

Санкт-Петербургский государственный электротехнический университет  
“ЛЭТИ” им. В. И. Ульянова (Ленина)  
(СПбГЭТУ “ЛЭТИ”)

---

**Направление подготовки:** 09.03.01 “Информатика и вычислительная техника”

**Профиль:** Организация и программирование вычислительных и информационных систем

**Факультет:** Компьютерных технологий и информатики

**Кафедра:** Вычислительной техники

*К защите допустить:*

**Заведующий кафедрой**

д. т. н., профессор \_\_\_\_\_ М. С. Куприянов

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
БАКАЛАВРА**

**Тема: Реализация и оптимизация связанных неблокирующих списков**

Студент \_\_\_\_\_ А. А. Кашин

Руководитель  
к. т. н., доцент \_\_\_\_\_ А. А. Пазников

Консультант  
по разработке и стандартизации  
программных средств  
к. э. н., доцент \_\_\_\_\_ М. А. Косухина

Консультант от кафедры  
к. т. н., доцент, с. н. с. \_\_\_\_\_ И. С. Зуев

Санкт-Петербург  
2023 г.

**Санкт-Петербургский государственный электротехнический университет  
“ЛЭТИ” им. В. И. Ульянова (Ленина)  
(СПбГЭТУ “ЛЭТИ”)**

Направление 09.03.01 «Информатика и  
вычислительная техника»  
Профиль Организация и программирование  
вычислительных и информационных систем  
Факультет компьютерных технологий и  
информатики  
Кафедра вычислительной техники

**УТВЕРЖДАЮ**  
Заведующий кафедрой ВТ  
д. т. н., профессор  
(М. С. Куприянов)  
“ \_\_\_\_ ” \_\_\_\_\_ 2023 г.

**ЗАДАНИЕ  
на выпускную квалификационную работу**

Студент Кашин Андрей Александрович

Группа № 9305

**1. Тема** Реализация и оптимизация связанных неблокирующих  
списков

*(утверждена приказом № \_\_\_\_\_ от \_\_\_\_\_)*

Место выполнения ВКР: Кафедра вычислительной техники СПбГЭТУ  
«ЛЭТИ»

**2. Объект и Предмет исследования:** объектом исследования являются разделяемые структуры данных, предметом исследования являются неблокирующие связанные списки.

**3. Цель:** реализация неблокирующего списка, применение и сравнительный анализ производительности списков из готовой библиотеки Libcds.

**4. Исходные данные:** русскоязычные и англоязычные статьи ученых по теме, серия статей, документация по Libcds и видео-лекции от автора библиотеки в сети Интернет.

- 5. Содержание:** обзор принципов работы неблокирующих списков, реализация алгоритма и применение структур данных из библиотеки Libcds, сравнение производительности структур данных Libcds при разных параметрах.
- 6. Технические требования:** 64-разрядный двухъядерный или с большим количеством ядер процессор Intel с тактовой частотой не ниже 1,8 ГГц. 4 ГБ ОЗУ. Место на жестком диске до 13 ГБ. Компиляторы GCC 4.8+/ clang 3.6+/ Intel C++ 15+/ MSVC++ 14 и выше.
- 7. Дополнительные разделы:** разработка и стандартизация программных средств.
- 8. Результаты:** Подготовленная программа, реализующая алгоритм неблокирующего списка, программы с тестами библиотеки Libcds.
- 9. Перечень отчетных материалов:** Пояснительная записка, реферат, аннотация, презентация.

Дата выдачи задания  
«04»марта2023 г.

Дата представления ВКР к защите  
«16» июня 2023г.

Руководитель

К. Т. Н., доцент

Студент

\_\_\_\_\_

\_\_\_\_\_

А. А. Пазников

А. А. Кашин

**Санкт-Петербургский государственный электротехнический университет  
“ЛЭТИ” им. В. И. Ульянова (Ленина)  
(СПбГЭТУ “ЛЭТИ”)**

---

Направление 09.03.01 «Информатика и  
вычислительная техника»  
Профиль Организация и программирование  
вычислительных и информационных систем  
Факультет компьютерных технологий и  
информатики  
Кафедра вычислительной техники

**УТВЕРЖДАЮ**  
Заведующий кафедрой ВТ  
д. т. н., профессор  
(М. С. Куприянов)  
“ \_\_\_\_ ” \_\_\_\_\_ 2023 г.

**КАЛЕНДАРНЫЙ ПЛАН  
выполнения выпускной квалификационной работы**

Тема Реализация и оптимизация связанных неблокирующих списков

Студент Кашин Андрей Александрович                      Группа № 9305

№ этапа	Наименование работ	Срок выполнения
1	Освоение материалов, необходимых для выполнения выпускной квалификационной работы	04.03.2023 – 18.04.2023
2	Написание теоретической части	19.04.2023 – 05.05.2023
3	Выполнение практической части	05.05.2023 – 31.05.2023
4	Описание практической части	01.06.2023 – 03.06.2023
7	Оформление пояснительной записки	04.06.2023 – 08.06.2023
8	Предварительное рассмотрение работы	09.06.2023 – 12.06.2023
9	Представление работы к защите	16.06.2023

Руководитель  
к. т. н., доцент  
Студент

\_\_\_\_\_

А. А. Пазников  
А. А. Кашин

## РЕФЕРАТ

Пояснительная записка содержит: 53 с., 21 рис., 4 табл., 2 приложения, 11 источников литературы.

Цель работы: исследование принципов работы неблокирующих связанных списков, реализация алгоритма списка Харриса по псевдокоду, исследование и применение библиотеки Libcds, сравнение производительности всех рассмотренных алгоритмов. Применяются программные продукты: CLion, Clang16, Libcds.

В настоящей работе приводится обзор принципов реализации неблокирующих списков: список Харриса, список Майкла, Lazy-список. Приводится реализация списка Харриса, примеры работы списков Харриса, Майкла и Lazy-списка, рассматриваются опции, реализованные в библиотеке для соответствующих структур, проводится сравнительный анализ производительности всех структур, проводится исследование влияния оптимизаций компилятора на производительность структур.

По результатам исследования создана программа, реализующая алгоритмы списка Харриса без операций физического удаления и программы, проводящие тесты реализованных в библиотеке Libcds списков Майкла и Lazy с различным набором опций.

Данная работа может быть использована для ознакомления с принципами работы неблокирующих связанных списков и библиотекой Libcds, реализованный в данной работе алгоритм списка Харриса может быть использован в многопоточных приложениях, где не требуется удаление данных, сравнение производительностей структур из Libcds и исследование методов оптимизации может быть использовано при внедрении библиотеки в готовые многопоточные проекты для большей эффективности.

# ABSTRACT

Purpose of the work: study of the beginning of operation of non-blocking linked lists, implementation of the Harris list library algorithm in pseudocode, study and application of Libcds, comparison of the performance of all algorithms. Software products used: Excel, CLion, Clang16, Libcds.

This paper provides an overview of the implementation of non-blocking lists: Harris' list, Michael's list, Lazy-list. The implementation of the Harris list is given, examples of the operation of the lists of Harris, Michael and the Lazy-list, the application task, the implementation of which in the library corresponds to the structure, a comparative analysis of the performance of all structures is carried out.

Based on the results of the study, a program was created that implements the Harris list algorithms without physical deletion operations and programs that test the Michael and Lazy lists implemented in the Libcds library with a different set of options.

This work can be used to get acquainted with the principles of operation of non-blocking linked lists and the Libcds library, the Harris list algorithm implemented in this work can be used in multithreaded applications where data deletion is not required, comparing the performance of Libcds structures can be used when implementing the library into ready-made multi-threaded projects for greater efficiency.

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	9
1 НЕБЛОКИРУЮЩИЕ АЛГОРИТМЫ.....	11
1.1 Алгоритм Харриса (Harris' non-blocking linked list) .....	13
1.2 Библиотека Libcds .....	17
2 ПРАКТИЧЕСКИЕ ТЕСТЫ АЛГОРИТМОВ.....	24
2.1 Алгоритм Харриса.....	24
2.2 Тесты с библиотекой Libcds .....	25
2.3 Исследование методов оптимизации, применяемых к Libcds.....	32
3 РАЗРАБОТКА И СТАНДАРТИЗАЦИЯ ПРОГРАММНЫХ СРЕДСТВ .....	36
3.1 Диаграмма Ганта .....	36
3.2 Расчёт затрат на выполнение проекта, расчет цены проекта и цены продукта.....	38
3.3 Обеспечение качества программного продукта.....	42
3.4 Определение кода программного продукта .....	43
3.5 Определение списка стандартов .....	43
ЗАКЛЮЧЕНИЕ .....	44
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	45
ПРИЛОЖЕНИЕ А .....	46
ПРИЛОЖЕНИЕ Б.....	51

## ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Clang – транслятор С-подобных языков.

LLVM – набор компиляторов из языков высокого уровня, системы оптимизации, интерпретации и компиляции в машинный код.

Boost – это совокупность библиотек классов для языка C++, реализующих интерфейс для повседневных подзадач в коде.

Libcds – библиотека с реализациями lock-free алгоритмов.

dead-lock – взаимная блокировка; случай, когда несколько потоков блокируют друг друга в ожидании освобождения ресурсов, которые заняты ими.

Lock-free – неблокирующие списки.

Мьютекс, семафор, критическая секция – примитивы синхронизации.



# ВВЕДЕНИЕ

Поддержка многопоточности – важная часть программ в современном мире, где любая система обрабатывает данные. Поскольку объем и сложность данных не прекращают расти, к обрабатывающим ПО повышаются требования производительности. Один из способов повышения производительности программы – исполнение на нескольких потоках. Однако параллельные алгоритмы сложнее, и поэтому с их применением возникают новые проблемы. Ключевая проблема многопоточных алгоритмов – непредсказуемость [1], которая следует из непостоянной скорости выполнения алгоритмов в потоках. Неблокирующие структуры данных, в частности, неблокирующие списки реализуют алгоритмы, которые дают возможность использовать их в многопоточных приложениях без внешней синхронизации.

Самостоятельная реализация подобных алгоритмов возможна, но требует много времени и трудозатрат.

В данной работе приведена самостоятельная реализация списка Харриса[3] на основе публикации ученого-создателя алгоритмов.

Библиотека Libcds [2] предоставляет реализации различных структур данных, таких как массивы, списки, хеш-таблицы и деревья, которые могут быть использованы в многопоточной среде.

Основная цель библиотеки - обеспечить высокую производительность и масштабируемость при работе с многопоточными структурами данных. Она использует различные техники для устранения гонок за ресурсами и dead-lock, такие как блокировки на уровне объектов и атомарные операции. Libcds предоставляет несколько опций для каждого из контейнеров для повышения производительности или выполнения требований работы алгоритмов для конкретного случая.

Библиотека создана Максимом Хижинским, ведущим инженером компании “VasExperts” в 2009 году и распространяется под лицензией Boost Software License 1.0 на Github и Sourceforge.

Целью данной работы является разработка алгоритма неблокирующего связанного списка Харриса, а также применение и оптимизация следующих lock-free структур данных, реализованных в Libcds:

- Michael List;
- Lazy List;

Объектом исследования являются разделяемые структуры данных, предметом исследования являются неблокирующие связанные списки.

Основной задачей для достижения цели работы станет исследование принципов работы неблокирующих связанных списков.

Реализация алгоритма и применение всех рассматриваемых структур данных будет выполняться в среде CLion [8] с использованием компилятора Clang 16 [7].

В первом разделе приводится обзор особенностей неблокирующих структур данных, во втором разделе приводится реализация, применение и оптимизация алгоритмов, сравнительный анализ показателей производительности, в третьем — дополнительный раздел «разработка и стандартизация программных средств».

По итогам выполнения работы приводится текст программ с реализацией алгоритма Харриса, а также текст программ с тестами структур данных Libcds.

# 1 Неблокирующие алгоритмы

Алгоритм может называться неблокирующим, если в любой момент его работы прекращение или остановка одного потока не вызовет прекращение или остановку другого потока.

В многопоточном программировании традиционно используются механизмы блокировки, такие, как мьютексы, семафоры и критические секции. С их помощью разработчик ПО может пометить некоторые ресурсы или участки кода, делая одновременный доступ к ним из нескольких разных потоков невозможным. Так избегается неправильная работа участков памяти с общим доступом, когда одновременный доступ повредит содержимое. Такая ошибка называется «состояние гонки» [1]. При применении примитивов синхронизации один поток, при попытке доступа к уже занятому ресурсу, приостанавливает свое выполнение и ожидает освобождения ресурса.

В таком подходе есть несколько проблем.

