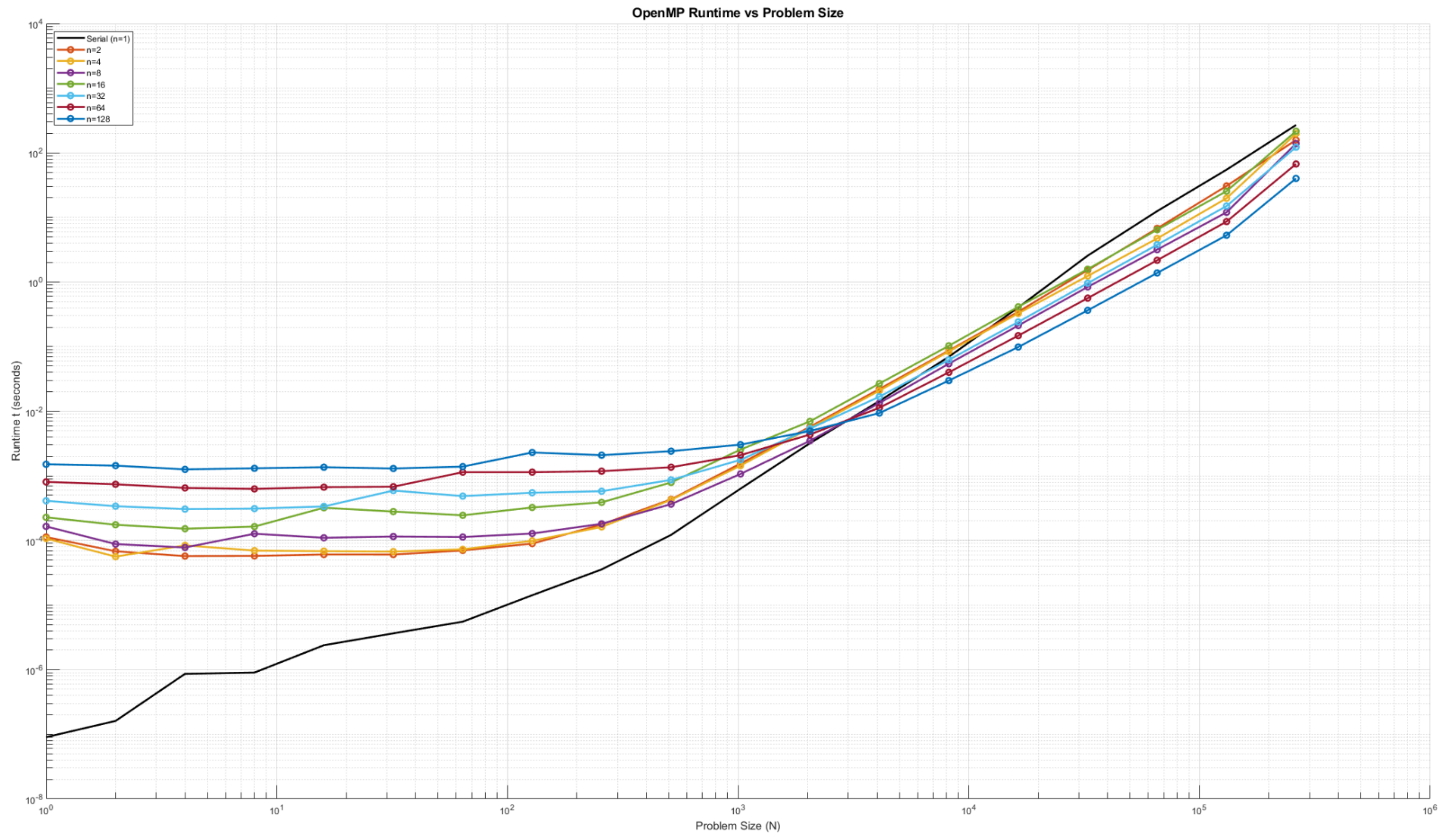
Brandon Nguyen, 827813045
COMPE596 - Spring 2026
2/6/2026

