1. **C++ Code (PA2_parallel_1.cpp)**

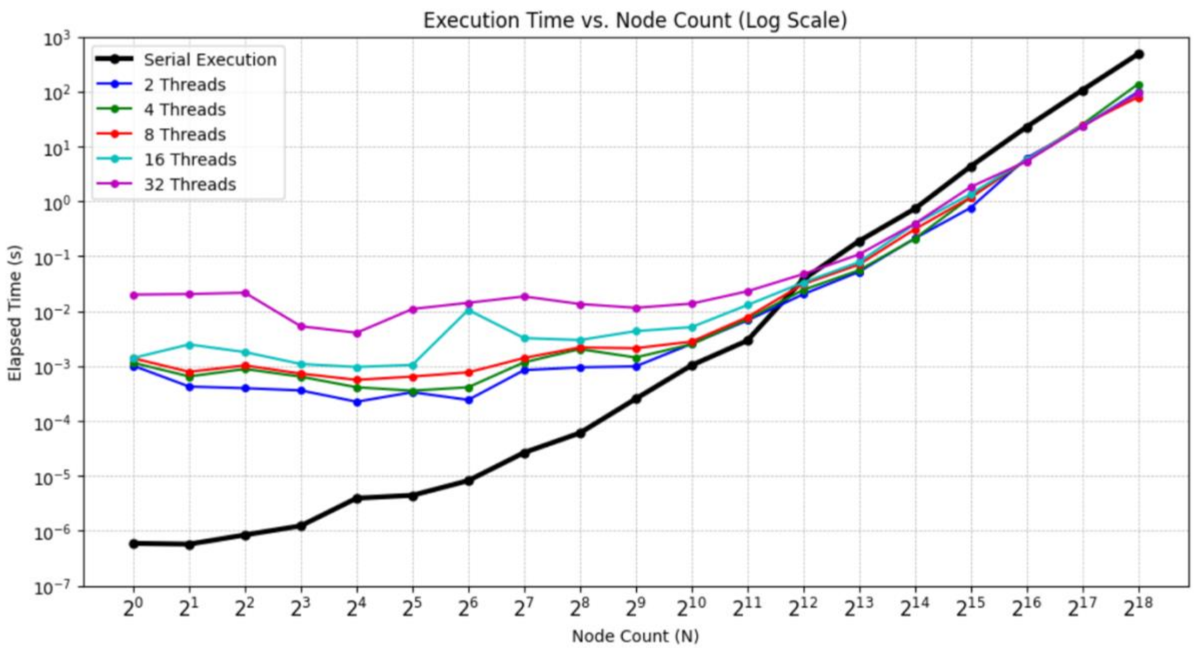
```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <vector>
#include <algorithm>
#include <omp.h>

using namespace std;

struct Node {
    int value;
    Node* next;
    Node* prev;
    omp_lock_t lock;
};

class LinkedList {
public:
    Node* head;
    omp_lock_t head_lock;

    LinkedList() {
        head = nullptr;
        omp_init_lock(&head_lock);
    }

    ~LinkedList() {
        omp_destroy_lock(&head_lock);
```

```
    }

    void insert(Node* newNode) {
        omp_set_lock(&head_lock);

        if (!head) {
            head = newNode;
            omp_unset_lock(&head_lock);
            return;
        }

        Node* prev = nullptr;
        Node* p = head;

        while (p && p->value < newNode->value) {
            prev = p;
            p = p->next;
        }

        newNode->next = p;
        newNode->prev = prev;

        if (prev) prev->next = newNode;
        else head = newNode;

        if (p) p->prev = newNode;

        omp_unset_lock(&head_lock);
    }
};
```

```
void insert_into_batch(vector<int> &local_values, int value) {
    local_values.push_back(value);
}

void merge_sorted_batch(LinkedList &global_list, vector<int> &sorted_values) {
    for (int val : sorted_values) {
        Node* newNode = new Node;
        newNode->value = val;
        newNode->next = nullptr;
        newNode->prev = nullptr;
        global_list.insert(newNode);
    }
}

int main(int argc, char* argv[]) {
    int N = stoi(argv[1]);
    int num_threads = stoi(argv[2]);

    LinkedList global_list;
    srand(time(nullptr));

    auto start = chrono::high_resolution_clock::now();

    vector<vector<int>> thread_local_data(num_threads);

    #pragma omp parallel num_threads(num_threads)
    {
        int thread_id = omp_get_thread_num();
        vector<int> local_values;
```

```
        #pragma omp for nowait
        for (int i = 0; i < N; i++) { #setting max value to N(number of nodes to prevent duplication)
            int value = rand() % N + 1;
            insert_into_batch(local_values, value);
        }


        sort(local_values.begin(), local_values.end());
        thread_local_data[thread_id] = move(local_values);
    }


    #pragma omp parallel for num_threads(num_threads)
    for (int i = 0; i < num_threads; i++) {
        merge_sorted_batch(global_list, thread_local_data[i]);
    }

    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed = end - start;

    cout << "Optimized Parallel Execution Time for N=" << N << " with " << num_threads
         << " threads: " << elapsed.count() << " sec\n";

    return 0;
}
```

2. **Plot of Execution Time vs Node Count**



3. **Based on your results, for which values of *n* do you observe speedup, and at which values of *N*?**

I start noticing speed up at 2^13 Node insertions for all parallel executions (2,4,8,16,32). Till 2^11 Node insertions, serial is quicker than any number of multi-threaded parallel execution. 2^12 thread is the break-even point for all nodes (similar run times for all threads – 1,2,4,8,16,32). As we move further to 2^18 Node insertions, parallel execution is considerably faster for all parallel threads by a considerable margin. It is faster ~5 times and the optimum thread count is 8 which takes 79.0464 seconds compared to 481.359 seconds for serial execution.

4. **Comment on your findings. In which circumstances is it beneficial to parallelize the insertion of data into a linked list?**

Serial should be the preferred option for node insertions up to 2^12 nodes. Any further order of node insertions beyond that point would be better suited for parallel execution.