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-211БВ-24

Студент: Бабицкий И.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 12.10.25

Москва, 2025

# Постановка задачи

**Вариант 1.**

Составить программу на языке **C**, обрабатывающую данные в многопоточном режиме: отсортировать массив целых чисел при помощи битонической сортировки. При обработке 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.

Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.

В отчёте привести исследование зависимости ускорения и эффективности алгоритма от входных данных и количества потоков. Получившиеся результаты необходимо объяснить.

# Общий метод и алгоритм решения

Использованные системные вызовы:

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);` – создает поток с заданными атрибутами, который начинает выполнение функции
- `int pthread_join(pthread_t thread, void **retval);` – ожидает завершение указанного потока.
- `int pthread_mutex_lock(pthread_mutex_t *mutex);` – блокирует мьютекс, чтобы предотвратить одновременный доступ из нескольких потоков.
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);` – разблокирует мьютекс.

Я составил программу на языке **C**, сортирующую массив целых чисел в многопоточном режиме при помощи битонической сортировки. При обработке использовал стандартные средства создания поток операционной системы **Unix**. Ограничение максимального количества потоков, работающих в один момент времени, реализовано заданием ключом запуска программы.

1. Проверяем соответствие команды запуска программы формату: `./a.out -l <length of array> -d <direction of sort> -t <count of threads>`.
2. Считываем входные данные.
3. Копируем исходный массив.
4. Начинаем отсчёт времени сортировки исходного массива и сортируем его при помощи последовательной битонической сортировки.
5. Завершаем отсчёт времени сортировки исходного массива и выводим результаты.
6. Начинаем отсчёт времени сортировки копии исходного массива и сортируем её при помощи параллельной битонической сортировки: создаём родительский поток для сортировки всего массива, затем дочерние для сортировки половинок и так далее рекурсивно, увеличивая каждый раз счётчик текущего количества потоков на **2**.
7. Завершаем отсчёт времени сортировки копии исходного массива и выводим результаты.

# Код программы

**main.c**

```
#include "../include/utilities.h"
#include "../include/bitonicSort.h"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define DIRECTION_UP      1
#define DIRECTION_DOWN    0
#define MIN_ARRAY_LENGTH  1
#define MAX_ARRAY_LENGTH  100000
#define MIN_THREADS_COUNT 1
#define MAX_THREADS_COUNT 2048

double sequential(int *array1, size_t length) {
  clock_t start = clock();
  bitonicSortSequential(array1, length, DIRECTION_UP);
  clock_t end = clock();
  double time = (double)(end - start) / CLOCKS_PER_SEC;

  char message[BUFSIZ];
  snprintf(message, BUFSIZ, "  sequential:   %lf seconds;\n", time);
  printMessage(message);
  printArray("sequential", array1, length);

  return time;
}

double parallel(int *array2, size_t length, int threadsCount) {
  clock_t start = clock();
  bitonicSortParallel(array2, length, DIRECTION_UP, threadsCount);
  clock_t end = clock();
  double time = (double)(end - start) / CLOCKS_PER_SEC;

  char message[BUFSIZ];
  snprintf(message, BUFSIZ, "  parallel:     %lf seconds, %d threads;\n", time,
           threadsCount);
  printMessage(message);
  printArray("parallel", array2, length);

  return time;
}

double acceleration(double sequentialTime, double parallelTime) {
  char message[BUFSIZ];
  double value = sequentialTime / parallelTime;
```

```
  snprintf(message, BUFSIZ, "  acceleration: %lf;\n", value);
  printMessage(message);

  return value;
}

void efficiency(double acceleration, int threadsCount) {
  char message[BUFSIZ];
  snprintf(message, BUFSIZ, "  efficiency:   %lf.\n", acceleration / threadsCount);
  printMessage(message);
}

int main(int argc, char *argv[]) {
  if (validateCommand(argc, argv) == false) {
    printMessage("Invalid command. Usage: ./a.out -l <length of array>
                 -d <direction of sort> -t <count of threads>\n");
    exit(EXIT_FAILURE);
  }

  size_t length;
  sscanf(argv[2], "%zu", &length);
  if (length < MIN_ARRAY_LENGTH || length > MAX_ARRAY_LENGTH) {
    char message[BUFSIZ];
    sprintf(
      message,
      "Invalid length of array. It must be a natural number from %d to %d.\n",
      MIN_ARRAY_LENGTH, MAX_ARRAY_LENGTH
    );
    printMessage(message);
    exit(EXIT_FAILURE);
  }

  int direction;
  sscanf(argv[4], "%d", &direction);
  if (direction != DIRECTION_UP && direction != DIRECTION_DOWN) {
    printMessage("Invalid direction. It must be 0 or 1.\n");
    exit(EXIT_FAILURE);
  }

  int threadsCount;
  sscanf(argv[6], "%d", &threadsCount);
  if (threadsCount < MIN_THREADS_COUNT || threadsCount > MAX_THREADS_COUNT) {
    printMessage("Invalid count of threads. It must be natural value.\n");
    exit(EXIT_FAILURE);
  }

  int array1[MAX_ARRAY_LENGTH];
  ssize_t tmp = parseArray(array1, length);
  if (tmp != (ssize_t)length) {
```

```
    printInt((int)tmp);
    printMessage("Invalid array or file.\n");
    exit(EXIT_FAILURE);
  }

  int array2[MAX_ARRAY_LENGTH];
  copyArray(array1, array2, length);

  printMessage("Bitonic sort:\n");
  const double sequentialTime = sequential(array1, length);
  const double parallelTime = parallel(array2, length, threadsCount);
  const double accelerationValue = acceleration(sequentialTime, parallelTime);
  efficiency(accelerationValue, threadsCount);
  printMessage("Sorted arrays written to \"./test/output_sequential.txt\"
               and \"./test/output_parallel.txt\".\n");

  return 0;
}
```

