Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего образования

«Санкт-Петербургский политехнический университет Петра Великого»

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Отчет по лабораторным работам

**Создание многопоточных программ на языке С++ с использованием OpenMP и POSIX Threads**

по дисциплине: Параллельные вычисления

Вариант 6

Выполнил студент гр. 13541/4 Гиззатуллина А.Р.

Руководитель, к.т.н., доц. Стручков И.В.

Санкт-Петербург

2017

# Содержание

[Содержание 2](#_Toc484064578)

[Задание 3](#_Toc484064579)

[Вариант 6 (OpenMP) 3](#_Toc484064580)

[Программа работы 3](#_Toc484064581)

[Ход работы 4](#_Toc484064582)

[Последовательный алгоритм на языке C++ 4](#_Toc484064583)

[Структура параллельной программы 5](#_Toc484064584)

[Параллельный алгоритм с использованием OpenMP 5](#_Toc484064585)

[Параллельный алгоритм с использованием POSIX Threads 8](#_Toc484064586)

[Запуск программы и результаты 10](#_Toc484064587)

[Тестирование работы приложения 13](#_Toc484064588)

[Эксперименты 15](#_Toc484064589)

[Подсчет вероятностных характеристик 18](#_Toc484064590)

[Выводы 19](#_Toc484064591)

[Приложение 20](#_Toc484064592)

# Задание

**Вариант 6 (OpenMP)**

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

## Программа работы

1. Для алгоритма из полученного задания написать последовательную программу на языке C или С++, реализующую этот алгоритм.
2. Для созданной последовательной программы необходимо написать 3-5 тестов, которые покрывают основные варианты функционирования программы.
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.

# Ход работы

## Характеристики рабочего устройства:

|  |
| --- |
| Процессор: Intel(R) Core(TM) i5-2450 CPU @2.50GHz 2.50GHz  Число ядер: 2  Число логических процессоров: 4 (hyper-threading) |

Для удобства анализа все варианты решения задачи были реализованы в одной программе. Полный листинг программы приведен в Приложении.

## Последовательный алгоритм на языке C++

Последовательное решение задачи выполнено с помощью рекурсивной функции *sum\_serial()*, принимающий на вход единственный параметр – дерево, в котором необходимо посчитать сумму потомков.

Листинг 1. Последовательный алгоритм

|  |
| --- |
| /\* Сумма потомков (последовательно) \*/  double sum\_serial(tnode \*tree)  {  if (tree != NULL)  {  double sum\_l = 0.0; // сумма левого поддерева  double sum\_r = 0.0; // сумма правого поддерева  // сумма потомков для левого поддерева  if (tree->left != NULL)  {  tree->left->sum = sum\_serial(tree->left);  sum\_l = tree->left->sum + tree->left->value;  }  // сумма потомков для правого поддерева  if (tree->right != NULL)  {  tree->right->sum = sum\_serial(tree->right);  sum\_r = tree->right->sum + tree->right->value;  }  // вернуть сумму потомков  return sum\_l + sum\_r;  }  return 0;  } |

## Структура параллельной программы

Структура параллельной программы приведена на Рисунке 1.

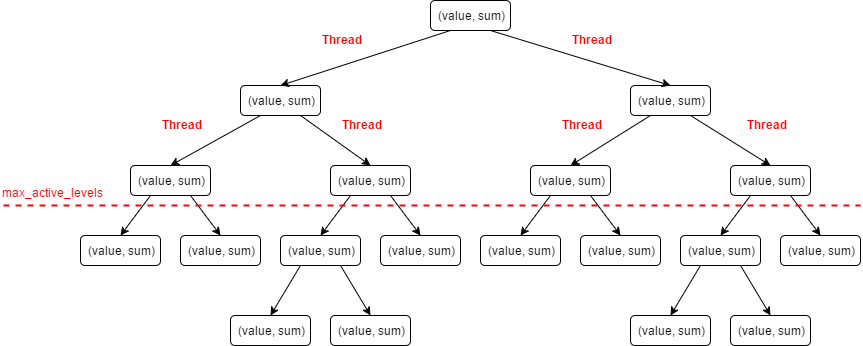


Рисунок 1. Структура параллельной программы

Каждый рекурсивный вызов функции для поддерева выполняется в новом потоке, при этом число оставшихся потоков уменьшается вдвое. При расходовании выделенного числа потоков (достижении предельной глубины вложенных параллельных регионов), программа продолжает вызывать функцию последовательно в своем потоке.

Переменная *sum*, в которой хранится сумма значений потомков конкретного узла, является разделяемой (shared) для потоков. В конце параллельной области происходит неявная барьерная синхронизация, т.е. все потоки останавливаются до тех пор, пока последний поток не выполнит код параллельной области целиком.

## Параллельный алгоритм с использованием OpenMP

Решение задачи с применением потоков OpenMP было реализовано в двух вариантах: c использованием задач и с использованием секций.

