**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 разделяемых структур данных в модели MPI RMA”

Студент _____ Е. В. Епифанцев

Руководитель
к. т. н., доцент _____ А. А. Пазников

Консультант по экономическому
Обоснованию _____ Т. Н. Лебедева

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

Санкт-Петербург
2023 г.

Направление: 09.03.01 “Информатика и вычислительная техника”
Профиль: “Вычислительные машины, комплексы, системы и сети”
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 Е. В. Епифанцев Группа № 9305

**1. Тема** Реализация разделяемых структур данных в модели MPI RMA
*(утверждена приказом № _____ от _____)*

Место выполнения ВКР: Кафедра ВТ

**2. Объект и предмет исследования:**
Объект исследования – разделяемые структуры данных. Предмет исследования – связный список с использованием блокировок, неблокирующая очередь Майкла и Скотта и неблокирующей стек Трайбера.

**3. Цель:**
Реализация разделяемого связного списка, очереди Майкла и Скотта и стека Трайбера в модели MPI RMA.

**4. Исходные данные:**
Научные статьи с описаниями алгоритмов, стандарт MPI.

**5. Содержание:**
Описание разделяемых структур данных, описание модели MPI RMA, реализация связного списка с использованием блокировок, реализация неблокирующей очереди Майкла и Скотта, реализация неблокирующего стека Трайбера.

**6. Технические требования:**
Структуры данных должны быть реализованы в модели MPI RMA и корректно работать на вычислительном кластере.

**7. Дополнительные разделы:**
Экономическое обоснование ВКР.

**8. Результаты:**
Пояснительная записка, программный код на языке программирования C.

Дата выдачи задания «_14_» _____марта_____2023_ г.

Дата представления ВКР к защите «_22_» _____июня_____2023_ г.

Студент _____ Е. В. Епифанцев

Руководитель _____ А. А. Пазников

**Санкт-Петербургский государственный электротехнический университет**

**“ЛЭТИ” им. В. И. Ульянова (Ленина)**
**(СПбГЭТУ “ЛЭТИ”)**

---

Направление: 09.03.01 “Информатика и вычислительная техника”
Профиль: “Вычислительные машины, комплексы, системы и сети”
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 разделяемых структур данных в модели MPI RMA

Студент Е. В. Епифанцев Группа № 9305

| № этапа | Наименование работ | Срок выполнения |
|---|---|---|
| 1 | Обзор литературы по теме работы | 15.03-21.03 |
| 2 | Изучение MPI и библиотеки OpenMPI | 21.04-27.04 |
| 3 | Реализация связного списка с использованием блокировок | 28.04-04.05 |
| 4 | Реализация очереди Майкла и Скотта | 05.05-15.05 |
| 5 | Реализация стека Трайбера | 16.05-24.05 |
| 6 | Тестирование реализованных структур на кластере | 25.05-26.05 |
| 7 | Оформление пояснительной записки | 27.05-08.06 |
| 8 | Представление работы к защите | 22.06.2023 |

Руководитель
к. т. н., доцент _____ А. А. Пазников
Студент _____ Е. В. Епифанцев

# РЕФЕРАТ

Пояснительная записка содержит: 51стр., 20рис., 4 таблицы, 1 приложение.

Выпускная квалификационная работа посвящена реализации разделяемых структур данных в модели MPI RMA.

Целью работы является разработка разделяемых структур данных (связный список, очередь Майкла и Скотта, стек Трайбера) в модели MPI RMA.

Объектом исследования являются разделяемые структуры данных. Предметом исследования являются: связный список с использованием блокировок, неблокирующая очередь Майкла и Скотта и неблокирующий стек Трайбера.

В рамках работы было приведено описание алгоритмов и выполнена реализация связного списка с использованием блокировок, очереди Майкла и Скотта и стека Трайбера на языке программирования C с использованием библиотеки OpenMPI. Для каждой из реализованных структур данных были проведены тесты на вычислительном кластере, с целью измерения и оценки их пропускной способности.

Полученные структуры, в отличие от их аналогов, могут применяться в вычислительных системах с распределенной памятью, например в кластерах, в задачах, где необходимы сложные вычисления или для реализации более сложных структур данных.

## ABSTRACT

In the work, a study was made of various approaches to the implementation of shared data structures, and the difficulties that arise in their development were analyzed. Programmatic implementations of three basic data structures have been created: a linked list using locks, a non-blocking queue by Michael and Scott, and a non-blocking Treiber stack in the MPI RMA model.

To evaluate the performance of the developed data structures, throughput measurements were carried out. The results of these measurements helped evaluate the effectiveness and performance of each data structure in various use cases.

**СОДЕРЖАНИЕ**

# ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

*ВС* – Вычислительная система

*Bottleneck* – узкое место в программе.

*CAS* – Compare And Swap, атомарная операция сравнения с обменом.

*DMM* – Distributed memory model, модель распределенной памяти.

*Lock-free* – свободный от блокировок, тип синхронизации алгоритма, при котором есть гарантия прогресса хотя бы одного потока.

*MPI* - Message Passing Interface, программный интерфейс для передачи сообщений в распределенных вычислительных системах.

*MR* – Memory reclamation, освобождение памяти.

Obstruction-free – без препятствий, тип неблокирующей синхронизации, при котором гарантируется прогресс одного потока, если другие не будут ему препятствовать.

*RMA* – Remote memory access, удаленный доступ к памяти.

*SMM* – Shared Memory Model, модель общей памяти.

*Wait-free* – свободный от ожидания, тип неблокирующей синхронизации, при котором гарантируется прогресс всех потоков

# ВВЕДЕНИЕ

Распределенная вычислительная система – это совокупность нескольких вычислительных машин, объединенных в коммуникационную сеть и взаимодействующих друг с другом через нее. В качестве отдельной машины вычислительной системы может выступать компьютер или сервер. Каждая машина имеет свою собственную оперативную память и может работать под управлением своей операционной системы.

Распределенные ВС нашли широкое применение в различных областях науки, например, в физике, химии, астрономии, биологии, фармакологии и так далее. В основном за счет того, что они обладают очень высокой вычислительной мощностью, которая требуется для выполнения сложных расчетов и различных задач моделирования.

Одним из основных отличий распределенной вычислительной системы от обычных компьютеров является используемая модель памяти – в распределенных системах, как следует из названия, используется модель распределенной памяти, в то время как в обычных компьютерах – модель общей памяти.

Именно это отличие не позволяет напрямую применять классические алгоритмы разделяемых структур данных в вычислительных системах с DMM.

Кроме того, большинство исследований в области разделяемых структур данных сфокусировано именно на ВС с SMM, в то время как для DMM таких исследований крайне мало. Поэтому задача реализации разделяемых структур данных, которые могут работать в ВС с DMM является крайне актуальной на сегодняшний день.

В настоящее время для создания программ для распределенных вычислительных систем активно применяется модель удаленного доступа к памяти (MPI RMA). Такая популярность обусловлена тем, что данная модель позво-

ляет сокращать время выполнения программ по сравнению с классической моделью передачи сообщений.

Целью данной работы является реализация трёх структур данных: связного списка с использованием блокировок, очереди Майкла и Скотта и стека Трайбера в модели MPI RMA для применения в вычислительной системе с распределенной памятью.

Объектом исследования являются разделяемые структуры данных. Предметом исследования – связный список с использованием блокировок, очередь Майкла и Скотта и стек Трайбера.

Для достижения обозначенной цели необходимо выполнить следующие задачи:

- Рассмотреть существующие реализации разделяемого списка, очереди и стека.
- Изучить модель MPI RMA.
- Спроектировать и реализовать разделяемые структуры на языке программирования C с использованием библиотеки OpenMPI.
- Протестировать реализованные структуры на вычислительном кластере с целью оценки их пропускной способности.

В первом разделе работы приводятся основные сведения о разделяемых структурах данных, типах алгоритмов синхронизации и проблемах возникающих при их проектировании и реализации. Во втором разделе описывается модель удаленного к памяти (MPI RMA) - рассматриваются RMA-окна, RMA-операции, типы синхронизации и обработки ошибок. Третий раздел представляет собой описание спроектированного связного списка с использованием блокировок в модели MPI RMA, там же приводится результат использования списка на одном вычислительном узле. Четвертый раздел посвящен неблокирующей очереди Майкла и Скотта, в нем приведены алгоритмы основных операций с очередью в виде текстового описания и псевдокода. Пятый раздел содержит в себе описание неблокирующего стека Трай-

бера. Так же как и в случае с очередью приводятся текстовые описания алгоритмов основных операций над стеком и их псевдокод. Шестой раздел посвящен тестированию всех реализованных структур данных на вычислительном кластере. В нём приводятся графики измерения пропускной способности всех структур и формулируются выводы о возможности применения разработанных структур. Седьмой раздел содержит экономическое обоснование выпускной квалификационной работы.

# 1 Разделяемые структуры данных

Разделяемая структура данных [1] представляет собой особый тип структуры данных, который может быть доступен для изменений нескольким параллельно работающим процессам или потокам. При этом такая структура данных гарантирует согласованность и целостность данных.

Разделяемые структуры данных находят широкое применение в параллельных и распределенных вычислениях, а также в многопоточных программных средах. Они позволяют разным процессам или потокам совместно работать с общими данными, ускоряя вычисления и повышая производительность.

## 1.1 Подходы к реализации разделяемых структур данных

В реализации разделяемых структур данных существует два подхода: с использованием блокировок и без блокировок.

Использование блокировок при реализации структур данных обычно является более простым с технической точки зрения. Во многих случаях алгоритмы и структуры данных, реализованные с использованием блокировок, оказываются достаточно эффективными и не уступают по производительности своим неблокирующим аналогам.

Однако использование неблокирующего подхода позволяет избежать проблем, связанных с взаимной блокировкой и инверсией приоритетов, которые возможны при использовании блокировок

Блокирующий подход основан на использовании блокирующих примитивах синхронизации. Среди них можно выделить следующие:

- Мьютекс – примитив синхронизации, который гарантирует доступ к конкурентным ресурсам лишь одному потоку, который первый сумел захватить мьютекс.

- Семафор – обобщение мьютекса. Главное отличие семафора от мьютекса заключается в том, что семафор может предоставлять доступ к конкурентным ресурсам нескольким потокам одновременно.
- Спинлок – примитив синхронизации. По принципу работы похож на мьютекс за исключением того, что поток, попавший в спинлок вместо того, чтобы быть снятым с исполнения планировщиком операционной системы и стать переведенным в режим ожидания освобождения мьютекса, циклически проверяет доступ к некоторому конкуретному ресурсу с помощью атомарных операций.

Неблокирующий же подход, в свою очередь, основан на использовании атомарных операций.

. Неблокирующие алгоритмы делятся на три типа:

- Без препятствий (obstruction-free) – поток, который был запущен в любой момент времени, завершит операцию за конечное число шагов. Это означает, что ни один другой поток не может препятствовать прогрессу данного потока.
- Без блокировок (lock-free) – хотя бы один поток завершит операцию за конечное число шагов. Другими словами, ни один поток не будет блокироваться бесконечно долго, и система всегда сможет сделать прогресс.
- Без ожиданий (wait-free) – каждый поток завершает свою операцию за конечное число шагов. Это означает, что ни один поток не будет ожидать завершения работы другого потока.

Стоит отметить, что описанные типы располагаются в порядке увеличения строгости. То есть, если, например, алгоритм является wait-free, то он одновременно является и lock-free и так далее.

## 1.2 Проблемы при проектировании разделяемых структур данных

Реализация разделяемых структур данных в многопоточной или многопроцессорной среде сопровождается рядом сложностей и проблем.

Гонки данных - когда несколько потоков или процессов одновременно пытаются изменить одну и ту же часть структуры данных, возникают гонки данных. Это может привести к непредсказуемому поведению и ошибкам, таким как потеря данных или некорректные результаты. Необходимо использовать механизмы синхронизации, такие как блокировки или атомарные операции, чтобы предотвратить возникновение гонок данных.

Синхронизация - для обеспечения согласованности данных при доступе нескольких потоков или процессов к разделяемым структурам данных необходимо использовать механизмы синхронизации. Однако, неправильная синхронизация может привести к взаимной блокировке (deadlock) или к резкому снижению производительности. Необходимо тщательно планировать и реализовывать механизмы синхронизации, чтобы избежать таких проблем.

Менеджмент памяти - проблема освобождения памяти (memory reclamation) возникает при разработке разделяемых структур данных в многопоточной среде, особенно при применение lock-free или wait-free алгоритмов. Она связана с освобождением памяти вслед за удалением элемента из структуры данных. Суть проблемы состоит в том, что очищение выделенной памяти под элемент структуры данных не всегда является безопасным. Например, если первый  поток сохраняет указатель на элемент структуры данных, а второй поток очищает память, ассоциированную с этим указателем, то первый поток окажется с указателем на случайное место в памяти, что может привести к самым разным последствиям.

Проблемы масштабируемости - разделяемые структуры данных должны быть эффективными и масштабируемыми в многопоточной среде. Некорректный выбор алгоритмов синхронизации может привести к появлению узких мест в программе.

Отладка - как правило, отладка и исправление ошибок в многопоточном коде может быть трудоемким и требовать специальных инструментов и методик.

