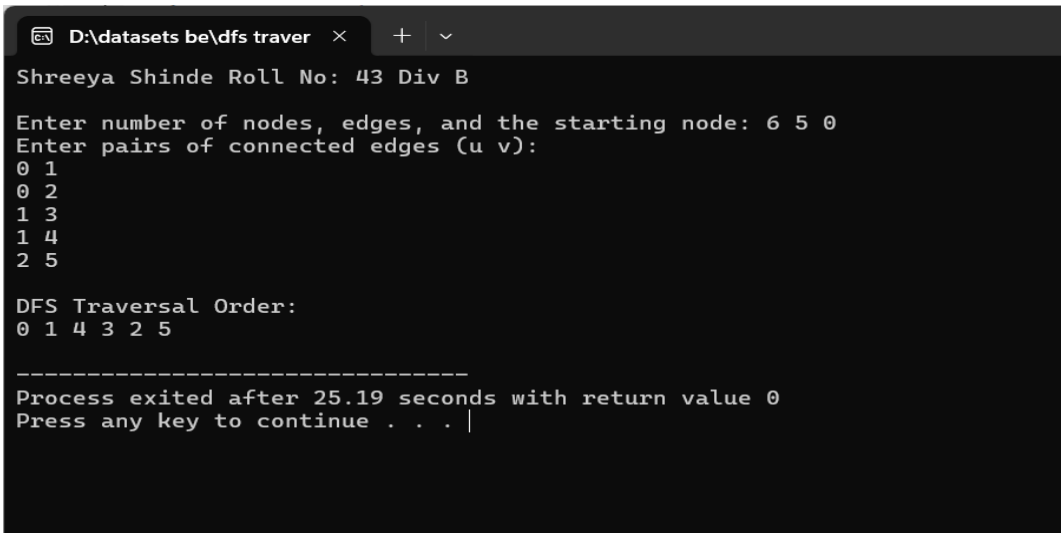
DFS traversal

```
#include <iostream>
#include <vector>
#include <stack>
#include <omp.h>
using namespace std;

const int MAX = 100000;
vector<int> graph[MAX];
bool visited[MAX];
omp_lock_t lock[MAX];

void dfs(int start_node)
{
   stack<int> s;
   s.push(start_node);
   while (!s.empty())
   {
      int curr_node = s.top();
      s.pop();

      omp_set_lock(&lock[curr_node]);
      if (!visited[curr_node])
      {
         visited[curr_node] = true;
         cout << curr_node << " ";
```

```
        }
        omp_unset_lock(&lock[curr_node]);

#pragma omp parallel for shared(s)
        for (int i = 0; i < graph[curr_node].size(); i++)
        {
            int adj_node = graph[curr_node][i];
            omp_set_lock(&lock[adj_node]);
            if (!visited[adj_node])
            {
#pragma omp critical
                {
                    s.push(adj_node);
                }
            }
            omp_unset_lock(&lock[adj_node]);
        }
    }
}

int main()
{
    // ?? Student Info
    cout << "Shreeya Shinde Roll No: 43 Div B\n\n";

    int n, m, start_node;
```

```
    cout << "Enter number of nodes, edges, and the starting node: ";
    cin >> n >> m >> start_node;

    cout << "Enter pairs of connected edges (u v):\n";
    for (int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

#pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
        omp_init_lock(&lock[i]);
    }

    cout << "\nDFS Traversal Order:\n";
    dfs(start_node);
    cout << endl;

    for (int i = 0; i < n; i++)
    {
        omp_destroy_lock(&lock[i]);
```

```
    }

    return 0;
}
```

