**Санкт-Петербургский политехнический университет Петра Великого**

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Отчет по лабораторной работе**

Дисциплина: «Параллельные вычисления»

**Работу выполнил:** студент группа 53501/3

\_\_\_\_\_\_\_\_\_Киселев А.А.

**Работу принял:**

\_\_\_\_\_\_\_\_\_доцент Стручков И.В.

Санкт-Петербург,

2016

1. **Цель работы**

Приобрести навыки разработки небольших программ с использованием библиотек для параллельных вычислений.

1. **Задачи работы**

* Для алгоритма из полученного задания написать последовательную программу на языке C или С++, реализующую этот алгоритм.
* Для созданной последовательной программы необходимо написать 3-5 тестов, которые покрывают основные варианты функционирования программы
* Проанализировать полученный алгоритм, выделить части, которые могут быть распараллелены, разработать структуру параллельной программы. Определить количество используемых потоков, а также правила и используемые объекты синхронизации.
* Написать реализацию параллельной программы с использованием POSIX потоков.
* Написать тесты для реализации параллельной программы с использованием POSIX.
* Написать реализацию параллельной программы с использованием OpenMP.
* Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты.
* 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.

1. **Реализация параллельной программы**

**Текст исходной задачи** (вариант № 6):

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

**Анализ задачи.**

Исходная задача подразумевает обход дерева и вычисление суммы дочерних элементов для каждой вершины и суммирование данного значения ко всем вершинам-родителям для данной вершины.

В качестве исходного дерева было взято двоичное AVL – дерево для получения максимальной сбалансированности дерева. Балансировка дерева производится на основе уравнивания значения высоты для каждой вершины. , отчего не возникнет вершин, высота которых отличается от вершин других уровней больше, чем на 1. Высота идеально сбалансированного дерева равна log2(N)-1 , где N – число вершин в дереве.

В качестве метода обхода взят обход дерева в глубину на основе стека вершин. На каждой итерации в вершину стека будет помещаться и изыматься (из вершины стека) очередная вершина из дерева. Для данной вершины вычисляется сумма дочерних элементов. Сами дочерние элементы также помещаются в вершину стека: сначала левый элемент, затем правый. Данный метод обхода позволит избежать ошибок переполнений стека вычислительного устройства (что характерно для варианта с рекурсией).

После получения суммы она записывается в локальное хранилище значений (карта - map). Происходит пересчет всех значений вершин, для которых очередная вершина является предком: итеративно перебираются значения вершин-родителей из локального хранилища сумм.

Алгоритм заканчивается, когда будет достигнута последняя необработанная вершина (крайняя левая для нашего случая).

Порядок обхода дерева представлен на рисунке 1.

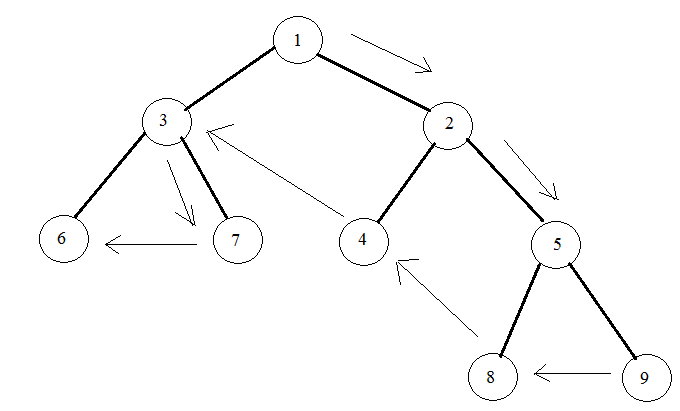


Рисунок 1. Порядок обхода дерева

**Алгоритм обхода** дерева для последовательной программы:

*//Обход дерева в ширину***void** wideTreeTraversal(nodeptr root, map<**int**,**int**> & summap)  
{  
 Stack \*q = createStack();  
 push(q, root);  
 **while** (q->size != 0) {  
 nodeptr tmp = (nodeptr) pop(q);  
 **int** s = 0;  
 **if** (tmp->left != NULL) {  
 push(q, tmp->left);  
 s += tmp->left->element;  
 }  
 **if** (tmp->right != NULL) {  
 push(q, tmp->right);  
 s += tmp->right->element;  
 }  
 summap.insert(pair<**int**,**int**>(tmp->element, s));  
 **while**(tmp != root)  
 {  
 tmp = tmp->parent;  
 **int** v = summap.at(tmp->element);  
 summap.erase(tmp->element);  
 summap.insert(pair<**int**,**int**>(tmp->element,v + s));  
 }  
  
 }  
 freeStack(&q);  
}

**Тестирование** последовательной программы.

Для тестирования написаны два теста:

- тест на выполнение программы: сравниваются полученный map и эталонный map.

- проверка метода вставки в map.

Полный текст программы с тестами представлен в приложении 1.

Анализ последовательного алгоритма будет представлен дальше.

1. **Реализация параллельной программы с использованием библиотеки Pthreads**

**Анализ последовательного алгоритма** для выделения параллельных частей.

Исходный алгоритм поддается распараллеливанию.

Во-первых, параллельно могут выполняться вычисления для двух не связанных вершин корня (левая и правая) и их потомки, находящиеся на одном уровне. Таким образом, каждый уровень дерева позволяет выделить 2^(уровень) потоков для параллельной обработки.

Во-вторых, каждый созданный поток будет содержать собственное локальное хранилище для хранения результатов вершин-предков. У поток появляется локальное хранилище.

В-третьих, в алгоритме можно выделить синхронизацию некоторых частей. Запись из локального map в общий map можно осуществлять для тех вершин, для которых сумма уже подсчитана. Такие вершины в ходе обработки дерева уже не меняют свои значения. Поэтому введение взаимного исключения для операции перемещения значения из локального хранилища в общее не занимает много времени по сравнению с другими вычислениями алгоритма.

Для реализации синхронизации в параллельной обработке введем дополнительный стек, в который будем помещать вершины-предки. Если для текущей вершины её родитель не равен вершине в дополнительном стеке, то вершина из стека удаляется, а её подсчитанная сумма записывается в общее хранилище.

