Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №8 по курсу «Объектно-ориентированное программирование»

Студент: М. В. Спиридонов
Преподаватель:
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2017

Лабораторная работа №8

Задача:
Целью лабораторной работы является:

1. Знакомство с параллельным программированием в C++.

Используя структуры данных, разработанные для лабораторной работы No6 (контейнер первого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класса-контейнера. Необходимо разработать два вида алгоритма:

- Обычный, без параллельных вызовов.
- С использованием параллельных вызовов. В этом случае, каждый рекурсивный вызов сортировки должен создаваться в отдельном потоке.

Для создания потоков использовать механизмы:

1. future
2. packaged_task/async

Для обеспечения потоко-безопасности структур данных использовать:

1. mutex
2. lock_guard

Фигуры. Квадрат, треугольник, прямоугольник.
Контейнер первого уровня. Массив.
Контейнер первого уровня. Массив.

1 Теория

Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцесорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading). Это способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно).

Термин охватывает совокупность вопросов параллелизма в программировании, а также создание эффективно действующих аппаратных реализаций. Теория параллельных вычислений составляет раздел прикладной теории алгоритмов. Существуют различные способы реализации параллельных вычислений. Например, каждый вычислительный процесс может быть реализован в виде процесса операционной системы, либо же вычислительные процессы могут представлять собой набор потоков выполнения внутри одного процесса ОС. Параллельные программы могут физически исполняться либо последовательно на единственном процессоре — перемежая по очереди шаги выполнения каждого вычислительного процесса, либо параллельно — выделяя каждому вычислительному процессу один или несколько процессоров (находящихся рядом или распределённых в компьютерную сеть).

2 Листинг

```
#ifndef TVECTOR_H
#define TVECTOR_H

#include "iostream"
#include "TAllocationBlock.h"
#include "TVectorItem.h"
//#include <memory>

template <class T>
class Iter {
public:
    Iter (int n, T *arr) {
        _n = n;
        _arr = arr;
    }

    T &operator * () {
        return _arr[_n];
    }

    T &operator -> () {
        return _arr[_n];
    }

    void operator ++ () {
        _n++;
    }

    Iter operator ++ (int) {
        ++(*this);
        return &this;
    }

    bool operator == (Iter const &i) {
        return _n == i._n &&
               _arr == i._arr;
    }

    bool operator != (Iter const &i) {
        return !(*this == i);
    }

private:
    int _n;
    T *_arr;
};

