Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

Группа: М8О-212БВ-24

Студент: Колосов Е.Е.

Преподаватель: Бахарев В.Д.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 10.17.25

Москва, 2025

**Постановка задачи**

**Вариант 1.**

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы.

Вариант: отсортировать массив целых чисел при помощи битонической сортировки.

**Общий метод и алгоритм решения**

**Общий метод:**

Реализовать параллельную битоническую сортировку с использованием POSIX потоков (pthread), ограничивая количество активных потоков, и сравнить её производительность с последовательной версией.

**Алгоритм решения:**

1. Реализовать последовательный алгоритм битонической сортировки.
2. Создать функцию (bitonic\_sort\_thread), которая будет работать в отдельном потоке.
3. В этой функции: разделить массив, создать потоки для сортировки половин (pthread\_create), дождаться их (pthread\_join), выполнить слияние (bitonic\_merge).
4. Добавить проверку количества активных потоков (с помощью mutex) и выполнить оставшуюся работу в текущем потоке, если лимит превышен.
5. В main: выделить память (mmap), заполнить массив, замерить время (clock\_gettime) последовательной сортировки, запустить параллельную сортировку в одном потоке, замерить её время, вычислить ускорение и эффективность.
6. Вывести результаты (write).
7. Запустить программу с разным количеством потоков и собрать данные для таблицы.

**Код программы**

#include <pthread.h>

#include <unistd.h>

#include <sys/mman.h>

#include <time.h>

#include <limits.h>

#define MY\_MALLOC(size) mmap(NULL, size, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0)

#define MY\_FREE(ptr, size) munmap(ptr, size)

static struct timespec get\_time\_counter() { struct timespec ts; clock\_gettime(CLOCK\_MONOTONIC, &ts); return ts; }

static double elapsed\_ms(struct timespec start, struct timespec end) {

long seconds = end.tv\_sec - start.tv\_sec;

long nanoseconds = end.tv\_nsec - start.tv\_nsec;

return (double)seconds \* 1000.0 + (double)nanoseconds / 1000000.0;

}

#define sleep\_ms(ms) usleep((ms) \* 1000)

typedef struct { int id; int\* arr; int lo; int n; int dir; int max\_threads; } thread\_data\_t;

int active\_threads = 0;

pthread\_mutex\_t thread\_count\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void bitonic\_compare(int\* arr, int idx1, int idx2, int dir) {

if ((arr[idx1] > arr[idx2]) == dir) {

swap(&arr[idx1], &arr[idx2]);

}

}

void bitonic\_merge(int\* arr, int lo, int n, int dir) {

if (n <= 1) return;

int m = 1;

while (m < n) m <<= 1;

m >>= 1;

for (int i = lo; i < lo + n - m; i++) {

bitonic\_compare(arr, i, i + m, dir);

}

bitonic\_merge(arr, lo, m, dir);

bitonic\_merge(arr, lo + m, n - m, dir);

}

void bitonic\_sort\_seq(int\* arr, int lo, int n, int dir) {

if (n <= 1) return;

int m = n / 2;

bitonic\_sort\_seq(arr, lo, m, 1);

bitonic\_sort\_seq(arr, lo + m, n - m, 0);

bitonic\_merge(arr, lo, n, dir);

}

void\* merge\_sort\_thread(void\* arg) {

thread\_data\_t\* data = (thread\_data\_t\*)arg;

bitonic\_merge(data->arr, data->lo, data->n, data->dir);

pthread\_mutex\_lock(&thread\_count\_mutex);

active\_threads--;

pthread\_mutex\_unlock(&thread\_count\_mutex);

return NULL;

}

void\* bitonic\_sort\_thread(void\* arg) {

thread\_data\_t\* data = (thread\_data\_t\*)arg;

if (data->n <= 1) {

pthread\_mutex\_lock(&thread\_count\_mutex);

active\_threads--;

pthread\_mutex\_unlock(&thread\_count\_mutex);

return NULL;

}

int m = data->n / 2;

pthread\_t t1, t2;

thread\_data\_t d1 = {0, data->arr, data->lo, m, 1, data->max\_threads};

thread\_data\_t d2 = {0, data->arr, data->lo + m, data->n - m, 0, data->max\_threads};

pthread\_mutex\_lock(&thread\_count\_mutex);

if (active\_threads >= data->max\_threads) {

pthread\_mutex\_unlock(&thread\_count\_mutex);

bitonic\_sort\_seq(data->arr, data->lo, m, 1);

bitonic\_sort\_seq(data->arr, data->lo + m, data->n - m, 0);

} else {

active\_threads += 2;

pthread\_mutex\_unlock(&thread\_count\_mutex);

pthread\_create(&t1, NULL, bitonic\_sort\_thread, &d1);

pthread\_create(&t2, NULL, bitonic\_sort\_thread, &d2);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

