Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»
Разработка многопоточной сортировки слиянием (Multi-threaded Merge Sort)

Студент: Л. Я. Вельтман
Преподаватель: А. А. Журавлёв
Группа: М8О-207Б

Дата:
Оценка:
Подпись:

Москва, 2019

Курсовой проект

Задача: Необходимо реализовать многопоточную сортировку слиянием. Формат запуска должен поддерживать считывание из файла и из стандартного ввода в терминале.

1 Описание

Реализация курсового проекта сводится к следующим задачам:

1. Изучить теоретическую часть сортировки слиянием.
2. Реализовать алгоритм многопоточной сортировки.
3. Учесть нюансы и обработать исключения.

2 Теоретическая часть

Сортировка слиянием (англ. Merge sort) — алгоритм сортировки, использующий O(n) дополнительной памяти и работающий за O(nlog(n)) времени.
Алгоритм использует принцип «разделяй и властвуй»: задача разбивается на подзадачи меньшего размера, которые решаются по отдельности, после чего их решения комбинируются для получения решения исходной задачи. Конкретно процедуру сортировки слиянием можно описать следующим образом:

1. Если в рассматриваемом массиве один элемент, то он уже отсортирован — алгоритм завершает работу.
2. Иначе массив разбивается на две части, которые сортируются рекурсивно.
3. После сортировки двух частей массива к ним применяется процедура слияния, которая по двум отсортированным частям получает исходный отсортированный массив.

В моем задании ясно говорится, что мне требуется именно многопоточная сортировка слиянием. Благодаря тому, что сортировка слиянием построена на принципе "Разделяй и властвуй выполнение данного алгоритма можно весьма эффективно распараллелить. При оценке асимптотики допускается, что возможен запуск неограниченного количества независимых процессов, что на практике не достижимо. Более того, при реализации имеет смысл ограничить количество параллельных потоков, что я и сделала. Для ограничения использовала функцию, которая возвращает число одновременно выполняемых потоков, то есть количество ядер процессора, и брала от этого числа логарифм степени 2. На моей машине ограничение потоков равно $log_2 4 = 2$.
Был применен алгоритм рекурсивного слияния массивов T[left1...right1] и T[left2...right2]в массив A[left3...right3].

1. Проверяем, что размер первого массива больше размера второго. Если это не так, то меняем местами левую позицию первого массива с левой позицией второго, аналогично для правых позиций.
2. Вычисляем середину первого массива mid1.
3. С помощью бинарного поиска находим середину второго массива mid2, причем T[mid2] $<$ T[mid1].
4. Вычисляем середину mid3 результирующего массива A с помощью формулы mid3 = left3 + (mid1 - left1) + (mid2 - left2)
5. В результирующем массиве позиции mid3 присваиваем значение T[mid1].

6. Сливаем T[left1...mid1 - 1] и T[left2...mid2 - 1] в A[left3...mid3 - 1]

7. Сливаем T[mid1 + 1...right1] и T[mid2...right2] в A[mid3 + 1...right3].

Оказалось, что алгоритм с применением бинарного поиска работает не так быстро, как бы хотелось, поэтому было решено модифицировать программу добавлением деления массива на 2 части по медиане. Это происходит на этапе слияния, когда на вход подается два массива с уже вычисленными левыми и правыми границами. Ищем максимум max среди двух элементов [r1, r2], так мы определим элемент, который должен стоять в конце результирующего массива, затем ищем минимум min из двух элементов [l1, l2] - это будет первым элементов в результирующем массиве. Вычисляем среднее значение min и max. Это число и будет медианой, по которой будет сливаться массив, все числа меньшие медианы будут стоять слева от нее, а большие - справа.
Так как существует ограничение на потоки, то при достижении этого ограничения, наш массив может быть не до конца отсортирован, поэтому нужно прибегнуть к вызову обычной последовательной сортировки слиянием.

