Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Отчет о лабораторной работе

Курс: Параллельные вычисления

Тема: Создание многопоточных программ на языке C++ с использованием Pthreads и OpenMP

Выполнил студент группы 13541/3 _____ Д.В. Круминьш
(подпись)

Преподаватель _____ И.В. Стручков
(подпись)

Санкт-Петербург
2018 г.

Содержание

1 Постановка задачи

1.1 Индивидуальное задание

Вариант 6, OpenMP.
Вершины дерева размечены числовыми значениями. Для каждой вершины рассчитать сумму чисел всех вершин, для которых данная вершина является корнем.

1.2 Программа работы

1. Для алгоритма из полученного задания написать последовательную программу на языке C или C++, реализующую этот алгоритм.
2. Для созданной последовательной программы необходимо написать 3-5 тестов, которые покрывают основные варианты функционирования программы. Для создания тестов можно воспользоваться механизмом Unit-тестов среды NetBeans, или описать входные тестовые данные в файлах. При использовании NetBeans необходимо в свойствах проекта установить ключ компилятора -pthread.
3. Проанализировать полученный алгоритм, выделить части, которые могут быть распараллелены, разработать структуру параллельной программы. Определить количество используемых потоков, а также правила и используемые объекты синхронизации.
4. Согласовать разработанную структуру и детали реализации параллельной программы с преподавателем.
5. Написать код параллельной программы и проверить ее корректность на созданном ранее наборе тестов. При необходимости найти и исправить ошибки.
6. Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты.
7. Сделать общие выводы по результатам проделанной работы:
    - Различия между способами проектирования последовательной и параллельной реализаций алгоритма.
    - Возможные способы выделения параллельно выполняющихся частей, Возможные правила синхронизации потоков
    - Сравнение времени выполнения последовательной и параллельной программ.

- Принципиальные ограничения повышения эффективности параллельной реализации по сравнению с последовательной.

2 Сведения о системе

Работа производилась на реальной системе, со следующими характеристиками:

Элемент	Значение
Имя ОС	Майкрософт Windows 10 Pro (Registered Trademark)
Версия	10.0.16299 Сборка 16299
Установленная оперативная память (RAM)	16,00 ГБ
Процессор	Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz, 2496 МГц, ядер: 4, логических процессоров: 4

Таблица 1: Сведения о системе

3 Структура проекта

Структура проекта выглядит следующим образом:

```
project
├── source code
│   ├── Main.cpp
│   ├── Tree.cpp
│   └── TreeUtils.cpp
└── header files
    ├── Tree.h
    └── TreeUtils.h
```

Рис. 1: Структура проекта

Точка входа, расположена в файле **Main.cpp**, в котором вызываются необходимые функции, реализованные в **TreeUtils.cpp**.

3.1 Структура бинарного дерева

Элемент дерева имеет двух потомков, своё значение и сумму значений всех его потомков.

```
7  /* Структура бинарного дерева*/
8  struct tnode
9  {
10     unsigned long long value = 0;    // числовое значение
11     unsigned long long sum = 0; // сумма значений дочерних узлов
12     struct tnode *left = NULL;  // левый потомок
13     struct tnode *right =NULL;  // правый потомок
14 };
15
16 /* Добавить узел */
17 tnode* addNode(unsigned long long v, tnode *tree);
```

Листинг 1: Отрывок Tree.h

Значение и сумма потомков хранятся в переменной типа **unsigned long long**, что позволяет работать с числами до 18 446 744 073 709 551 615.

3.2 Вспомогательные функции

Вспомогательные функции реализованы в файле **treeUtils.cpp**, особого внимания стоит уделить генерации случайного числа(функция llrand).

```
17 unsigned long long llrand() {
18     unsigned long long r = 0;
19
20     for (int i = 0; i < 5; ++i) {
21         r = (r << 15) | rand();
22     }
23
24     return r & MAX_VALUE;
25 }
```

Листинг 2: Отрывок TreeUtils.cpp

Запись long long занимает 64 бита, а стандартный оператор rand() позволяет генерировать значение лишь до 32767 (15 бит). Для покрытия всех битовых значений, используется побитовый сдвиг(15 раз за итерацию).

По завершению цикла, биты полученного значения обрезаются, в соответствии с заданным максимальным значением(MAX_VALUE(4 294 967 295)).

