Санкт-Петербургский государственный политехнический университет
Институт Информационных Технологий и Управления
Кафедра компьютерных систем и программных технологий

Отчет по лабораторным работам
по предмету "Параллельные вычисления"

Параллельная обработка Б-деревьев

Работу выполнил студент гр. 53501/3 _____ Мартынов С. А.

Работу принял преподаватель _____ Моисеев М.Ю.

Санкт-Петербург
2014

Оглавление

Введение

В последнее время тактовая частота центральных процессоров прекратила бурный рост, который наблюдался в последние десятилетия и производители сосредоточились на увеличении количества ядер. Это открыло новые возможности для разработчиков, т.к. позволило увеличить производительность программ не прибегая к механическому вертикальному масштабированию.

Воспользоваться преимуществами нескольких ядер относительно просто для серверных приложений, где каждый поток может независимо обрабатывать отдельный запрос от клиента, но этого значительно сложнее добиться для клиентских приложений, поскольку в этом случае обычно потребуется преобразовать код, интенсивно использующий вычисления и средства межпроцессной и межпоселковой синхронизации.

В данной работе рассмотрена работа со следующими инструментами:

1. Pthtreads;
2. OpenMP;
3. C++11;
4. Синхронизация средствами Posix.

В качестве задачи было принято решение реализовать поточную обработку B-дерева, часто применяемого в базах данных.

Исходный код всех представленных листингов доступен по адресу `https://github.com/SemenMartynov/SPbPU_ParallelProgramming`.

Реализация Б-дерева

Б-дерево - это сбалансированное, сильно ветвистое дерево (как правило, во внешней памяти). В нашей реализации, оно состоит из двух классов BTree и BTreeNode. Заголовочные файлы этих классов представлены в листинге 1 и листинге 2. Реализацию можно изучить на гитхабе, ссылка была приведена во введении.

Листинг 1: заголовочный файл класса BTree

```
#ifndef TEST_BTREE_H_
#define TEST_BTREE_H_

#include "BTreeNode.h"

/**
 * BTree
 */
class BTree {
public:
  /**
   * @brief Constructor (Initializes tree as empty)
   *
   * @param t minimum degree
   */
  BTree(int t);

  /**
   * @brief Destructor
   */
  virtual ~BTree();
  BTree(BTree&&) = delete;
  BTree(const BTree&) = delete;
  BTree& operator=(BTree&&) = delete;
  BTree& operator=(const BTree&) = delete;

  /**
   * A function to traverse all nodes in a subtree
   *
```

```
30      * @return
31      */
32     std::string to_str();
33 
34     /**
35      * function to search a key in this tree
36      *
37      * @param key
38      * @return
39      */
40     bool exist(int key);
41 
42     /**
43      * The main function that inserts a new key in this B-Tree
44      *
45      * @param key
46      * @return
47      */
48     bool insert(int key);
49 
50     /**
51      * The main function that removes a new key in thie B-Tree
52      *
53      * @param key
54      * @return
55      */
56     bool remove(int key);
57 
58 private:
59     const int t; /**< Minimum degree*/
60     BTreeNode *root; /**< Pointer to root node*/
61 };
62 
63 #endif /* TEST_BTREE_H_ */
```