3 Реализация

```cpp
#include <iostream>
#include <fstream>
#include <thread>
#include <vector>
#include <cmath>
#include <exception>
#include <stdexcept>
#include <algorithm>
#include <utility>
#include <chrono>


bool ReadFile(std::ifstream& file, std::vector<int64_t>& vec1) {
    if (file.is_open() && file.peek() != EOF) {
        int64_t number;
        int64_t size = 1;
        bool flag = true;
        while (size) {
            file >> number;
            if (file.eof()) { break; }
            if (flag) {
                size = number;
                flag = false;
            } else {
                vec1.emplace_back(number);
                --size;
            }
        }
    } else {
        return false;
    }

    file.close();
    return true;
}


void Reader(std::vector<int64_t>& vec1) {
    int64_t number, size;
    std::cin >> size;
    while (size) {
        std::cin >> number;
        vec1.emplace_back(number);
        --size;
    }
}

```

```
48
49 void OrdinaryMerge(std::vector<int64_t>& tmp, int64_t l1, int64_t r1, int64_t l2,
       int64_t r2, std::vector<int64_t>& res, int64_t lres) {
50     int64_t rres = lres + (r1 - l1) + (r2 - l2) + 1;
51
52     while (lres <= rres) {
53         if (l1 > r1) {
54             res[lres] = tmp[l2++];
55         } else if (l2 > r2) {
56             res[lres] = tmp[l1++];
57         } else {
58             res[lres] = (tmp[l1] < tmp[l2]) ? tmp[l1++] : tmp[l2++];
59         }
60         ++lres;
61     }
62 }
63
64
65 void OrdinaryMergeSort(std::vector<int64_t>& tmp, int64_t l1, int64_t r1, std::vector<
       int64_t>& res) {
66       if (l1 >= r1) { return; }
67
68       int64_t mid = (l1 + r1) / 2;
69
70       OrdinaryMergeSort(res, l1, mid, tmp);
71       OrdinaryMergeSort(res, mid + 1, r1, tmp);
72
73       OrdinaryMerge(tmp, l1, mid, mid + 1, r1, res, l1);
74
75    }
76
77
78 int64_t BinarySearch(int64_t key, std::vector<int64_t>& vec, int64_t l, int64_t r) {
79     r = std::max(l, r + 1);
80     while (l < r ) {
81         int64_t mid = (l + r) / 2;
82         if (vec[mid] < key) {
83             l = mid + 1;
84         } else {
85             r = mid;
86         }
87     }
88     return l;
89 }
90
91
92 void MedianaSearch(int64_t key, std::vector<int64_t>& vec, int64_t l1, int64_t r1,
       int64_t l2, int64_t r2, std::vector<int64_t>& length) {
93
```

```
94      uint64_t l = std::min(vec[l1], vec[l2]);
95      uint64_t r = std::max(vec[r1], vec[r2]);
96      int64_t infirst, insecond;
97      int64_t mediana = (l + r) / 2;
98      infirst = std::upper_bound(vec.begin() + l1, vec.begin() + r1, mediana) - vec.begin
            ();
99      insecond = std::upper_bound(vec.begin() + l2, vec.begin() + r2, mediana) - vec.
            begin();
100      if (vec[infirst] > mediana && infirst != l1) {
101          --infirst;
102      } else if (vec[infirst] == mediana) {
103          while (vec[infirst] == mediana && infirst < r1 ) {
104              ++infirst;
105          }
106          if (vec[infirst] > mediana) {
107              --infirst;
108          }
109      }
110      if (vec[insecond] > mediana && insecond != l2) {
111          --insecond;
112      } else if (vec[insecond] == mediana) {
113          while (vec[insecond] == mediana && insecond < r2) {
114              ++insecond;
115          }
116          if (vec[insecond] > mediana) {
117              --insecond;
118          }
119      }
120      length.emplace_back(infirst - l1 + 1);
121      length.emplace_back(infirst);
122      length.emplace_back(insecond - l2 + 1);
123      length.emplace_back(insecond);
124
125      return ;
126  }
127
128
129  void Merge(std::vector<int64_t>& tmp, int64_t l1, int64_t r1, int64_t l2, int64_t r2,
        std::vector<int64_t>& res, int64_t lres, int64_t depth, bool limiter) {
130      int64_t n1 = r1 - l1 + 1;
131      int64_t n2 = r2 - l2 + 1;
132
133      if (depth <= 0) {
134          OrdinaryMerge(tmp, l1, r1, l2, r2, res, lres);
135          return;
136      }
137
138      if (n1 < n2) {
139          Merge(tmp, l2, r2, l1, r1, res, lres, depth, limiter);
```

```
140     }
141     if (n1 == 0) { return; }
142     int64_t infirst, inffirst, insecond, total;
143     if (limiter) {
144         int64_t mid1 = (l1 + r1) / 2;
145         int64_t mid2 = BinarySearch(tmp[mid1], tmp, l2, r2);
146         total = lres + (mid1 - l1) + (mid2 - l2) ;
147 
148         res[total] = tmp[mid1];
149         infirst = mid1 - 1 - l1;
150         inffirst = mid1 - l1;
151         insecond = mid2 - 1;
