«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

К защите допустить

Зав. 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 _____ И.С. Зуев

Санкт-Петербург

2023

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Утверждаю
Зав. кафедрой ВТ
_____ М.С. Куприянов
«___»_____20___ г.

Студент Е.В. Епифанцев Группа 9305

1. Тема: Реализация разделяемых структур данных в модели MPI RMA

Место выполнения ВКР: кафедра ВТ

2. Объект и предмет исследования:

Объектом исследования являются разделяемые структуры данных.

Предметом исследования являются связный список с использованием блокировок, неблокирующая очередь Майкла и Скотта, неблокирующий стек Трайбера

3. Цель: Разработать набор базовых структур данных (связный список, очередь и стек) в модели MPI RMA.

4. Исходные данные: Научные статьи, посвященные разделяемым структурам данных, стандарт MPI, документация OpenMPI.

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

6. Технические требования: структуры должны быть разработаны в модели MPI RMA и должны корректно работать на вычислительном кластере.

7. Дополнительные разделы: Экономическое обоснование ВКР.

8. Результаты: Пояснительная записка, исходный код разработанных структур данных на языке программирования С.

Дата выдачи задания «_14_»_____марта_____2023_ г.

Дата представления ВКР к защите «_22_»_____июня_____2023_ г.

Студент _____ Е.В. Епифанцев

Руководитель _____ А.А. Пазников

КАЛЕНДАРНЫЙ ПЛАН ВЫПОЛНЕНИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Утверждаю
Зав. кафедрой ВТ
_____ М.С. Куприянов
«___»_____20___ г.

Студент Е.В. Епифанцев Группа 9305

Тема работы: Реализация разделяемых структур данных в модели MPI RMA

№ п/п	Наименование работ	Срок выполнения
1	Описание разделяемых структуры данных	19.04 – 20.04
2	Изучение и описание модели MPI RMA	21.04 – 27.04
3	Реализация связного списка с использованием блокировок	28.04 – 30.04
4	Реализация неблокирующей очереди Майкла и Скотта	02.05 – 07.05
5	Реализация неблокирующего стека Трайбера	08.05 – 13.05
6	Тестирование полученных структур данных, измерение их пропускных способностей	15.05 – 16.05
7	Оформление пояснительной записки	25.05 – 08.06
8	Представление работы к защите	22.06.2023

Студент _____ Е.В. Епифанцев

Руководитель _____ А.А. Пазников

РЕФЕРАТ

Тема: Реализация разделяемых структур данных в модели MPI RMA.

Цель: разработка базовых структур данных (связный список, очередь и стек) в модели MPI RMA.

Объектом исследования являются разделяемые структуры данных.

Предметом исследования являются: связный список с использованием блокировок, неблокирующая очередь Майкла и Скотта, неблокирующий стек Трайбера в модели удаленного доступа к памяти (MPI RMA).

Результат работы: в рамках работы были разработаны три структуры данных – связный список с использованием блокировок, неблокирующая очередь Майкла и Скотта и неблокирующий стек Трайбера в модели удаленного доступа к памяти (MPI RMA). Для каждой из реализованных структуры данных были проведены тесты, с целью измерения и оценки их пропускной способности.

Полученные структуры данных могут применяться в вычислительных системах с распределенной памятью, например в кластерах, в задачах, где выполняются 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

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.

В настоящее время для создания программ для распределенных вычислительных систем активно применяется модель удаленного доступа к памяти (MPI RMA). Данная модель позволяет сокращать время выполнения программ по сравнению с моделью передачи сообщений.

Разделяемая структура данных – это структура данных, оперировать одновременно которой могут несколько процессов или потоков в рамках процесса.

Большинство исследований в области разделяемых структур данных сфокусировано на создании структур, предназначенных для использования в системах с общей памятью. Однако, эти структуры и связанные с ними алгоритмы не могут быть применены напрямую в системах с распределенной памятью. Таким образом, возникает задача разработки разделяемых структур данных, которые могут работать в вычислительных системах с распределенной памятью.

Целью данной работы является создание набора разделяемых структур данных, которые могут быть применены в системах с распределенной памятью и должны быть реализованы в модели MPI RMA.

1. РАЗДЕЛЯЕМЫЕ СТРУКТУРЫ ДАННЫХ

1.1 Описание.

Разделяемая структура данных 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.

1.2 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 $i \neq j$.

1.3. Подходы к реализации.

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. Неблокирующие алгоритмы делятся на три типа:

1) 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;
2) 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;
3) 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.

1.4. Трудности, возникающие при программной реализации.

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

1) 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;

2) 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;
3) 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;
4) 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;
5) Отладка - как правило, отладка и исправление ошибок в многопоточном коде может быть трудоемким и требовать специальных инструментов и методик;

2. МОДЕЛЬ MPI RMA

MPI RMA (Remote Memory Access) – 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.

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.

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 целевого 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.