Алгоритм реализации параллельного алгоритма представлен ниже:

*//Обход дерева в ширину***void**\* wideTreeTraversalWithThreads(**void**\* arg) {  
 Stack \*q = createStack(); //Стек для бхода  
 Stack \*parents = createStack();//Стек для хранения вершин предков  
 push(q,( (arguments\*)arg )->root);  
 push(parents,( (arguments\*)arg )->root);  
 **while** (q->size != 0) {  
 nodeptr tmp;  
 **if**( (tmp = (nodeptr) pop(q)) != NULL )  
 {  
 **if**( (tmp->parent != NULL ) && (tmp->parent != parents->data[parents->size-1]) && (pthread\_map[( (arguments\*)arg )->id].count(parents->data[parents->size-1]->element) != 0 ) ){  
 nodeptr eraseptr;  
 **do** {  
 pthread\_mutex\_lock(&mutex);  
 insertToMap(parents->data[parents->size - 1]->element,  
 pthread\_map[((arguments \*) arg)->id].at(parents->data[parents->size - 1]->element));  
 pthread\_mutex\_unlock(&mutex);   
 eraseptr = pop(parents);  
 }**while**(tmp->parent != eraseptr->parent);  
 }  
 }  
  
 **int** s = 0;  
 **if** (tmp->left != NULL) {  
 push(q, tmp->left);  
 s += tmp->left->element;  
 **if**( (tmp->left->left != NULL) || (tmp->left->right) != NULL)  
 push(parents,tmp->left);  
 }  
 **if** (tmp->right != NULL) {  
 push(q, tmp->right);  
 s += tmp->right->element;  
 **if**( (tmp->right->left != NULL) || (tmp->right->right) != NULL)  
 push(parents,tmp->right);  
 }  
 pthread\_map[( (arguments\*)arg )->id].insert(pair<**int**,**int**>(tmp->element, s));  
 **if**(s == 0) {  
 pthread\_mutex\_lock(&mutex);  
 insertToMap(tmp->element,s);  
 pthread\_mutex\_unlock(&mutex);   
 }  
 **while**(tmp != ( (arguments\*)arg )->root)  
 {  
 tmp = tmp->parent;  
 **int** v = pthread\_map[( (arguments\*)arg )->id].at(tmp->element);  
 pthread\_map[( (arguments\*)arg )->id].erase(tmp->element);  
 pthread\_map[( (arguments\*)arg )->id].insert(pair<**int**,**int**>(tmp->element,v + s));  
 }  
 }  
 **while**(parents->size != 0) {  
 pthread\_mutex\_lock(&mutex);  
 insertToMap(parents->data[parents->size - 1]->element,  
 pthread\_map[((arguments \*) arg)->id].at(parents->data[parents->size - 1]->element));  
 pthread\_mutex\_unlock(&mutex;  
 pop(parents);  
 }  
 freeStack(&q);  
 freeStack(&parents);  
 pthread\_mutex\_lock(&mutex);  
 countEndThreads++;  
 pthread\_mutex\_unlock(&mutex);  
}

**Тестирование параллельной программы на Pthreads.**

Для тестирования написано 3 теста для 2, 4, и 8 потоков выполнения программы. Тесты проверяют я правильность вычисления значений для потоков сравнивая с эталонным значением.

Полный тест параллельной программы представлен в приложении 2.

1. **Реализация параллельной программы с использованием библиотеки OpenMP**

Библиотека OpenMP обеспечивает более удобный способ параллельной обработки.

Для реализации напишем:

- автоматизированную выборку вершин для 2,4,8 потоков; Автоматизированная выборка вершин производится путем помещения в очередь вершин определенного уровня и их присвоение массиву корней для потоков.

*//Автоматизация распредления вершин***void** auto\_config(nodeptr bsroot, **int** level,**int** num, Stack\* nodestack)  
{  
 **if**(level != log2(num)) {  
 **if**(level == log2(num)-1) {  
 push(nodestack, bsroot->left);  
 push(nodestack, bsroot->right);  
 }  
 level++;  
 auto\_config(bsroot->left, level,num,nodestack);  
 auto\_config(bsroot->right, level,num,nodestack);  
 }

- алгоритм параллельной обработки останется прежним;

- для синхронизации будем использовать критическую секцию из библиотеки OpenMP.

Пример создания потоков:

#pragma omp parallel num\_threads(8) // Задаем число потоков  
{  
 #pragma omp for  
 **for**(**int** n = 0; n < num; ++n) // Создаем задачи для потоков  
 wideTreeTraversalWithThreads(n);  
}

Полный тест параллельной реализации с использованием OpenMP представлен в приложении 3.

1. **Проведение тестовых испытаний программы**

Для сравнения временных характеристик работы полученных алгоритмов проведем тестовые испытания каждого алгоритма для 2,4, и 8 потоков. Для каждого типа проведем 100 запусков и подсчитаем среднее время выполнения, запишем в таблицу. Исходное дерево содержит 100000 элементов.

Результаты тестовых испытаний представлены в таблице 1. В таблице записано математическое ожидание (среднее время) и среднеквадратичное отклонение.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Библиотека | Последовательное выполнение, мкс | Параллельное выполнение, кол. потоков | | |
| 2,мкс | 4,мкс | 8,мкс |
| Pthreads | 4031969 (~4c)  ±131306 | 2491038(~2,5c)  ±21748.2 | 1167117 (>1c)  ±115836 | 774187 (<1c)  ±25672.8 |
| OpenMP | 4015656(~4c)  ±73546.6 | 2454413(~2,5с)  ±49167.1 | 1143117(>1c)  ±29708.6 | 790951 (<1c)  ±129139 |

Таблица 1. Временные характеристики

Среднее время выполнения примерно одинаковое для каждой из библиотек, но OpenMP уступает по производительности. Испытания проводились на компьютере с 4 ядрами с 4 аппаратными потоками (Intel Core I5). Видим, что синхронизация также не испортила время выполнения, значит, введение синхронизаторов оказалось эффективным.

Заметим, что время для 8 потоков оказалось наилучшим. Эксперименты проводились на Windows 8.1. Подобный результат можно объяснить тем, что процессорное время выполнения операции в один такт позволяет выполнить действия последовательно для двух потоков, значит, за один процессорный такт для 4 ядер способны выполняться вычисления примерно для 8 потоков.

1. **Вывод**

В данной лабораторной работе была проведена разработка параллельных алгоритмов на основе последовательной программы для двух библиотек Pthreads и OpenMP. Реализация для двух библиотек существенна: библиотека OpenMP предоставляет более удобный способ организации параллельных программ за счет наличия готовых синтаксических конструкций параллельного выполнения в составе библиотеки.

Реализации параллельного и последовательного алгоритмов отличаются для исходной задачи в нашем случае. Параллельная реализация имеет дополнительные вычисления.

Введение параллелизма для последовательной программы возможно как на основе входных данных, так и на основе операций (команд). В полученной реализации содержится параллелизм на основе входных данных, когда параллельное выполнение осуществляется для разных и не связанных частей деревьев.

Синхронизация оказала положительное влияние, но потребовала внесения дополнительных вычислений. В общем случае синхронизацию нужно вводить таким образом, чтобы синхронизируемый блок содержал как можно меньше операций.

Параллельные программы оказались эффективнее последовательных. В общем случае параллельная реализация будет эффективнее, если правильно будет составлен алгоритм, и синхронизируемые блоки будут занимать меньше вычислительного времени по сравнению с общим потоком вычислений. На эффективность параллельной обработки влияют:

- число ядер(аппаратных потоков) вычислительного устройства;

- организация алгоритма вычислений;

- разделение локальной и общей памяти потоков;

- эффект от введенной синхронизации (если она необходима).