152     } else {
153         int64_t min, max;
154         int64_t min_pos, max_pos;
155 
156         if (tmp[l1] > tmp[l2]) {
157                min = tmp[l2];
158                min_pos = l2;
159         } else {
160                min = tmp[l1];
161                min_pos = l1;
162         }
163         if (tmp[r1] > tmp[r2]) {
164                max = tmp[r1];
165                max_pos = r1;
166         } else {
167                max = tmp[r2];
168                max_pos = r2;
169         }
170 
171         int64_t mediana = (min + max) / 2;
172         int64_t answer;
173         std::vector<int64_t> length;
174 
175         MedianaSearch(mediana, tmp, l1, r1, l2, r2, length);
176         infirst = length[1];
177         inffirst = length[1];
178         insecond = length[3];
179 
180         total = length[0] + length[2] - 1;
181         if (min_pos == l2) {
182             answer = length[1] ;
183         } else if (min_pos == l1) {
184             answer = length[3];
185         }
186         res[total] = tmp[answer];
187     }
188     --depth;
```

```
189
190
191     std::thread tt1(Merge, std::ref(tmp), l1, l1 + infirst, l2, insecond , std::ref(res
            ), lres, depth, limiter);
192     std::thread tt2(Merge, std::ref(tmp), l1 + inffirst + 1, r1, insecond + 1, r2, std
            ::ref(res), total + 1 , depth, limiter);
193     tt1.join();
194     tt2.join();
195 }
196
197
198 void MergeSortForOne(std::vector<int64_t>& tmp, int64_t l1, int64_t r1, std::vector<
        int64_t>& res, int64_t depth, bool limiter) {
199
200     if (depth <= 0) {
201         OrdinaryMergeSort(tmp, l1, r1, res);
202         return;
203     }
204     if (l1 >= r1) {
205         return;
206     } else {
207         int64_t mid = (l1 + r1) / 2;
208
209         --depth;
210         std::thread t1(MergeSortForOne, std::ref(res), l1, mid, std::ref(tmp), depth,
                limiter);
211         std::thread t2(MergeSortForOne, std::ref(res), mid + 1, r1, std::ref(tmp),
                depth, limiter);
212         t1.join();
213         t2.join();
214         Merge(tmp, l1, mid, mid + 1, r1, res, l1, depth, limiter);
215     }
216 }
217
218
219 void CountingSort(std::vector<int64_t>& array, int64_t n) {
220     std::vector<int64_t>::iterator iter = std::max_element(array.begin(), array.end());
221     int64_t index = std::distance(array.begin(), iter);
222     int64_t k = array[index];
223     std::vector<int64_t> second(n);
224     std::vector<int64_t> c(k + 1, 0);
225
226     for (int64_t i = 0; i < n; ++i) {
227         ++c[array[i]];
228     }
229
230     for (int64_t i = 1; i < k + 1; ++i) {
231        c[i] += c[i - 1];
232     }
```

```
233
234     for (int64_t i = n; i > 0; --i) {
235         second[c[array[i - 1]] - 1] = array[i - 1];
236         --c[array[i - 1]];
237     }
238     for(size_t i = 0; i < n; ++i) {
239         array[i] = second[i];
240     }
241 }
242
243
244 int main (int argc, char const *argv[]) {
245     std::vector<int64_t> vec1;
246     try {
247         if (argc == 2) {
248             std::string filename = argv[1];
249             std::ifstream file(filename);
250
251             if (!ReadFile(file, vec1)) {
252                 Reader(vec1);
253             }
254         } else {
255             Reader(vec1);
256         }
257     }
258     catch (const std::exception& e) {
259         std::cout << e.what() << std::endl;
260     }
261
262     int64_t l1 = 0, r1 = vec1.size() - 1;
263
264     std::vector<int64_t> tmp1_1(vec1.size());
265     tmp1_1.assign(vec1.begin(), vec1.end());
266
267     int64_t depth = log2(std::thread::hardware_concurrency());
268
269     bool limiter = (depth == 2) ? false : true;
270
271     try {
272         std::thread t1(MergeSortForOne, std::ref(tmp1_1), l1, r1, std::ref(vec1), depth
                , limiter);
273         t1.join();
274     } catch (std::runtime_error &ex) {
275         std::cerr << ex.what();
276     }
277
278
279     if (argc == 2) {
280         std::string filename = "output";
```

```
281         std::ofstream file(filename);
282         if (file.is_open()) {
283             for (int i = 0; i < vec1.size(); ++i) {
284                 file << vec1[i] << std::endl;
285           }
286           file << '\n';
287         }
288         file.close();
289     } else {
290         for (int i = 0; i < vec1.size(); ++i) {
291             std::cout << vec1[i] << ' ';
292         }
293         std::cout << "\n";
294     }
295     return 0;
296 }
```