### 1.3 Разделяемая структура данных в модели MPI RMA

В модели MPI RMA разделяемая структура данных представляет собой область памяти, доступ к которой может быть разделен между несколькими процессами. Каждый процесс выделяет сегмент памяти, который будет использоваться для хранения части структуры данных. Таким образом, разделяемая структура данных разбивается на несколько частей, распределенных между отдельными узлами вычислительной системы. Так, к примеру, голова очереди может находиться в памяти процесса i, а хвост очереди в памяти процесса j, и при этом возможно, что $i \neq j$.

## 2 Модель MPI RMA

MPI RMA (Remote Memory Access) [2] – это модель программирования, реализованная в библиотеке MPI (Message Passing Interface), которая предоставляет возможность прямого доступа к памяти удаленных процессов без необходимости явного обмена сообщениями.

### 2.1 Описание модели MPI RMA

В MPI RMA для межпроцессного взаимодействия вместо привычного механизма приема и отправки сообщений используется прямой доступ к памяти удаленных процессов. Такой доступ обеспечивается с помощью односторонних коммуникаций (one-sided communications). В рамках этих коммуникаций процессы выполняют RMA-операции, которые должны находиться внутри специальных областей кода, обеспечивающих синхронизацию - эпохах (epochs).

### 2.2. Виды синхронизации в модели MPI RMA

MPI RMA предоставляет два варианта синхронизации: активную синхронизацию (active target synchronization) и пассивную синхронизацию (passive target synchronization).

При использовании активной синхронизации данные перемещаются из памяти одного процесса в память другого, и при этом оба процесса активно участвуют в синхронизации. Инициирующий процесс (origin process) начинает эпоху доступа (access epoch), получая доступ к памяти целевого процесса (target process), который в свою очередь начинает эпоху воздействия (exposure epoch)  на свой участок памяти, выделенный под доступ другим процессам.

Во время этих эпох инициирующий процесс может выполнять RMA-операции, позволяющие записывать или считывать данные из памяти целево-

го процесса. Это позволяет эффективно обмениваться информацией между процессами без необходимости использования промежуточных сообщений.

При пассивной синхронизации перемещение данных и синхронизация осуществляется только инициирующим процессом, именно поэтому данный тип синхронизации получил название «пассивный». В данной работе будет применятся пассивный метод синхронизации, так как он имеет меньшее количество накладных расходов по сравнению с активной синхронизацией.

При использовании пассивного типа синхронизации инициирующий процесс открывает эпоху доступа с помощью функций MPI_Win_lock() / MPI_Win_lock_all(), внутри которой выполняет RMA-операции. В завершении эпохи вызывающий процесс должен вызвать MPI_Win_unlock() / MPI_Win_unlock_all() соответственно.

Схема пассивной синхронизации приведена на рисунке 2.1

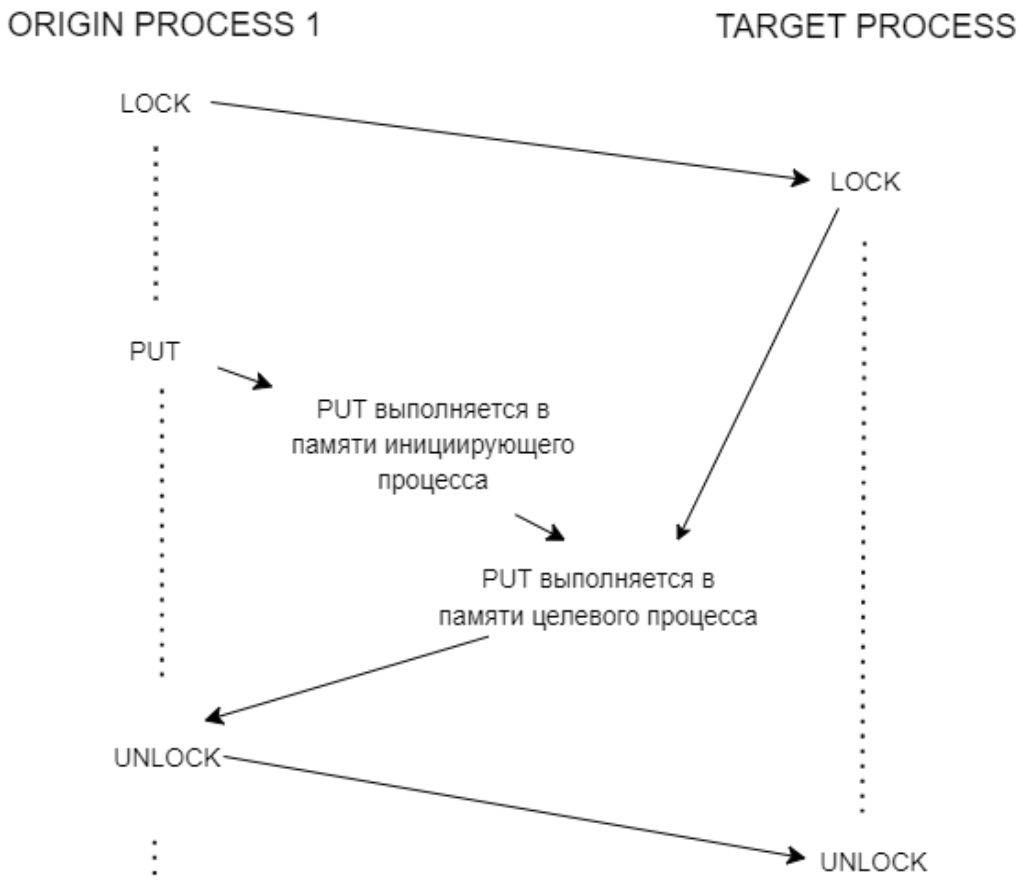


Рисунок 2.1 – Схема пассивной синхронизации

### 2.3. RMA-Окна

Сегмент памяти, который процесс делает доступным для операций чтения или записи другим процессам называется окно (window). В качестве окна может выступать как определенная часть памяти процесса, так и вся память процесса целиком. Так же каждое окно связано с определенным коммуникатором, который определяет группу процессов, имеющих доступ к нему.

На рисунке 2.2 приведен пример межпроцессного взаимодействия в модели MPI RMA.

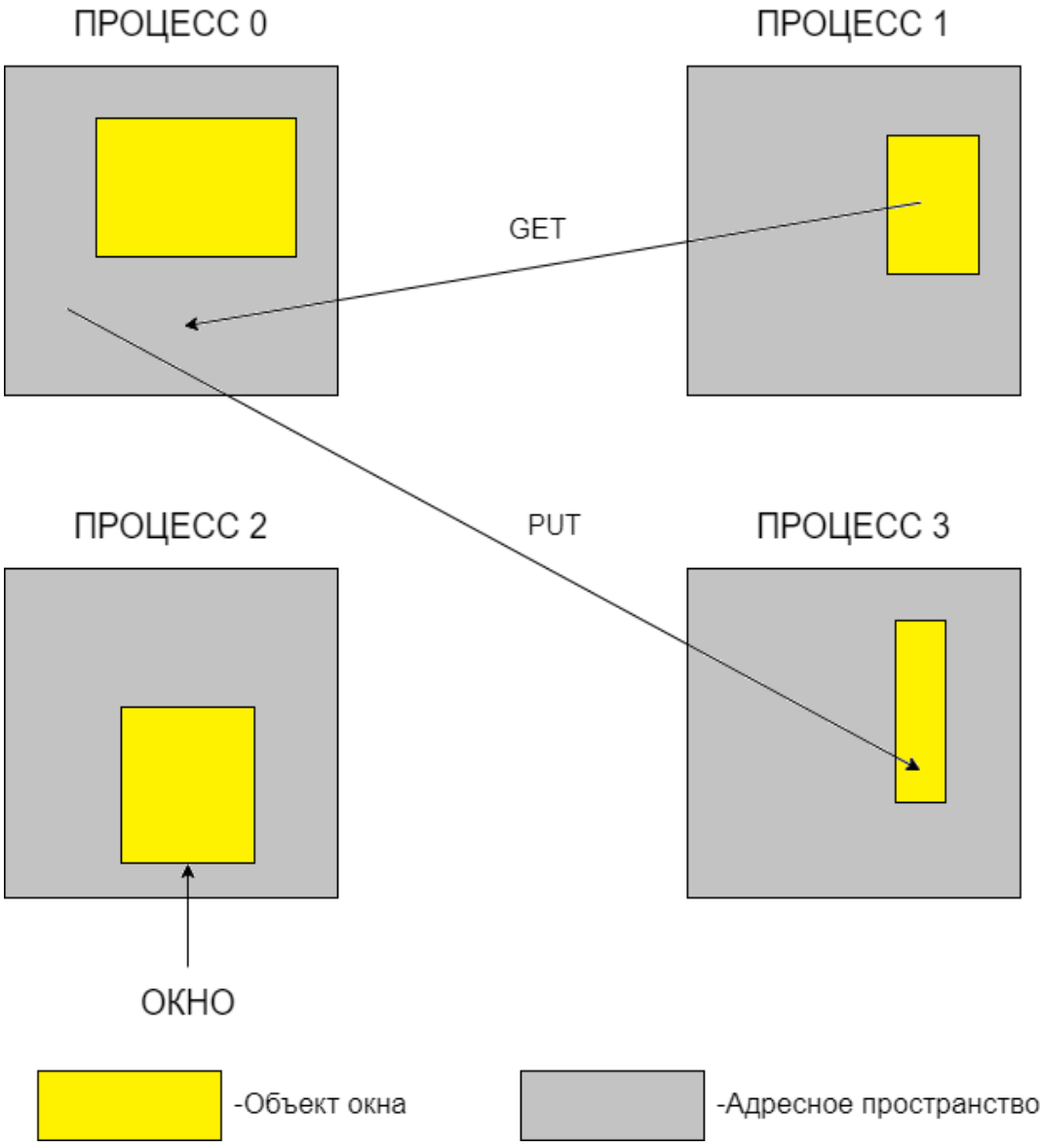


Рисунок 2.2 – Пример коммуникации процессов с использованием MPI RMA

Для работы с окнами в MPI реализованы следующие функции:

- MPI_Win_create() – данная функция создает окно и ассоциирует его с определенным сегментом памяти процесса. Она принимает в качестве аргументов указатель на начало сегмента памяти, размер сегмента, единицу смещения для окна, которая определяет единицу измерения для адресации памяти, информацию о параметрах доступа к окну и коммуникатор, определяющий группу процессов, имеющих доступ к окну.
- MPI_Win_create_dynamic() – данная функция позволяет создавать динамическое окно, то есть такое окно, размер которого может изменяться во время выполнения программы. В качестве аргументов данная функция принимает размер окна, единицу смещения, информацию о параметрах доступа к окну и коммуникатор.
- MPI_Alloc_mem() – эта функция динамически выделяет память, которая будет использоваться для RMA операций. В качестве аргументов функция принимает размер требуемой памяти в байтах и информацию о выравнивании памяти.
- MPI_Free_mem() – данная функция используется для освобождения памяти, выделенной с помощью функции MPI_Alloc_mem(). В качестве аргументов функция принимает указатель на начало выделенной памяти.
- MPI_Win_attach() – эта функция используется для присоединения сегмента памяти, созданного с помощью функции MPI_Alloc_mem(), к динамическому окну.
- MPI_Win_detach() – эта функция используется для отсоединения сегмента памяти от динамического окна.
- MPI_Win_lock() – данная функция используется для установки блокировки доступа к окну и защиты его от одновременного доступа нескольких процессов. При этом данная функция может вызываться

с одним из следующих флагов – MPI_LOCK_SHARED или MPI_LOCK_EXCLUSIVE. Флаг MPI_LOCK_EXCLUSIVE говорит о том, что окно доступно только одному процессу, а все остальные процессы, желающие получить доступ к этому окну, блокируются до тех пор, пока окно не станет доступно. Флаг MPI_LOCK_SHARED, напротив, предоставляет доступ к окну нескольким процессам, однако, только для операции чтения.

- MPI_Win_unlock() – данная функция используется для снятия блокировки с окна, которая ранее была установлена с помощью функции MPI_Win_lock().

### 2.4 RMA-операции

RMA-операции, которые могут быть использованы процессом-инициатором в течении эпохи доступа, представлены ниже:

- MPI_Put() – эта операция позволяет записать данные из локального сегмента памяти процесса в удаленный сегмент памяти другого процесса. Данная операция является неблокирующей. Схема работы функции приведена на рисунке 2.3.