Данный метод используется при генерации случаного дерева. Для дерева также реализованы функции экспорта(exportTreeToFile) и импорта(importTreeFromFile) в файл, для более правдивого сравнения между реализациями.

Полный код приведен в листинге 9.

4 Алгоритм решения

4.1 Последовательная реализция

Алгоритм заключается в рекурсивном вызове функции getSumOfAllChilds для подсчета суммы значений всех потомков.

```
118 unsigned long long getSumOfAllChilds(tnode* tree) {
119     if (tree != NULL){
120         unsigned long long leftSum = 0;
121         unsigned long long rightSum = 0;
122 
123         if (tree->left != NULL) {
124             tree->left->sum = getSumOfAllChilds(tree->left);
125             leftSum = tree->left->sum + tree->left->value;
126         }
127 
128         if (tree->right != NULL)
129         {
130             tree->right->sum = getSumOfAllChilds(tree->right);
131             rightSum = tree->right->sum + tree->right->value;
132         }
133 
134         return leftSum + rightSum;
135     }
136     return 0;
137 }
```

Листинг 3: Отрывок TreeUtils.cpp

4.2 Параллельный алгоритм с использованием Pthreads

В отличии от последовательной реализации, в данном случае, накладывается ограничение, на количество возможных потоков.

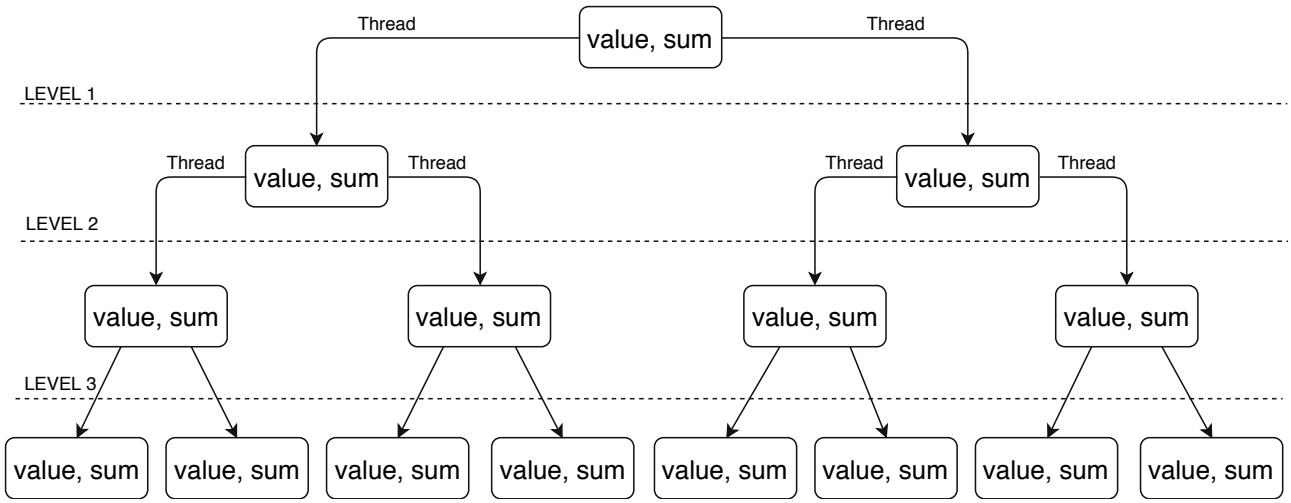

Рис. 2: Уровни вложенности параллельной программы

Каждый вызов подсчета суммы потомоков выполняется в отдельном потоке, это происходит до дех пор, пока имеются свободные ядра процессора, после их исчерпания, подсчет выполняется последовательно.

В моем случае, на машине имеется 4 ядра, по завершению 2 уровня вложенности, будет выполняться 4 потока, и далее, для каждого из них, подсчет будет производиться последовательно.

Для синхронизации, выполнение каждого параллельного потока приостанавливается до тех пор, пока все порожденные потоки не закончат вычисления.

Исходный код приведен в листинге 9.

4.3 Параллельный алгоритм с использованием OpenMP

Алгоритм для OpenMP аналогичен Pthreads. Благодаря использованию деректив (некоторому подобию аннотаций из Java), удалось сократить количество строк реализации.

