Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе

по курсу «Параллельные вычисления»

по теме «Создание многопоточных программ на языке C++ с использованием Pthreads и OpenMP»

Выполнил студент гр. 13541/2:
Ерниязов Т.Е.

Проверил преподаватель:
Стручков И.В.

Санкт-Петербург
2019 г.

1 Цель работы

1.1 Постановка задачи

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 Характеристики системы

Работа производилась на реальной системе, со следующими характеристиками:

Элемент	**Значение**
Имя ОС	Майкрософт Windows 10 Home
Установленная оперативная память (RAM)	32,00 ГБ
Процессор	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 2208 МГц, ядер: 6, логических процессоров: 12
Тип системы	64-разрядная операционная система

Таблица 1: Сведения о системе

3 Структура проекта

Структура проекта выглядит следующим образом:

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.

Листинг 1: Отрывок Tree.h

```
7
8  struct tnode
9  {
10   unsigned long long value = 0;
11   unsigned long long sum = 0;
12   struct tnode *left = NULL;
13   struct tnode *right =NULL;
14 };
15
16
17 tnode* addNode(unsigned long long v, tnode *tree);
```

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**.

Помимо функций счета суммы дочерних узлов разными способами, там также есть функция для генерации дерева с помощью генератора псевдослучайных чисел, в качестве seed ему дается текущее время.

Полный код приведен в листинге 8.

4 Алгоритм решения

4.1 Последовательная реализция

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

Листинг 2: Отрывок TreeUtils.cpp

```
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 }
```

4.2 Параллельный алгоритм с использованием Pthreads

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

Каждый вызов подсчета суммы потомков выполняется в отдельном потоке, это происходит до дех пор, пока имеются свободные логические процессоры, после их исчерпания, подсчет выполняется последовательно.

Для распараллеливания будем выбирать определенный уровень дерева, на каждый элемент которого выделяется по потоку. Например, если это уровень 3, то для его распараллеливания требуется 4 потока. Также, необходимо заполнить предыдущие уровни потоками, из которых создаются дочернии потоки. Поэтому для 3 уровня в сумме требуется 7 потоков.

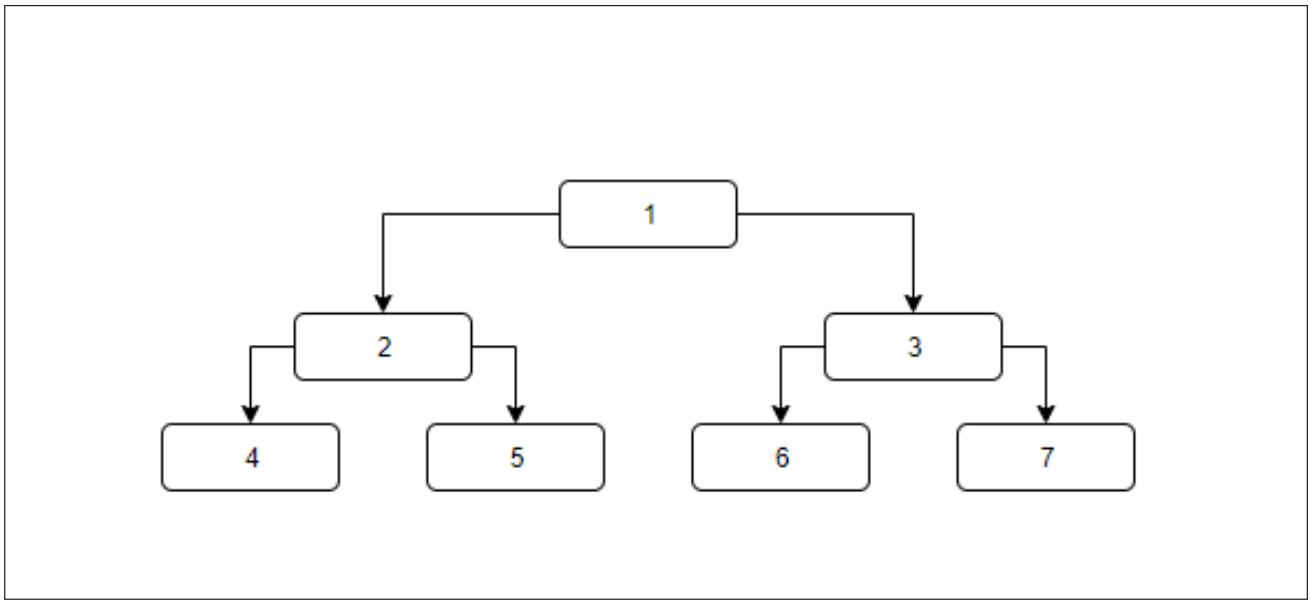

Рис. 2: Создание вложенности

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

Исходный код приведен в листинге 8.