4 Тест производительности

Тест производительности представляет собой сравнение работы многопоточной сортировки слиянием в 1, 2, 3, 4 потока с обычной последовательной сортировкой слиянием, std::sort, sort (bash), линейной сортировкой подсчетом.

Я написала линейную сортировку подсчетом. Так как она не поддерживает отрицательные числа (моя реализация), то тесты были заменены. Отрицательных чисел нет, но размерность тестов сохранилась.

```
1  void CountingSort(std::vector<int64_t>& array, int64_t n) {
2      std::vector<int64_t>::iterator iter = std::max_element(array.begin(), array.end());
3      int64_t index = std::distance(array.begin(), iter);
4      int64_t k = array[index];
5      std::vector<int64_t> second(n);
6      std::vector<int64_t> c(k + 1, 0);
7
8      for (int64_t i = 0; i < n; ++i) {
9          ++c[array[i]];
10     }
11
12     for (int64_t i = 1; i < k + 1; ++i) {
13        c[i] += c[i - 1];
14     }
15
16     for (int64_t i = n; i > 0; --i) {
17         second[c[array[i - 1]] - 1] = array[i - 1];
18         --c[array[i - 1]];
19     }
20     for(size_t i = 0; i < n; ++i) {
21         array[i] = second[i];
22     }
23 }
```

Тесты:
randtest1kk - 1 миллион строк с рандомными числами;
uptest1kk - 1 миллион строк чисел, отсортированных по возрастанию;
downtest1kk - 1 миллион строк чисел, отсортированных по убыванию;

randtest10kk - 10 миллионов строк с рандомными числами;
uptest10kk - 10 миллионов строк чисел, отсортированных по возрастанию;
downtest10kk - 10 миллионов строк чисел, отсортированных по убыванию;

randtest100kk - 100 миллионов строк с рандомными числами;
uptest100kk - 100 миллионов строк чисел, отсортированных по возрастанию;
downtest100kk - 100 миллионов строк чисел, отсортированных по убыванию;

