**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра вычислительной техники**

**ОТЧЕТ**

**по лабораторной работе № 1**

**по дисциплине «Параллельные алгоритмы и системы»**

**Тема: Хэш-таблица с закрытой адресацией**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9307 |  | Николаев Н.В. |
| Преподаватель |  | Пазников А.А. |

Оглавление

[Цель работы 3](#_Toc131731446)

[Основные положения 3](#_Toc131731447)

[Измерения 3](#_Toc131731448)

[1. Без оптимизаций 4](#_Toc131731449)

[2. Использование битовых операций 4](#_Toc131731450)

[3. Использование внутренних функций С++ 4](#_Toc131731451)

[4. Использование распараллеливания с помощью OpenMP 5](#_Toc131731452)

[Вывод 5](#_Toc131731453)

[**Приложение** 5](#_Toc131731454)

# Цель работы

Разработать хэш-таблицу с закрытой адресацией и оптимизировать операции вставки и поиска.

# Основные положения

Было решено реализовать хеш-таблицу для пары строка-строка, так как шаблонные классы и структуры будут работать медленнее и хуже обрабатываются компилятором.

В качестве ячеек таблицы используется вспомогательная структура Node. Ключи и значения являются объектами стандартной библиотеки строк. Сама таблица задается значениями максимального размера. Так как хеш-таблица с закрытой адресацией, при коллизии обновляем значение занятой ячейки.

Получение и удаления выполняются аналогично поиском подряд идущих элементов, пока не найдем элемент с таким же ключом.

# Измерения

Измерения будут происходить по заранее написанному тесту, где 10000 циклов уходит на прогрев процессора, 10000 циклов уходит на вставку и получение элементов из таблицы. При этом внутри программы замеряется время выполнения, а с помощью утилиты perf в системе Linux будет произведен замер других характеристик, таких как кэш-миссы

Для анализа быстродействия алгоритмы будут использоваться следующие параметры:

* Task-clock – время выполнения задачи
* Context-switch – количество переключений контекста выполнения
* Cache-misses – количество обращений к кэшу, когда необходимых данных в нем нет
* Branch-misses – количество неправильных предсказаний условных переходов

# Без оптимизаций

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Компилятор | Task-clock | Context-switches | Cache-misses | Branch-misses |
| GCC | 6366.49 | 319 | 23.351% | 0.02% |
| CLANG | 5732.61 | 505 | 21.317% | 0.02% |

# Использование битовых операций

Одним из методов оптимизации является избавление от операции взятия остатка на битовую операция «И». Однако, после тестирования выяснилось, что такой метод не совсем подошел к нашей программе, скорее всего из-за самостоятельно реализованной ячейки таблицы, а также крайне простой хэш-фукнции:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Компилятор | Task-clock | Context-switches | Cache-misses | Branch-misses |
| GCC | 19239 | 733 | 28.33% | 0.02% |
| CLANG | 16543 | 941 | 25.2% | 0.02% |

# Использование внутренних функций С++

Вместо того, чтобы использовать собственный класс Node, можно использовать vector или list, который имеет удобные методы для управления памятью, выделения, расширения и т.п. В нашем случае, мы будем использовать std::pair двумерный массив для представления таблицы. Как видно из таблицы, приведенной ниже, избавление от самописной структуры Node привело к значительному увеличению производительности.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Компилятор | Task-clock | Context-switches | Cache-misses | Branch-misses |
| GCC | 735.26 | 8 | 16.799% | 0.02% |
| CLANG | 639.36 | 407 | 16.387% | 0.03% |

# Использование распараллеливания с помощью OpenMP

Циклы внутри методов get и insert можно распараллелить с помощью библиотеки OpenMP. Из таблицы можно сделать вывод, что распараллеливание не сильно улучшает производительность, особенно для компилятора GCC. Однако, это может быть связано с особенностью вставки и поиска в хэш-таблице с закрытой адресаций.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Компилятор | Task-clock | Context-switches | Cache-misses | Branch-misses |
| GCC | 813.15 | 12 | 15.319% | 0.02% |
| CLANG | 552.65 | 458 | 14.978% | 0.02% |

# Вывод

В ходе выполнения лабораторной работы был оптимизирован алгоритм операции вставки и поиска в хэш-таблице с закрытой адресацией.

