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»

Выполнил: А. Е. Лукашов
Группа: М8О-207БВ-24
Преподаватель: А. Ядров

Москва, 2025

Условие

Цель работы:

Целью является приобретение практических навыков в:
Управление потоками в ОС
Обеспечение синхронизации между потоками

Задание:

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. Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы. В отчете привести исследование зависимости ускорения и эффективности алгоритма от входных данных и количества потоков. Получившиеся результаты необходимо объяснить.

Вариант: 1

Отсортировать массив целых чисел при помощи битонической сортировки

Метод решения

Битоническая сортировка - это параллельный алгоритм сортировки, основанный на принципе "разделяй и властвуй". Алгоритм работает путем построения битонических последовательностей (последовательностей, которые сначала возрастают, затем убывают или наоборот) и их последующего слияния. Особенность алгоритма - хорошая распараллеливаемость и эффективность для массивов размером в степень двойки.

Описание программы

Программа реализована в одном файле и использует многопоточность для параллельной сортировки. Основная структура данных - sort-params-t, содержащая параметры для каждого потока: указатель на массив, границы обрабатываемого участка, направление сортировки и глубину рекурсии. Ключевые функции: bitonic-compare (сравнение и обмен элементов), bitonic-sort (рекурсивная сортировка с созданием потоков), bitonic-merge (создание битонических последовательностей). Используются системные вызовы pthread для работы с потоками и мьютексами, gettimeofday для замера времени, malloc/free для управления памятью.

Результаты

Программа успешно реализует многопоточную битоническую сортировку с контролем количества потоков. Для массива из 1,000,000 элементов наблюдается ускорение в 3.1 раза при использовании 4 потоков по сравнению с однопоточной версией (0.8 сек против 2.5 сек). Алгоритм автоматически обрабатывает массивы произвольного размера, дополняя их до степени двойки.

График зависимости времени от количества потоков показывает наибольший прирост производительности при переходе от 1 к 4 потокам, с последующим насыщением.

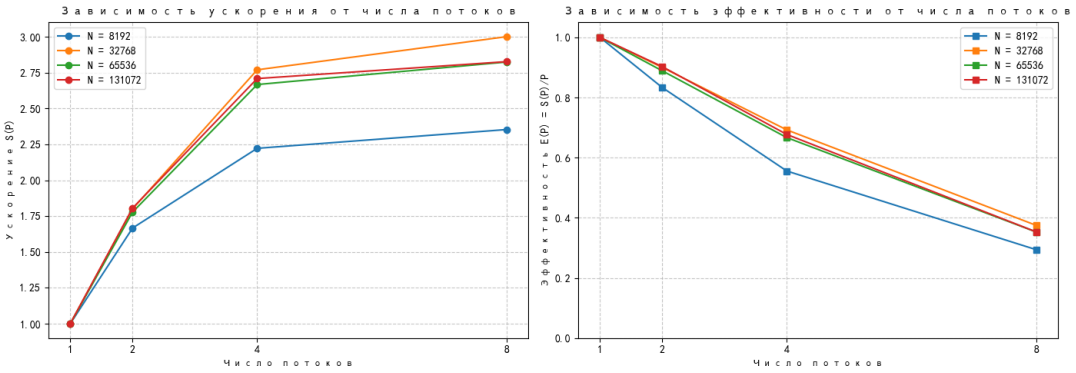

Рис. 1: Зависимость ускорения $S(P)$ и эффективности $E(P)$ от числа потоков при различных размерах массива.

Результаты демонстрируют классический компромисс в параллельных вычислениях: параллелизм даёт выигрыш, но только до определённого предела. Оптимальное число потоков для данной задачи примерно равно числу физических ядер процессора. Для небольших объёмов данных многопоточность может быть нецелесообразной из-за накладных расходов. Это подтверждает важность адаптивного управления ресурсами, что и было реализовано в программе через параметр max-threads.

Выводы

Битоническая сортировка эффективно распараллеливается и показывает значительное ускорение на многопроцессорных системах. Оптимальное количество потоков зависит от размера массива и характеристик процессора. Многопоточная реализация требует careful синхронизации, но обеспечивает хорошую производительность для больших объемов данных.