Первый вариант реализован в функции *sum\_parallel\_task()*. Входными параметрами функции являются дерево, с которым работаем, и количество создаваемых потоков *nthreads*. Задача (*task*) помещается внутрь параллельной области и задает блок операторов, который может выполняться в отдельном потоке. Задача не создает новый поток, она помещается в пул, из которого ее может взять один из свободных потоков для выполнения. Для нашего примера задачи создаются для вычислений в каждой ветке (Рисунок 2).

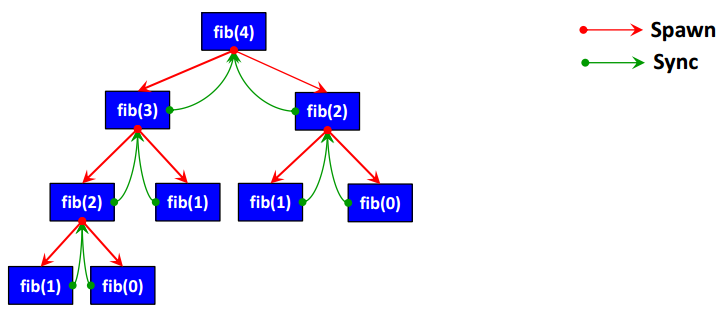


Рисунок 2. Структура параллельной программы с использованием задач

Для гарантированного завершения в точке вызова всех запущенных задач используется директива *taskwait* – нить, выполнившая данную директиву, приостанавливается до тех пор, пока не будут завершены все ранее запущенные данной нитью независимые задачи.

Листинг 2. Параллельный алгоритм (OpenMP tasks)

|  |
| --- |
| /\* Сумма потомков (параллельно задачами) \*/  double sum\_parallel\_task(tnode \*tree, int nthreads)  {  // пока не пустой узел  if (tree != NULL)  double sum\_l = 0.0; // сумма левого поддерева  double sum\_r = 0.0; // сумма правого поддерева  // последовательное выполнение при диспользовании всех выделенных потоков  if (nthreads <= 1)  return sum\_serial(tree);  if (tree->left != NULL)  {  // каждый рекурсивный вызов - это задача  #pragma omp task shared(sum\_l)  {  // в каждом вложенном регионе уменьшаем число выделяемых потоков вдвое  tree->left->sum = sum\_parallel\_task(tree->left, nthreads/2);  sum\_l = tree->left->sum + tree->left->value;  }  }  if (tree->right != NULL)  {  #pragma omp task shared(sum\_r)  {  tree->right->sum = sum\_parallel\_task(tree->right, nthreads - nthreads/2);  sum\_r = tree->right->sum + tree->right->value;  }  }  // ожидаем завершение дочерних задач  #pragma omp taskwait  return sum\_l + sum\_r;  }  return 0;  } |

Распараллеливание с помощью секций применяется в функции *sum\_parallel\_section()*. Директива *OpenMP sections* используется для выделения участков программы в области параллельных структурных блоков, выполняющихся в отдельных параллельных потоках. Для данной задачи происходит рекурсивное выделение двух секций – для левого и правого поддерева. При этом при достижения максимального уровня вложенности программа продолжает вычисления в текущей секции.

Листинг 3. Параллельный алгоритм (OpenMP sections)

|  |
| --- |
| /\* Сумма потомков (параллельно секциями) \*/  double sum\_parallel\_section(tnode \*tree)  {  if (tree != NULL)  {  double sum\_l = 0.0; // сумма левого поддерева  double sum\_r = 0.0; // сумма правого поддерева  int a = omp\_get\_max\_active\_levels();  // при достижении максимальной глубины вложенных параллельных областей  if(omp\_get\_active\_level() >= omp\_get\_max\_active\_levels())  return sum\_serial(tree);  #pragma omp parallel num\_threads(2)  {  #pragma omp sections  {  #pragma omp section  {  // сумма потомков для левого поддерева  if (tree->left != NULL)  {  tree->left->sum = sum\_parallel\_section(tree->left);  sum\_l = tree->left->sum + tree->left->value;  }  }  #pragma omp section  {  // сумма потомков для правого поддерева  if (tree->right != NULL)  {  tree->right->sum = sum\_parallel\_section(tree->right);  sum\_r = tree->right->sum + tree->right->value;  }  }  }  }  return sum\_l + sum\_r;  }  return 0;  } |

## Параллельный алгоритм с использованием POSIX Threads

Параллельный алгоритм с использованием *pthreads* реализован в функции *sum\_parallel\_pthread()*. Входным параметром функции является структура *ptArg*, которая содержит в себе указатель на корень дерева и переменную sum:

|  |
| --- |
| /\*Аргументы потока pthread\*/  struct ptArg {  struct tnode \*tree;  int nthreads;  double sum;  }; |

Для вычислений в каждом поддереве создается свой поток командой *pthread\_create()*. После отработки алгоритма ожидаем завершение всех созданных потоков командой *pthread\_join().*