COMPE596 P02 - OpenMP Node Insertion Bidirectional Linked List

1) Runtime Plot



2) Code Listing

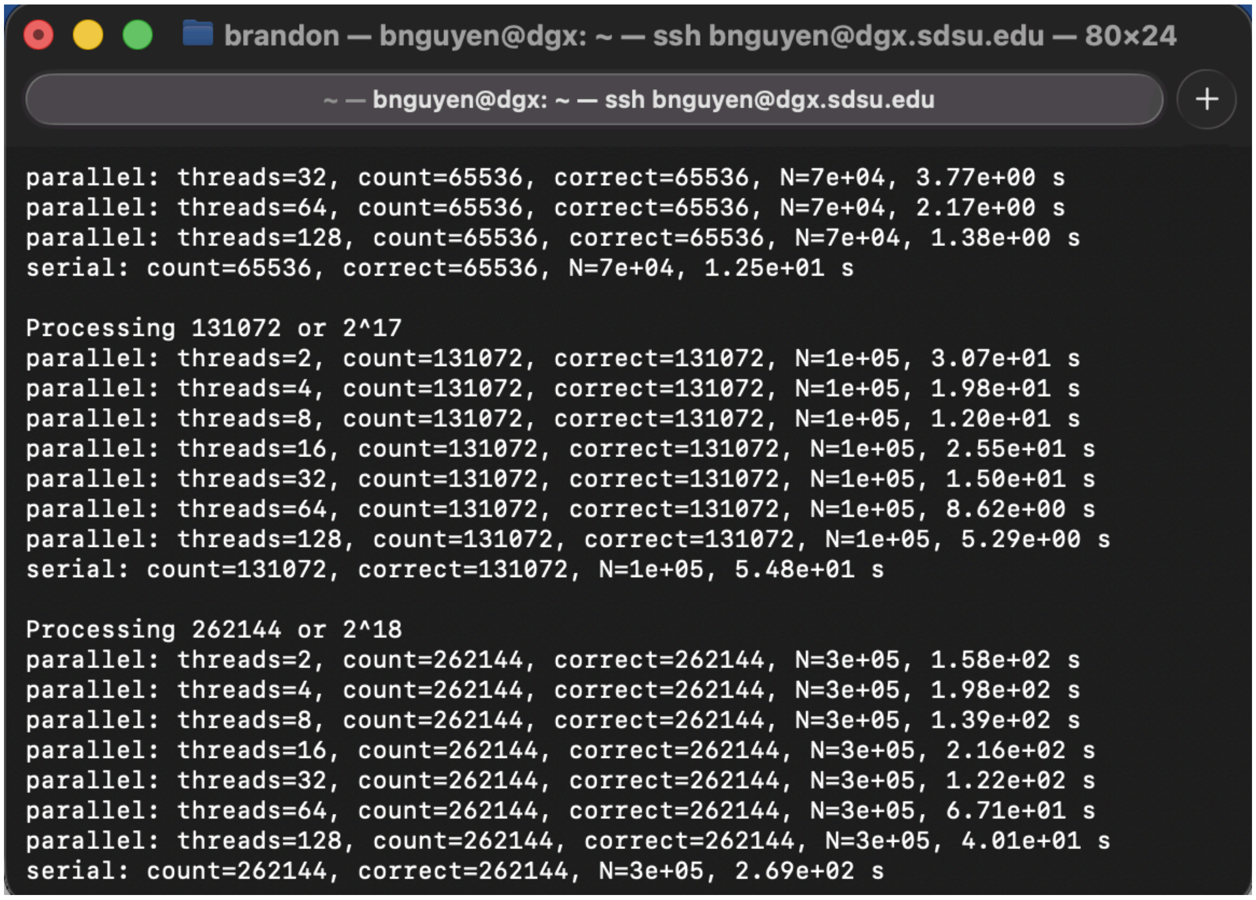
// Main Program

```
#include <omp.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>

typedef struct Node {
    int val;
    struct Node *prev;
    struct Node *next;
    omp_lock_t lock;
} Node;

static uint32_t lcg_step(uint32_t *state) {
    *state = (*state * 1664525u) + 1013904223u;
    return *state;
}

static Node *create_sentinel(void) {
    Node *head = (Node *)calloc(1, sizeof(Node));
    if (!head) { fprintf(stderr, "Failed to allocate sentinel\n"); exit(1); }
    head->val = INT_MIN;
    head->prev = NULL;
    head->next = NULL;
    omp_init_lock(&head->lock);
    return head;
}

static void free_list(Node *head) {
    Node *cur = head;
    while (cur) {
        Node *tmp = cur->next;
        omp_destroy_lock(&cur->lock);
        free(cur);
        cur = tmp;
    }
}

static int validate_sorted(Node *head, int *is_sorted_out) {
    int count = 0, sorted = 1;
    Node *cur = head->next;
    int last_val = INT_MIN;
    while (cur) {
        count++;
        if (cur->val < last_val) sorted = 0;
        last_val = cur->val;
        if (cur->next && cur->next->prev != cur) sorted = 0;
        cur = cur->next;
    }
    if (is_sorted_out) *is_sorted_out = sorted;
    return count;
}

static void insert_serial(Node *head, int value) {
    Node *prev = head;
    Node *cur = head->next;
    while (cur && cur->val < value) { prev = cur; cur = cur->next; }

    Node *new_node = (Node *)calloc(1, sizeof(Node));
    if (!new_node) { fprintf(stderr, "Failed to allocate node\n"); exit(1); }
    new_node->val = value;
    new_node->prev = prev;
    new_node->next = cur;
    omp_init_lock(&new_node->lock);

    prev->next = new_node;
    if (cur) cur->prev = new_node;
}

static void insert_parallel(Node *head, int value) {
    Node *prev = head;
    omp_set_lock(&prev->lock);

    Node *cur = prev->next;
    if (cur) omp_set_lock(&cur->lock);

    while (cur && cur->val < value) {
        omp_unset_lock(&prev->lock);
        prev = cur;
        cur = cur->next;
        if (cur) omp_set_lock(&cur->lock);
    }

    Node *new_node = (Node *)calloc(1, sizeof(Node));
    if (!new_node) { fprintf(stderr, "Failed to allocate node\n"); exit(1); }
    new_node->val = value;
    new_node->prev = prev;
    new_node->next = cur;
    omp_init_lock(&new_node->lock);

