

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:
Ернязов 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:

```

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.

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 реализация

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         return leftSum + rightSum;
134     }
135     return 0;
136 }
137

```

4.2 Параллельный алгоритм с использованием Pthreads

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

Каждый вызов подсчета суммы потомков выполняется в отдельном потоке, это происходит до тех 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 дочерние потоки. Поэтому для 3 уровня в сумме требуется 7 потоков.

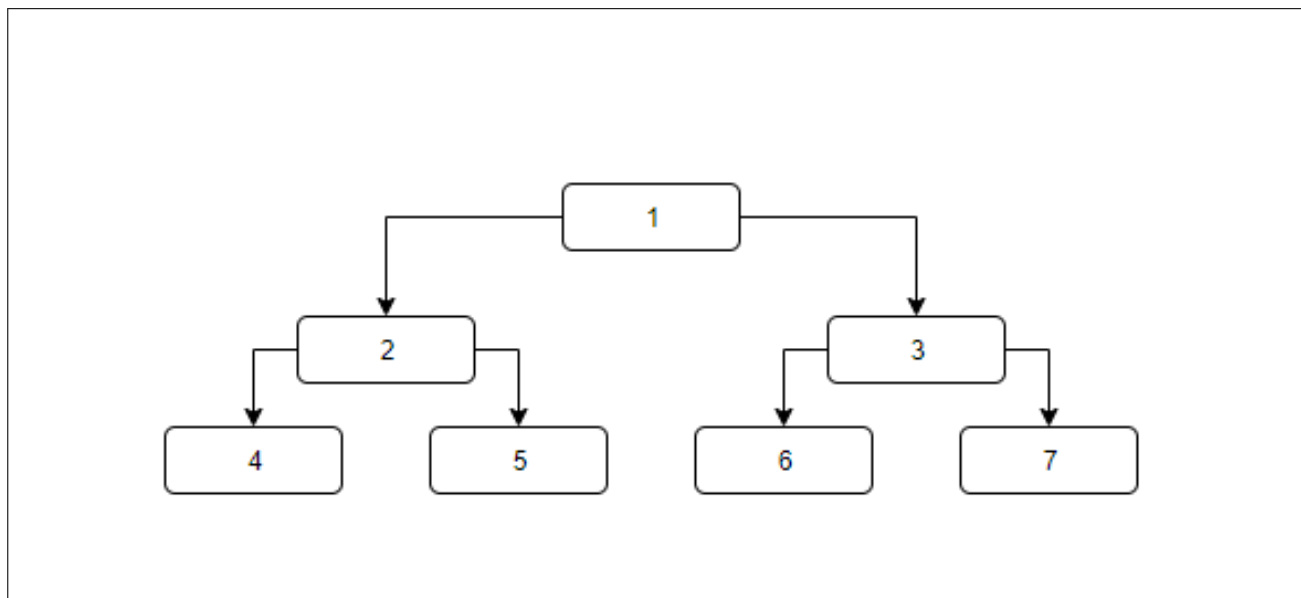


Рис. 2: Создание вложенности

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 `omp_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 директивы:

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

Код каждой директивы `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 число 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.