Самая очевидная проблема – пока поток заблокирован, он не выполняет полезную работу, т.е. простаивает. Если такой поток будет выполнять важное действие, от которого зависит работа остальных потоков, или у него большой приоритет, то вся система просто останавливается, пока ресурсы заняты одним единственным потоком.

Другая проблема – «dead-lock» [1]. Несколько потоков могут застрять в ожидании освобождения ресурсов друг друга.

Если у потоков есть явные приоритеты, то может произойти следующий случай: есть три потока с высоким, средним и низким приоритетом – потоки High, Medium и Low соответственно. Есть ресурс Resource, который в данный момент времени блокируется Low потоком. При освобождении Resource активируется поток Medium, имеющий более высокий приоритет, чем Low, снова блокируя Resource. В таком случае поток High, обладающий наибольшим приоритетом, не может выполняться. Такая ситуация называется «priority inversion».

Неблокирующие алгоритмы решают подобные проблемы, увеличивая объем работы, который совершается параллельно, и уменьшая число последовательных действий в многопоточной системе.

Неблокирующие структуры данных подразделяются на три основных вида:

- Obstruction-free;
- Lock-free;
- Wait-free;

Общий смысл Obstruction-free структур сводится к правилу «данный поток закончит свою работу за ограниченное число шагов, при условии, что он изолирован от других потоков. Такие структуры являются самыми «слабыми» по требованиям из неблокирующих структур.

Lock-free структуры предполагают, что часть потоков завершают свое выполнение за ограниченное число шагов.

Wait-free структуры являются самыми строгими: каждый поток должен завершить любую свою операцию за конечное число шагов.

Основная проблема неблокирующих структур заключается в сложности их разработки, т.к. необходимо решать т.н. АВА-проблему [1]. Самая большая проблема в lock-free алгоритмах – удаление элементов. Обычно оно происходит в два этапа: логическое и физическое удаление. Если элемент А был удален, то на его место далее может быть вставлен элемент В, и из-за того, что адреса данных совпадают, а сами данные нет, в структуре может остаться ссылка на освобожденный участок памяти, что приведет к ошибке.

Решение такой проблемы может быть разным: можно в целом отказаться от операций удаления, а можно прибегнуть к использованию методов Safe-Memory-Reclamation (SMR).

В основе неблокирующих алгоритмов лежат операции атомарного изменения (RMW, read-modify-write) [2]. В целом, атомарные операции – операции, проводящие неделимые изменения памяти, т.е. ни один другой поток не сможет наблюдать промежуточный результат операции, когда исходный поток применяет атомарную операцию. RMW-операции являются более сложными атомарными операциями. Суть и реализация зависят от платформы, поэтому не везде такие операции будут считаться lock-free.

### **1.1 Алгоритм Харриса (Harris' non-blocking linked list)**

Алгоритм Харриса [3] сводится к использованию атомарных операций CAS (compare-and-swap). Список Харриса является сортированным по возрастанию, без повторов элементов, односвязным списком.

В самом простом случае алгоритм рассматривает линейный список узлов Node с двумя полями:

- Key, в котором хранится сам элемент;
- Next, в котором хранится ссылка на следующий элемент списка;

В основе алгоритма лежит метод разметки элементов. Все просто – тот элемент, который в данный момент удаляется другим процессом, помечается каким-либо образом, например в ссылке Next у него наименее значащий бит может быть равен нулю или единице.

Операция вставки в список реализовывается просто – применяется атомарная операция по отношению к полям next элементов слева и справа от вставляемого элемента. Если операция CAS по присваиванию левого элемента списка в данный проход цикла не выполнялась, цикл идет заново, пока выполнение не будет успешным.

Принципиальная схема алгоритма вставки элемента CAS-операциями изображена на рисунке 1.1

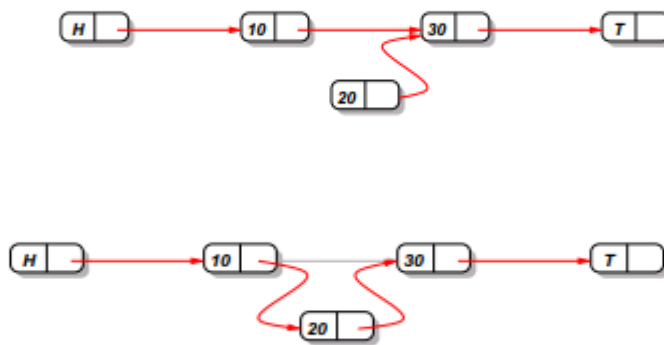


Рисунок 1.1 — Схема алгоритма вставки

Псевдокод функции вставки представлен ниже:

```
bool MyInsert (key) {
    Node*new_node = new Node(key);
    Node*right_node, *left_node;
    do {
        right_node = MySearch (key, &left_node);
        //поиск позиции для вставки
        if ((right_node != tail) && (right_node->key ==key))
            //если не найдено – выход
            return false;
        //присваивание ссылки во вставляемый элемент
        new_node->next = right_node;
        if (CAS(&(left_node->next), right_node, new_node))
            //присваивание ссылки в левый элемент.
    } while (true);
    return true;
}
```

Операция поиска находит по ключу элемент, ключ которого больше или равен искомому. Он является правым элементом в поиске. Далее находится первый элемент слева, т.е. ключ у которого меньше, чем у правого элемента. Обязательные условия – оба элемента не должны быть маркированы, должны находиться рядом, левый элемент меньше ключа поиска, правый – больше.

Псевдокод приведен ниже.

```
private Node *List::search (KeyType search_key, Node **left_node) {
    Node *left_node_next, *right_node;
    search_again:
    do {
        Node *t = head;
        Node *t_next = head.next;
        /* 1: находим левый и правый элементы, правый является искомым
        if (!is_marked_reference(t_next)) {
            (*left_node) = t;
            left_node_next = t_next;
        }
        t = get_unmarked_reference(t_next);
        if (t == tail) break;
        t_next = t.next;
        } while (is_marked_reference(t_next) || (t.key < search_key));
        right_node = t;
        /* 2: проверяем, что элементы являются соседями */
        if (left_node_next == right_node)
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again;
            else
                return right_node;
        /* 3: удаляем физически */
        if (CAS (&(left_node.next), left_node_next, right_node))
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again;
            else
                return right_node;
    }
}
```

Функция поиска служит одновременно еще и функцией удаления, поскольку, если в какой-то момент она пройдет по маркированному элементу, то она производит физическое удаление.

Функция удаления пытается удалить элемент с заданным ключом. Удаление происходит в два этапа – логическое и физическое.

Схема работы удаления CAS-операциями изображена на рисунке 1.2

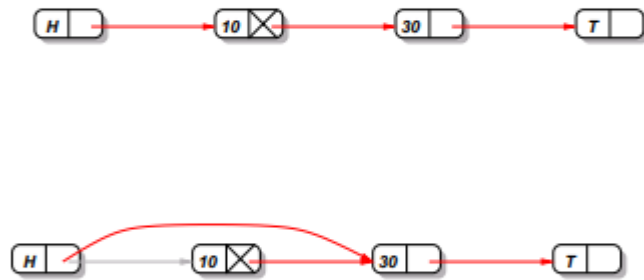


Рисунок 1.2 — Схема работы функции удаления

Псевдокод функции удаления:

```
public boolean List::delete (KeyType search_key) {
    Node *right_node, *right_node_next, *left_node;
    do {
        right_node = search (search_key, &left_node);
        if ((right_node == tail) || (right_node.key != search_key))
//элемент не найден
            return false;
        right_node_next = right_node.next;
        if (!is_marked_reference(right_node_next))
            if (CAS (&(right_node.next),
                    right_node_next, get_marked_reference (right_node_next)))
//логическое удаление
                break;
    } while (true);
    if (!CAS (&(left_node.next), right_node, right_node_next))
//физическое удаление
        right_node = search (right_node.key, &left_node);
    return true;
}
```

Функция проверки на существование элемента в списке реализована на основе функции поиска.

```
public boolean List::find (KeyType search_key) {
    Node *right_node, *left_node;
    right_node = search (search_key, &left_node);
    if ((right_node == tail) ||
        (right_node.key != search_key))
        return false;
    else
        return true;
}
```



## 1.2 Библиотека Libcds

Libcds [2] является C++ библиотекой lock-free структур данных и методов безопасного освобождения памяти. Она предлагает очень простой интерфейс и готовые решения для многопоточного проекта.

Примерный код взаимодействия с библиотекой выглядит следующим образом:

```
#include <cds/init.h> //cds::Initialize и cds::Terminate
#include <cds/gc/hp.h> //cds::gc::HP (Hazard Pointer)

int main(int argc, char** argv)
{
    // Инициализируем libcds
    cds::Initialize() ;
    {
        // Инициализируем Hazard Pointer синглтон
        cds::gc::HP hpGC ;
        cds::threading::Manager::attachThread() ;

        // Всё, libcds готова к использованию
        // Далее располагается ваш код
        ...
    }
    // Завершаем libcds
    cds::Terminate() ;
}
```

Начало работы с библиотекой происходит в вызове `cds::Initialize()`, которая инициализирует глобальные данные и определяет топологию работы системы.

Далее нам нужно объявить схему сборки мусора, т.н. `garbage collector(GC)`. Такой объект во всей программе должен быть один, и все взаимодействие с ним реализовано внутри алгоритмов контейнеров.

После этого необходимо вызвать метод `attachThread()`, чтобы подключить текущий поток к инфраструктуре библиотеки. После этого мы можем взаимодействовать с со структурами данных.

В библиотеке `libcds` реализованы два варианта структур – `invasive` и `non-invasive`. Ключевое отличие в том, что первые работают с самими

данными, а вторые копируют данные себе. В данной работе будут рассмотрены лишь non-invasive структуры.

В библиотеке реализован ряд структур данных с определенным набором опций [5].

Опции для структур данных, рассматриваемые в данной работе, включают в себя:

- SMR scheme – способ сборки мусора
- Back-off – стратегия поведения при обнаружении параллельной работы

В libcds для рассматриваемых алгоритмов реализовано 3 метода безопасного удаления памяти (SMR):

### **Hazard pointers (HP).**

Схема предложена Майклом и предназначена для защиты локальных ссылок на элементы lock-free структуры данных. Является одной из самых известных схем отложенного удаления.

Особенность в том, что у каждого потока есть локальный массив указателей HP. Прежде, чем работать с указателем на элемент, поток помещает его в этот массив, при этом заполнять массив может только поток-создатель, а читать его могут все потоки. В среднем размер массива для операций с разными структурами не превышает 5 указателей.

При удалении поток помещает нужный указатель в массив dlist. Как только массив dlist полностью заполняется, вызывается Scan(). В этой функции сначала со всех потоков собираются HP, не равные нулю, потом из массива dlist удаляются указатели, не являющиеся HP, т.е. которые можно безопасно удалить, т.к. с ними не работает ни один поток.

Псевдокод алгоритма представлен ниже.

```

// Константы
// P : число потоков
// K : число hazard pointer в одном потоке
// N : общее число hazard pointers = K*P
// R : batch size,  $R-N=\Omega(N)$ , например,  $R=2*N$ 
// Массив Hazard Pointer потока
// Пишет в него только поток-владелец
void * HP[K]
// текущий размер dlist (значения 0..R)
unsigned dcount = 0;
// массив готовых к удалению данных
void* dlist[R];
// Удаление данных
// Помещает данные в массив dlist
void RetireNode( void * node ) {
    dlist[dcount++] = node;
    // Если массив заполнен – вызываем основную функцию Scan
    if (dcount == R)
        Scan();
}
// Удаляет все элементы массива dlist, которые не объявлены
// как Hazard Pointer
void Scan() {
    unsigned i;
    unsigned p=0;
    unsigned new_dcount = 0; // 0 .. N
    void * hptr, plist[N], new_dlist[N];
    // Stage 1 – проходим по всем HP всех потоков
    // Собираем общий массив plist защищенных указателей
    for (unsigned t=0; t < P; ++t) {
        void ** pHPThread = get_thread_data(t)->HP ;
        for (i = 0; i < K; ++i) {
            hptr = pHPThread[i];
            if ( hptr != nullptr )
                plist[p++] = hptr;
        }
    }
    // сортировка hazard pointer'ов
    // удаление элементов, не объявленных как hazard
    for ( i = 0; i < R; ++i ) {
        if ( binary_search(dlist[i], plist))
            new_dlist[new_dcount++] = dlist[i];
        else
            free(dlist[i]);
    }
    //формирование нового массива отложенных элементов.
    for (i = 0; i < new_dcount; ++i )
        dlist[i] = new_dlist[i];
    dcount = new_dcount;
}

```

Схема работы НР представлена на рисунке 1.3.

