Санкт-Петербургский Политехнический Университет Петра Великого

Кафедра компьютерных систем и программных технологий

Отчёт по курсовой работе

Курс: «Параллельные вычисления»

Тема: «Расчет суммы чисел для каждой вершины дерева»

Выполнил студент:

Бояркин Никита Сергеевич
Группа: 13541/3

Проверил:

Стручков Игорь Вячеславович

Санкт-Петербург
2018 г.

Содержание

Курсовая работа

1.1 Цель работы

Целью данной работы является получение студентами навыков создания многопоточных программ с использованием интерфейсов POSIX Threads, OpenMP и MPI.

1.2 Программа работы

- Для алгоритма из полученного задания написать последовательную программу на языке C или C++, реализующую этот алгоритм.
- 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.
- Проанализировать полученный алгоритм, выделить части, которые могут быть распараллелены, разработать структуру параллельной программы. Определить количество используемых потоков, а также правила и используемые объекты синхронизации.
- Согласовать разработанную структуру и детали реализации параллельной программы с преподавателем.
- Написать код параллельной программы и проверить ее корректность на созданном ранее наборе тестов. При необходимости найти и исправить ошибки.
- 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.

1.3 Индивидуальное задание

Вариант №6

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

Средство распараллеливания – **MPI**.

1.4 Компиляция и компоновка проекта

В качестве среды разработки используется IDE Netbeans, в качестве компилятора и компоновщика используется специальная обертка над компилятором g++ и компоновщиком ld, которая запускает их с определенными флагами командной строки для осуществления распараллеливания MPI:

```
1 nikita@nikita-VirtualBox:~$ mpic++.openmpi -v
2 Using built-in specs.
3 COLLECT_GCC=/usr/bin/g++
```

```
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper
Target: x86_64-linux-gnu
< ... >
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.9)
```

Дополнительные флаги, необходимые для компиляции проекта:

```
mpic++.openmpi -std=c++11 <sources>
```

Для запуска многопроцессного приложения MPI используется специальная команда, если ее не использовать, приложение запустится на одном процессе:

```
mpirun.openmpi <executable>
```

1.5 Ход работы

1.5.1 Формализация задачи и детали реализации

- Дерево не бинарное, так как это явно не указывается в формулировке задания.
- Реализация дерева – собственный класс на языке C++.
- Расчет суммы чисел всех вершин, для которых данная вершина является корнем, является задачей нахождения нового дерева с вершинами равными этим суммам, пример представлен на рис. 1.1.
- Различные алгоритмы реализации расчета дерева сумм (рекурсивная, последовательная, многопоточная pthread, многопроцессная MPI) являются функциями, вызываемыми на объекте дерева.
- Численные значения вершин дерева – числа с плавающей точкой, могут быть отрицательными.
- Точность значений с плавающей точкой при получении дерева сумм может теряться для упрощения задачи.
- Объект дерева задается из символьной строки и может быть конвертирован в символьную строку через функции, вызываемые непосредственно на объекте дерева или статически на классе.

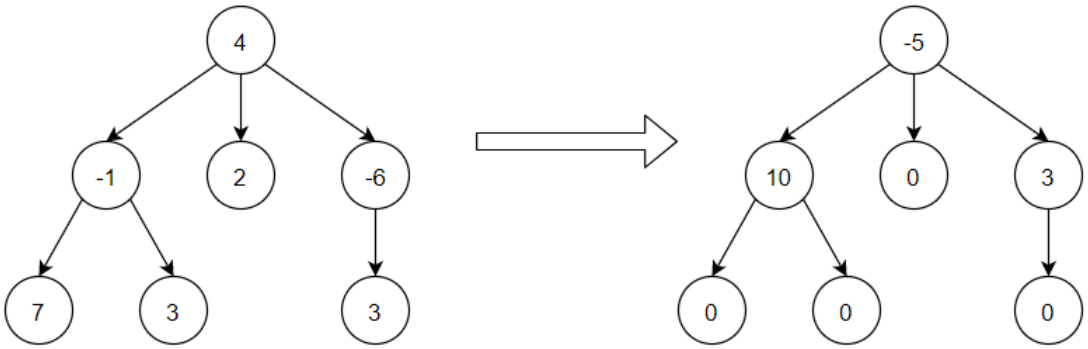

Рис. 1.1: Нахождение нового дерева с суммами вершин

1.5.2 Разработка структуры данных и общих функций

Класс дерева tree имеет достаточно несложную реализацию:

```
< ... >

class tree {
private:
    < ... >

    class node {
    private:
```

```
        double mValue;
        std::vector<tree::node> mChilds;

        < ... >

    public:
        node();
        node(const double value);
        static tree::node parseNode(const char* value);
        std::string toString() const;
        void setValue(const double value);
        double getValue() const;
        void addChild(const tree::node& child);
        std::vector<tree::node> getChilds() const;

        < ... >
    };

private:
    tree::node mRoot;
    tree(const tree::node root);
public:
    static tree parseTree(const char* value);
    std::string toString() const;

    < ... >
};

< ... >
```

На классе дерева есть подкласс вершины node. Каждая вершина содержит собственное численное значение и набор вершин для которых данная вершина является корнем.

Само дерево содержит только корневую вершину.