Тесты	randtest1kk	uptest1kk	downtest1kk
Parallel Merge Sort 1 thread user	0.169 seconds 0m1.790s	0.086 seconds 0m1.551s	0.09 seconds 0m1.563s
Parallel Merge Sort 2 thread user	0.132 seconds 0m1.877s	0.076 seconds 0m1.623s	0.077 seconds 0m1.733s
Parallel Merge Sort 3 thread user	0.125 seconds 0m1.867s	0.077 seconds 0m1.668s	0.08 seconds 0m1.658s
Parallel Merge Sort 4 thread user	0.132 seconds 0m1.905s	0.084 seconds 0m1.652s	0.085 seconds 0m1.673s
Sequal Merge Sort user	0.311 seconds 0m1.520s	0.16 seconds 0m1.305s	0.166 seconds 0m1.335s
std::sort user	0.064 seconds 0m1.476s	0.001 seconds 0m1.370s	0.002 seconds 0m1.410s
Linear Counting Sort user	100.549 seconds 1m20.188s	103.292 seconds 1m20.185s	111.943 seconds 1m23.227s
sort from bash	user 0m1.088s	user 0m1.139s	user 0m2.395s

Тесты	randtest10kk	uptest10kk	downtest10kk
Parallel Merge Sort 1 thread	1.933 seconds	0.996 seconds	1.041 seconds
user	0m17.965s	0m15.565s	0m16.097s
Parallel Merge Sort 2 thread	1.712 seconds	0.873 seconds	0.891 seconds
user	0m19.650s	0m17.251s	0m17.403s
Parallel Merge Sort 3 thread	1.421 seconds	0.947 seconds	0.948 seconds
user	0m19.757s	0m16.966s	0m17.042s
Parallel Merge Sort 4 thread	1.761 seconds	0.956 seconds	1 seconds
user	0m20.170s	0m17.103s	0m17.409s
Sequal Merge Sort	3.64 seconds	1.893 seconds	1.914 seconds
user	0m16.060s	0m14.040s	0m14.080s
std::sort	0.753 seconds	0.013 seconds	0.027 seconds
user	0m15.628s	0m13.713s	0m13.692s
Linear Counting Sort	189.074 seconds	146.789 seconds	132.752 seconds
user	1m55.846s	1m47.014s	1m43.623s
sort from bash	0m39.020s	0m11.415s	0m12.357s

Тесты	randtest100kk	uptest100kk	downtest100kk
Parallel Merge Sort 1 thread 2m3.552s	27.144 seconds 1m39.435s	15.489 seconds	15.404 seconds user 1m46.718
Parallel Merge Sort 2 thread 3m51.390s	28.346 seconds 3m12.896s	17.309 seconds 3m1.880s	20.238 seconds user
Parallel Merge Sort 3 thread 2m6.646s	15.525 seconds 1m47.141s	15.189 seconds 1m49.376s	11.025 seconds user
Parallel Merge Sort 4 thread user	15.33 seconds 2m12.633s	12.502 seconds 1m57.242s	14.46 seconds 1m58.133s
Sequal Merge Sort user	45.513 seconds 2m59.318s	34.497 seconds 2m42.997s	24.704 seconds 2m29.869s
std::sort user	24.391 seconds 3m9.602s	0.234 seconds 2m34.925s	1.674 seconds 2m29.551s
Linear Counting Sort user	1647.64 seconds 7m54.768s	2294.97 seconds 3m30.745s	150.576 seconds 3m38.582s
sort from bash	12m21.939s	3m24.166s	3m43.246s

Самой быстрой сортировкой оказалась std::sort из библиотеки STL. Стандартом не оговаривается, какой именно алгоритм должен быть реализован у std::sort, но сложность обязательно должна быть O(nlogn). Поэтому обычная быстрая сортировка не может быть использована, а Introsort или интроспективная сортировка — использует быструю сортировку и переключается на пирамидальную сортировку, когда глубина рекурсии превысит некоторый заранее установленный уровень (например, логарифм от числа сортируемых элементов). Этот подход сочетает в себе достоинства обоих методов с худшим случаем O(nlogn) и быстродействием, сравнимым с быстрой сортировкой. Могу предположить, что в сортировке предусмотрены критические случаи, для которых придумали нужные и важные оптимизации, что и позволяет этому алгоритму обгонять мою реализацию многопоточного алгоритма слияния. По моему мнению, второе место занимает двухпоточная сортировка слиянием, она показывает довольно стабильные результаты на любых данных, только алгоритм слияния в три потока показывает лучшее время на рандомных тестах, но на отсортированных данных двухпоточная сортировка выигрывает на некоторое количество секунд.