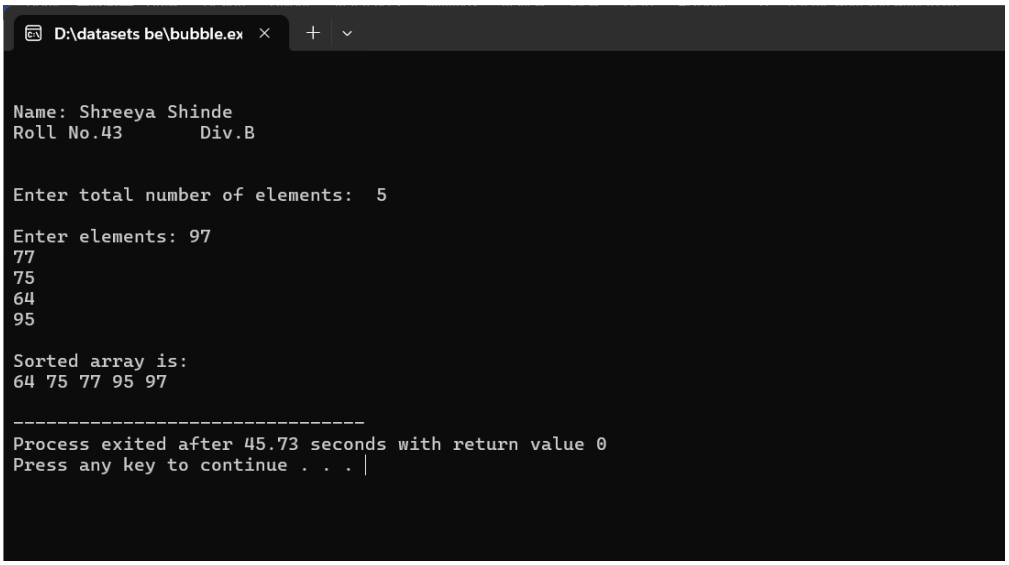

D:\datasets be\dfs traver
Shreeya Shinde Roll No: 43 Div B
Enter number of nodes, edges, and the starting node: 6 5 0
Enter pairs of connected edges (u v):
0 1
0 2
1 3
1 4
2 5
DFS Traversal Order:
0 1 4 3 2 5

Process exited after 25.19 seconds with return value 0
Press any key to continue . . .

Bubble sort

```
#include <iostream>
#include <cstdlib>
#include <omp.h>
using namespace std;
void bubble(int *, int);
void swap(int &, int &);
void bubble(int *a, int n)
{
   for (int i = 0; i < n; i++)
   {
      int first = i % 2;
#pragma omp parallel for shared(a, first)
      for (int j = first; j < n - 1; j += 2)
      {
         if (a[j] > a[j + 1])
         {
            swap(a[j], a[j + 1]);
         }
      }
   }
}
void swap(int &a, int &b)
{
   int temp;
   temp = a;
```

```
    a = b;
    b = temp;
}
int main()
{
    cout << "\n\nName: Shreeya Shinde\nRoll No.43 \t Div.B\n\n";
    int *a, n;
    cout << "\nEnter total number of elements: ";
    cin >> n;
    a = new int[n];
    cout << "\nEnter elements: ";
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }
    bubble(a, n);
    cout << "\nSorted array is:\n";
    for (int i = 0; i < n; i++) {
    cout << a[i] << " ";}
    cout << endl;
    delete[] a;
    return 0;
}
```


D:\datasets be\bubble.ex
Name: Shreeya Shinde
Roll No.43 Div.B
Enter total number of elements: 5
Enter elements: 97
77
75
64
95
Sorted array is:
64 75 77 95 97

Process exited after 45.73 seconds with return value 0
Press any key to continue . . .