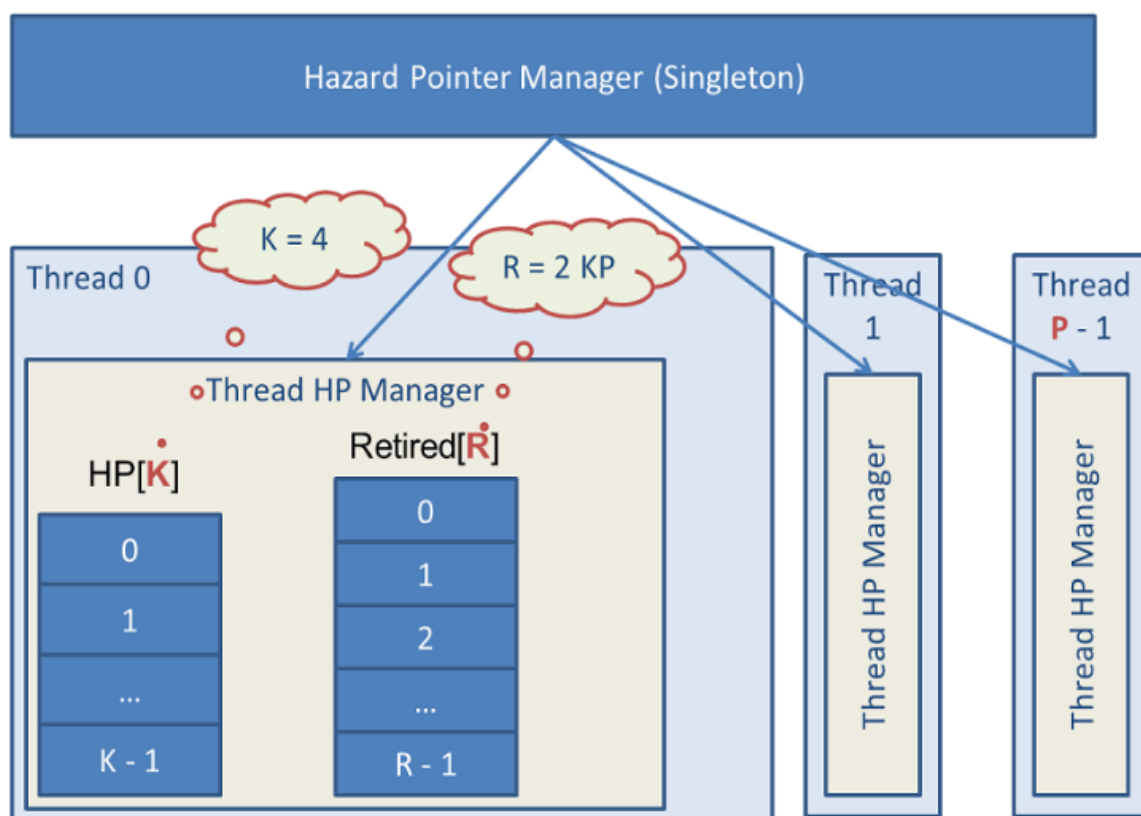


Рисунок 1.3 — Схема работы Hazard Pointer

### Dynamic hazard pointers (DHP)

Является вариантом реализации НР, но с несколькими отличиями, представленными в таблице 1.1.

Таблица 1.1 — Сравнение НР и DHP

Особенность	НР	DHP
Максимальное число НР указателей на поток	Ограничено, указывается во время инициализации	Изменяется, память выделяется, когда необходимо
Максимальное число элементов на удаление	Ограничено, указывается во время инициализации	Ограничено, изменяется, зависит от числа потоков и размера массива НР в каждом потоке
Поддерживаемое число потоков	Ограничено, верхний порог указывается при инициализации.	Нет ограничений

## User-space Read Copy Update (URCU)

Чтобы объяснить схему работы URCU, определим, что потоки, которые используют данные (читают или модифицируют), будем называть читателями, а потоки, которые удаляют данные, - писателями.

Каждый раз, когда писатель удаляет элемент из структуры, он объявляет т.н. *grace-period*—состояние системы, когда для данного элемента есть потоки, которые его «читают». Перед объявлением *grace-period*, писатель убирает все ссылки на удаляемый элемент, т.е. совершает логическое удаление, чтобы для данного элемента не появилось новых читателей. *Grace-period* длится до тех пор, пока данный элемент читает любой поток, и только после окончания этого периода данный элемент можно безопасно удалять, поскольку на него никто не ссылается.

На рисунке 1.4 представлена схема работы. При чтении элемента читатель вызывает функцию `rcu_read_lock()`, объявляя о входе в «критическую секцию чтения». Выход из критической секции объявляется соответственно функцией `rcu_read_unlock()`

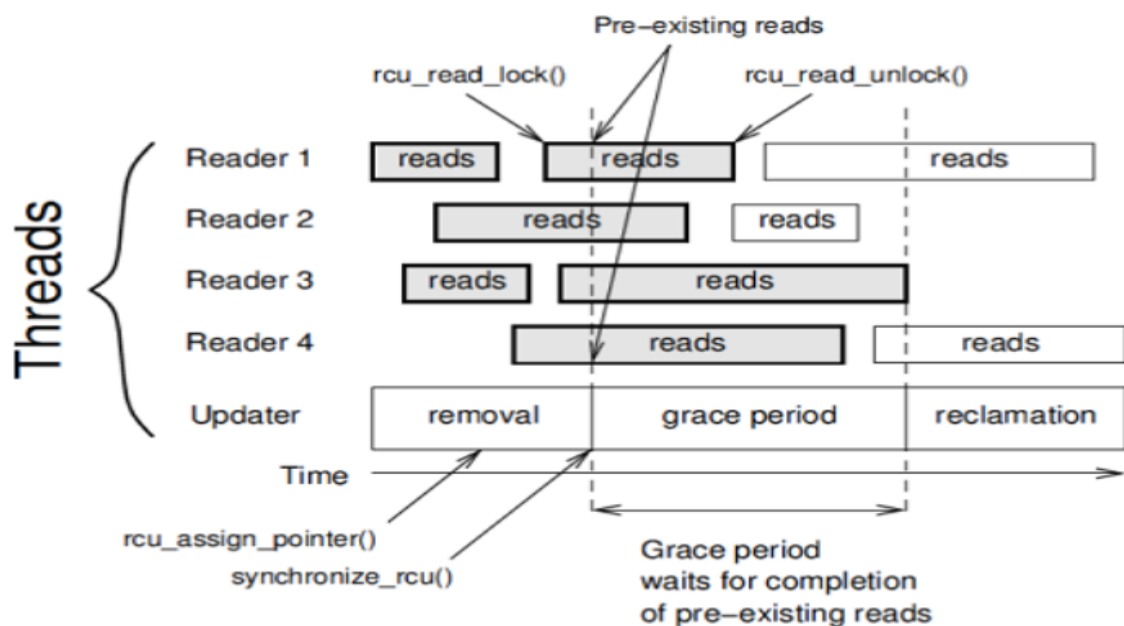


Рисунок 1.4 — Схема работы URCU

В URCU схеме представлено 5 вариантов, но на доступной платформе поддерживается всего 3:

- URCU instant – реализация полностью соответствует вышеописанному алгоритму;
- URCU buffered – элементы на физическое удаление сначала помещаются в буфер на основе lock-free очереди определенного размера. Удаление происходит при переполнении буфера;
- URCU threaded – очистка буфера выполняется еще одним потоком, который никак не взаимодействует с исходной структурой данных.

**Back-off стратегии** – опция, которая определяет поведение потока в ситуации, когда данный поток не может выполнить операцию, потому что она конфликтует с параллельными действиями другого потока. В основном это происходит во время CAS (compare-and-swap) операций, когда два потока одновременно исполняют CAS по отношению к одному элементу. В случае большой нагрузки на элемент только один поток сможет выполнить необходимую операцию, остальные будут впустую использовать ресурсы.

В libcds представлено для списков представлено два варианта back-off стратегий:

- Delay – при неудачной CAS поток входит в режим ожидания на определенный промежуток времени;
- Exponential-backoff – поток входит в режим ожидания на промежуток времени, определяемый параметрами min, max, причем каждый последующий период ожидания, начиная с min, увеличивается в два раза.

Back-off стратегии выгодны, поскольку обычно время на смену контекста гораздо больше, чем время на выполнение CAS.

В самой библиотеке мы рассмотрим два варианта структур связанных списков, а именно Michael List и Lazy List.

## **Michael List**

Список Майкла [4] является связанным отсортированным списком без повторов.

Список Майкла, как и список Харриса, основывается на двухфазном удалении, но изначально создатель алгоритма предложил свою схему сборки мусора, называемую Hazard Pointer, которая была рассмотрена ранее. Список Харриса предполагает удаление цепочек помеченных элементов, однако эта схема не подходит для Hazard Pointer, потому что при удалении элемента его сначала нужно защитить, «объявив» его hazard. Поэтому вариант списка Майкла реализовывает лишь одиночное удаление, потому что список hazard указателей для каждого потока ограничен.

## **Lazy List**

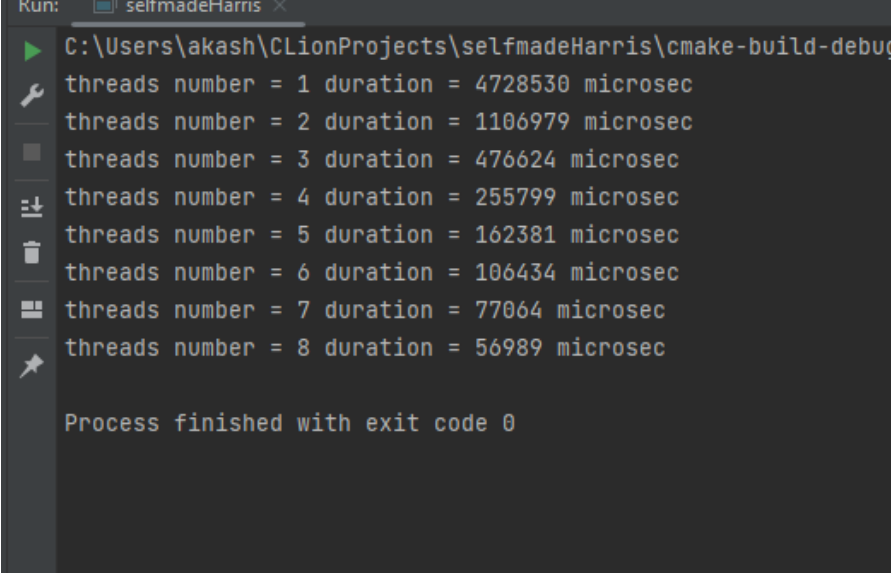
Lazy List[6] также является отсортированным списком, однако он не является в полной мере неблокирующим списком. Необходимо ввести новую терминологию: Coarse-grained locking – при модификации структуры данных доступ к ней блокируется целиком, что лишает структуру перспектив оптимизации многопоточных проектов. Fine-grained locking подразумевает, что в списке блокируется доступ только к части элементов. Lazy List использует оптимистичный подход: каждая операция, проходящая по списку, никак не блокирует данные, однако, когда она находит необходимый элемент, блокируется лишь нужный элемент и два его соседа. Такой подход позволяет просто реализовать структуру данных, но лишает ее части прироста производительности.

## 2 Практические тесты алгоритмов.

В данном разделе будут рассмотрены тесты алгоритмов в заданных условиях.

### 2.1 Алгоритм Харриса

Алгоритм [3] был реализован на языке C++ и скомпилирован в среде CLion с компилятором Clang на процессоре Intel Core i7-8565U @ 1.80ГГц. Результат показан на рисунке 2.1. В этом и дальнейших замерах времени совершалось суммарно 3200 операций вставки и удаления элементов с ключом типа `<int>`.



```
Run: selfmadeHarris X
C:\Users\akash\CLionProjects\selfmadeHarris\cmake-build-debug
threads number = 1 duration = 4728530 microsec
threads number = 2 duration = 1106979 microsec
threads number = 3 duration = 476624 microsec
threads number = 4 duration = 255799 microsec
threads number = 5 duration = 162381 microsec
threads number = 6 duration = 106434 microsec
threads number = 7 duration = 77064 microsec
threads number = 8 duration = 56989 microsec

Process finished with exit code 0
```

Рисунок 2.1 — Результат работы программы



На рисунке 2.2 представлен график зависимости времени исполнения операций от числа потоков в Excel.