Листинг 4. Параллельный алгоритм (Pthreads)

|  |
| --- |
| /\* Сумма потомков (параллельно pthread) \*/  void\* sum\_parallel\_pthread(void \*args)  {  // инициализируем специальную структуру данных  ptArg \*arg = (ptArg \*)args;    // пока не пустой узел  if (arg->tree != NULL) {  double sum\_l = 0.0; // сумма левого поддерева  double sum\_r = 0.0; // сумма правого поддерева    // последовательное выполнение при расходовании всех потоков  if (arg->nthreads <= 1) {  arg->sum = sum\_serial(arg->tree);  return 0;  }  // поток для левого поддерева  pthread\_t thread\_l;  int status\_l; // статус завершения pthread\_create  int status\_addr\_l;  ptArg arg\_l;  arg\_l.sum = 0.0;  if (arg->tree->left != NULL) {  arg\_l.tree = arg->tree->left;  arg\_l.nthreads = arg->nthreads/2;  status\_l = pthread\_create(&thread\_l,NULL,sum\_parallel\_pthread,(void\*) &arg\_l);  if (status\_l != 0) {  printf("main error: can't create thread, status = %d\n", status\_l);  exit(ERROR\_CREATE\_THREAD);  }  }  // поток для правого поддерева  pthread\_t thread\_r;  int status\_r; // статус завершения pthread\_create  int status\_addr\_r;  ptArg arg\_r;  arg\_r.sum = 0.0;  if (arg->tree->right != NULL) {  arg\_r.tree = arg->tree->right;  arg\_r.nthreads = arg->nthreads - arg->nthreads/2;  status\_r = pthread\_create(&thread\_r,NULL,sum\_parallel\_pthread,(void\*) &arg\_r);  if (status\_r != 0) {  printf("main error: can't create thread, status = %d\n", status\_r);  exit(ERROR\_CREATE\_THREAD);  }  }  // ожидание завершения потоков  status\_l = pthread\_join(thread\_l, (void\*\*)&status\_addr\_l);  if (status\_l != SUCCESS) {  printf("main error: can't join thread, status = %d\n", status\_l);  exit(ERROR\_JOIN\_THREAD);  }  if (arg->tree->left != NULL) {  arg->tree->left->sum = arg\_l.sum;  sum\_l = arg->tree->left->sum + arg->tree->left->value;  }  status\_r = pthread\_join(thread\_r, (void\*\*)&status\_addr\_r);  if (status\_r != SUCCESS) {  printf("main error: can't join thread, status = %d\n", status\_r);  exit(ERROR\_JOIN\_THREAD);  }  if (arg->tree->right != NULL) {  arg->tree->right->sum = arg\_r.sum;  sum\_r = arg->tree->right->sum + arg->tree->right->value;  }  arg->sum = sum\_l + sum\_r;  }  return 0;  } |

## Запуск программы и результаты

Для реализации описанных функций была создана программа, листинг которой представлен ниже. Число узлов дерева задается переменной *arr\_size.* Для параллельных алгоритмов число выделяемых потоков задается переменной *nthreads* (по умолчанию выделяется число потоков равное числу логических процессоров на рабочем устройстве).

Для каждого алгоритма засекаем время выполнения для возможности последующего сравнения скорости их работы.