Приложение 1. Последовательная программа

*//  
// Created by Admin on 08.03.2016.  
//*#include **<iostream>**#include **<map>**#include **<chrono>**#include **"stack.h"  
  
using namespace** std;  
  
*//Тест обхода в глубину, результаты обработки теста должны соответствовать массиву значений , полученным при обходе в ширину***void** test1()   
{  
 nodeptr root = NULL;  
 nodeptr parent = NULL;  
 bstree tree;  
 **for**(**int** i = 0; i < 100000; i++) {  
 tree.insert(i, root,parent);  
 }  
 map<**int**,**int**> modelmap;  
 map<**int**,**int**> testmap;  
 wideTreeTraversal(root,modelmap);  
 depthTreeTraversal(root,testmap);  
 **if**(modelmap == testmap)  
 {  
 cout << **"test1 success"** << endl;  
 } **else** {  
 cout << **"test1 failed"** << endl;  
 }  
 modelmap.clear();  
 testmap.clear();  
}  
  
*//Добавление в карту***void** insertToMap(**int** &n, **int** &s, map<**int**,**int**> &summap)   
{   
 summap.insert(pair<**int**,**int**>(n,s));  
}  
  
*//Тест на проверку вставки в map***void** test2()   
{   
 map<**int**,**int**> testmap;  
 **int** key = 1;  
 **int** value = 3;  
 insertToMap(key,value,testmap);  
 **if**(testmap.find(1)->second == 3)  
 {  
 cout << **"test2 success"** << endl;  
 }  
 **else** {  
 cout << **"test2 failed"** << endl;  
 }  
 testmap.clear();  
}  
  
  
nodeptr mainroot = NULL;  
*//Обход дерева в глубину***void** depthTreeTraversal(nodeptr root ,map<**int**,**int**> &summap)  
{  
 **if**(mainroot == NULL)  
 mainroot = root;  
 **if** (root != NULL) {  
 nodeptr leftnode = root->left;  
 nodeptr rightnode = root->right;  
 *//cout << root->element;  
 //if(root->parent != NULL)  
 // cout << ": parent = " << root->parent->element;  
 //cout << " child: ";* **int** s = 0;  
 **if**(leftnode != NULL) {  
 *//cout << leftnode->element << " ";* s += leftnode->element;  
  
 }  
 **if**(rightnode != NULL) {  
 *//cout << rightnode->element << " ";* s += rightnode->element;  
 }  
 *//cout << root->height;  
 //cout << endl;* insertToMap(root->element,s,summap);  
 **while**(root != mainroot)  
 {  
 root = root->parent;  
 **int** v = summap.at(root->element);  
 summap.erase(root->element);  
 summap.insert(pair<**int**,**int**>(root->element,v + s));  
 }  
 depthTreeTraversal(leftnode,summap);  
 depthTreeTraversal(rightnode,summap);  
 }  
}  
*//Обход дерева в ширину***void** wideTreeTraversal(nodeptr root, map<**int**,**int**> & summap)  
{  
 Stack \*q = createStack();  
 push(q, root);  
 **while** (q->size != 0) {  
 nodeptr tmp = (nodeptr) pop(q);  
 **int** s = 0;  
 **if** (tmp->left != NULL) {  
 push(q, tmp->left);  
 s += tmp->left->element;  
 }  
 **if** (tmp->right != NULL) {  
 push(q, tmp->right);  
 s += tmp->right->element;  
 }  
 summap.insert(pair<**int**,**int**>(tmp->element, s));  
 **while**(tmp != root)  
 {  
 tmp = tmp->parent;  
 **int** v = summap.at(tmp->element);  
 summap.erase(tmp->element);  
 summap.insert(pair<**int**,**int**>(tmp->element,v + s));  
 }  
  
 }  
 freeStack(&q);  
}  
**int** main()  
{  
 test1();  
 test2();  
 map<**int**,**int**> summap;  
 nodeptr root = NULL;  
 nodeptr parent = NULL;  
 bstree tree;  
 **for**(**int** i = 0; i < 100000; i++) {  
 tree.insert(i, root,parent);  
 }  
 cout<<**"height: "**<<tree.bsheight(root)<<endl;  
 **auto** start\_time = std::chrono::high\_resolution\_clock::now();  
 wideTreeTraversal(root,summap);  
 **auto** end\_time = std::chrono::high\_resolution\_clock::now();  
 **auto** time = end\_time-start\_time;  
 printf(**"%d\n"**,summap.size());  
 cout << endl;  
 cout << std::chrono::duration\_cast<std::chrono::microseconds>(time).count() << endl;  
  
 cout << summap.size() << endl;  
  
 **return** 0;  
}

Приложение 2. Параллельная программа с использованием Pthreads

*//  
// Created by Admin on 01.03.2016.  
//*#include **<pthread.h>**#include **<stdio.h>**#include **<iostream>**#include **<chrono>**#include **<windows.h>**#include **<sys/time.h>**#include **<unistd.h>**#include **<map>**#include **"avl.h"**#include **"stack.h"  
using namespace** std;  
  
pthread\_mutex\_t mutex;  
CRITICAL\_SECTION CriticalSection;  
map<**int**,**int**>\* summap;  
map<**int**,**int**>\* pthread\_map = **new** map<**int**,**int**>[8];  
  
**struct** arguments {  
 **int** id;  
 nodeptr root;  
};  
  
**int** countWorkThreads;*//Число работающих потоков***int** countEndThreads;*//Число законченных потоков***void**\* wideTreeTraversalWithThreads(**void**\* arg);  
  
*//Тесты для параллельной обработки***void** test4threads(nodeptr bsroot)  
{  
 pthread\_t thread[8];  
 **int** status[8];  
 **int** status\_addr[8];  
 **int** threadid[8];  
 countEndThreads = 0;  
 countWorkThreads = 0;  
 **for**(**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
  
 **for** (**int** i = 0; i < 4; i++) {  
 **if** (i == 0) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 1) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 2) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 3) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 countWorkThreads++;  
 pthread\_detach(thread[i]);  
 }  
 **while**(countEndThreads < countWorkThreads)  
 usleep(1);  
  
 map<**int**,**int**> testmap;  
 **int** rootsum = bsroot->left->element + bsroot->right->element  
 + bsroot->left->left->element  
 + bsroot->left->right->element  
 + bsroot->right->left->element  
 + bsroot->right->right->element  
 + pthread\_map[0].at(bsroot->left->left->element)  
 + pthread\_map[1].at(bsroot->left->right->element)  
 + pthread\_map[2].at(bsroot->right->left->element)  
 + pthread\_map[3].at(bsroot->right->right->element);  
 testmap.insert(pthread\_map[0].begin(),pthread\_map[0].end());  
 testmap.insert(pthread\_map[1].begin(),pthread\_map[1].end());  
 testmap.insert(pthread\_map[2].begin(),pthread\_map[2].end());  
 testmap.insert(pthread\_map[3].begin(),pthread\_map[3].end());  
 testmap.insert(pair<**int**,**int**> (bsroot->element,rootsum));  
 testmap.insert(pair<**int**,**int**> (bsroot->left->element,bsroot->left->left->element + bsroot->left->right->element + pthread\_map[0].at(bsroot->left->left->element)  
 + pthread\_map[1].at(bsroot->left->right->element)));  
 testmap.insert(pair<**int**,**int**> (bsroot->right->element,bsroot->right->left->element + bsroot->right->right->element + pthread\_map[2].at(bsroot->right->left->element)  
 + pthread\_map[3].at(bsroot->right->right->element)));  
  