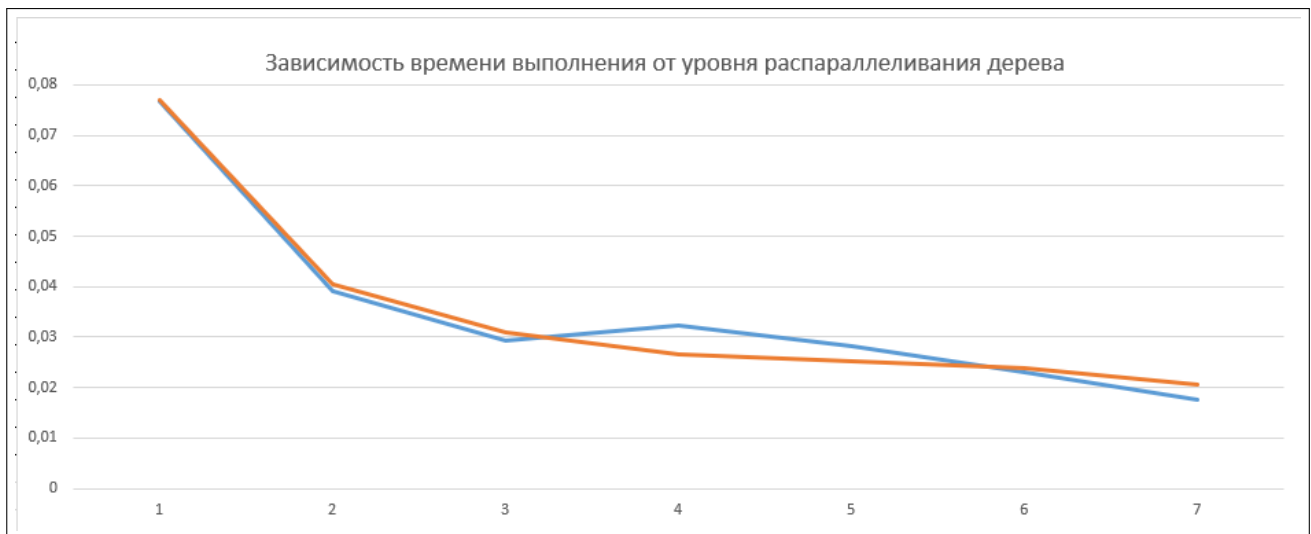


Рис. 4

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 контролирует сам фреймворк.

Эксперименты показали, что прирост производительности начался при наличии в дереве более 100 000 узлов. Наилучшие результаты были получены на 7 уровне распараллеливания, где удалось добиться прироста производительности 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

```
1  #include <stdio.h>
2  #include <ctime>
3  #include <fstream>
4  #include <omp.h>
5  #include <math.h>
6  #include <chrono>
7  #include <ctime>
8  #include <sched.h>
9  #include <stdlib.h>
10 #include <iostream>
11 #include <stdint.h>
12 #include <string>
13 #include <sched.h>
14 #include <unistd.h>
15 #include <stdlib.h>
16 #include <stdio.h>
17 #include "Tree.h"
18 #include "TreeUtils.h"
19 #include <cmath>
20
21 using namespace std;
22
23 std::chrono::system_clock::time_point timeBefore, timeAfter;
24
25
26 const int level = 6;
27
28 void defaultSum(tnode* tree){
29     timeBefore = std::chrono::high_resolution_clock::now();
30     unsigned long long sum = getSumOfAllChilds(tree);
31     timeAfter = std::chrono::high_resolution_clock::now();
32
33     printf("[Default] Time: %lf\n", std::chrono::duration_cast<std::chrono::duration<double>>(timeAfter -
34         timeBefore).count());
35     printf("[Default] Sum: %llu\n", sum);
36 }
37
38 void pthreadSum(tnode* tree){
39
40     int threads = pow(2, level) - 1; ;
41     pthreadArg arg;
42     arg.tree = tree;
43     arg.threadCount = threads;
44     printf("[Pthread] Number of threads: %llu\n", threads);
45     timeBefore = std::chrono::high_resolution_clock::now();
46     getSumOfAllChilds_Pthread((void *) &arg);
47     timeAfter = std::chrono::high_resolution_clock::now();
48
49     printf("[Pthread] Time: %lf\n", std::chrono::duration_cast<std::chrono::duration<double>>(timeAfter -
50         timeBefore).count());
51     printf("[Pthread] Sum: %llu\n", arg.tree->sum);
52 }
53
54 void openMpSum(tnode* tree){
55
56     omp_set_nested(1);
57     omp_set_max_active_levels(level-1);
58
59     timeBefore = std::chrono::high_resolution_clock::now();
60     unsigned long long sum = getSumOfAllChilds_OpenMP(tree);
61     timeAfter = std::chrono::high_resolution_clock::now();
62
63     printf("[OpenMP] Time: %lf\n", std::chrono::duration_cast<std::chrono::duration<double>>(timeAfter -
64         timeBefore).count());
65     printf("[OpenMP] Sum: %llu\n", sum);
66 }
67
68 int main(){
69     tnode* tree = makeRandomTree();
70
71     defaultSum(tree);
72     openMpSum(tree);
73     pthreadSum(tree);
74
75     return 0;
76 }
```

Приложение 2

Листинг 5: Tree.h

```

1  #pragma once
2
3  #include <iostream>
4
5  using namespace std;
6
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);
18
19 tnode* makeNewTree(unsigned long long v, tnode *tree);