Листинг 5. Основная программа

|  |
| --- |
| #include "stdafx.h"  #include "SumTree.h"  using namespace std;  int main()  {  system("chcp 1251");  system("cls");  srand(time(NULL));    // размерность массива определяет число узлов в дереве  int arr\_size = 1000000;  printf("Size : \t\t\t%ld\n", arr\_size);    // массив под дерево  double \*arrForTree = new double[arr\_size];  // заполнение массива случайными числами [0,1)  for (int i = 0; i < arr\_size; i++)  {  arrForTree[i] = (double)rand() / (RAND\_MAX + 1.0);  }    // Формирование бинарных деревьев под каждый алгоритм  tnode\* rootSerial = NULL;  tnode\* rootParallelSections = NULL;  tnode\* rootParallelTasks = NULL;  tnode\* rootParallelPthread = NULL;  for (int i = 0; i < arr\_size; i++)  {  rootSerial = addNode(arrForTree[i], rootSerial);  rootParallelSections = addNode(arrForTree[i], rootParallelSections);  rootParallelTasks = addNode(arrForTree[i], rootParallelTasks);  rootParallelPthread = addNode(arrForTree[i], rootParallelPthread);  }  // временные засечки  double start\_time, end\_time;  // задаем число потоков для параллельных алгоритмов  int nthreads = omp\_get\_num\_procs(); // omp\_get\_num\_procs() = 4  printf("Threads number : \t%i\n\n", nthreads);    /\* Последовательный алгоритм.\*/  // старт таймер  start\_time = omp\_get\_wtime();  // считать последовательно  rootSerial->sum = sum\_serial(rootSerial);  // стоп таймер  end\_time = omp\_get\_wtime();  // вывод времени  printf("Serial time: \t\t\t%lf sec.\n", end\_time-start\_time);    /\* Параллельный алгоритм с использованием секций OpenMP.\*/    // включить вложенный параллелизм  omp\_set\_nested(1);  // максимальная глубина вложенных областей  omp\_set\_max\_active\_levels(log2(nthreads));    // старт таймер  start\_time = omp\_get\_wtime();  // считаем параллельно секциями  rootParallelSections->sum = sum\_parallel\_section(rootParallelSections);  // стоп таймер  end\_time = omp\_get\_wtime();  // вывод времени  printf("Parallel (sections) time: \t%lf sec.\n", end\_time-start\_time);    /\* Параллельный алгоритм с использованием задач OpenMP.\*/  // старт таймер  start\_time = omp\_get\_wtime();  // считаем параллельно задачами  #pragma omp parallel  {  #pragma omp single nowait  rootParallelTasks->sum = sum\_parallel\_task(rootParallelTasks,nthreads);  }  // стоп таймер  end\_time = omp\_get\_wtime();  // вывод времени  printf("Parallel (tasks) time: \t\t%lf sec.\n", end\_time-start\_time);  /\* Параллельный алгоритм с использованием Pthread.\*/  // старт таймер  start\_time = omp\_get\_wtime();  // считаем параллельно потоками pthread  ptArg arg;  arg.tree = rootParallelPthread;  arg.nthreads = nthreads;  arg.sum = 0.0;  sum\_parallel\_pthread((void \*) &arg);  rootParallelPthread->sum = arg.sum;  // стоп таймер  end\_time = omp\_get\_wtime();  // вывод времени  printf("Parallel (pthread) time: \t%lf sec.\n", end\_time-start\_time);  system("pause");  // Освобождение памяти.  delete[] arrForTree;  freeTree(rootSerial);  freeTree(rootParallelSections);  freeTree(rootParallelTasks);  freeTree(rootParallelPthread);  return 0;  } |

Результат работы приложения:

|  |
| --- |
| Size: 1000000  Serial time: 0.125471 sec.  Parallel (sections) time: 0.075299 sec.  Parallel (tasks) time: 0.072970 sec.  Parallel (pthread) time: 0.075015 sec.  Для продолжения нажмите любую клавишу . . . |

Из полученных результатов можно увидеть, что распараллеливание программы любым из методов позволяет ускорить время вычислений на 40%:

* OpenMP sections:
* OpenMP tasks:
* Pthreads:

Данное значение обусловлено наличием двух ядер процессора на рабочем устройстве.

# Тестирование работы приложения

## Модульные тесты

Unit-тесты для проверки функционирования основных методов программы были написаны с использованием средств Visual Studio 2010.

Листинг 6. Unit-тесты

|  |
| --- |
| #include "stdafx.h"  #include "SumTree.h"  using namespace System;  using namespace System::Text;  using namespace System::Collections::Generic;  using namespace Microsoft::VisualStudio::TestTools::UnitTesting;  namespace TestStruchkov  {  [TestClass]  public ref class UnitTest  {  public:  /\* Тест логарифма по основанию 2\*/  [TestMethod]  void TestLog2()  {  Assert::AreEqual(3, log2(8)); // (предполагаемый, полученный на деле)  }  /\* Тест пустого дерева\*/  [TestMethod]  void TestNullTree()  {  Assert::AreEqual(0.0, sum\_serial(NULL));  }  /\* Тест дерева из единственного узла\*/  [TestMethod]  void TestSingleTree()  {  tnode node = {5.0,0.0,NULL,NULL};  Assert::AreEqual(0.0, sum\_serial(&node));  }  /\* Тест последовательного расчета суммы дерева\*/  [TestMethod]  void TestSumSerial()  {  tnode\* root = NULL;  root = addNode(10,root); // корень 10  addNode(5,root); // узел 5  addNode(15,root); // узел 15  addNode(7,root); // узел 7  addNode(20.3,root); // узел 20.3  double result = 5+15+7+20.3; // сумма потомков  Assert::AreEqual(result, sum\_serial(root));  freeTree(root);  }  /\* Тест параллельного расчета суммы дерева (sections)\*/  [TestMethod]  void TestSumParallel()  {  tnode\* root = NULL;  root = addNode(10,root); // корень 10  addNode(5,root); // узел 5  addNode(15,root); // узел 15  addNode(7,root); // узел 7  addNode(20.3,root); // узел 20.3  double result = 5+15+7+20.3; // сумма потомков  Assert::AreEqual(result, sum\_parallel\_section(root));  freeTree(root);  }  /\* Тест параллельного расчета суммы дерева (pthread)\*/  [TestMethod]  void TestSumParallelPthread()  {  // строим дерево  tnode\* root = NULL;  root = addNode(10,root); // корень 10  addNode(5,root); // узел 5  addNode(15,root); // узел 15  addNode(7,root); // узел 7  addNode(20.3,root); // узел 20.3  double result = 5+15+7+20.3; // сумма потомков  ptArg arg;  arg.tree = root;  arg.nthreads = 2;  arg.sum = 0.0;  sum\_parallel\_pthread((void \*) &arg);  Assert::AreEqual(result, arg.sum);  freeTree(root);  }  };  } |