```

```
48 template <class T>
49 class TVector {
50 public:
51
52     void FastPushBack (T &data);
53
54     void Clear ();
55
56     T &operator [] (int index);
57
58     int Size () const;
59
60     int Capacity () const;
61
62     Iter <T> begin ();
63
64     Iter <T> end ();
65
66     TVector &operator = (const TVector <T> &inp);
67
68     TVector ();
69
70     void Sort(int low, int hight);
71
72     TVector (const TVector <T> &inp);
73
74     friend std::ostream &operator << (std::ostream &os, const TVector <T> &tVector);
75
76     explicit TVector (int n);
77
78     ~TVector ();
79
80 private:
81     void swap(T* a, T* b);
82     int partition (int low, int high);
83     //static TAllocationBlock vector_allocator;
84     T *privateArray;
85     int privateArraySize;
86     int privateArrayOccupiedSize;
87     static int const SIZE_DIFFERENCE = 1;
88     static int const DEFAULT_INT_VALUE = 0;
89 };
90
91 #include "TVector.hpp"
92
93 #endif //TVECTOR_H
94
95 //TVector.hpp
96 #ifndef TVECTOR_HPP
```

```
97  #define TVECTOR_HPP
98
99  #include "TVector.h"
100
101 template <class T>
102 std::ostream &operator << (std::ostream &os, const TVector <T> &tVector) {
103     for (int i = 0; i < tVector.Capacity(); ++i) {
104         os << tVector[i]->Print() << std::endl;
105     }
106     return os;
107 }
108 template <class T>
109 void TVector<T>::swap(T* a, T* b)
110 {
111     T t = *a;
112     *a = *b;
113     *b = t;
114 }
115
116 template <class T>
117 int TVector<T>::partition (int low, int high)
118 {
119     double pivot = privateArray[high]->Square(); // pivot
120     int i = (low - 1); // Index of smaller element
121
122     for (int j = low; j <= high- 1; j++)
123     {
124         // If current element is smaller than or
125         // equal to pivot
126         if (privateArray[j]->Square() <= pivot)
127         {
128             i++; // increment index of smaller element
129             swap(&privateArray[i], &privateArray[j]);
130         }
131     }
132     swap(&privateArray[i + 1], &privateArray[high]);
133     return (i + 1);
134 }
135
136 template <class T>
137 void TVector<T>::Sort (int low, int high) {
138
139     //clock_t start = clock();
140     if (low <= high)
141     {
142         /* pi is partitioning index, arr[p] is now
143            at right place */
144         int pi = partition(low, high);
145
```

```
146         // Separately sort elements before
147         // partition and after partition
148         Sort(low, pi - 1);
149         Sort(pi + 1, high);
150     }
151     //clock_t end = clock();
152     //printf("\nCounter SortTime = %lf\n", (double)(end - start) / CLOCKS_PER_SEC);
153 }
154 
155 template <class T>
156 void TVector <T>::FastPushBack (T &data) {
157     if (privateArraySize == privateArrayOccupiedSize) {
158         privateArraySize *= 2;
159         T *result = new T[privateArraySize];
160 
161         for (int index = DEFAULT_INT_VALUE; index <= privateArrayOccupiedSize; index++)
                 {
162             if (index != privateArrayOccupiedSize) {
163                 result[index] = privateArray[index];
164             } else {
165                 result[index] = data;
166                 break;
167             }
168         }
169         delete[] privateArray;
170         privateArray = result;
171         privateArrayOccupiedSize++;
172     } else {
173         privateArray[privateArrayOccupiedSize] = data;
174         privateArrayOccupiedSize++;
175     }
176 }
177 
178 template <class T>
179 void TVector <T>::Clear () {
180     delete[] privateArray;
181     privateArraySize = DEFAULT_INT_VALUE;
182     privateArrayOccupiedSize = DEFAULT_INT_VALUE;
183     privateArray = new T[privateArraySize];
184 }
185 
186 template <class T>
187 T &TVector <T>::operator [] (int index) {
188     return this->privateArray[index];
189 }
190 
191 template <class T>
192 int TVector <T>::Size () const {
193     return privateArraySize;
```

```
194 }
195
196 template <class T>
197 int TVector <T>::Capacity () const {
198     return this->privateArrayOccupiedSize;
199 }
200
201 template <class T>
202 TVector <T> &TVector <T>::operator = (const TVector <T> &inp) {
203     return *this;
204 }
205
206 template <class T>
207 TVector <T>::TVector () {
208     privateArraySize = DEFAULT_INT_VALUE;
209     privateArrayOccupiedSize = DEFAULT_INT_VALUE;
210     privateArray = new T[privateArraySize];
211 }
212
213 template <class T>
214 TVector <T>::TVector (const int n) {
215     privateArraySize = n;
216     privateArrayOccupiedSize = DEFAULT_INT_VALUE;
217     privateArray = new T[privateArraySize];
218 }
219
220 template <class T>
221 TVector <T>::TVector (const TVector <T> &inp) {
222     privateArraySize = inp.privateArraySize;
223     privateArrayOccupiedSize = inp.privateArrayOccupiedSize;
224     privateArray = inp.privateArray;
225 }
226
227 template <class T>
228 Iter<T> TVector<T>::begin () {
229     return Iter<T>(0,privateArray);
230 }
231
232 template <class T>
233 Iter<T> TVector<T>::end () {
234     return Iter<T>(privateArrayOccupiedSize,privateArray);
235 }
236
237 template <class T>
238 TVector <T>::~TVector () {
239     delete[] privateArray;