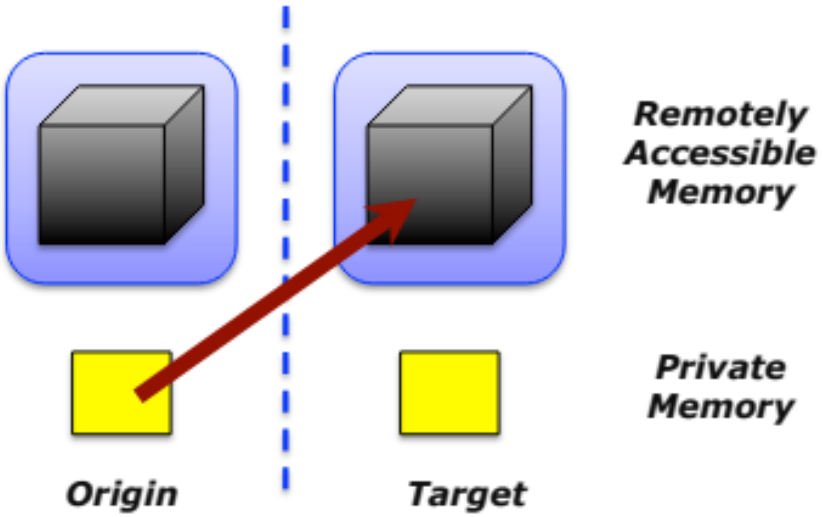


Рисунок 2.3 – Порядок работы функции MPI_Put()

- MPI_Get() – эта операция позволяет прочитать данные из удаленного сегмента памяти другого процесса и сохранить их в локальной памяти. Данная операция является неблокирующей. Схема работы функции приведена на рисунке 2.4.

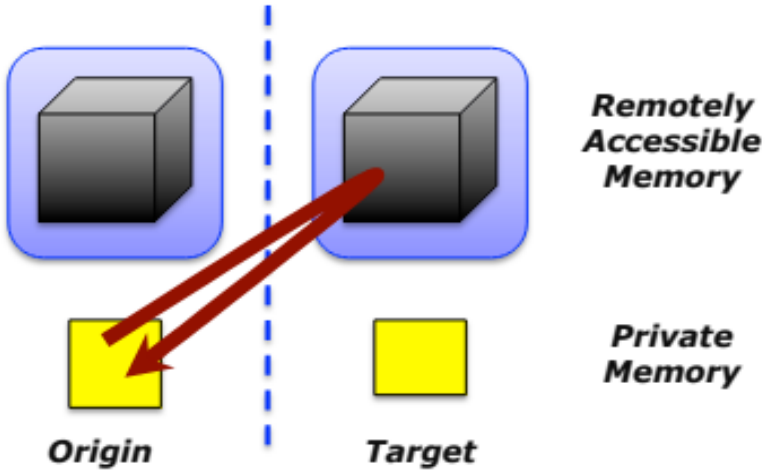


Рисунок 2.4 – Порядок работы функции MPI_Get()

- MPI_Accumulate() – данная операция позволяет накапливать (accumulate) данные из локального сегмента памяти процесса в удаленном сегменте памяти другого процесса с использованием операции накопления, такой как, например, сложение или умножение. Схема работы функции приведена на рисунке 2.5.

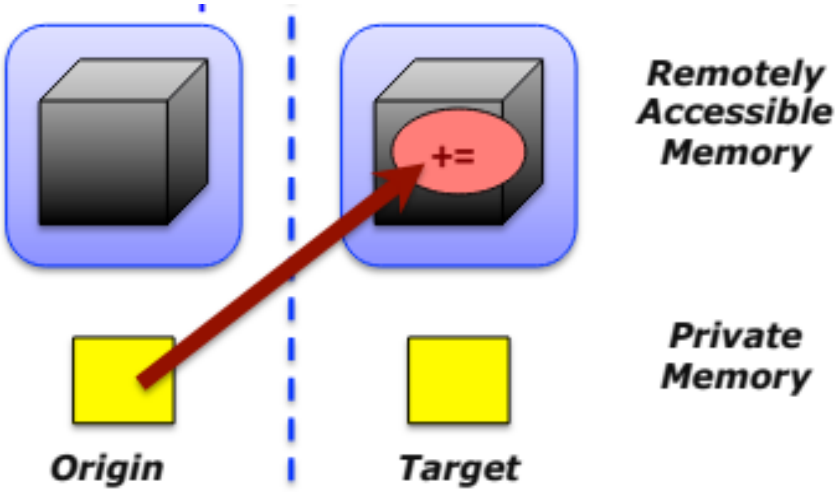


Рисунок 2.5 – Порядок работы функции MPI_Accumulate()

- MPI_Get_accumulate() – эта операция объединяет операции чтения (get) и накопления (accumulate), позволяя прочитать данные из удаленного сегмента памяти другого процесса и одновременно выполнить операцию накопления. Схема работы функции приведена на рисунке 2.6.

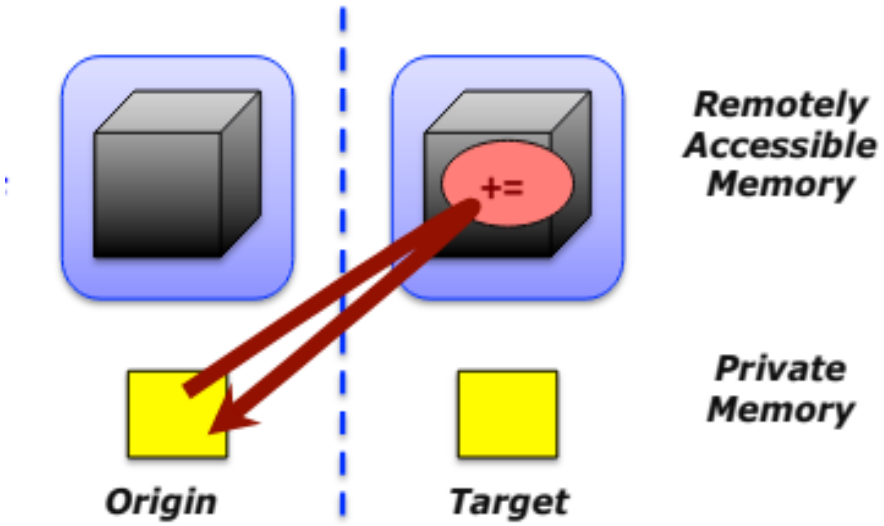


Рисунок 2.6 – Порядок работы функции MPI_Get_accumulate()

- MPI_Fetch_and_op() – данная операция выполняет чтение и некоторую атомарную операцию над удаленным сегментом памяти другого процесса;
- MPI_Compare_and_swap() – эта операция выполняет атомарное сравнение и обмен значений в удаленном сегменте памяти другого процесса;
- MPI_Win_flush() – данная функция используется для обеспечения видимости всех локальных обновлений в окне для всех других процессов, связанных с окном. Обычно данная функция используется в паре с операциями записи или чтения, чтобы гарантировать синхронизацию и видимость изменений в окне;
- MPI_Win_flush_local() – эта функция используется для обеспечения видимости локальных обновлений окна только в рамках локального процесса;

## 2.5 Обработка ошибок

Обработка ошибок является важной частью разработки параллельных программ. При использовании MPI RMA возможны различные типы ошибок: ошибки синхронизации, ошибки доступа к памяти и другие.

MPI предоставляет следующий функционал для обнаружения и обработок ошибок:

1) Коды ошибок – MPI определяет набор стандартных кодов ошибок, которые могут быть возвращены при вызове MPI-функций. Эти коды позволяют программисту определить и обработать различные типы ошибок.

2) Написание пользовательских обработчиков ошибок – MPI предоставляет возможность создания пользовательских обработчиков ошибок. Программист может определить свои собственные обработчики ошибок, которые будут срабатывать при появлении определенных типов ошибок. Это позволяет гибко реагировать на ошибки и принимать соответствующие действия, например, повторно пытаться выполнить операцию или записывать ошибку в журнал логов.

## 3 Связный список с использованием блокировок

Связный список – это структура данных, используемая для организации и хранения коллекции элементов. Он состоит из элементов, каждый из которых содержит данные и ссылку на следующий элемент в списке. Таким образом, элементы списка связаны между собой последовательно, образуя цепочку. При этом, если элемент является последним в списке, то его указатель на следующий элемент равен пустому указателю (null).

Распределенный связный список – это связный список, элементы которого находятся в памяти процессов, которые работают с данным списком и выполнятся на распределенной вычислительной системе.

Таким образом, можно получить схематичное представление распределенного связного списка [рисунок 3.1].

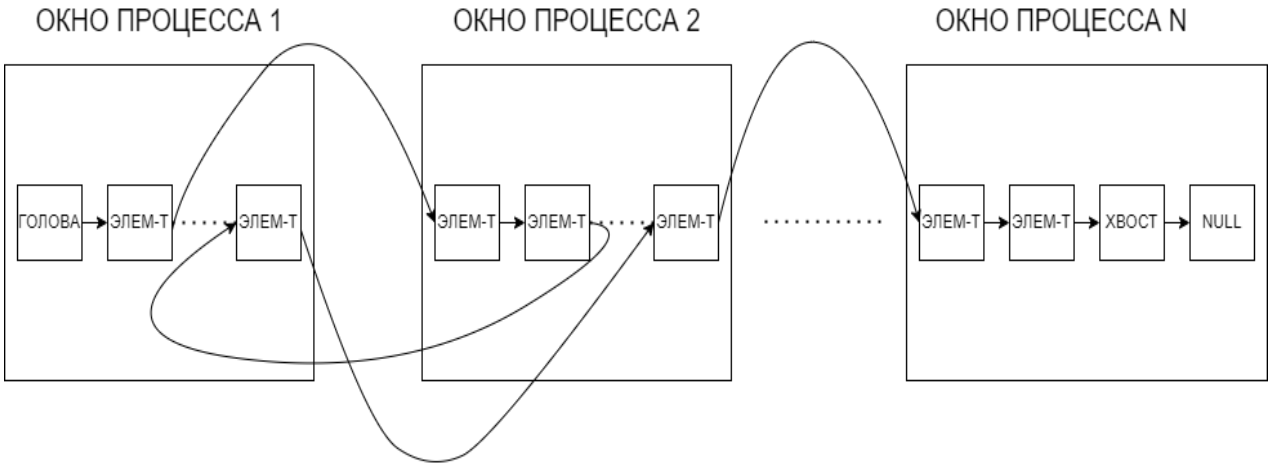


Рисунок 3.1 – Структура распределенного связного списка

### 3.1 Описание представления элемента списка и указателя

Для представления элемента списка создана структура node. Поля структуры и их назначения следующие:

- Id – уникальный номер элемента в списке (ключ).
- Val – пользовательские данные.

- logicalyDeleted – флаг, сигнализирующий о том, что данный элемент был логически удален из списка, но все еще является доступным для других процессов.
- canBeReclaimed – флаг, сигнализирующий о том, что данный узел был логически удален из списка, и теперь память, выделенная под данный элемент может быть использована заново.
- next – указатель на следующий элемент списка.

Для представления указателя на элемент списка была создана структура nodePtr. Поля данной структуры и их назначение:

- rank – ранг процесса, в окне которого расположен элемент.
- disp – смещение внутри окна, начиная с которого в памяти расположен элемент.

### 3.2 Алгоритмы операции над списком

Связный список поддерживает три основных операции – вставка нового элемента после заданного по ключу элемента, удаление элемента по ключу и поиск элемента по ключу.

Здесь и далее в псевдокоде будут использоваться следующие обозначения:

- Lock() – функция MPI_Win_lock().
- Get() – функция MPI_Get().
- Flush() – функция MPI_Win_flush().
- Unlock() – функция MPI_Win_unlock().
- Fetch() – функция MPI_Fetch_and_op().
- Put() – функция MPI_Put().
- CAS() – функция MPI_Compare_and_swap().
- EmptyNode – пустой узел списка.
- Head – указатель на голову списка.
- nullPtr – null указатель.

Опишем алгоритм поиска элемента и приведем его псевдокод [рисунок 3.2]:

1. Установить текущий указатель на голову (2 строка).
2. Создать пустой буфер под считывание узла списка (3 строка).
3. Запустить цикл – пока текущий указатель не равен нулевому указателю (4 строка).
4. Захватить Lock на окно с проверяемым узлом списка (5 строка).
5. Считать узел списка по текущему указателю (6 строка).
6. Если ключ текущего узла является тем, который был передан в функцию, отпустить Lock и вернуть указатель на данный узел – поиск завершен, элемент найден, иначе запомнить указатель на следующий узел и продолжить поиск (8 – 13 строка).
7. Отпустить Lock и сместить указатель на следующий узел (14 – 15 строка).
8. Если узел с заданным ключом не был найден в списке, то вернуть null (строка 17).

```
1:  function SEARCH(id, head, win)
2:      curNodePtr ← head
3:      curNode ← emptyNode
4:      while curNodePtr ≠ nullPtr do
5:          Lock(EXCLUSIVE, curNodePtr.rank, win)
6:          Get(curNodePtr, curNode, win)
7:          Flush(curNodePtr.rank, win)
8:          if curNode.id == id then
9:              Unlock(curNodePtr.rank, win)
10:             return curNodePtr
11:         else
12:             next ← curNode.next
13:         end if
14:         Unlock(curNodePtr.rank, win)
15:         curNodePtr ← next
16:     end while
17:     return nullPtr
18: end function
```

Рисунок 3.2 – Псевдокод функции поиска элемента

Опишем алгоритм вставки нового элемента после заданного и приведем его псевдокод [рисунок 3.3]:

1. Вызвать функцию поиска элемента, после которого необходимо выполнить вставку (4 строка).
2. Если элемент не найден в списке, то вставка не удалась (5-6 строка).
3. Если элемент найден, то выделить память под новый элемент, захватить Lock на окно с найденным элементом, считать по найденному адресу узел списка (8 – 11 строка).
4. Если найденный элемент не удален логически из списка, заменить его next указатель на вновь созданный узел, а созданному узлу в поле next установить старый указатель, который был в найденном узле (12 – 15 строка).
5. Иначе найденный элемент помечен как логически удаленный, значит, изменение указателя next в нем запрещена (16 – 18 строка).
6. Отпустить Lock и вернуться из функции (19 – 21 строки).

```
1:  function INSERTAFTER(id, newVal, key, rank, head, win)
2:      curNodePtr ← nullPtr
3:      fetched ← nullPtr
4:      curNodePtr ← search(key, head, win)
5:      if curNodePtr == nullPtr then
6:          return
7:      else
8:          newNode ← allocElem(id, newVal, rank, win)
9:          Lock(EXCLUSIVE, curNodePtr.rank, win)
10:         Get(curNode, curNodePtr, win)
11:         Flush(curNodePtr.rank, win)
12:         if curNode.logicalyDeleted ≠ 1 then
13:             Fetch(newNode, fetched, curNode → next, REPLACE, win)
14:             Flush(curNodePtr.rank, win)
15:             newNode.next ← fetched
16:         else
17:             Unlock(curNodePtr.rank, win)
18:             return
19:         end if
20:         Unlock(curNodePtr.rank, win)
21:     end if
22: end function
```

Рисунок 3.3 – Псевдокод функции вставки нового

Опишем алгоритм удаления элемента из списка и приведем его псевдокод [рисунок 3.4]:

1. Вызвать функцию поиска элемента, который необходимо удалить (6 строка).
2. Если элемент не найден в списке, то удаление не удалось, выйти из функции (7-8 строка).
3. Если элемент найден, то захватить Lock на окно, где находится найденный элемент. Провести операцию сравнения с обменом на поле logicalyDeleted и вернуть результат в переменную result (10-12 строка).
4. Если в переменной result единица, то какой-то другой процесс уже логически удалил данный элемент. Необходимо разблокировать окно и выйти из функции (16, 26 строка).
5. Если в переменной result ноль, то найденный элемент не был логически удален, нужно считать и сохранить указатель next из этого элемента (13 - 15 строка). Сохранение указателя на следующий узел возможно, так как при логическом удалении элемента из списка, появляется гарантия того, что следующий за данным элементом узел не будет изменен. Единственное, что с этим элементом может произойти – это его удаление;
6. В цикле вызывать функцию traverseAndDelete(), которая проходит по списку и ищет элемент, указатель next которого равен найденному ранее элементу для удаления curNodePtr. Когда данный элемент обнаруживается, функция меняет next указатель предыдущего к curNodePtr элементу на ранее сохраненный next и возвращает 1 (18 – 20 строка).
7. После удаления элемента из списка установить его флаг canBeReclaimed. Это будет означать, что память, выделенная под этот узел, может быть использована заново и быть проинициализирована новыми значениями (21 – 23 строка).

```
1: function DELETE(key, head, win)
2:     curNodePtr ← nullPtr
3:     curNode ← emptyNode
4:     testDelete ← 1
5:     free ← 1
6:     curNodePtr ← search(key, head, win)
7:     if curNodePtr == nullPtr then
8:         return
9:     else
10:        Lock(EXCLUSIVE, curNodePtr.rank, win)
11:        CAS(curNodePtr.offset + offsetof(node, mark), 0, 1, result)
12:        Flush(curNodePtr.rank, win)
13:        if result ≠ 1 then
14:            Get(next, curNodePtr.offset + offsetof(node, next), win)
15:        end if
16:        Unlock(curNodePtr.rank, win)
17:        if result ≠ 1 then
18:            while testDelete == 0 do
19:                testDelete ← traverseAndDelete(head, curNodePtr, next, win)
20:            end while
21:            Lock(EXCLUSIVE, curNodePtr.rank, win)
22:            Put(free, curNodePtr.offset(node, canBeReclaimed), win)
23:            Unlock(curNodePtr.rank, win)
24:        end if
25:    end if
26: end function
```

Рисунок 3.4 – Псевдокод функции удаления элемента из списка

### 3.3 Менеджмент памяти

Одной из проблем, которая может возникнуть при проектировании разделяемых структур данных, является освобождение выделенной памяти вслед за удалением элемента из структуры данных. Данная проблема получила название memory reclamation.

При проектировании блокирующих структур данных данная проблема решается достаточно просто. Обычно для этого достаточно захватить Lock на элемент, память под который необходимо очистить. Это даст гарантию того, что никакой другой процесс или поток не будет оперировать этим элементом, следовательно, очистка памяти в данном случае будет безопасной.

В приведенной реализации связного списка с использованием блокировок каждый процесс создает и хранит массив элементов списка, под которые

он выделяет память. Перед созданием нового элемента в функции allocNode() процесс блокирует массив элементов и проходит по нему в поисках элемента, который является удаленным из списка (маркером этого является установленный флаг canBeReclaimed), если такой элемент найден, то это означает, что он может быть использован для повторной инициализации вместо создания нового элемента.

## 4 Неблокирующая очередь Майкла и Скотта

Очередь – это структура данных, которая реализует метод FIFO («первым вошел – первым вышел»), то есть первый элемент, добавленный в очередь, будет первым, который выйдет из нее.

В отличие от других структур данных, очередь ориентирована на сохранение порядка элементов, гарантируя, что элементы будут обрабатываться в том порядке, в котором они были добавлены. Это делает очередь незаменимой при работе с задачами, где требуется учет времени поступления или приоритета элементов.

### 4.1 Структура и описание очереди Майкла и Скотта

Очередь Майкла и Скотта [5] реализована в виде односвязного списка, где каждый элемент содержит пользовательские данные и указатель на следующий элемент. Если элемент является последним в списке, то его next указатель равен null. Очередь представлена двумя указателями – на голову и хвост. Удаление элементов (операция enqueue) происходит с головы, а добавление новых элементов, осуществляется с хвоста (операция dequeue).

Изначально очередь состоит лишь из одного элемента, на который ссылаются указатели головы и хвоста. Данный элемент называется dummy. При этом данные, которые хранятся в этом узле, не имеют значения. Смысл данного элемента – в упрощении программной реализации операций добавления и удаления элементов.

### 4.2 Алгоритмы операций над очередью

Очередь поддерживает две операции – enqueue (вставка нового элемента в очередь) и dequeue (удаление элемента из очереди).

При выполнении операции вставки возникает проблема, заключающаяся в невозможности атомарного добавления элемента в очередь и изменения указателя хвоста на вновь добавленный элемент. Для решения данной про-

блемы можно применить технику помощи (helping). Суть этой техники заключается в следующем: при вставке нового элемента в очередь, если операция CAS, выполняемая для установки указателя tail.next в значение null для нового хвоста, не удалась (то есть другой процесс успел выполнить вставку и tail.next больше не равен null), текущий процесс может выполнить CAS (T, tail, tail.next.get()) для корректного обновления указателя хвоста очереди T на вновь считанный элемент. Если CAS выполнен успешно, то хвост перенесен успешно. Если же он выполнен неудачно, то это значит, что T уже не указывает на tail, а значит, другой процесс переместил хвост. При любом результате выполнения операции CAS процесс должен вернуться к добавлению нового элемента в очередь.

Опишем алгоритм добавления элемента в очередь и приведем его псевдокод [рисунок 4.1]:

1. Инициализировать локальные переменные tmpTail и tailNext значением null (2-3 строка).
2. Выделить память под новый элемент и инициализировать его поля (4 строка).
3. В цикле пока вставка не завершилась с успехом: считать в локальную переменную tmpTail текущий хвост очереди (строка 6).
4. Выполнить операцию CAS для добавления нового элемента в очередь (7 строка).
5. Если добавление выполнено успешно, то попытаться перенести указатель очереди на только что добавленный элемент и вернуться из функции (9 – 11 строка).
6. Если добавление не удалось, значит какой-то другой процесс уже вставил новый элемент после tmpTail, значит необходимо помочь перенести указатель очереди на новый хвост и вернуться к добавлению нового элемента (12 – 15 строка).

```
1: function ENQUE(val, rank, q, win)
2:     tmpTail ← nullPtr
3:     tailNext ← nullPtr
4:     newNode ← allocElem(val, rank, win)
5:     while True do
6:         tmpTail ← getTail(q, win)
7:         CAS(tmpTail, nullPtr, newNode, result)
8:         Flush(tmpTail.rank, win)
9:         if result == nullPtr then
10:            CAS(q.tail, tmpTail, newNode, result)
11:            return
12:        else
13:            tailNext ← getTail(q, win)
14:            CAS(q.tail, tmpTail, tailNext, result)
15:            Flush(0, win)
16:        end if
17:    end while
18: end function
```

Рисунок 4.1 – Псевдокод алгоритма вставки элемента в очередь

Опишем алгоритм удаления элемента из очередь и приведем его псевдокод:

1. Считать в локальные переменные head, tail и afterHead значения текущей головы, хвоста и следующего за головным элементом узел соответственно (3 – 5 строка).
2. Если указатель на голову и хвост совпадают, то это еще не означает, что очередь пуста, ведь между считыванием головы, хвоста и следующего за головным элементом другой процесс мог вставить новый элемент в очередь, поэтому очередь пуста, только если все три указателя равны null (6 – 8 строка).
3. Если обнаружено, что afterHead не null,то необходимо переместить указатель головы очереди на afterHead и вернуться к попытке удаления элемента (10 – 11 строка).
4. Если указатель на голову и хвост не совпадают, то очередь гарантированно не пуста, необходимо переместить указатель головы на afterHead и выйти из функции (14 – 18 строка).

```
1: function DEQUEUE(q, win)
2:     while True do
3:         head ← getHead(q, win)
4:         tail ← getTail(q, win)
5:         afterHead ← getNextHead(head, win)
6:         if tail == head then
7:             if afterHead == nullPtr then
8:                 return
9:             else
10:                CAS(q.tail, tail, afterHead, result)
11:                Flush(0, win)
12:            end if
13:        else
14:            CAS(q.head, head, afterHead, result)
15:            Flush(0, win)
16:            if result == head then
17:                return
18:            end if
19:        end if
20:    end while
21: end function
```

Рисунок 4.2 – Псевдокод алгоритма удаления элемента из очереди

Стоит отметить, что функции getHead(), getTail() и getNextHead() используют атомарные операции для чтения текущей головы, хвоста и следующего за текущей головой элемента.

## 5 Неблокирующий стек Трайбера

Стек – это структура данных, которая реализует метод LIFO («Последний вошел, первый вышел»), то есть последний элемент, добавленный в стек, будет первым, который выйдет из него.

Стек является важной структурой данных, которая применяется во множестве алгоритмов, среди которых можно отметить алгоритмы на графах и деревьях, где стек применяется для обхода.

### 5.1 Структура и описание стека Трайбера

Стек Трайбера [6] – это масштабируемый стек без блокировок, предоставляющий неблокирующий механизм добавления и удаления элементов, что позволяет конкурентным потокам безопасно выполнять операции над стеком без блокировок или ожидания других потоков.

Структура стека следующая: стек построен на односвязном списке, в узлах которого находятся пользовательские данные и указатель на следующий элемент. Так же в программе необходимо хранить указатель на голову стека.

### 5.2 Алгоритмы операций над стеком

Стек поддерживает две операции – добавление нового элемента (push) и удаление элемента (pop).

Опишем алгоритм добавления нового элемента в стек и приведем его псевдокод [рисунок 5.1]:

1. В локальные переменные curHead (текущая голова), newHead (новая голова), result (переменная для хранения результата операции CAS) записать значение null (2 - 4 строка).
2. Выделить память под новый элемент и проинициализировать поля структуры (5 строка).

3. В цикле, пока вставка не прошла успешно: считать в локальную переменную curHead указатель на текущую голову стека, изменить указатель newHead.next на curHead, выполнить операцию CAS(H, curHead, newHead) и записать результат выполнения в переменную res (6 – 10 строка).
4. Если CAS выполнен успешно, вставка завершена, выйти из функции (11 – 13 строка).
5. Если CAS провалился (то есть если другой процесс добавил новый элемент), то вернуться к началу цикла и повторить попытку вставки элемента еще раз.

```
1:  function PUSH(val, rank, s, win)
2:      curHead ← nullPtr
3:      newHead ← nullPtr
4:      result ← nullPtr
5:      newHead ← allocElem(val, win)
6:      while True do
7:          curHead ← getHead(s, win)
8:          changeNext(curHead, newHead, win)
9:          CAS(s.head, curHead, newHead, result)
10:         Flush(s.head.rank, win)
11:         if result == curHead then
12:             return
13:         end if
14:     end while
15: end function
```

Рисунок 5.1 – Псевдокод алгоритма вставки нового элемента в стек

Опишем алгоритм удаления элемента из стека и приведем его псевдокод [рисунок 5.2]:

1. В локальные переменные curHead (текущая голова), nextHead (следующий за головным элемент), result (переменная для хранения результата операции CAS) записать значение null (2 - 4 строка).
2. В цикле: считать текущую голову стека в переменную curHead (6 строка).