Задать дерево можно из символьной строки в определенном формате:

```
4(-1(7(),3()),2(),-6(3()))
```

Данная строка соответствует следующему дереву:

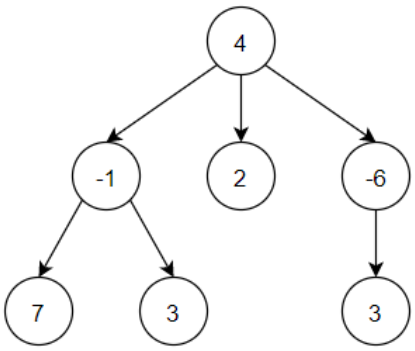

Рис. 1.2: Пример дерева из символьной строки

Реализация функции перевода строки в дерево:

```
< ... >

tree::node tree::node::parseNode(const char* value) {
    tree::node result = tree::node();

    std::stack<std::pair<tree::node*, std::string>> stackStringNodes;
    stackStringNodes.push(std::pair<tree::node*, std::string>(nullptr, std::string(value)
    ));

    while (!stackStringNodes.empty()) {
```

```
10          std::pair<tree::node*, std::string> currentPair = stackStringNodes.top();
11          stackStringNodes.pop();
12          tree::node* parentNodePtr = currentPair.first;
13          std::string currentStringNodePtr = currentPair.second;
14
15          std::size_t openBracketIndex = currentStringNodePtr.find_first_of(tree::
        OPEN_BRACKET);
16          if (openBracketIndex == std::string::npos || openBracketIndex == 0) {
17              throw std::invalid_argument("No open bracket or no value.");
18          }
19
20          std::size_t closeBracketIndex = currentStringNodePtr.find_last_of(tree::
        CLOSE_BRACKET);
21          if (closeBracketIndex == std::string::npos || openBracketIndex >=
        closeBracketIndex || closeBracketIndex != currentStringNodePtr.length() - 1) {
22              throw std::invalid_argument("No close bracket or close bracket isn't last
        symbol.");
23          }
24
25          std::string valueString = currentStringNodePtr.substr(0, openBracketIndex);
26          double parsedValue = std::stod(valueString);
27
28          std::string childsString = currentStringNodePtr.substr(openBracketIndex + 1,
        closeBracketIndex - (openBracketIndex + 1));
29          std::vector<std::string> childNodes;
30          std::string buffer;
31          int64_t bracketEquality = 0;
32          for(const auto& currentSymbol : childsString) {
33              if (currentSymbol == tree::OPEN_BRACKET) {
34                  ++bracketEquality;
35              } else if (currentSymbol == tree::CLOSE_BRACKET) {
36                  --bracketEquality;
37              }
38
39              if (bracketEquality < 0) {
40                  throw std::invalid_argument("Invalid brackets.");
41              }
42
43              if (bracketEquality == 0 && currentSymbol == tree::DELIMITER) {
44                  childNodes.push_back(buffer);
45                  buffer.clear();
46              } else {
47                  buffer += currentSymbol;
48              }
49          }
50
51          if (bracketEquality != 0) {
52              throw std::invalid_argument("Invalid brackets.");
53          }
54
55          if (!buffer.empty()) {
56              childNodes.push_back(buffer);
57              buffer.clear();
58          }
59
60          tree::node* parsedNodePtr = nullptr;
61          if (parentNodePtr == nullptr) {
62              parsedNodePtr = &result;
63              parsedNodePtr->mValue = parsedValue;
64          } else {
65              parentNodePtr->mChilds.insert(parentNodePtr->mChilds.begin(), tree::node(
        parsedValue));
66              parsedNodePtr = &(parentNodePtr->mChilds.front());
67          }
68
69          if (childNodes.empty()) {
```

```
70              continue;
71          }
72
73          for(auto& currentChildNode : childNodes) {
74              stackStringNodes.push(std::pair<tree::node*, std::string>(parsedNodePtr,
    currentChildNode));
75          }
76      }
77
78      return result;
79  }
80
81  < ... >
```

Функция целенаправленно была реализована без использования рекурсии, для того, чтобы можно было быстро парсить большие деревья.

1.5.3 Разработка последовательной реализации

Самой простой реализацией является рекурсивная:

```
1  < ... >
2
3  tree::node tree::node::getSimpleRecursiveSumSubNode() const {
4      double sum = 0;
5      tree::node result = tree::node();
6      for (const auto& currentChild : mChilds) {
7          sum += currentChild.mValue;
8          result.addChild(currentChild.getSimpleRecursiveSumSubNode());
9      }
10
11     result.mValue = sum;
12     return result;
13 }
14
15 < ... >
```

Проблемой рекурсии в C++ является то, что стек быстро заканчивается, а компилятор не всегда распознает рекурсию и не разворачивает ее в цикл.

Таким образом для больших деревьев намного лучше подходит реализация при помощи цикла:

```
1  < ... >
2
3  tree::node tree::node::getSimpleCyclicSumSubNode() const {
4      tree::node result = tree::node(*this);
5
6      std::stack<std::pair<tree::node*, tree::node*>> stackNodes;
7      stackNodes.push(std::pair<tree::node*, tree::node*>(nullptr, &result));
8
9      while (!stackNodes.empty()) {
10         std::pair<tree::node*, tree::node*> currentPair = stackNodes.top();
11         stackNodes.pop();
12         tree::node* parentNodePtr = currentPair.first;
13         tree::node* currentNodePtr = currentPair.second;
14
15         if (parentNodePtr != nullptr) {
16             parentNodePtr->mValue += currentNodePtr->mValue;
17         }
18
19         currentNodePtr->mValue = 0;
20
21         if (currentNodePtr->mChilds.empty()) {
22             continue;
23         }
24
25         for (auto currentChild = currentNodePtr->mChilds.end() - 1; currentChild >=
    currentNodePtr->mChilds.begin(); --currentChild) {
```

```
26              stackNodes.push(std::pair<tree::node*, tree::node*>(currentNodePtr, &(*
    currentChild)));
27          }
28      }
29
30      return result;
31  }
32
33  < ... >
```

Для разворачивания рекурсии в цикл был добавлен собственный стек, который хранит указатели на вершину и ее корень, для последующего суммирования.