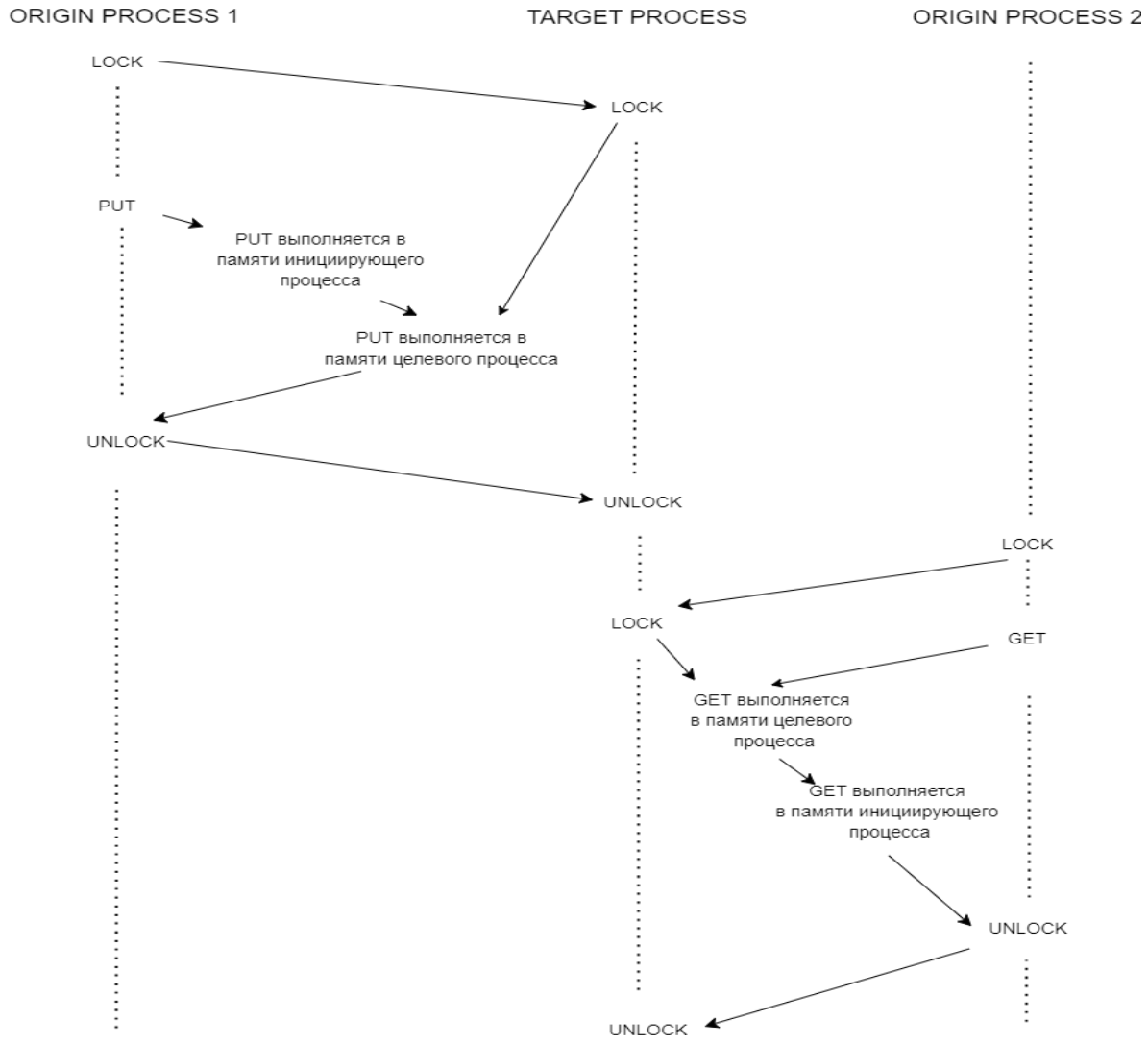

Рисунок 2.1. Схема пассивной синхронизации.

2.3. 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.

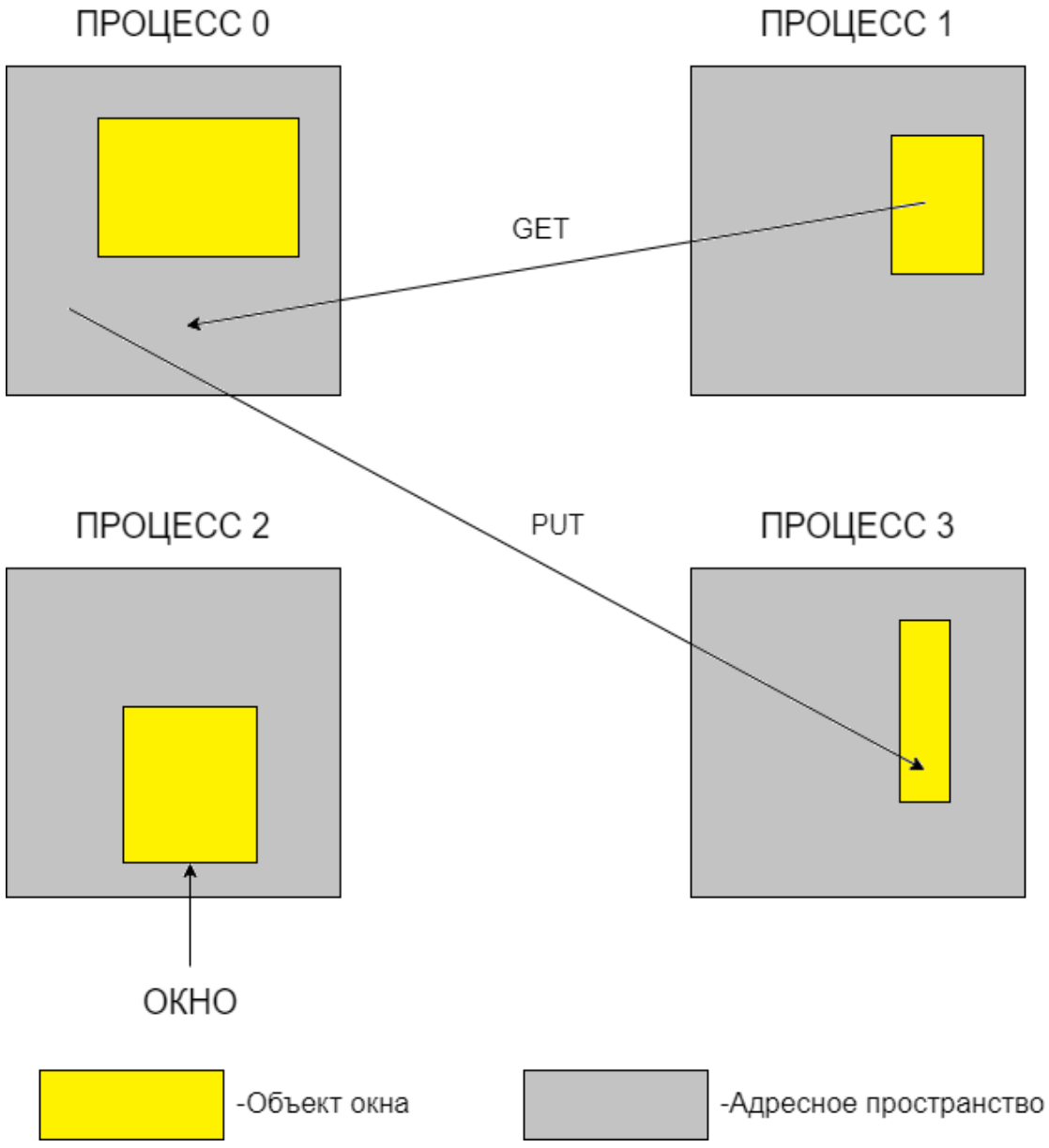

Рисунок 2.2 Пример коммуникации процессов с использованием MPI RMA.

Для работы с окнами в MPI реализованы следующие функции:

1) 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;
2) 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;
3) 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;
4) 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;
5) MPI_Win_attach() – эта функция используется для присоединения сегмента памяти, созданного с помощью функции MPI_Alloc_mem(), к динамическому окну;
6) MPI_Win_detach() – эта функция используется для отсоединения сегмента памяти от динамического окна;
7) 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;

8) 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:

1) MPI_Put() – эта операция позволяет записать данные из локального сегмента памяти процесса в удаленный сегмент памяти другого процесса. Данная операция является неблокирующей;

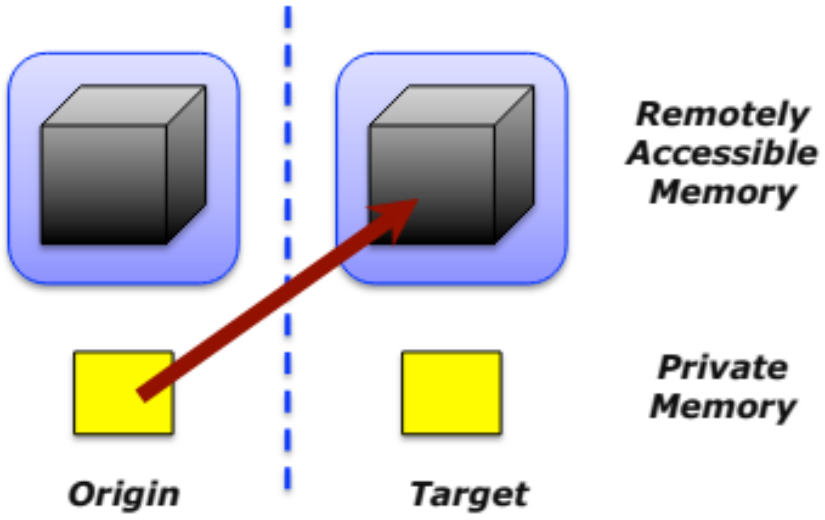

Рисунок 2.3. Порядок работы функции MPI_Put().