    prev->next = new_node;
    if (cur) cur->prev = new_node;

    if (cur) omp_unset_lock(&cur->lock);
    omp_unset_lock(&prev->lock);
}

static int parse_N(const char *s) {
    if (!s || !*s) return -1;

    if (strncmp(s, "2^", 2) == 0) {
        int k = atoi(s + 2);
        if (k < 0 || k > 30) return -1;
        return 1 << k;
    }

    char *end = NULL;
    double v = strtod(s, &end);
    if (end == s || v < 1.0 || v > 2147483647.0) return -1;
    return (int)(v + 0.5);
}

static int max_pow2(int x) {
    int p = 1;
    while ((p << 1) > 0 && (p << 1) <= x) p <<= 1;
    return p;
}

int main(int argc, char **argv) {
    int N = 1024;
    if (argc >= 2) {
        int parsed = parse_N(argv[1]);
        if (parsed < 1) { fprintf(stderr, "Usage: %s <N | 1eK | 2^K>\n", argv[0]); return 2; }
        N = parsed;
    }

    int max_threads = 8;
    const char *env = getenv("OMP_NUM_THREADS");
    if (env && *env) { int t = atoi(env); if (t >= 1) max_threads = t; }
    max_threads = max_pow2(max_threads);

    int *vals = (int *)malloc((size_t)N * sizeof(int));
    if (!vals) { fprintf(stderr, "malloc failed\n"); return 1; }
    uint32_t rng = 1u;
    for (int i = 0; i < N; i++) vals[i] = (int)(lcg_step(&rng) % 1000000u) + 1;


    for (int nthreads = 2; nthreads <= max_threads; nthreads <<= 1) {
        Node *head = create_sentinel();
        omp_set_num_threads(nthreads);

        double t0 = omp_get_wtime();
#pragma omp parallel for schedule(static)
        for (int i = 0; i < N; i++) insert_parallel(head, vals[i]);
        double t1 = omp_get_wtime();

        int sorted = 0;
        int count = validate_sorted(head, &sorted);
        int correct = (sorted && count == N) ? count : -1;
        printf("parallel: threads=%d, count=%d, correct=%d, N=%.0e, %.2e s\n",
               nthreads, count, correct, (double)N, (t1 - t0));

        free_list(head);
    }

    {
        Node *head = create_sentinel();
        double t0 = omp_get_wtime();
        for (int i = 0; i < N; i++) insert_serial(head, vals[i]);
        double t1 = omp_get_wtime();

        int sorted = 0;
        int count = validate_sorted(head, &sorted);
        int correct = (sorted && count == N) ? count : -1;
        printf("serial: count=%d, correct=%d, N=%.0e, %.2e s\n",
               count, correct, (double)N, (t1 - t0));
        free_list(head);
    }

    free(vals);
    return 0;
}
```