Листинг 2: заголовочный файл класса BTreeNode

```
1 #ifndef TEST_BTREENODE_H_
2 #define TEST_BTREENODE_H_
3 
4 /**
5  * BTreeNode
6  */
7 class BTreeNode {
8 public:
```

```
9   /**
10   * @brief Constructor
11   *
12   * @param t minimum degree
13   * @param leaf
14   */
15  BTreeNode(int t, bool leaf);
16
17  /**
18   * @brief Destructor
19   */
20  virtual ~BTreeNode();
21  //virtual ~BTreeNode() = default;
22  BTreeNode(BTreeNode&&) = delete;
23  BTreeNode(const BTreeNode&) = delete;
24  BTreeNode& operator=(BTreeNode&&) = delete;
25  BTreeNode& operator=(const BTreeNode&) = delete;
26
27  /**
28   * A function to traverse all nodes in a subtree rooted with this node
29   *
30   * @return
31   */
32  std::string to_str();
33
34  /**
35   * A function to search a key in subtree rooted with this node.
36   *
37   * @param key
38   * @return NULL if key is not present
39   */
40  bool exist(int key);
41
42  /**
43   * A function that returns the index of the first key that is greater
44   * or equal to key
45   *
46   * @param key
47   * @return
48   */
49  int findKey(int key);
50
51  /**
52   * A utility function to insert a new key in the subtree rooted with
53   * this node. The assumption is, the node must be non-full when this
```

```
54      * function is called
55      *
56      * @param key
57      */
58     void insertNonFull(int key);
59
60     /**
61      * A utility function to split the child y of this node. i is index
62      * of y in child array C[].  The Child y must be full when this
63      * function is called
64      *
65      * @param i
66      * @param y
67      */
68     void splitChild(int i, BTreeNode *y);
69
70     /**
71      * A wrapper function to remove the key key in subtree rooted with
72      * this node.
73      *
74      * @param key
75      */
76     void remove(int key);
77
78     /**
79      * A function to remove the key present in idx-th position in
80      * this node which is a leaf
81      *
82      * @param idx
83      */
84     void removeFromLeaf(int idx);
85
86     /**
87      * A function to remove the key present in idx-th position in
88      * this node which is a non-leaf node
89      *
90      * @param idx
91      */
92     void removeFromNonLeaf(int idx);
93
94     /**
95      * A function to get the predecessor of the key- where the key
96      * is present in the idx-th position in the node
97      *
98      * @param idx
```

```
99      * @return
100     */
101    int getPred(int idx);
102
103    /**
104     * A function to get the successor of the key- where the key
105     * is present in the idx-th position in the node
106     *
107     * @param idx
108     * @return
109     */
110    int getSucc(int idx);
111
112    /**
113     * A function to fill up the child node present in the idx-th
114     * position in the C[] array if that child has less than t-1 keys
115     * @param idx
116     */
117    void fill(int idx);
118
119    /**
120     * A function to borrow a key from the C[idx-1]-th node and place
121     * it in C[idx]th node
122     *
123     * @param idx
124     */
125    void borrowFromPrev(int idx);
126
127    /**
128     * A function to borrow a key from the C[idx+1]-th node and place it
129     * in C[idx]th node
130     *
131     * @param idx
132     */
133    void borrowFromNext(int idx);
134
135    /**
136     * A function to merge idx-th child of the node with (idx+1)th child of
137     * the node
138     *
139     * @param idx
140     */
141    void merge(int idx);
142
143    /**
```

```
144     * Make BTree friend of this so that we can access private members of
145     * this class in BTree functions
146     */
147   friend class BTree;
148
149 private:
150   const int t; /**< Minimum degree (defines the range for number of keys)*/
151   BTreeNode** children; /**< An array of child pointers*/
152   int* keys; /**< An array of keys*/
153   int keysnum; /**< Current number of keys*/
154   bool leaf; /**< Is true when node is leaf. Otherwise false*/
155 };
156
157 #endif /* TEST_BTREENODE_H_ */
```

Корректность кода проверялась при помощи Google test framework (результаты тестированя представлены в листинге 3, код самих тестов находится на гитхабе) и valgrind (смотри изображение 1).

Листинг 3: Лог тестирование программы при помощи Google test framework

```
1  Running tests...
2  [==========] Running 8 tests from 3 test cases.
3  [----------] Global test environment set-up.
4  [----------] 4 tests from BTree3Test
5  [ RUN      ] BTree3Test.Add
6  [       OK ] BTree3Test.Add (0 ms)
7  [ RUN      ] BTree3Test.Remove
8  [       OK ] BTree3Test.Remove (1 ms)
9  [ RUN      ] BTree3Test.EmptyRemove
10 [       OK ] BTree3Test.EmptyRemove (0 ms)
11 [ RUN      ] BTree3Test.BigTest
12 [       OK ] BTree3Test.BigTest (0 ms)
13 [----------] 4 tests from BTree3Test (1 ms total)
14
15 [----------] 3 tests from BTree4Test
16 [ RUN      ] BTree4Test.BigTest
17 [       OK ] BTree4Test.BigTest (0 ms)
18 [ RUN      ] BTree4Test.EmptyRemove
19 [       OK ] BTree4Test.EmptyRemove (0 ms)
20 [ RUN      ] BTree4Test.RandomTest
21 [       OK ] BTree4Test.RandomTest (1 ms)
22 [----------] 3 tests from BTree4Test (1 ms total)
23
24 [----------] 1 test from BTree2Test
25 [ RUN      ] BTree2Test.BigTest
```

```
[       OK ] BTree2Test.BigTest (0 ms)
[----------] 1 test from BTree2Test (0 ms total)

[----------] Global test environment tear-down
[==========] 8 tests from 3 test cases ran. (2 ms total)
[  PASSED  ] 8 tests.
```