4.3 Параллельный алгоритм с использованием OpenMP

Алгоритм для OpenMP аналогичен Pthreads. С помощью функции opm_set_nested() можно включить вложенную параллельность. Далее, с помощью mp_get_active_level () и omp_get_max_active_levels () можно узнавать текущий уровень вложенности и максимально установленный. Таким образом реализация параллельности через OpenMP и Pthreads схожа, так как необходимо задавать конкретный уровень в дереве, который будет распараллелен.

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.

Листинг 3: Отрывок TreeUtils.cpp

```
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
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
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 }
```

5 Тестирование

5.1 Эксперименты

5.1.1 Эксперимент 1

Количество потоков: 7
Количество узлов: от 1000 до ∼100 000 000

Число узлов	Последовательный	OpenMP	Pthreads
1000	0.0000001	0.001962	0.001001
10000	0.0000977	0.0001137	0.000998
100000	0.002988	0.002962	0.002991
1000000	0.071808	0.032921	0.032861
10000000	0.867665	0.298656	0.339623
100000000	10.620637	5.096909	5.037491

Таблица 2: Зависимость от количества узлов

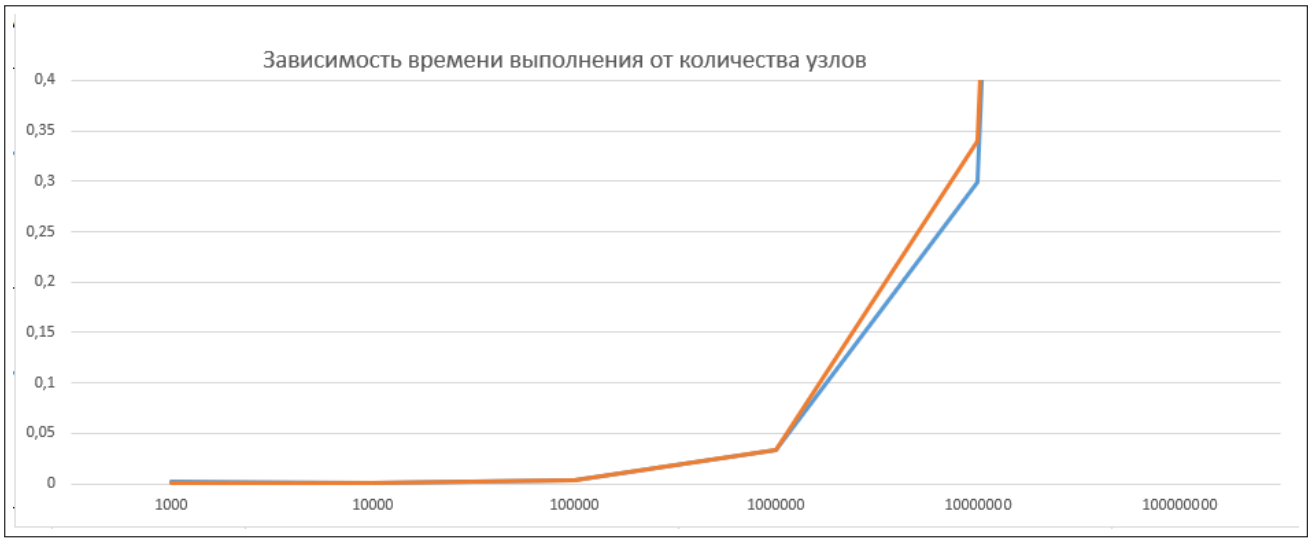

Рис. 3: Зависимость времени от количества узлов

Оранжевым цветом отмечен график OpenMP, а синим Pthreads.
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 узлов: ∼1000 000

Уровень дерева	Последовательный	OpenMP	Pthreads
1	0.074722	0.076929	0.076598
2	-	0.040453	0.039092
3	-	0.031054	0.029388
4	-	0.026451	0.032337
5	-	0.025190	0.028274
6	-	0.023903	0.023047
7	-	0.020494	0.017593

Таблица 3: Зависимость от количества потоков

Наилучшие показатели были получены при 7 уровне.
При 7 уровне дерева, прирост производительности составил:

- 73% - OpenMP;
- 77% - Pthreads.