// Makefile

```
SHELL := bash
CC ?= gcc
CFLAGS ?= -O2 -std=c11 -Wall -Wextra
OMPFLAG ?= -fopenmp
PY ?= python3

.PHONY: all clean run

all: P02

p02: P02.c
	$(CC) $(CFLAGS) $(OMPFLAG) -o $@ $<

# Bash-driver sweep (N = 2^0 .. 2^18)
run: p02
	@export OMP_PLACES=threads; \
	export OMP_PROC_BIND=true; \
	export OMP_NUM_THREADS=128; \
	i=0; \
	while [ $$i -le 18 ]; do \
		N=$$((2**$$i)); \
		echo "Processing $$N or 2^$$i"; \
		./p02 $$N; \
		echo ""; \
		i=$$(($$i + 1)); \
	done

clean:
	rm -f P02
```

3) Program Output


brandon — bnguyen@dgx: ~ — ssh bnguyen@dgx.sdsu.edu — 80×24
~ — bnguyen@dgx: ~ — ssh bnguyen@dgx.sdsu.edu

```
parallel: threads=32, count=65536, correct=65536, N=7e+04, 3.77e+00 s
parallel: threads=64, count=65536, correct=65536, N=7e+04, 2.17e+00 s
parallel: threads=128, count=65536, correct=65536, N=7e+04, 1.38e+00 s
serial: count=65536, correct=65536, N=7e+04, 1.25e+01 s

Processing 131072 or 2^17
parallel: threads=2, count=131072, correct=131072, N=1e+05, 3.07e+01 s
parallel: threads=4, count=131072, correct=131072, N=1e+05, 1.98e+01 s
parallel: threads=8, count=131072, correct=131072, N=1e+05, 1.20e+01 s
parallel: threads=16, count=131072, correct=131072, N=1e+05, 2.55e+01 s
parallel: threads=32, count=131072, correct=131072, N=1e+05, 1.50e+01 s
parallel: threads=64, count=131072, correct=131072, N=1e+05, 8.62e+00 s
parallel: threads=128, count=131072, correct=131072, N=1e+05, 5.29e+00 s
serial: count=131072, correct=131072, N=1e+05, 5.48e+01 s

Processing 262144 or 2^18
parallel: threads=2, count=262144, correct=262144, N=3e+05, 1.58e+02 s
parallel: threads=4, count=262144, correct=262144, N=3e+05, 1.98e+02 s
parallel: threads=8, count=262144, correct=262144, N=3e+05, 1.39e+02 s
parallel: threads=16, count=262144, correct=262144, N=3e+05, 2.16e+02 s
parallel: threads=32, count=262144, correct=262144, N=3e+05, 1.22e+02 s
parallel: threads=64, count=262144, correct=262144, N=3e+05, 6.71e+01 s
parallel: threads=128, count=262144, correct=262144, N=3e+05, 4.01e+01 s
serial: count=262144, correct=262144, N=3e+05, 2.69e+02 s
```

4) MATLAB Plot Script

```
close all
clc
data = load('data.txt');
N      = data(:,1);
serial = data(:,2);
t2     = data(:,3);
t4     = data(:,4);
t8     = data(:,5);
t16    = data(:,6);
t32    = data(:,7);
t64    = data(:,8);
t128   = data(:,9);
figure;
hold on;

plot(N, serial, 'k-', 'LineWidth', 2);
plot(N, t2,   '-o', 'LineWidth', 2);
plot(N, t4,   '-o', 'LineWidth', 2);
plot(N, t8,   '-o', 'LineWidth', 2);
plot(N, t16,  '-o', 'LineWidth', 2);
plot(N, t32,  '-o', 'LineWidth', 2);
plot(N, t64,  '-o', 'LineWidth', 2);
plot(N, t128, '-o', 'LineWidth', 2);

set(gca, 'XScale', 'log');
set(gca, 'YScale', 'log');

xlabel('Problem Size (N)', 'FontSize', 12);
ylabel('Runtime t (seconds)', 'FontSize', 12);
title('OpenMP Runtime vs Problem Size', 'FontSize', 14);

legend('Serial (n=1)', ...
       'n=2', 'n=4', 'n=8', 'n=16', 'n=32', 'n=64', 'n=128', ...
       'Location', 'northwest');
grid on;
set(gcf,'Color','white');
hold off;
```

5) Discussion

In P02, we are tasked with writing a program using the DGX server to observe OpenMP locking to synchronize the parallel in-order insertion of Node structures into a bidirectional linked list. Once again, there is to be a serial and parallel implementation of the program, preferably together in a single program this time. For this programming assignment, we are also supposed to use a bash script driver in Makefile to help run the main program. Lastly, we are then required to plot our terminal output results in MATLAB via a data text file in order to precisely analyze and compare between runtime and Node count. After implementing the Bidirectional Linked List program, we observe node insertion based from 2^0 to 2^18. According to my findings in the graphs above and my device’s CPU specs, it appears that parallel performance improves significantly only when the node count becomes large. As node count increases, the parallel implementation shows speedup. The optimal performance happens at higher thread counts like 64 and 128. Overall, this second programming assignment was insightful in terms of learning methods to efficiently run our program using a Makefile while also working with bidirectional linked lists.