2) MPI_Get() – эта операция позволяет прочитать данные из удаленного сегмента памяти другого процесса и сохранить их в локальной памяти. Данная операция является неблокирующей;

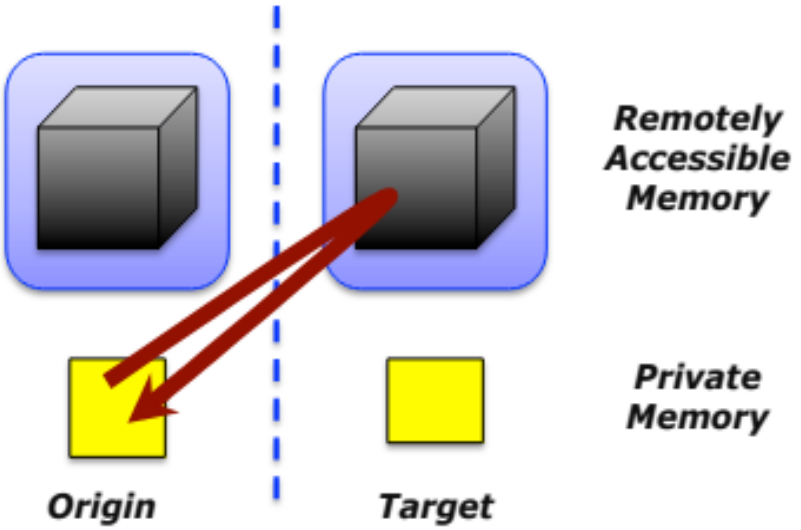

Рисунок 2.4. Порядок работы функции MPI_Get().

3) 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;

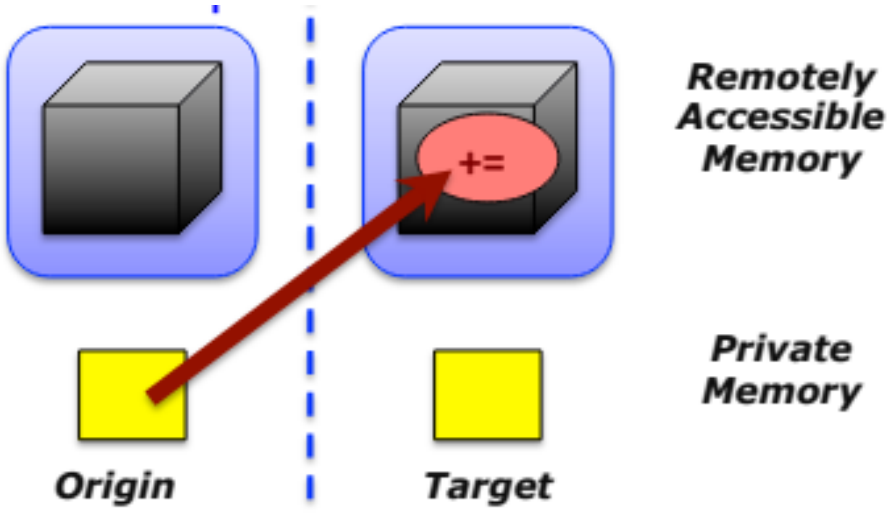

Рисунок 2.5. Порядок работы функции MPI_Accumulate().

4) 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;

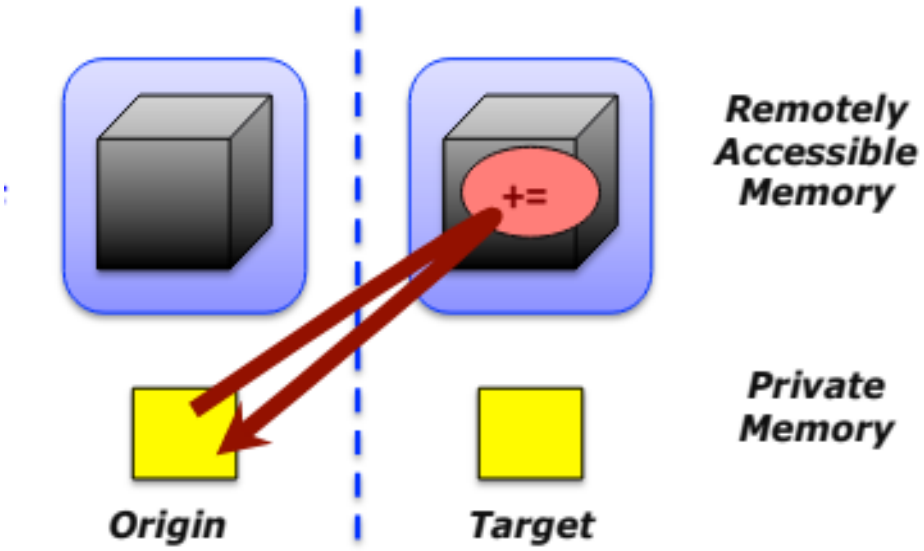

Рисунок 2.6. Порядок работы функции MPI_Get_accumulate().

5) MPI_Fetch_and_op() – данная операция выполняет чтение и некоторую атомарную операцию над удаленным сегментом памяти другого процесса;
6) MPI_Compare_and_swap() – эта операция выполняет атомарное сравнение и обмен значений в удаленном сегменте памяти другого процесса;
7) 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;
8) 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 схематическое представление распределенного связного списка:

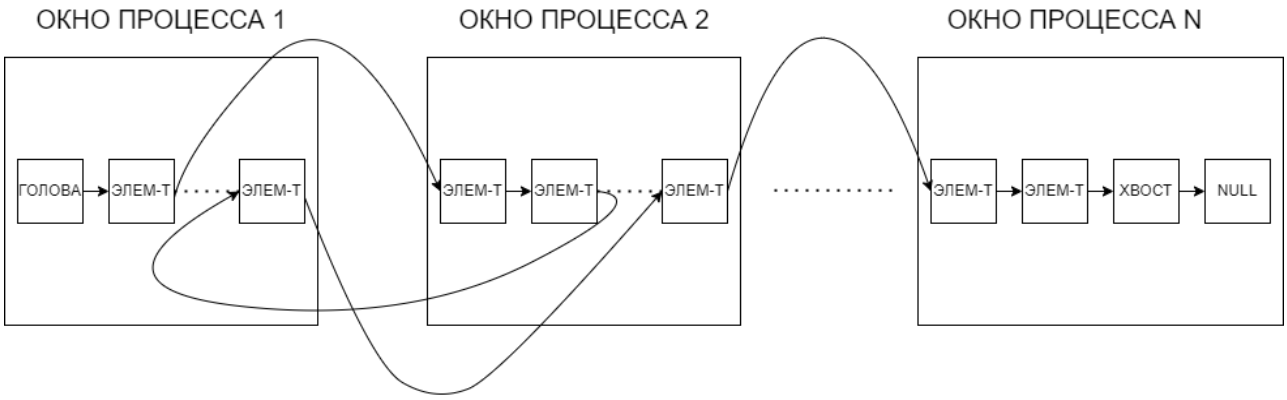

Рисунок 3.1. Структура распределенного связного списка.