Для сравнения производительности и времени работы моей многопоточной сортировки слиянием я использовала команду sort в bash. Ключ -n нужен для сортировки

строк по числовому значению, ключ -o - выводить результат в файл. Она дала средние показатели на тестах в 1 миллион и 10 миллионов данных. Но ужасно себя повела при 100 миллионах рандомных данных, она работала дольше всех, время работы команды sort превышает работу других сортировок почти в 4 раза. К сожалению, я не смогла найти никакой информации об оценке работы алгоритма(ов), который(ые) реализован(ы) внутри.

Хуже всех работает сортировка подсчетом, ее сложность O(k + n), где n - количество элементов, k - наибольший элемент последовательности, который, по сути, отвечает за размерность алфавита. k мог принять наибольшее число в тестах, например, 9 223 372 036 854 775 807 (верхняя граница типа int64_t), сложив это число с n = 10 000 000, то мы получим число операций явно большее, чем O(nlogn) - сложность при данном n = 10 000 000, за которую работает сортировка слиянием.

Сразу заметна разница между последовательным и параллельным алгоритмом. Многопоточный вариант сортировки слиянием обогнал последовательный метод. Но потраченное именно процессорное время от начала запуска команды time до конца работы у параллельного варианта больше. Дополнительное время могло потратиться из-за издержек на создание и/или порождение новых потоков. Даже, если параллельная программа написана правильно, издержки, создаваемые конкурентными конструкторами, могут перегрузить среду выполнения и производительность программы уменьшится. Поэтому оптимальным решением будет применять параллелизм только в том случае, если размер массива больше порогового (глубина рекурсии), который обычно соответствует количеству ядер, после чего программа возвращается к последовательному алгоритму, что я и применила в решении своей задачи.
Чтобы оценить производительность, количество затраченной памяти и возможные утечки я использовала утилиту Valgrind.

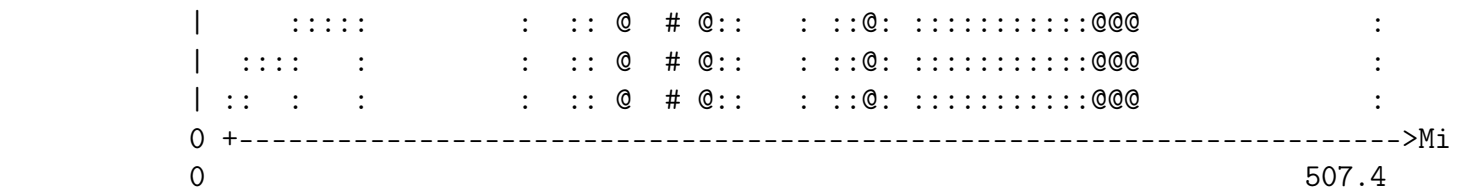
```
linuxxxoid@linuxxxoid-MS-7721:~/MAI/DA/CW$ g++ cwda.cpp -std=c++11 -lpthread
-o da
linuxxxoid@linuxxxoid-MS-7721:~/MAI/DA/CW$ valgrind ./da test150k
==3271== Memcheck,a memory error detector
==3271== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==3271== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3271== Command: ./da test150k
==3271==
==3271==
==3271== HEAP SUMMARY:
==3271==     in use at exit: 0 bytes in 0 blocks
==3271==   total heap usage: 143 allocs,143 frees,5,500,880 bytes allocated
==3271==
==3271== All heap blocks were freed --no leaks are possible
==3271==
```

```
==3271== For counts of detected and suppressed errors,rerun with: -v
==3271== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Так как программа отлажена и работает корректно, то ошибок и утечек памяти не наблюдается.
Для измерения потребления памяти я использовала heap profiler - Massif. Он измеряет сколько памяти в куче задействовала программа.