 **if**(testmap == \*summap)  
 cout << **"test 4 success"** << endl;  
 **else** cout << **"test 4 failed"** << endl;  
}  
  
**void** test8threads(nodeptr bsroot)  
{  
 pthread\_t thread[8];  
 **int** status[8];  
 **int** status\_addr[8];  
 **int** threadid[8];  
 countEndThreads = 0;  
 countWorkThreads = 0;  
 **for**(**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
 **for** (**int** i = 0; i < 8; i++) {  
 **if** (i == 0) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->left->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 1) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->left->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 2) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->right->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 3) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->right->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 4) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->left->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 5) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->left->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 6) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->right->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 7) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->right->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 countWorkThreads++;  
 pthread\_detach(thread[i]);  
 }  
 **while**(countEndThreads < countWorkThreads)  
 usleep(1);  
  
 map<**int**,**int**> testmap;  
 testmap.insert(pthread\_map[0].begin(),pthread\_map[0].end());  
 testmap.insert(pthread\_map[1].begin(),pthread\_map[1].end());  
 testmap.insert(pthread\_map[2].begin(),pthread\_map[2].end());  
 testmap.insert(pthread\_map[3].begin(),pthread\_map[3].end());  
 testmap.insert(pthread\_map[4].begin(),pthread\_map[4].end());  
 testmap.insert(pthread\_map[5].begin(),pthread\_map[5].end());  
 testmap.insert(pthread\_map[6].begin(),pthread\_map[6].end());  
 testmap.insert(pthread\_map[7].begin(),pthread\_map[7].end());  
 **int** rootsum = bsroot->left->element + bsroot->right->element  
 + bsroot->left->left->element + bsroot->left->right->element  
 + bsroot->right->left->element + bsroot->right->right->element  
 + bsroot->left->left->left->element  
 + bsroot->left->left->right->element  
 + bsroot->left->right->left->element  
 + bsroot->left->right->right->element  
 + bsroot->right->left->left->element  
 + bsroot->right->left->right->element  
 + bsroot->right->right->left->element  
 + bsroot->right->right->right->element  
 + pthread\_map[0].at(bsroot->left->left->left->element)  
 + pthread\_map[1].at(bsroot->left->left->right->element)  
 + pthread\_map[2].at(bsroot->left->right->left->element)  
 + pthread\_map[3].at(bsroot->left->right->right->element)  
 + pthread\_map[4].at(bsroot->right->left->left->element)  
 + pthread\_map[5].at(bsroot->right->left->right->element)  
 + pthread\_map[6].at(bsroot->right->right->left->element)  
 + pthread\_map[7].at(bsroot->right->right->right->element);  
 testmap.insert(pair<**int**,**int**> (bsroot->element,rootsum));  
 testmap.insert(pair<**int**,**int**> (bsroot->left->element,bsroot->left->left->element  
 + bsroot->left->right->element  
 + bsroot->left->left->left->element  
 + bsroot->left->left->right->element  
 + pthread\_map[0].at(bsroot->left->left->left->element)  
 + pthread\_map[1].at(bsroot->left->left->right->element)  
 + bsroot->left->right->left->element  
 + bsroot->left->right->right->element  
 + pthread\_map[2].at(bsroot->left->right->left->element)  
 + pthread\_map[3].at(bsroot->left->right->right->element)));  
 testmap.insert(pair<**int**,**int**> (bsroot->right->element,bsroot->right->left->element  
 + bsroot->right->right->element  
 + bsroot->right->left->left->element  
 + bsroot->right->left->right->element  
 + pthread\_map[4].at(bsroot->right->left->left->element)  
 + pthread\_map[5].at(bsroot->right->left->right->element)  
 + bsroot->right->right->left->element  
 + bsroot->right->right->right->element  
 + pthread\_map[6].at(bsroot->right->right->left->element)  
 + pthread\_map[7].at(bsroot->right->right->right->element)));  
 testmap.insert(pair<**int**,**int**> (bsroot->left->left->element,  
 bsroot->left->left->left->element  
 + bsroot->left->left->right->element  
 + pthread\_map[0].at(bsroot->left->left->left->element)  
 + pthread\_map[1].at(bsroot->left->left->right->element)));  
 testmap.insert(pair<**int**,**int**> (bsroot->left->right->element,  
 bsroot->left->right->left->element  
 + bsroot->left->right->right->element  
 + pthread\_map[2].at(bsroot->left->right->left->element)  
 + pthread\_map[3].at(bsroot->left->right->right->element)));  
 testmap.insert(pair<**int**,**int**> (bsroot->right->left->element,  
 bsroot->right->left->left->element  
 + bsroot->right->left->right->element  
 + pthread\_map[4].at(bsroot->right->left->left->element)  
 + pthread\_map[5].at(bsroot->right->left->right->element)));  
 testmap.insert(pair<**int**,**int**> (bsroot->right->right->element,  
 bsroot->right->right->left->element  
 + bsroot->right->right->right->element  
 + pthread\_map[6].at(bsroot->right->right->left->element)  
 + pthread\_map[7].at(bsroot->right->right->right->element)));  
 **if**(testmap == \*summap)  
 cout << **"test 8 success"** << endl;  
 **else** cout << **"test 8 failed"** << endl;  
}  
  
**void** test2threads(nodeptr bsroot)  
{  
 pthread\_t thread[8];  
 **int** status[8];  
 **int** status\_addr[8];  
 **int** threadid[8];  
 countEndThreads = 0;  
 countWorkThreads = 0;  
 **for**(**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
  
 **for** (**int** i = 0; i < 2; i++) {  
 **if** (i == 0) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 1) {  
 arguments\* arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 countWorkThreads++;  
 pthread\_detach(thread[i]);  
 }  
 **while**(countEndThreads < countWorkThreads)  
 usleep(1);  
 map<**int**,**int**> testmap;  
 testmap.insert(pthread\_map[0].begin(),pthread\_map[0].end());  
 testmap.insert(pthread\_map[1].begin(),pthread\_map[1].end());  
 **int** rootsum = bsroot->left->element + bsroot->right->element + pthread\_map[0].at(bsroot->left->element)  
 + pthread\_map[1].at(bsroot->right->element);  
 testmap.insert(pair<**int**,**int**> (bsroot->element,rootsum));  
 **if**(testmap == \*summap)  
 cout << **"test 2 success"** << endl;  
 **else** cout << **"test 2 failed"** << endl;  
}  
  