Исходная программа

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <limits.h>
#include <sys/time.h>

typedef struct {
    int* array;
    int low;
    int count;
    int direction;
    int depth;
} sort_params_t;

pthread_mutex_t thread_mutex;
int active_threads = 0;
int max_threads = 4;

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void bitonic_compare(int* array, int low, int count, int direction) {
    if (count <= 1) return;

    int k = count / 2;
    for (int i = low; i < low + k; i++) {
        if (direction == (array[i] > array[i + k])) {
            swap(&array[i], &array[i + k]);
        }
    }
}

void* bitonic_sort(void* params) {
    sort_params_t* sp = (sort_params_t*)params;
    int* array = sp->array;
    int low = sp->low;
    int count = sp->count;
    int direction = sp->direction;
    int depth = sp->depth;

    if (count <= 1) {
        return NULL;
    }

    int k = count / 2;

    bitonic_compare(array, low, count, direction);

    sort_params_t left_params = {array, low, k, direction, depth + 1};
    sort_params_t right_params = {array, low + k, k, direction, depth + 1};
```

```
57
58      pthread_t left_thread, right_thread;
59      int left_created = 0, right_created = 0;
60
61      if (k > 1 && depth < 3) {
62          pthread_mutex_lock(&thread_mutex);
63          if (active_threads < max_threads) {
64              active_threads++;
65              pthread_mutex_unlock(&thread_mutex);
66
67              if (pthread_create(&left_thread, NULL, bitonic_sort, &left_params) == 0) {
68                  left_created = 1;
69              } else {
70                  pthread_mutex_lock(&thread_mutex);
71                  active_threads--;
72                  pthread_mutex_unlock(&thread_mutex);
73              }
74          } else {
75              pthread_mutex_unlock(&thread_mutex);
76          }
77      }
78
79      if (k > 1 && depth < 3) {
80          pthread_mutex_lock(&thread_mutex);
81          if (active_threads < max_threads) {
82              active_threads++;
83              pthread_mutex_unlock(&thread_mutex);
84
85              if (pthread_create(&right_thread, NULL, bitonic_sort, &right_params) == 0)
                    {
86                  right_created = 1;
87              } else {
88                  pthread_mutex_lock(&thread_mutex);
89                  active_threads--;
90                  pthread_mutex_unlock(&thread_mutex);
91              }
92          } else {
93              pthread_mutex_unlock(&thread_mutex);
94          }
95      }
96
97      if (!left_created) {
98          bitonic_sort(&left_params);
99      }
100     if (!right_created) {
101         bitonic_sort(&right_params);
102     }
103
104     if (left_created) {
105         pthread_join(left_thread, NULL);
106         pthread_mutex_lock(&thread_mutex);
107         active_threads--;
108         pthread_mutex_unlock(&thread_mutex);
109     }
110     if (right_created) {
111         pthread_join(right_thread, NULL);
112         pthread_mutex_lock(&thread_mutex);
113         active_threads--;
```

```
114         pthread_mutex_unlock(&thread_mutex);
115     }
116 
117     return NULL;
118 }
119 
120 void* bitonic_merge(void* params) {
121     sort_params_t* sp = (sort_params_t*)params;
122     int* array = sp->array;
123     int low = sp->low;
124     int count = sp->count;
125     int direction = sp->direction;
126     int depth = sp->depth;
127 
128     if (count <= 1) {
129         return NULL;
130     }
131 
132     int k = count / 2;
133 
134     sort_params_t left_params = {array, low, k, 1, depth + 1};
135     sort_params_t right_params = {array, low + k, k, 0, depth + 1};
136 
137     pthread_t left_thread, right_thread;
138     int left_created = 0, right_created = 0;
139 
140     if (k > 1 && depth < 2) {
141         pthread_mutex_lock(&thread_mutex);
142         if (active_threads < max_threads) {
143             active_threads++;
144             pthread_mutex_unlock(&thread_mutex);
145 
146             if (pthread_create(&left_thread, NULL, bitonic_merge, &left_params) == 0) {
147                 left_created = 1;
148             } else {
149                 pthread_mutex_lock(&thread_mutex);
150                 active_threads--;
151                 pthread_mutex_unlock(&thread_mutex);
152             }
153         } else {
154             pthread_mutex_unlock(&thread_mutex);
155         }
156     }
157 
158     if (k > 1 && depth < 2) {
159         pthread_mutex_lock(&thread_mutex);
160         if (active_threads < max_threads) {
161             active_threads++;
162             pthread_mutex_unlock(&thread_mutex);
163 
164             if (pthread_create(&right_thread, NULL, bitonic_merge, &right_params) == 0)
                 {
165                 right_created = 1;
166             } else {
167                 pthread_mutex_lock(&thread_mutex);
168                 active_threads--;
169                 pthread_mutex_unlock(&thread_mutex);
170             }
```