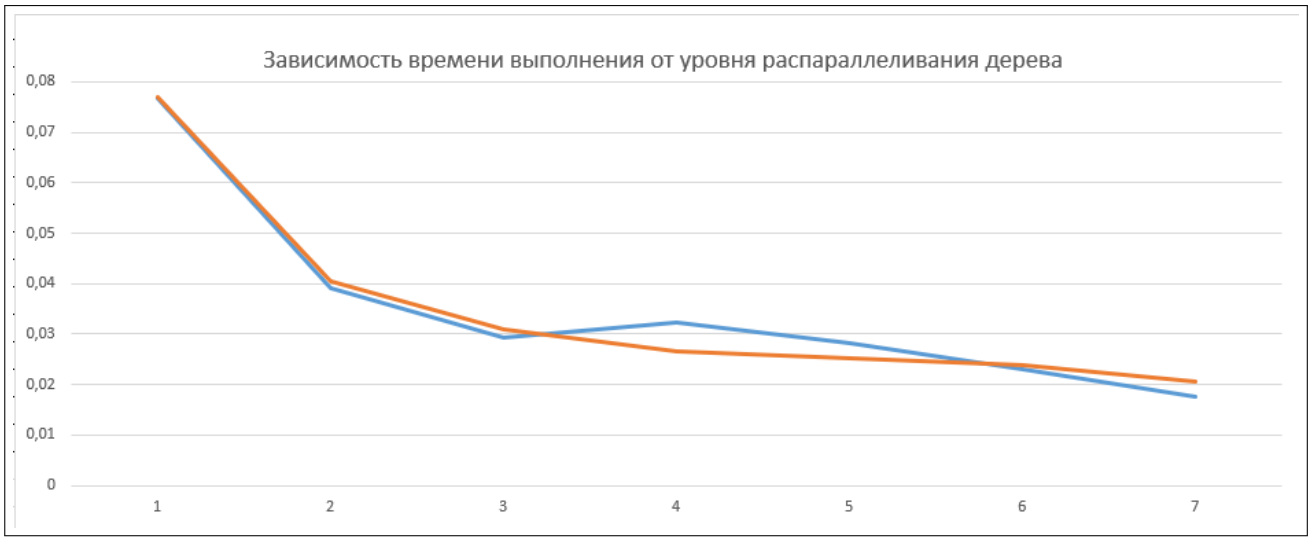

Рис. 4

Оранжевым цветом отмечен график OpenMP, а синим Pthreads.

5.1.3 Эксперимент 3

Уровень дерева: 7
Количество узлов: ∼1 000 000

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**
0.057854	0.017542	0.018543
0.062498	0.016095	0.018358
0.057588	0.018543	0.018545
0.059537	0.019054	0.018175
0.058706	0.021475	0.022548
0.057586	0.017537	0.019528
0.058916	0.016553	0.019520
0.056577	0.017567	0.015146
0.061485	0.016566	0.016592
0.057659	0.018544	0.019520

Таблица 4: Тестовая выборка для анализа

Характеристика	**Последовательный**	**OpenMP**	**Pthreads**
Среднее значение	0.05884	0.0179401	0.0186427
Дисперсия	0.00000315305136	0.00000221220345	0.00000342634181
Доверительный интервал (P = 0.95)	(0.0576;0.0601)	(0.0169;0.019)	(0.0173;0.02)

Таблица 5: Вероятностные характеристики

Как видно из представленных характеристик, **openMP** является лучшим решением. У него лучше средняя скорость вычислений и дисперсия чем у pthreads.

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 деревее более 100 000 узлов. Наилучшие результаты были получены на 7 уровне распараллеливания, где удалось добиться прироста производительность в 73% для OpenMP и 77% для Pthreads. Возможно, данный показатель, можно повысить если избавиться от многих процессов, работающих в фоне. Также для достижения максимального выигрыша в производительности в этой задаче необходимо, чтобы дерево было как можно более сбалансировано. Так как, если на выбранном уровне распараллеливания один из элементов не существует, то поток будет простаивать. Более того, операция сложения является слишком примитивной, поэтому на ней меньше виден прирост производительности.