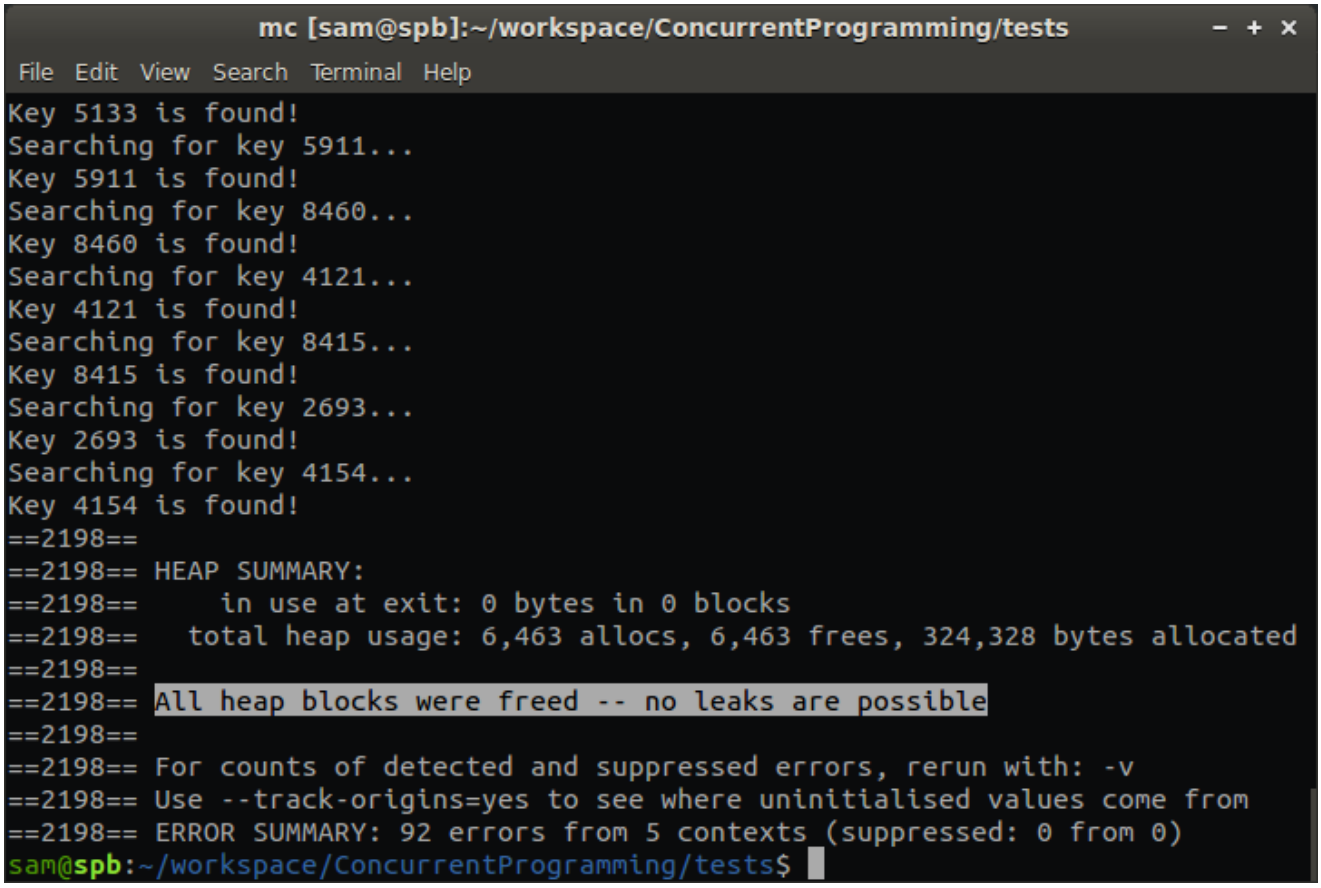

Рис. 1: Поиск утечек памяти при помощи valgrind

Однопоточный алгоритм

В самом простом виде, алгоритм представляет из себя строгую последовательность команд. В реализацию специально добавлены задержки по времени (определяемые при помощи генератора случайных чисел), имитирующие работу. Количество задач (т.е. количество запросов к хранимой структуре) определяется передаваемым параметром. Листинг 4 содержит код однопотопной реализации программы, работающей с б-деревом.