1.5.4 Разработка многопоточной реализации при помощи pthread

Распараллеливание подобного алгоритма подразумевает создание нового потока для каждой вершины дерева. Однако, такая реализация абсолютно неэффективна, намного быстрее и эффективнее по памяти будет реализация с ограничением количества работающих потоков до определенного значения. Кроме того, каждый поток, отрабатывая собственную вершину не должен ожидать другие, а должен переходить к следующей необработанной вершине.

Распараллеленный алгоритм будет строиться на основе реализации при помощи цикла из предыдущего пункта. Таким образом должны быть синхронизированы следующие операции, связанные с общими данными:

- Добавление и считывание данных из стека вершин.
- Обнуление значения вершины, запись суммы в значение вершины.
- Инкрементация и декрементация количества работающих потоков, проверка на количество работающих потоков.

Данные операции могут быть синхронизированы при помощи трех мьютексов.

В связи с особенностью библиотеки pthread необходимо разработать структуру, содержащую указатели на все необходимые общие данные. Эта структура передается как единственный аргумент каждому из потоков. Кроме того необходимо создать статическую функцию, которая будет вызываться в потоках:

```
1  < ... >
2
3  class node {
4      private:
5          < ... >
6
7          struct parg {
8              std::mutex* stackNodesMutex;
9              std::mutex* valuesMutex;
10 #ifdef POOL_ENABLED
11             std::mutex* threadsMutex;
12             const uint8_t* threadsLimit;
13             uint8_t* threadsCount;
14 #endif
15             std::stack<std::pair<tree::node*, tree::node*>>* stackNodes;
16         };
17
18         static void* getPthreadCyclicSumSubNodeHandler(void* argument);
19
20 < ... >
```

Максимальное количество потоков выбивается равным количеству доступных CPU при помощи функции std::thread::hardware_concurrency() для обеспечения наилучшей производительности [1] Многопоточная реализация алгоритма:

```
1  < ... >
2
3  tree::node tree::node::getPthreadCyclicSumSubNode() const {
4      tree::node result = tree::node(*this);
5
6  #ifdef POOL_ENABLED
```

```
7       std::mutex threadsMutex;
8       uint8_t threadsLimit = (uint8_t) std::thread::hardware_concurrency();
9       uint8_t threadsCount = 0;
10  #endif
11
12      std::mutex valuesMutex;
13      std::mutex stackNodesMutex;
14      std::stack<std::pair<tree::node*, tree::node*>> stackNodes;
15      stackNodes.push(std::pair<tree::node*, tree::node*>(nullptr, &result));
16
17      struct tree::node::parg pargument;
18      pargument.valuesMutex = &valuesMutex;
19      pargument.stackNodesMutex = &stackNodesMutex;
20
21  #ifdef POOL_ENABLED
22      pargument.threadsMutex = &threadsMutex;
23      pargument.threadsLimit = &threadsLimit;
24      pargument.threadsCount = &threadsCount;
25  #endif
26
27      pargument.stackNodes = &stackNodes;
28
29      pthread_t thread;
30      pthread_create(&thread, nullptr, &tree::node::getPthreadCyclicSumSubNodeHandler, (
        void*) &pargument);
31      pthread_join(thread, nullptr);
32      return result;
33  }
34
35  void* tree::node::getPthreadCyclicSumSubNodeHandler(void* argument) {
36      struct tree::node::parg pargument = *((parg*) argument);
37
38  #ifdef POOL_ENABLED
39      pargument.threadsMutex->lock();
40      ++(*(pargument.threadsCount));
41      pargument.threadsMutex->unlock();
42  #endif
43
44      bool retry = true;
45
46      std::vector<pthread_t> threads;
47      while (retry) {
48          retry = false;
49
50          // Get last stack pair.
51
52          pargument.stackNodesMutex->lock();
53
54          if (pargument.stackNodes->empty()) {
55              pargument.stackNodesMutex->unlock();
56              continue;
57          }
58
59          std::pair<tree::node*, tree::node*> currentPair = pargument.stackNodes->top();
60          pargument.stackNodes->pop();
61
62          pargument.stackNodesMutex->unlock();
63
64          tree::node* parentNodePtr = currentPair.first;
65          tree::node* currentNodePtr = currentPair.second;
66
67          // Set values for current and parent nodes.
68
69          pargument.valuesMutex->lock();
70
71          if (parentNodePtr != nullptr) {
```

```
72              parentNodePtr->mValue += currentNodePtr->mValue;
73          }
74
75          currentNodePtr->mValue = 0;
76
77          pargument.valuesMutex->unlock();
78
79          if (currentNodePtr->mChilds.empty()) {
80              // Set new task for this thread.
81              retry = true;
82              continue;
83          }
84
85          // Push child to vector.
86
87          pargument.stackNodesMutex->lock();
88
89          for (auto currentChild = currentNodePtr->mChilds.end() - 1; currentChild >=
    currentNodePtr->mChilds.begin(); --currentChild) {
90              pargument.stackNodes->push(std::pair<tree::node*, tree::node*>(currentNodePtr
    , &(*currentChild)));
91          }
92
93          pargument.stackNodesMutex->unlock();
94
95          // Create N-1 new threads to resolve task.
96          for (uint32_t threadIndex = 0; threadIndex < currentNodePtr->mChilds.size() - 1;
    ++threadIndex) {
97  #ifdef POOL_ENABLED
98              pargument.threadsMutex->lock();
99              if (*(pargument.threadsCount) >= (*(pargument.threadsLimit))) {
100                 pargument.threadsMutex->unlock();
101                 break;
102             }
103             pargument.threadsMutex->unlock();
104 #endif
105
106             pthread_t thread;
107             pthread_create(&thread, nullptr, &tree::node::
    getPthreadCyclicSumSubNodeHandler, (void*) &pargument);
108             threads.push_back(thread);
109         }
110
111         // Set new task for this thread.
112         retry = true;
113     }
114
115     // Wait other threads to continue.
116     for (const auto& currentThread : threads) {
117         pthread_join(currentThread, nullptr);
118     }
119
120 #ifdef POOL_ENABLED
121     pargument.threadsMutex->lock();
122     --(*(pargument.threadsCount));
123     pargument.threadsMutex->unlock();
124 #endif
125     return nullptr;
126 }
127
128 < ... >
```

