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;

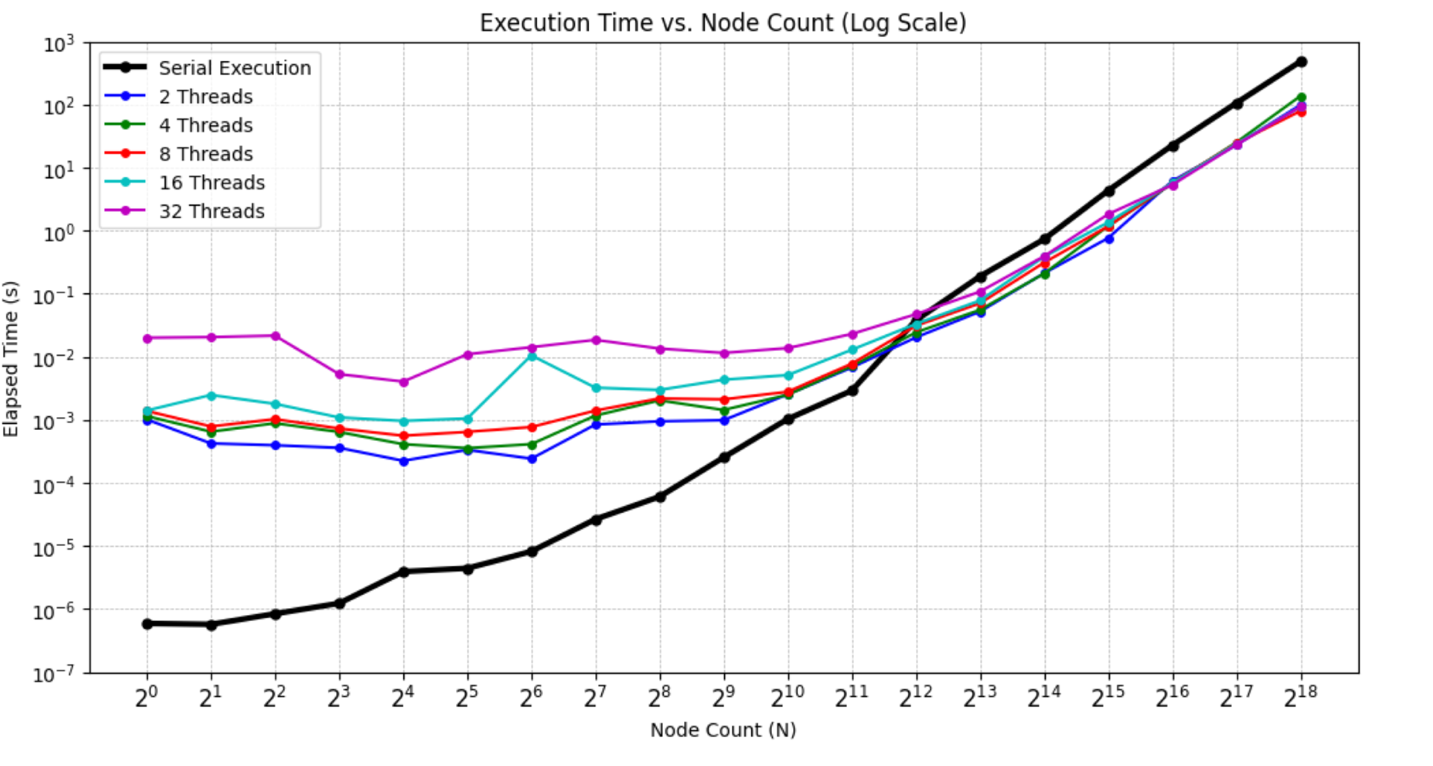
cout << "Optimized Parallel Execution Time for N=" << N << " with " << num\_threads

<< " threads: " << elapsed.count() << " sec\n";

return 0;

}

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



1. **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.

1. **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.