Листинг 4: Однопоточная реализация

```
1  #include <iostream>
2  #include <numeric>
3  #include <algorithm>
4  #include <sstream>
5
6  #include <chrono>
7  #include <thread>
8
9  #include "btree/BTree.h"
10
11 #define keys_number 1000
12
13 int main(int argc, char* argv[]) {
14   if (argc != 2) {
15     std::cout << "Too few parameters!" << std::endl;
16     exit(1);
17   }
18   std::istringstream ss(argv[1]);
19   int task_size;
20   if (!(ss >> task_size)) {
21     std::cout << "Invalid number " << argv[1] << std::endl;
22     exit(1);
23   }
24
25   std::default_random_engine generator;
26   std::uniform_int_distribution<int> distribution(0, keys_number);
27   BTree tree4(4); // B-Tree with minimum degree 4
28
```

```
29    std::vector<int> keys(keys_number); // vector with keys_number ints.
30    std::iota(keys.begin(), keys.end(), 0); // Fill with 0, 1, ..., 9999.
31
32    std::random_shuffle(std::begin(keys), std::end(keys)); // the first
          shuffle
33    std::for_each(keys.begin(), keys.end(), [&](int key) {
34      std::cout << "add" << std::endl;
35      tree4.insert(key); // add new key
36        std::this_thread::sleep_for(
37            std::chrono::milliseconds(
38                distribution(generator) / 16));//sleep
39      });
40
41    // The cycle will be performed as many times
42    // as specified in the variable task_size.
43    for (int i = 0; i != task_size; ++i) {
44      int key = distribution(generator);
45      std::cout << "Searching for key " << key << "..." << std::endl;
46      if (tree4.exist(key))
47        std::cout << "Key " << key << " is found!" << std::endl;
48      std::this_thread::sleep_for(
49          std::chrono::milliseconds(distribution(generator) / 2)); //sleep
50    }
51
52    std::random_shuffle(std::begin(keys), std::end(keys)); // the second
          shuffle
53    std::for_each(keys.begin(), keys.end(), [&](int key) { // remove
54          tree4.remove(key);
55          std::this_thread::sleep_for(
56              std::chrono::milliseconds(
57                  distribution(generator) / 16));//sleep
58        });
59
60    exit(0);
61  }
```

Многопоточные программирование с использованием Pthreads

В той или иной форме, библиотека Pthread представлена на всех современных операционных системах. Она предоставляет низкоуровневый механизм управления процессами что даёт большую производительность но и требует больших трудозатрат. В листинге 5 один поток пополняет дерево, один удаляет элементы, и ещё несколько осуществляют поиск по дереву. Поток удаления не начнёт свою работу, пока поток добавления не завершит.

Листинг 5: Работа с B-деревом при помощи Pthreads

```
1  #include <iostream>
2  #include <numeric>
3  #include <algorithm>
4  #include <sstream>
5
6  #include <chrono>
7  #include <thread>
8
9  #include <pthread.h>
10 #include "btree/BTree.h"
11
12 #define keys_number 1000
13
14 pthread_rwlock_t rwlock;
15 BTree tree4(4); // B-Tree with minimum degree 4
16
17 std::vector<int> keys(keys_number); // vector with keys_number ints.
18
19 // random generator
20 std::default_random_engine generator;
21 std::uniform_int_distribution<int> distribution(0, keys_number);
22
23 void* check(void* param) {
24   int tasks = *(int*) param;
25
```

```
26    for (int i = 0; i != tasks; ++i) {
27      pthread_rwlock_rdlock(&rwlock);
28      int key = distribution(generator);
29      std::cout << "Searching for key " << key << "..." << std::endl;
30      if (tree4.exist(key)) {
31        std::cout << "Key " << key << " is found!" << std::endl;
32        std::cout.flush();
33      }
34      std::this_thread::sleep_for(
35          std::chrono::milliseconds(distribution(generator) / 2)); //sleep
36      pthread_rwlock_unlock(&rwlock);
37    }
38    return nullptr;
39  }
40
41  void* insert(void* param) {
42    // feel vector
43    std::for_each(keys.begin(), keys.end(), [&](int key) {
44      pthread_rwlock_wrlock(&rwlock);
45      tree4.insert(key);
46      std::this_thread::sleep_for(
47          std::chrono::milliseconds(distribution(generator) / 16)); //sleep
48        pthread_rwlock_unlock(&rwlock);
49      });
50
51    return nullptr;
52  }
53
54  int main(int argc, char* argv[]) {
55    if (argc != 2) {
56      std::cout << "Too few parameters!" << std::endl;
57      exit(1);
58    }
59    std::istringstream ss(argv[1]);
60    int task_size;
61    if (!(ss >> task_size)) {
62      std::cout << "Invalid number " << argv[1] << std::endl;
63      exit(1);
64    }
65
66    pthread_rwlock_init(&rwlock, nullptr);
67
68    std::iota(keys.begin(), keys.end(), 0); // Fill with 0, 1, ..., 9999.
69    std::random_shuffle(std::begin(keys), std::end(keys)); // the first
        shuffle
```

```
70
71    // Number of processor cores
72    unsigned concurentThreads = std::thread::hardware_concurrency() - 1;
73    if (concurentThreads < 1)
74      concurentThreads = 1;
75    // Threads
76    std::vector<pthread_t> threads(concurentThreads);
77    pthread_t inserter;
78    int tasks = task_size / concurentThreads;
79
80    // starts threads
81    pthread_create(&inserter, NULL, insert, NULL);
82    for (auto thread : threads)
83      pthread_create(&thread, NULL, check, &tasks);
84
85    // Wait for the filling of the vector, and then run the cleanup.
86    pthread_join(inserter, NULL);
87    std::for_each(keys.begin(), keys.end(), [&](int key) {
88      pthread_rwlock_wrlock(&rwlock);
89      tree4.remove(distribution(generator));
90      std::this_thread::sleep_for(
91          std::chrono::milliseconds(distribution(generator) / 16)); //sleep
92        pthread_rwlock_unlock(&rwlock);
93      });
94
95    // wait for all threads to complete.
96    for (auto thread : threads)
97      pthread_join(thread, NULL);
98
99    pthread_rwlock_destroy(&rwlock);
100   return 0;
101 }
```

