МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра вычислительной техники

Отчет по лабораторной работе №2
по дисциплине
«Параллельные алгоритмы и системы»
Тема: «Сбалансированное бинарное дерево»

Студент гр. 9306 _____ Смирнов А.В.

Преподаватель _____ Пазников А.А.

Санкт-Петербург
2023

СОДЕРЖАНИЕ

Цель

Практическое закрепление понимания общих идей оптимизации алгоритмов.

Постановка задачи

Требуется оптимизировать программу, проанализировать полученные результаты, построить графики. При оптимизации использовать распараллеливание OpenMP. Эксперименты провести на двух компиляторах: Clang, GCC. Для выполнения лабораторной работы был выбран язык программирования C.

Описание алгоритма

Листинг 1 – Код алгоритма без оптимизации

```
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>
#include <limits.h>

typedef struct AVLNode {
    int data;
    int height;
    struct AVLNode *left;
    struct AVLNode *right;
} AVLNode;

// функция для создания нового узла
AVLNode *newNode(int data) {
    AVLNode *node = (AVLNode *)malloc(sizeof(AVLNode));
    node->data = data;
    node->height = 1;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// функция для получения высоты узла
int height(AVLNode *node) {
    if (node == NULL)
        return 0;
    return node->height;
}

// функция для получения баланса узла
int balanceFactor(AVLNode *node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

int max(int a, int b) {
    if(a>b) return a;
    else return b;
}

// функция для поворота вправо
AVLNode *rotateRight(AVLNode *y) {
    AVLNode *x = y->left;
    AVLNode *T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = 1 + max(height(y->left), height(y->right));
    x->height = 1 + max(height(x->left), height(x->right));
    return x;
}

// функция для поворота влево
AVLNode *rotateLeft(AVLNode *x) {
    AVLNode *y = x->right;
    AVLNode *T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = 1 + max(height(x->left), height(x->right));
    y->height = 1 + max(height(y->left), height(y->right));
    return y;
}

// функция для добавления нового элемента в дерево
AVLNode *insert(AVLNode *node, int data) {
    if (node == NULL)
        return newNode(data);

    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else
        return node;
```

```
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = balanceFactor(node);

    // Проверка нарушения баланса и восстановление
    if (balance > 1 && data < node->left->data)
        return rotateRight(node);

    if (balance < -1 && data > node->right->data)
        return rotateLeft(node);

    if (balance > 1 && data > node->left->data) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }

    if (balance < -1 && data < node->right->data) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}


float tdiff(struct timeval *start, struct timeval *end){
    return (end -> tv_sec - start -> tv_sec) + 1e-6*(end -> tv_usec - start ->
tv_usec);
};

int main()
{
    int N_ITER = 10000000;
    AVLNode* rootp = NULL;
    struct timeval start, end;
    printf("Hello AVL-TREE!!!\n");

    printf("START...\n");
    gettimeofday(&start, NULL);

    rootp = newNode(1+rand()%N_ITER);
    for(int i = 1; i<N_ITER; ++i)
        rootp = insert(rootp, 1+rand()%N_ITER);

    gettimeofday(&end, NULL);
    printf("STOP | execution time: %0.6f\n", tdiff(&start, &end));
    return 0;
}
```

Ниже представлены замеры времени выполнения программы, добавляющей 10 000 000 вершин.

Оптимизация алгоритма

Рассмотрим некоторые методы оптимизации программы с использованием компиляторов GCC и Clang. Каждый последующий метод будет добавляться к успешно выполненному предыдущему.

1. GCC

Без каких-либо мероприятий по оптимизации программы среднее время выполнения программы составило 9,6234 мс.

1.1 Отделение объявления

После того как было применено изменение, кажущееся на первый взгляд незначительным, результат исполнения программы изменился практически вдвое, а именно, было отделено объявление функций от описаний тел функций, то есть код принял вид как в листинге 2. Этот шаг дал среднее время выполнения программы 4.5550 секунд.