В частности были использованы следующие дерективы:

- #pragma omp parallel num_threads(2)
- #pragma omp sections
- #pragma omp section

Код каждой дерективы section выполняется одним потоком.

```
160 unsigned long long getSumOfAllChilds_OpenMP(tnode* tree) {
161     if (tree != NULL) {
```

```
162         unsigned long long leftSum = 0;
163     unsigned long long rightSum = 0;
164 
165     if (omp_get_active_level() >= omp_get_max_active_levels())
166             return getSumOfAllChilds(tree);
167 
168         #pragma omp parallel num_threads(2)
169         {
170             #pragma omp sections
171             {
172                 #pragma omp section
173                 {
174                     // сумма потомков для левого поддерева
175                     if (tree->left != NULL){
176                         tree->left->sum = getSumOfAllChilds_OpenMP(tree->left);
177                         leftSum = tree->left->sum + tree->left->value;
178                     }
179         }
180 
181         #pragma omp section
182         {
183                     // сумма потомков для правого поддерева
184                     if (tree->right != NULL){
185                         tree->right->sum = getSumOfAllChilds_OpenMP(tree->right
    ↪ );
186                         rightSum = tree->right->sum + tree->right->value;
187                     }
188         }
189             }
190     }
191         return leftSum + rightSum;
192     }
193     return 0;
194 }
```

Листинг 4: Отрывок TreeUtils.cpp

5 Тестирование

5.1 Эксперименты

5.1.1 Эксперимент 1

Количество потоков: 4 (равно числу логических процессоров)
Количество узлов: от 100 до ~10 000 000

Число узлов	Последовательный	OpenMP	Pthreads
100	0.000002	0.000816	0.000918
1000	0.000019	0.000545	0.000736
10000	0.000368	0.000654	0.000852
99932	0.001575	0.001193	0.001290
993761	0.015306	0.010708	0.010508
9486172	0.143309	0.079550	0.083088