Многопоточные программирование с использованием OpenMP

Наиболее удобным средством для быстрого распараллеливания уже написанного кода является OpenMP. С другой стороны, его сложнее отлаживать и контролировать. К тому же, не являясь частью языка или библиотеки, OpenMP не поддерживается некоторыми статическими анализаторами на этапе разработки, что тоже создаёт неудобства. На листинге 6 представлена реализация задачи построения B-дерева с использованием OpenMP.
Минусом библиотеки является отсутствие в её составе механизма, похожего на RW-Lock. Его можно выразить из других средств, но это уже является отходом от изначальной идеи удобства.

Листинг 6: Работа с B-деревом из OpenMP

```
1  #include <iostream>
2  #include <numeric>
3  #include <algorithm>
4  #include <sstream>
5
6  #include <chrono>
7  #include <thread>
8
9  #include <omp.h>
10 #include "btree/BTree.h"
11
12 #define keys_number 1000
13
14 int main(int argc, char* argv[]) {
15   if (argc != 2) {
16     std::cout << "Too few parameters!" << std::endl;
17     exit(1);
18   }
19   std::istringstream ss(argv[1]);
20   int task_size;
21   if (!(ss >> task_size)) {
22     std::cout << "Invalid number " << argv[1] << std::endl;
```

```
23      exit(1);
24    }
25
26    std::default_random_engine generator;
27    std::uniform_int_distribution<int> distribution(0, keys_number);
28    BTree tree4(4); // B-Tree with minimum degree 4
29
30    std::vector<int> keys(keys_number); // vector with keys_number ints.
31    std::iota(keys.begin(), keys.end(), 0); // Fill with 0, 1, ..., 9999.
32
33    int nthreads, tid;
34    /* Fork a team of threads with each thread having a private tid variable
          */
35  #pragma omp parallel private(tid)
36    {
37      /* Obtain and print thread id */
38      tid = omp_get_thread_num();
39      std::cout << "Thread " << tid << " active" << std::endl;
40
41      /* Only master thread does this */
42      if (tid == 0) {
43        std::cout << "Number of threads = " << omp_get_num_threads()
44            << std::endl;
45
46        std::random_shuffle(std::begin(keys), std::end(keys)); // the first
              shuffle
47        std::for_each(keys.begin(), keys.end(), [&](int key) { // add
48              tree4.insert(key);
49              std::this_thread::sleep_for(
50                  std::chrono::milliseconds(distribution(generator) / 16));//
                      sleep
51            });
52
53        // I would like to place delete-records-code here,
54        // but there is no RWLocak in OpenMP
55      } else { // In other parallel threads
56        // Get threads number
57        nthreads = omp_get_num_threads() - 1;
58        if (nthreads < 1)
59          nthreads = 1;
60  #pragma omp parallel for
61        for (int i = 0; i < task_size / nthreads; ++i) {
62          int key = distribution(generator);
63          std::cout << "#" << tid << " Searching for key " << key << "..."
64              << std::endl;
```

```
65          if (tree4.exist(key))
66            std::cout << "Key " << key << " is found!" << std::endl;
67          std::this_thread::sleep_for(
68              std::chrono::milliseconds(distribution(generator) / 2));
69        }
70      }
71    } /* All threads join master thread and terminate */
72    std::random_shuffle(std::begin(keys), std::end(keys)); // the second
        shuffle
73    std::for_each(keys.begin(), keys.end(), [&](int key) { // remove
74          tree4.remove(key);
75          std::this_thread::sleep_for(
76              std::chrono::milliseconds(distribution(generator) / 16));//sleep
77        });
78
79    return 0;
80  }
```

Многопоточные программирование с использованием C++11

Представленный в 2011-м году стандарт языка C++ получил поддержку потоков. По сути, это более высокоуровневая абстракция над библиотекой Pthreads, но работать становится на много приятнее за счёт использования таких механизмов как лямбды. Код в итоге получается удобно читаемым и компактным. На листинге 7 представлена работа с C++11. Дополнительные механизмы синхронизации были перенесены из библиотеки boost в стандарт C++14.