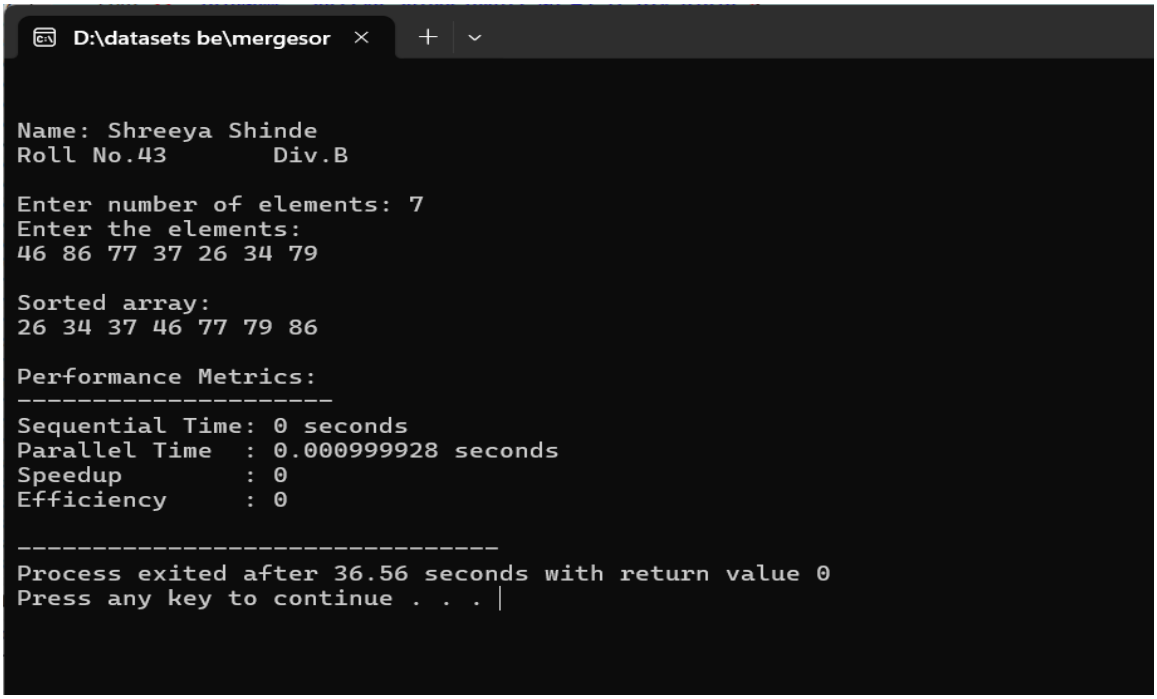
Merge Sort

```
#include <iostream>
#include <omp.h>
#include <vector>
using namespace std;
void merge(vector<int> &arr, int l, int m, int r)
{
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2)
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
}
void mergeSortSequential(vector<int> &arr, int l, int r)
{
    if (l < r)
    {
```

```
        int m = l + (r - l) / 2;
        mergeSortSequential(arr, l, m);
        mergeSortSequential(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
void mergeSortParallel(vector<int> &arr, int l, int r, int depth = 0)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;
        if (depth < 4)
        {
#pragma omp parallel sections
            {
#pragma omp section
                mergeSortParallel(arr, l, m, depth + 1);
#pragma omp section
                mergeSortParallel(arr, m + 1, r, depth + 1);
            }
        }
        else
        {
            mergeSortSequential(arr, l, m);
            mergeSortSequential(arr, m + 1, r);
        }
```

```
        merge(arr, l, m, r);
    }
}
int main()
{
    cout << "\n\nName: Shreeya Shinde\nRoll No.43 \t Div.B\n\n";
    int n;
    cout << "Enter number of elements: ";
    cin >> n;
    vector<int> arr(n), arrSeq(n);
    cout << "Enter the elements:\n";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    arrSeq = arr; // Copy input for sequential sort
    double start = omp_get_wtime();
    mergeSortSequential(arrSeq, 0, n - 1);
    double end = omp_get_wtime();
    double seqTime = end - start;
    start = omp_get_wtime();
    mergeSortParallel(arr, 0, n - 1);
    end = omp_get_wtime();
    double parTime = end - start;
    cout << "\nSorted array:\n";
    for (int i = 0; i < n; i++)
```

```
    cout << arr[i] << " ";
  cout << "\n";
  double speedup = seqTime / parTime;
  int numThreads = omp_get_max_threads();
  double efficiency = speedup / numThreads;
  cout << "\nPerformance Metrics:";
  cout << "\n---------------------";
  cout << "\nSequential Time: " << seqTime << " seconds";
  cout << "\nParallel Time  : " << parTime << " seconds";
  cout << "\nSpeedup        : " << speedup;
  cout << "\nEfficiency     : " << efficiency << endl;
  return 0;
}
```