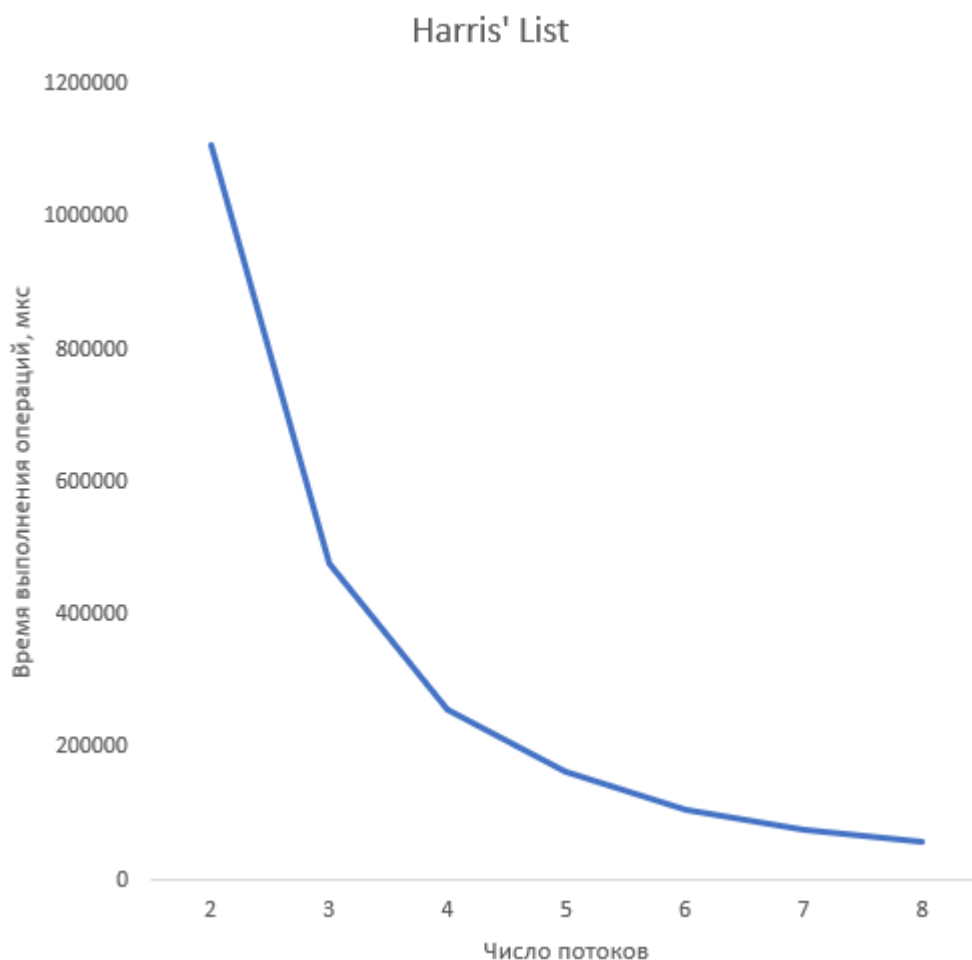


Рисунок 2.2— График зависимости времени выполнения от числа потоков

Как видно из графика, алгоритм действительно работает, и его производительность зависит от числа потоков. Простая итерация по циклу не выявила оставшихся ссылок на несуществующий элемент.

## 2.2 Тесты с библиотекой Libcds

Воспользуемся гибкой системой настройки структур Libcds [5]. Для каждого контейнера в следующих тестах применяется комбинация из трех опций, перечисленных в первом разделе, а именно:

- SMR scheme;
- Back-off strategy;

Поскольку нас интересует достижение максимальной производительности, рассматривать в данной работе мы будем только работу алгоритмов на 7-8 потоках.

Прежде чем начать сравнивать производительности контейнеров в Libcds, стоит определить параметры, в рамках которых мы проводим сравнение.

Каждый список содержит в себе ключ типа `<int>`, 3200 операций вставки и удаления.

Каждый список проходит цикл операций в 10 разных режимах 5 раз, высчитывался средний результат. Перед каждым замером все 8 потоков выполняли пустую работу, чтобы задействовать больше ресурсов процессора.

5 разных back-off стратегий:

- Exp type A задается параметрами `min=1`, `max = 64`
- Exp type B задается параметрами `min = 16`, `max = 1024`
- Delay 1 mcs задается задержкой в 1 микросекунду
- Delay 5 mcs задается задержкой в 5 микросекунд
- Delay10 mcs задается задержкой в 10 микросекунд

## Michael List

Производительность Michael List [4] с НР изображена на рисунке 2.3.

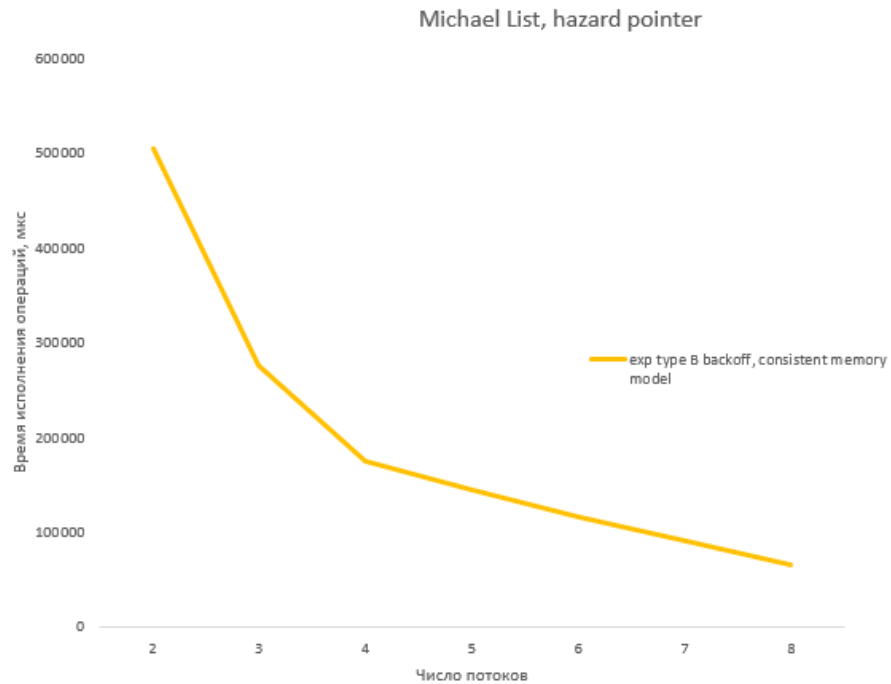


Рисунок 2.3 — График зависимости времени выполнения от числа потоков

Минимальное время выполнения составляет 64998 микросекунд

Производительность Michael List с DHP изображена на рисунке 2.4.

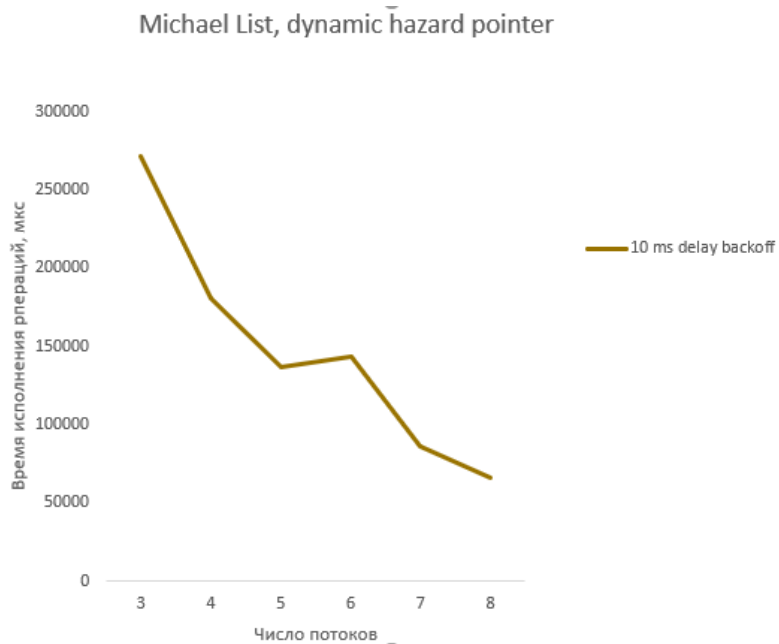


Рисунок 2.4 — График зависимости времени выполнения от числа потоков

Минимальное время выполнения составило 65433 микросекунды.

Производительность Michael List с URCU<buffered>изображена на рисунке 2.5.

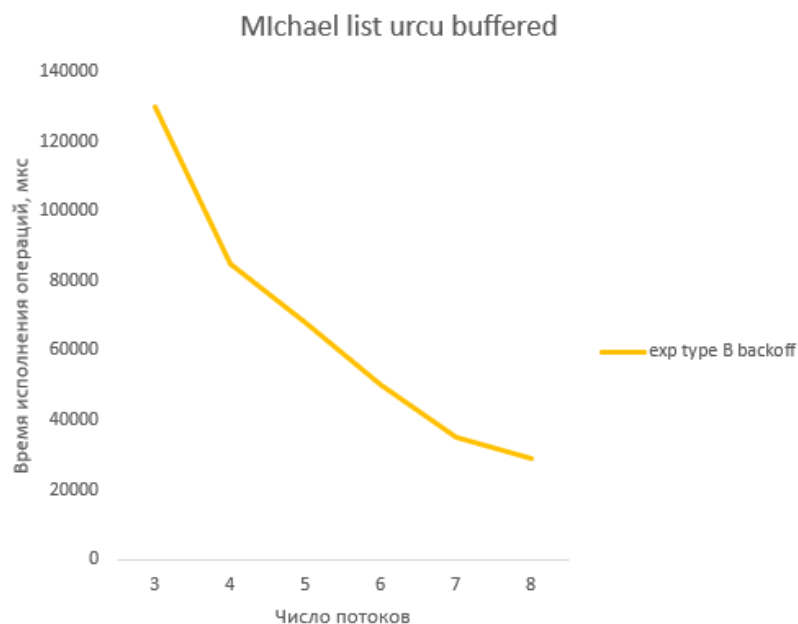


Рисунок 2.5 — График зависимости времени выполнения от числа потоков

Минимальное время выполнения составило 28213 микросекунд.

Производительность Michael List с URCU <instant>изображена на рисунке 2.6.

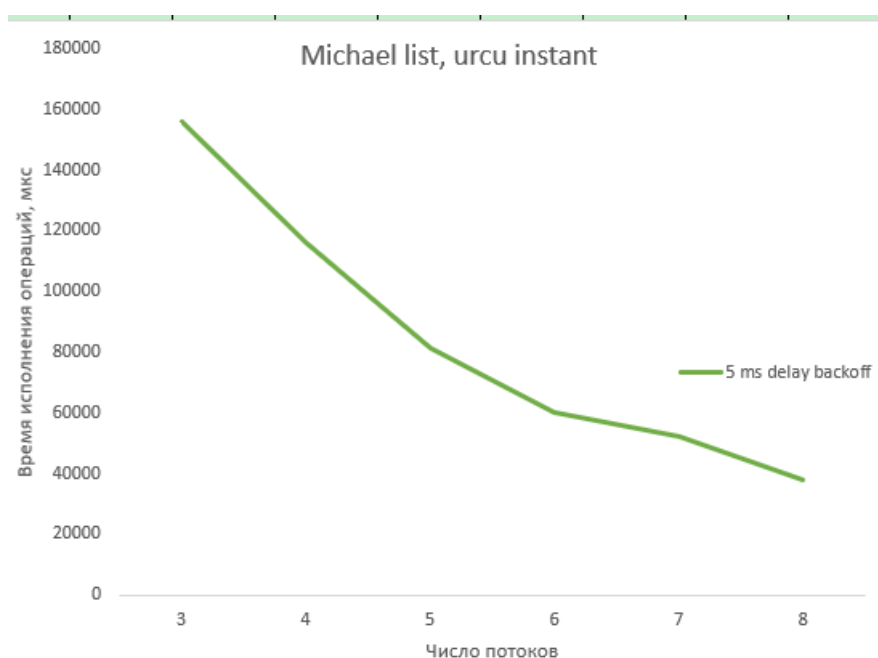


Рисунок 2.6 — График зависимости времени выполнения от числа потоков

Минимальное время выполнения составило 37535 микросекунд.

Производительность Michael List с URCU <threaded> изображена на рисунке 2.7.

