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**

**2.1.** Битоническая сортировка

* Это алгоритм сортировки, работающий на основе битонических последовательностей.
* Алгоритм делит массив на две половины, сортирует одну по возрастанию, другую — по убыванию.
* Затем сливает их в одну отсортированную последовательность.
* Алгоритм рекурсивен и может быть распараллелен на этапах сортировки подмассивов и слияния.

**2.2.** Последовательная версия

* Реализована рекурсивная функция bitonic\_sort\_seq().
* Использует bitonic\_merge() для слияния битонических последовательностей.

**2.3.** Многопоточная версия

* Использует POSIX Threads или Windows API.
* Каждый рекурсивный вызов может порождать новый поток (если active\_threads < max\_threads).
* Используется mutex для синхронизации доступа к счётчику активных потоков.
* Для слияния (bitonic\_merge) также может создаваться отдельный поток.

**2.4.** Ограничение числа потоков

* Введена переменная active\_threads, защищённая mutex.
* При превышении лимита (max\_threads) дальнейшие вызовы выполняются последовательно.

**Код программы**

**main.c \***

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#ifdef \_WIN32

#include <windows.h>

#include <process.h>

#define pthread\_t HANDLE

#define pthread\_create(handle, attr, func, arg) \

(\*(handle) = CreateThread(NULL, 0, func, arg, 0, NULL), (\*(handle) == NULL ? -1 : 0))

#define pthread\_join(handle, status) (WaitForSingleObject(handle, INFINITE) == WAIT\_FAILED ? -1 : 0)

#define pthread\_exit(val) ExitThread(0)

#define sleep\_ms(ms) Sleep(ms)

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

DWORD WINAPI merge\_sort\_thread(LPVOID arg);

DWORD WINAPI bitonic\_sort\_thread(LPVOID arg);

#else

#include <pthread.h>

#include <unistd.h>

#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;

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

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

#endif

int logical\_cores = 0;

void print\_array(int\* arr, int n) {

for (int i = 0; i < n; i++) printf("%d ", arr[i]);

printf("\n");

}

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);

}

int active\_threads = 0;

#ifdef \_WIN32

CRITICAL\_SECTION thread\_count\_mutex;

#else

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

#endif

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

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

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

#ifdef \_WIN32

EnterCriticalSection(&thread\_count\_mutex);

#else

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

#endif

active\_threads--;

#ifdef \_WIN32

LeaveCriticalSection(&thread\_count\_mutex);

#else

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

#endif

return NULL;

}

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

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

if (data->n <= 1) {

#ifdef \_WIN32

EnterCriticalSection(&thread\_count\_mutex);

#else

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

#endif

active\_threads--;

#ifdef \_WIN32

LeaveCriticalSection(&thread\_count\_mutex);

#else

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

#endif

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};

#ifdef \_WIN32

EnterCriticalSection(&thread\_count\_mutex);

#else

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

#endif

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

#ifdef \_WIN32

LeaveCriticalSection(&thread\_count\_mutex);

#else

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

#endif

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;

#ifdef \_WIN32

LeaveCriticalSection(&thread\_count\_mutex);

#else

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

#endif

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};

#ifdef \_WIN32

EnterCriticalSection(&thread\_count\_mutex);

#else

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

#endif

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

active\_threads++;

#ifdef \_WIN32

LeaveCriticalSection(&thread\_count\_mutex);

#else

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

#endif

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

pthread\_join(t3, NULL);

} else {

#ifdef \_WIN32

LeaveCriticalSection(&thread\_count\_mutex);

#else

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

#endif

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

}

#ifdef \_WIN32

EnterCriticalSection(&thread\_count\_mutex);

#else

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

#endif

active\_threads--;

#ifdef \_WIN32

LeaveCriticalSection(&thread\_count\_mutex);

#else

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

#endif

return NULL;

}

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

srand(time(NULL));

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

arr[i] = rand() % 10000;

}

}

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

if (argc != 3) {

fprintf(stderr, "Usage: %s --threads N\n", argv[0]);

return 1;