```
linuxxxoid@linuxxxoid-MS-7721:~/MAI/DA/CW$ valgrind --tool=massif ./da test150k
==3979== Massif,a heap profiler
==3979== Copyright (C) 2003-2017,and GNU GPL'd,by Nicholas Nethercote
==3979== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3979== Command: ./da test150k
==3979==
==3979==
linuxxxoid@linuxxxoid-MS-7721:~/MAI/DA/CW$ ms_print massif.out.3979 | head
--lines=100
--------------------------------------------------------------------------------
Command:            ./da test150k
Massif arguments:   (none)
ms_print arguments: massif.out.3979
--------------------------------------------------------------------------------


MB
3.223^                                    ::::
|                    :   ::@@@#:@::   ::::@:::::::::::::@@@@@@@@@@@@@@@@@:
|                    :   : @  # @::   : ::@: ::::::::::::@@@               :
|                    :   : @  # @::   : ::@: ::::::::::::@@@               :
|                    :   : @  # @::   : ::@: ::::::::::::@@@               :
|                    :   : @  # @::   : ::@: ::::::::::::@@@               :
|                    :   : @  # @::   : ::@: ::::::::::::@@@               :
|                    :   : @  # @::   : ::@: ::::::::::::@@@               :
|                    ::::: @  # @::   : ::@: ::::::::::::@@@               :
|                    :  :: @  # @::   : ::@: ::::::::::::@@@               :
|                    :  :: @  # @::   : ::@: ::::::::::::@@@               :
|         :          :  :: @  # @::   : ::@: ::::::::::::@@@               :
|         :          :  :: @  # @::   : ::@: ::::::::::::@@@               :
|         :          :  :: @  # @::   : ::@: ::::::::::::@@@               :
|         ::::::::::::  :: @  # @::   : ::@: ::::::::::::@@@               :
|     :   :          :  :: @  # @::   : ::@: ::::::::::::@@@               :
|     :   :          :  :: @  # @::   : ::@: ::::::::::::@@@               :
```




```
Number of snapshots: 78
Detailed snapshots: [15,16 (peak),19,29,53,63,73]
```

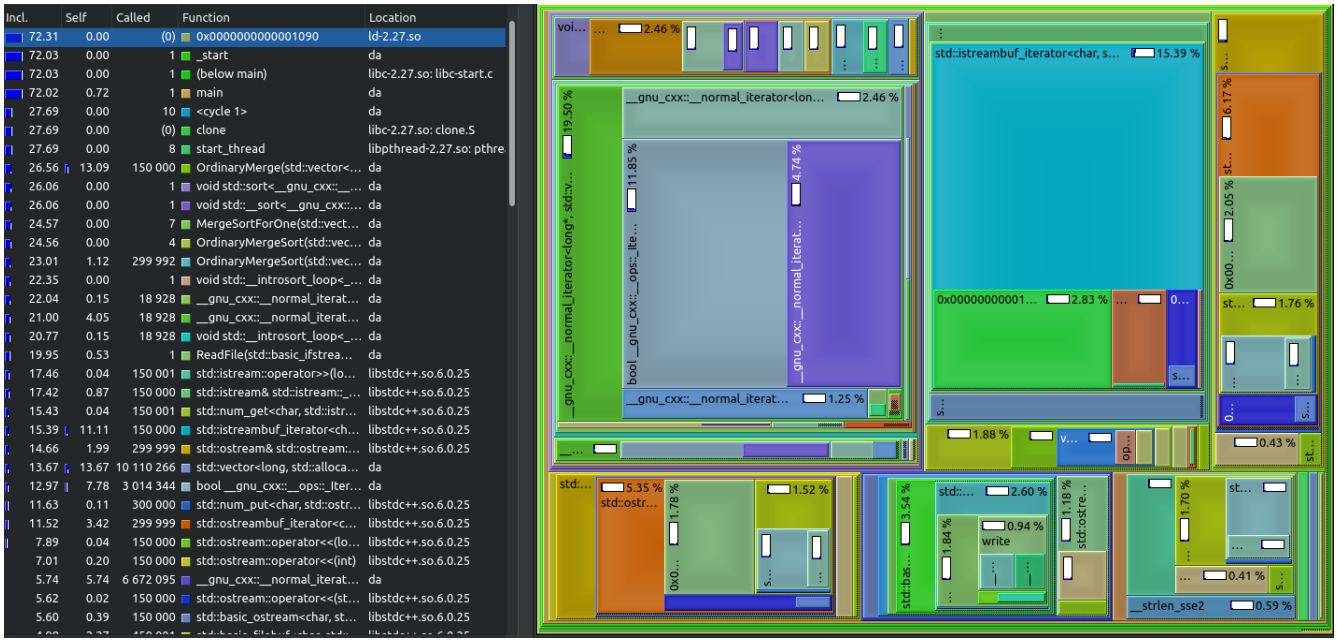
n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	2,295,368	72,712	72,704	8	0
2	6,888,963	147,024	146,984	40	0
3	11,323,552	212,560	212,520	40	0
4	20,163,283	474,712	474,664	48	0
5	20,190,769	343,632	343,592	40	0
6	37,870,161	867,928	867,880	48	0
7	37,924,271	605,776	605,736	40	0
8	73,268,702	1,654,360	1,654,312	48	0
9	73,376,060	1,130,064	1,130,024	40	0
10	144,105,363	3,227,224	3,227,176	48	0
11	144,319,217	2,178,640	2,178,600	40	0
12	164,747,084	2,170,440	2,170,408	32	0
13	171,399,743	3,376,136	3,375,704	432	0
14	185,081,897	3,376,208	3,375,760	448	0
15	188,527,245	3,378,344	3,377,768	576	0

(а) график, показывающий потребление памяти во время выполнения программы, и (б) подробную информацию об ответственных местах распределения в различных точках программы, включая точку пикового выделения памяти. Существует не более одного пикового снимка. Пиковый снимок является подробным снимком и записывает точку, где потребление памяти было наибольшим. Снимок пика представлен на графике полосой, состоящей из символов «#».
Callgrind - это инструмент профилирования, который записывает историю вызовов между функциями в ходе выполнения программы в виде графа вызовов.
callgrind_annotate - эта команда считывает данные профиля и печатает отсортированные списки функций, опционально с примечанием источника. Для графической визуализации данных использовала KCachegrind.