3. Если текущая голова стека пуста, значит стек пуст, выйти из функции (7 – 9 строка).
4. Если стек не пуст, то считать следующий за головным элемент в переменную nextHead и выполнить операцию CAS(H, curHead, nextHead) (10 – 12 строка).
5. Если CAS выполнен успешно, то элемент успешно удален, выйти из функции (13 – 15 строка).
6. Если CAS провалился, то это значит, что другой процесс вставил новый элемент, нужно вернуться к началу цикла и повторить попытку удаления еще раз.

```
1:  function POP(s, win)
2:      curHead ← nullPtr
3:      nextHead ← nullPtr
4:      result ← nullPtr
5:      while True do
6:          curHead ← getHead(s, win)
7:          if curHead == nullPtr then
8:              return
9:          end if
10:         nextHead ← getNextHead(curHead, win)
11:         CAS(s.head, curHead, nextHead, result)
12:         Flush(s.head.rank, win)
13:         if result == curHead then
14:             return
15:         end if
16:     end while
17: end function
```

Рисунок 5.2 – Псевдокод алгоритма удаления элемента из стека

Стоит отметить, что функции getHead() и getNextHead() содержат атомарные операции чтения текущей головы и следущего за головным элементом соответственно.

## 6 Тестирование реализованных структур на кластере

Экспериментальное исследование проводилось на вычислительном кластере с 4 вычислительными узлами. При этом на каждом из узлов находилось по 1 4-ядерному процессору линейки Intel Xeon с базовой частотой 2 ГГц и максимальной частотой 3.2 ГГц. В качестве MPI-библиотеки использовалась OpenMPI 4.1.2. На рисунке 6.1 приведена конфигурация отдельно взятого узла кластера.

```
processor       : 7
vendor_id       : GenuineIntel
cpu family      : 6
model           : 106
model name      : Intel Xeon Processor (Icelake)
stepping        : 0
microcode       : 0x1
cpu MHz         : 1995.312
cache size      : 16384 KB
physical id     : 0
siblings        : 8
core id         : 3
cpu cores       : 4
apicid          : 7
initial apicid  : 7
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
```

Рисунок 6.1 – Подробная информация о процессорах, использующихся на кластере

### 6.1 Тестирование очереди и стека

Для тестирования очереди и стека был разработан тест, в котором каждый из процессов выполнял по 10000 операций вставки/удаления элементов (тип операции выбирался случайно).

Для оценки масштабируемости реализованных структур данных принята метрика «пропускная способность», которая вычисляется следующим образом $b = \frac{n*p}{t}$, где n – количество операций, p –количество процессов, t – общее время выполнения теста.

Пример тестирования очереди Майкла и Скотта на вычислительном кластере приведен на рисунке 6.2.

```
------val 9980 was inserted by rank 14 at displacement 559bac4c92f0 next rank 14 next displacement 559bac4c9310------
------val 9981 was inserted by rank 14 at displacement 559bac4c9310 next rank 14 next displacement 559bac4c9330------
------val 9986 was inserted by rank 14 at displacement 559bac4c9330 next rank 14 next displacement 559bac4c9350------
------val 9987 was inserted by rank 14 at displacement 559bac4c9350 next rank 14 next displacement 559bac4c9830------
------val 9989 was inserted by rank 14 at displacement 559bac4c9830 next rank 14 next displacement 559bac4c9850------
------val 9990 was inserted by rank 14 at displacement 559bac4c9850 next rank 14 next displacement 559bac4c9870------
------val 9992 was inserted by rank 14 at displacement 559bac4c9870 next rank 14 next displacement 559bac4c9890------
------val 9993 was inserted by rank 14 at displacement 559bac4c9890 next rank 14 next displacement 559bac4c9d70------
------val 9994 was inserted by rank 14 at displacement 559bac4c9d70 next rank 14 next displacement 559bac4c9d90------
------val 9996 was inserted by rank 14 at displacement 559bac4c9d90 next rank 14 next displacement 559bac4c9db0------
------val 9998 was inserted by rank 14 at displacement 559bac4c9db0 next rank 2047 next displacement 0------
Total element count = 457
Expected element count = 457
Test result: total elapsed time = 46.377310 ops/sec = 3449.962939
Queue Integrity: True
rank 2 of all 16 ranks was working on node master-node
rank 3 of all 16 ranks was working on node master-node
rank 1 of all 16 ranks was working on node master-node
rank 0 of all 16 ranks was working on node master-node
rank 7 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-ynih
rank 5 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-ynih
rank 4 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-ynih
rank 6 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-ynih
rank 11 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-awef
rank 15 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-ebab
rank 8 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-awef
rank 13 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-ebab
rank 9 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-awef
rank 14 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-ebab
rank 10 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-awef
rank 12 of all 16 ranks was working on node cl1vpq12ej5uakm60r3r-ebab
```

Рисунок 6.2 - Пример тестирования очереди на кластере

Отобразим результаты экспериментов на графике [рисунки 6.3, 6.4].

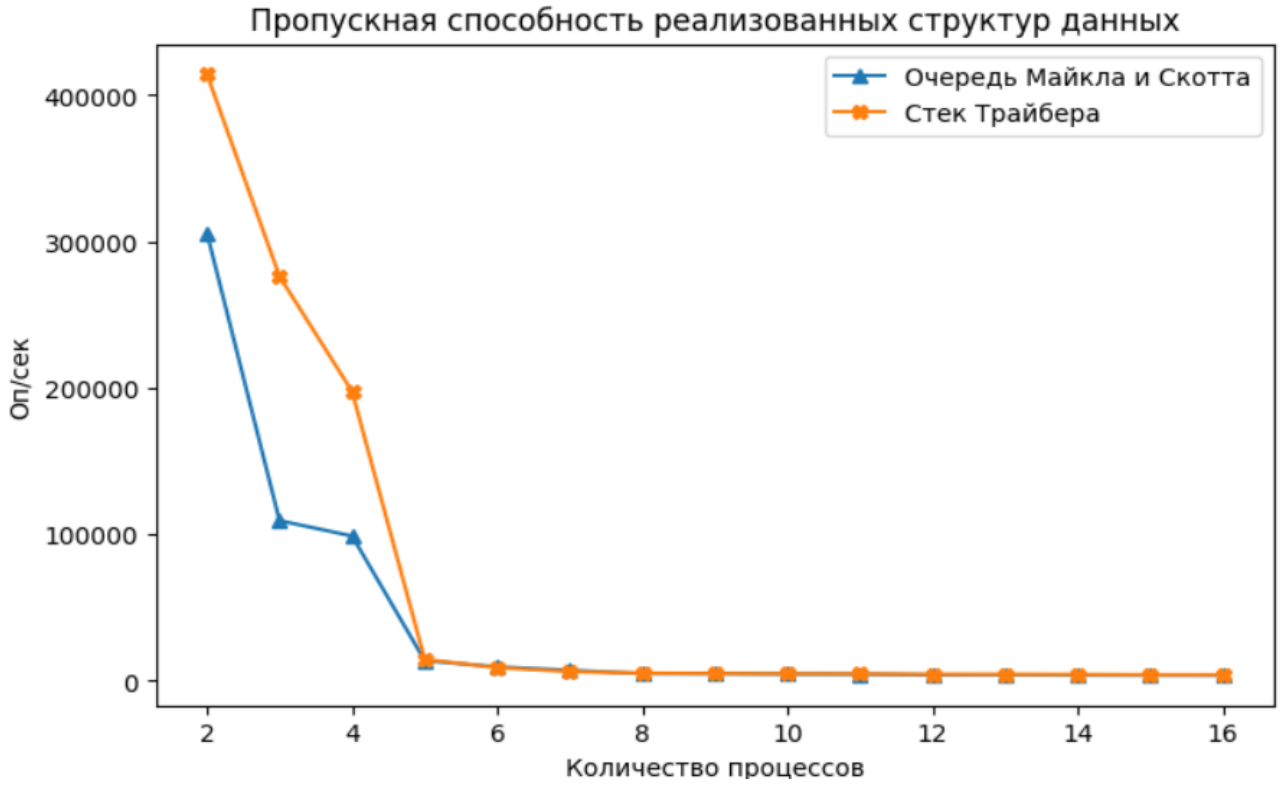


Рисунок 6.3 – Пропускная способность очереди и стека

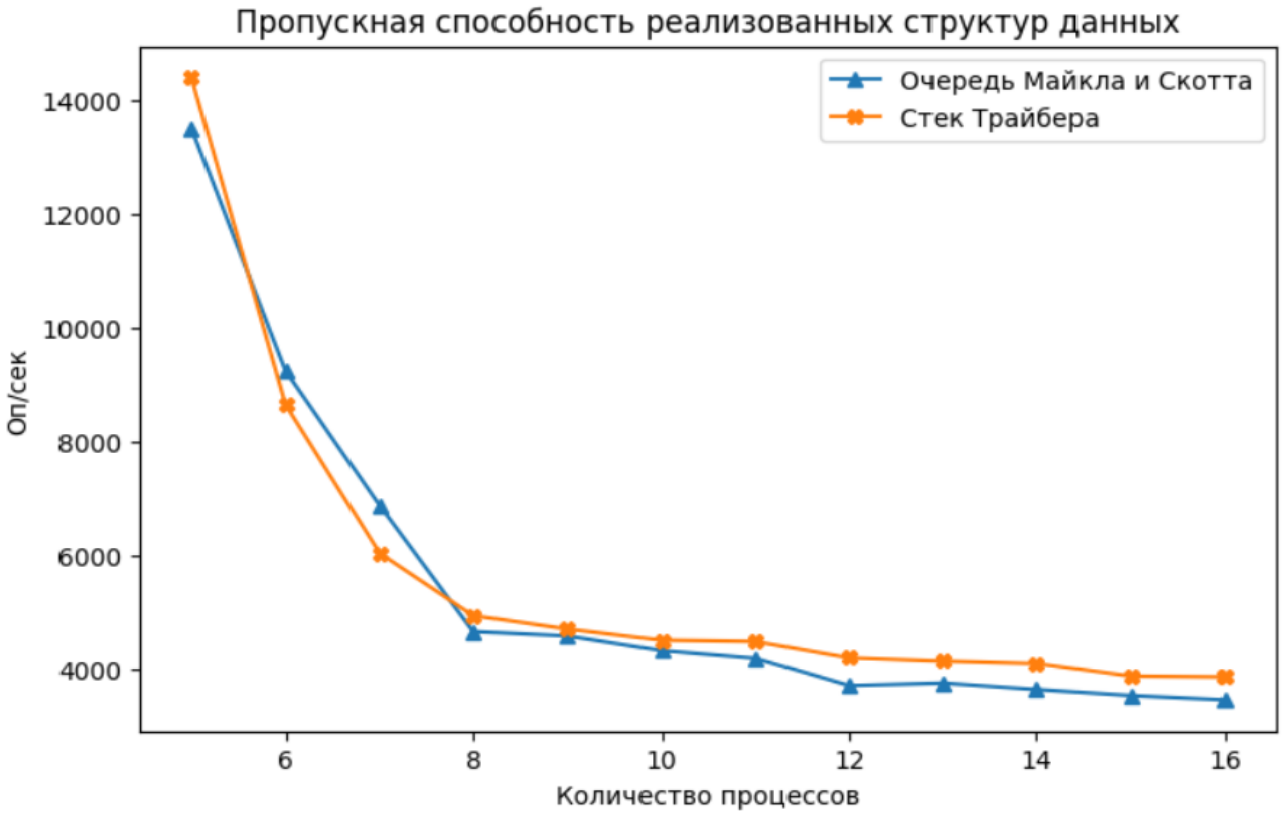


Рисунок 6.4 – Пропускная способность очереди и стека (6 - 16 процессов)

В пределах одного вычислительного узла (4 процесса) пропускная способность обеих структур находится на достаточно высоком уровне. Ее снижение на промежутке от 2 до 4 процессов можно объяснить тем, что с возрастанием числа процессов возрастает и конкуренция за доступ к отдельным элементам структуры данных, которая требует синхронизации процессов.

Когда в работу включаются несколько узлов, то пропускная способность заметно снижается, это объясняется дополнительными накладными расходами на передачу данных между процессами по сети. Несмотря на это, пропускная способность остается на приемлемом уровне (около 4000 оп/с).

### 6.2 Тестирование связного списка

Для тестирования списка был разработан тест, в котором процесс с рангом ноль изначально формировал список длиной 512 элементов, а затем каждый из остальных процессов, начиная с головы, либо удалял текущий эле-

мент, либо вставлял новый элемент после текущего (тип операции выбирался равновероятно). Отобразим результаты экспериментов на графике [рисунок 6.5].