3.1. Описание представления элемента списка и указателя на него.

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

1) Id – уникальный номер элемента в списке (ключ);
2) Val – пользовательские данные;
3) logicalyDeleted – флаг, сигнализирующий о том, что данный элемент был логически удален из списка, но все еще является доступным для других процессов;

4) canBeReclaimed – флаг, сигнализирующий о том, что данный узел был логически удален из списка, и теперь память, выделенная под данный элемент может быть использована заново;
5) next – указатель на следующий элемент списка.

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

1) rank – ранг процесса, в окне которого расположен элемент;
2) 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:

1) Lock() – функция MPI_Win_lock();
2) Get() – функция MPI_Get();
3) Flush() – функция MPI_Win_flush();
4) Unlock() – функция MPI_Win_unlock();
5) Fetch() – функция MPI_Fetch_and_op();
6) Put() – функция MPI_Put();
7) CAS() – функция MPI_Compare_and_swap();
8) EmptyNode – пустой узел списка;
9) Head – указатель на голову списка;

Опишем алгоритм поиска элемента и приведем его псевдокод:

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:

1) Вызвать функцию поиска элемента, после которого необходимо выполнить вставку (4 строка);

2) Если элемент не найден в списке, то вставка не удалась, выйти из функции (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, а значит изменение указателя next в нем запрещена - вставка не удалась. Нужно отпустить Lock и вернуться из функции (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:

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.

```
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.

3.4. Тестирования списка на одном вычислительном узле.

Результат запуска программы приведен на рисунке 6. Запуск производился на компьютере с процессором Intel Core i5-8300H (4 ядра). Базовая частота процессора – 2,3 ГГц, максимальная – 4.0 ГГц. При запуске процесс с рангом 0 изначально формировал связный список длины 1000 элементов, а затем каждый из процессов производил по 1000 операций над этим списком. Тип операции выбирался случайно. Для контроля корректности полученного результата было введено два счетчика – totalElementCount (фактическое количество элементов в полученном списке) и expectedElementCount (то количество элементов, которое ожидается, оно вычисляется на основе выполненных операций). Если оба этих счетчика совпадают, то тест пройден успешно.

Для подсчета общего времени использовалась функция MPI_Wtime().

```
-----id 33011: val 2000 was inserted by rank 1 at displacement 4dd6d90 marked 0 next rank 2 next displacement c5705290-----
-----id 24016: val 2000 was inserted by rank 2 at displacement c5705290 marked 0 next rank 3 next displacement ee5d52c0-----
-----id 24024: val 2000 was inserted by rank 3 at displacement ee5d52c0 marked 0 next rank 1 next displacement 4dd5320-----
-----id 24008: val 2000 was inserted by rank 1 at displacement 4dd5320 marked 0 next rank 1 next displacement 4dd52c0-----
-----id 12004: val 2000 was inserted by rank 1 at displacement 4dd52c0 marked 0 next rank 1 next displacement 4dd5290-----
-----id 9003: val 2000 was inserted by rank 1 at displacement 4dd5290 marked 0 next rank -1 next displacement 0-----
Rank 0 success insert = 1088, failed insert = 384, success delete = 49, failed delete = 478, reclaimed = 89
Rank 1 success insert = 289, failed insert = 214, success delete = 262, failed delete = 235, reclaimed = 0
Rank 2 success insert = 235, failed insert = 256, success delete = 226, failed delete = 283, reclaimed = 0
Rank 3 success insert = 382, failed insert = 116, success delete = 373, failed delete = 129, reclaimed = 0
Total element count = 1084
Expected element count = 1084
List Integrity: True
Test result: total elapsed time = 7.724936 ops/sec = 517.803649
```

Рисунок 3.5. Результат запуска списка на одном узле.

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 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 проблемы 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:

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.

4.3. Тестирование очереди на одном вычислительном узле.

Результат запуска программы приведен на рисунке 4.3. Запуск производился на компьютере с процессором Intel Core i5-8300H (4 ядра). Базовая частота процессора – 2,3 ГГц, максимальная – 4.0 ГГц. Каждый из процессов производил порядка 20000 операций вставки/удаления, тип операции выбирался случайно.

Каждый процесс вел локальный счетчик успешно добавленных/удаленных элементов, а затем отсылал результаты корневому процессу для проверки целостности очереди.

Для подсчета общего времени использовалась функция MPI_Wtime().

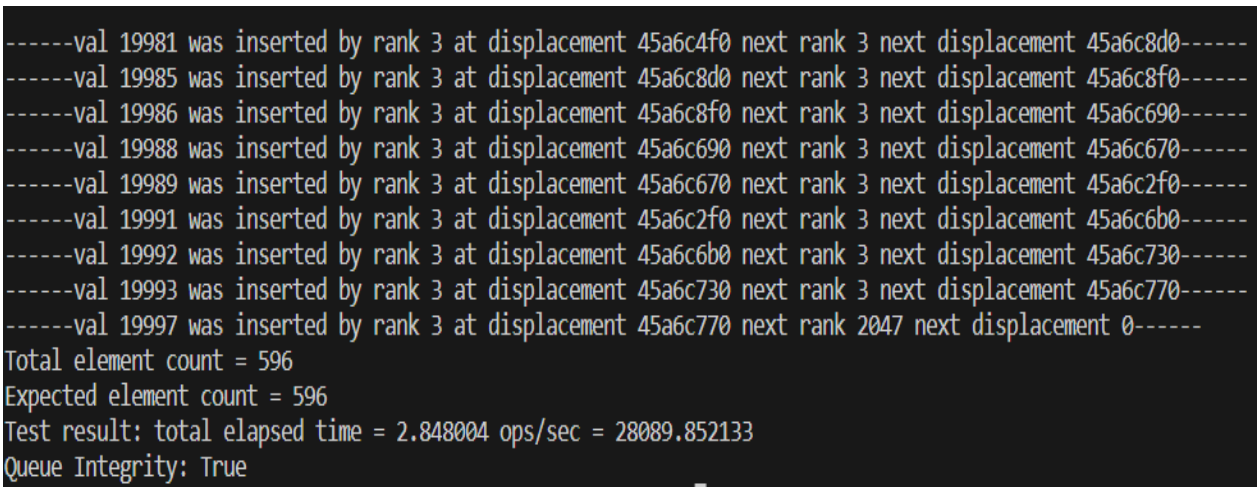

Рисунок 4.3. Пример запуска очереди на одном узле.

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.

5.1. Структура и описание стека Трайбера.

Стек Трайбера – 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:

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:

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.

5.3. Тестирование стека на одном вычислительном узле.

Результат запуска программы приведен на рисунке 5.3. Каждый из процессов производил порядка 20000 операций вставки/удаления, тип операции выбирался случайно.

Каждый процесс вел локальный счетчик успешно добавленных/удаленных элементов, а затем отсылал результаты корневому процессу для проверки целостности очереди.

Для подсчета общего времени использовалась функция MPI_Wtime().

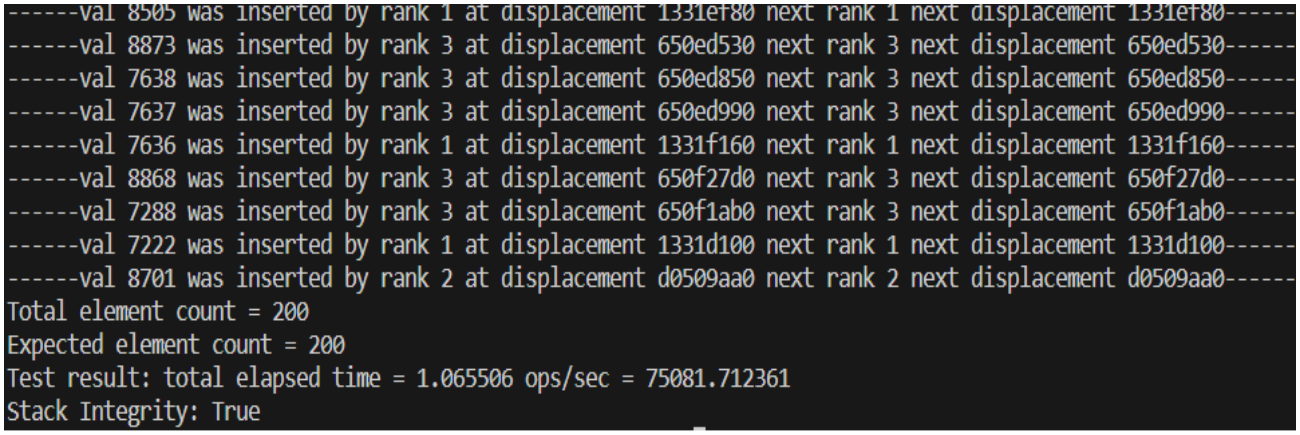

Рисунок 5.3. Пример запуска стека на одном узле.

6. ТЕСТИРОВАНИЕ РЕАЛИЗОВАННЫХ СТРУКТУР ДАННЫХ НА КЛАСТЕРЕ.

Экспериментальное исследование проводилось на вычислительном кластере. В экспериментах использовалось 4 вычислительных узла. 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.