1.5.5 Разработка многопроцессной реализации при помощи MPI

Message Passing Interface (MPI) – программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу [2] Основным средством коммуникации между процессами в MPI является передача сообщений друг другу.

В первую очередь MPI ориентирован на системы с распределенной памятью, то есть когда затраты на передачу данных велики, в то время как OpenMP ориентирован на системы с общей памятью (многоядерные с общим кешем). Обе технологии могут использоваться совместно, чтобы оптимально использовать в кластере многоядерные системы.

К сожалению, данная задача совсем не подходит для распараллеливания при помощи MPI [3]. Адресные пространства каждого из процессов MPI разделены, поэтому не получится обходить дерево подобным образом. MPI хорошо подходит для размещения или обмена простыми общими данными: целыми числами, числами с плавающей точкой, массивами чисел и др. Однако, поместить в разделяемую память подобный объект дерева не получится, поэтому распараллелить такой алгоритм черезвычайно сложно и скорее всего полученная реализация будет намного медленнее реализаций, описанных до этого. Для задач, задействующих большие объемы неструктурированной общей памяти, лучше подходит OpenMP.

В связи с этим, было принято решение распараллелить алгоритм расчета экспоненты для демонстрации особенностей работы с MPI:

```
< ... >

long double getExponentSimple(int32_t presision) {
    long double result = 0;
    for (int32_t indexStep = 0; indexStep <= presision; ++indexStep) {
        long double factorial = 1;
        if (indexStep > 1) {
            for (int32_t indexFact = 1; indexFact <= indexStep; ++indexFact) {
                factorial *= indexFact;
            }
        }

        long double value = 1 / factorial;
        result += value;
    }

    return result;
}

< ... >

#ifdef MPI_ENABLED

long double getExponentMpi(int32_t presision) {
    int32_t rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_Bcast(&presision, 1, MPI_INT, 0, MPI_COMM_WORLD);

    long double sum = 0;
    for (int32_t indexStep = rank; indexStep <= presision; indexStep += size) {
        long double factorial = 1;
        if (indexStep > 1) {
            for (int32_t indexFact = 1; indexFact <= indexStep; ++indexFact) {
                factorial *= indexFact;
            }
        }

        long double value = 1 / factorial;
        sum += value;
    }

    long double result;
    MPI_Reduce(&sum, &result, 1, MPI_LONG_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    return result;
```

```
}

#endif

< ... >
```

Заметим, что в реализацим подсчета экспоненты при помощи MPI на каждый процесс приходится в N раз меньше итераций цикла, где N это количество процессов.

Функции не задействующие MPI должны вызываться только на нулевом ранге (обеспечение единственности процесса), а задействующие должны вызываться для всех процессов:

```
< ... >

#ifdef MPI_ENABLED
#include <mpi.h>
#endif

< ... >

int main (int argc, char** argv) {
#ifdef MPI_ENABLED
    if (int32_t init = MPI_Init(&argc, &argv)) {
        MPI_Abort(MPI_COMM_WORLD, init);
        return 0x1;
    }

    int32_t rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    std::cout << "Rank " << rank << " of " << size << "." << std::endl;

    if (rank == 0) {
#endif

    long double result = getExponentSimple(100);
    std::cout << "Simple result: " << result << std::endl;

#ifdef MPI_ENABLED
    }

    long double result = getExponentMpi(100);

    if (rank == 0) {
        std::cout << "MPI result: " << result << std::endl;
    }

    MPI_Finalize();
#endif

    return 0x0;
}

< ... >
```

В результате работы было выведено:

```
Rank 0 of 6.
Rank 1 of 6.
Rank 2 of 6.
Rank 4 of 6.
Rank 5 of 6.
Simple result: 2.71828
Rank 3 of 6.
MPI result: 2.71828
```

Таким образом, задача была распараллелена на шесть процессов, в результате чего был получен правильный результат вычисления экспоненты.