Листинг 7: Потоки C++11

```
1  #include <iostream>
2  #include <algorithm>
3  #include <sstream>
4  #include <vector>
5  #include <future>
6
7  #include <chrono>
8  #include <thread>
9
10 #include "btree/BTree.h"
11
12 #define keys_number 1000
13
14 // std::shared_mutex will be part of the C++14 Standard Library
15 int main(int argc, char* argv[]) {
16   if (argc != 2) {
17     std::cout << "Too few parameters!" << std::endl;
18     exit(1);
19   }
20   std::istringstream ss(argv[1]);
21   int task_size;
22   if (!(ss >> task_size)) {
23     std::cout << "Invalid number " << argv[1] << std::endl;
24     exit(1);
```

```
25  }
26
27  std::default_random_engine generator;
28  std::uniform_int_distribution<int> distribution(0, keys_number);
29  BTree tree4(4); // B-Tree with minimum degree 4
30
31  std::vector<int> keys(keys_number); // vector with keys_number ints.
32  std::iota(keys.begin(), keys.end(), 0); // Fill with 0, 1, ..., 9999.
33
34  std::random_shuffle(std::begin(keys), std::end(keys)); // the first
        shuffle
35  std::for_each(keys.begin(), keys.end(), [&](int key) { // add
36        tree4.insert(key);
37        std::this_thread::sleep_for(
38            std::chrono::milliseconds(distribution(generator) / 16));//sleep
39      });
40
41  std::cout << "Main thread id: " << std::this_thread::get_id() << std::endl
        ;
42  std::vector<std::future<void>> futures;
43
44  for (int i = 0; i != task_size; ++i) {
45    auto fut = std::async(std::launch::async, [&]
46    {
47      int key = distribution(generator);
48      std::cout << "Searching for key " << key << "..." << std::endl;
49      if (tree4.exist(key))
50      std::cout << "Key " << key << " is found!" << std::endl;
51      std::this_thread::sleep_for(
52          std::chrono::milliseconds(distribution(generator) / 2)); //sleep
53      });
54    futures.push_back(std::move(fut));
55  }
56  std::for_each(futures.begin(), futures.end(), [](std::future<void> &fut)
57  {
58    fut.wait();
59  });
60
61  std::random_shuffle(std::begin(keys), std::end(keys)); // the second
        shuffle
62  std::for_each(keys.begin(), keys.end(), [&](int key) { // remove
63        tree4.remove(key);
64        std::this_thread::sleep_for(
65            std::chrono::milliseconds(distribution(generator) / 16));//sleep
66      });
```

```
67
68    return 0;
69 }
```

Многопроцессные приложения с POSIX синхронизацией

В этой программе используется семафор, вмещающий 4 (в зависимости от процессора) потребителя. При этом в программе создаётся 6 процессов (средствами вызова fork()) и основной процесс. Дочерние процессы проверяют наличие ключа в дереве, а родительский постепенно уничтожает эти ключи. Каждый дочерний процесс работает какое-то время, которое определяется генератором случайных чисел. Код, реализующий эту идею, представлен в листинге 8.