```
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.

Для тестирования очереди и стека был разработан тест, в котором каждый из процессов выполнял по 10000 операций вставки/удаления элементов из структуры данных. При этом тип операции выбирался равновероятно.

Для оценки масштабируемости реализованных структур данных принята метрика «пропускная способность», которая вычисляется следующим образом $b = \frac{n*k}{t}$, где n – количество операций, k –количество процессов, t – обще 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 тестирования на графике:

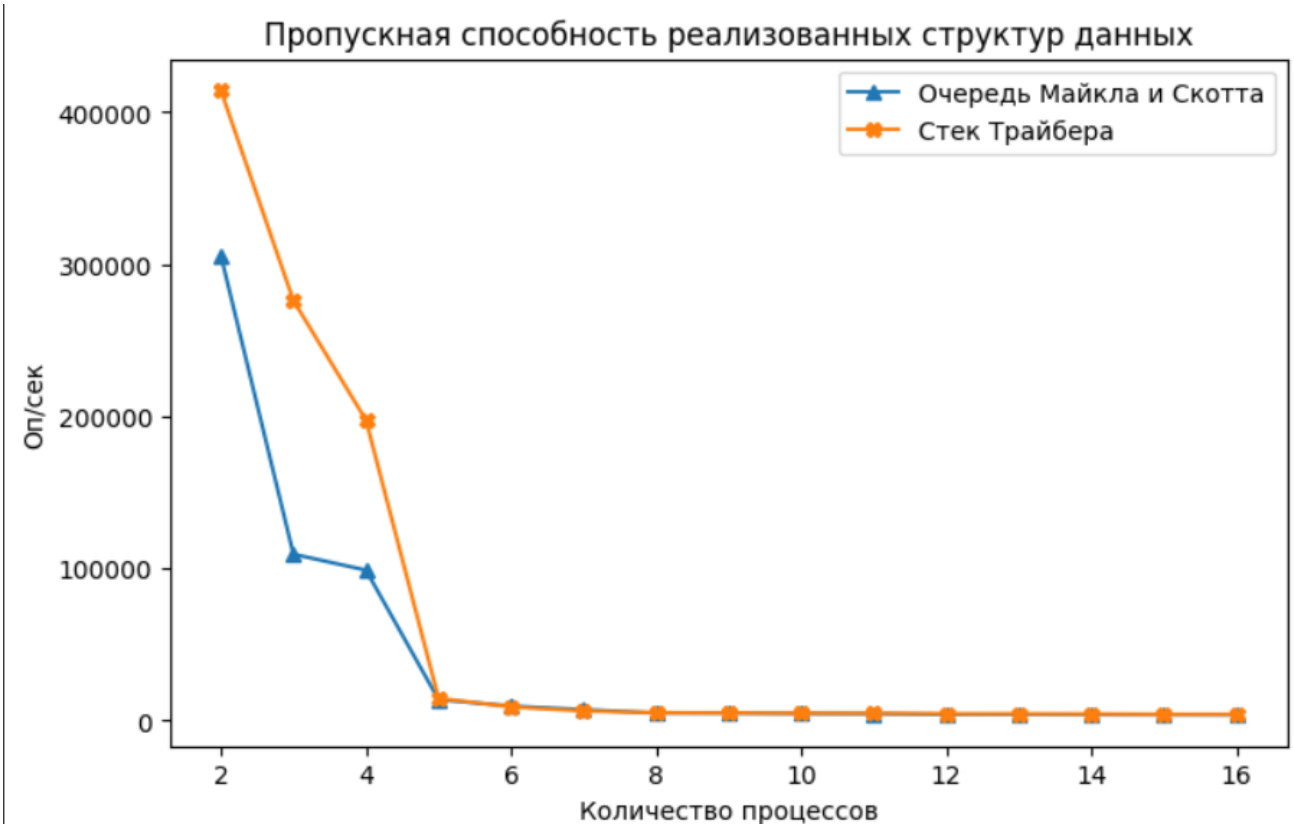

Рисунок 6.3. График пропускной способности очереди и стека.