Оригинальный алгоритм показал, что метрика Cache-misses и Task-clock можно минимизировать

Замена взятия остатка на битовое «И» отработало хуже всего, из-за самописной структуры ячейки и простейшей хэш-функции

Лучше всего помогло избавление от самописной структуры ячейки таблицы, в то время как OpenMP не сильно увеличил производительность на фоне использования стандартных библиотек с++

## **Приложение**

#include <iostream>  
#include <list>  
#include <chrono>  
class HashNode {  
public:  
 std::string key;  
 std::string value;  
 HashNode(std::string key, std::string value) {  
 this->key = key;  
 this->value = value;  
 }  
};  
  
class HashTable {  
private:  
 HashNode\*\* table;

int capacity; *// capacity of the hash table* int hashCode(std::string key); public:  
 HashTable(int capacity) : capacity(capacity) {  
 table = new HashNode\*[capacity];  
 for (int i = 0; i < capacity; i++)  
 table[i] = NULL; *// initialize the table with empty linked lists* }  
  
 void insert(std::string key, std::string value) {  
  
 int hash = (hashCode(key) % capacity);  
 while (table[hash] != NULL && !std::equal(table[hash]->key.begin(), table[hash]->key.end(), key.begin(), key.end()))  
 hash = (hash + 1) % capacity;  
  
 if (table[hash] == NULL){  
 table[hash] = new HashNode(key,value);  
 }  
 else  
 table[hash]->value = value;  
  
 }  
  
 std::string get(std::string key) {  
 int hash = (hashCode(key) % capacity);  
 while (table[hash] != NULL && !std::equal(table[hash]->key.begin(), table[hash]->key.end(), key.begin(), key.end()))  
 hash = (hash + 1) % capacity;  
  
 if (table[hash] == NULL)  
 return "";  
 else  
 return table[hash]->value;  
 }};  
  
int HashTable::hashCode(std::string key)  
{  
 int sum = 0;  
 for (char c : key) {  
 sum += c;  
 }  
 return sum % capacity;  
}  
  
  
void generateTestData(int size, HashTable& table) {  
 for (int i = 0; i < size; ++i) {  
 table.insert(std::to\_string(i + 10), std::to\_string(i));  
 }  
}  
  
void getTestData(int size, HashTable& table) {  
 for (int i = 0; i < size; ++i) {  
 printf("number: %s",table.get(std::to\_string(i + 10)).c\_str());  
 }  
}  
  
void gitTest(int warmUpIterations, int mainIterations, int maxInt, HashTable& map) {  
  
 *//разогрев* std::string result = "";  
 for (long i = 0; i < warmUpIterations; i++) {  
 int num1 = (int)i % maxInt;  
 int num2 = num1 \* 2;  
 int num3 = num1 + 1;  
 map.insert(std::to\_string(num1), std::to\_string(num2));  
 std::string value = map.get(std::to\_string(num3));  
 if (!value.empty()) {  
 result += value;  
 *//std::cout << value << std::endl;* }  
 }  
  
 *//основное тестирование* result = "";  
 auto start = std::chrono::high\_resolution\_clock::now();  
 for (long i = 0; i < mainIterations; i++) {  
 int num1 = (int)i % maxInt;  
 int num2 = num1 \* 2;  
 int num3 = num1 + 1;  
 map.insert(std::to\_string(num1), std::to\_string(num2));  
 std::string value = map.get(std::to\_string(num3));  
 if (!value.empty()) {  
 result += value;  
 std::cout << value << std::endl;  
 }  
 }  
 auto stop = std::chrono::high\_resolution\_clock::now();  
 auto testTime = std::chrono::duration\_cast<std::chrono::duration<double>>(stop - start);  
 *//std::cout << result << std::endl;* std::cout << "Test time: " << testTime.count() << "seconds" << std::endl;  
}  
  
void mapTest(int size, HashTable& table) {  
 generateTestData(size, table);  
 getTestData(size, table);  
}  
  
int main() {  
 HashTable ht(100001);  
gitTest(100000, 100000, 1000, ht);  
 return 0;  
}