Листинг 8: Синхронизация средствами POSIX

```
1  #include <iostream>
2  #include <numeric>
3  #include <algorithm>
4  #include <sstream>
5  #include <vector>
6
7  #include <chrono>
8  #include <thread>
9
10 #include <semaphore.h>
11 #include <unistd.h>
12 #include <sys/wait.h>
13 #include <fcntl.h>
14 #include <sys/mman.h>
15
16 #include "btree/BTree.h"
17
18 #define keys_number 1000
19
20 // The number of child processes
21 const int thread_num = 6;
22
23 // How many requests each child process executes
24 int task_num = 20;
```

```
25
26 // The number of simultaneously active processes.
27 const int sem_num = std::thread::hardware_concurrency();
28
29 int main(int argc, char* argv[]) {
30   if (argc != 2) {
31     std::cout << "Too few parameters!" << std::endl;
32     exit(1);
33   }
34   std::istringstream ss(argv[1]);
35   int task_size;
36   if (!(ss >> task_size)) {
37     std::cout << "Invalid number " << argv[1] << std::endl;
38     exit(1);
39   }
40   task_num = task_size / thread_num;
41
42   //Actually I do not really care about these variables,
43   //but they are needed later.
44   pid_t wpid;
45   int status = 0;
46
47   // timer for random generator
48   typedef std::chrono::high_resolution_clock clock;
49   clock::time_point beginning = clock::now();
50
51   std::default_random_engine generator;
52   std::uniform_int_distribution<int> distribution(0, keys_number);
53   BTree tree4(4); // B-Tree with minimum degree 4
54
55   // Create and initialize semaphore
56   int shm;
57   sem_t * mutex;
58
59   if ((shm = shm_open("/myshm", O_CREAT | O_TRUNC | O_RDWR, 0666)) < 0) {
60     std::perror("shm_open");
61     exit(1);
62   }
63
64   if (ftruncate(shm, sizeof(sem_t)) < 0) {
65     std::perror("ftruncate");
66     exit(1);
67   }
68
69   if ((mutex = (sem_t*)mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE,
```

```
    MAP_SHARED,
70      shm, 0)) == MAP_FAILED) {
71    std::perror("mmap");
72    exit(1);
73  }
74
75  if (sem_init(mutex, 1, sem_num) < 0) {
76    std::perror("Semaphore initialization");
77    exit(0);
78  }
79  // forked child processes use the same semaphore
80
81  std::vector<int> keys(keys_number); // vector with keys_number ints.
82  std::iota(keys.begin(), keys.end(), 0); // Fill with 0, 1, ..., 9999.
83
84  std::random_shuffle(keys.begin(), keys.end()); // the first shuffle
85  for (auto key : keys) { // add
86    tree4.insert(key);
87    std::this_thread::sleep_for(
88        std::chrono::milliseconds(distribution(generator) / 16));
89  }
90
91  // Create and run children
92  for (int i = 0; i != thread_num; ++i) {
93    if (fork() == 0) { // child process
94      // obtain a seed from the timer
95      clock::duration seed = clock::now() - beginning;
96      generator.seed(static_cast<uint64_t>(seed.count()));
97      for (int j = 0; j != task_num; j++) {
98        sem_wait(mutex);
99        int key = distribution(generator);
100       std::cout << "Searching for key " << key << "..." << std::endl;
101       if (tree4.exist(key))
102         std::cout << "Key " << key << " is found!" << std::endl;
103       std::this_thread::sleep_for(
104           std::chrono::milliseconds(distribution(generator) / 2));
105       sem_post(mutex);
106     }
107     exit(0);
108   }
109 }
110
111 // The parent process waits for the semaphore capture on par with everyone
    .
112 std::random_shuffle(std::begin(keys), std::end(keys)); // the second
```

```
    shuffle
113   std::for_each(keys.begin(), keys.end(), [&](int key) { // remove
114         sem_wait(mutex);
115         tree4.remove(key);
116         std::this_thread::sleep_for(
117             std::chrono::milliseconds(distribution(generator) / 16));
118         sem_post(mutex);
119       });
120
121   // this way, the father waits for all the child processes
122   while ((wpid = wait(&status)) > 0)
123     ;
124
125   sem_close(mutex);
126   shm_unlink("/myshm");
127   return 0;
128 }
```

Сравнение реализаций

Строго сравнивать эти реализации нельзя, т.к. каждая из них имеет свои особенности. Но в общем виде можно сравнить время выполнения одинаковых задач.
Результаты замеров представлены в таблице 1.

-	100	300	500	700	900	1100
naive	86.79 сек	136.41 сек	189.04 сек	240.7 сек	290.3 сек	338.19 сек
pthreads	70.71 сек	87.68 сек	105.66 сек	120.08 сек	135.69 сек	152.9 сек
openmp	61.89 сек	61.88 сек	73.71 сек	91.1 сек	111.75 сек	121.82 сек
cpp11	61.07 сек	61.09 сек	62.01 сек	62.49 сек	62.92 сек	63.36 сек
ipc	65.82 сек	67.84 сек	73.91 сек	90.55 сек	110.39 сек	120.74 сек

Графическое представление этой таблице показано на рисунке 2.