**bitonicSort.h**

```
#ifndef BITONIC_SORT
#define BITONIC_SORT

#include <stddef.h>

void bitonicSortSequential(int array[], size_t length, int direction);

void bitonicSortParallel(int array[], size_t length, int direction, int _maxThreads);

#endif
```

**bitonicSort.c**

```
#include "../include/bitonicSort.h"
#include <pthread.h>
#include <stdlib.h>
#include <stdbool.h>

void compareAndSwap(int array[], size_t i, size_t j, int direction) {
  if ((array[i] > array[j]) == direction) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }
}

void bitonicMergeSequential(int array[], size_t start, size_t length, int direction) {
  if (length < 2) {
```

```
    return;
  }

  size_t middle = length / 2;
  for (size_t i = start; i != start + middle; i++) {
    compareAndSwap(array, i, i + middle, direction);
  }
  bitonicMergeSequential(array, start, middle, direction);
  bitonicMergeSequential(array, start + middle, middle, direction);
}

void bitonicRecursiveSortSequential(int array[], size_t start, size_t length,
                                    int direction) {
  if (length < 2) {
    return;
  }

  size_t middle = length / 2;
  bitonicRecursiveSortSequential(array, start, middle, direction);
  bitonicRecursiveSortSequential(array, start + middle, middle, !direction);
  bitonicMergeSequential(array, start, length, direction);
}

void bitonicSortSequential(int array[], size_t lenght, int direction) {
  if (array == NULL) {
    return;
  }
  bitonicRecursiveSortSequential(array, 0, lenght, direction);
}

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int currentThreads;
int maxThreads;

typedef struct {
  int *array;
  size_t start;
  size_t length;
  int direction;
} ThreadArgs;

void *bitonicRecursiveSortParallel(void *args);

void bitonicMergeParallel(int array[], size_t start, size_t length, int direction) {
  if (length < 2) {
    return;
  }

  size_t middle = length / 2;
```

```
  for (size_t i = start; i < start + middle; i++) {
    compareAndSwap(array, i, i + middle, direction);
  }

  pthread_mutex_lock(&mutex);
  bool doParallel = (currentThreads + 2 <= maxThreads) ? true : false;
  if (doParallel) {
    currentThreads += 2;
  }
  pthread_mutex_unlock(&mutex);
  if (!doParallel) {
    bitonicMergeSequential(array, start, middle, direction);
    bitonicMergeSequential(array, start + middle, middle, direction);
    return;
  }

  pthread_t thread1;
  ThreadArgs *args1 = (ThreadArgs *)malloc(sizeof(ThreadArgs));
  args1->array = array;
  args1->start = start;
  args1->length = middle;
  args1->direction = direction;
  pthread_create(&thread1, NULL, bitonicRecursiveSortParallel, args1);

  pthread_t thread2;
  ThreadArgs *args2 = (ThreadArgs *)malloc(sizeof(ThreadArgs));
  args2->array = array;
  args2->start = start + middle;
  args2->length = middle;
  args2->direction = !direction;
  pthread_create(&thread2, NULL, bitonicRecursiveSortParallel, args2);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  bitonicMergeParallel(array, start, length, direction);
}

void *bitonicRecursiveSortParallel(void *args) {
  ThreadArgs *_args = (ThreadArgs *)args;
  int *array = _args->array;
  size_t start = _args->start;
  size_t length = _args->length;
  int direction = _args->direction;

  if (length < 2) {
    return NULL;
  }
```

```
  size_t middle = length / 2;

  pthread_mutex_lock(&mutex);
  bool doParallel = (currentThreads + 2 <= maxThreads) ? true : false;
  if (doParallel) {
    currentThreads += 2;
  }
  pthread_mutex_unlock(&mutex);
  if (!doParallel) {
    bitonicRecursiveSortSequential(array, start, middle, direction);
    bitonicRecursiveSortSequential(array, start + middle, middle, !direction);
    bitonicMergeSequential(array, start, length, direction);
    return NULL;
  }

  pthread_t thread1;
  ThreadArgs args1 = {.array = array, .start = start, .length = middle,
                      .direction = direction};
  pthread_create(&thread1, NULL, bitonicRecursiveSortParallel, &args1);

  pthread_t thread2;
  ThreadArgs args2 = {.array = array, .start = start + middle, .length = middle,
                      .direction = !direction};
  pthread_create(&thread2, NULL, bitonicRecursiveSortParallel, &args2);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  bitonicMergeParallel(array, start, length, direction);

  pthread_mutex_lock(&mutex);
  currentThreads -= 2;
  pthread_mutex_unlock(&mutex);

  return NULL;
}

void bitonicSortParallel(int array[], size_t length, int direction, int _maxThreads) {
  currentThreads = 0;
  maxThreads = _maxThreads;

  ThreadArgs args = {.array = array, .start = 0, .length = length,
                     .direction = direction};
  pthread_t mainThread;
  pthread_create(&mainThread, NULL, bitonicRecursiveSortParallel, &args);
  pthread_join(mainThread, NULL);
}
```