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

Листинг 4: main.cpp

```
#include <stdio.h>
#include <ctime>
#include <fstream>
#include <omp.h>
#include <math.h>
#include <chrono>
#include <ctime>
#include <sched.h>
#include <stdlib.h>
#include <iostream>
#include <stdint.h>
#include <string>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include "Tree.h"
#include "TreeUtils.h"
#include <cmath>

using namespace std;

std::chrono::system_clock::time_point timeBefore, timeAfter;


const int level = 6;

void defaultSum(tnode* tree){
    timeBefore = std::chrono::high_resolution_clock::now();
    unsigned long long sum = getSumOfAllChilds(tree);
    timeAfter = std::chrono::high_resolution_clock::now();

    printf("[Default] Time: %lf\n", std::chrono::duration_cast<std::chrono::duration<double>>(timeAfter - timeBefore).count());
    printf("[Default] Sum: %llu\n", sum);
}

void pthreadSum(tnode* tree){

    int threads = pow(2, level) - 1; ;
    pthreadArg arg;
    arg.tree = tree;
    arg.threadCount = threads;
    printf("[Pthread] Number of threads: %llu\n", threads);
    timeBefore = std::chrono::high_resolution_clock::now();
    getSumOfAllChilds_Pthread((void *) &arg);
    timeAfter = std::chrono::high_resolution_clock::now();

    printf("[Pthread] Time: %lf\n", std::chrono::duration_cast<std::chrono::duration<double>>(timeAfter - timeBefore).count());
    printf("[Pthread] Sum: %llu\n", arg.tree->sum);
}

void openMpSum(tnode* tree){


    omp_set_nested(1);
    omp_set_max_active_levels(level-1);

    timeBefore = std::chrono::high_resolution_clock::now();
    unsigned long long sum = getSumOfAllChilds_OpenMP(tree);
    timeAfter = std::chrono::high_resolution_clock::now();

    printf("[OpenMP] Time: %lf\n", std::chrono::duration_cast<std::chrono::duration<double>>(timeAfter - timeBefore).count());
    printf("[OpenMP] Sum: %llu\n", sum);
}

int main(){
    tnode* tree = makeRandomTree();

    defaultSum(tree);
    openMpSum(tree);
    pthreadSum(tree);

    return 0;
}
```

Приложение 2

Листинг 5: Tree.h

```
#pragma once

#include <iostream>

using namespace std;


struct tnode
{
  unsigned long long value = 0;
  unsigned long long sum = 0;
  struct tnode *left = NULL;
  struct tnode *right =NULL;
};


tnode* addNode(unsigned long long v, tnode *tree);

tnode* makeNewTree(unsigned long long v, tnode *tree);
```

Приложение 3

Листинг 6: Tree.cpp

```
#include <stdio.h>
#include <fstream>
#include <iostream>

#include "Tree.h"

using namespace std;

tnode* addNode(unsigned long long v, tnode *tree)
{
  if (tree == NULL)
  {
    tree = makeNewTree(v, tree);
  }
  else if (v < tree->value)
    tree->left = addNode(v, tree->left);
  else if(v > tree->value)
    tree->right = addNode(v, tree->right);
  return(tree);
}


tnode* makeNewTree(unsigned long long v, tnode *tree)
{
    tree = new tnode;
    tree->value = v;
    tree->sum = 0;
    tree->right = NULL;
    tree->left = NULL;
    return(tree);
}
```

Приложение 4

Листинг 7: TreeUtils.h

```
#pragma once

#include "Tree.h"

#define MAX_VALUE 0xFFFFFFFF
#define GENERATE_COUNT 1000000

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

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

tnode* makeRandomTree();
unsigned long long getSumOfAllChilds(tnode* tree);
unsigned long long getSumOfAllChilds_OpenMP(tnode* tree);
void* getSumOfAllChilds_Pthread(void *args);
```