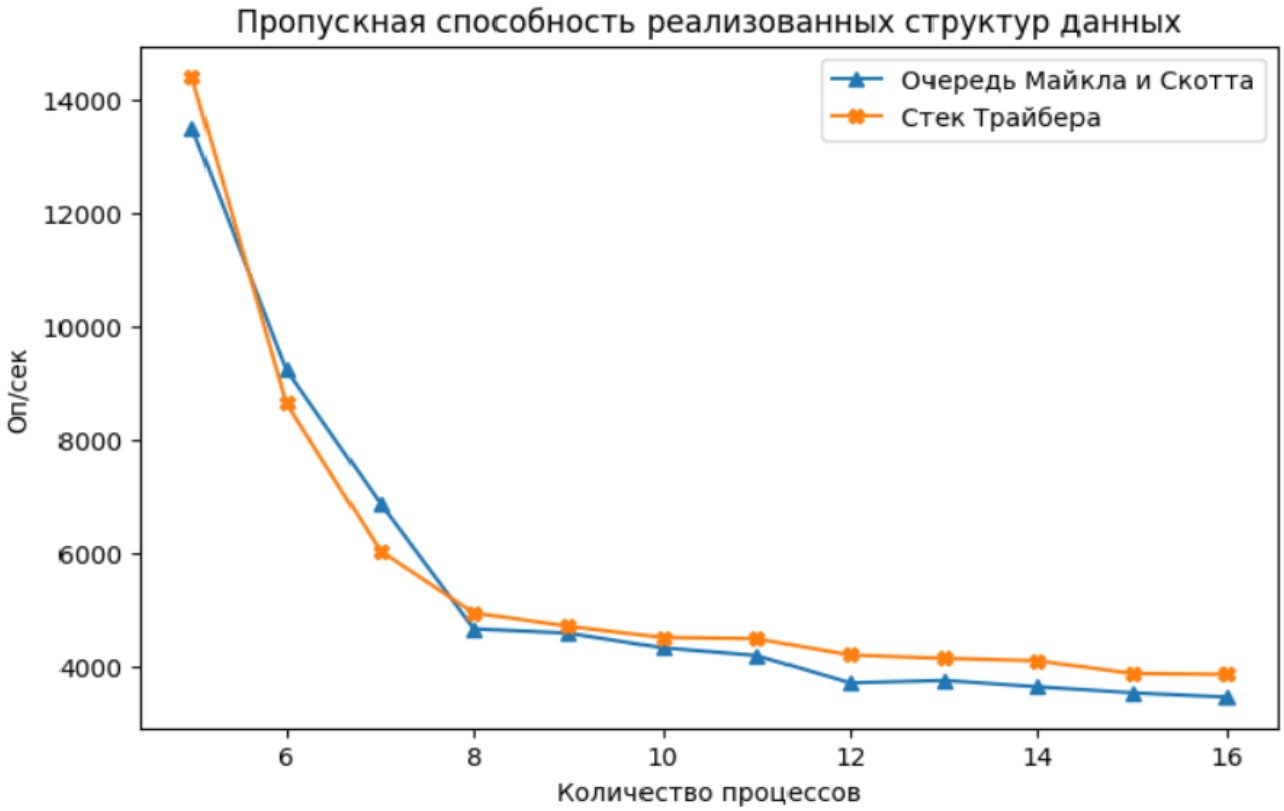

Рисунок 6.4. График пропускной способности очереди и стека (6 - 16 процессов).

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 при передачи данных между процессами по сети. Несмотря на это пропускная способность остается на приемлемом уровне.

Результаты измерения пропускной способности связного списка с блокировками приведен на рисунке 6.5.

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.

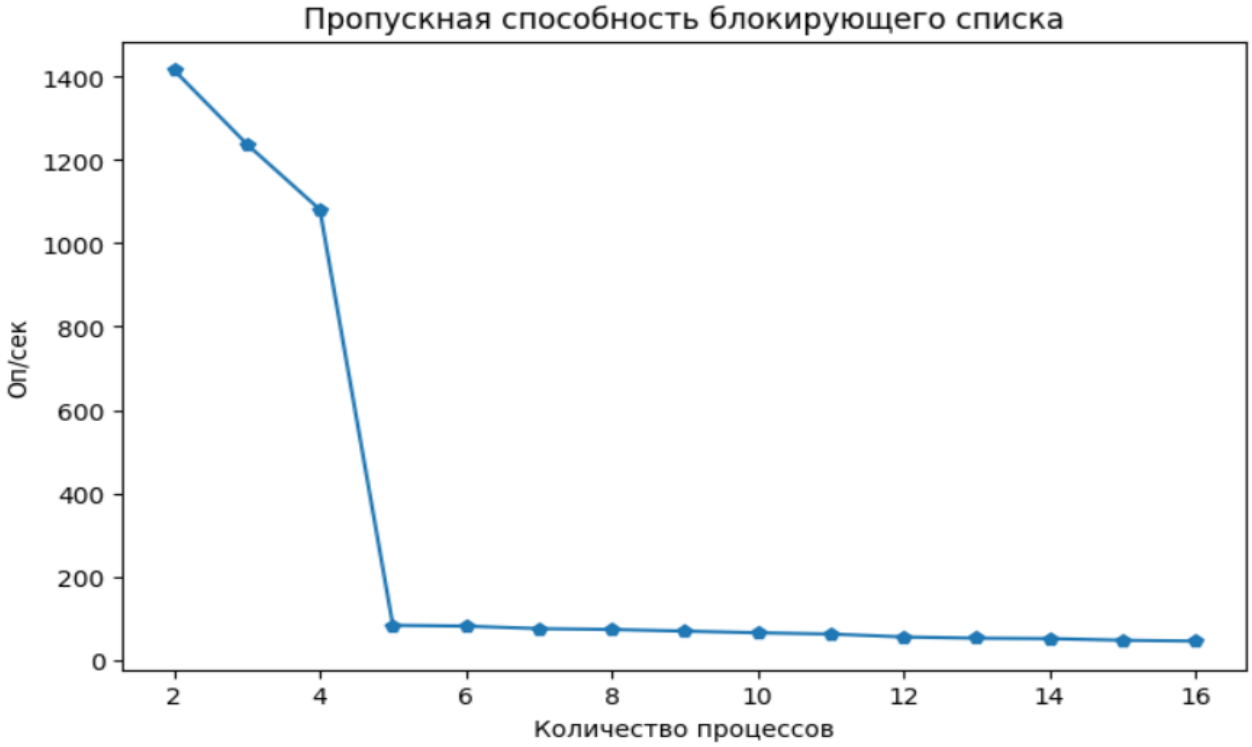

Рисунок 6.5. Пропускная способность списка.