Листинг 2 – Разделение объявлений функций

```
...
AVLNode *newNode(int data);
int height(AVLNode *node);
int balanceFactor(AVLNode *node);
int max(int a, int b);
AVLNode *rotateRight(AVLNode *y);
AVLNode *rotateLeft(AVLNode *x);
AVLNode *insert(AVLNode *node, int data);
...
int main()
{
    ...
}

AVLNode *newNode(int data) {
    ...
}

int height(AVLNode *node) {
    ...
}

int balanceFactor(AVLNode *node) {
    ...
}

int max(int a, int b) {
    ...
}

AVLNode *rotateRight(AVLNode *y) {
    ...
}

AVLNode *rotateLeft(AVLNode *x) {
    ...
}

AVLNode *insert(AVLNode *node, int data) {
    ...
}
```

1.2 Группировка функций

Путем подбора различных вариаций расположений функций в коде было достигнуто оптимальное расположение, при котором время исполнения программы сократилось на 0,1 секунды и составило 4.4093 секунды.

Листинг 3 – Измененный порядок функций

```
int main() {}
AVLNode *insert(AVLNode *node, int data) {}
int height(AVLNode *node) {}
int max(int a, int b) {}
AVLNode *rotateRight(AVLNode *y) {}
AVLNode *rotateLeft(AVLNode *x) {}
AVLNode *newNode(int data) {}
int balanceFactor(AVLNode *node) {}
```

1.3 Встраивание функций

Для сокращения расходования ресурсов на вызов функций, эти функции были интегрированы туда откуда они вызывались (function inlining). Были заменены функции *heigh, max, balanceFactor*. Более подробно про то на что были заменены вызовы функций можно посмотреть в результирующем коде программы. Данный шаг сократил время исполнения программы до 3,3231 секунд.

Листинг 4 – function inlining

```
БЫЛО:
node->height = 1 + max(height(node->left), height(node->right));

СТАЛО:
int max1 = node->left==NULL?1:node->left->height+1;
int max2 = node->right==NULL?1:node->right->height+1;
node->height = max1>max2?max1:max2;
```

1.4 Предсказание переходов

В моменте, когда происходит добавление нового узла, алгоритму приходится сначала перебрать имеющееся дерево, чтобы найти место куда будет встраиваться новый узел, это значит, что наиболее вероятно при этом переборе выбираемый узел будет ненулевой. А значит можно оптимизировать этот момент. Это уменьшило время, необходимое на выполнение программы, до 3,2373 секунд.

Листинг 5 – branch prediction

```
#define likely(x)   __builtin_expect (!!(x), 1)
#define unlikely(x) __builtin_expect (!!(x), 0)

БЫЛО:
if (node == NULL)
return newNode(data);
...
int max1 = node->left==NULL?1:node->left->height+1;
int max2 = node->right==NULL?1:node->right->height+1;
node->height = max1>max2?max1:max2;

СТАЛО:
if (unlikely(node == NULL))
return newNode(data);
...
int max1, max2;
if(likely(node->left==NULL))
max1 = 1;
else max1 = node->left->height+1;
if(likely(node->right==NULL))
max2 = 1;
else max2 = node->right->height+1;
node->height = max1>max2?max1:max2;
```

1.5 Отделение частных случаев

В подавляющем большинстве случаев при вызове функции insert, так как эта функция вызывается рекурсивно, нет необходимости дополнительно вызывать еще один экземпляр функции для создания новой вершины. Действия которые были выполнены сложно назвать каким то одним словом, но это чем то похоже на function inlining или на fynction/cycle splitting. Данный шаг весьма спорный и дал незначительное ускорение работы программы, что также можно списать на некоторую погрешность измерений. В итоге время выполнения программы сократилось до 3,1901 секунд.