Таблица 2: Зависимость от количества узлов

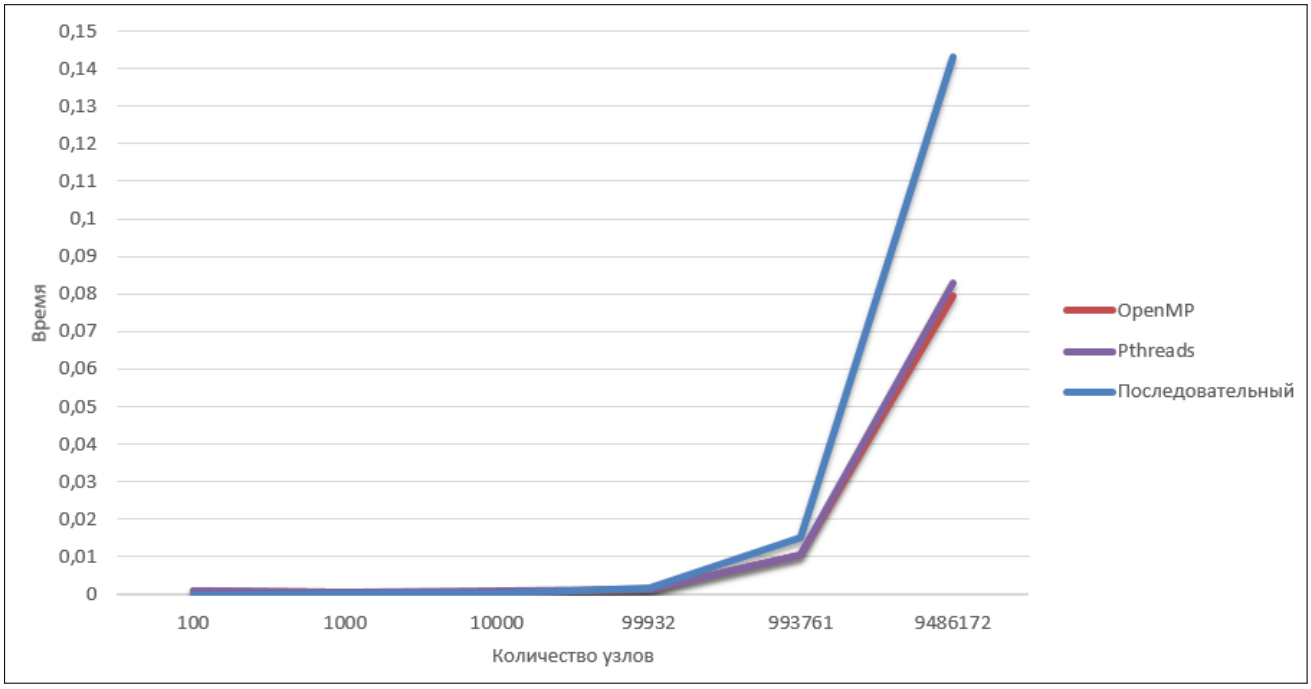

Рис. 3: Зависимость времени от количества узлов

Из эксперимента видно, что:

- до 100 000 элементов, лидировало последовательное решение, после чего, уступило параллельным решениям.
- OpenMP и Pthreads в целом показывали похожие результаты.

Для более точных результатов, необходимо провести большее числов экспериментов.

5.1.2 Эксперимент 2

Количество потоков: от 4 до 100
Количество узлов: ∼10 000 000

Число потоков	Последовательный	OpenMP	Pthreads
1	0.143858	0.135808	0.139525
2	-	0.101601	0.099593
4	-	0.058160	0.059163
6	-	0.059542	0.065316
8	-	0.056849	0.055439
12	-	0.054794	0.054404
16	-	0.048021	0.052502
20	-	0.045488	0.051346
32	-	0.055369	0.050358
50	-	0.055524	0.054464
80	-	0.054730	0.050457
100	-	0.053519	0.055099

Таблица 3: Зависимость от количества потоков

Наилучшие показатели были получены при 16 и 20 потоках.

При 16 потоках, прирост производительности составил:

- 67% - OpenMP;
- 64% - Pthreads.

При 20 потоках, прирост производительности составил:

- 68% - OpenMP;
- 64% - Pthreads.