1.5.6 Тестирование

Тестирование производительности MPI

Для тестирования производительности MPI на примере функции вычисления экспоненты разработаем тест, который замеряет время выполнения с высокой точностью через функции класса std::chrono::high_resolution_c

```
< ... >

static const int32_t EXPONENT_PRESISION = 10000;
static const int32_t EXPONENT_PERFORMANSE_RETRIES = 2;

< ... >

void testMpiPerformance() {
    int32_t rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    std::chrono::system_clock::time_point timeBefore, timeAfter;
    long double result;

    if (rank == 0) {
        std::cout << "treetests testMpiPerformance" << std::endl;

        timeBefore = std::chrono::high_resolution_clock::now();
        for (uint32_t retryIndex = 0; retryIndex < EXPONENT_PERFORMANSE_RETRIES; ++retryIndex) {
            result = getExponentSimple(EXPONENT_PRESISION);
        }
        timeAfter = std::chrono::high_resolution_clock::now();

        double resultTime = std::chrono::duration_cast<std::chrono::duration<double>>(timeAfter - timeBefore).count() * 1000;
        std::cout << "Simple exponent algorithm duration: " << resultTime << " milliseconds." << std::endl;

        auto presision = std::cout.precision();
        std::cout.precision(30);
        std::cout << "Simple exponent algorithm result: " << result << std::endl;
        std::cout.precision(presision);

        timeBefore = std::chrono::high_resolution_clock::now();
    }

    for (uint32_t retryIndex = 0; retryIndex < EXPONENT_PERFORMANSE_RETRIES; ++retryIndex) {
        result = getExponentMpi(EXPONENT_PRESISION);
    }

    if (rank == 0) {
        timeAfter = std::chrono::high_resolution_clock::now();

        double resultTime = std::chrono::duration_cast<std::chrono::duration<double>>(timeAfter - timeBefore).count() * 1000;
        std::cout << "MPI exponent algorithm duration: " << resultTime << " milliseconds." << std::endl;

        double presision = std::cout.precision();
        std::cout.precision(30);
        std::cout << "MPI exponent algorithm result: " << result << std::endl;
        std::cout.precision(presision);
    }
}

< ... >
```

Результаты тестирования производительности вычисления экспоненты:

№	precision	iterations	Simple, ms	MPI, ms
1	10	200	0.042692	152.628
2	100	20	1.21271	8.19007
3	5000	10	7790.65	1656.67
4	10000	2	10197.7	1923.33

Как и ожидалось, с увеличением сложности вычисления экспоненты (увеличением точности) распараллеливание MPI дает все большее преимущество по сравнению с простым алгоритмом.

Для подсчета статистики разработаем скрипт на языке python, который принимает путь к файлу с выборкой как аргумент командной строки и выводит статистические данные, такие как мат ожидание, дисперсия, радиус и интервал:

```
import numpy
import scipy.stats
import math
import sys

if __name__ == '__main__':
    filename = sys.argv[1]
    descriptor = open(filename, "r")

    sample = []
    for line in descriptor:
        if line:
            sample.append(float(line.strip()))

    descriptor.close()

    length = len(sample)
    if length < 2:
        sys.exit()

    mean = numpy.mean(sample)
    summary = numpy.sum(sample)

    dispersion = 0
    for element in sample:
        dispersion += (element - mean) ** 2;

    dispersion /= length - 1

    radius = scipy.stats.t.ppf((1 + 0.9) / 2, length - 1) * scipy.stats.sem(sample) /
    math.sqrt(length)

    print("Count - %d.\nTotal - %f ms.\nMean - %f ms.\nDispersion - %f.\nRadius - %f ms.\
    nInterval - [%f ms, %f ms]." % (length, summary, mean, dispersion, radius, mean -
    radius, mean + radius))
```

Вычислим статистические данные на наборе значений:

	precision	iterations	mean, ms	dispersion	radius, ms	interval, ms
Simple	10	200	0.000255	0.000000	0.000000	[0.000255, 0.000255]
MPI	10	200	0.113854	0.477850	0.005712	[0.108142, 0.119566]
Simple	100	20	0.019282	0.000000	0.000011	[0.019272, 0.019293]
MPI	100	20	0.567151	5.676893	0.205994	[0.361158, 0.773145]
Simple	5000	10	699.0373	3307.407968	10.542245	[688.495055, 709.579545]
MPI	5000	10	156.2293	1221.598	6.40698	[149.822320, 162.636280]
Simple	10000	2	4075.785000	4458.512450	210.791203	[3864.993797, 4286.576203]
MPI	10000	2	886.405500	10.511112	10.234858	[876.170642, 896.640358]

Вычислим зависимость от количества процессов MPI для точности 4000 и числа измерений 100 (вычисление производится на процессоре с 6 ядрами):

process count	total, ms	mean, ms	dispersion	radius, ms	interval, ms
1	29908.916	299.08916	41.855861	0.107421	[298.981739, 299.196581]
2	15151.683	151.51683	37.224975	0.101304	[151.415526, 151.618134]
4	8441.8148	84.418148	86.420509	0.154354	[84.263794, 84.572502]
6	7938.9617	79.38961	330.026356	0.301637	[79.087980, 79.691254]
8	8681.0363	86.810363	152.559181	0.205083	[86.605280, 87.015446]
10	7435.9975	74.359975	103.587356	0.168991	[74.190984, 74.528966]
12	7653.9843	76.539843	184.187987	0.225341	[76.314502, 76.765184]

Построим график зависимости времени выполнения (вертикальная ось в секундах) от количества процессов(горизонтальная ось):

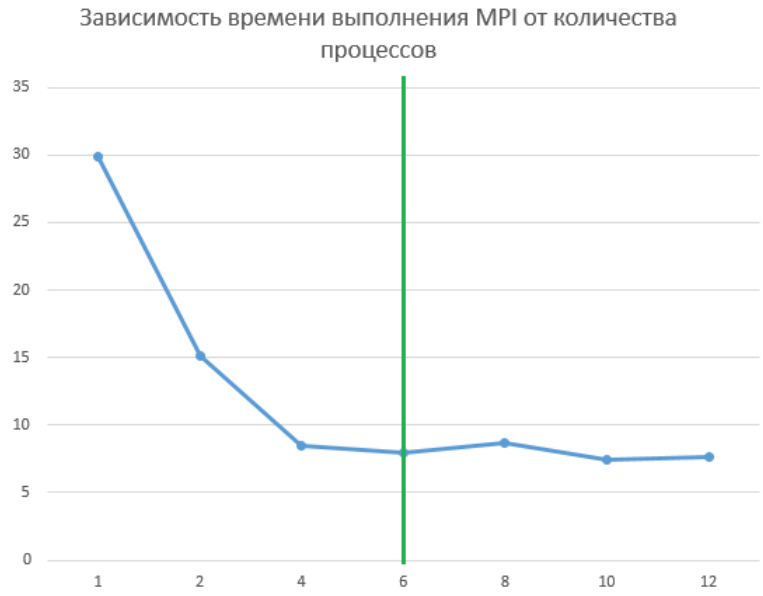

Рис. 1.3: Зависимость времени выполнения MPI от количества процессов

Зеленым отмечено количество ядер в текущем процессоре. Можно заметить, что наиболее значимый выигрыш в производительности наблюдается при переходе от одного процесса к двум (практически в два раза). Также неплохой выигрыш наблюдается при переходе от двух процессов к четырем. Последующее наращивание количества процессов не сильно улучшает производительность.