Листинг 6 – fynction/cycle splitting

```
БЫЛО:
...
int main {
...
    for(int i = 0; i<N_ITER; ++i)
        rootp = insert(rootp, 1+rand()%N_ITER);
...
}
...
AVLNode *insert(AVLNode *node, int data) {
    if (unlikely(node == NULL))
        return newNode(data);

    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else
        return node;
    ...
}
...

СТАЛО:
...
int main {
...
    rootp = newNode(1+rand()%N_ITER);
    for(int i = 1; i<N_ITER; ++i)
        rootp = insert(rootp, 1+rand()%N_ITER);
...
}
...
AVLNode *insert(AVLNode *node, int data) {
    if (data < node->data)
        if(unlikely(node->left == NULL))
            node->left = newNode(data);
        else node->left = insert(node->left, data);
    else if (data > node->data)
        if(unlikely(node->right==NULL))
            node->right = newNode(data);
        else node->right = insert(node->right, data);
    else
        return node;
    ...
}
...
```

1.6 Ключ оптимизации

Далее пробовались различные ключи компилятора для оптимизации программы, но ни один из них не дал значительную прибавку к результату. Скорее всего это связано с тем, что выше принятые меры усложнили структуру кода, и компилятор не справляется с его оптимизацией, говоря простыми словами «перемудрили».