```

Приложение 3

Листинг 6: Tree.cpp

```

1  #include <stdio.h>
2  #include <fstream>
3  #include <iostream>
4
5  #include "Tree.h"
6
7  using namespace std;
8
9  tnode* addNode(unsigned long long v, tnode *tree)
10 {
11     if (tree == NULL)
12     {
13         tree = makeNewTree(v, tree);
14     }
15     else if (v < tree->value)
16         tree->left = addNode(v, tree->left);
17     else if (v > tree->value)
18         tree->right = addNode(v, tree->right);
19     return(tree);
20 }
21
22
23 tnode* makeNewTree(unsigned long long v, tnode *tree)
24 {
25     tree = new tnode;
26     tree->value = v;
27     tree->sum = 0;
28     tree->right = NULL;
29     tree->left = NULL;
30     return(tree);
31 }

```

Приложение 4

Листинг 7: TreeUtils.h

```

1  #pragma once
2
3  #include "Tree.h"
4
5  #define MAX_VALUE 0xFFFFFFFF
6  #define GENERATE_COUNT 1000000
7
8  #define SUCCESS 0
9  #define ERROR_CREATE_THREAD -1
10 #define ERROR_JOIN_THREAD -2
11
12 struct pthreadArg {
13     struct tnode *tree;
14     int threadCount;
15 };
16
17 tnode* makeRandomTree();
18 unsigned long long getSumOfAllChilds(tnode* tree);
19 unsigned long long getSumOfAllChilds_OpenMP(tnode* tree);
20 void* getSumOfAllChilds_Pthread(void *args);

```

Приложение 5

Листинг 8: TreeUtils.cpp

```
1  #include <ctime>
2  #include <fstream>
3  #include <omp.h>
4  #include <pthread.h>
5
6  #include "TreeUtils.h"
7
8  unsigned long long llrand() {
9      unsigned long long r = 0;
10
11      for (int i = 0; i < 5; ++i) {
12          r = (r << 15) | rand();
13      }
14
15      return r & MAX_VALUE;
16  }
17
18  tnode* makeRandomTree() {
19
20      tnode* tree = new tnode;
21      tree->value = MAX_VALUE / 2;
22
23      srand(unsigned(time(NULL)));
24      for (int i = 0; i < GENERATE_COUNT; i++) {
25          unsigned long long random = llrand();
26          addNode(random, tree);
27      }
28
29      printf("Random tree generated\n");
30      printf("Number of nodes: %llu\n", GENERATE_COUNT);
31      return tree;
32  }
33
34
35  unsigned long long getSumOfAllChilds(tnode* tree) {
36      if (tree != NULL){
37          unsigned long long leftSum = 0;
38          unsigned long long rightSum = 0;
39
40          if (tree->left != NULL) {
41              tree->left->sum = getSumOfAllChilds(tree->left);
42              leftSum = tree->left->sum + tree->left->value;
43          }
44
45          if (tree->right != NULL)
46          {
47              tree->right->sum = getSumOfAllChilds(tree->right);
48              rightSum = tree->right->sum + tree->right->value;
49          }
50
51          return leftSum + rightSum;
52      }
53      return 0;
54  }
55
56  unsigned long long getSumOfAllChilds_OpenMP(tnode* tree) {
57      if (tree != NULL) {
58          unsigned long long leftSum = 0;
59          unsigned long long rightSum = 0;
60
61          if (omp_get_active_level() >= omp_get_max_active_levels ())
62              return getSumOfAllChilds(tree);
63
64          #pragma omp parallel num_threads(2)
65          {
66              #pragma omp sections
67              {
68                  #pragma omp section
69                  {
70
71                      if (tree->left != NULL){
72                          tree->left->sum = getSumOfAllChilds_OpenMP(tree->left);
73                          leftSum = tree->left->sum + tree->left->value;
74                      }
75                  }
76
77                  #pragma omp section
78                  {
79
80                      if (tree->right != NULL){
81                          tree->right->sum = getSumOfAllChilds_OpenMP(tree->right);
82                          rightSum = tree->right->sum + tree->right->value;
83                      }
84                  }
85              }
86      }
```

```

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
105     int leftJoinStatus, rightJoinStatus;
106     int leftCreateStatus, rightCreateStatus;
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*) &
114             leftArg);
115         if (leftCreateStatus != 0) {
116             printf("[ERROR] Can't create thread. Status: %d\n", leftCreateStatus);
117             exit(ERROR_CREATE_THREAD);
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*) &
127             rightArg);
128         if (rightCreateStatus != 0) {
129             printf("[ERROR] Can't create thread. Status: %d\n", rightCreateStatus);
130             exit(ERROR_CREATE_THREAD);
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 }

```