**utilities.h**

```
#ifndef UTILITIES_H
#define UTILITIES_H

#include <stdbool.h>
#include <unistd.h>
#include <fcntl.h>

bool validateCommand(int argc, char *argv[]);

ssize_t parseArray(int array[], size_t length);

void copyArray(int src[], int dest[], size_t length);

void printMessage(const char *message);

void printInt(int number);

void printDouble(double number);

void printArray(const char *sortType, const int array[], size_t length);

#endif
```

**utilities.c**

```
#include "../include/utilities.h"
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define MIN_THREADS_COUNT 1
#define MAX_THREADS_COUNT 256

bool validateCommand(int argc, char *argv[]) {
  if (argc != 7) {
    return false;
  }
  if (strcmp(argv[1], "-l") != 0) {
    return false;
  }
  if (strcmp(argv[3], "-d") != 0) {
    return false;
  }
  if (strcmp(argv[5], "-t") != 0) {
    return false;
  }
  return true;
```

```
}

ssize_t parseArray(int array[], size_t length) {
  const size_t BIG_BUFSIZ = BUFSIZ * 128;
  char buffer[BIG_BUFSIZ];
  const ssize_t bytesRead = read(STDIN_FILENO, buffer, BIG_BUFSIZ - 1);
  if (bytesRead < 1) {
    return -1;
  }
  buffer[bytesRead] = '\0';

  int n;
  int scanned = 0;
  size_t i = 0;
  while (i != length && scanned < bytesRead
         && sscanf(&buffer[scanned], "%d%n", &array[i], &n) == 1) {
    scanned += n;
    ++i;
  }

  return (ssize_t)i;
}

void copyArray(int src[], int dest[], size_t length) {
  for (size_t i = 0; i != length; ++i) {
    dest[i] = src[i];
  }
}

void printMessage(const char *message) {
  write(STDERR_FILENO, message, strlen(message));
}

void printInt(int number) {
  char strNumber[BUFSIZ];
  snprintf(strNumber, BUFSIZ, "%d", number);
  write(STDOUT_FILENO, strNumber, strlen(strNumber));
}

void printDouble(double number) {
  char strNumber[BUFSIZ];
  snprintf(strNumber, BUFSIZ, "%lf", number);
  write(STDOUT_FILENO, strNumber, strlen(strNumber));
}

void printArray(const char *sortType, const int array[], size_t length) {
  char buffer[BUFSIZ];

  snprintf(buffer, BUFSIZ, "./test/output_%s.txt", sortType);
```

```
  int fd = open(buffer, O_WRONLY | O_CREAT | O_TRUNC, 0644);

  for (size_t i = 0; i != length; ++i) {
    snprintf(buffer, BUFSIZ, "%d ", array[i]);
    write(fd, buffer, strlen(buffer));
  }
}
```

