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 MPI»

Выполнил студент гр. 13541/2:
Волкова Мария Дмитриевна

Проверил преподаватель:
Стручков И.В.

Санкт-Петербург
2019 г.

1 Цель работы

1.1 Постановка задачи

Вариант 6, MPI.
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	Ubuntu 14.04
Установленная оперативная память (RAM)	12,00 ГБ
Процессор	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 2208 МГц, ядер: 4
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
9  struct tnode
10 {
11   long value = 0;
12   long sum = 0;
13   long level = 0;
14   struct tnode *left = NULL;
15   struct tnode *right =NULL;
16 };
```

Значение и сумма потомков хранятся в переменной типа **long**.

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 константа, чтобы генерировалась одинаковая последовательность чисел (нужно для MPI).

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.

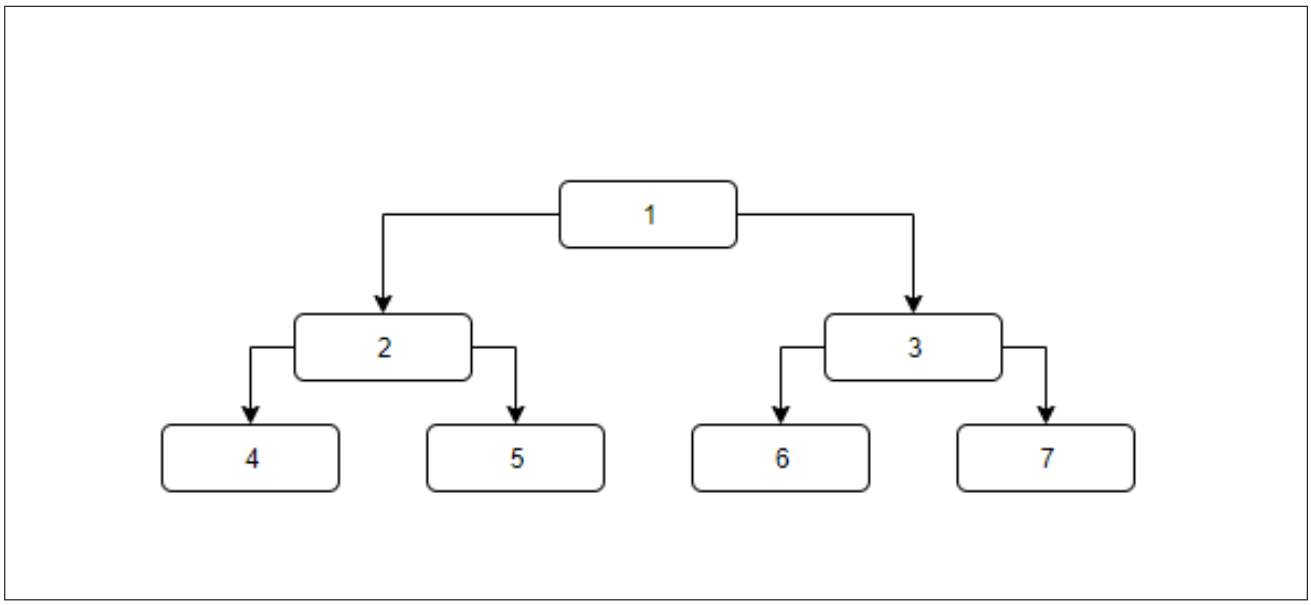

Рис. 2: Создание вложенности

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 MPI

Алгоритм для MPI аналогичен Pthreads. С помощью функции MPI_Isend можно сдлеать неблокирующую отправку сообщения (не ждет Recv от принимателя). Это необходимо для того, чтобы избежать взаимной блокировки (deadlock). Чтобы выдать каждому процессу конкретное поддерево, сначала запускается функция (prepareSubTreesForMPI()), которая собирает все узлы с определенного уровня, а потом каждый процесс забирает свой. После приема результата от каждого процесса, мастер процесс считает оставшиеся узлы. Таким образом реализация параллельности через MPI и Pthreads схожа, так как необходимо задавать конкретный уровень в дереве, который будет распараллелен.

Чтобы контролировать уровень дерева, в структуру узла дерева было внесено поле level, которое хранит информацию, на каком уровне лежит данный элемент.

Листинг 3: Отрывок TreeUtils.cpp