  
  
**void** insertToMap(**int** &n, **int** &s)  
{  
  
 summap->insert(pair<**int**,**int**>(n,s));  
   
}  
*//Обход дерева в ширину***void**\* wideTreeTraversalWithThreads(**void**\* arg) {  
 Stack \*q = createStack();  
 Stack \*parents = createStack();  
 push(q,( (arguments\*)arg )->root);  
 push(parents,( (arguments\*)arg )->root);  
 **while** (q->size != 0) {  
 nodeptr tmp;  
 **if**( (tmp = (nodeptr) pop(q)) != NULL )  
 {  
 **if**( (tmp->parent != NULL ) && (tmp->parent != parents->data[parents->size-1]) && (pthread\_map[( (arguments\*)arg )->id].count(parents->data[parents->size-1]->element) != 0 ) ){  
 nodeptr eraseptr;  
 **do** {  
 pthread\_mutex\_lock(&mutex);  
 insertToMap(parents->data[parents->size - 1]->element,  
 pthread\_map[((arguments \*) arg)->id].at(parents->data[parents->size - 1]->element));  
 pthread\_mutex\_unlock(&mutex);  
 */\*EnterCriticalSection(&CriticalSection);  
 insertToMap(parents->data[parents->size - 1]->element,  
 pthread\_map[((arguments \*) arg)->id].at(parents->data[parents->size - 1]->element));  
 LeaveCriticalSection(&CriticalSection);\*/* eraseptr = pop(parents);  
 }**while**(tmp->parent != eraseptr->parent);  
 }  
 }  
  
 **int** s = 0;  
 **if** (tmp->left != NULL) {  
 push(q, tmp->left);  
 s += tmp->left->element;  
 **if**( (tmp->left->left != NULL) || (tmp->left->right) != NULL)  
 push(parents,tmp->left);  
 }  
 **if** (tmp->right != NULL) {  
 push(q, tmp->right);  
 s += tmp->right->element;  
 **if**( (tmp->right->left != NULL) || (tmp->right->right) != NULL)  
 push(parents,tmp->right);  
 }  
 pthread\_map[( (arguments\*)arg )->id].insert(pair<**int**,**int**>(tmp->element, s));  
 **if**(s == 0) {  
 pthread\_mutex\_lock(&mutex);  
 insertToMap(tmp->element,s);  
 pthread\_mutex\_unlock(&mutex);  
 */\* EnterCriticalSection(&CriticalSection);  
 insertToMap(tmp->element,s);  
 LeaveCriticalSection(&CriticalSection);\*/* }  
 **while**(tmp != ( (arguments\*)arg )->root)  
 {  
 tmp = tmp->parent;  
 **int** v = pthread\_map[( (arguments\*)arg )->id].at(tmp->element);  
 pthread\_map[( (arguments\*)arg )->id].erase(tmp->element);  
 pthread\_map[( (arguments\*)arg )->id].insert(pair<**int**,**int**>(tmp->element,v + s));  
 }  
 }  
 **while**(parents->size != 0) {  
 pthread\_mutex\_lock(&mutex);  
 insertToMap(parents->data[parents->size - 1]->element,  
 pthread\_map[((arguments \*) arg)->id].at(parents->data[parents->size - 1]->element));  
 pthread\_mutex\_unlock(&mutex);  
 */\*EnterCriticalSection(&CriticalSection);  
 insertToMap(parents->data[parents->size - 1]->element,  
 pthread\_map[((arguments \*) arg)->id].at(parents->data[parents->size - 1]->element));  
 LeaveCriticalSection(&CriticalSection);\*/* pop(parents);  
 }  
 freeStack(&q);  
 freeStack(&parents);  
 pthread\_mutex\_lock(&mutex);  
 countEndThreads++;  
 pthread\_mutex\_unlock(&mutex);  
}  
  
*//Обход дерева в глубину***void** depthTreeTraversalWithThreads(nodeptr root)  
{  
 **if** (root != NULL) {  
 nodeptr leftnode = root->left;  
 nodeptr rightnode = root->right;  
 cout << root->element;  
 **if**(root->parent != NULL)  
 cout << **": parent = "** << root->parent->element;  
 cout << **" child: "**;  
 **int** s = 0;  
 **if**(leftnode != NULL) {  
 cout << leftnode->element << **" "**;  
 }  
 **if**(rightnode != NULL) {  
 cout << rightnode->element << **" "**;  
 }  
 cout << endl;  
 depthTreeTraversalWithThreads(leftnode);  
 depthTreeTraversalWithThreads(rightnode);  
 }  
}  
**int** main() {  
  
 **int** C = 100;  
 pthread\_t thread[8];  
 **int** status[8];  
 **int** status\_addr[8];  
 **int** threadid[8];  
 **if** (pthread\_mutex\_init(&mutex,NULL) != 0 )  
 {  
 printf(**"Mutex fail!\n"**);  
 **return** 1;  
 }  
 InitializeCriticalSection(&CriticalSection);  
 summap = **new** map<**int**,**int**>();  
 countWorkThreads = 0;  
 countEndThreads = 0;  
  
 nodeptr bsroot = NULL;  
 nodeptr bsparent = NULL;  
 bstree bstree;  
 **for**(**int** i = 0; i < 100000; i++) {  
 bstree.insert(i, bsroot,bsparent);  
 }  
 cout << **"height = "** << bstree.bsheight(bsroot) << endl;  
  
 arguments\* arg = **new** arguments;  
 arg->id = 0;  
 arg->root = bsroot;  
  
 *//Время вычисления для всего дерева* **int** sum1 = 0;  
 **for**(**int** k = 0; k < C; k++) {  
 **auto** start\_time = std::chrono::high\_resolution\_clock::now();  
 wideTreeTraversalWithThreads(arg);  
 **auto** end\_time = std::chrono::high\_resolution\_clock::now();  
 **auto** time = end\_time - start\_time;  
 sum1+=std::chrono::duration\_cast<std::chrono::microseconds>(time).count();  
 **for** (**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
 }  
 cout << endl;  
 cout << endl << **"Sequence time for all tree: "** << sum1/C << endl;  
 cout << summap->size() << endl;  
 cout << endl;  
 test4threads(bsroot);  
 test8threads(bsroot);  
 test2threads(bsroot);  
 summap->clear();  
 *//Время для параллельной обработки 4 потоков* countEndThreads = 0;  
 countWorkThreads = 0;  
 **for**(**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
  