}

pthread\_t t3;

thread\_data\_t d3 = {0, data->arr, data->lo, data->n, data->dir, data->max\_threads};

pthread\_mutex\_lock(&thread\_count\_mutex);

if (active\_threads < data->max\_threads) {

active\_threads++;

pthread\_mutex\_unlock(&thread\_count\_mutex);

pthread\_create(&t3, NULL, merge\_sort\_thread, &d3);

pthread\_join(t3, NULL);

} else {

pthread\_mutex\_unlock(&thread\_count\_mutex);

bitonic\_merge(data->arr, data->lo, data->n, data->dir);

}

pthread\_mutex\_lock(&thread\_count\_mutex);

active\_threads--;

pthread\_mutex\_unlock(&thread\_count\_mutex);

return NULL;

}

static unsigned int seed = 12345;

void generate\_random\_array(int\* arr, int n) {

for (int i = 0; i < n; i++) {

seed = seed \* 1103515245 + 12345;

arr[i] = (seed >> 16) % 10000;

}

}

void my\_memcpy(void\* dest, const void\* src, size\_t n) {

char\* d = (char\*)dest;

const char\* s = (const char\*)src;

for (size\_t i = 0; i < n; i++) {

d[i] = s[i];

}

}

void\* my\_malloc(size\_t size) {

return MY\_MALLOC(size);

}

void my\_free(void\* ptr, size\_t size) {

if (ptr) MY\_FREE(ptr, size);

}

int int\_to\_str(long num, char\* buffer, int buffer\_size) {

if (buffer\_size <= 0) return 0;

if (num == 0) {

if (buffer\_size > 1) {

buffer[0] = '0';

buffer[1] = '\0';

return 1;

} else {

return 0;

}

}

int sign = 0;

if (num < 0) {

sign = 1;

if (num == LONG\_MIN) {

const char\* min\_str = "-9223372036854775808";

int len = 0;

while (min\_str[len] != '\0' && len < buffer\_size - 1) {

buffer[len] = min\_str[len];

len++;

}

if (len < buffer\_size) {

buffer[len] = '\0';

return len;

} else {

return 0;

}

}

num = -num;

}

char temp[21];

int temp\_len = 0;

while (num > 0 && temp\_len < 20) {

temp[temp\_len++] = (num % 10) + '0';

num /= 10;

}

int start = 0;

if (sign && buffer\_size > temp\_len + 1) {

buffer[0] = '-';

start = 1;

} else if (sign) {

return 0;

}

if (buffer\_size <= start + temp\_len) {

return 0;

}

for (int i = 0; i < temp\_len; i++) {

buffer[start + i] = temp[temp\_len - 1 - i];

}

buffer[start + temp\_len] = '\0';

return start + temp\_len;

}

int double\_to\_str(double num, char\* buffer, int buffer\_size) {

if (buffer\_size <= 0) return 0;

long long\_num\_part = (long)num;

if (num < 0 && num != (double)long\_num\_part) {

long\_num\_part--;

}

double frac\_part = num - long\_num\_part;

char int\_buf[25];

int int\_len = int\_to\_str(long\_num\_part, int\_buf, 25);

if (int\_len <= 0) return 0;

long frac\_as\_int = (long)(frac\_part \* 100 + (frac\_part >= 0 ? 0.5 : -0.5));

if (frac\_as\_int < 0) frac\_as\_int = -frac\_as\_int;

char frac\_buf[5];

frac\_buf[0] = '.';

int\_to\_str(frac\_as\_int, frac\_buf + 1, 4);

int total\_len = int\_len + 3;

if (buffer\_size <= total\_len) {

return 0;

}

for (int i = 0; i < int\_len; i++) {

buffer[i] = int\_buf[i];

}

for (int i = 0; i < 3; i++) {

buffer[int\_len + i] = frac\_buf[i];

}

buffer[total\_len] = '\0';

return total\_len;

}

void print\_string(const char\* str) {

int len = 0;

while (str[len] != '\0') len++;

write(STDOUT\_FILENO, str, len);

}

void println\_string(const char\* str) {

print\_string(str);

write(STDOUT\_FILENO, "\n", 1);

}

void print\_int(int num) {

char buf[12];

if (int\_to\_str(num, buf, 12) > 0) {

print\_string(buf);

}

}

void print\_double(double num) {

char buf[30];

if (double\_to\_str(num, buf, 30) > 0) {

print\_string(buf);

}

}

int main(int argc, char\* argv[]) {

if (argc != 3) {

return 1;

}

int num\_threads = 0;

char\* num\_str = argv[2];

while (\*num\_str >= '0' && \*num\_str <= '9') {

num\_threads = num\_threads \* 10 + (\*num\_str - '0');

num\_str++;