```
160 long getSumOfAllChilds_MPI(tnode* tree, int level, int rank, int numprocs) {
161   MPI_Request request;
162   long Result = 0;
163   MPI_Status status;
164   int tag = 50;
165   std::vector<tnode*> buf;
166   double starttime, endtime = 0.0;
167   prepareSubTreesForMPI(tree, level, &buf);
168
169
170   if (rank == 0) {
171     starttime= MPI_Wtime();
172   }
173
174   if (rank != 0) {
175       tnode* subTree = buf.at(rank-1);
```

```
176         long drobSum = getSumOfAllChilds(subTree) + subTree->value;
177         MPI_Isend(&drobSum, 1, MPI_LONG, 0, tag, MPI_COMM_WORLD, &request);
178     }
179     if (rank == 0) {
180       for (int i = 1; i < numprocs; i++) {
181         long res = 0;
182         MPI_Recv(&res, 1, MPI_LONG, i, tag, MPI_COMM_WORLD, &status);
183         Result += res;
184       }
185       Result = Result + getLastNodes(level, tree) - tree->value;
186       endtime = MPI_Wtime();
187       printf("[MPI]: Time %lf\n", endtime - starttime);
188       printf("[MPI]: Sum %llu\n", Result);
189
190     }
191
192     return 0;
193 }
```

5 Тестирование

5.1 Эксперименты

5.1.1 Эксперимент 1

Уровень дерева: 2
Количество узлов: от 1000 до ∼100 000 000

Число узлов	Последовательный	MPI	Pthreads
1000	0.0000001	0.002467	0.001131
10000	0.0000935	0.0002183	0.001265
100000	0.002811	0.003145	0.002991
1000000	0.074004	0.038129	0.032861
10000000	0.834581	0.318656	0.339623
100000000	10.007311	5.464815	5.037491

Таблица 2: Зависимость от количества узлов

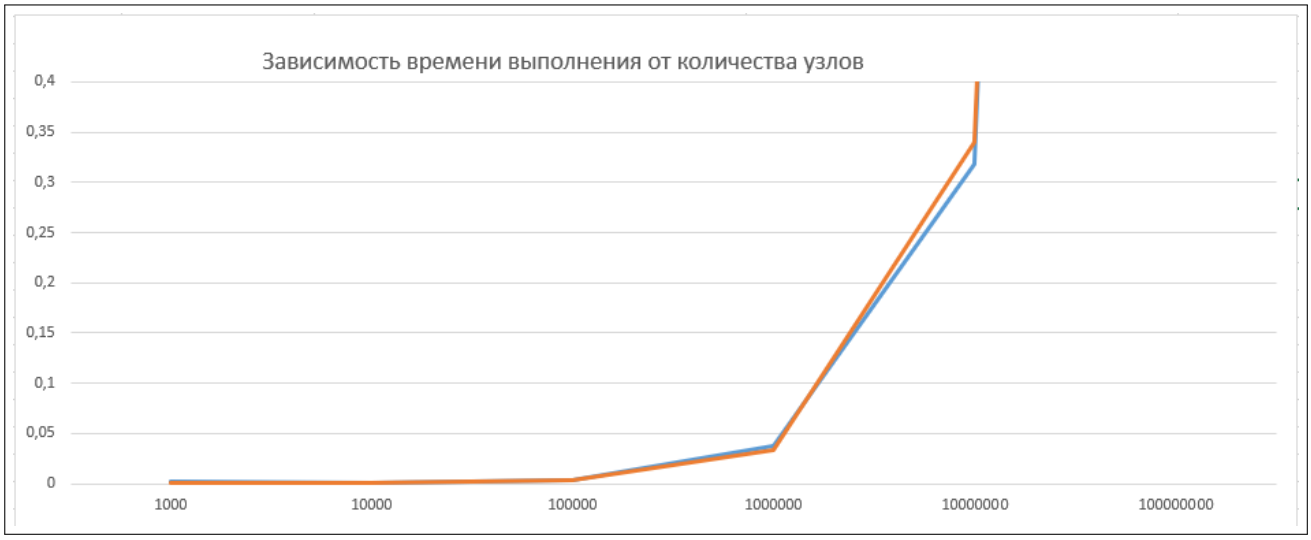

Рис. 3: Зависимость времени от количества узлов

Оранжевым цветом отмечен график Pthreads, а синим MPI.
Из эксперимента видно, что:

- до 100 000 элементов, лидировало последовательное решение, после чего, уступило параллельным решениям.
- MPI 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	MPI	Pthreads
1	0.074722	0.103929	0.074653