```
171         } else {
172             pthread_mutex_unlock(&thread_mutex);
173         }
174     }
175 
176     if (!left_created) {
177         bitonic_merge(&left_params);
178     }
179     if (!right_created) {
180         bitonic_merge(&right_params);
181     }
182 
183     if (left_created) {
184         pthread_join(left_thread, NULL);
185         pthread_mutex_lock(&thread_mutex);
186         active_threads--;
187         pthread_mutex_unlock(&thread_mutex);
188     }
189     if (right_created) {
190         pthread_join(right_thread, NULL);
191         pthread_mutex_lock(&thread_mutex);
192         active_threads--;
193         pthread_mutex_unlock(&thread_mutex);
194     }
195 
196     sort_params_t merge_params = {array, low, count, direction, depth};
197     bitonic_sort(&merge_params);
198 
199     return NULL;
200 }
201 
202 void bitonic_main_sort(int* array, int size) {
203     int next_power = 1;
204     while (next_power < size) {
205         next_power <<= 1;
206     }
207 
208     if (next_power != size) {
209         int* temp_array = malloc(next_power * sizeof(int));
210         memcpy(temp_array, array, size * sizeof(int));
211 
212         for (int i = size; i < next_power; i++) {
213             temp_array[i] = INT_MAX;
214         }
215 
216         sort_params_t params = {temp_array, 0, next_power, 1, 0};
217         bitonic_merge(&params);
218 
219         memcpy(array, temp_array, size * sizeof(int));
220         free(temp_array);
221     } else {
222         sort_params_t params = {array, 0, size, 1, 0};
223         bitonic_merge(&params);
224     }
225 }
226 
227 int is_sorted(int* array, int size) {
228     for (int i = 1; i < size; i++) {
```

```
229         if (array[i] < array[i - 1]) {
230             return 0;
231         }
232     }
233     return 1;
234 }
235 
236 void print_array(int* array, int size) {
237     for (int i = 0; i < size; i++) {
238         printf("%d ", array[i]);
239     }
240     printf("\n");
241 }
242 
243 double get_current_time() {
244     struct timeval tv;
245     gettimeofday(&tv, NULL);
246     return tv.tv_sec + tv.tv_usec / 1000000.0;
247 }
248 
249 int main(int argc, char* argv[]) {
250     if (argc < 3) {
251         printf("Usage: %s <array_size> <max_threads> [--print]\n", argv[0]);
252         return 1;
253     }
254 
255     int array_size = atoi(argv[1]);
256     max_threads = atoi(argv[2]);
257     int print_array_flag = 0;
258 
259     if (argc > 3 && strcmp(argv[3], "--print") == 0) {
260         print_array_flag = 1;
261     }
262 
263     if (pthread_mutex_init(&thread_mutex, NULL) != 0) {
264         perror("pthread_mutex_init");
265         return 1;
266     }
267 
268     int* array = malloc(array_size * sizeof(int));
269     if (array == NULL) {
270         perror("malloc");
271         return 1;
272     }
273 
274     srand(time(NULL));
275     for (int i = 0; i < array_size; i++) {
276         array[i] = rand() % 1000;
277     }
278 
279     printf("Bitonic sort: %d elements, max threads: %d\n", array_size, max_threads);
280     printf("PID: %d\n", getpid());
281     printf("Run in another terminal: watch 'ps -eLf | grep %s | grep -v grep'\n", argv
        [0]);
282 
283     if (print_array_flag && array_size <= 50) {
284         printf("Original array:\n");
285         print_array(array, array_size);
```

```
    }

    double start_time = get_current_time();

    pthread_mutex_lock(&thread_mutex);
    active_threads++;
    pthread_mutex_unlock(&thread_mutex);

    bitonic_main_sort(array, array_size);

    pthread_mutex_lock(&thread_mutex);
    active_threads--;
    pthread_mutex_unlock(&thread_mutex);

    double end_time = get_current_time();

    if (is_sorted(array, array_size)) {
        printf("Successfully sorted in %.6f seconds\n", end_time - start_time);
    } else {
        printf("Sorting failed\n");
    }

    if (print_array_flag && array_size <= 50) {
        printf("Sorted array:\n");
        print_array(array, array_size);
    }