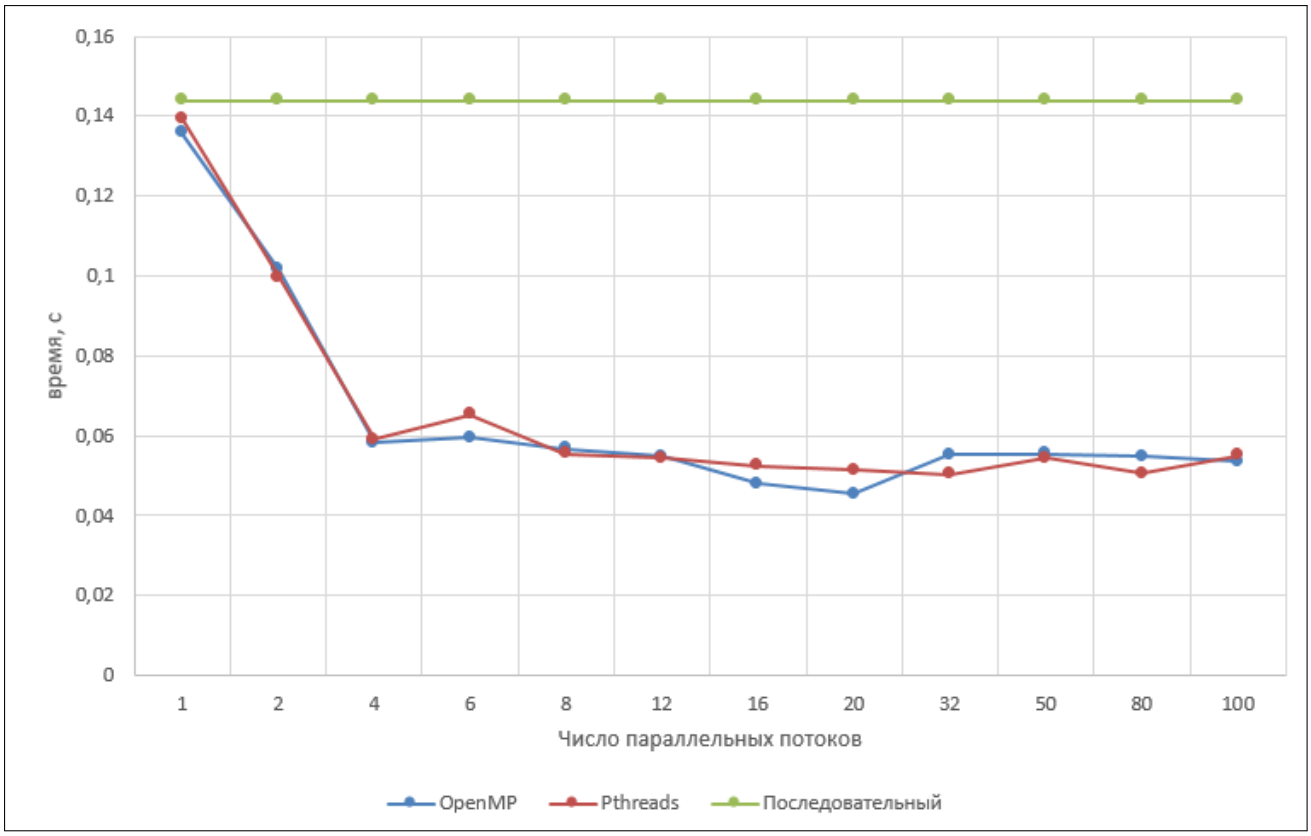

Рис. 4: Зависимость от числа выделенных потоков

5.1.3 Эксперимент 3

Количество потоков: 20
Количество узлов: ∼10 000 000

В данном эксперименте проводится многократный запуск при одних и тех же характеристиках, для того чтобы вычислить:

- математическое ожидание;
- дисперсию;
- доверительный интервал для оценки среднего.

Что позволит более объективно оценить результаты алгоритмов.

Последовательный	OpenMP	Pthreads
0.137561	0.060061	0.054078
0.145025	0.059195	0.062078
0.125417	0.056599	0.061191

0.125292	0.052370	0.050059
0.125991	0.062664	0.054928
0.124468	0.063160	0.056934
0.126880	0.056418	0.049538
0.126643	0.059022	0.054152
0.148706	0.053417	0.050933
0.127368	0.053595	0.054733

Таблица 4: Тестовая выборка для анализа

Характеристика	**Последовательный**	**OpenMP**	**Pthreads**
Среднее значение	0.1313351	0.0576501	0.0548624
Дисперсия	3,89155E-06	1,22757E-07	4,45013E-06
Доверительный интервал (P = 0.95)	[0,125742485 - 0,131245679]	[0,05529049 - 0,057612372]	[0,05221334 - 0,054820044]

Таблица 5: Вероятностные характеристики

Как видно из представленных характеристик, **pthreads** является лучшим решением. Хоть у него и высокая дисперсия, средняя скорость вычисления используя его выше чем у OpenMP.

Вывод

В данной работе были рассмотрены методы распараллеливания программ с использованием **OpenMP** и **Pthreads**.

Реализация на OpenMP заняла меньшее количество строк кода, по сравнению с Pthreads. Например, в Pthreads необходимо использовать функцию **pthread_join** для синхронизации потоков, в то время как в OpenMp это контроллирует сам фреймворк.

Эксперименты показали, что прирост проиводительности начался при наличии в дереве более 100 000 узлов. Наилучшие результаты были получены при 20 потоках, где удалось добиться прироста производительность в 68% для OpenMP и 64% для Pthreads. Само вычисление было выполнено в 3.2 раза быстре последовательного решения. Возможно, данный показатель в 3.2 раза, можно повысить если избавиться от многих процессов, работающих в фоне.

Отсюда можно сделать вывод, что распараллеливании программ имеет смысл в трудоемких задачах, в то время как в тривиальных задачах, последовательное решение будет быстрее.