2	-	0.040453	0.037912
3	-	0.035054	0.031388
4	-	0.031401	0.029486

Таблица 3: Зависимость от количества потоков

Наилучшие показатели были получены при 4 уровне. При дальнейшем увеличении уровня pthread не позволял создавать больше потоков и выдавал ошибку.

При 4 уровне дерева, прирост производительности составил:

- 69% - MPI;
- 76% - Pthreads.

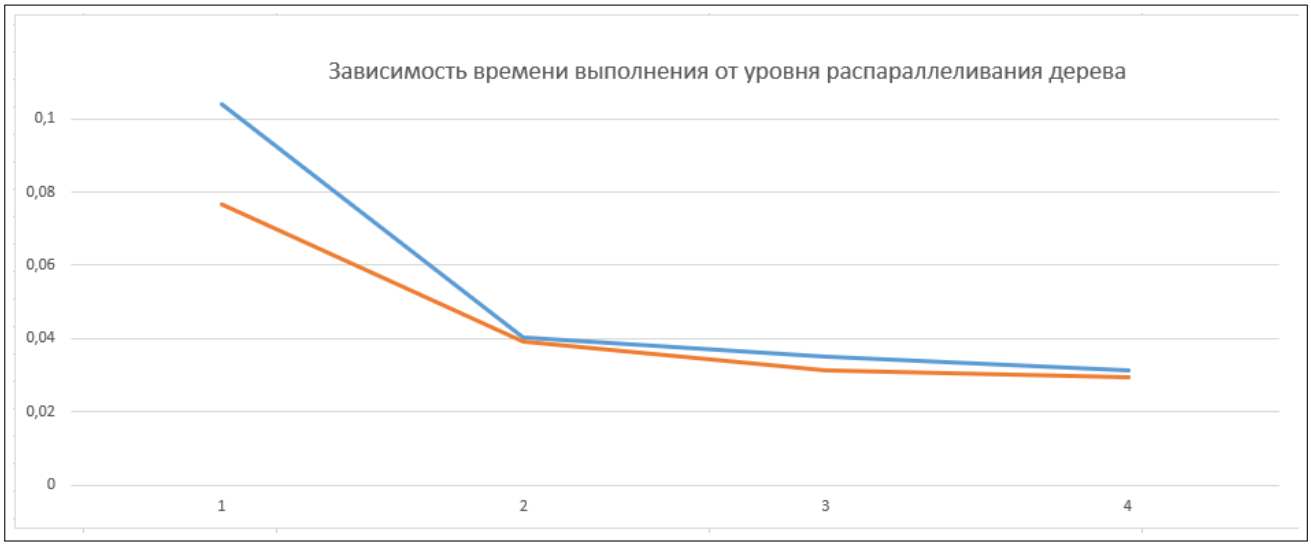

Рис. 4

Оранжевым цветом отмечен график Pthreads, а синим MPI.

5.1.3 Эксперимент 3

Уровень дерева: 4
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	MPI	Pthreads
0.057643	0.023482	0.016581
0.062972	0.020743	0.017967
0.057816	0.018492	0.015576

0.059537	0.024657	0.0189796
0.058706	0.021475	0.021683
0.057586	0.017537	0.020907
0.056716	0.016553	0.019520
0.056577	0.019492	0.015146
0.061485	0.024686	0.019487
0.057079	0.017457	0.020610

Таблица 4: Тестовая выборка для анализа

Характеристика	**Последовательный**	**MPI**	**Pthreads**
Среднее значение	0.0586117	0.0204574	0.0186456
Дисперсия	0.00000408395721	0.00000833532504	0.00000461749964
Доверительный интервал (P = 0.95)	(0.0572;0.0601)	(0.0184;0.0225)	(0.0171;0.0202)

Таблица 5: Вероятностные характеристики

Как видно из представленных характеристик, **pthreads** является лучшим решением. У него лучше средняя скорость вычислений и дисперсия по сравнению с MPI.

Вывод

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

Реализация на MPI 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 MPI это контроллирует сам фреймворк.

Эксперименты показали, что прирост проиводительности начался при наличии в деревее более 100 000 узлов. Наилучшие результаты были получены на 4 уровне распараллеливания, где удалось добиться прироста производительность в 69% для MPI и 76% для Pthreads. Возможно, данный показатель, 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

Листинг 4: Main.cpp