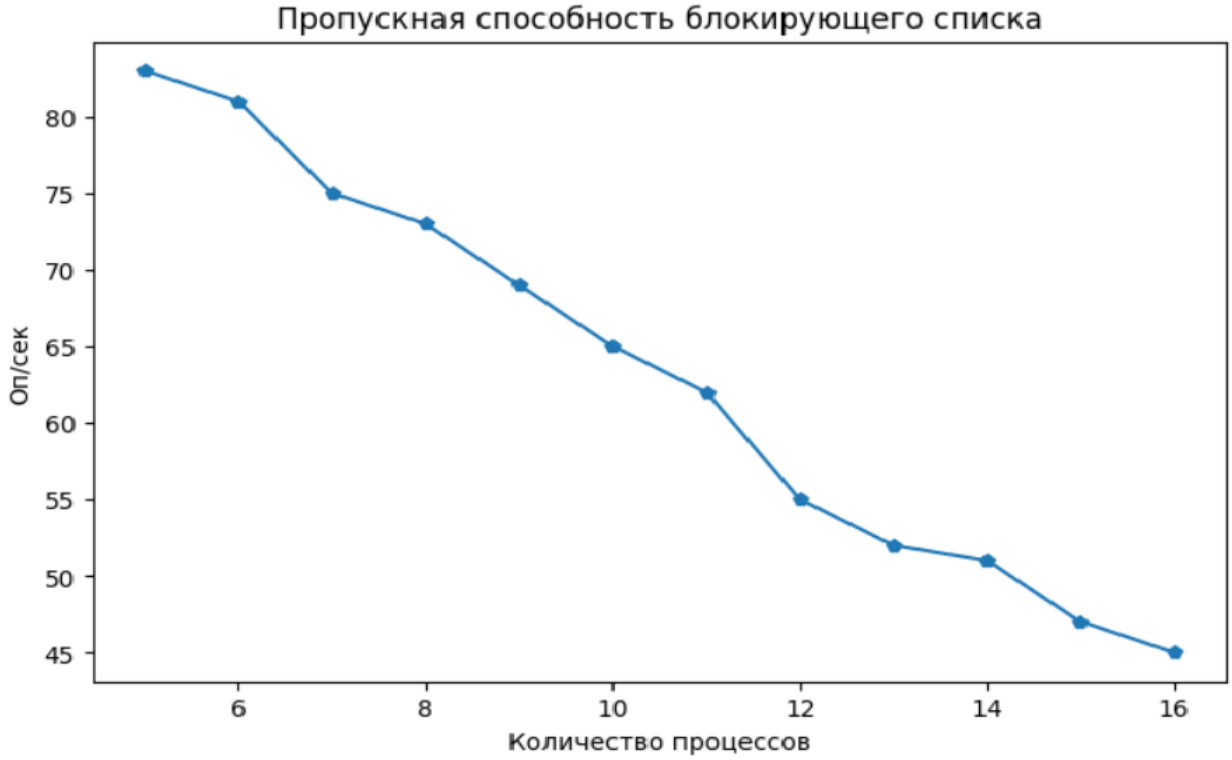

Рисунок 6.6. Пропускная способность списка (6 - 16 процессов).

7. ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ВКР.

В настоящее время суперкомпьютеры применяются в различных областях науки, где необходимо решать сложные вычислительные задачи, например, в физике, астрономии, химии, биологии и климатологии.

При этом, одним из существенных отличий суперкомпьютеров от обычных компьютеров является то, что отдельные узлы суперкомпьютеров являются системами, использующих модель распределенной памяти, в то время как вторые, используют модель общей памяти.

В то же время большая часть исследований в области разделяемых структур данных посвящена проектированию структур, которые могут выполняться в системах, использующих модель общей памяти. Похожих исследований для систем, использующих модель распределенной памяти крайне мало. При этом прямое использование структур, реализованных в системах с SMM, оказывается невозможным в системах с DMM, что делает задачу проектирования разделяемых структур данных для систем с DMM актуальной.

7.1. 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 примем единицу человеко-день.

Таблица 7.1 – План-график выполнения работ

№ работы	Наименование работы	Исполнитель	Длительность работы, человеко-день
1	Консультации с научным руководителем	Руководитель	2
		Студент	2

2	Постановка задачи, выдача технического задания	Руководитель	1
3	Изучение разделяемых структуры данных	Студент	2
4	Изучение модели MPI RMA	Студент	7
5	Реализация связного списка с использованием блокировок	Студент	3
6	Реализация неблокирующей очереди Майкла и Скотта	Студент	6
7	Реализация неблокирующего стека Трайбера	Студент	5
8	Тестирование полученных структур данных, измерение их пропускных способностей	Руководитель	1
		Студент	1
9	Оформление пояснительной записки	Студент	15

Таким образом, на выполнение всех работ было затрачено:

-студентом: 41 ч.дн.

-руководителем: 2 ч.дн.

7.2. Расчет расходов на оплату труда исполнителей.

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

7.2.1 Расчет основной заработной платы исполнителей.

Используя данные о месячных заработных платах руководителя и студента (принимается равной заработной плате инженера) можно рассчитать стоимость оплаты одного человека-дня для каждого исполнителя.

Месячный оклад руководителя составляет 48000 рублей, а студента – 28000 рублей. Стоимость человека-дня вычисляется путем деления месячного оклада исполнителя на количество рабочих дней в месяце (принимается равным 21).

Следовательно, стоимость человека дня составляет:

-для руководителя: $СЧД_{рук} = \frac{48000}{21} = 2285.7$ руб/день;

-для студента $СЧД_{студ} = \frac{28000}{21} = 1333{,}33$ руб/день.

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

-для руководителя: $З_{осн.з.пл} = 2285.7 * 2 = 4571{,}4$ руб.;

-для студента: $З_{осн.з.пл.} = 1333,33 * 41 = 54666{,}53$ руб.;

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.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

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

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

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

3. 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.

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

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

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

ПРИЛОЖЕНИЕ А

Программный код основных функций связного списка.

```
typedef struct {
	int rank;
	MPI_Aint disp;
} nodePtr;

typedef struct {
	int id;
	int val;
	char logicalyDeleted;
	char canBeReclaimed;
	nodePtr next;
} node;

MPI_Aint allocElem(int id, int val, int rank, MPI_Win win) {
	MPI_Aint disp;
	node* allocNode;

	MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, 0, win);
		for(int i = 0; i < allocNodeCount; i++){
			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);

	allocNode->id = id;
	allocNode->val = val;
	allocNode->next = nullPtr;
	allocNode->logicalyDeleted = 0;
	allocNode->canBeReclaimed = 0;

	MPI_Win_attach(win, allocNode, sizeof(node));

	if (allocNodeCount == allocNodeSize) {
		allocNodeSize += 100;
		allocNodesTmp = (node**)realloc(allocNodes,
		allocNodeSize * sizeof(node*));
		if (allocNodesTmp != NULL)
			allocNodes = allocNodesTmp;
		else {
			printf("Error while allocating memory!\n");
			return -1;
		}
	}

	allocNodes[allocNodeCount] = allocNode;
	allocNodeCount++;

	MPI_Get_address(allocNode, &disp);

	return disp;
}

nodePtr search(int id, nodePtr head, MPI_Win win)
```

```
	{
		nodePtr curNodePtr = head, next = {0};
		node curNode = {0};

		while(curNodePtr.rank != -1 && curNodePtr.disp !=
(MPI_Aint)MPI_BOTTOM){
			MPI_Win_lock(MPI_LOCK_EXCLUSIVE, curNodePtr.rank, 0,
			win);
			MPI_Get((void*)&curNode, sizeof(node), MPI_BYTE,
			curNodePtr.rank, curNodePtr.disp,
			sizeof(node), MPI_BYTE, win);
			MPI_Win_flush(curNodePtr.rank, win);
			if(curNode.id == id) {
				MPI_Win_unlock(curNodePtr.rank, win);
				return curNodePtr;
			} else next = curNode.next;
			MPI_Win_unlock(curNodePtr.rank, win);
			curNodePtr = next;
		}
		return nullPtr;
	}
	void insertAfter(int id, int newVal, int key, int rank, nodePtr
head, MPI_Win win)
	{
		node curNode;
		nodePtr newNode, curNodePtr, fetched;
		curNodePtr = search(key, head, win);
		if(curNodePtr.rank == nullPtr.rank && curNodePtr.disp ==
	nullPtr.disp) {
			failedIns++;
			return;
		}
		else {
			newNode.rank = rank;
```

```
                newNode.disp = allocElem(id, newVal, rank, win);
                if(newNode.disp == -1) return;
                        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, curNodePtr.rank,
        0, win);

                MPI_Get((void*)&curNode, sizeof(node), MPI_BYTE,
                curNodePtr.rank, curNodePtr.disp,
                        sizeof(node), MPI_BYTE, win);
                MPI_Win_flush(curNodePtr.rank, win);
                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 {
                        MPI_Win_unlock(curNodePtr.rank, win);
                        failedIns++;
                        return;
                }
                MPI_Win_unlock(curNodePtr.rank, win);
        }
    }
    int traverseAndDelete(nodePtr head, nodePtr nodeToDelete, nodePtr
nextAfterDeleted, MPI_Win win)
    {
        nodePtr curNodePtr = head, next = {0};
```

```
            node curNode = {0};
            while(curNodePtr.rank != -1 && curNodePtr.disp !=
(MPI_Aint)MPI_BOTTOM){
                  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, curNodePtr.rank, 0,
win);
                  MPI_Get((void*)&curNode, sizeof(node), MPI_BYTE,
curNodePtr.rank, curNodePtr.disp,
                        sizeof(node), MPI_BYTE, win);
                  MPI_Win_flush(curNodePtr.rank, win);
                  if(curNode.next.rank == nodeToDelete.rank &&
curNode.next.disp == nodeToDelete.disp) {
                        if(curNode.logicalyDeleted == 0){
                              MPI_Put((void*)&nextAfterDeleted,
sizeof(nodePtr), MPI_BYTE, curNodePtr.rank,
                              curNodePtr.disp + offsetof(node, next),
sizeof(nodePtr), MPI_BYTE, win);
                              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);
                  }
                  curNodePtr = next;
            }
      }
      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;
      curNodePtr = search(key, head, win);
      if(curNodePtr.rank == nullPtr.rank && curNodePtr.disp ==
nullPtr.disp) {
            failedDel++;
            return;
      } else {
            MPI_Win_lock(MPI_LOCK_EXCLUSIVE, curNodePtr.rank, 0,
            win);
            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);
            if(result != 1) {
                  while(testDelete == 0){
                        testDelete = traverseAndDelete(head,
                        curNodePtr, next, win);
                  }
                  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, curNodePtr.rank,
            0, win);
                        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++;
      }}
```