Тестирование алгоритмов на дереве

Тестирование алгоритмов на дереве (рекурсивного, последовательного и многопоточного) производится на трех деревьях с различным числом вершин:

- Маленькое дерево – 36 вершин.
- Среднее дерево – 7014 вершин.
- Большое дерево – 598528 вершин.

Первый тест основывается на тезисе, что результат всех трех алгоритмов на одном и том же дереве должен быть одинаковым:

```
< ... >

void testSmallTreeSingleEquality() {
    std::cout << "treetests testSmallTreeSingleEquality" << std::endl;

    tree testTree = parseTreeFromResourceFile(SMALL_TREE_FILENAME);
    std::cout << "Parsing finished." << std::endl;
```

```
8
9       // Simple recursive algorithm.
10
11      std::string simpleRecursiveStringTree = testTree.getSimpleRecursiveSumSubTree().
        toString();
12
13      // Simple cyclic algorithm.
14
15      std::string simpleCyclicStringTree = testTree.getSimpleCyclicSumSubTree().toString();
16
17      // Pthread cyclic algorithm.
18
19      std::string pthreadCyclicStringTree = testTree.getPthreadCyclicSumSubTree().toString
        ();
20
21      if (simpleRecursiveStringTree != simpleCyclicStringTree || simpleRecursiveStringTree
        != pthreadCyclicStringTree) {
22          std::cout << "%TEST_FAILED% time=0 testname=testSmallTreeSingleEquality (
        treetests) message=("
23                  << std::to_string(simpleRecursiveStringTree == simpleCyclicStringTree)
        << ", " << std::to_string(simpleRecursiveStringTree == pthreadCyclicStringTree)
24                  << ")" << std::endl;
25      }
26  }
27
28  < ... >
```

Подобный тест повторяется для всех трех тестовых деревьев.

Второй тест основывается на тезисе, что если все три алгоритма повторить дважды (или N раз) на одном и том же дереве, то должен получиться одинаковый результат:

```
1   < ... >
2
3   void testSmallTreeDoubleEquality() {
4       std::cout << "treetests testSmallTreeDoubleEquality" << std::endl;
5
6       tree testTree = parseTreeFromResourceFile(SMALL_TREE_FILENAME);
7       std::cout << "Parsing finished." << std::endl;
8
9       // Simple recursive algorithm.
10
11      std::string simpleRecursiveStringTree = testTree.getSimpleRecursiveSumSubTree().
        getSimpleRecursiveSumSubTree().toString();
12
13      // Simple cyclic algorithm.
14
15      std::string simpleCyclicStringTree = testTree.getSimpleCyclicSumSubTree().
        getSimpleCyclicSumSubTree().toString();
16
17      // Pthread cyclic algorithm.
18
19      std::string pthreadCyclicStringTree = testTree.getPthreadCyclicSumSubTree().
        getPthreadCyclicSumSubTree().toString();
20
21      if (simpleRecursiveStringTree != simpleCyclicStringTree || simpleRecursiveStringTree
        != pthreadCyclicStringTree) {
22          std::cout << "%TEST_FAILED% time=0 testname=testSmallTreeDoubleEquality (
        treetests) message=("
23                  << std::to_string(simpleRecursiveStringTree == simpleCyclicStringTree)
        << ", " << std::to_string(simpleRecursiveStringTree == pthreadCyclicStringTree)
24                  << ")" << std::endl;
25      }
26  }
27
28  < ... >
```

Подобный тест также повторяется для всех трех тестовых деревьев.

Тестирование производительности также производится замерами времени выполнения при нескольких итерациях:

```
< ... >

void testSmallTreePerformance() {
    std::cout << "treetests testSmallTreePerformance" << std::endl;

    tree testTree = parseTreeFromResourceFile(SMALL_TREE_FILENAME);
    std::cout << "Parsing finished." << std::endl;

    // Simple recursive algorithm.

    auto timeBefore = std::chrono::high_resolution_clock::now();
    for (uint32_t retryIndex = 0; retryIndex < SMALL_TREE_PERFORMANSE_RETRIES; ++
    retryIndex) {
        testTree.getSimpleRecursiveSumSubTree();
    }
    auto timeAfter = std::chrono::high_resolution_clock::now();

    double resultTime = std::chrono::duration_cast<std::chrono::duration<double>>(
    timeAfter - timeBefore).count() * 1000;
    std::cout << "Simple recursive algorithm duration: " << resultTime << " milliseconds.
    " << std::endl;

    // Simple cyclic algorithm.

    timeBefore = std::chrono::high_resolution_clock::now();
    for (uint32_t retryIndex = 0; retryIndex < SMALL_TREE_PERFORMANSE_RETRIES; ++
    retryIndex) {
        testTree.getSimpleCyclicSumSubTree();
    }
    timeAfter = std::chrono::high_resolution_clock::now();

    resultTime = std::chrono::duration_cast<std::chrono::duration<double>>(timeAfter -
    timeBefore).count() * 1000;
    std::cout << "Simple cyclic algorithm duration: " << resultTime << " milliseconds."
    << std::endl;

    // Pthread cyclic algorithm.

    timeBefore = std::chrono::high_resolution_clock::now();
    for (uint32_t retryIndex = 0; retryIndex < SMALL_TREE_PERFORMANSE_RETRIES; ++
    retryIndex) {
        testTree.getPthreadCyclicSumSubTree();
    }
    timeAfter = std::chrono::high_resolution_clock::now();

    resultTime = std::chrono::duration_cast<std::chrono::duration<double>>(timeAfter -
    timeBefore).count() * 1000;
    std::cout << "Pthread cyclic algorithm duration: " << resultTime << " milliseconds."
    << std::endl;
}

< ... >
```