```
#include <stdio.h>
#include <ctime>
#include <fstream>
#include <omp.h>
#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <stdint.h>
#include <string>
#include <mpi.h>
#include "Tree.h"
#include "TreeUtils.h"


using namespace std;

double startTime, endTime;

const int level = 2;

void defaultSum(tnode* tree){
    startTime = omp_get_wtime();
    long sum = getSumOfAllChilds(tree);
    endTime = omp_get_wtime();
    printf("[Default] Sum: %llu\n", sum);
    printf("[Default] Time: %lf\n", endTime - startTime);
}

void pthreadSum(tnode* tree){

    int threads = 7;

    pthreadArg arg;
    arg.tree = tree;
    arg.threadCount = threads;
    startTime = omp_get_wtime();
    getSumOfAllChilds_Pthread((void *) &arg);
    endTime = omp_get_wtime();
    printf("[Pthread] Sum: %llu\n", arg.tree->sum
);
    printf("[Pthread] Time: %lf\n", endTime - startTime);
}


int main(int argc, char* argv[]){
  int rank, numprocs;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      tnode* tree = makeRandomTree();
      getSumOfAllChilds_MPI(tree, level, rank, numprocs);
  if (rank == 0) {
        defaultSum(tree);
      pthreadSum(tree);
  }
      MPI_Finalize();
      return 0;
}
```

Приложение 2

Листинг 5: Tree.h

```
#pragma once

#include <iostream>
#include <vector>

using namespace std;


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

tnode* addNode(long v, long level, tnode *tree);
```

```
19
20 tnode* makeNewTree(long v, long level, 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(long v, long level, tnode *tree)
10 {
11   if (tree == NULL)
12   {
13     tree = makeNewTree(v, level, tree);
14   }
15   else if (v < tree->value)
16     tree->left = addNode(v, level + 1, tree->left);
17   else if(v >= tree->value)
18     tree->right = addNode(v, level + 1, tree->right);
19   return(tree);
20 }
21
22
23 tnode* makeNewTree(long v, long level, tnode *tree)
24 {
25     tree = new tnode;
26     tree->value = v;
27     tree->sum = 0;
28     tree->level = level;
29     tree->right = NULL;
30     tree->left = NULL;
31     return(tree);
32 }
```

Приложение 4

Листинг 7: TreeUtils.h

```
1  #pragma once
2
3  #include "Tree.h"
4
5
6  #define MAX_VALUE 0xFFFFFFFF
7  #define GENERATE_COUNT 100000
8
9  #define SUCCESS 0
10 #define ERROR_CREATE_THREAD -1
11 #define ERROR_JOIN_THREAD   -2
12
13 struct pthreadArg {
14     struct tnode *tree;
15     int threadCount;
16 };
17
18
19 tnode* makeRandomTree();
20 long getLastNodes(int level, tnode* tree);
21 long getSumOfAllChilds(tnode* tree);
22 long getSumOfAllChilds_MPI(tnode* tree, int level, int rank, int numprocs);
23 void* getSumOfAllChilds_Pthread(void *args);
```

Приложение 5

Листинг 8: TreeUtils.cpp

```
1  #include <ctime>
2  #include <fstream>
3  #include <vector>
4  #include <mpi.h>
5
```