 **int** sum4 = 0;  
 **for**(**int** k = 0; k < C; k++) {  
 **auto** start\_time = std::chrono::high\_resolution\_clock::now();  
 **for** (**int** i = 0; i < 4; i++) {  
 **if** (i == 0) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 1) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 2) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 3) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 countWorkThreads++;  
 pthread\_detach(thread[i]);  
 }  
 **while** (countEndThreads < countWorkThreads)  
 usleep(1);  
  
 **int** rootsum = bsroot->left->element + bsroot->right->element  
 + bsroot->left->left->element  
 + bsroot->left->right->element  
 + bsroot->right->left->element  
 + bsroot->right->right->element  
 + pthread\_map[0].at(bsroot->left->left->element)  
 + pthread\_map[1].at(bsroot->left->right->element)  
 + pthread\_map[2].at(bsroot->right->left->element)  
 + pthread\_map[3].at(bsroot->right->right->element);  
 summap->insert(pair<**int**, **int**>(bsroot->element, rootsum));  
 summap->insert(pair<**int**, **int**>(bsroot->left->element,  
 bsroot->left->left->element + bsroot->left->right->element +  
 pthread\_map[0].at(bsroot->left->left->element)  
 + pthread\_map[1].at(bsroot->left->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->right->element,  
 bsroot->right->left->element + bsroot->right->right->element +  
 pthread\_map[2].at(bsroot->right->left->element)  
 + pthread\_map[3].at(bsroot->right->right->element)));  
  
  
 **auto** end\_time = std::chrono::high\_resolution\_clock::now();  
  
 **auto** time = end\_time - start\_time;  
 sum4 += std::chrono::duration\_cast<std::chrono::microseconds>(time).count();  
 **for**(**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
 }  
 cout << endl <<**"Parallel time - 4 threads: "** << sum4/C << endl;  
  
 cout << summap->size() << endl;  
 *//Время параллельного выисления для 8 потоков* countEndThreads = 0;  
 countWorkThreads = 0;  
  
 summap->clear();  
 **for**(**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
  
 **int** sum8 = 0;  
 **for**(**int** k = 0; k < C; k++) {  
 **auto** start\_time = std::chrono::high\_resolution\_clock::now();  
 **for** (**int** i = 0; i < 8; i++) {  
 **if** (i == 0) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->left->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 1) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->left->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 2) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->right->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 3) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left->right->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 4) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->left->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 5) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->left->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 6) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->right->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 7) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right->right->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 countWorkThreads++;  
 pthread\_detach(thread[i]);  
 }  
 **while** (countEndThreads < countWorkThreads)  
 usleep(1);  
  
 **int** rootsum = bsroot->left->element + bsroot->right->element  
 + bsroot->left->left->element + bsroot->left->right->element  
 + bsroot->right->left->element + bsroot->right->right->element  
 + bsroot->left->left->left->element  
 + bsroot->left->left->right->element  
 + bsroot->left->right->left->element  
 + bsroot->left->right->right->element  
 + bsroot->right->left->left->element  
 + bsroot->right->left->right->element  
 + bsroot->right->right->left->element  
 + bsroot->right->right->right->element  
 + pthread\_map[0].at(bsroot->left->left->left->element)  
 + pthread\_map[1].at(bsroot->left->left->right->element)  
 + pthread\_map[2].at(bsroot->left->right->left->element)  
 + pthread\_map[3].at(bsroot->left->right->right->element)  
 + pthread\_map[4].at(bsroot->right->left->left->element)  
 + pthread\_map[5].at(bsroot->right->left->right->element)  
 + pthread\_map[6].at(bsroot->right->right->left->element)  
 + pthread\_map[7].at(bsroot->right->right->right->element);  
 summap->insert(pair<**int**, **int**>(bsroot->element, rootsum));  
 summap->insert(pair<**int**, **int**>(bsroot->left->element, bsroot->left->left->element  
 + bsroot->left->right->element  
 + bsroot->left->left->left->element  
 + bsroot->left->left->right->element  
 + pthread\_map[0].at(bsroot->left->left->left->element)  
 + pthread\_map[1].at(bsroot->left->left->right->element)  
 + bsroot->left->right->left->element  
 + bsroot->left->right->right->element  
 + pthread\_map[2].at(bsroot->left->right->left->element)  
 + pthread\_map[3].at(bsroot->left->right->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->right->element, bsroot->right->left->element  
 + bsroot->right->right->element  
 + bsroot->right->left->left->element  
 + bsroot->right->left->right->element  
 + pthread\_map[4].at(bsroot->right->left->left->element)  
 + pthread\_map[5].at(bsroot->right->left->right->element)  
 + bsroot->right->right->left->element  
 + bsroot->right->right->right->element  
 + pthread\_map[6].at(bsroot->right->right->left->element)  
 +  
 pthread\_map[7].at(bsroot->right->right->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->left->left->element,  
 bsroot->left->left->left->element  
 + bsroot->left->left->right->element  
 + pthread\_map[0].at(bsroot->left->left->left->element)  
 + pthread\_map[1].at(bsroot->left->left->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->left->right->element,  
 bsroot->left->right->left->element  
 + bsroot->left->right->right->element  
 + pthread\_map[2].at(bsroot->left->right->left->element)  
 + pthread\_map[3].at(bsroot->left->right->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->right->left->element,  
 bsroot->right->left->left->element  
 + bsroot->right->left->right->element  
 + pthread\_map[4].at(bsroot->right->left->left->element)  
 + pthread\_map[5].at(bsroot->right->left->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->right->right->element,  
 bsroot->right->right->left->element  
 + bsroot->right->right->right->element  
 + pthread\_map[6].at(bsroot->right->right->left->element)  
 + pthread\_map[7].at(bsroot->right->right->right->element)));  
  
 **auto** end\_time = std::chrono::high\_resolution\_clock::now();  
  
 **auto** time = end\_time - start\_time;  
 sum8 += std::chrono::duration\_cast<std::chrono::microseconds>(time).count();  
 **for**(**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
 }  
 cout << endl <<**"Parallel time - 8 threads: "** << sum8/C << endl;  
  
 cout << summap->size() << endl;  
  
 *//Время параллельного выисления для 2 потоков* countEndThreads = 0;  
 countWorkThreads = 0;  
 **for**(**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
  
 summap->clear();  
 **int** sum2 = 0;  
 **for**(**int** k = 0; k < C; k++) {  
 **auto** start\_time = std::chrono::high\_resolution\_clock::now();  
 **for** (**int** i = 0; i < 2; i++) {  
 **if** (i == 0) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->left;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 **else if** (i == 1) {  
 arguments \*arg = **new** arguments;  
 arg->id = i;  
 arg->root = bsroot->right;  
 **if** ((status[i] = pthread\_create(&thread[i], NULL, &wideTreeTraversalWithThreads, (**void** \*) arg)) != 0) {  
 printf(**"Can't create thread!\n"**);  
 }  
 }  
 countWorkThreads++;  
 pthread\_detach(thread[i]);  
 }  
 **while** (countEndThreads < countWorkThreads)  
 usleep(1);  
 **int** rootsum = bsroot->left->element + bsroot->right->element + pthread\_map[0].at(bsroot->left->element)  
 + pthread\_map[1].at(bsroot->right->element);  
 summap->insert(pair<**int**, **int**>(bsroot->element, rootsum));  
 **auto** end\_time = std::chrono::high\_resolution\_clock::now();  
  