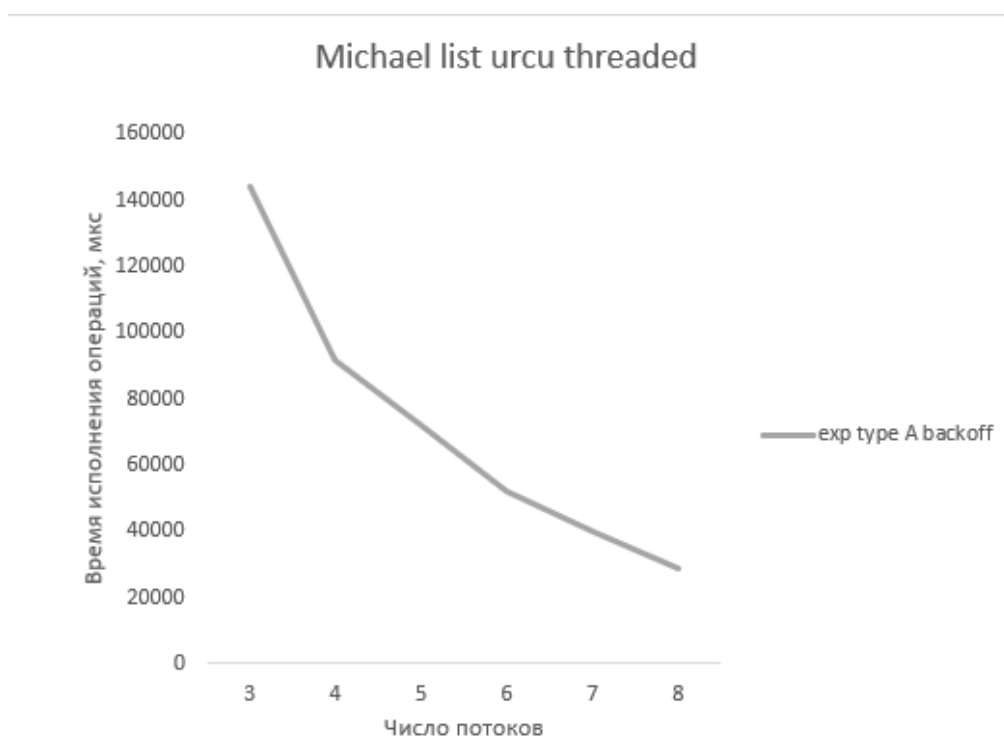


Рисунок 2.7 — График зависимости времени выполнения от числа потоков **Lazy List**

Производительность Lazy List [6] с HP изображена на рисунке 2.8

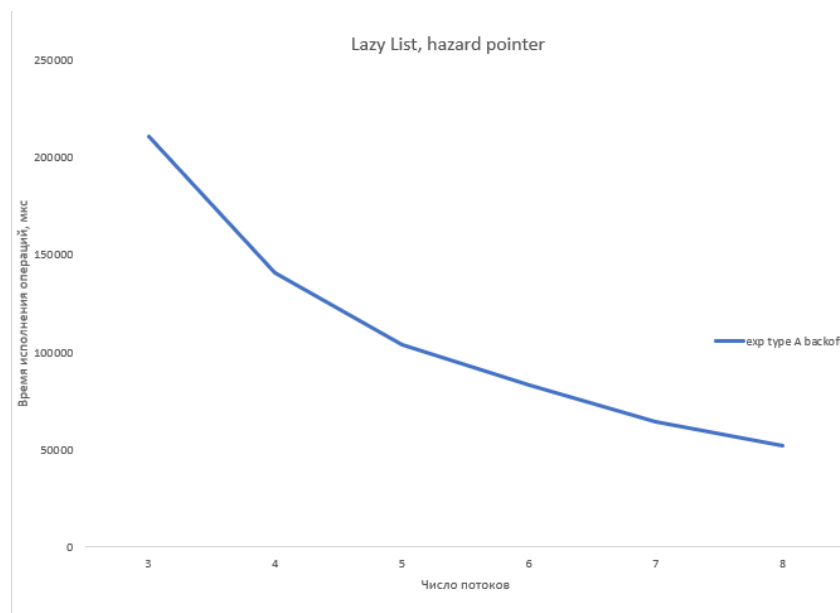


Рисунок 2.8 — График зависимости времени выполнения от числа потоков

Минимальное время выполнения составляет 52004 микросекунды

Производительность Lazy List с DHP изображена на рисунке 2.9.

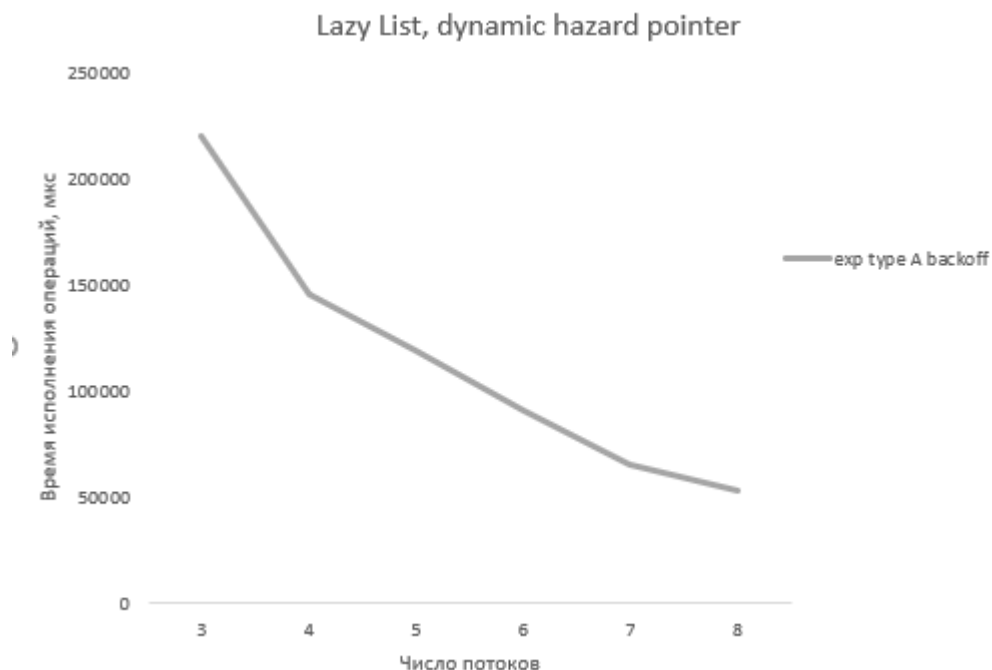


Рисунок 2.9 — График зависимости времени выполнения от числа потоков

Минимальное время выполнения составило 53631 микросекунду.

Производительность Lazy List с URCU <buffered> изображена на рисунке 2.10

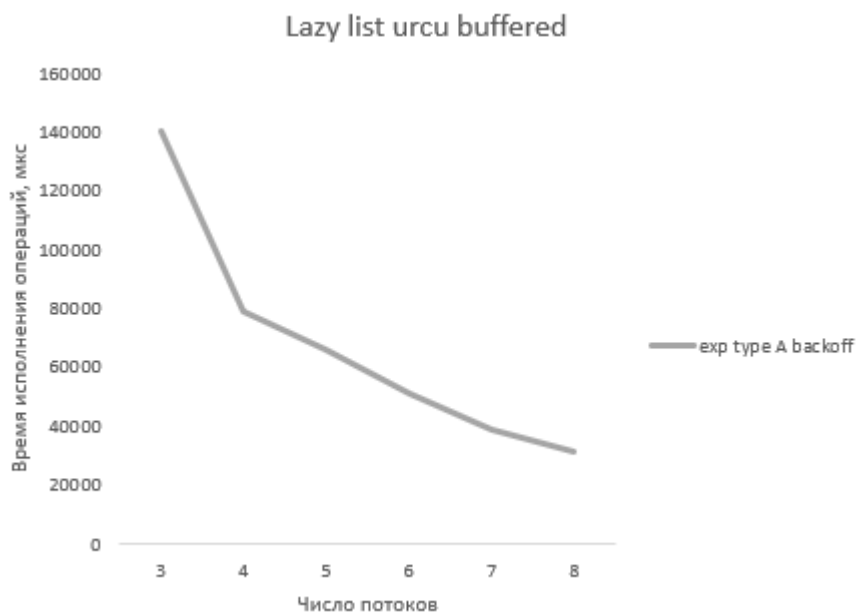


Рисунок 2.10 — График зависимости времени выполнения от числа потоков

Минимальное время выполнения составило 20091 микросекунду.

Производительность Michael List с URCU <instant> изображена на рисунке 2.11.

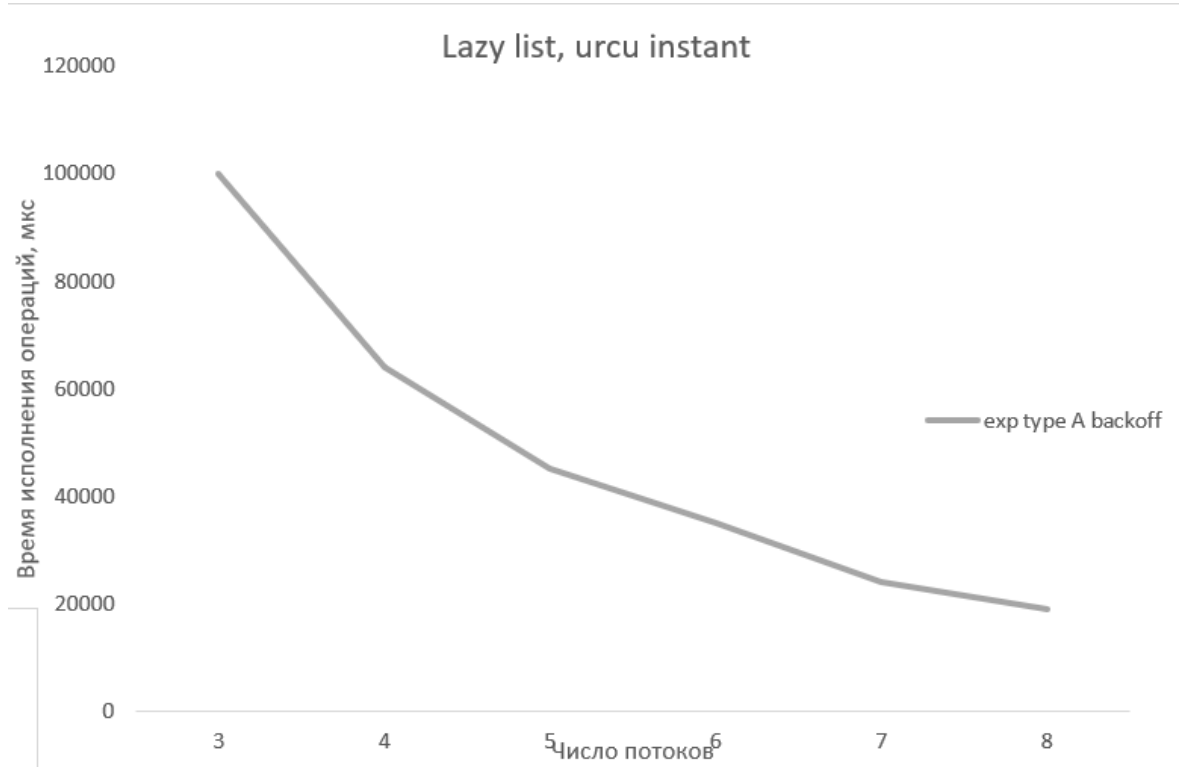


Рисунок 2.11— График зависимости времени выполнения от числа потоков  
Минимальное время выполнения составило 18799 микросекунд.

Производительность Michael List с URCU <threaded> изображена на рисунке 2.12

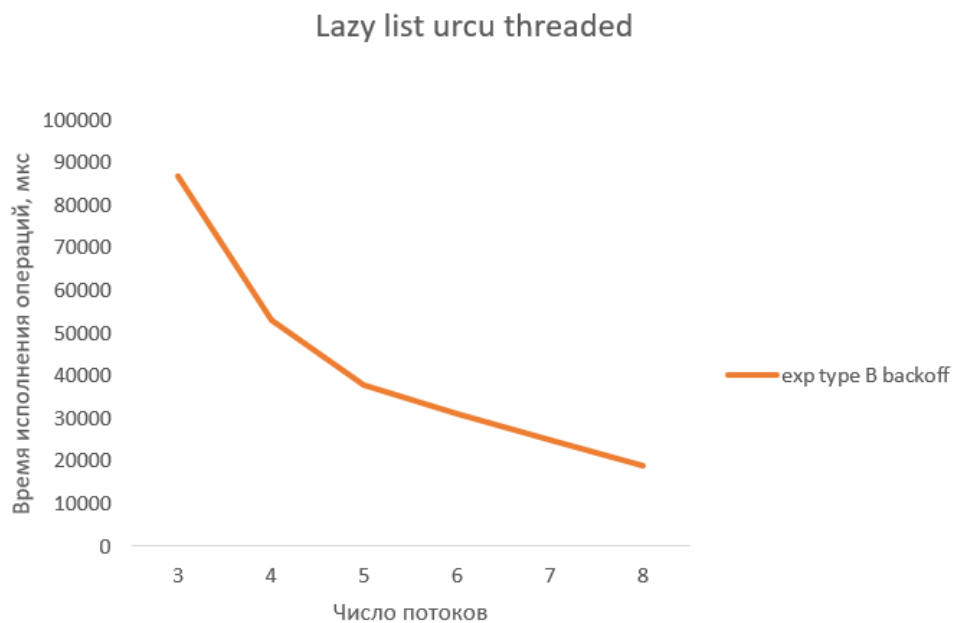


Рисунок 2.12 — График зависимости времени выполнения от числа потоков

Минимальное время выполнения составило 18990 микросекунд.