# Протокол работы программы

```
$ ./a.out -l 100000 -d 0 -t 4 < ./test/input.txt
Bitonic sort:
  sequential:   0.038942 seconds;
  parallel:     0.029755 seconds, 4 threads;
  acceleration: 1.308755;
  efficiency:   0.218126.
Sorted arrays written to "./test/output_sequential.txt"
and "./test/output_parallel.txt".

$ ./a.out -l 100000 -d 0 -t 8 < ./test/input.txt
Bitonic sort:
  sequential:   0.033894 seconds;
  parallel:     0.029745 seconds, 8 threads;
  acceleration: 1.139486;
  efficiency:   0.113949.
Sorted arrays written to "./test/output_sequential.txt"
and "./test/output_parallel.txt".

$ ./a.out -l 100000 -d 0 -t 16 < ./test/input.txt
Bitonic sort:
  sequential:   0.036961 seconds;
  parallel:     0.026220 seconds, 16 threads;
  acceleration: 1.409649;
  efficiency:   0.078314.
Sorted arrays written to "./test/output_sequential.txt"
and "./test/output_parallel.txt".

$ ./a.out -l 100000 -d 0 -t 22 < ./test/input.txt
Bitonic sort:
  sequential:   0.041108 seconds;
  parallel:     0.050048 seconds, 22 threads;
  acceleration: 0.821371;
  efficiency:   0.034224.
Sorted arrays written to "./test/output_sequential.txt"
and "./test/output_parallel.txt".

$ ./a.out -l 100000 -d 0 -t 128 < ./test/input.txt
Bitonic sort:
  sequential:   0.036722 seconds;
  parallel:     0.117059 seconds, 128 threads;
```

```
  acceleration: 0.313705;
  efficiency:   0.002413.
Sorted arrays written to "./test/output_sequential.txt"
and "./test/output_parallel.txt".

$ ./a.out -l 100000 -d 0 -t 256 < ./test/input.txt
Bitonic sort:
  sequential:   0.041721 seconds;
  parallel:     0.156267 seconds, 256 threads;
  acceleration: 0.266985;
  efficiency:   0.001035.
Sorted arrays written to "./test/output_sequential.txt"
and "./test/output_parallel.txt".

$ ./a.out -l 100000 -d 0 -t 512 < ./test/input.txt
Bitonic sort:
  sequential:   0.035897 seconds;
  parallel:     0.255531 seconds, 512 threads;
  acceleration: 0.140480;
  efficiency:   0.000273.
Sorted arrays written to "./test/output_sequential.txt"
and "./test/output_parallel.txt".
```

## Вывод

В ходе выполнения лабораторной работы на многопоточность и параллельное программирование я на практике изучил работу с потоками в операционной системе **Unix** и битоническую сортировку. Результаты выполнения программы при разном количестве потоков представляю в виде таблицы:

| **Число потоков** | **Время исполнения (мс)** | **Ускорение** | **Эффективность** |
|---|---|---|---|
| 1 | 40.505 | 1 | 1 |
| 4 | 29.755 | 1.308755 | 0.218126 |
| 8 | 29.745 | 1.139486 | 0.113949 |
| 16 | 26.220 | 1.409649 | 0.078314 |
| 22* | 50.048 | 0.821371 | 0.034224 |
| 128 | 117.059 | 0.313705 | 0.002413 |
| 256 | 156.267 | 0.266985 | 0.001035 |
| 512 | 255.531 | 0.140480 | 0.000273 |

* — количество логических ядер в системе.

Расчёты:

- ускорение = `T1 / Tn`, где `T1` — время сортировки одним потоком, `Tn` — время сортировки `n` потоками;
- эффективность = ускорение / `n`.

Из данных в таблицы следует, что при количестве потоков *меньше* количества логических ядер в системе значение эффективности высокое. При количестве потоков *равное* количеству логических ядер в системе значение эффективности достигает предела. При количестве потоков *больше* количества логических ядер в системе значение эффективности низкое.