Таблица 1– Ключи оптимизации

Ключ	-O1	-O2	-O3
Время выполнения	3,3131	3,4026	3,3869

Листинг 7 – результирующий код программы

```
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>
#include <limits.h>

#define likely(x)   __builtin_expect (!!(x), 1)
#define unlikely(x) __builtin_expect (!!(x), 0)

// структура для представления узлов дерева
typedef struct AVLNode {
    int data;
    int height;
    struct AVLNode *left;
    struct AVLNode *right;
} AVLNode;

AVLNode *newNode(int data);
int height(AVLNode *node);
int balanceFactor(AVLNode *node);
int max(int a, int b);
AVLNode *rotateRight(AVLNode *y);
AVLNode *rotateLeft(AVLNode *x);
AVLNode *insert(AVLNode *node, int data);

float tdiff(struct timeval *start, struct timeval *end){
    return (end -> tv_sec - start -> tv_sec) + 1e-6*(end -> tv_usec - start ->
tv_usec);
};

int main()
{
    int N_ITER = 10000000;
    AVLNode* rootp = NULL;
    struct timeval start, end;
    printf("Hello AVL-TREE!!!\n");

    printf("START...\n");
    gettimeofday(&start, NULL);

    rootp = newNode(1+rand()%N_ITER);
    for(int i = 1; i<N_ITER; ++i)
        rootp = insert(rootp, 1+rand()%N_ITER);

    gettimeofday(&end, NULL);
    printf("STOP | execution time: %0.6f\n", tdiff(&start, &end));
    return 0;
}

// функция для добавления нового элемента в дерево
AVLNode *insert(AVLNode *node, int data) {
    if (data < node->data)
        if(unlikely(node->left == NULL))
            node->left = newNode(data);
        else node->left = insert(node->left, data);
    else if (data > node->data)
        if(unlikely(node->right==NULL))
            node->right = newNode(data);
        else node->right = insert(node->right, data);
    else
        return node;

    int max1, max2;
    if(likely(node->left==NULL))
        max1 = 1;
    else max1 = node->left->height+1;
    if(likely(node->right==NULL))
        max2 = 1;
    else max2 = node->right->height+1;
    node->height = max1>max2?max1:max2;

    int balance = (node->left==NULL?0:node->left->height) - (node-
>right==NULL?0:node->right->height);

    // Проверка нарушения баланса и восстановление
    if (balance > 1 && data < node->left->data)
        return rotateRight(node);

    if (balance < -1 && data > node->right->data)
```

```
        return rotateLeft(node);

    if (balance > 1 && data > node->left->data) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }

    if (balance < -1 && data < node->right->data) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}


// функция для поворота вправо
AVLNode *rotateRight(AVLNode *y) {
    int max1, max2;
    AVLNode *x = y->left;
    AVLNode *T2 = x->right;
    x->right = y;
    y->left = T2;
    max1 = y->left==NULL?1:y->left->height+1;
    max2 = y->right==NULL?1:y->right->height+1;
    y->height = max1>max2?max1:max2;