}

int num\_threads = atoi(argv[2]);

int n = 1000000;

int\* arr = (int\*)malloc(n \* sizeof(int));

generate\_random\_array(arr, n);

int\* seq\_arr = (int\*)malloc(n \* sizeof(int));

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

clock\_t start = clock();

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

clock\_t end = clock();

double seq\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC \* 1000; // в миллисекундах

printf("Sequential time: %.2f ms\n", seq\_time);

int\* par\_arr = (int\*)malloc(n \* sizeof(int));

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

#ifdef \_WIN32

InitializeCriticalSection(&thread\_count\_mutex);

#endif

active\_threads = 1;

pthread\_t main\_thread;

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

start = clock();

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

pthread\_join(main\_thread, NULL);

end = clock();

double par\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC \* 1000;

printf("Parallel time with %d threads: %.2f ms\n", num\_threads, par\_time);

printf("Speedup: %.2f\n", seq\_time / par\_time);

printf("Efficiency: %.2f\n", (seq\_time / par\_time) / num\_threads);

free(arr);

free(seq\_arr);

free(par\_arr);

#ifdef \_WIN32

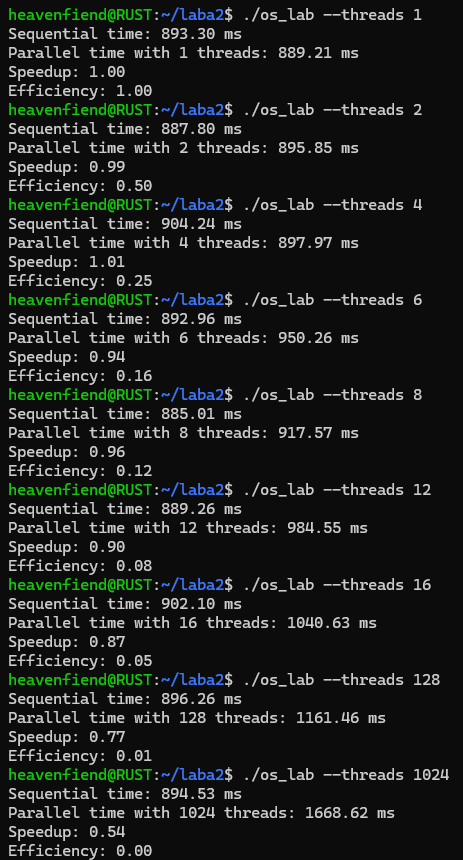
DeleteCriticalSection(&thread\_count\_mutex);

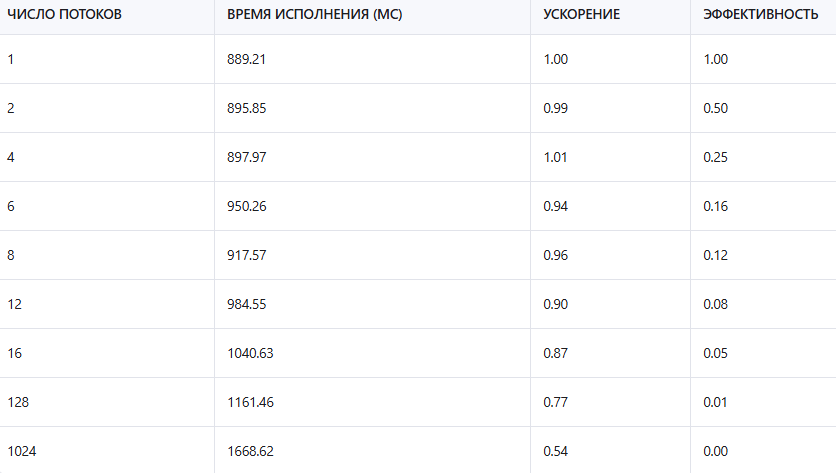
#endif

return 0;