Результаты тестирования производительности:

№	nodes	iterations	Recursive, ms	Cyclic, ms	Pthread, ms
1	36	20000	720.38	454.585	13467.2
2	7014	200	2914.3	957.112	1742.87
3	598528	2	9870.23	809.911	1182.03

Таким образом, рекурсивная реализация ожидаемо является самой медленной из трех.

Распараллеленная реализация оказалась медленней реализации через цикл. Это объясняется тем, что задача выполняемая на каждой вершине слишком простая (сумма двух чисел), а на контекстные переключения

уходит больше времени, чем выигрывается.

Вычислим статистические данные на наборе значений:

	nodes	iterations	mean, ms	dispersion	radius, ms	interval, ms
Recursive	36	20000	0.035285	0.002819	0.000004	[0.035281, 0.035290]
Cyclic	36	20000	0.024768	0.005636	0.000006	[0.024761, 0.024774]
Pthread	36	20000	0.618434	0.798031	0.000073	[0.618360, 0.618507]
Recursive	7014	200	13.127591	1.725761	0.010855	[13.116737, 13.138446]
Cyclic	7014	200	4.104959	0.004363	0.000546	[4.104413, 4.105505]
Pthread	7014	200	8.087841	6.705221	0.021396	[8.066445, 8.109236]
Recursive	598528	2	4794.750000	9253.440800	303.675053	[4491.074947, 5098.425053]
Cyclic	598528	2	399.068000	536.543282	73.123988	[325.944012, 472.191988]
Pthread	598528	2	585.838500	4573.791724	213.498920	[372.339580, 799.337420]

Высчитывать зависимость от количества потоков бессмысленно, так как количество потоков с такой простой задачей никак не повлияет на результат.

Усложним задачу для каждой вершины добавив задержку во все реализации алгоритма:

```
< ... >

void delay() {
    double temp = 1;
    for (uint32_t delayIndex = 1; delayIndex < 1000; ++delayIndex) {
        if (delayIndex % 2 == 0) {
            temp *= 10;
        } else {
            temp /= 10;
        }
    }
}

< ... >

tree::node tree::node::getSimpleRecursiveSumSubNode() const {
    < ... >

    for (const auto& currentChild : mChilds) {
        delay(); // <----------

        sum += currentChild.mValue;

        < ... >
    }

    < ... >
}

< ... >

tree::node tree::node::getSimpleCyclicSumSubNode() const {
    < ... >

    while (!stackNodes.empty()) {
        < ... >

        if (parentNodePtr != nullptr) {
            delay(); // <----------

            parentNodePtr->mValue += currentNodePtr->mValue;
        }

        < ... >
    }

```

```
47      < ... >
48  }
49
50  < ... >
51
52  void* tree::node::getPthreadCyclicSumSubNodeHandler(void* argument) {
53      < ... >
54
55      while (retry) {
56          < ... >
57
58          // Set values for current and parent nodes.
59
60          delay(); // <----------
61
62          pargument.valuesMutex->lock();
63
64          if (parentNodePtr != nullptr) {
65              parentNodePtr->mValue += currentNodePtr->mValue;
66          }
67
68          currentNodePtr->mValue = 0;
69
70          pargument.valuesMutex->unlock();
71
72          < ... >
73
74      < ... >
75  }
76
77  < ... >
```

Для многопоточной реализации важно добавить задержку в область, не синхронизируемую мьютексами, так как выигрыш от многопоточной реализации достигается только вычислениями, которые можно выполнять параллельно без ущерба для целостности.

Результаты тестирования производительности с добавлением искусственной задержки:

№	nodes	iterations	Recursive, ms	Cyclic, ms	Pthread, ms
1	36	20000	3745.25	3501.24	14416.3
2	7014	200	8955.39	7005.93	2999.5
3	598528	2	15390.8	6068.39	2551.58

Рекурсивная реализация все еще наиболее медленная, зато многопоточная стала намного быстрее относительно других. Для каждого потока появилась реальная задача, которую можно выполнять параллельно, поэтому контекстные переключения уже не так значительно влияют на общую производительность.

Вычислим статистические данные на наборе значений:

	nodes	iterations	mean, ms	dispersion	radius, ms	interval, ms
Recursive	36	20000	0.187652	0.004227	0.000005	[0.187647, 0.187658]
Cyclic	36	20000	0.174525	0.000037	0.000001	[0.174524, 0.174525]
Pthread	36	20000	0.527426	0.380192	0.000051	[0.527375, 0.527477]
Recursive	7014	200	44.077959	4.952748	0.018389	[44.059570, 44.096347]
Cyclic	7014	200	34.709729	0.062914	0.002073	[34.707656, 34.711801]
Pthread	7014	200	12.814718	8.034588	0.023421	[12.791297, 12.838139]
Recursive	598528	2	7347.590	9549.620	308.496709	[7039.093291, 7656.086709]
Cyclic	598528	2	2988.885	456.926450	67.480865	[2921.404135, 3056.365865]
Pthread	598528	2	1053.120000	2000.913800	141.212025	[911.907975, 1194.332025]

Можно заметить, что у многопоточного алгоритма дисперсия существенно выше чем у других алгоритмов, даже при том, что среднее значение наименьшее.