```
linuxxxoid@linuxxxoid-MS-7721:~/MAI/DA/CW$ valgrind -tool=callgrind ./da test150k
```

```
==4294== Callgrind,a call-graph generating cache profiler
==4294== Copyright (C) 2002-2017,and GNU GPL'd,by Josef Weidendorfer et al.
==4294== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4294== Command: ./da test150k
==4294==
==4294== For interactive control,run 'callgrind_control -h'.
==4294==
==4294== Events : Ir
==4294== Collected : 532099512
==4294==
==4294== I refs: 532,099,512
linuxxxoid@linuxxxoid-MS-7721:~/MAI/DA/CW$ callgrind_annotate callgrind.out.4294-----
Profile data file 'callgrind.out.4294'(creator: callgrind-3.13.0)
--------------------------------------------------------------------------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 -100119476
Trigger: Program termination
Profiled target: ./da test150k (PID 4294,part 1)
Events recorded: Ir
Events shown: Ir
Event sort order: Ir
Thresholds: 99
Include dirs:
User annotated:
Auto-annotation: off

--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
532,099,512 PROGRAM TOTALS
linuxxxoid@linuxxxoid-MS-7721:~/MAI/DA/CW$ kcachegrind ./da test150k
```



5 Выводы

Выполнив курсовой проект по курсу «Дискретный анализ», я приобрела практические и теоретические навыки благодаря полученным в течении курса знаниям. Задачи с применением многопоточности всегда интересные. Они решают две главные задачи:

1. Одновременно выполняют несколько действий.
2. Ускоряют вычисления.

Действительно, многопоточность решает важные задачи, ее использование ускоряет работу программ. Но при неправильном использовании она создает проблем. Существует две проблемы:

1. Взаимная блокировка (deadlock). Несколько потоков находятся в ожидании ресурсов, занятых друг другом, и ни один из них не может продолжать выполнение.
2. Состояние гонки (race condition). Работа программы зависит от того, в каком порядке выполняются части кода.

Потому мало просто придумать параллельный алгоритм, нужно учесть нюансы реальной работы на машинах и предусмотреть возможные проблемы и реализовать их решения.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))