 **auto** time = end\_time - start\_time;  
 sum2 = std::chrono::duration\_cast<std::chrono::microseconds>(time).count();  
 **for**(**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
 }  
 cout << endl <<**"Parallel time - 2 threads: "** << sum2/C << endl;  
  
 cout << summap->size() << endl;  
 *//Освобождаем ресурсы* pthread\_mutex\_destroy(&mutex);  
 **delete**(summap);  
  
  
 **return** 0;  
}

Приложение 3. Параллельная программа с использованием OpenMP

*//  
// Created by Admin on 21.03.2016.  
//*#include **<iostream>**#include **<map>**#include **<chrono>**#include **<omp.h>**#include **"stack.h"**#include **<math.h>  
  
using namespace** std;  
  
map<**int**,**int**>\* pthread\_map = **new** map<**int**,**int**>[8];  
nodeptr\* nodes;  
map<**int**,**int**> \* summap;  
**void** insertToMap(**int** &n, **int** &s)  
{  
 summap->insert(pair<**int**,**int**>(n,s));  
}  
*//Обход дерева в ширину, TASK***void** wideTreeTraversalWithThreads(**int** &id) {  
 Stack \*q = createStack();  
 Stack \*parents = createStack();  
 push(q, nodes[id]);  
 push(parents, nodes[id]);  
 **while** (q->size != 0) {  
 nodeptr tmp;  
 **if**( (tmp = (nodeptr) pop(q)) != **NULL** )  
 {  
 **if**( (tmp->parent != **NULL** ) && (tmp->parent != parents->data[parents->size-1]) && (pthread\_map[id].count(parents->data[parents->size-1]->element) != 0 ) ){  
 nodeptr eraseptr;  
 **do** {  
 #pragma omp critical  
 {  
 insertToMap(parents->data[parents->size - 1]->element, pthread\_map[id].at(parents->data[parents->size - 1]->element));  
 }  
 eraseptr = pop(parents);  
 }**while**(tmp->parent != eraseptr->parent);  
 }  
 }  
  
 **int** s = 0;  
 **if** (tmp->left != **NULL**) {  
 push(q, tmp->left);  
 s += tmp->left->element;  
 **if**( (tmp->left->left != **NULL**) || (tmp->left->right) != **NULL**)  
 push(parents,tmp->left);  
 }  
 **if** (tmp->right != **NULL**) {  
 push(q, tmp->right);  
 s += tmp->right->element;  
 **if**( (tmp->right->left != **NULL**) || (tmp->right->right) != **NULL**)  
 push(parents,tmp->right);  
 }  
 pthread\_map[id].insert(pair<**int**,**int**>(tmp->element, s));  
 **if**(s == 0) {  
 #pragma omp critical  
 {  
 insertToMap(tmp->element,s);  
 }  
 }  
 **while**(tmp != nodes[id])  
 {  
 tmp = tmp->parent;  
 **int** v = pthread\_map[id].at(tmp->element);  
 pthread\_map[id].erase(tmp->element);  
 pthread\_map[id].insert(pair<**int**,**int**>(tmp->element,v + s));  
 }  
 }  
 **while**(parents->size != 0) {  
 #pragma omp critical  
 {  
 insertToMap(parents->data[parents->size - 1]->element,pthread\_map[id].at(parents->data[parents->size - 1]->element));  
 }  
 pop(parents);  
 }  
 freeStack(&q);  
 freeStack(&parents);  
}  
  
*//Автоматизация распредления вершин***void** auto\_config(nodeptr bsroot, **int** level,**int** num, Stack\* nodestack)  
{  
 **if**(level != log2(num)) {  
 **if**(level == log2(num)-1) {  
 push(nodestack, bsroot->left);  
 push(nodestack, bsroot->right);  
 }  
 level++;  
 auto\_config(bsroot->left, level,num,nodestack);  
 auto\_config(bsroot->right, level,num,nodestack);  
 }  
}  
  
**int** main(**void**) {  
 **int** C = 100;  
 nodeptr bsroot = **NULL**;  
 nodeptr bsparent = **NULL**;  
 bstree bstree;  
 summap = **new** map<**int**, **int**>();  
 **for** (**int** i = 0; i < 100000; i++) {  
 bstree.insert(i, bsroot, bsparent);  
 }  
 cout << **"height = "** << bstree.bsheight(bsroot) << endl;  
 **int** num = 8;  
 nodes = **new** nodeptr[num];  
 Stack \*nodestack = createStack();*//стек для вершин* auto\_config(bsroot, 0, num, nodestack);  
 **for** (**int** i = 0; i < num; i++) {  
 nodes[i] = (nodeptr) pop(nodestack);  
 }  
 freeStack(&nodestack);  
  
 **int** sum8 = 0;  
 **for** (**int** k = 0; k < C; k++) {  
 **auto** start\_time = std::chrono::high\_resolution\_clock::now();  
#pragma omp parallel num\_threads(8)  
 {  
#pragma omp for  
 **for** (**int** n = 0; n < num; ++n)  
 wideTreeTraversalWithThreads(n);  
 }  
  