    max1 = x->left==NULL?1:x->left->height+1;
    max2 = x->right==NULL?1:x->right->height+1;
    x->height = max1>max2?max1:max2;
    return x;
}

// функция для поворота влево
AVLNode *rotateLeft(AVLNode *x) {
    int max1, max2;
    AVLNode *y = x->right;
    AVLNode *T2 = y->left;
    y->left = x;
    x->right = T2;
    max1 = x->left==NULL?1:x->left->height+1;
    max2 = x->right==NULL?1:x->right->height+1;
    x->height = max1>max2?max1:max2;
    max1 = y->left==NULL?1:y->left->height+1;
    max2 = y->right==NULL?1:y->right->height+1;
    y->height = max1>max2?max1:max2;
    return y;
}

// функция для создания нового узла
AVLNode *newNode(int data) {
    AVLNode *node = (AVLNode *)malloc(sizeof(AVLNode));
    node->data = data;
    node->height = 1;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```

2. Clang

Те же самые коды программ были запущены с использованием компилятора Clang, и были получены следующие показатели времени исполнения программы

2.1 Отделение объявления функций: 5,3294 с;

2.2 Function grouping: 4,4141 с;

2.3 Function inlining: 3,8383с;

2.4 Branch prediction: 3,6905 с;

2.5 Function splitting: 3,6367 с.

Сравнительный анализ

Ниже представлена таблица с информацией о производительности методов с использованием различных компиляторов.

Таблица 2 – Сравнение производительности методов оптимизации

Метод оптимизации	GCC		
	Running time	Прирост относительно прошлого шага	Прирост относительно варианта без оптимизации
Без оптимизации	9,6234	1,00	1,00
+ Отделение объявлений функций	4,555	2,11	2,11
+ Function grouping	3,7727	1,21	2,55
+ Function inlining	3,3231	1,14	2,90
+ Branch prediction	3,2373	1,03	2,97
+ Function splitting	3,1901	1,01	3,02

Таблица 3 – Сравнение производительности методов оптимизации

Метод оптимизации	Clang		
	Running time	Прирост относительно прошлого шага	Прирост относительно варианта без оптимизации
Без оптимизации	10,3933	1,00	1,00
Отделение объявлений	5,3294	1,95	1,95
+ Function grouping	4,4141	1,21	2,35
+ Function inlining	3,8383	1,15	2,71
+ Branch prediction	3,6905	1,04	2,82
+ Function splitting	3,6367	1,01	2,86

По полученным результатам были построены графики производительности программы (см. рис. 1-2).

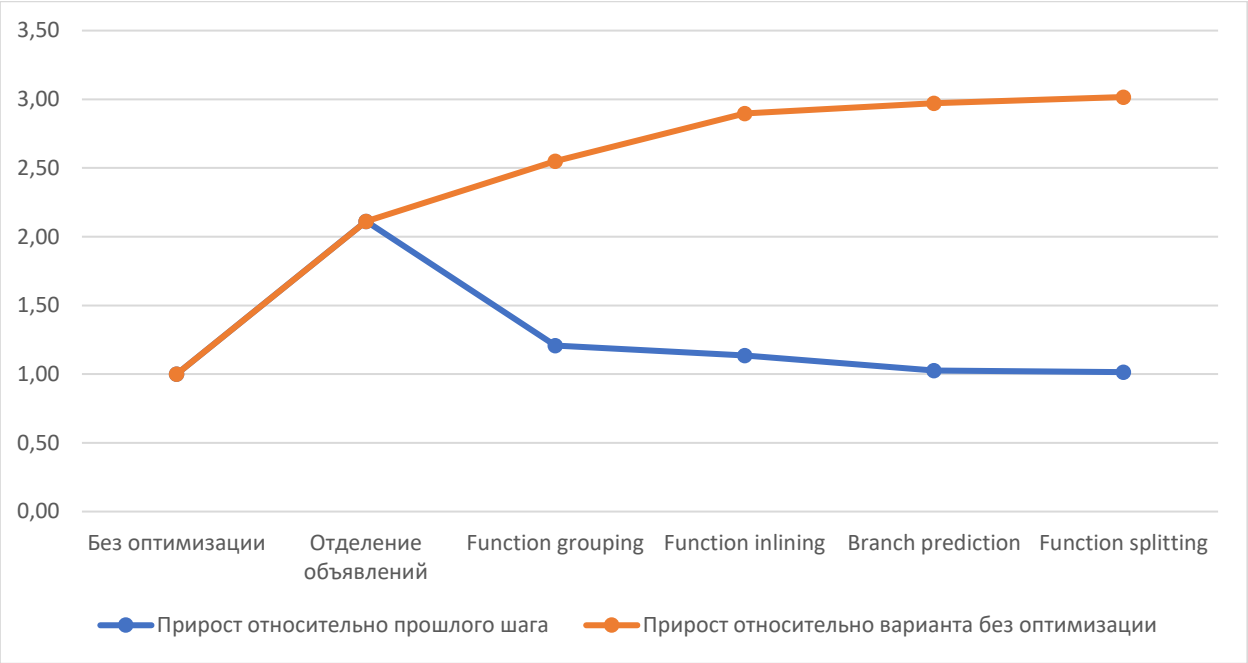

Рисунок 1 – График производительности программы с использованием компилятора GCC

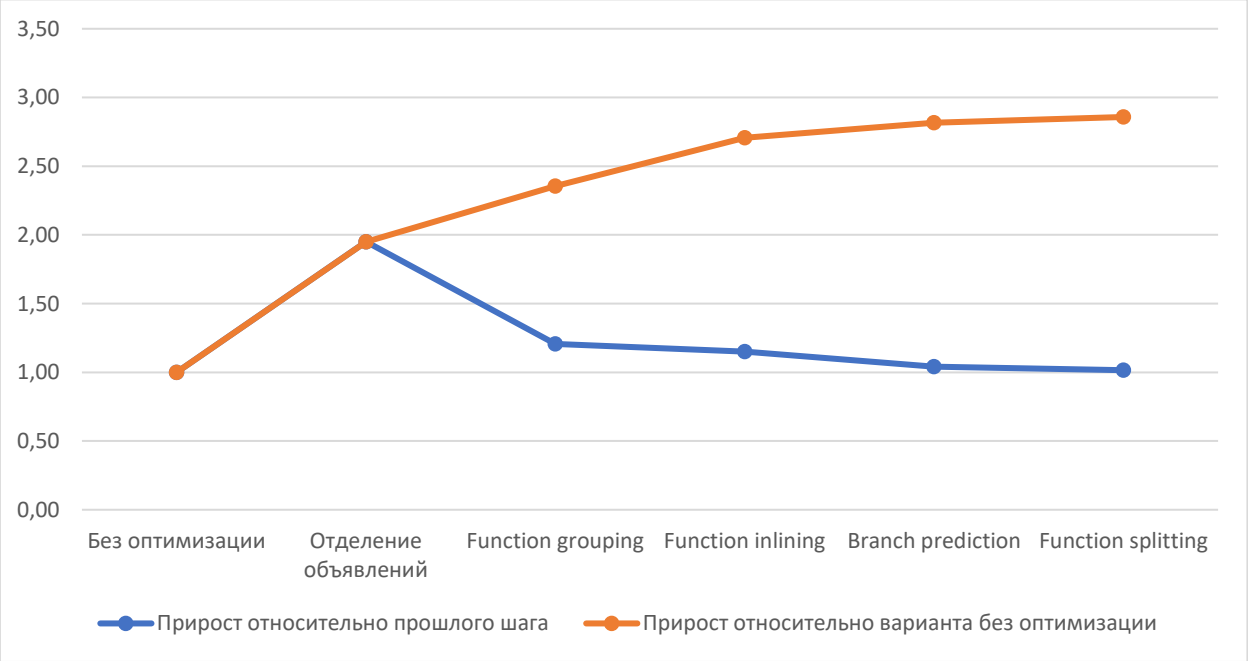

Рисунок 2 – График производительности программы с использованием компилятора Clang

Вывод

В результате выполнения лабораторной работы были получены знания о процессах оптимизации, а также навыки их практического применения. Были реализованы разные методы оптимизации программы, выполняющей вставку 10 000 000 элементов в сбалансированное AVL-дерево, которые суммарно дали прирост производительности примерно в 3 раза.