Приложение 1

```
1  #include <stdio.h>
2  #include <ctime>
3  #include <fstream>
4  #include <omp.h>
5  #include <math.h>
6
7  #include "Tree.h"
8  #include "TreeUtils.h"
9
10 using namespace std;
11
12 double startTime, endTime;     // временные засечки
13
14 int log2(int n ){
15     return int(log(n)/log(2));
16 }
17
18 void defaultSum(){
19     tnode* tree = importTreeFromFile();
20
21     startTime = omp_get_wtime();
22     unsigned long long sum = getSumOfAllChilds(tree);
23     endTime = omp_get_wtime();
24
25     printf("[Default] Time: %lf\n", endTime - startTime);
26     printf("[Default] Sum: %llu\n", sum);
27 }
28
29 void pthreadSum(){
30     tnode* tree = importTreeFromFile();
31
32     int threads = omp_get_num_procs();
33     unsigned long long elementCount = getCountOfFile();
34     printf("Threads: %i, count of elements: %llu\n", threads, elementCount);
35
36     pthreadArg arg;
37     arg.tree = tree;
38     arg.threadCount = threads;
39
40     startTime = omp_get_wtime();
41     getSumOfAllChilds_Pthread((void *) &arg);
42     endTime = omp_get_wtime();
43
```

```
44      printf("[Pthread] Time: %lf\n", endTime - startTime);
45      printf("[Pthread] Sum: %llu\n", arg.tree->sum);
46  }
47
48  void openMpSum(){
49      tnode* tree = importTreeFromFile();
50
51      int threads = omp_get_num_procs();
52      unsigned long long elementCount = getCountOfFile();
53      printf("Procs: %i, count of elements: %llu\n", threads, elementCount);
54
55      omp_set_nested(1);
56      omp_set_max_active_levels((log2(20)));
57
58      startTime = omp_get_wtime();
59      unsigned long long sum = getSumOfAllChilds_OpenMP(tree);
60      endTime = omp_get_wtime();
61
62      printf("[OpenMP] Time: %lf\n", endTime - startTime);
63      printf("[OpenMP] Sum: %llu\n", sum);
64  }
65
66  int main(){
67      tnode* tree = makeRandomTree();
68      exportTreeToFile(tree);
69
70      defaultSum();
71      openMpSum();
72      pthreadSum();
73
74      return 0;
75  }
```

Листинг 5: Main.cpp

Приложение 2

```
1  #pragma once
2
3  #include <iostream>
4
5  using namespace std;
6
```

```
7  /* Структура бинарного дерева*/
8  struct tnode
9  {
10     unsigned long long value = 0;    // числовое значение
11     unsigned long long sum = 0; // сумма значений дочерних узлов
12     struct tnode *left = NULL;  // левый потомок
13     struct tnode *right =NULL;  // правый потомок
14 };
15
16 /* Добавить узел */
17 tnode* addNode(unsigned long long v, tnode *tree);
```

Листинг 6: Tree.h

Приложение 3

```
1  #include <stdio.h>
2  #include <fstream>
3  #include <iostream>
4
5  #include "Tree.h"
6
7  using namespace std;
8
9  /* Добавить узел */
10 tnode* addNode(unsigned long long v, tnode *tree)
11 {
12     // Если дерева нет, то формируем корень
13     if (tree == NULL)
14     {
15         tree = new tnode;        // память под узел
16         tree->value = v;         // значение
17         tree->sum = 0;           // сумма дочерних
18         tree->left = NULL;       // ветви инициализируем пустотой
19         tree->right = NULL;
20     }
21     else if (v < tree->value)   // условие добавление левого потомка
22         tree->left = addNode(v, tree->left);
23     else if(v > tree->value)
24         tree->right = addNode(v, tree->right);
25     return(tree);
26 }
```

Листинг 7: Tree.cpp

Приложение 4

```
#pragma once

#include "Tree.h"

/* Константы */
#define MAX_VALUE 0xFFFFFFFF          // 4 294 967 295 максимальное( значения для
    ↪ генерации)
#define GENERATE_COUNT 9999999       // количество генераций случайного числа
extern const char* EXTERNAL_FILE;    // внеший файл для экспорта и импорта дерева

#define SUCCESS 0
#define ERROR_CREATE_THREAD -1
#define ERROR_JOIN_THREAD   -2

struct pthreadArg {
    struct tnode *tree;
    int threadCount;
};

/* Прототипы функций */
tnode* makeRandomTree();
void exportTreeToFile(tnode* tree);
tnode* importTreeFromFile();
unsigned long long getCountOfFile();
unsigned long long getSumOfAllChilds(tnode* tree);
unsigned long long getSumOfAllChilds_OpenMP(tnode* tree);
void* getSumOfAllChilds_Pthread(void *args);
```