Из всех вариантов опций наилучший результат быстродействия показал Lazy List со следующими опциями:

- Backoff стратегия exponential type A
- URCU SMR схема с алгоритмом<instant>

## **2.3 Исследование методов оптимизации, применяемых к Libcds.**

Для исследования влияния оптимизаций на производительность структур повысим масштаб измерений, увеличив число операций в 10 раз до 32000.

Lazy List с URCU<instant> показал результат в 1725006 микросекунд.

Применение оптимизации показало результат по времени исполнения 32000 операций на 8 потоках в 521003 микросекунды.

Совокупность инструментов для оптимизации и разработки LLVM, и Clang [7] в частности, поддерживают широкий спектр оптимизаций исходного кода, называемый pass. Один optimization pass означает одну замену инструкций исходного кода на другую, работающую быстрее. В список optimization pass в Clang на настройке ключа компилятора «максимальной скорости» -Ofast всего было произведено 850 pass. Из них:

- 762 -inline pass: LLVM optimizer в результате профилирования применил оптимизацию инлайнинга, т.е. объединил большинство методов, вызываемых в коде, в один большой.
- 18 -gvn pass: LLVM optimizer нашел 18 «ненужных» операций присваивания, и заменил их статическими значениями.

Остальные проходы связаны с оптимизацией методов и алгоритмов, связанных с тестами, замером времени и генерацией входной последовательности.



Посмотрим исполнение программы в профайлере. На рисунке 2.13 изображена сводка использования потоков в приложении.

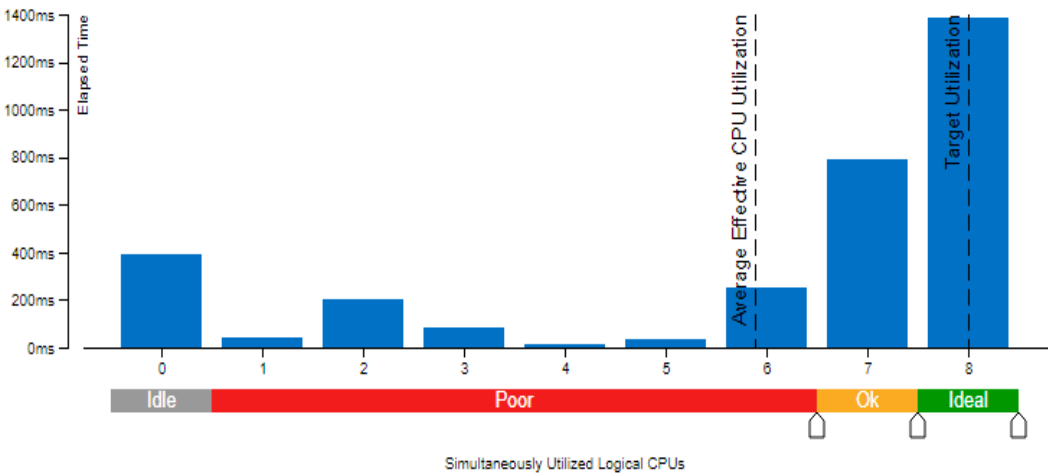


Рисунок 2.13 — График активности потоков в программа

На рисунке 2.14 изображена на временной диаграмме изображена активность каждого из потоков. Заметим, что слева расположена активность «разогревающих потоков».

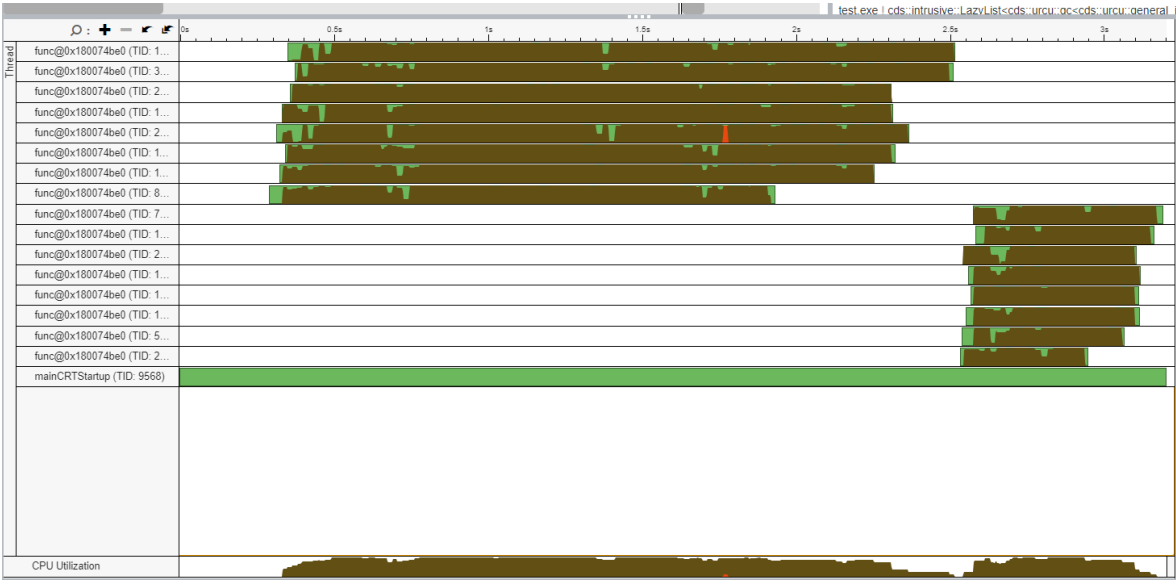


Рисунок 2.14 — Временная диаграмма активности потоков

На сводке по затратам процессорного времени, изображенного на рисунке 2.15 видно, что 14,6 секунд всего процессорного времени занимает функция сравнения.

Function	CPU Time: Total ▾ ⌵	CPU Time: Self ⌵
cds::intrusive::LazyList<cds::urcu::gc<cds::urcu::general_instant<std::mutex,cds::back	98.2%	0s
cds::intrusive::LazyList<cds::urcu::gc<cds::urcu::general_instant<std::mutex,cds::back	97.7%	2.529s
MyThreadFunctionExpAConsistentWarmUp	79.2%	0s
std::less<int>::operator()	77.5%	14.626s
cds::opt::details::make_comparator_from_less<std::less<int> >::operator()	77.5%	0s
cds::details::binary_functor_wrapper<int,cds::opt::details::make_comparator_from_less	77.5%	0s
MyThreadFunctionExpAConsistent	20.8%	0s
std::atomic<cds::details::marked_ptr<cds::intrusive::lazy_list::node<cds::urcu::gc<cds::	2.7%	0.513s
cds::details::operator==	2.7%	0.512s
cds::details::operator!=	1.4%	0.248s
free_dbg	0.9%	0.172s
operator delete	0.9%	0s

Рисунок 2.15 — Интерфейс профайлера с функциями и занимаемым ими процессорным временем

	CPU Time: Total ▾ ⌵
> >,int,my_traitsExpAConsistent>::intrusive_traits>::search	100.0%
e_accessor>::operator()	79.3%
	79.3%
	79.3%
	2.8%
	2.8%
	1.5%

Callees	CPU Time: Total ▾ ⌵	CPU
▼ cds::intrusive::LazyList<cds::urcu::gc<cds::urcu::	100.0%	
▼ cds::details::binary_functor_wrapper<int,cds::	79.3%	
▼ cds::opt::details::make_comparator_from_k	79.3%	
std::less<int>::operator()	79.3%	
▶ std::atomic<cds::details::marked_ptr<cds::intr	2.8%	
▶ cds::details::operator==	2.8%	

Рисунок 2.16 — Интерфейс профайлера с функциями и занимаемым ими процессорным временем

На рисунке 2.16 видно, что функция поиска в Lazy списке, которая на рисунке 2.15 занимает 2.529 секунд процессорного времени, 79,3% своего времени исполнения тратит на операцию сравнения `less`.

Функции, на которые в программе тестирования структуры данных тратилось больше всего времени, в основном, используют функции сравнения, которые нельзя заменить на более быстрые.

Из данных сводок видно, что структура эффективно работает с несколькими потоками. Нет ощутимых простоев потоков, но это в основном связано с простотой данных внутри структуры и выбранной конфигурацией.

Однако дальнейшие эксперименты обнаружили следующую особенность – при «неинвазивной» настройке ключа компилятора Clang-O1 время исполнения операций потоками почти не меняется. Это значит, что основные оптимизации были произведены `pass`, которые применяются при `-O1`.

Проведенные при `-O1` и `-Ofast` оптимизации в большинстве своем представляют оптимизации `-inline`. Отсутствие других оптимизаций в коде связано с тем, что в исходном коде библиотеки Libcds описаны алгоритмы, оптимизированные для разных архитектур. Кроме того, в исходном коде используются достаточно сложные алгоритмы, основанные на указателях, что делает невозможным выполнение классических приемов оптимизации, основанных на векторизации и развертке циклов (`-vectorize` и `-loop-unroll`).

## **3 Разработка и стандартизация программных средств**

В данном разделе приводится описание теоретических затрат на проект, диаграмма Ганта, описывающая процесс разработки программ и определение кода программного продукта.

### **3.1 Диаграмма Ганта**

Результатом данной работы является сравнение производительности алгоритмов, для чего был написан программный код. Чтобы контролировать ход выполнения работы и при необходимости вносить изменения в организацию выполнения работы необходимо формализовать представление совокупности необходимых работы.

Так как ВКР является небольшим проектом, то для планирования и управления ходом проекта подойдет диаграмма Ганта.

Диаграмма Ганта представляет собой визуальное представление хода работ в виде отрезков времени. Каждый этап работы (подзадача) характеризуется названием, датами начала и конца этапа работы, визуальным представлением в виде отрезка, и исполнителями. Но в данном графике нет явной корреляции между отдельной работой и объемами ресурсов для ее выполнения, что может создать препятствие при изменении графика работ.

Из-за этого недостатка будем использовать диаграмму Ганта для наглядного представления о графиках работы. На таблице 3.1 представлена диаграмма Ганта для данного проекта.

37

Номер	Работы	Начало	Конец	Временные периоды										Исполнители
				04. 04	11. 04	18. 04	25. 04	01. 05	08. 05	15. 05	22. 05	29. 05	09. 06	
1	Выбор темы	01.04	14.04	=====										Кашин А. А.
2	Изучение материала	15.04	25.04	=====										Кашин А. А.
3	Изучение алгоритма	25.04	31.04	=====										Кашин А. А.
4	Реализация первого алгоритма	01.05	08.05	=====										Кашин А. А.
5	Отлаживание программы	08.05	14.05	=====										Кашин А. А.
6	Написание теории	15.05	17.05	=====										Кашин А. А. Пазников А. А.
7	Изучение библиотеки libcds	17.05	24.05	=====										Кашин А. А.
8	Отладка программы для тестов	25.05	26.05	=====										Кашин А. А.
9	Прогон тестов	27.05	30.05	=====										Кашин А. А.
10	Написание практической части	01.06	03.06	=====										Кашин А. А.
11	Оформление отчета	03.06	06.06	=====										Кашин А. А. Пазников А. А.. Косухина М. А.
12	Подготовка диплома к сдаче	06.06	09.06	=====										Кашин А. А.