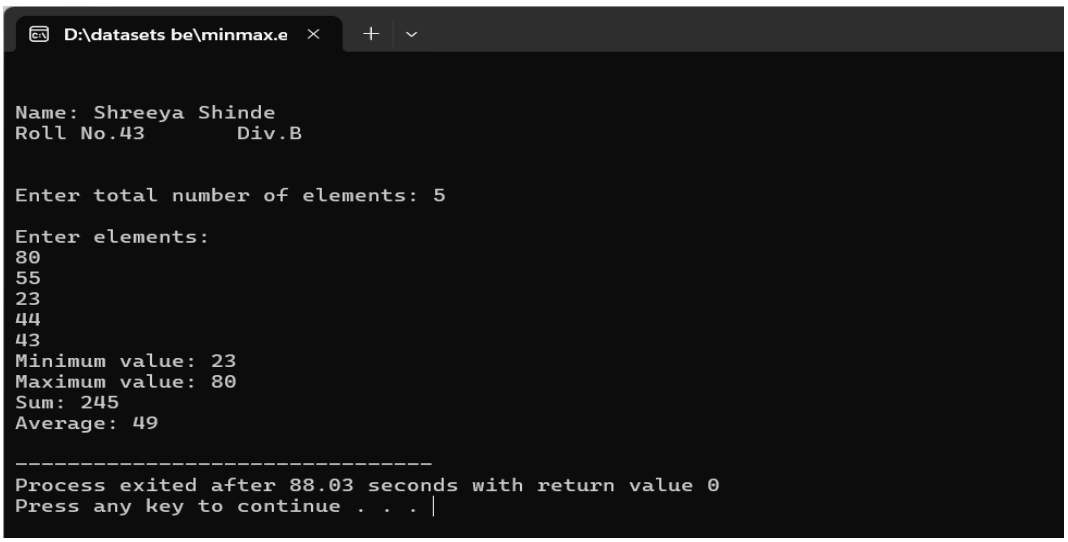
Min Max Sum Avg using parallel Reduction

```
#include <iostream>
#include <omp.h>
#include <climits>
using namespace std;
void min_reduction(int arr[], int n)
{
   int min_value = INT_MAX;
#pragma omp parallel for reduction(min : min_value)
   for (int i = 0; i < n; i++)
   {
      if (arr[i] < min_value)
      {
         min_value = arr[i];
      }
   }
   cout << "Minimum value: " << min_value << endl;
}
void max_reduction(int arr[], int n)
{
   int max_value = INT_MIN;
#pragma omp parallel for reduction(max : max_value)
   for (int i = 0; i < n; i++)
   {
      if (arr[i] > max_value)
      {
```

```
            max_value = arr[i];
        }
    }
    cout << "Maximum value: " << max_value << endl;
}
void sum_reduction(int arr[], int n)
{
    int sum = 0;
#pragma omp parallel for reduction(+ : sum)
    for (int i = 0; i < n; i++)
    {
        sum += arr[i];
    }
    cout << "Sum: " << sum << endl;
}
void average_reduction(int arr[], int n)
{
    if (n <= 1)
    {
        cout << "Average: Cannot calculate (array size too small)" << endl;
        return;
    }
    int sum = 0;
#pragma omp parallel for reduction(+ : sum)
    for (int i = 0; i < n; i++)
    {
```

```
        sum += arr[i];
    }
    cout << "Average: " << static_cast<double>(sum) / n << endl;
}
int main()
{
    cout << "\n\nName: Shreeya Shinde\nRoll No.43\t Div.B\n\n";
    int *arr, n;
    cout << "\nEnter total number of elements: ";
    cin >> n;
    if (n <= 0)
    {
        cerr << "Error: Array size must be positive" << endl;
        return 1;
    }
    arr = new int[n];
    cout << "\nEnter elements:\n";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    min_reduction(arr, n);
    max_reduction(arr, n);
    sum_reduction(arr, n);
    average_reduction(arr, n);
    delete[] arr;
```

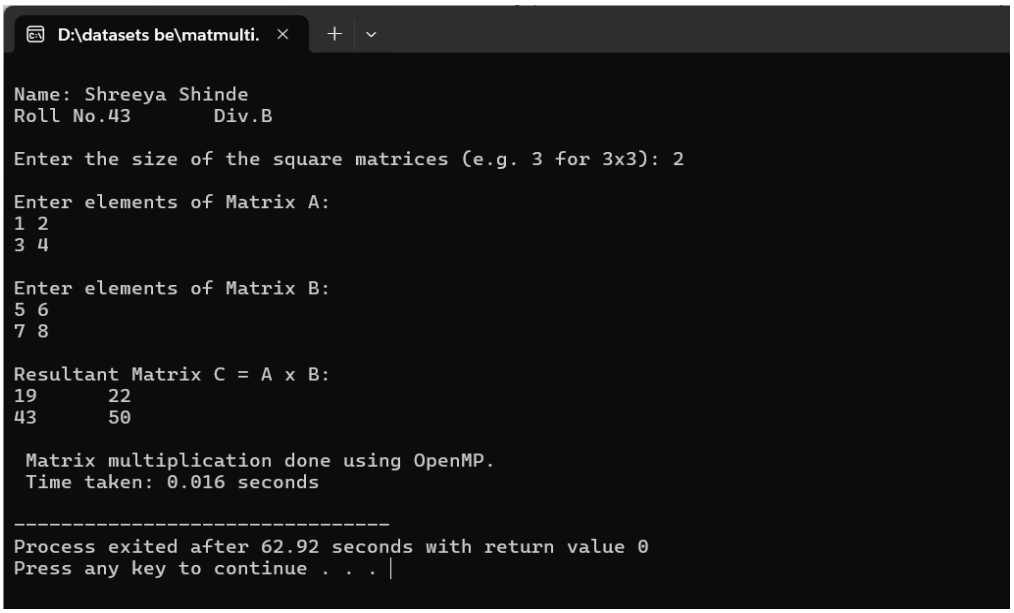
```
    return 0;
}
```


D:\datasets be\minmax.e
Name: Shreeya Shinde
Roll No.43 Div.B
Enter total number of elements: 5
Enter elements:
80
55
23
44
43
Minimum value: 23
Maximum value: 80
Sum: 245
Average: 49
Process exited after 88.03 seconds with return value 0
Press any key to continue . . .