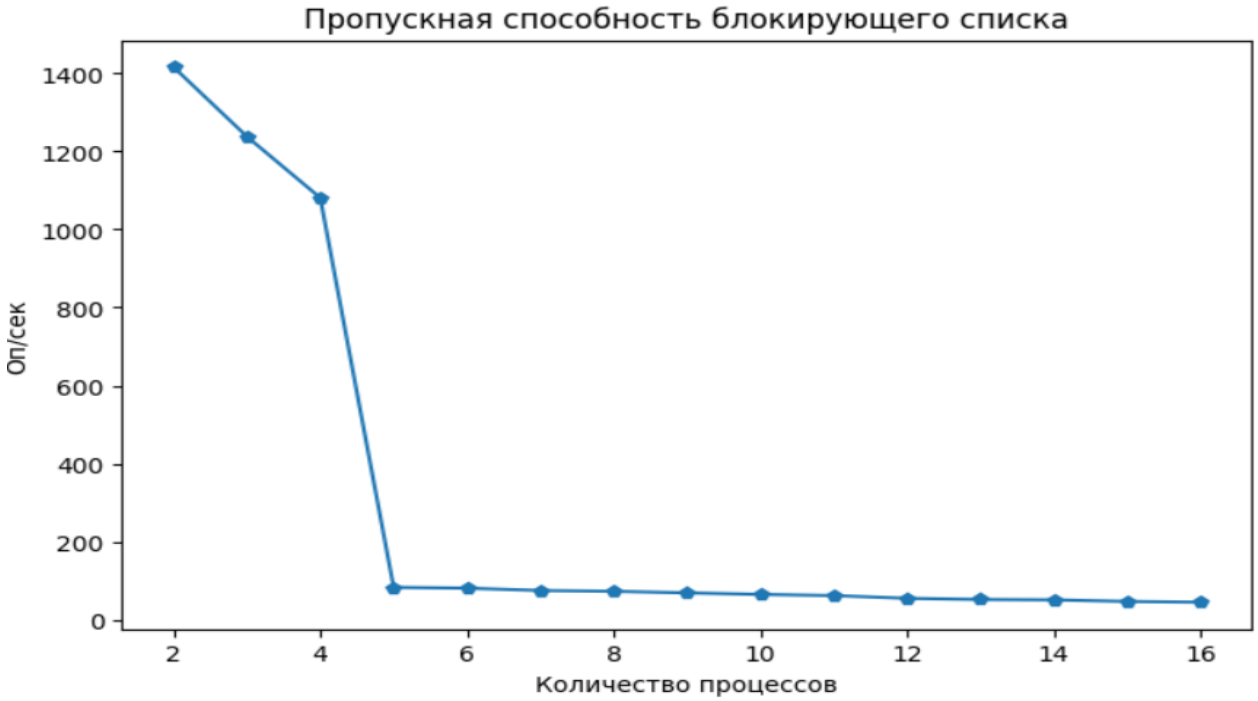


Рисунок 6.5 – Пропускная способность списка

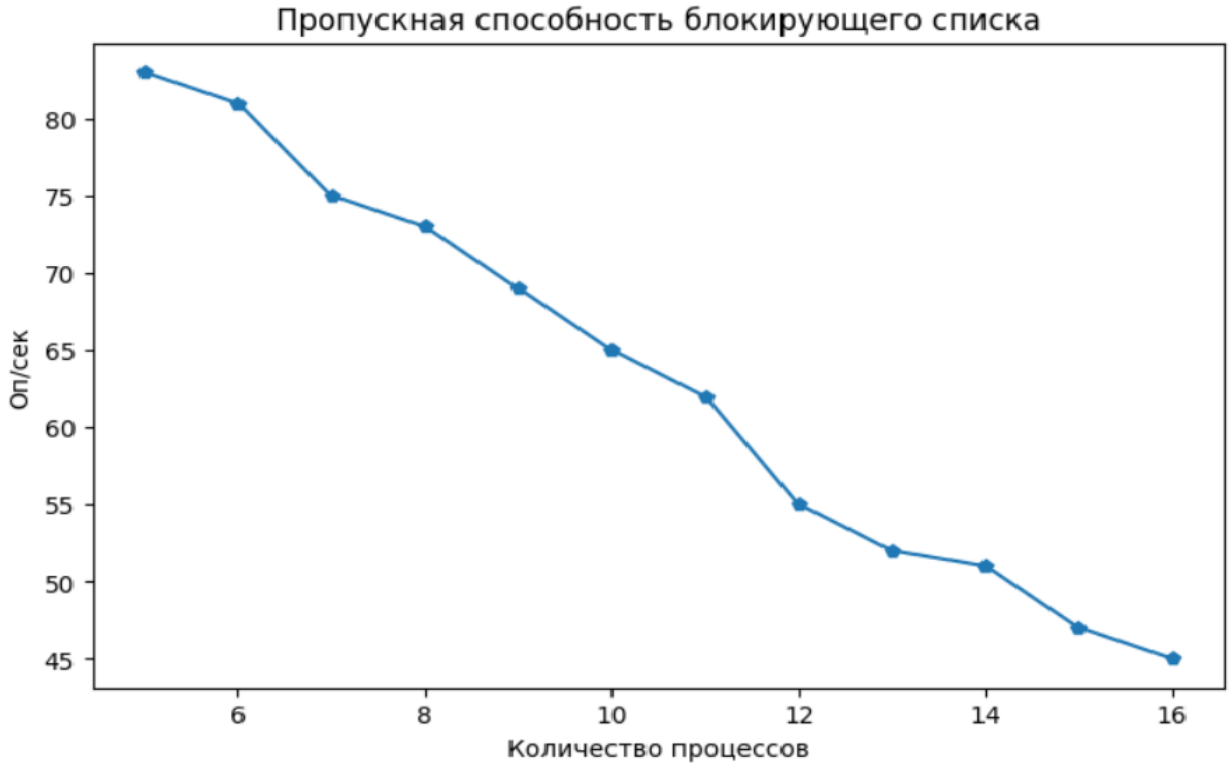


Рисунок 6.6 – Пропускная способность списка (6 - 16 процессов)

Как и ожидалось, использование блокирующего метода синхронизации приводит к плохой масштабируемости. Это можно объяснить дополнительными накладными расходами на захват и освобождение блокировки, а так же тем фактом, что блокировка распространяется не на отдельный узел списка, а на окно, которое хранит несколько элементов, то есть блокировка распространяется на несколько элементов списка сразу.

Исходя из проведенных экспериментов, можно сделать вывод о том, что блокирующий метод синхронизации хоть и отличается своей простотой и наглядностью в плане программной реализации, он показывает гораздо меньшую пропускную способность, чем неблокирующая синхронизация.

В то же время, реализованные очередь и стек имеют один общий недостаток – хранение данных о структуре (указатель на голову и хвост в случае очереди и указатель на голову в случае стека) в корневом процессе, что является узким местом. Тем не менее, данные структуры показывают неплохую масштабируемость и являются защищенными от взаимных блокировок и инверсий приоритетов.

## 7 Экономическое обоснование ВКР

Экономическая оценка труда – это важный инструмент, который помогает определить затраты на разработку программного продукта и потенциальную прибыль, которую она может принести.

### 7.1 Концепция экономического обоснования

Реализованные в ходе выполнения работы структуры данных выгодно отличаются от своих аналогов, так как могут быть использованы в вычислительных системах с распределенной памятью, например, в вычислительных кластерах или суперкомпьютерах.

### 7.2 Составление плана-графика выполнения работ

Для определения совокупной трудоемкости написания приведенных программ необходимо составить детализированный план-график, данные из которого затем будут применяться для дальнейших расчетов. В качестве измерения трудоемкости принята единица человеко-день.

План-график работ представлен в таблице 7.1.

Таблица 7.1 – План-график выполнения работ

| Наименование работы | Исполнитель | Длительность, человеко-дни |
|---|---|---|
| Консультации с научным руководителем | Руководитель | 2 |
| | Студент | 2 |
| Постановка задачи, выдача ТЗ | Руководитель | 1 |
| Реализация связного списка с блокировками | Студент | 7 |
| Реализация неблокирующей очереди | Студент | 11 |
| Реализация неблокирующего стека | Студент | 9 |
| Тестирование полученных структур данных | Руководитель | 2 |
| | Студент | 1 |
| Оформление пояснительной записки | Студент | 13 |

Таким образом, на выполнение всех работ было затрачено:

-студентом: 43 ч. дн.

-руководителем: 5 ч. дн.

## 7.3 Расчет расходов на оплату труда исполнителей и отчислений на социальные нужды

На основе совокупной трудоемкости выполняемых работ и ставки исполнителя за день можно оценить расходы на основную и дополнительную заработную плату исполнителей.

### 7.3.1 Расчет основной заработной платы исполнителей

Используя данные о месячных заработных платах руководителя и студента (принимается равной заработной плате инженера согласно [8]), рассчитаем стоимость оплаты одного человека-дня для каждого исполнителя. Месячный оклад руководителя составляет 85000 рублей, а студента – 77763 рублей [9]. Стоимость человека-дня вычисляется путем деления месячного оклада исполнителя на количество рабочих дней в месяце (принимается равным 21 [8]).

Следовательно, стоимость человека дня составляет:

-для руководителя: $\text{СЧД}_{\text{рук}} = \frac{85000}{21} = 4047{,}6 \text{ руб/день}$;

-для студента $\text{СЧД}_{\text{студ}} = \frac{77763}{21} = 3703 \text{ руб/день}$.

Используя полученные значения стоимости человека дня для исполнителей и вычисленный объем работ, вычислим величину основной заработной платы:

-для руководителя: $З_{\text{осн.з.пл}} = 4047{,}6 * 5 = 20238 \text{ руб.}$;

-для студента: $З_{\text{осн.з.пл}} = 3703 * 43 = 159229 \text{ руб.}$;

Значит, общая основная заработная плата составляет:

$$З_{\text{осн.з.пл}} = 20238 + 159229 = 179467 \text{ руб.}$$

### 7.3.2 Расчет дополнительной заработной платы исполнителей

Расчет дополнительной заработной платы выполняется по формуле:

$$З_{\text{доп.з.пл}} = З_{\text{осн.з.пл}} * \frac{H_{\text{доп}}}{100},$$

где $H_{\text{доп}}$ – норматив дополнительной заработной платы.

Примем $H_{\text{доп}} = 14\%$ [8], следовательно расходы на дополнительную заработную плату составят:

$$З_{\text{доп.з.пл.}} = 179467 * 0{,}14 = 20925{,}38 \text{ руб.}$$

### 7.3.3 Расчет социальных отчислений

При расчете себестоимости продукта следует учесть отчисления на социальные нужды, которые включают в себя страховые взносы на обязательное социальное, пенсионное и медицинское страхование. Для расчета социальных отчислений используется формула:

$$З_{\text{соц}} = (З_{\text{осн.з.пл}} + З_{\text{доп.з.пл}}) * \frac{H_{\text{соц}}}{100},$$

Тариф страховых взносов на обязательное социальное, пенсионное и медицинское страхование в 2023 году составляет $H_{\text{соц}} = 30\%$, [14] следовательно социальные отчисления составляют:

$$З_{\text{соц}} = (179467 + 20925) * 0{,}3 = 60117{,}6 \text{ руб.}$$

## 7.4 Затраты на материалы, необходимые для выполнения работы

Так же следует оценить затраты на материалы, которые использовались в процессе работы.

Затраты на материалы, используемые в работе, рассчитываются по формуле:

$$З_м = \sum_{l=1}^{L} G_l Ц_l (1 + \frac{H_{т.з}}{100}),$$

Транспортные расходы принимаются равными 10%. [8]

Оценка затрат на сырье и материалы приведена в таблице 7.2

Таблица 7.2 – Затраты на сырье и материалы.

| Материал | Норма расхода, ед. | Цена за единицу, руб. | Сумма, руб. |
|---|---|---|---|
| Бумага А4 “Снегурочка” 500л. [10] | 60 | 1, 67 | 100, 33 |
| Картридж для принтера Canon LBP-2900 i-Sensys [11] | 1 | 1200, 00 | 1200, 00 |
| Транспортные расходы | | | 130, 03 |
| Итого | | | 1430, 36 |

### 7.5 Затраты на услуги сторонних организаций.

Для выполнения работы были использованы услуги компании Уют Телеком для предоставления доступа к сети Интернет. В качестве тарифа был выбран “Уют” с ежемесячной платой в размере 440 руб/мес вместе с НДС по ставке 20/120. [15] Стоимость услуги без НДС составляет:

$$440 - 440 * \frac{0,2}{1 + 0,2} = 366,66 \text{ руб.}$$

### 7.6 Затраты на содержание и эксплуатацию оборудования.

Расходы на электроэнергию так же необходимо учесть при оценке себестоимости разработки.

Во время выполнения работы электроэнергию потребляли следующие устройства:

- Ноутбук ASUS TUF GAMING FX 504GM-EN481T.
- Принтер Canon LBP-2900.

Потребляемая мощность ноутбуком составляет 0,45 кВт/час [12], а принтером 0,25 кВт/час. [13]

Время использования принтера – 1 рабочий час, ноутбука – 344 рабочих часа.

Учитывая, что тариф электроэнергии в дневное время в Санкт-Петербурге с 01.12.2022 по 31.12.2023 составляет 6,51 рубля за кВт/час, можно рассчитать суммарные затраты на электроэнергию:

$$6{,}51 * 0{,}45 * 344 + 6{,}51 * 0{,}25 * 1 = 1009{,}37 \text{ руб.}$$

### 7.7 Издержки на амортизационные отчисления

В ходе выполнения работы использовалось оборудование, представленное в таблице 7.3.