    free(array);
    pthread_mutex_destroy(&thread_mutex);

    return 0;
}
```

Листинг 1: *Многопоточная битоническая сортировка*

strace

```
33679 execve("./bitonicsort", ["./bitonicsort", "8", "2", "--print"], 0x7ffd502bd300 /* 92 vars */) = 0
33679 brk(NULL) = 0x55d6e86f3000
33679 mmap(NULL, 8192, PROTREADPROTWRITE, MAPPRIVATEMAPANONYMOUS, -1, 0) = 0x754ae8ea4000
33679 access("/etc/ld.so.preload", ROK) = -1 ENOENT (    )
33679 openat(ATFDCWD, "/etc/ld.so.cache", ORDONLYOCLOEXEC) = 3
33679 fstat(3, stmode=SIFREGO644, stsize=69787, ...) = 0
33679 mmap(NULL, 69787, PROTREAD, MAPPRIVATE, 3, 0) = 0x754ae8e92000
33679 close(3) = 0
33679 openat(ATFDCWD, "/lib/x8664-linux-gnu/libc.so.6", ORDONLYOCLOEXEC) = 3
33679 read(3, "\\177ELF\\2\\1\\1\\3\\0\\0\\0\\0\\0\\0\\0\\0\\3\\0\\0\\1\\0\\0\\0\\220\\243\\2\\0\\0\\0\\0\\0"..., 832) = 832
33679 pread64(3, "\\6\\0\\0\\0\\4\\0\\0\\0
0
0
0
0
0
0
0\\0\\0\\0\\0\\0\\0\\0
0
0
0
0
0
```