```
6  #include "TreeUtils.h"
7
8  tnode* makeRandomTree() {
9
10     tnode* tree = new tnode;
11     tree->value = 5;
12
13     srand(1);
14     for (int i = 0; i < GENERATE_COUNT; i++) {
15         long random = rand()%10;
16         addNode(random, 0, tree);
17     }
18
19     return tree;
20 }
21
22
23 void prepareSubTreesForMPI(tnode* tree, int level, std::vector<tnode*> *buf) {
24
25   if (tree != NULL){
26       if(level > tree->level)  {
27           prepareSubTreesForMPI(tree->left, level, buf);
28           prepareSubTreesForMPI(tree->right, level, buf);
29       }
30
31       if(level == tree->level) {
32           buf->push_back(tree);
33       }
34   } else {printf("lol");}
35 }
36
37 long getSumOfAllChilds(tnode* tree) {
38   if (tree != NULL){
39     long leftSum = 0;
40     long rightSum = 0;
41
42     if (tree->left != NULL) {
43       tree->left->sum = getSumOfAllChilds(tree->left);
44       leftSum = tree->left->sum + tree->left->value;
45     }
46
47     if (tree->right != NULL)
48     {
49       tree->right->sum = getSumOfAllChilds(tree->right);
50       rightSum = tree->right->sum + tree->right->value;
51     }
52
53     return leftSum + rightSum;
54   }
55   return 0;
56 }
57
58 long getSumOfAllChilds_MPI(tnode* tree, int level, int rank, int numprocs) {
59   MPI_Request request;
60   long Result = 0;
61   MPI_Status status;
62   int tag = 50;
63   std::vector<tnode*> buf;
64   double starttime, endtime = 0.0;
65   prepareSubTreesForMPI(tree, level, &buf);
66
67
68   if (rank == 0) {
69     starttime= MPI_Wtime();
70   }
71
72   if (rank != 0) {
73       tnode* subTree = buf.at(rank-1);
74       long drobSum = getSumOfAllChilds(subTree) + subTree->value;
75       MPI_Isend(&drobSum, 1, MPI_LONG, 0, tag, MPI_COMM_WORLD, &request);
76   }
77   if (rank == 0) {
78     for (int i = 1; i < numprocs; i++) {
79       long res = 0;
80       MPI_Recv(&res, 1, MPI_LONG, i, tag, MPI_COMM_WORLD, &status);
81       Result += res;
82     }
83     Result = Result + getLastNodes(level, tree) - tree->value;
84     endtime = MPI_Wtime();
85     printf("[MPI]: Time %lf\n", endtime - starttime);
86     printf("[MPI]: Sum %llu\n", Result);
87
88   }
89
90   return 0;
91 }
92
93 long getLastNodes(int level, tnode* tree) {
94   if (tree != NULL && level > tree->level) {
95       return getLastNodes(level, tree->left) + getLastNodes(level, tree->right) + tree->value;
96   }
```

```
  return 0;
}

void* getSumOfAllChilds_Pthread(void *args){
    pthreadArg *arg = (pthreadArg *)args;

    if (arg->tree != NULL){
        long leftSum = 0;
  long rightSum = 0;


  if (arg->threadCount <= 1){
            arg->tree->sum = getSumOfAllChilds(arg->tree);
            return 0;
  }
        int leftJoinStatus, rightJoinStatus;
        int leftCreateStatus, rightCreateStatus;


  pthread_t leftThread;
  pthreadArg leftArg;
  if (arg->tree->left != NULL){
            leftArg.tree = arg->tree->left;
            leftArg.threadCount = arg->threadCount/2;
            leftCreateStatus = pthread_create(&leftThread, NULL, getSumOfAllChilds_Pthread, (void*) &
                leftArg);
            if (leftCreateStatus != 0) {
                printf("[ERROR] Can't create thread. Status: %d\n", leftCreateStatus);
    exit(ERROR_CREATE_THREAD);
            }
  }


  pthread_t rightThread;
  pthreadArg rightArg;
  if (arg->tree->right != NULL){
            rightArg.tree = arg->tree->right;
            rightArg.threadCount =  arg->threadCount/2;
            rightCreateStatus = pthread_create(&rightThread, NULL, getSumOfAllChilds_Pthread, (void*) &
                rightArg);
            if (rightCreateStatus != 0) {
                printf("[ERROR] Can't create thread. Status: %d\n", rightCreateStatus);
    exit(ERROR_CREATE_THREAD);
            }
  }


  leftCreateStatus = pthread_join(leftThread, (void**)&leftJoinStatus);
  if (leftCreateStatus != SUCCESS) {
            printf("[ERROR] Can't join thread. Status: %d\n", leftCreateStatus);
            exit(ERROR_JOIN_THREAD);
  }
  if (arg->tree->left != NULL){
            arg->tree->left->sum = leftArg.tree->sum;
            leftSum = arg->tree->left->sum + arg->tree->left->value;
  }

  rightCreateStatus = pthread_join(rightThread, (void**)&rightJoinStatus);
  if (rightCreateStatus != SUCCESS) {
            printf("[ERROR] Can't join thread. Status: %d\n", rightCreateStatus);
            exit(ERROR_JOIN_THREAD);
  }
  if (arg->tree->right != NULL){
            arg->tree->right->sum = rightArg.tree->sum;
            rightSum = arg->tree->right->sum + arg->tree->right->value;
  }

  arg->tree->sum = leftSum + rightSum;
    }
    return 0;
}
```