Листинг 8: TreeUtils.h

Приложение 5

```
#include <ctime>
#include <fstream>
#include <omp.h>
```

```
4
5  #include "TreeUtils.h"
6
7  const char* EXTERNAL_FILE = "externalFile.txt";
8
9  /*
10     Генерация случайного значения типа unsigned long long.
11
12     Максимальный размер unsigned long long — FFFF FFFF FFFF FFFF
13     или 18 446 744 073 709 551 615. В treeUtils.h задана
14     константа максимального значения(4 294 967 295), для того,
15     чтобы несколько ограничить диапазон для генерации.
16 */
17 unsigned long long llrand() {
18     unsigned long long r = 0;
19
20     /*
21         Запись long long занимает 64 бита, а стандартный оператор
22         rand() позволяет генерировать значение лишь до 32767 (15 бит).
23         Для покрытия всех битовых значений, используется побитовый
24         сдвиг(15 раз за итерацию).
25
26         По завершению цикла, биты полученного значения обрезаются,
27         в соответствии с максимальным значением.
28     */
29     for (int i = 0; i < 5; ++i) {
30         r = (r << 15) | rand();
31     }
32
33     return r & MAX_VALUE;
34 }
35
36 tnode* makeRandomTree() {
37     /*
38         Для равномерного распределения значений между ветками корня дерева,
39         значение корня равно половине от максимального возможного случайного числа
40     */
41     tnode* tree = new tnode;
42     tree->value = MAX_VALUE / 2;
43
44     srand(unsigned(time(NULL)));
45     for (int i = 0; i < GENERATE_COUNT; i++) {
46         unsigned long long random = llrand();
47         addNode(random, tree);
48     }
```

```
49
50      printf("[+] Random tree generated\n");
51      return tree;
52  }
53
54  void writeBinaryTree(tnode *tree, ostream &out) {
55      if (!tree) {
56          out << "# ";
57      }
58      else {
59          out << tree->value << " ";
60          writeBinaryTree(tree->left, out);
61          writeBinaryTree(tree->right, out);
62      }
63  }
64
65  bool readNextNum(ifstream &fin, long long &num) {
66      while (fin.peek() == ' ')
67          fin.ignore();
68
69      bool bNum = false;
70      char c = fin.peek();
71      if (c >= '0' && c <= '9') {
72          fin >> num;
73          bNum = true;
74      }
75      else
76          fin.ignore();
77
78      return bNum;
79  }
80
81  void readBinaryTree(tnode *&tree, ifstream &fin) {
82      if (fin.eof())
83          return;
84
85      long long num;
86      if (readNextNum(fin, num)){
87          tree = new tnode;
88          tree->value = num;
89          readBinaryTree(tree->left, fin);
90          readBinaryTree(tree->right, fin);
91      }
92  }
93
```

```
94  void exportTreeToFile(tnode* tree) {
95      filebuf fb;
96      fb.open(EXTERNAL_FILE, ios::out);
97      ostream out(&fb);
98
99      writeBinaryTree(tree, out);
100
101     printf("[+] Tree exported to file %s\n", EXTERNAL_FILE);
102 }
103
104 tnode* importTreeFromFile() {
105     ifstream fin;
106     fin.open(EXTERNAL_FILE, ios::in);
107
108     tnode* tree;
109     readBinaryTree(tree, fin);
110     fin.close();
111
112     printf("[+] Tree imported from file %s\n", EXTERNAL_FILE);
113
114     return tree;
115 }
116
117 unsigned long long getCountOfFile(){
118     unsigned long long count = 0;
119
120     ifstream fin;
121     fin.open(EXTERNAL_FILE, ios::in);
122
123     bool readingNum = false;
124     while(fin.peek() != EOF){
125         char c ;
126         fin.get(c);
127         if (c >= '0' && c <= '9')
128             readingNum = true;
129         else if(readingNum){
130             readingNum = false;
131             count++;
132         }
133     }
134
135     fin.close();
136     return count;
137 }
138
```

```
139 unsigned long long getSumOfAllChilds(tnode* tree) {
140     if (tree != NULL){
141         unsigned long long leftSum = 0;
142         unsigned long long rightSum = 0;
143 
144         if (tree->left != NULL) {
145             tree->left->sum = getSumOfAllChilds(tree->left);
146             leftSum = tree->left->sum + tree->left->value;
147         }
148 
149         if (tree->right != NULL)
150         {
151             tree->right->sum = getSumOfAllChilds(tree->right);
152             rightSum = tree->right->sum + tree->right->value;