### 3.2 Расчёт затрат на выполнение проекта, расчет цены проекта и цены продукта.

Для расчета затрат на выполнение ВКР принимаются условия:

- коэффициент загрузки исполнителя ВКР (студента) равен 1 ( $K_{\text{загр.испол.}} = 1$ )
- коэффициент загрузки соисполнителя (руководителя) равен 0.05 ( $K_{\text{загр.рук.}} = 0.05$ )
- коэффициент загрузки консультанта по дополнительному разделу равен 0.04 ( $K_{\text{загр.конс.}} = 0.04$ )

Таким образом, цена проекта может быть рассчитана по формуле 1.1

$$C_{\text{пр}} = \sum_{i=1}^n C_{\text{полн } i} T_i K_{\text{загр } i} \quad (1.1),$$

где:

- $C_{\text{полн } i}$  - полная дневная стоимость работы  $i$  – го специалиста  
▪ [руб./день]
- $T_i$  – время участия  $i$  – го специалиста в работе над проектом [дней]
- $K_{\text{загр } i}$  – коэффициент загрузки  $i$  – го специалиста работами в проекте
- $n$  – число специалистов, занятых в проекте.

При выполнении расчетов используются следующие обозначения и расчетные соотношения:

- $Z_{зп}$  – средняя зарплата работника (специалиста) [руб./месяц], для студента принимается как одна четвертая от среднего показателя для специалиста [9].

- $T_{ср}$  – среднемесячное число рабочих дней (в период выполнения проекта);

- $Z_d$  – тарифная дневная ставка работника [руб./день];

$$Z_d = Z_{зп} / T_{ср}$$

- $\Phi$  – процент (доля) страховых взносов, исчисляемых от фонда заработной платы, 30,2%;

- $C_{сд}$  – величина страховых взносов на работника в день [руб./день]

$$C_{сд} = Z_d \times \Phi$$

- $Z_{дс}$  – дневная оплата работника с учетом страховых взносов [руб./день]

$$Z_{дс} = Z_d + C_{сд} = Z_d \times (1 + \Phi)$$

- $H$  – процент (доля) накладных расходов. Процент накладных расходов для ЛЭТИ условно принимается 42%[1] ;

- $C_{нр}$  – накладные расходы на одного работника в день [руб./день];

$$C_{нр} = Z_d \times H$$

- $C_{ч/д}$  – стоимость человека/дня [руб./день];

$$C_{ч/д} = Z_{дс} + C_{нр} = Z_d \times (1 + \Phi + H)$$

- $\Pi$  – доля (процент) прибыли (средняя прибыль), 15%;

- $C_{прд}$  – дневная прибыль на одного работника [руб./день];

$$C_{прд} = C_{ч/д} \times \Pi$$

- $C_{дсс}$  – дневная ставка специалиста (без учета НДС) [руб./день];

$$C_{дсс} = C_{ч/д} + C_{прд} = C_{ч/д} \times (1 + \Pi)$$

- НДС – ставка налога на добавленную стоимость;

Сведем все данные в таблицы 3.3 и 3.4.

Таблица 3.3 – Полные затраты в день для студента

Наименование статей	ед. изм.	нормативы/ затраты	Примечание
Величина среднемесячной начисленной заработной платы специалиста (Нс1) [11]	руб./месяц	40 000,00	Оклад сотрудника Zзп
Среднемесячное количество рабочих дней (Тср) [10]	дней./месяц	20,58	Среднее за 2023 год
Расчет ставки специалиста в день:			
Тарифная ставка дневная (Нс)	руб./день	1 943,63	$Z_d = Z_{зп} / T_{ср}$
Страховые взносы 30,2% от суммы зарплаты работников	руб./день	586,97	$C_{сд} = Z_d \times \Phi$
Оплата основных работников с со страховыми взносами (Zдс)	руб./день	2 530,60	$Z_{дс} = Z_d + C_{дс} = Z_d \times (1 + \Phi)$
Накладные расходы (Снр )	руб./день	816,32	$C_{нр} = Z_d \times H$
Себестоимость одного человек/дня (Сч/д)	руб./день	3 346,92	$C_{ч/д} = Z_{дс} + C_{нр} = Z_d \times (1 + \Phi + H)$
Дневная прибыль (Спрд)	руб./день	502,03	$C_{прд} = C_{ч/д} \times \Pi$
Ставка специалиста без учета НДС (Сдсс)	руб./день	3 848,95	$C_{дсс} = C_{ч/д} + C_{прд}$
Дневная сумма НДС (Сндс)	руб./день	769,79	$C_{ндс} = C_{дсс} \times H_{дс}$
Ставка специалиста в день с учётом НДС (Сполн)	руб./день	4 618,74	Сполн



Таблица 3.4 – Полные затраты в день для руководителя и консультанта по дополнительному разделу

Наименование статей	ед. изм.	нормативы/ затраты	Примечание
Величина среднемесячной начисленной заработной платы специалиста (Нс1)	руб./месяц	160 000,00	Оклад сотрудника Ззп
Среднемесячное количество рабочих дней (Тср)	дней./месяц	20,58	Среднее за 2023 год
Расчет ставки специалиста в день:			
Тарифная ставка дневная (Нс)	руб./день	7 774,53	$Z_d = Z_{зп} / T_{ср}$
Страховые взносы 30,2% от суммы зарплаты работников	руб./день	2 347,91	$C_{сд} = Z_d \times \Phi$
Оплата основных работников со страховыми взносами (Zдс)	руб./день	10 122,43	$Z_{дс} = Z_d + C_{сд} = Z_d \times (1 + \Phi)$
Накладные расходы (Снр )	руб./день	3 265,30	$C_{нр} = Z_d \times H$
Себестоимость одного человек/дня (Сч/д)	руб./день	13 387,73	$C_{ч/д} = Z_{дс} + C_{нр} = Z_d \times (1 + \Phi + H)$
Дневная прибыль (Спрд)	руб./день	2 008,15	$C_{прд} = C_{ч/д} \times \Pi$
Ставка специалиста без учета НДС (Сдсс)	руб./день	15 395,88	$C_{дсс} = C_{ч/д} + C_{прд}$
Дневная сумма НДС (Сндс)	руб./день	3 079,17	$C_{ндс} = C_{дсс} \times H_{ндс}$
Ставка специалиста в день с учётом НДС (Сполн)	руб./день	18 475,06	Сполн

Исходя из данных в таблицах можно рассчитать полную цену проекта по формуле 1.1.

$$C_{\text{пр}} = (4\,618,74 \text{ руб.} \cdot 30 \cdot 1) + (18\,475,06 \text{ руб.} \cdot 10 \cdot 0,07) + (18\,475,06 \text{ руб.} \cdot 2 \cdot 0,05) = 153\,342,24 \text{ руб.}$$

Данный проект пока что не рассчитан на вывод его на рынок, поэтому делать расчет цены программы не имеет смысла

### 3.3 Обеспечение качества программного продукта

При разработке программного продукта должны быть предусмотрены мероприятия по обеспечению его качества, в данном случае таким мероприятием является отладка программы. Отладка подразумевает нахождение ошибок в логике программы путем ее запуска и сверки результатов. Наглядно процесс обеспечения качества продукта представлен на рисунке 4.3.

Работа по отладке является цикличной, поэтому будем рассмотрим цикл Демминга.

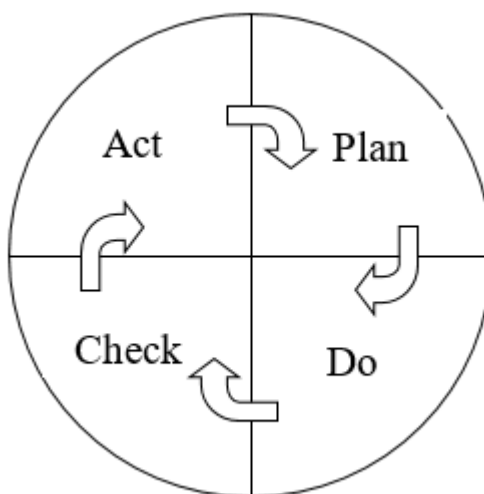


Рисунок 4.3 — цикл Деминга

В рамках цикла Демминга предусмотрен следующий порядок действий:

- P – (Plan) – установить конечную цель и определить ресурсы для достижения, в ВКР целью будет написание программного кода, который можно использовать в других проектах;
- D – (Do) – внедрение процессов, на данном этапе будет происходить написание блока в соответствии с целями;
- C – (Check) – контроль (мониторинг) кода, здесь происходит проверка кода на правильность работы;
- A – (Act) – отладка программного кода.

### **3.4 Определение кода программного продукта**

В данной работе был написан код для прикладных задач, т.е. по классификации ОКПД2 ВКР подпадает под 62.01.11 [9] — услуги по проектированию, разработке информационных технологий для прикладных задач и тестированию программного обеспечения.

По классификации ОКП ВКР подпадает под 5012309 [9] — Программные средства организации и обслуживания вычислительного процесса.

### **3.5 Определение списка стандартов**

При написании данной работы не планировалось выпускать продукт на рынок, поэтому код программы не соответствует каким-либо стандартам, но при оформлении ВКР стоит обратить внимание на следующие ГОСТы: ГОСТ 19.701-90, ГОСТ 19.102-77, ГОСТ 19.201-78, ГОСТ 19.301-79, ГОСТ 19.401-78 [9].

## ЗАКЛЮЧЕНИЕ

В работе приведен обзор одной из основных неблокирующих структур.

Приведена реализация структуры, код отлажен.

Приведен обзор и принцип работы готовых неблокирующих структур, описаны варианты их модификации. Выполнено сравнение скорости работы структур в заданных условиях.

Выяснилось, что реализованный алгоритм Харриса уступает в производительности алгоритмам из Libcds, среди которых наилучшую производительность показал Lazy List с методом SMR - URCU Instant.

Результаты работы в виде обзора ключевых опций готовых структур и принципов их работы можно использовать в разработке новых структур данных, а опыт сравнения алгоритмов на реальном примере поможет выбрать оптимальную структуру для рассмотренной платформы и задач. Исследование оптимизации на примере компилятора Clang для алгоритмов Libcds позволит разработчикам многопоточных приложений эффективнее применять неблокирующие структуры данных.

Дополнительный раздел разработка и стандартизация программных средств позволил оценить затраты на дальнейшую разработку структур данных, упростить процесс разработки и дал представления о необходимых условиях для выпуска продукта на рынок.

Дальнейшие исследования по данной теме могут затронуть дальнейшее улучшение производительности списка Харриса или сравнение lock-based алгоритмов.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Пазников А. А., Табаков // Распределенная очередь с ослабленной семантикой выполнения операций в модели удаленного доступа к памяти// Вестник Томского государственного университета. Управление, вычислительная техника и информатика. 2020.
2. Серия статей автора Libcds // Хабр [Электронный ресурс]. URL: <https://habr.com/ru/articles/195770/> (дата обращения: 04.04.2023)
3. Timothy L. Harris // A pragmatic implementation of non-blocking linked-lists = Реализация неблокирующих связанных списков. // University of Cambridge. 2001.
4. Maged M. Michel, Michael L. Scott. // Simple, Fast, and Practical Non Blocking Concurrent Queue Algorithms = Простые, быстрые и практичные алгоритмы конкурентной очереди // In PODC.ACM. 1995.
5. CDS 2.3.2 Documentation // Документация по библиотеке CDS [Электронный ресурс]. URL: <https://libcds.sourceforge.net/doc/cds-api/index.html> (дата обращения: 05.04.2023)
6. S. Heller // A lazy concurrent list-based set algorithm = Наивный потокобезопасный алгоритм структуры данных на списках. //University of Rochester2006
7. Clang // Официальный сайт «Clang» [Электронный ресурс]. URL: <https://clang.llvm.org/> (дата обращения: 15.05.2023)
8. JetBrains CLion // Официальный сайт «JetBrains CLion» [Электронный ресурс]. URL: <https://www.jetbrains.com/ru-ru/clion/> (дата обращения: 15.05.2023)
9. Фомин В. И. Разработка и стандартизация программных средств. Учебное пособие. СПб: Издательство СПбУУЭ, 2020. 35 с.
10. Количество дней (календарных/рабочих/выходных и праздничных) и нормы рабочего времени в 2023 году. [Электронный ресурс] // КонсультантПлюс: [сайт]. [2023]. URL: [https://www.consultant.ru/document/cons\\_doc\\_LAW\\_419959/c69042fcbbcf86817f8871212f4df28bf268c0b7/](https://www.consultant.ru/document/cons_doc_LAW_419959/c69042fcbbcf86817f8871212f4df28bf268c0b7/) (дата обращения: 22.05.2023)
11. Зарплаты в ИТ. [Электронный ресурс] // Хабр-Карьера: [сайт]. [2023]. URL: <https://career.habr.com/salaries> (дата обращения: 22.05.2023)