 **int** rootsum = bsroot->left->element + bsroot->right->element  
 + bsroot->left->left->element + bsroot->left->right->element  
 + bsroot->right->left->element + bsroot->right->right->element  
 + bsroot->left->left->left->element  
 + bsroot->left->left->right->element  
 + bsroot->left->right->left->element  
 + bsroot->left->right->right->element  
 + bsroot->right->left->left->element  
 + bsroot->right->left->right->element  
 + bsroot->right->right->left->element  
 + bsroot->right->right->right->element  
 + pthread\_map[7].at(bsroot->left->left->left->element)  
 + pthread\_map[6].at(bsroot->left->left->right->element)  
 + pthread\_map[5].at(bsroot->left->right->left->element)  
 + pthread\_map[4].at(bsroot->left->right->right->element)  
 + pthread\_map[3].at(bsroot->right->left->left->element)  
 + pthread\_map[2].at(bsroot->right->left->right->element)  
 + pthread\_map[1].at(bsroot->right->right->left->element)  
 + pthread\_map[0].at(bsroot->right->right->right->element);  
 summap->insert(pair<**int**, **int**>(bsroot->element, rootsum));  
 summap->insert(pair<**int**, **int**>(bsroot->left->element, bsroot->left->left->element  
 + bsroot->left->right->element  
 + bsroot->left->left->left->element  
 + bsroot->left->left->right->element  
 + pthread\_map[7].at(bsroot->left->left->left->element)  
 + pthread\_map[6].at(bsroot->left->left->right->element)  
 + bsroot->left->right->left->element  
 + bsroot->left->right->right->element  
 + pthread\_map[5].at(bsroot->left->right->left->element)  
 + pthread\_map[4].at(bsroot->left->right->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->right->element, bsroot->right->left->element  
 + bsroot->right->right->element  
 + bsroot->right->left->left->element  
 + bsroot->right->left->right->element  
 + pthread\_map[3].at(bsroot->right->left->left->element)  
 + pthread\_map[2].at(bsroot->right->left->right->element)  
 + bsroot->right->right->left->element  
 + bsroot->right->right->right->element  
 + pthread\_map[1].at(bsroot->right->right->left->element)  
 +  
 pthread\_map[0].at(bsroot->right->right->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->left->left->element,  
 bsroot->left->left->left->element  
 + bsroot->left->left->right->element  
 + pthread\_map[7].at(bsroot->left->left->left->element)  
 + pthread\_map[6].at(bsroot->left->left->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->left->right->element,  
 bsroot->left->right->left->element  
 + bsroot->left->right->right->element  
 + pthread\_map[5].at(bsroot->left->right->left->element)  
 + pthread\_map[4].at(bsroot->left->right->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->right->left->element,  
 bsroot->right->left->left->element  
 + bsroot->right->left->right->element  
 + pthread\_map[3].at(bsroot->right->left->left->element)  
 + pthread\_map[2].at(bsroot->right->left->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->right->right->element,  
 bsroot->right->right->left->element  
 + bsroot->right->right->right->element  
 + pthread\_map[1].at(bsroot->right->right->left->element)  
 + pthread\_map[0].at(bsroot->right->right->right->element)));  
  
  
 **auto** end\_time = std::chrono::high\_resolution\_clock::now();  
 **auto** time = end\_time - start\_time;  
 sum8 += std::chrono::duration\_cast<std::chrono::microseconds>(time).count();  
 **for** (**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
 }  
 cout << endl;  
 cout << endl << **"Parallel time for 8 threads: "** << sum8 / C << endl;  
 cout << summap->size() << endl;  
 summap->clear();  
 **for** (**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
  
 num = 4;  
 nodes = **new** nodeptr[num];  
 nodestack = createStack();*//стек для вершин* auto\_config(bsroot, 0, num, nodestack);  
 **for** (**int** i = 0; i < num; i++) {  
 nodes[i] = (nodeptr) pop(nodestack);  
 }  
 freeStack(&nodestack);  
  
  
 **int** sum4 = 0;  
 **for** (**int** k = 0; k < C; k++) {  
 **auto** start\_time = std::chrono::high\_resolution\_clock::now();  
#pragma omp parallel num\_threads(4)  
 {  
#pragma omp for  
 **for** (**int** n = 0; n < num; ++n)  
 wideTreeTraversalWithThreads(n);  
 }  
  
 **int** rootsum = bsroot->left->element + bsroot->right->element  
 + bsroot->left->left->element  
 + bsroot->left->right->element  
 + bsroot->right->left->element  
 + bsroot->right->right->element  
 + pthread\_map[3].at(bsroot->left->left->element)  
 + pthread\_map[2].at(bsroot->left->right->element)  
 + pthread\_map[1].at(bsroot->right->left->element)  
 + pthread\_map[0].at(bsroot->right->right->element);  
 summap->insert(pair<**int**, **int**>(bsroot->element, rootsum));  
 summap->insert(pair<**int**, **int**>(bsroot->left->element,  
 bsroot->left->left->element + bsroot->left->right->element +  
 pthread\_map[3].at(bsroot->left->left->element)  
 + pthread\_map[2].at(bsroot->left->right->element)));  
 summap->insert(pair<**int**, **int**>(bsroot->right->element,  
 bsroot->right->left->element + bsroot->right->right->element +  
 pthread\_map[1].at(bsroot->right->left->element)  
 + pthread\_map[0].at(bsroot->right->right->element)));  
 **auto** end\_time = std::chrono::high\_resolution\_clock::now();  
 **auto** time = end\_time - start\_time;  
 sum4 += std::chrono::duration\_cast<std::chrono::microseconds>(time).count();  
 **for** (**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
 }  
 cout << endl;  
 cout << endl << **"Parallel time for 4 threads: "** << sum4 / C << endl;  
 cout << summap->size() << endl;  
 summap->clear();  
 **for** (**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
  
 num = 2;  
 nodes = **new** nodeptr[num];  
 nodestack = createStack();*//стек для вершин* auto\_config(bsroot, 0, num, nodestack);  
 **for** (**int** i = 0; i < num; i++) {  
 nodes[i] = (nodeptr) pop(nodestack);  
 }  
 freeStack(&nodestack);  
  
 **int** sum2 = 0;  
 **for** (**int** k = 0; k < C; k++) {  
 **auto** start\_time = std::chrono::high\_resolution\_clock::now();  
#pragma omp parallel num\_threads(2)  
 {  
#pragma omp for  
 **for** (**int** n = 0; n < num; ++n)  
 wideTreeTraversalWithThreads(n);  
 }  
 **int** rootsum = bsroot->left->element + bsroot->right->element + pthread\_map[1].at(bsroot->left->element)  
 + pthread\_map[0].at(bsroot->right->element);  
 summap->insert(pair<**int**, **int**>(bsroot->element, rootsum));  
 **auto** end\_time = std::chrono::high\_resolution\_clock::now();  
 **auto** time = end\_time - start\_time;  
 sum2 += std::chrono::duration\_cast<std::chrono::microseconds>(time).count();  
 **for** (**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
 }  
 cout << endl;  
 cout << endl << **"Parallel time for 2 threads: "** << sum2 / C << endl;  
 cout << summap->size() << endl;  
 summap->clear();  
 **for** (**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
  
  
 num = 1;  
 nodes[0] = bsroot;  
 **int** sum1 = 0;  
 **for** (**int** k = 0; k < C; k++) {  
 **auto** start\_time = std::chrono::high\_resolution\_clock::now();  
 #pragma omp parallel num\_threads(1)  
 {  
 #pragma omp for  
 **for** (**int** n = 0; n < num; ++n)  
 wideTreeTraversalWithThreads(n);  
 }  
 **auto** end\_time = std::chrono::high\_resolution\_clock::now();  
 **auto** time = end\_time - start\_time;  
 sum1+=std::chrono::duration\_cast<std::chrono::microseconds>(time).count();  
  
 **for** (**int** i = 0; i < 8; i++)  
 pthread\_map[i].clear();  
 }  
 cout << endl;  
 cout << endl << **"Sequence time: "** << sum1/C << endl;  
 cout << summap->size() << endl;  
 **delete**(summap);  
 **for**(**int** i = 0; i < 8; i++)  
 **delete**(&pthread\_map[i]);  
 **return** 0;  
}