}

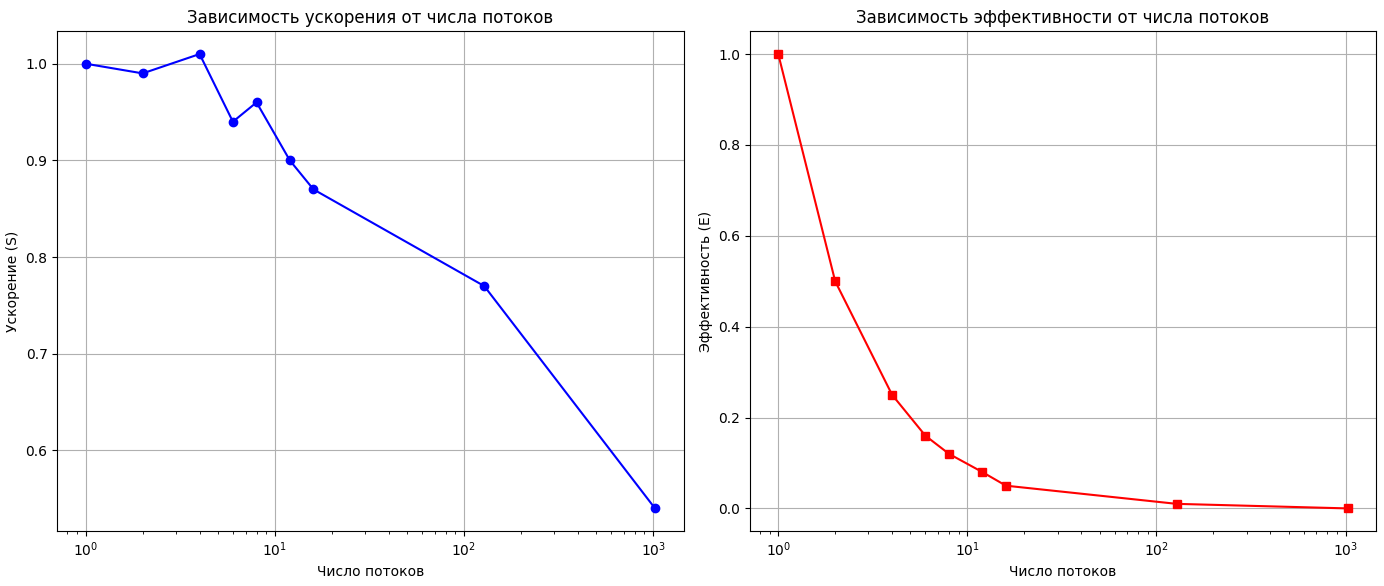
**Протокол работы программы**





**Влияние количества потоков на производительность**

* При 1 потоке:
  + Ускорение = 1.00, эффективность = 1.00.
  + Поведение аналогично последовательной версии, так как используется только один поток.
* При 2–4 потоках:
  + Ускорение близко к 1.00, эффективность падает.
  + На 4 потоках ускорение = 1.01, эффективность = 0.25.
  + Это связано с тем, что накладные расходы на создание и синхронизацию потоков начинают превышать выгоду от распараллеливания.
* При 6–8 потоках:
  + Ускорение падает до 0.94–0.96.
  + Эффективность продолжает снижаться (до 0.12).
  + Несмотря на то, что 8 потоков соответствует числу логических ядер, битоническая сортировка не распараллеливается эффективно из-за структуры алгоритма.
* При 12 и более потоках:
  + Ускорение продолжает снижаться (до 0.54 при 1024 потоках).
  + Эффективность стремится к 0.
  + При большом количестве потоков накладные расходы на создание, синхронизацию и переключение контекста становятся критичными.



**Вывод**

Параллельная версия битонической сортировки не дала ускорения из-за высоких накладных расходов на создание и синхронизацию потоков. Эффективность значительно падает с увеличением числа потоков, особенно при их избытке. Это объясняется неэффективной структурой распараллеливания и частыми блокировками. Для достижения реального ускорения рекомендуется использовать другие алгоритмы или оптимизировать текущую реализацию.