Приложение 5

Листинг 8: TreeUtils.cpp

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

#include "TreeUtils.h"

unsigned long long llrand() {
    unsigned long long r = 0;

    for (int i = 0; i < 5; ++i) {
        r = (r << 15) | rand();
    }

    return r & MAX_VALUE;
}

tnode* makeRandomTree() {

    tnode* tree = new tnode;
    tree->value = MAX_VALUE / 2;

    srand(unsigned(time(NULL)));
    for (int i = 0; i < GENERATE_COUNT; i++) {
        unsigned long long random = llrand();
        addNode(random, tree);
    }

    printf("Random tree generated\n");
    printf("Number of nodes: %llu\n", GENERATE_COUNT);
    return tree;
}


unsigned long long getSumOfAllChilds(tnode* tree) {
  if (tree != NULL){
    unsigned long long leftSum = 0;
    unsigned long long rightSum = 0;

    if (tree->left != NULL) {
      tree->left->sum = getSumOfAllChilds(tree->left);
      leftSum = tree->left->sum + tree->left->value;
    }

    if (tree->right != NULL)
    {
      tree->right->sum = getSumOfAllChilds(tree->right);
      rightSum = tree->right->sum + tree->right->value;
    }

    return leftSum + rightSum;
  }
  return 0;
}

unsigned long long getSumOfAllChilds_OpenMP(tnode* tree) {
    if (tree != NULL) {
        unsigned long long leftSum = 0;
  unsigned long long rightSum = 0;

  if (omp_get_active_level() >= omp_get_max_active_levels ())
            return getSumOfAllChilds(tree);

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

                    if (tree->left != NULL){
                        tree->left->sum = getSumOfAllChilds_OpenMP(tree->left);
                        leftSum = tree->left->sum + tree->left->value;
                    }
    }

    #pragma omp section
    {

                    if (tree->right != NULL){
                        tree->right->sum = getSumOfAllChilds_OpenMP(tree->right);
                        rightSum = tree->right->sum + tree->right->value;
                    }
    }
            }
  }
```

```
87          return leftSum + rightSum;
88      }
89      return 0;
90  }
91
92  void* getSumOfAllChilds_Pthread(void *args){
93      pthreadArg *arg = (pthreadArg *)args;
94
95      if (arg->tree != NULL){
96          unsigned long long leftSum = 0;
97    unsigned long long rightSum = 0;
98
99
100   if (arg->threadCount <= 1){
101             arg->tree->sum = getSumOfAllChilds(arg->tree);
102             return 0;
103   }
104         int leftJoinStatus, rightJoinStatus;
105         int leftCreateStatus, rightCreateStatus;
106
107
108   pthread_t leftThread;
109   pthreadArg leftArg;
110   if (arg->tree->left != NULL){
111             leftArg.tree = arg->tree->left;
112             leftArg.threadCount = (arg->threadCount - 1)/2;
113             leftCreateStatus = pthread_create(&leftThread, NULL, getSumOfAllChilds_Pthread, (void*) &
                      leftArg);
114             if (leftCreateStatus != 0) {
115                 printf("[ERROR] Can't create thread. Status: %d\n", leftCreateStatus);
116     exit(ERROR_CREATE_THREAD);
117             }
118   }
119
120
121   pthread_t rightThread;
122   pthreadArg rightArg;
123   if (arg->tree->right != NULL){
124             rightArg.tree = arg->tree->right;
125             rightArg.threadCount =  (arg->threadCount - 1)/2;
126             rightCreateStatus = pthread_create(&rightThread, NULL, getSumOfAllChilds_Pthread, (void*) &
                      rightArg);
127             if (rightCreateStatus != 0) {
128                 printf("[ERROR] Can't create thread. Status: %d\n", rightCreateStatus);
129     exit(ERROR_CREATE_THREAD);
130             }
131   }
132
133
134   leftCreateStatus = pthread_join(leftThread, (void**)&leftJoinStatus);
135   if (leftCreateStatus != SUCCESS) {
136             printf("[ERROR] Can't join thread. Status: %d\n", leftCreateStatus);
137             exit(ERROR_JOIN_THREAD);
138   }
139   if (arg->tree->left != NULL){
140             arg->tree->left->sum = leftArg.tree->sum;
141             leftSum = arg->tree->left->sum + arg->tree->left->value;
142   }
143
144   rightCreateStatus = pthread_join(rightThread, (void**)&rightJoinStatus);
145   if (rightCreateStatus != SUCCESS) {
146             printf("[ERROR] Can't join thread. Status: %d\n", rightCreateStatus);
147             exit(ERROR_JOIN_THREAD);
148   }
149   if (arg->tree->right != NULL){
150             arg->tree->right->sum = rightArg.tree->sum;
151             rightSum = arg->tree->right->sum + arg->tree->right->value;
152   }
153
154   arg->tree->sum = leftSum + rightSum;
155     }
156     return 0;
157 }
```