Таблица 7.3 – Список используемого оборудования

| Номенклатура | Цена, руб. |
|---|---|
| Ноутбук ASUS TUF GAMING FX 504GM-EN481T [12] | 54 999, 00 |
| Принтер Canon LBP-2900 [13] | 15 999, 00 |
| ИТОГО | 70998, 00 |

Для определения амортизационных начислений вычислительной техники используется следующая формула:

$$A_i = \text{Ц}_{\text{п.н.}i} \cdot \frac{\text{Н}_{ai}}{100},$$

где $A_i$– ежегодная сумма амортизационных отчислений i-ого объекта,

$\text{Ц}_{\text{п.н.}i}$– первоначальная стоимость объекта,

$H_{\text{а}}$– годовая норма амортизационных отчислений, рассчитывается по следующей формуле:

$$H_a = \frac{1}{\text{Т}_{\text{Н}}} * 100$$

где $Т_Н$– нормативный срок службы объекта (год). Для техники составляет 3 года.

Общая стоимость ежегодных амортизационных отчислений равна

$$A = \sum_{j=0}^{j} A_j,$$

где А – общая стоимость амортизационных отчислений,

$j$– количество рассматриваемых объектов.

Общие амортизационные отчисления за год составят:

$$A = 70998 * 0{,}33 = 23429{,}34 \text{ руб.}$$

Величина амортизационных отчислений по основным средствам, используемым в работе за период выполнения работы, рассчитывается по формуле:

$$A_{i\text{ВКР}} = A_i \cdot \frac{T_{i\text{ВКР}}}{365},$$

где $A_i$– отчисления за год по основному средству (руб.),

$T_{i\text{ВКР}}$– время, в течение которого используется основное средство (дни).

Следовательно, амортизационные отчисления по основным средствам за период выполнения работы составят:

$$A_{i\text{ВКР}} = 23429,34 \cdot \frac{43}{365} = 2760,16 \text{ руб}$$

### 7.8 Накладные расходы

К статье “накладные расходы” относятся расходы на управлении и хозяйственное обслуживание. Поскольку ВКР выполнялась на кафедре университета, в данной работе к накладным расходам можно отнести только лишь расходы на обеспечение доступа к сети интернет, которые были рассчитаны в пункте 5 раздела 7.

### 7.9 Себестоимость работы

В таблице 7.4 приведена смета затрат на ВКР.

Таблица 7.4 – Смета затрат на ВКР

| № п/п | Наименование статьи | Сумма, руб. |
|---|---|---|
| 1. | Расходы на оплату труда | 200392,38 |
| 2. | Отчисления на социальные нужды | 60117,6 |
| 3. | Материалы | 1430 |
| 4. | Затраты по работам, выполняемым сторонними организациями | 366, 66 |
| 5. | Затраты на содержание и эксплуатацию оборудования | 1009, 37 |
| 6. | Амортизационные отчисления | 2760 |
| ИТОГО затрат | | 266076,01 |

### 7.10 Выводы

В данном разделе была рассчитана себестоимость реализованных структур данных.

Общая сумма затрат составила 266076,01 рублей. Большая часть всех затрат связана с заработной платой исполнителей и отчислениями на социальные нужды.

Несмотря на то, что сумма разработки достаточно высока, нельзя не отметить ее полезность - разработанные структуры данных могут применяться в вычислительных системах с распределенной памятью для различного рода научных задач среди которых можно выделить:

- моделирование процессов.
- решение математических задач.
- выполнение расчетов.
- визуализация и представление данных.

И это лишь малая часть того, где смогут найти применение разработанные структуры.

# ЗАКЛЮЧЕНИЕ

В работе были рассмотрены два подхода к реализации разделяемых структур данных – блокирующий и неблокирующий. Для каждого из подходов были рассмотрены достоинства и недостатки, присущие им.

В результате выполнения выпускной квалификационной работы были разработаны 3 структуры данных в модели MPI RMA – связный список с использованием блокировок, неблокирующая очередь Майкла и Скотта и неблокирующий стек Трайбера.

Для каждой из реализованных структур были проведены тесты и замеры пропускной способности на вычислительном кластере, которые показали, что неблокирующая очередь и стек достаточно хорошо масштабируются как минимум до 16 процессов. В то же время связный список с использованием блокировок показал гораздо меньшую масштабируемость из-за выбора блокирующей синхронизации.

Реализованные структуры данных могут найти применение на вычислительных системах с распределенной памятью, например, для реализации более сложных структур данных (очередь с приоритетом, развёрнутый связный список и т.д.) или для хранения и упорядочивания данных, полученных в ходе сложных вычислений.

Для дальнейшего развития работы можно отойти от идеи централизации данных в памяти корневого процесса, так как это может позволить увеличить пропускную способность структур данных.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Maurice Herliny, Nir Shavit, Victor Luchangco The Art of Multiprocessor programming = Искусство мультипроцессорного программирования. М.: Newnes, 2020. 576 с.

2. MPI Forum // URL: https://www.mpi-forum.org/ (дата обращения: 21.04.2023).

3. Богачёв К.Ю. Основы параллельного программирования. М.: Бином, 2015. 342с.

4. Уильямс Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. М.: ДМК Пресс, 2012. 672 с.

5. Maged M. Michel, Michael L. Scott. // Simple, Fast, and Practical Non-Blocking Concurrent Queue Algorithms = Простые, быстрые и практичные алгоритмы конкуретной очереди // In PODC.ACM. 1995.

6. Treiber R. Kent // System programming: Coping with parallelism = Системное программирование: борьба с параллелизмом // Technical Report RJ 5118, IBM Almaden Research Center. 1986.

7. Mvapich // URL: https://mvapich.cse.ohio-state.edu (дата обращения: 24.04.2023).

8. Алексеева О. Г. Методические указания по технико-экономическому обоснования выпускных квалификационных работ бакалавров: Метод. указания, СПб.: Из-во СПбГЭТУ “ЛЭТИ”, 2013.

9. ГородРабот // URL: https://gorodrabot.ru/salary?p=инженер&l=санкт-петербург (дата обращения: 31.05.2023).

10. https://fix-price.com/catalog/kantstovary/p-5710265-bumaga-snegurochka-a4-500-listov (дата обращения 31.05.2023)

11.https://imprints.ru/toner-cartridges/canon/lbp-2900-i-sensys/ (дата обращения: 31.05.2023)

12. https://www.dns-shop.ru/product/cc717b58520c3332/156-noutbuk-asus-tuf-gaming-fx504gm-en481t-seryj/ (дата обращения: 31.05.2023)

13.https://www.dns-shop.ru/product/6c8521d9bba252d7/printer-lazernyj-canon-lbp-2900/ (дата обращения: 31.05.2023)

14.https://www.nalog.gov.ru/rn30/news/activities_fts/13055866/ (дата обращения: 31.05.2023)

15. https://uut-telecom.ru/tarify-internet/ (дата обращения 31.05.2023)