## ПРИЛОЖЕНИЕ А

### Код программы алгоритма Харриса

// программа, реализующая связный список Харриса

```
#include<iostream>
#include <vector>
#include <thread>
#include <random>
#define BOOST_ALL_DYN_LINK
#define BOOST_ALL_NO_LIB
#define BOOST_CHRONO_HEADER_ONLY
#include<boost/chrono.hpp>
#define NUM_OPER 3200
template <typename KeyType> class Node {
public:
    KeyType key;
    Node *next;
    Node() = default;
    explicit Node (KeyType key) {
        this->key = key;
    }
};

template <typename KeyType> class List {
public:
    Node<KeyType> *head;
    Node<KeyType> *tail;
    List() {
        head = new Node<KeyType> ();
        //An instance of the List class contains two fields which
        //identify the head and tail = new Node<KeyType>the tail.
        //Instances of Node contain two fields identifying the key
        and //successor of the node.
        head->next = tail;
        tail->next = nullptr;
    };
    bool is_marked_reference(Node<KeyType> *p) {
        return (uint64_t)p & 0x15;
    }
    Node<KeyType> * get_unmarked_reference(Node<KeyType> *p) {
        return (Node<KeyType>*)( (uint64_t)p & ~0x1);
    }
    Node<KeyType> * get_marked_reference( Node<KeyType> *p) {
        return (Node<KeyType>*)( (uint64_t)p | 0x1);
    }
    bool MyInsert (KeyType key) {
        Node<KeyType> *new_node = new Node<KeyType>(key);
        //The List::MyInsert method attempts to insert a new node
        with the supplied key.
        Node<KeyType> *right_node, *left_node;
        do {
            right_node = MySearch (key, &left_node);
            //The List::search operation finds the left
            //and right nodes for a particular search key
            if ((right_node != tail) && (right_node->key == key))
                return false;
            new_node->next = right_node;
        } while (true);
    }
};
```

```

        if (__sync_bool_compare_and_swap(&(left_node->next),
                                         right_node, new_node))
            //__sync_bool_compare_and_swap(addr,o,n) is a CAS
            //operation that atomically compares the contents
            //of addr against the old value o and if they
            //match writes n to that location.
            //__sync_bool_compare_and_swap returns a boolean
            //indicating whether this update took place.
            return true;
    } while (true); //B3
}

//List::MyInsert uses List::MySearch to locate the pair of
//nodes between which
//the new node is to be inserted. The update itself takes
//place with a single CAS
//operation (C2) which swings the reference in left
//node.next from right node
//to the new node.
bool MyDelete (KeyType search_key) {
    Node<KeyType> *right_node, *right_node_next, *left_node;

    //The List::MyDelete method attempts to remove a
    //node containing the supplied key.
    do {
        right_node = MySearch (search_key, &left_node);
        if ((right_node == tail) || (right_node->key !=
                                     search_key))
            return false;
        right_node_next = right_node->next;
        if (!is_marked_reference(right_node_next))

            //The reference contained in the next field of a
            //node may be in //one of two
            //states: marked or unmarked.
            //Intuitively a marked node
            //is one which should be ignored because some
            //process is deleting it. The function is marked
            //reference(r) returns true if and only if r is a
            //marked reference.
            //Similarly get_marked_reference(r) and
            //get_unmarked_reference(r) convert
            //between marked and unmarked references.

            if(__sync_bool_compare_and_swap(&(right_node
                                             ->next),right_node_next, get_marked_reference
                                             (right_node_next)))
                MySearch(right_node->key, &left_node);
            break;
    } while (true);
    if (!__sync_bool_compare_and_swap(&(left_node->next),
                                     right_node, right_node_next)){
        right_node = MySearch (right_node->key, &left_node);
        // for(int it = 0; it<12300000;it++){
    }
    return true;
}

```

```

        //List::MyDelete uses List::MySearch to locate the node to
        //delete and then uses
        //a two-stage process to perform the deletion. Firstly, the
        //node is logically deleted
        //by marking the reference contained in right node.next
        //Secondly, the node
        //is physically deleted. This may be performed directly (C4)
        //or within a separate
        //invocation of MySearch.
    }
bool MyFind (KeyType search_key) {
    Node<KeyType>*right_node, *left_node;
    //The List::MyFind method tests whether the list contains a
    node with
    //the supplied key
    right_node = MySearch (search_key, &left_node);
    if ((right_node == tail) ||
        (right_node->key != search_key))
        return false;
    else
        return true;
}
private:
Node<KeyType> *MySearch (KeyType search_key,
                        Node <KeyType>*&left_node) {
    Node<KeyType> *left_node_next, *right_node;

    //List::MySearch takes a search key and returns references
    //to two nodes called the left node and right node for that
    //key. The method ensures that these nodes satisfy a number
    //of conditions. Firstly, the key of the left node must be
    //less than the search key and the key of the right node
    //must be greater than or equal to the search key. Secondly,
    //both nodes must be unmarked. Finally, the right node
    //must be the immediate successor of the left node.

search_again:
    do {
        Node<KeyType> *t = head;
        Node<KeyType> *t_next = head->next;
        // 1: Find left_node and right_node
        do {
            if (!is_marked_reference(t_next)) {
                (*left_node) = t;
                left_node_next = t_next;
            }
            t = get_unmarked_reference(t_next);
            if (t == tail)
                break;
            t_next = t->next;
        } while (is_marked_reference(t_next) || (t->key < search_key));
        right_node = t;
        // 2: Check nodes are adjacent
        if (left_node_next == right_node)
            if ((right_node != tail) &&
                is_marked_reference(right_node->next))
                goto search_again;
            else
                return right_node;
    }

```



```

        else{}
        // 3: Remove one or more marked nodes
        if (__sync_bool_compare_and_swap(&(*left_node)-
>next, left_node_next, right_node))
            if ((right_node != tail) & is_marked_refer-
ence(right_node->next)){
                // delete right_node;
                goto search_again;
            }
        else
            return right_node;
    } else{}
} while (true);
}
};
//The List::MyFind invokes List::MySearch and
//examines the resulting right node.

int intRand(const int& min, const int& max);
template <typename KeyType> int MyThreadFunction(int
num_threads, List<KeyType> * cont);
//шаблоны для тестов
template <typename KeyType> int InitThreads( int num_threads,
List<KeyType>* cont);
int intRand(const int& min, const int& max) {
    static thread_local std::mt19937 generator;
    std::uniform_int_distribution<int> distribution(min, max);
    return distribution(generator);
}
template <typename KeyType> int MyThreadFunction(const int
num_threads, List< KeyType >* cont) {
    for (int a = 0; a < NUM_OPER / num_threads; a++) {
        cont->MyInsert(intRand(10, NUM_OPER * 10));
        cont->MyDelete(intRand(10, NUM_OPER * 10));
    }
    return 0;
}
template <typename KeyType> int InitThreads(const int
num_threads, List< KeyType>* cont) {
    std::vector<std::thread > th_vec;
    boost::chrono::steady_clock::time_point start =
boost::chrono::steady_clock::now();
    for (int i = 0; i < num_threads; i++)
    {
        th_vec.push_back(std::thread(MyThreadFunction<KeyType>,
num_threads, cont));
    }
    for (int i = 0; i < num_threads; i++) {
        th_vec[i].join();
    }
    boost::chrono::steady_clock::time_point stop =
boost::chrono::steady_clock::now();
    std::cout <<"threads number = " << num_threads << " duration
= "
<< boost::chrono::duration_cast<boost::chrono::microsec-
onds>(stop - start).count()
<< " microsec\n";
    return 0;
}

```

```
}  
int main() {  
    for (int counter = 1; counter <= 8; counter++) {  
        List<int> MyList;  
        int k = 0;  
        InitThreads(counter,&MyList);  
    }  
    return 0;  
}
```

## Приложение Б.

### Код программы тестов Libcds

```
//код программы, где тестируются различные конфигурации структур
//данных из libcds
#define BOOST_ALL_NO_LIB
#define BOOST_CHRONO_HEADER_ONLY
#include <iostream>
#include<random>
#include <vector>
#include<thread>
#include<boost/chrono.hpp>
#include <cds/init.h>
#include <cds/urcu/general_instant.h>
#include <cds/container/lazy_list_rcu.h>
#include<cds/threading/details/wintls.h>
#include<cds/algo/atomic.h>
#include <cds/algo/backoff_strategy.h>
#include <cds/opt/options.h>
#define NUM_OPER 3200
namespace cc = cds::container;
namespace bkoff = cds::backoff;
//структура back-off
struct traits_A : public bkoff::exponential_const_traits
{
    static size_t lower_bound;
    static size_t upper_bound;
};
size_t traits_A::lower_bound = 16;
size_t traits_A::upper_bound = 512;
typedef bkoff::exponential<traits_A> expBackOffA;
typedef struct cds::opt::memory_model<cds::opt::v::relaxed_ordering> sequential_consistent;
struct my_compare {
    int operator()(int i1, int i2)
    {
        return i1 - i2;
    }
};
//структура traits для контейнера
struct my_traitsExpAConsistent : public cds::container::lazy_list::traits
{
    sequential_consistent seq;
    typedef cds::opt::back_off<expBackOffA> backoff;
};
typedef cds::urcu::gc< cds::urcu::general_instant<>> rcu_gpb;
typedef cds::container::LazyList< rcu_gpb, int, my_traitsExpAConsistent > traits_based_listExpAConsistent;
int InitThreadsExpAConsistent(int num_threads,
traits_based_listExpAConsistent* cont);
int MyThreadFunctionExpAConsistent(const int num_threads,
traits_based_listExpAConsistent* cont);
int intrRand(const int& min, const int& max);
```

```

//потокобезопасная функция генерации чисел
int intrand(const int& min, const int& max) {
    static thread_local std::mt19937 generator;
    std::uniform_int_distribution<int> distribution(min, max);
    return distribution(generator);
}

int MyThreadFunctionExpAConsistent(const int num_threads,
traits_based_listExpAConsistent* cont)
{

    if (!cds::threading::Manager::isThreadAttached())
        cds::threading::Manager::attachThread();
    for (int a = 0; a < NUM_OPER / num_threads; a++) {
        cont->insert(intrand(10, NUM_OPER *1000 ));
        cont->insert(intrand(10, NUM_OPER *1000 ));
    }

    //      std::cout << cont->size() << std::endl;
    cds::threading::Manager::detachThread();
    return 0;
}
//функции для разогрева мощностей процессора
int MyThreadFunctionExpAConsistentWarmUp(const int num_threads,
traits_based_listExpAConsistent* cont)
{
    if (!cds::threading::Manager::isThreadAttached())
        cds::threading::Manager::attachThread();
    for (int a = 0; a < 64000/ num_threads; a++) {
        cont->insert(intrand(10, NUM_OPER * 1000));
        cont->insert(intrand(10, NUM_OPER * 1000));
    }
    cds::threading::Manager::detachThread();
    return 0;
}

//основная функция с потоками, где засекается время
int InitThreadsExpAConsistent(const int num_threads,
traits_based_listExpAConsistent* cont) {
    std::cout << "\nwarming up.." << std::endl;
    std::vector<std::thread > th_vec;
    for (int i = 0; i < num_threads; i++) {
        th_vec.push_back(std::thread(MyThreadFunctionExpA-
ConsistentWarmUp, num_threads, cont));
    }
    for (int i = 0; i < num_threads; i++) {
        th_vec[i].join();
    }
    th_vec.clear();
    cont->clear();
    std::cout << "\nbenchmark start" << std::endl;
    boost::chrono::system_clock::time_point start =

```

```

boost::chrono::system_clock::now();
    for (int i = 0; i < num_threads; i++)
    {
        th_vec.push_back(std::thread(MyThreadFunctionExpACon-
sistent, num_threads, cont));
    }
    for (int i = 0; i < num_threads; i++) {
        th_vec[i].join();
    }
    th_vec.clear();
    cont->clear();
    boost::chrono::system_clock::time_point stop =
boost::chrono::system_clock::now();
    std::cout<< boost::chrono::duration_cast<boost::chrono::mi-
croseconds>(stop - start).count()<< "\n";
    return 0;
}
//объявление числа потоков
namespace MyEnum
{
    enum Type
    {

h = 8
    };

    static const Type All[] = {
h };
}
int main() {
    cds::Initialize();
    {
//основная часть взаимодействия с библиотекой
        rcu_gpb GC;
        cds::threading::Manager::attachThread();
        traits_based_listExpAConsistent MyListExpAConsistent;
        std::cout << "\nExp A backoff runtime support
                                " << std::endl;
        for (const auto enum_iter : MyEnum::All) {
            InitThreadsExpAConsistent(enum_iter,
                                        &MyListExpAConsistent);
            MyListExpAConsistent.clear();
        }
        cds::threading::Manager::detachThread();
        cds::Terminate();
        return 0;
    }
}

```