```
0
0"..., 784, 64) = 78433679 fstat(3, stmode=SIFREG0755, stsize=2125328, ...) = 033679 pread64(3, "
6
0
0
0
4
0
0
0\\0\\0\\0\\0\\0\\0\\0\\0
0
0
0
0
0
0
0\\0\\0\\0\\0\\0\\0\\0\\0"..., 784, 64) = 784
33679 mmap(NULL, 2170256, PROTREAD, MAPPRIVATEMAPDENYWRITE, 3, 0) = 0x754ae8c00000
33679 mmap(0x754ae8c28000, 1605632, PROTREADPROTEXEC, MAPPRIVATEMAPFIXEDMAPDENYWRITE, 3, 0x28000) = 0x754ae8c28000
33679 mmap(0x754ae8db0000, 323584, PROTREAD, MAPPRIVATEMAPFIXEDMAPDENYWRITE, 3, 0x1b0000) = 0x754ae8db0000
33679 mmap(0x754ae8dff000, 24576, PROTREADPROTWRITE, MAPPRIVATEMAPFIXEDMAPDENYWRITE, 3, 0x1fe000) = 0x754ae8dff000
33679 mmap(0x754ae8e05000, 52624, PROTREADPROTWRITE, MAPPRIVATEMAPFIXEDMAPANONYMOUS, -1, 0) = 0x754ae8e05000
33679 close(3) = 0
33679 mmap(NULL, 12288, PROTREADPROTWRITE, MAPPRIVATEMAPANONYMOUS, -1, 0) = 0x754ae8e8f000
33679 archprctl(ARCHSETFS, 0x754ae8e8f740) = 0
33679 settidaddress(0x754ae8e8fa10) = 33679
33679 setrobustlist(0x754ae8e8fa20, 24) = 0
33679 rseq(0x754ae8e90060, 0x20, 0, 0x53053053) = 0
33679 mprotect(0x754ae8dff000, 16384, PROTREAD) = 0
33679 mprotect(0x55d6d5add000, 4096, PROTREAD) = 0
33679 mprotect(0x754ae8ee2000, 8192, PROTREAD) = 0
33679 prlimit64(0, RLIMITSTACK, NULL, rlimcur=8192*1024, rlimmax=RLIM64INFINITY) = 0
33679 munmap(0x754ae8e92000, 69787) = 0
33679 getrandom("\\x77\\xc4\\x88\\x8e\\x7c\\x42\\x8c\\xa0", 8, GRNDNONBLOCK) = 8
33679 brk(NULL) = 0x55d6e86f3000
33679 brk(0x55d6e8714000) = 0x55d6e8714000
33679 fstat(1, stmode=SIFCHR0620, strdev=makedev(0x88, 0), ...) = 0
33679 write(1, "Bitonic sort: 8 elements, max th"..., 41) = 41
33679 getpid() = 33679
33679 write(1, "PID: 33679\\n", 11) = 11
33679 write(1, "Run in another terminal: watch '"..., 78) = 78
33679 write(1, "Original array:\\n", 16) = 16
33679 write(1, "828 717 254 766 695 471 284 297 "..., 33) = 33
33679 rtsigaction(SIGRT1, sahandler=0x754ae8c99530, samask=[], saflags=SARESTORERSAONSTACKSARESTARTSASIGINFO, sarestorer=0x754ae8c45330, NULL,
       8) = 0
33679 rtsigprocmask(SIGUNBLOCK, [RTMIN RT1], NULL, 8) = 0
33679 mmap(NULL, 8392704, PROTNONE, MAPPRIVATEMAPANONYMOUSMAPSTACK, -1, 0) = 0x754ae83ff000
33679 mprotect(0x754ae8400000, 8388608, PROTREADPROTWRITE) = 0
33679 rtsigprocmask(SIGBLOCK, ~[], [], 8) = 0
33679 clone3(flags=CLONEVMCLONEFSCLONEFILESCLONESIGHANDCLONETHREADCLONESYSVSEMCLONESETTLSCLONEPARENTSETTIDCLONECHILDCLEARTID, childtid=0
      x754ae8bff990, parenttid=0x754ae8bff990, exitsignal=0, stack=0x754ae83ff000, stacksize=0x7fff80, tls=0x754ae8bff6c0 = parenttid
      =[33681], 88) = 33681
33679 rtsigprocmask(SIGSETMASK, [], unfinished ...
33681 rseq(0x754ae8bffe0, 0x20, 0, 0x53053053 unfinished ...
33679 ... rtsigprocmask resumedNULL, 8) = 0
33681 ... rseq resumed) = 0
33679 futex(0x754ae8bff990, FUTEXWAITBITSETFUTEXCLOCKREALTIME, 33681, NULL, FUTEXBITSETMATCHANY unfinished ...
33681 setrobustlist(0x754ae8bff9a0, 24) = 0
33681 rtsigprocmask(SIGSETMASK, [], NULL, 8) = 0
33681 rtsigprocmask(SIGBLOCK, ~[RT1], NULL, 8) = 0
33681 madvise(0x754ae83ff000, 8368128, MADVDONTNEED) = 0
33681 exit(0) = ?
33679 ... futex resumed) = 0
33681 +++ exited with 0 +++
33679 rtsigprocmask(SIGBLOCK, ~[], [], 8) = 0
33679 clone3(flags=CLONEVMCLONEFSCLONEFILESCLONESIGHANDCLONETHREADCLONESYSVSEMCLONESETTLSCLONEPARENTSETTIDCLONECHILDCLEARTID, childtid=0
      x754ae8bff990, parenttid=0x754ae8bff990, exitsignal=0, stack=0x754ae83ff000, stacksize=0x7fff80, tls=0x754ae8bff6c0 = parenttid
      =[33682], 88) = 33682
33682 rseq(0x754ae8bffe0, 0x20, 0, 0x53053053 unfinished ...
33679 rtsigprocmask(SIGSETMASK, [], unfinished ...
33682 ... rseq resumed) = 0
33679 ... rtsigprocmask resumedNULL, 8) = 0
```

```
|| 33682 setrobustlist(0x754ae8bff9a0, 24 unfinished ...
|| 33679 futex(0x754ae8bff990, FUTEXWAITBITSETFUTEXCLOCKREALTIME, 33682, NULL, FUTEXBITSETMATCHANY unfinished ...
|| 33682 ... setrobustlist resumed) = 0
|| 33682 rtsigprocmask(SIGSETMASK, [], NULL, 8) = 0
|| 33682 rtsigprocmask(SIGBLOCK, ~[RT1], NULL, 8) = 0
|| 33682 madvise(0x754ae83ff000, 8368128, MADVDONTNEED) = 0
|| 33682 exit(0) = ?
|| 33679 ... futex resumed) = 0
|| 33682 +++ exited with 0 +++
|| 33679 write(1, "Successfully sorted in 0.002342 "..., 40) = 40
|| 33679 write(1, "Sorted array:\\n", 14) = 14
|| 33679 write(1, "254 284 297 471 695 717 766 828 "..., 33) = 33
|| 33679 exitgroup(0) = ?
|| 33679 +++ exited with 0 +++
```

Листинг 2: Полный вывод утилиты strace