# ПРИЛОЖЕНИЕ А

## Программный код реализованных структур.

```
/*

    Связный список с использованием блокировок
    Реализовал: Е. В. Епифанцев
    Дата: 30.04.2023
*/

typedef struct { // Структура указателя на элемент списка
    int rank; // ранг окна
    MPI_Aint disp; // смещение внутри окна
} nodePtr;

typedef struct { // Структура элемента списка
    int id;     // Ключ элемента
    int val; // Пользовательские данные
    char logicalyDeleted; // Маркер логического удаления
    char canBeReclaimed; // Маркер освобождения памяти
    nodePtr next; // Указатель на следующий элемент списка
} node;

// Функция выделения памяти под элемент и присоединения его к окну (ключ,
// значение, ранг процесса, окно)
MPI_Aint allocElem(int id, int val, int rank, MPI_Win win) {
    MPI_Aint disp;
    node* allocNode;

    // Перед выделением памяти проверяем наличие свободных
    // элементов в пуле
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, 0, win);
          for(int i = 0; i < allocNodeCount; i++){
                // Если обнаружен удаленный элемент, то использовать его
                if(allocNodes[i]->canBeReclaimed == 1){
                      allocNodes[i]->id = id;
                      allocNodes[i]->val = val;
                      allocNodes[i]->next = nullPtr;
                      allocNodes[i]->logicalyDeleted = 0;
                      allocNodes[i]->canBeReclaimed = 0;
                      reclaimedNodes++;
                      MPI_Get_address(allocNodes[i], &disp);
                      MPI_Win_unlock(rank, win);
                      return disp;
                }
          }
    MPI_Win_unlock(rank, win);

    MPI_Alloc_mem(sizeof(node), MPI_INFO_NULL, &allocNode);

    // Заполняем поля структуры
    allocNode->id = id;
    allocNode->val = val;
    allocNode->next = nullPtr;
    allocNode->logicalyDeleted = 0;
    allocNode->canBeReclaimed = 0;

    // Присоединяем память к окну
    MPI_Win_attach(win, allocNode, sizeof(node));
```

```
        if (allocNodeCount == allocNodeSize) { // Если размер пула элементов
        //недостаточен, расширяем его
              allocNodeSize += 100;
              allocNodesTmp = (node**)realloc(allocNodes, allocNodeSize *
              sizeof(node*));
              if (allocNodesTmp != NULL)
                    allocNodes = allocNodesTmp;
              else {
                    printf("Error while allocating memory!\n");
                    return -1;
              }
        }

        // Запоминаем выделенный элемент в пуле
        allocNodes[allocNodeCount] = allocNode;
        allocNodeCount++;

        // Получаем смещение внутри окна
        MPI_Get_address(allocNode, &disp);

        // Возвращаем адрес смещения нового элемента внутри окна
        return disp;
  }

  // Функция поиска элемента в списке по ключу
  // (ключ, голова списка, окно)
  nodePtr search(int id, nodePtr head, MPI_Win win)
  {
        nodePtr curNodePtr = head, next = {0};
        node curNode = {0};

        // Пока не дошли до конца списка
        while(curNodePtr.rank != -1){

         // Захватить Lock на окно с текущим элементом
         MPI_Win_lock(MPI_LOCK_EXCLUSIVE, curNodePtr.rank, 0, win);

         // Считать текущий элемент
         MPI_Get((void*)&curNode, sizeof(node), MPI_BYTE, curNodePtr.rank,
         curNodePtr.disp,sizeof(node), MPI_BYTE, win);

         MPI_Win_flush(curNodePtr.rank, win);

         // Если ключ текущего элемента искомый, вернуть его из функции и
         //отпустить Lock
         if(curNode.id == id) {

          MPI_Win_unlock(curNodePtr.rank, win);

          return curNodePtr;
         } else next = curNode.next;

         MPI_Win_unlock(curNodePtr.rank, win);

         curNodePtr = next;
        }
        // Если элемент не найден, то вернуть null
        return nullPtr;
  }
```

```
// Функция вставки нового элемента в список (ключ нового элемента, новое
//значение, ключ вставки, ранг процесса, голова списка, окно)
void insertAfter(int id, int newVal, int key, int rank, nodePtr head, MPI_Win
win)
{
      node curNode;
      nodePtr newNode, curNodePtr, fetched;

      // Вызвать функцию поиска элемента по заданному ключу
      curNodePtr = search(key, head, win);

      // Если элемент не найден, выйти из функции
      if(curNodePtr.rank == nullPtr.rank) {
            failedIns++;
            return;
      }

      // Если элемент найден
      else {
          // Создать новый элемент
          newNode.rank = rank;
          newNode.disp = allocElem(id, newVal, rank, win);

          if(newNode.disp == -1) return;

          // Захватить Lock на окно с найденным элементом
          MPI_Win_lock(MPI_LOCK_EXCLUSIVE, curNodePtr.rank, 0,win);

          // Считать поля найденного элемента
          MPI_Get((void*)&curNode, sizeof(node), MPI_BYTE, curNodePtr.rank,
          curNodePtr.disp, sizeof(node), MPI_BYTE, win);

          MPI_Win_flush(curNodePtr.rank, win);


          // Если элемент не был логически удален, заменяем указатель next
          //нового элемента на найденный элемент по ключу

          if(curNode.logicalyDeleted != 1){
            MPI_Fetch_and_op(&newNode.rank, &fetched.rank, MPI_INT,
            curNodePtr.rank,curNodePtr.disp + offsetof(node, next.rank),
            MPI_REPLACE, win);

            MPI_Fetch_and_op(&newNode.disp, &fetched.disp, MPI_AINT,
            curNodePtr.rank, curNodePtr.disp + offsetof(node, next.disp),
            MPI_REPLACE, win);

            MPI_Win_flush(curNodePtr.rank, win);

            ((node*)newNode.disp)->next.rank = fetched.rank;
            ((node*)newNode.disp)->next.disp = fetched.disp;
            successIns++;
          } else {
            // Иначе если элемент был логически удален, отпустить // Lock и
            вернуться из функции
            MPI_Win_unlock(curNodePtr.rank, win);
            failedIns++;
            return;
          }
    MPI_Win_unlock(curNodePtr.rank, win);
  }
}
```

```
// Вспомогательная функция удаления элемента
int traverseAndDelete(nodePtr head, nodePtr nodeToDelete, nodePtr
nextAfterDeleted, MPI_Win win)
{
     nodePtr curNodePtr = head, next = {0};
     node curNode = {0};

     // Пока не обнаружен конец списка
     while(curNodePtr.rank != -1){

       // Захватить Lock на окно с текущим элементом
       MPI_Win_lock(MPI_LOCK_EXCLUSIVE, curNodePtr.rank, 0, win);
       // Считать текущий элемент
       MPI_Get((void*)&curNode, sizeof(node), MPI_BYTE, curNodePtr.rank,
       curNodePtr.disp, sizeof(node), MPI_BYTE, win);

       MPI_Win_flush(curNodePtr.rank, win);

       // Если next указатель считанного элемента указывает на узел, который
       // нужно удалить
       if(curNode.next.rank == nodeToDelete.rank && curNode.next.disp ==
       nodeToDelete.disp) {
         // Если элемент не был логически удален, то изменить его next
         //указатель
         if(curNode.logicalyDeleted == 0){

             // Поменять next указатель
             MPI_Put((void*)&nextAfterDeleted, sizeof(nodePtr), MPI_BYTE,
             curNodePtr.rank,curNodePtr.disp + offsetof(node, next),
             sizeof(nodePtr), MPI_BYTE, win);

             // Отпутить Lock
             MPI_Win_unlock(curNodePtr.rank, win);
             return 1;
         } else {
             MPI_Win_unlock(curNodePtr.rank, win);
             return 0;
         }
       } else {
         next = curNode.next;
         MPI_Win_unlock(curNodePtr.rank, win);
       }
     // Продолжить проход по списку
     curNodePtr = next;
    }
}

// Функция удаления элемент из списка (ключ, голова списка, окно)
void Delete(int key, nodePtr head, MPI_Win win)
{
      nodePtr curNodePtr, next;
      node curNode;
      char mark = 1, emptyMark = 0, result = 0, free = 1;
      int testDelete = 0;

      // Вызвать функцию поиска элемент по заданному ключу
      curNodePtr = search(key, head, win);

      // Если элемент не найден, вернуться из функции
      if(curNodePtr.rank == nullPtr.rank) {
         failedDel++;
         return;
```

```
      // Если элемент найден
      } else {
          // Захватить Lock на окно с найденным элементом
          MPI_Win_lock(MPI_LOCK_EXCLUSIVE, curNodePtr.rank, 0, win);

          // С помощью CAS установить маркер логического удаления
          MPI_Compare_and_swap((void*)&mark, (void*)&emptyMark,
          (void*)&result, MPI_BYTE, curNodePtr.rank, curNodePtr.disp +
          offsetof(node, logicalyDeleted), win);

          MPI_Win_flush(curNodePtr.rank, win);

          if(result != 1) {
               MPI_Get((void*)&next, sizeof(nodePtr),MPI_BYTE,curNodePtr.rank,
               curNodePtr.disp + offsetof(node, next),sizeof(nodePtr),
               MPI_BYTE, win);

               MPI_Win_unlock(curNodePtr.rank, win);

               // Пока удаление на завершится успешно вызывать функцию
               //traverseAndDelete
               if(result != 1) {
                    while(testDelete == 0){
                    testDelete = traverseAndDelete(head, curNodePtr, next,
                    win);
               }
               MPI_Win_lock(MPI_LOCK_EXCLUSIVE, curNodePtr.rank, 0, win);

               // Установить маркер того, что память под элемент может теерь
               //быть использована заново
               MPI_Put((void*)&free, 1, MPI_INT,
               curNodePtr.rank,curNodePtr.disp + offsetof(node,
               canBeReclaimed), 1, MPI_INT, win);

               MPI_Win_unlock(curNodePtr.rank, win);
               successDel++;
      } else failedDel++;
    }
}

/*

      Очередь Майкла и Скотта
      Реализовал: Е. В. Епифанцев
      Дата: 08.05.2023
*/

typedef struct { // Структура указателя на элемент очереди
      uint64_t rank : 11; // Ранг окна
      uint64_t offset : 53; // Смещение внутри окна
} nodePtr;

typedef struct { // Структура элемента очереди
      int val; // Пользовательские данные
      nodePtr next; // Указатель на следующий элемент очереди
} node;

typedef struct{ // Структура очереди
      nodePtr dummy; // указатель на dummy узел
      nodePtr head; // Указатель на голову
      nodePtr tail; // Указатель на хвост
} Queue;
```

```
// Функция получения хвоста очереди (очередь, окно)
nodePtr getTail(Queue q, MPI_Win win)
{
      nodePtr tail = { 0 }, curNodePtr = { 0 };

      // Атомарно считать в переменную tail текущий хвост очереди
      MPI_Fetch_and_op(NULL, (void*)&tail, MPI_LONG_LONG, 0,
      q.tail.offset + offsetof(node, next), MPI_NO_OP, win);

      MPI_Win_flush(0, win);

      // Вернуть текущий хвост
      return tail;
}

// Функция вставки нового элемент в очередь (новое значение, ранг, очередь,
// окно)
void enq(int val, int rank, Queue q, MPI_Win win)
{
      nodePtr newNode = { 0 }, result = { 0 }, tmpTail = { 0 }, tmpTailUpdated
      = { 0 }, tailNext = { 0 };

      // Создать новый элемент
      newNode.rank = rank;
      newNode.offset = allocElem(val, rank, win);

      // В CAS-цикле
      while(1){
           // Считать текущий хвост в переменную tmpTail
           tmpTail = getTail(q, win);

           // С помощью CAS вставить новый элемент в очередь
           MPI_Compare_and_swap((void*)&newNode, (void*)&nullPtr,
           (void*)&result, MPI_LONG_LONG,tmpTail.rank, tmpTail.offset +
           offsetof(node, next), win);

           MPI_Win_flush(tmpTail.rank, win);

           // Если вставке удалась
           if(result.rank == nullPtr.rank){

                // Попытаться перенести указатель хвоста на новый элемент
                MPI_Compare_and_swap((void*)&newNode, (void*)&tmpTail,
                (void*)&result, MPI_LONG_LONG,0, q.tail.offset +
                offsetof(node, next), win);

                MPI_Win_flush(0, win);

                succEnq++;
                return;

           // Если вставка не удалась, значит другой процесс вставил свой
           //элемент, нужно помочь перенести хвост очереди
           } else {
           tailNext = getTail(q, win);
           MPI_Compare_and_swap((void*)&tailNext, (void*)&tmpTail,
           (void*)&result, MPI_LONG_LONG,0, q.tail.offset + offsetof(node,
           next), win);

           MPI_Win_flush(0, win);
           }
      }
}
```

```
// Функция получения текущей головы очереди (очередь, окно)
nodePtr getHead(Queue q, MPI_Win win)
{
      nodePtr result = { 0 };

      // Атомарно считать в переменную result текущую голову
      MPI_Fetch_and_op(NULL, (void*)&result, MPI_LONG_LONG, 0,
      q.head.offset + offsetof(node, next), MPI_NO_OP, win);
      MPI_Win_flush(0, win);

      // Вернуть result
      return result;
}

// Функция считывания следующего за головным элемента (указатель на голову,
// окно)
nodePtr getNextHead(nodePtr head, MPI_Win win)
{
      nodePtr result = { 0 };

      // Если голова null, то выйти из функции
      if(head.rank == nullPtr.rank) return head;

      // Атомарно считать следующий за головным элемент в result
      MPI_Fetch_and_op(NULL, (void*)&result, MPI_LONG_LONG,
      head.rank,head.offset + offsetof(node, next), MPI_NO_OP, win);

      MPI_Win_flush(head.rank, win);

      // Вернуть result
      return result;
}

// Функция считывания элемента по указателю (указатель на элемент, окно)
int readVal(nodePtr ptr, MPI_Win win)
{
      int result = 0;

      // В переменную result считать данные по ptr
      MPI_Get((void*)&result, 1, MPI_INT, ptr.rank, ptr.offset +
      offsetof(node, val),1, MPI_INT, win);

      MPI_Win_flush(ptr.rank, win);

      printf("Val %d\n", result);
}

// Функция удаления элемента из очереди (очередь, окно)
void deq(Queue q, MPI_Win win)
{
      nodePtr tail = { 0 }, head = { 0 }, afterHead = { 0 }, result = { 0 };

      // В CAS-цикле
      while(1){
          // Считать текущую голову в head
          head = getHead(q, win);

          // Считать текущий хвост в tail
          tail = getTail(q, win);

          // Считать следующий за головным элемент в afterHead
          afterHead = getNextHead(head, win);
```

```
          // Если голова и хвост совпадают
          if(tail.rank == head.rank && tail.offset == head.offset){
               // Если следующий после head элемент null, то очередь пуста
               if(afterHead.rank == nullPtr.rank) {
                   return;
               } else {

               // Помочь перенести хвост очереди
               MPI_Compare_and_swap((void*)&afterHead,
               (void*)&tail,(void*)&result, MPI_LONG_LONG,0, q.tail.offset +
               offsetof(node, next), win);

               MPI_Win_flush(0, win);
               }

          // Голова и хвост не совпадают
          } else {

               // С помощью CAS извлечь элемент из очереди
               MPI_Compare_and_swap((void*)&afterHead, (void*)&head,
               (void*)&result, MPI_LONG_LONG,0, q.head.offset + offsetof(node,
               next), win); MPI_Win_flush(0, win);

               if(result.rank == head.rank && result.offset == head.offset) {
               readVal(afterHead, win);
               succDeq++;
               return;
          }
         }
       }
}


/*

      Стек Трайбера
      Реализовал: Е. В. Епифанцев
      Дата: 13.05.2023
*/

typedef struct { // Структура указателя на элемент стека
      uint64_t rank : 11; // Ранг окна
      uint64_t offset : 53; // Смещение внутри окна
} nodePtr;

typedef struct { // Структура элемента стека
      int val; // Значение элемента
      nodePtr next; // Указатель на следующий элемент стека
} node;

typedef struct{ // Структура стека
      nodePtr dummy; // Указатель на dummy-элемент
      nodePtr head; // Указатель на голову
} Stack;


// Функция чтения значения элемента по указателю (указтель на элемент, окно)
int readVal(nodePtr ptr, MPI_Win win)
{
      int result = 0;
```

```
      // Считать значение элемента в result
      MPI_Get((void*)&result, 1, MPI_INT, ptr.rank, ptr.offset +
      offsetof(node, val),1, MPI_INT, win);

      MPI_Win_flush(ptr.rank, win);

      printf("Val %d\n", result);
}

// Функция считывания текущей головы стека (стек, окно)
nodePtr getHead(Stack s, MPI_Win win)
{
      nodePtr result = {0};

      // Атомарно считать в переменную result текущую голову стека
      MPI_Fetch_and_op(NULL, (void*)&result, MPI_LONG_LONG, s.dummy.rank,
      s.dummy.offset + offsetof(node, next), MPI_NO_OP, win);

      MPI_Win_flush(s.dummy.rank, win);

      return result;
}

// Функция изменения указателя next newHead на oldHead (указатель на oldHead,
// указатель на newHead, окно)

void changeNext(nodePtr oldHead, nodePtr newHead, MPI_Win win){
      nodePtr result = { 0 };

      //Атомарно изменить указатель newHead.next на oldHead
      MPI_Fetch_and_op((void*)&oldHead, (void*)&result, MPI_LONG_LONG,
      newHead.rank,newHead.offset + offsetof(node, next), MPI_REPLACE, win);

      MPI_Win_flush(newHead.rank, win);

}

// Функция вставки нового элемента в стек (новое значение, ранг, стек, окно)
void push(int val, int rank, Stack s, MPI_Win win)
{
      nodePtr curHead = {0}, newHead = {0}, result = {0};

      // Создать новый элемент
      newHead.rank = rank;
      newHead.offset = allocElem(val, win);

      // В CAS-цикле
      while(1){
          // Считать текущую голову в переменную curHead
          curHead = getHead(s, win);

          // Изменить указатель newHead.next на curHead
          changeNext(curHead, newHead, win);

          MPI_Compare_and_swap((void*)&newHead, (void*)&curHead,
          (void*)&result, MPI_LONG_LONG, s.dummy.rank, s.dummy.offset +
          offsetof(node, next), win);

          MPI_Win_flush(s.dummy.rank, win);

          if(result.rank == curHead.rank && result.offset == curHead.offset) {
               succPush++;
               return;
```

```
            }
        }
}

// Функция считывания следующего за головным элемента
nodePtr getNextHead(nodePtr head, MPI_Win win)
{
        nodePtr result = {0};

        // Считать атомарно в переменную result следующий за головным элемент
        MPI_Fetch_and_op(NULL, (void*)&result, MPI_LONG_LONG,
        head.rank,head.offset + offsetof(node, next), MPI_NO_OP, win);

        MPI_Win_flush(head.rank, win);

        // Вернуть result
        return result;
}

void pop(Stack s, MPI_Win win)
{
        nodePtr curHead = {0}, result = {0}, nextHead = {0};

        //В CAS-цикле
        while(1){
                // Считать текущую голову в curHead
                curHead = getHead(s, win);

                // Если голова null, стек пуст, вернуться из функции
                if(curHead.rank == nullPtr.rank) return;

                // Считать следующий за головным элемент
                nextHead = getNextHead(curHead, win);

                // С помощью CAS выполнить удаление текущей головы
                MPI_Compare_and_swap((void*)&nextHead, (void*)&curHead,
                (void*)&result, MPI_LONG_LONG, s.dummy.rank, s.dummy.offset +
                offsetof(node, next), win);

                MPI_Win_flush(s.dummy.rank, win);

                if(result.rank == curHead.rank && result.offset ==
                curHead.offset) {
                        succPop++;
                        return;
                }
        }
}
```