240     privateArraySize = DEFAULT_INT_VALUE;
241     privateArrayOccupiedSize = DEFAULT_INT_VALUE;
242 }
```

```
243
244 void merge(TYPE *data, int left, int right, int tid) {
245     if (DEBUG) {
246         printf("[%d] Merging %d to %d\n", tid, left, right);
247     }
248     int ctr = 0;
249     int i = left;
250     int mid = left + ((right - left) / 2);
251     int j = mid + 1;
252     int *c = (int *) malloc((right - left + 1) * sizeof(int));
253     while (i <= mid && j <= right) {
254         if (data[i] <= data[j]) {
255             c[ctr++] = data[i++];
256         } else {
257             c[ctr++] = data[j++];
258         }
259     }
260     // Either i = mid + 1 OR j = right + 1
261     if (i == mid + 1) {
262         while (j <= right) {
263             c[ctr++] = data[j++];
264         }
265     } else {
266         while (i <= mid) {
267             c[ctr++] = data[i++];
268         }
269     }
270     // Copy the data back !
271     i = left;
272     ctr = 0;
273     while (i <= right) {
274         data[i++] = c[ctr++];
275     }
276     free(c);
277     return;
278 }
279
280 void *merge_sort_threaded(void *arg) {
281     thread_data_t *data = (thread_data_t *) arg;
282     int l = data->left;
283     int r = data->right;
284     int t = data->tid;
285     if (r - l + 1 <= minLenght) {
286         // Length is too short, let us do a |qsort|.
287         if (DEBUG) {
288             printf("[%d] Calling qsort(%d, %d).\n", t, l, r);
289         }
290         qsort(data->array + l, r - l + 1, sizeof(TYPE), my_comp);
291     } else {
```

```
292         // Try to create two threads and assign them work.
293         int m = l + ((r - l) / 2);
294         // Data for thread 1
295         thread_data_t data_0;
296         data_0.left = l;
297         data_0.right = m;
298         data_0.array = data->array;
299         pthread_mutex_lock(&lock_number_of_threads);
300             data_0.tid = number_of_threads++;
301         pthread_mutex_unlock(&lock_number_of_threads);
302         // Create thread 1
303         pthread_t thread0;
304         int rc = pthread_create(&thread0,
305                                 NULL,
306                                 merge_sort_threaded,
307                                 &data_0);
308         int created_thread_0 = 1;
309         if (rc) {
310             // Failed to create thread, call |qsort|.
311             if (DEBUG) {
312                 printf("[%d] Failed to create thread, calling qsort.", data_0.tid);
313             }
314             created_thread_0 = 0;
315             qsort(data->array + l, m - l + 1, sizeof(TYPE), my_comp);
316         }
317         // Data for thread 2
318         thread_data_t data_1;
319         data_1.left = m + 1;
320         data_1.right = r;
321         data_1.array = data->array;
322         pthread_mutex_lock(&lock_number_of_threads);
323             data_1.tid = number_of_threads++;
324         pthread_mutex_unlock(&lock_number_of_threads);
325         // Create thread 2
326         pthread_t thread1;
327         rc = pthread_create(&thread1,
328                             NULL,
329                             merge_sort_threaded,
330                             &data_1);
331         int created_thread_1 = 1;
332         if (rc) {
333             // Failed to create thread, call |qsort|.
334             if (DEBUG) {
335                 printf("[%d] Failed to create thread, calling qsort.", data_1.tid);
336             }
337             created_thread_1 = 0;
338             qsort(data->array + m + 1, r - m, sizeof(TYPE), my_comp);
339         }
340         // Wait for the created threads.
```

```
341         if (created_thread_0) {
342             pthread_join(thread0, NULL);
343         }
344         if (created_thread_1) {
345             pthread_join(thread1, NULL);
346         }
347         // Ok, both done, now merge.
348         // left - l, right - r
349         merge(data->array, l, r, t);
350     }
351     pthread_exit(NULL);
352     return NULL;
353 }
354 
355 void merge_sort(TYPE *array, int start, int finish) {
356     thread_data_t data;
357     data.array = array;
358     data.left = start;
359     data.right = finish;
360     // Initialize the shared data.
361     number_of_threads = 0;
362     pthread_mutex_init(&lock_number_of_threads, NULL);
363     data.tid = 0;
364     // Create and initialize the thread
365     pthread_t thread;
366     int rc = pthread_create(&thread,
367                             NULL,
368                             merge_sort_threaded,
369                             &data);
370     if (rc) {
371         if (DEBUG) {
372             printf("[%d] Failed to create thread, calling qsort.",
373                    data.tid);
374         }
375         qsort(array + start,
376               finish - start + 1,
377               sizeof(TYPE),
378               my_comp);
379     }
380     // Wait for thread, i.e. the full merge sort algo.
381     pthread_join(thread, NULL);
382     return;
383 }
384 #endif // TVECTOR_HPP
```

3 Выводы

В данной лабораторной работе я получил алгоритмы работы с параллельным программированием в C++. Реализовал алгоритм быстрой сортировки и распараллелил его.