Matrix Multiplication

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
   int n;
   cout << "\nName: Shreeya Shinde\nRoll No.43 \t Div.B\n";
   cout << "\nEnter the size of the square matrices (e.g. 3 for 3x3): ";
   cin >> n;
   float A[n][n], B[n][n], C[n][n];
   cout << "\nEnter elements of Matrix A:\n";
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         cin >> A[i][j];
   cout << "\nEnter elements of Matrix B:\n";
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         cin >> B[i][j];
#pragma omp parallel for collapse(2)
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         C[i][j] = 0;
   double start = omp_get_wtime();
#pragma omp parallel for collapse(2)
   for (int i = 0; i < n; i++)
```

```
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
  double end = omp_get_wtime();
  cout << "\nResultant Matrix C = A x B:\n";
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < n; j++)
        cout << C[i][j] << "\t";
    cout << endl;
  }
  cout << "\n Matrix multiplication done using OpenMP.";
  cout << "\n Time taken: " << end - start << " seconds\n";
  return 0;
}
```



HPC-mini project database query

```
#include <vector>
#include <iostream>
#include <windows.h> // Windows API for threading

struct Query
{
   int id;
   std::vector<int> conditions;
};

std::vector<int> data; // Global data
HANDLE mutex;
int total_matches = 0;

DWORD WINAPI thread_function(LPVOID lpParam);

struct ThreadData
{
   Query *query;
};

int main()
{
   std::cout << "\nName: Shreeya Shinde\nRoll No.43 \t Div.B\n";
   int n;
```

```
std::cout << "Enter total number of data elements: ";
std::cin >> n;
data.resize(n);
std::cout << "Enter data elements:\n";
for (int i = 0; i < n; i++)
{
   std::cin >> data[i];
}

int num_queries;
std::cout << "\nEnter number of queries: ";
std::cin >> num_queries;

std::vector<Query> queries(num_queries);
for (int i = 0; i < num_queries; i++)
{
   int k;
   std::cout << "Enter number of conditions for Query " << i + 1 << ": ";
   std::cin >> k;
   std::cout << "Enter " << k << " condition values:\n";
   queries[i].id = i;
   queries[i].conditions.resize(k);
   for (int j = 0; j < k; j++)
   {
      std::cin >> queries[i].conditions[j];
   }
```

```
}



// Create mutex
mutex = CreateMutex(NULL, FALSE, NULL);

// Create threads
std::vector<HANDLE> threads(num_queries);
std::vector<ThreadData> thread_data(num_queries);
for (int i = 0; i < num_queries; i++)
{
   thread_data[i].query = &queries[i];
   threads[i] = CreateThread(
      NULL, 0, thread_function, &thread_data[i], 0, NULL);
}

// Wait for all threads
WaitForMultipleObjects(num_queries, &threads[0], TRUE, INFINITE);

// Clean up
for (int i = 0; i < threads.size(); i++)
{
   CloseHandle(threads[i]);
}
CloseHandle(mutex);
```

```
    std::cout << "\n\nFinal total matches found: " << total_matches << std::endl;

    return 0;
}

DWORD WINAPI thread_function(LPVOID lpParam)
{
    ThreadData *data_ptr = (ThreadData *)lpParam;
    Query *query = data_ptr->query;

    int local_count = 0;
    for (int i = 0; i < data.size(); i++)
    {
        int record = data[i];
        for (int j = 0; j < query->conditions.size(); j++)
        {
            int cond = query->conditions[j];
            if (cond != 0 && record % cond == 0)
            {
                local_count++;
                break;
            }
        }
    }
```

```
    // Safely update global count
    WaitForSingleObject(mutex, INFINITE);
    std::cout << "Thread for Query " << query->id + 1 << " found " << local_count
              << " matches. Total so far: " << (total_matches + local_count) << std::endl;
    total_matches += local_count;
    ReleaseMutex(mutex);

    return 0;
}
```