

**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: Севастьянов В.С.
Преподаватель: Поповкин А. В.
Группа: 08-207
Вариант: 19
Дата:
Оценка:
Подпись:

Москва, 2017

Лабораторная работа №8

1 Цель работы

- Знакомство с параллельным программированием в C++.

2 Задача

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер 1-ого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класс-контейнера.

Необходимо разработать два вида алгоритма:

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

Для создания потоков использовать механизмы:

- future
- packaged task/async

Для обеспечения потокобезопасности структур использовать механизмы:

- mutex
- lock quard

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера.
- Удалять фигуры из контейнера.
- Проводить сортировку контейнера.

Фигуры: Прямоугольник, ромб, трапеция

Контейнер 1: N-дерево **Контейнер 2:** Массив

3 Описание

Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).

Параллельное программирование может быть сложным, но его легче понять, если считать его не “трудным”, а просто “немного иным”. Оно включает в себя все черты более традиционного, последовательного программирования, но в параллельном программировании имеются три дополнительных, четко определенных этапа:

- Определение параллелизма: анализ задачи с целью выделить подзадачи, которые могут выполняться одновременно.
- Выявление параллелизма: изменение структуры задачи таким образом, чтобы можно было эффективно выполнять подзадачи. Для этого часто требуется найти зависимости между подзадачами и организовать исходный код так, чтобы ими можно было эффективно управлять.
- Выражение параллелизма: реализация параллельного алгоритма в исходном коде с помощью системы обозначений параллельного программирования.

4 Исходный код

```
1 | template<class T>
2 | size_t Tree<T>::partition(size_t arr[], size_t low, size_t high)
3 | {
4 |     int pivot = arr[high]; // pivot
5 |     int i = (low - 1); // Index of smaller element
6 |
7 |     for (int j = low; high != 0 && j <= high - 1; j++)
8 |     {
9 |         // If current element is smaller than or
10 |         // equal to pivot
11 |         if (arr[j] <= pivot)
12 |         {
13 |             i++; // increment index of smaller element
14 |             //swap(&arr[i], &arr[j]);
15 |             size_t tmp = arr[i];
16 |             arr[i] = arr[j];
17 |             arr[j] = tmp;
18 |         }
19 |     }
20 |     //swap(&arr[i + 1], &arr[high]);
21 |     size_t tmp = arr[i + 1];
```

```

22     arr[i + 1] = arr[high];
23     arr[high] = tmp;
24     return (i + 1);
25 }
26
27 template<class T>
28 void Tree<T>::quickSort(size_t arr[], size_t low, size_t high)
29 {
30     if (low < high)
31     {
32
33         size_t pi = partition(arr, low, high);
34         if (pi > 0)
35             quickSort(arr, low, pi - 1);
36         quickSort(arr, pi + 1, high);
37     }
38 }
39
40 size_t FutureQuickSort(size_t *arr, size_t left, size_t right)
41 {
42     size_t i = left, j = right;
43
44     size_t tmp;
45
46     size_t pivot = arr[(left + right) / 2];
47
48     while (i <= j) {
49         while (arr[i] < pivot)
50             i++;
51         while (arr[j] > pivot)
52             j--;
53         if (i <= j) {
54             tmp = arr[i];
55             arr[i] = arr[j];
56             arr[j] = tmp;
57             i++;
58             j--;
59         }
60     }
61
62     if (i < right) {
63         std::packaged_task<size_t(size_t *, size_t, size_t)> task(FutureQuickSort);
64         auto result = task.get_future();
65
66         std::thread task_td(std::move(task), std::ref(arr), i, right);
67         task_td.join();
68         result.get();
69     }
70     if (left < j) {

```

```
71 |     std::packaged_task<size_t(size_t *, size_t, size_t)> task(FutureQuickSort);
72 |     auto result = task.get_future();
73 |
74 |     std::thread task_td(std::move(task), std::ref(arr), left, j);
75 |     task_td.join();
76 |     result.get();
77 | }
78 | return 0;
79 | }
```

5 Выводы

Благодаря этой лабораторной работе я получил опыт применения потоков в C++, что, несомненно, пригодится в дальнейшем. Распараллеливание вычислений - незаменимая вещь для уменьшения времени работы над каким-то алгоритмом, например сортировки работают в разы быстрее, чем в однопоточной реализации.