Вычислим зависимость от количества потоков Ptherad алгоритма для дерева с количеством вершин 598528 и числа измерений 30 (вычисление производится на процессоре с 6 ядрами):

threads count	total, ms	mean, ms	dispersion	radius, ms	interval, ms
1	92572.1	3085.736667	669.155106	1.465102	[3084.271564, 3087.201769]
2	32923.23	1097.441	18467.619761	7.696797	[1089.744203, 1105.137797]
4	31187.036	1039.567867	2430.774867	2.792396	[1036.775471, 1042.360262]
6	32249.43	1074.981	730.856823	1.531161	[1073.449839, 1076.512161]
8	35215.12	1173.837333	4114.585951	3.633019	[1170.204314, 1177.470352]
10	34528.08	1150.936	2198.330673	2.655529	[1148.280471, 1153.591529]
12	33508.83	1116.961	801.721989	1.603675	[1115.357325, 1118.564675]

Построим график зависимости времени выполнения (вертикальная ось в секундах) от количества потоков (горизонтальная ось):

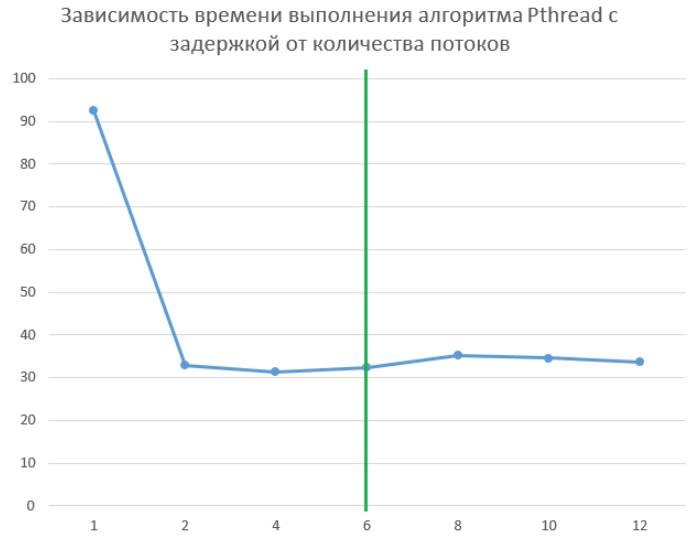

Рис. 1.4: Зависимость времени выполнения алгоритма Pthread от количества потоков

Уже при двух потоках время выполнения практически оптимальное, однако дисперсия на двух потоках очень большая и постепенно уменьшается с наращиванием количества потоков до шести, что хорошо иллюстрируется следующим графиком:

Рис. 1.5: Зависимость дисперсии алгоритма Pthread от количества потоков

1.6 Вывод

В ходе работы было разработано несколько алгоритмов для нахождения дерева сумм: рекурсивный, циклический и многопоточный (с ограничением количества потоков и без). Многопоточный алгоритм без ограничения количества потоков не рассматривался в работе ввиду общей неэффективности и непрохождении тестов на больших деревьях. Кроме того, был разработан демонстрационный MPI проект по поиску экспоненты с заданной точностью и его последующем тестировании.

Рекурсивная реализация алгоритмов плоха не только тем, что стек может переполниться, но и показывает в целом меньшую производительность, чем такой же алгоритм, реализованный через цикл.

Многопоточная реализация не является общим решением: приходится жертвовать производительностью на небольших данных, кроме того потоки занимают большое количество памяти. Многопоточную реализацию имеет смысл применять, когда для каждого потока есть трудоемкая подзадача, которую можно выполнять параллельно, чтобы контекстные переключения не так значительно влияли на общую производительность. Кроме того, имеет смысл ограничить количество потоков равным количеству доступных CPU для наилучшей производительности.

Для задач с необходимостью доступа к сложноструктурированной общей памяти MPI не очень подходит. Это объясняется тем, что деревья, графы и подобные структуры в C++ весьма тяжело положить в разделяемую память, поэтому придется придумывать сложные неэффективные способы распараллеливания с передачей стандартных сообщений MPI. Однако, MPI отлично подходит для распараллеливания математических алгоритмов по типу вычисления экспоненты или числа пи, кроме того, MPI широко используется для распараллеливания программ для кластеров и суперкомпьютеров.

Наиболее значимый выигрыш в производительности вычисления экспоненты MPI наблюдается при переходе от одного процесса к двум (практически в два раза). Также неплохой выигрыш наблюдается при переходе от двух процессов к четырем. Последующее наращивание количества процессов не сильно улучшает производительность.

Литература

[1] Setting Ideal size of Thread PoolElectronic resource, Stackoverflow. — URL: https://stackoverflow.com/questions/16128436/setting-ideal-size-of-thread-pool (online; accessed: 29.04.2018).

[2] Лекция 5. Технологии параллельного программирования. Message Passing Interface (MPI) [Электронный ресурс], Parallel.ru. — URL: https://parallel.ru/vvv/mpi.html (дата обращения: 29.04.2018).

[3] Parallelize n-ary tree traversal using MPIElectronic resource, Stackoverflow. — URL: https://stackoverflow.com/questions/20378239/parallelize-n-ary-tree-traversal-using-mpi (online; accessed: 29.04.2018).