}

int n = 1000000;

int\* arr = (int\*)my\_malloc(n \* sizeof(int));

if (!arr) return 1;

generate\_random\_array(arr, n);

int\* seq\_arr = (int\*)my\_malloc(n \* sizeof(int));

if (!seq\_arr) { my\_free(arr, n \* sizeof(int)); return 1; }

my\_memcpy(seq\_arr, arr, n \* sizeof(int));

struct timespec start = get\_time\_counter();

bitonic\_sort\_seq(seq\_arr, 0, n, 1);

struct timespec end = get\_time\_counter();

double seq\_time = elapsed\_ms(start, end);

int\* par\_arr = (int\*)my\_malloc(n \* sizeof(int));

if (!par\_arr) { my\_free(arr, n \* sizeof(int)); my\_free(seq\_arr, n \* sizeof(int)); return 1; }

my\_memcpy(par\_arr, arr, n \* sizeof(int));

active\_threads = 1;

pthread\_t main\_thread;

thread\_data\_t data = {0, par\_arr, 0, n, 1, num\_threads};

start = get\_time\_counter();

pthread\_create(&main\_thread, NULL, bitonic\_sort\_thread, &data);

pthread\_join(main\_thread, NULL);

end = get\_time\_counter();

double par\_time = elapsed\_ms(start, end);

double speedup = seq\_time / par\_time;

double efficiency = speedup / num\_threads;

print\_string("Sequential time: ");

print\_double(seq\_time);

println\_string(" ms");

print\_string("Parallel time with ");

print\_int(num\_threads);

print\_string(" threads: ");

print\_double(par\_time);

println\_string(" ms");

print\_string("Speedup: ");

print\_double(speedup);

println\_string("");

print\_string("Efficiency: ");

print\_double(efficiency);

println\_string("");

my\_free(arr, n \* sizeof(int));

my\_free(seq\_arr, n \* sizeof(int));

my\_free(par\_arr, n \* sizeof(int));

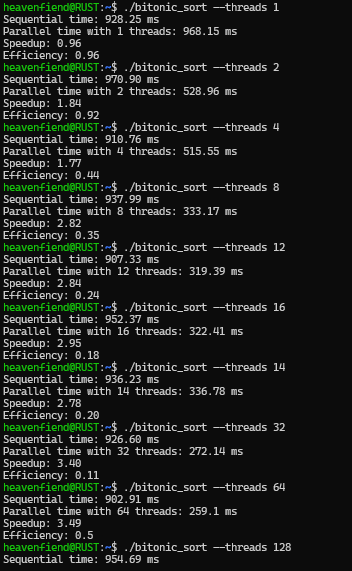
return 0;