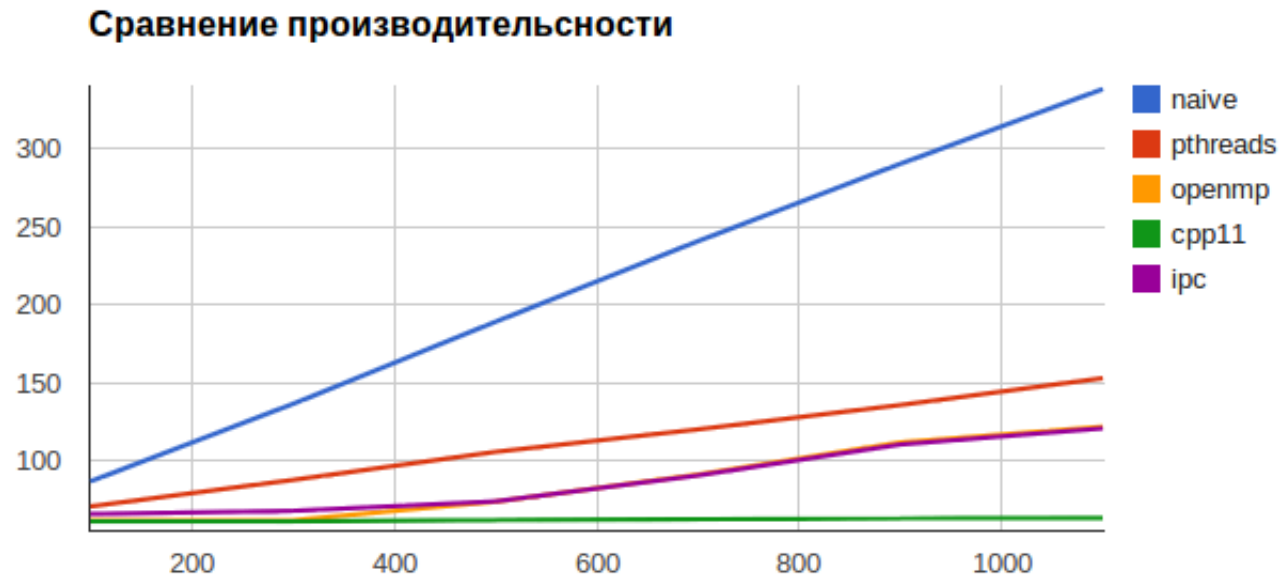

Рис. 2: График производительности различных технологий

Тестирование проводилось на процессоре Intel(R) Core(TM)2 Quad CPU Q8300 @2.50GHz с 8 GB RAM. Для измерения скорости использовалась встроенная утилита time с параметром –f "%e"для отображения общего количества затраченных секунд.

Из графика и таблицы видно, что OpenMP и встроенные средства Posix дали очень близкие результаты. Наиболее быстрая работа была у реализации на C++, видимо за счёт асинхронного вызова процедур с не ограниченным количеством процессов. Наихудший вариант, как и следовало ожидать, однопоточная реализация. На графике можно видеть линейную зависимость от размера входа.

Заключение

Многопоточность является естественным продолжением многозадачности, точно также как виртуальные машины, позволяющие запускать несколько ОС на одном компьютере, представляют собой логическое развитие концепции разделения ресурсов.

Уровень контроля над потоками в многопоточном приложении выше, чем уровень контроля приложения над дочерними процессами. Кроме того, многопоточные программы не склонны оставлять за собой вереницы зомби или «осиротевших» независимых процессов. Для отслеживания подобных ситуаций можно использовать специальный отладчики и средства, предоставляемые операционной системой.

Из инструментов, рассмотренных в данной работе наилучшее впечатление оставил о себе C++11. В C++11, работа с потокам осуществляется по средствам класса std::thread (доступного из заголовочного файла <thread>), который может работать с регулярными функциями, лямбдами и функторами. В данной работе не покрыты все возможности языка, вопрос требует дополнительного изучения.

Список литературы

1. Х.М. Дейтел,П.Дж. Дейтел, Д.Р. Чофнес. Операционные системы: Основы и принципы. Третье
2. издание. Пер. с англ. - М.ООО"Бинм-Пресс 2009 г. - 1024 с.
3. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления — СПб: БХВ-Петербург, 2002. — 608 с.
4. Немнюгин С., Стесик О. - Параллельное программирование для многопроцессорных вычислительных систем. - СПб. БХВ-Петербург,2002. – 400с.
5. Робачевский А., Немнюгин С., Стесик О. - Операционная система UNIX, 2 изд., СПб: БХВ 2010.- 656с.
6. Боресков А.В., Харламов А.А. Основы работы с технологией CUDA. М:ДМК-Пресс. 2010, -232с.
7. B. Nichols, D. Buttlar, J.P. Farrell: Pthreads Programming - A POSIX Standard for Better Multiprocessing, O'Reilly, 1996.-288p.
8. B. Eckel. Thinking in Java (4th Edition). Prentice Hall, 2006.-1150p.
9. B. Goetiz, T. Peierls. Java concurrency in practice. Addison-Wesley Professional, 2006, -384p.
10. IEEE standard SystemC language reference manual, IEEE Std 1666, 2005 – http://standards.ieee.org/getieee/1666/download/1666-2005.pdf
11. Официальный сайт OpenMP – http://openmp.org/wp/
12. Message Passing Interface Forum – http://www.mpi-forum.org/