Результаты тестирования:

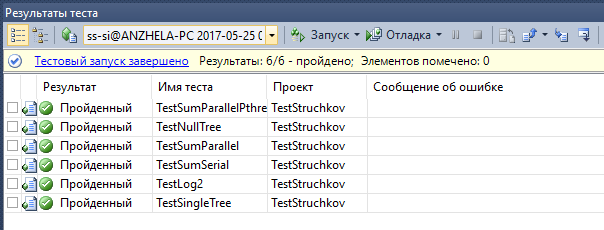


Рисунок 3. Результаты тестирования программы

## Эксперименты

Исследуем зависимость времени выполнения алгоритмов от количества узлов бинарного дерева (число выделяемых потоков равно числу логических процессоров):

Таблица 1. Зависимость от количества узлов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Число узлов | Последовательный | Параллельный | | |
| OpenMP (sections) | OpenMP (tasks) | POSIX threads |
| 100 | 0.000039 | 0.002095 | 0.000114 | 0.004252 |
| 1000 | 0.000069 | 0.001790 | 0.000120 | 0.003266 |
| 10000 | 0.000903 | 0.002128 | 0.000681 | 0.004355 |
| 100000 | 0.011531 | 0.009422 | 0.008174 | 0.011399 |
| 500000 | 0.062279 | 0.040049 | 0.038376 | 0.043335 |
| 1000000 | 0.151614 | 0.116494 | 0.122371 | 0.108747 |
| 5000000 | 0.847389 | 0.552983 | 0.586042 | 0.562480 |

Желтым маркером выделено самое быстрое время выполнения для конкретного числа узлов. Для наглядного отображения зависимости на основе полученных данных построим график:

Рисунок 4. График зависимости скорости выполнения алгоритмов от числа узлов дерева

Как видим, при количестве узлов до 100 000 параллельные алгоритмы уступают в скорости работы последовательному. И только при больших значениях распараллеливание вычислений дает преимущество.

Теперь исследуем зависимость времени выполнения алгоритмов от количества выделенных потоков. Число узлов в дереве - 1 000 000.

Таблица 2. Зависимость от числа потоков

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Число потоков | Последовательный, сек | Параллельный, сек | | |
| OpenMP (sections) | OpenMP (tasks) | POSIX threads |
| 2 | 0.123623 | 0.119969 | 0.132326 | 0.135186 |
| 4 | 0.125070 | 0.075299 | 0.072970 | 0.075015 |
| 6 | 0.128785 | 0.108205 | 0.111871 | 0.105058 |
| 8 | 0.131050 | 0.068711 | 0.074194 | 0.073601 |
| 10 | 0.132343 | 0.074105 | 0.074031 | 0.079673 |
| 12 | 0.130208 | 0.059333 | 0.065671 | 0.067210 |
| 14 | 0.127700 | 0.077431 | 0.080096 | 0.086045 |
| 16 | 0.129787 | 0.085640 | 0.067298 | 0.097827 |
| 18 | 0.130565 | 0.076287 | 0.049697 | 0.102140 |
| 20 | 0.126404 | 0.066419 | 0.047926 | 0.106288 |
| 32 | 0.125728 | 0.073017 | 0.060250 | 0.115464 |
| 64 | 0.125975 | 0.112044 | 0.078223 | 0.116484 |
| 128 | 0.130026 | 0.106303 | 0.065002 | 0.111235 |

Желтым маркером выделено лучшее время выполнения для конкретного числа потоков. Красным – для конкретного алгоритма.

Рисунок 5. График зависимости скорости выполнения алгоритмов от числа выделенных потоков

Исходя из полученных данных, можно сделать следующие выводы:

* При применении секций OpenMP оптимально выделять 12 потоков.
* При применении задач OpenMP – 20 потоков.
* При применении POSIX threads – 12 потоков.
* Нецелесообразно выделять потоки, не кратные числу логических процессоров.
* При выделении более 20 потоков (для pthreads – 12) соблюдается Закон Амдала: выделение большего числа потоков приводит лишь к ухудшению производительности.

## Подсчет вероятностных характеристик

Для более объективной оценки работы приложения необходимо провести многократный запуск алгоритмов с одинаковыми параметрами и сделать статистический анализ выборок: вычислить математическое ожидание, дисперсию и доверительный интервал для оценки среднего каждого эксперимента.

Проведем оценку показателей, выделив параллельным алгоритмам 4 потока для работы с 1 000 000 узлов дерева. Сделаем выборку из 10 запусков программы.