153         }
154 
155         return leftSum + rightSum;
156     }
157     return 0;
158 }
159 
160 unsigned long long getSumOfAllChilds_OpenMP(tnode* tree) {
161     if (tree != NULL) {
162         unsigned long long leftSum = 0;
163     unsigned long long rightSum = 0;
164 
165     if (omp_get_active_level() >= omp_get_max_active_levels())
166             return getSumOfAllChilds(tree);
167 
168         #pragma omp parallel num_threads(2)
169         {
170             #pragma omp sections
171             {
172                 #pragma omp section
173                 {
174                     // сумма потомков для левого поддерева
175                     if (tree->left != NULL){
176                         tree->left->sum = getSumOfAllChilds_OpenMP(tree->left);
177                         leftSum = tree->left->sum + tree->left->value;
178                     }
179         }
180 
181         #pragma omp section
182         {
183                     // сумма потомков для правого поддерева
```

```
184                     if (tree->right != NULL){
185                         tree->right->sum = getSumOfAllChilds_OpenMP(tree->right
    ↪ );
186                         rightSum = tree->right->sum + tree->right->value;
187                     }
188         }
189             }
190     }
191         return leftSum + rightSum;
192     }
193     return 0;
194 }
195
196 void* getSumOfAllChilds_Pthread(void *args){
197     pthreadArg *arg = (pthreadArg *)args;
198
199     if (arg->tree != NULL){
200         unsigned long long leftSum = 0;
201     unsigned long long rightSum = 0;
202
203     // Когда доступно 1 или менее потоков, используется метод без распараллеливания
204     if (arg->threadCount <= 1){
205             arg->tree->sum = getSumOfAllChilds(arg->tree);
206             return 0;
207     }
208         int leftJoinStatus, rightJoinStatus; // статус pthread_join
209         int leftCreateStatus, rightCreateStatus;// статус завершения
    ↪ pthread_create
210
211     // поток для левого поддерева
212     pthread_t leftThread;
213     pthreadArg leftArg;
214     if (arg->tree->left != NULL){
215             leftArg.tree = arg->tree->left;
216             leftArg.threadCount = arg->threadCount/2;
217             leftCreateStatus = pthread_create(&leftThread, NULL,
    ↪ getSumOfAllChilds_Pthread, (void*) &leftArg);
218             if (leftCreateStatus != 0) {
219                 printf("[ERROR] Can't create thread. Status: %d\n",
    ↪ leftCreateStatus);
220         exit(ERROR_CREATE_THREAD);
221             }
222     }
223
224     // поток для правого поддерева
```

```
225     pthread_t rightThread;
226     pthreadArg rightArg;
227     if (arg->tree->right != NULL){
228             rightArg.tree = arg->tree->right;
229             rightArg.threadCount =  arg->threadCount/2;
230             rightCreateStatus = pthread_create(&rightThread, NULL,
    ↪ getSumOfAllChilds_Pthread, (void*) &rightArg);
231             if (rightCreateStatus != 0) {
232                 printf("[ERROR] Can't create thread. Status: %d\n",
    ↪ rightCreateStatus);
233          exit(ERROR_CREATE_THREAD);
234             }
235     }
236
237         // ожидание завершения потоков
238     leftCreateStatus = pthread_join(leftThread, (void**)&leftJoinStatus);
239     if (leftCreateStatus != SUCCESS) {
240             printf("[ERROR] Can't join thread. Status: %d\n", leftCreateStatus)
    ↪ ;
241             exit(ERROR_JOIN_THREAD);
242     }
243     if (arg->tree->left != NULL){
244             arg->tree->left->sum = leftArg.tree->sum;
245             leftSum = arg->tree->left->sum + arg->tree->left->value;
246     }
247
248     rightCreateStatus = pthread_join(rightThread, (void**)&rightJoinStatus);
249     if (rightCreateStatus != SUCCESS) {
250             printf("[ERROR] Can't join thread. Status: %d\n", rightCreateStatus
    ↪ );
251             exit(ERROR_JOIN_THREAD);
252     }
253     if (arg->tree->right != NULL){
254             arg->tree->right->sum = rightArg.tree->sum;
255             rightSum = arg->tree->right->sum + arg->tree->right->value;
256     }
257
258     arg->tree->sum = leftSum + rightSum;
259     }
260     return 0;
261 }
```

Листинг 9: TreeUtils.cpp