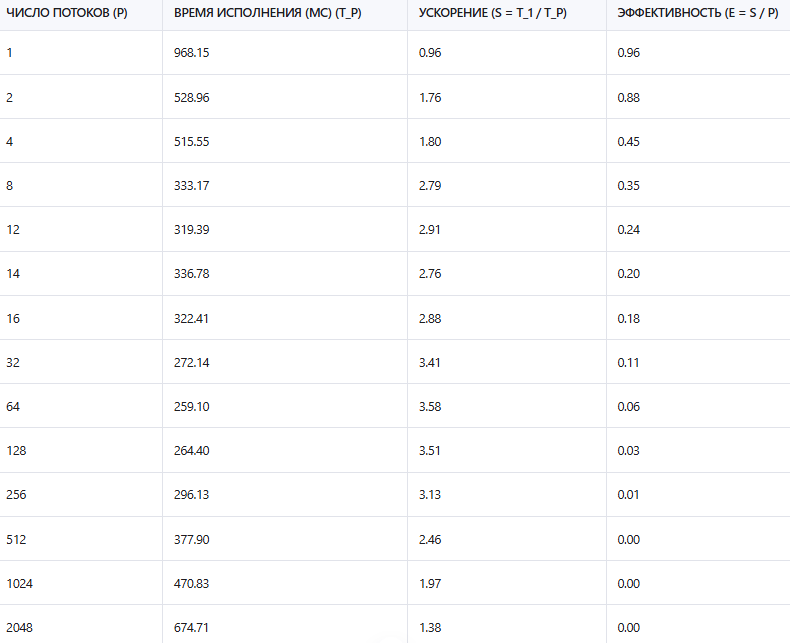
}

**Протокол работы программы**

gcc -pthread -o bitonic\_sort main.c

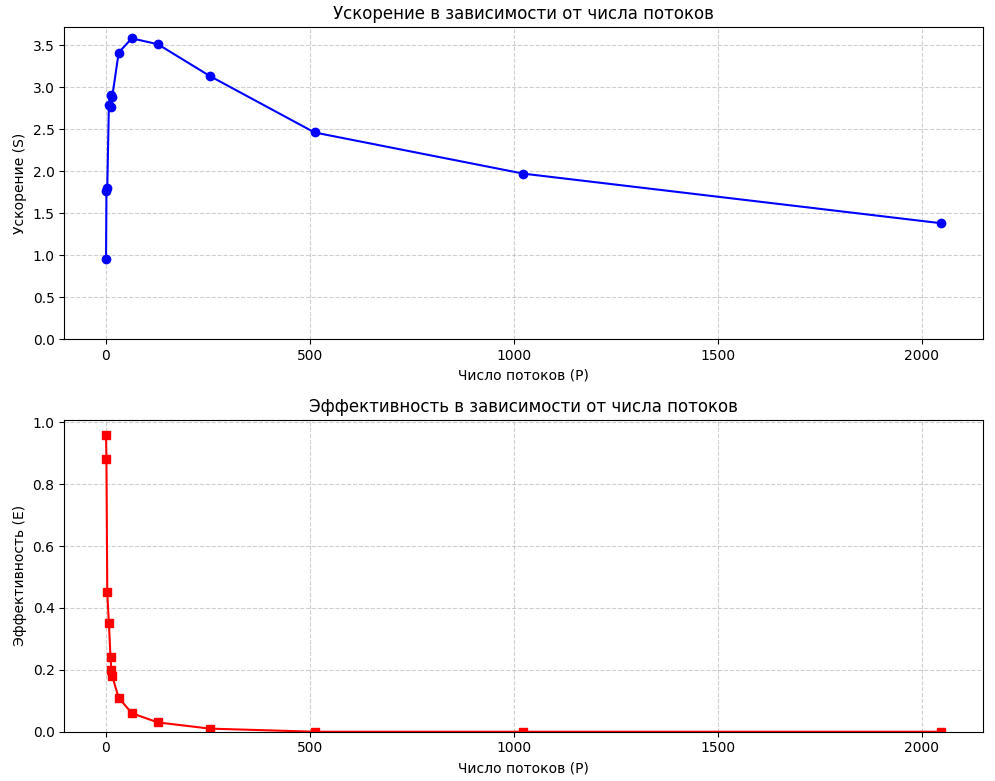
./bitonic\_sort --threads (потоки)





**Влияние количества потоков на производительность**

1. Фаза роста производительности (P от 1 до ~16):
   * Поведение: При увеличении числа потоков с 1 до 16 (или около того) время выполнения (T\_P) значительно снижается, а ускорение (S) растёт. Эффективность (E) также высока на начальных этапах (например, E близка к 0.9 при 2 потоках).
   * Обоснование: В этой фазе алгоритм эффективно использует дополнительные ядра процессора. Битоническая сортировка рекурсивно делит задачу, и параллельное выполнение сортировки отдельных частей массива позволяет задействовать вычислительные ресурсы. Накладные расходы на создание и синхронизацию потоков ещё невелики по сравнению с полезной работой. Ускорение близко к линейному (S примерно пропорционально P) на начальных шагах.
2. Фаза насыщения и стабилизации ускорения (P ~16 до ~32-64):
   * Поведение: Рост ускорения замедляется. Время выполнения продолжает уменьшаться, но не так резко. Эффективность (E) начинает падать, так как S растёт медленнее, чем P.
   * Обоснование: Количество потоков приближается к количеству логических ядер процессора. Возникает конкуренция за ресурсы (ЦП, кэш, память). Накладные расходы на синхронизацию (mutex) и планирование ОС начинают играть бóльшую роль. Ускорение не пропорционально увеличению потоков, эффективность падает.
3. Фаза деградации производительности (P >> количество ядер, например, 128, 512, 1024 и т.д.):
   * Поведение: Ускорение (S) начинает снижаться, а время выполнения (T\_P) может увеличиваться. Эффективность (E) стремится к 0.
   * Обоснование: Количество потоков значительно превышает количество доступных ядер. Процессор вынужден часто переключаться между огромным количеством потоков (context switching), что требует значительных ресурсов. Накладные расходы на синхронизацию (mutex) и управление потоками становятся доминирующими, превышая выгоду от параллельных вычислений. Система тратит больше времени на "организационные" задачи, чем на полезную работу.



**Вывод**

Параллельная битоническая сортировка показывает улучшение производительности при увеличении числа потоков до определённого порога (примерно 32-64 потока), после которого ускорение начинает снижаться. Максимальное ускорение (~3.6) достигается при умеренном количестве потоков, а эффективность (S/P) падает с ростом P, стремясь к 0 при очень большом числе потоков. Это объясняется ростом накладных расходов на синхронизацию и переключение контекста. Таким образом, оптимальное количество потоков для данной реализации и системы лежит в диапазоне 16-64, обеспечивая баланс между ускорением и эффективностью.