Таблица 3. Выборка для анализа

|  |  |  |  |
| --- | --- | --- | --- |
| Последовательный, сек | Параллельный, сек | | |
| OpenMP (sections) | OpenMP (tasks) | POSIX threads |
| 0.125070 | 0.075299 | 0.072970 | 0.075015 |
| 0.130208 | 0.084513 | 0.071649 | 0.076489 |
| 0.127700 | 0.074765 | 0.073416 | 0.089461 |
| 0.129787 | 0.079165 | 0.079854 | 0.079146 |
| 0.130565 | 0.079414 | 0.076474 | 0.074128 |
| 0.126404 | 0.094561 | 0.087941 | 0.104164 |
| 0.125728 | 0.089451 | 0.089414 | 0.084164 |
| 0.125975 | 0.071451 | 0.077421 | 0.071464 |
| 0.125975 | 0.069414 | 0.078944 | 0.084146 |
| 0.130026 | 0.074915 | 0.071461 | 0.079144 |

Расчет показателей был произведен с помощью средств Microsoft Excel 2013:

Таблица 4. Вероятностные характеристики

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Показатель | Последовательный | Параллельный | | |
| OpenMP (sections) | OpenMP (tasks) | POSIX threads |
| Среднее значение | 0.1277438 | 0.079295 | 0.077954 | 0.081732 |
| Дисперсия | 4.74239E-06 | 6.42544E-05 | 4.06091E-05 | 9.15466E-05 |
| Доверительный интервал (P = 0.95) | [0.126394071-0.127722219] | [0.074326595-0.079215363] | [0.074004738-0.077891249] | [0.075801903-0.081637282] |

Исходя из полученных результатов можно сделать вывод, что наилучшим образом работает алгоритм распараллеливания с применением задач в OpenMP. Данный алгоритм обеспечивает самую быструю среднюю скорость вычислений и минимальное значение дисперсии.

# Выводы

В ходе данной лабораторной работы нами были получены навыки распараллеливания программы с применением библиотек OpenMP и Pthreads для решения задачи расчета суммы значений потомков узлов бинарного дерева.

Для созданной программы были написаны 6 модульных тестов с проверкой правильности вычислений основных функций.

Были проведены эксперименты с измерением скорости работы разных алгоритмов в зависимости от числа узлов дерева и выделенных потоков. Результаты экспериментов показали, что самым оптимальным количеством выделенных потоков является значение равное числу логических процессоров на рабочем устройстве.

Для одного набора параметров было сформировано 10 выборок и проведен анализ вероятностных характеристик с оценкой среднего значения, дисперсии и доверительных интервалов.

По итогам проделанной работы можно сделать вывод, что разные методы многопоточного распараллеливания алгоритмов не имеют между собой существенных различий, однако их применение очень эффективно в сравнении с последовательным выполнением программы, особенно при работе с большим количеством обрабатываемых данных на устройствах с многоядерными процессорами.

# Приложение

Листинг 7. Заголовочный файл программы

|  |
| --- |
| // SumTree.h - заголовочный файл: содержит описание функций для работы с деревьями  /\* Структура бинарного дерева\*/  struct tnode  {  double value; // числовое значение  double sum; // сумма значений дочерних узлов  struct tnode \*left; // левый потомок  struct tnode \*right; // правый потомок  };  /\* Добавить узел \*/  tnode\* addNode(double v, tnode \*tree);  /\* Очистить дерево \*/  void freeTree(tnode \*tree);  /\* Вывод дерева \*/  void printTree(tnode \*tree);  /\* Сумма потомков (последовательно) \*/  double sum\_serial(tnode \*tree);  /\* Сумма потомков (параллельно задачами) \*/  double sum\_parallel\_task(tnode \*tree, int nthreads);  /\* Сумма потомков (параллельно секциями) \*/  double sum\_parallel\_section(tnode \*tree);  /\* Сумма потомков (параллельно pthred) \*/  void\* sum\_parallel\_pthread(void \*args);  /\* Логарифм по основанию 2 \*/  int log2( unsigned int n ); |