ПРИЛОЖЕНИЕ Б

Программный код основных функций неблокирующей очереди.

```
typedef struct {
	uint64_t rank : 11;
	uint64_t offset : 53;
} nodePtr;

typedef struct {
	int val;
	nodePtr next;
} node;

typedef struct{
	nodePtr dummy;
	nodePtr head;
	nodePtr tail;
} Queue;

uint64_t allocElem(int val, int rank, MPI_Win win) {
	MPI_Aint disp;
	node* allocNode;
	MPI_Alloc_mem(sizeof(node), MPI_INFO_NULL, &allocNode);
	allocNode->val = val;
	allocNode->next = nullPtr;
	MPI_Win_attach(win, allocNode, sizeof(node));
	if (allocNodeCount == allocNodeSize) {
		allocNodeSize += 100;
		allocNodesTmp = (node**)realloc(allocNodes,
	allocNodeSize * sizeof(node*));
		if (allocNodesTmp != NULL)
			allocNodes = allocNodesTmp;
		else {
```

```
                printf("Error while allocating memory!\n");
                return 0;
            }
        }
        allocNodes[allocNodeCount] = allocNode;
        allocNodeCount++;
        MPI_Get_address(allocNode, &disp);
        return disp;
}

nodePtr getTail(Queue q, MPI_Win win)
{
        nodePtr tail = { 0 }, curNodePtr = { 0 };
        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);
        return tail;
}

void enq(int val, int rank, Queue q, MPI_Win win)
{
        nodePtr newNode = { 0 }, result = { 0 }, tmpTail = { 0 },
        tmpTailUpdated = { 0 }, tailNext = { 0 };
        newNode.rank = rank;
        newNode.offset = allocElem(val, rank, win);
        while(1){
            tmpTail = getTail(q, win);

            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);
```

```
                if(result.rank == nullPtr.rank && result.offset
                == nullPtr.offset){
                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;
          } 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);
          }
     }
}
nodePtr getHead(Queue q, MPI_Win win)
{
     nodePtr result = { 0 };
     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);
     return result;
}

nodePtr getNextHead(nodePtr head, MPI_Win win)
{
     nodePtr result = { 0 };
     if(head.rank == nullPtr.rank) return head;
```

```
	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);
	return result;
}

int readVal(nodePtr ptr, MPI_Win win)
{
	int result = 0;
	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);
}

void deq(Queue q, MPI_Win win)
{
	nodePtr tail = { 0 }, head = { 0 }, afterHead = { 0 },
	result = { 0 };
	while(1){
		head = getHead(q, win);
		tail = getTail(q, win);
		afterHead = getNextHead(head, win);
			if(tail.rank == head.rank && tail.offset ==
	head.offset){
			if(afterHead.rank == nullPtr.rank &&
		afterHead.offset == nullPtr.offset) {
				return;
			} else {
				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);
               }
          } else {
               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;
               }
          }
     }
}
```

ПРИЛОЖЕНИЕ В

Программный код основных функций неблокирующего стека.

```
typedef struct {
      uint64_t rank : 11;
      uint64_t offset : 53;
} nodePtr;

typedef struct {
      int val;
      nodePtr next;
} node;

typedef struct{
    nodePtr dummy;
      nodePtr head;
} Stack;

uint64_t allocElem(int val, MPI_Win win) {
      MPI_Aint disp;
      node* allocNode;

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

      allocNode->val = val;
      allocNode->next = nullPtr;

      MPI_Win_attach(win, allocNode, sizeof(node));

      if (allocNodeCount == allocNodeSize) {
            allocNodeSize += 100;
            allocNodesTmp = (node**)realloc(allocNodes,
            allocNodeSize * sizeof(node*));
```

```
            if (allocNodesTmp != NULL)
                  allocNodes = allocNodesTmp;
            else {
                  printf("Error while allocating memory!\n");
                  return 0;
            }
      }
      allocNodes[allocNodeCount] = allocNode;
      allocNodeCount++;
      MPI_Get_address(allocNode, &disp);
      return disp;
}

int readVal(nodePtr ptr, MPI_Win win)
{
      int result = 0;
            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);
}

nodePtr getHead(Stack s, MPI_Win win)
{
    nodePtr result = {0};
             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;
}

void changeNext(nodePtr oldHead, nodePtr newHead, MPI_Win win){
```

```
        nodePtr result = { 0 };
         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);
    }

    void push(int val, int rank, Stack s, MPI_Win win)
    {
        nodePtr curHead = {0}, newHead = {0}, result = {0};
        newHead.rank = rank;
        newHead.offset = allocElem(val, win);
        while(1){
            curHead = getHead(s, win);
            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;
              }
        }
    }

    nodePtr getNextHead(nodePtr head, MPI_Win win)
    {
        nodePtr result = {0};

        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);

    return result;
}

void pop(Stack s, MPI_Win win)
{
    nodePtr curHead = {0}, result = {0}, nextHead = {0};
    while(1){
        curHead = getHead(s, win);
           if(curHead.rank == nullPtr.rank) return;
        nextHead = getNextHead(curHead, win);
            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;
           }
    }
}
```