Листинг 8. Файл реализации алгоритмов

|  |
| --- |
| // SumTree.cpp - содержит функции для работы с деревьями  #include "stdafx.h"  #include "SumTree.h"  using namespace std;  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* РАБОТА С БИНАРНЫМ ДЕРЕВОМ \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* Добавить узел \*/  tnode\* addNode(double v, tnode \*tree)  {  // Если дерева нет, то формируем корень  if (tree == NULL)  {  tree = new tnode; // память под узел  tree->value = v; // значение  tree->sum = 0; // сумма дочерних  tree->left = NULL; // ветви инициализируем пустотой  tree->right = NULL;  }  else if (v < tree->value) // условие добавление левого потомка  tree->left = addNode(v,tree->left);  else // условие добавление правого потомка  tree->right = addNode(v,tree->right);  return(tree);  }  /\* Очистить дерево \*/  void freeTree(tnode \*tree)  {  if(tree!=NULL) {  freeTree(tree->left);  freeTree(tree->right);  delete tree;  }  }  /\* Вывод дерева (в префиксной форме) \*/  void printTree(tnode \*tree)  {  if (tree != NULL)  {  cout << tree->value << " " << tree->sum << endl; //Отображаем корень дерева  printTree(tree->left); //Рекурсивная функция для левого поддерева  printTree(tree->right); //Рекурсивная функция для правого поддерева  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ПОСЛЕДОВАТЕЛЬНЫЙ АЛГОРИТМ \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* Сумма потомков (последовательно) \*/  double sum\_serial(tnode \*tree)  {  // пока не пустой узел  if (tree != NULL)  {  double sum\_l = 0.0; // сумма левого поддерева  double sum\_r = 0.0; // сумма правого поддерева  // сумма потомков для левого поддерева  if (tree->left != NULL)  {  tree->left->sum = sum\_serial(tree->left);  sum\_l = tree->left->sum + tree->left->value;  }  // сумма потомков для правого поддерева  if (tree->right != NULL)  {  tree->right->sum = sum\_serial(tree->right);  sum\_r = tree->right->sum + tree->right->value;  }  // вернуть сумму потомков  return sum\_l + sum\_r;  }  return 0;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ С ИСПОЛЬЗОВАНИЕМ OPENMP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* Сумма потомков (параллельно задачами) \*/  double sum\_parallel\_task(tnode \*tree, int nthreads)  {  // пока не пустой узел  if (tree != NULL)  {  double sum\_l = 0.0; // сумма левого поддерева  double sum\_r = 0.0; // сумма правого поддерева  // последовательное выполнение при выделении одного потока  if (nthreads <= 1)  return sum\_serial(tree);  if (tree->left != NULL)  {  #pragma omp task shared(sum\_l)  {  // в каждом вложенном регионе уменьшаем число выделяемых потоков вдвое  tree->left->sum = sum\_parallel\_task(tree->left, nthreads/2);  sum\_l = tree->left->sum + tree->left->value;  }  }  if (tree->right != NULL)  {  #pragma omp task shared(sum\_r)  {  tree->right->sum = sum\_parallel\_task(tree->right, nthreads - nthreads / 2);  sum\_r = tree->right->sum + tree->right->value;  }  }  #pragma omp taskwait  return sum\_l + sum\_r;  }  return 0;  }  /\* Сумма потомков (параллельно секциями) \*/  double sum\_parallel\_section(tnode \*tree)  {  // пока не пустой узел  if (tree != NULL)  {  double sum\_l = 0.0; // сумма левого поддерева  double sum\_r = 0.0; // сумма правого поддерева  // последовательное выполнение при макс уровне вложенности  if(omp\_get\_active\_level() >= omp\_get\_max\_active\_levels())  return sum\_serial(tree);  #pragma omp parallel num\_threads(2)  {  #pragma omp sections  {  #pragma omp section  {  // сумма потомков для левого поддерева  if (tree->left != NULL)  {  tree->left->sum = sum\_parallel\_section(tree->left);  sum\_l = tree->left->sum + tree->left->value;  }  }  #pragma omp section  {  // сумма потомков для правого поддерева  if (tree->right != NULL)  {  tree->right->sum = sum\_parallel\_section(tree->right);  sum\_r = tree->right->sum + tree->right->value;  }  }  }  }  return sum\_l + sum\_r;  }  return 0;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ С ИСПОЛЬЗОВАНИЕМ PTHREAD \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* Сумма потомков (параллельно pthread) \*/  void\* sum\_parallel\_pthread(void \*args)  {  // инициализируем специальную структуру данных  ptArg \*arg = (ptArg \*)args;    // пока не пустой узел  if (arg->tree != NULL)  {  double sum\_l = 0.0; // сумма левого поддерева  double sum\_r = 0.0; // сумма правого поддерева    // последовательное выполнение при расходовании всех потоков  if (arg->nthreads <= 1)  {  arg->sum = sum\_serial(arg->tree);  return 0;  }  // поток для левого поддерева  pthread\_t thread\_l;  int status\_l;  int status\_addr\_l;  ptArg arg\_l;  arg\_l.sum = 0.0;  if (arg->tree->left != NULL)  {  arg\_l.tree = arg->tree->left;  arg\_l.nthreads = arg->nthreads/2;  status\_l = pthread\_create(&thread\_l, NULL, sum\_parallel\_pthread, (void\*) &arg\_l);  if (status\_l != 0) {  printf("main error: can't create thread, status = %d\n", status\_l);  exit(ERROR\_CREATE\_THREAD);  }  }  // поток для правого поддерева  pthread\_t thread\_r;  int status\_r; // статус завершения pthread\_create  int status\_addr\_r; // статус pthread\_join  ptArg arg\_r;  arg\_r.sum = 0.0;  if (arg->tree->right != NULL)  {  arg\_r.tree = arg->tree->right;  arg\_r.nthreads = arg->nthreads - arg->nthreads/2;  status\_r = pthread\_create(&thread\_r, NULL, sum\_parallel\_pthread, (void\*) &arg\_r);  if (status\_r != 0) {  printf("main error: can't create thread, status = %d\n", status\_r);  exit(ERROR\_CREATE\_THREAD);  }  }    // ожидание завершения потоков  status\_l = pthread\_join(thread\_l, (void\*\*)&status\_addr\_l);  if (status\_l != SUCCESS) {  printf("main error: can't join thread, status = %d\n", status\_l);  exit(ERROR\_JOIN\_THREAD);  }  if (arg->tree->left != NULL)  {  arg->tree->left->sum = arg\_l.sum;  sum\_l = arg->tree->left->sum + arg->tree->left->value;  }  status\_r = pthread\_join(thread\_r, (void\*\*)&status\_addr\_r);  if (status\_r != SUCCESS) {  printf("main error: can't join thread, status = %d\n", status\_r);  exit(ERROR\_JOIN\_THREAD);  }  if (arg->tree->right != NULL)  {  arg->tree->right->sum = arg\_r.sum;  sum\_r = arg->tree->right->sum + arg->tree->right->value;  }    arg->sum = sum\_l + sum\_r;  }  return 0;  }  /\* Логарифм по основанию 2 \*/  int log2( unsigned int n )  {  return unsigned int(log( (double)n ) / log( 2.0 ));  } |

Листинг 9. Файл запуска приложения

|  |
| --- |
| #include "stdafx.h"  #include "SumTree.h"  using namespace std;  int main()  {  system("chcp 1251");  system("cls");  srand(time(NULL));  // файл для записи результатов  ofstream fout("results.txt");    // размерность массива определяет число узлов в дереве  int arr\_size = 1000000;  fout << "Size : " << arr\_size << endl;    // массив под дерево  double \*arrForTree = new double[arr\_size];  // заполнение массива случайными числами [0,1)  for (int i = 0; i < arr\_size; i++)  {  arrForTree[i] = (double)rand() / (RAND\_MAX + 1.0);  }    // Формирование бинарных деревьев под каждый алгоритм  tnode\* rootSerial = NULL;  tnode\* rootParallelSections = NULL;  tnode\* rootParallelTasks = NULL;  tnode\* rootParallelPthread = NULL;  for (int i = 0; i < arr\_size; i++)  {  rootSerial = addNode(arrForTree[i], rootSerial);  rootParallelSections = addNode(arrForTree[i], rootParallelSections);  rootParallelTasks = addNode(arrForTree[i], rootParallelTasks);  rootParallelPthread = addNode(arrForTree[i], rootParallelPthread);  }  // временные засечки  double start\_time, end\_time;  // задаем число потоков для параллельных алгоритмов  int nthreads = omp\_get\_num\_procs(); // omp\_get\_num\_procs() = 4  fout << "Threads number : " << nthreads << endl << endl;    /\* Последовательный алгоритм.\*/  // старт таймер  start\_time = omp\_get\_wtime();  // считать последовательно  rootSerial->sum = sum\_serial(rootSerial);  // стоп таймер  end\_time = omp\_get\_wtime();  // вывод времени  fout << "Serial time: " << end\_time-start\_time << endl;    /\* Параллельный алгоритм с использованием секций OpenMP.\*/    // включить вложенный параллелизм  omp\_set\_nested(1);  // максимальная глубина вложенных областей  omp\_set\_max\_active\_levels(log2(nthreads));    // старт таймер  start\_time = omp\_get\_wtime();  // считаем параллельно секциями  rootParallelSections->sum = sum\_parallel\_section(rootParallelSections);  // стоп таймер  end\_time = omp\_get\_wtime();  // вывод времени  fout << "Parallel (sections) time: " << end\_time-start\_time << endl;  /\* Параллельный алгоритм с использованием задач OpenMP \*/  // старт таймер  start\_time = omp\_get\_wtime();  // считаем параллельно задачами  #pragma omp parallel  {  #pragma omp single nowait  rootParallelTasks->sum = sum\_parallel\_task(rootParallelTasks,nthreads);  }  // стоп таймер  end\_time = omp\_get\_wtime();  // вывод времени  fout << "Parallel (tasks) time: " << end\_time-start\_time << endl;  /\*  \*\* Параллельный алгоритм с использованием Pthread  \*/  // старт таймер  start\_time = omp\_get\_wtime();  // считаем параллельно потоками pthread  ptArg arg;  arg.tree = rootParallelPthread;  arg.nthreads = nthreads;  arg.sum = 0.0;  sum\_parallel\_pthread((void \*) &arg);  rootParallelPthread->sum = arg.sum;  // стоп таймер  end\_time = omp\_get\_wtime();  // вывод времени  fout << "Parallel (pthread) time: " << end\_time-start\_time << endl;  fout.close();  // Освобождение памяти.  delete[] arrForTree;  freeTree(rootSerial);  freeTree(rootParallelSections);  freeTree(rootParallelTasks);  freeTree(rootParallelPthread);  return 0;  } |