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

Направление подготовки: 09.04.01 “Информатика и вычислительная техника”
Магистерская программа: “Распределенные интеллектуальные системы и технологии”

Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

К защите допустить:
Заведующий кафедрой

д. т. н., профессор _____ М. С. Куприянов

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Тема: “ Распределенный стек без блокировок в модели удаленного доступа к памяти”

Студент _____ Д.П. Державин

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

Консультант по
оценке и защите результатов
интеллектуальной деятельности
к. э. н., доцент _____ М.Н. Магомедов

Консультант от кафедры
_____ М.Н. Гречухин

Санкт-Петербург
2023 г.

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

Направление: 09.04.01 "Информатика и вычислительная техника"
Магистерская программа: "Распределенные интеллектуальные системы и технологии"
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
М. С. Куприянов
"___" _____ 202__г.

ЗАДАНИЕ
на выпускную квалификационную работу

Студент Д.П. Державин Группа № 7307

1. Тема: Распределенный стек без блокировок в модели удаленного доступа к памяти

Место выполнения ВКР: Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

2. Объект и предмет исследования:
Объектом исследования являются распределённые структуры данных. Предметом исследования является распределенный стек без использования блокировок.

3. Цель:
Реализация распределенного стека без использования блокировок в модели удалённого доступа к памяти.

4. Исходные данные:
Научные статьи и книги на тему параллельной обработки разделяемых структур данных, стандарт межпроцессорного взаимодействия MPI, библиотека MPICH, модель удалённого доступа к памяти, язык C++.

5. Содержание:
Обзор литературы по теме работы. Изучение стандарта MPI и библиотеки MPICH. Проектирование структуры данных. Реализация структуры данных в коде на языке C++. Проведение экспериментов и анализ результатов.

6. Технические требования:
Структура данных должна быть реализована без блокировок на языке программирования C++ с применением стандарта MPI и библиотеки MPICH и корректно обрабатываться на вычислительном кластере.

7. Дополнительные разделы:

Оценка и защита результатов интеллектуальной деятельности.

8. Результаты:

Пояснительная записка, исходный код на языке программирования C++.

Дата выдачи задания «__» _____ 202__ г.	Дата представления ВКР к защите «__» _____ 202__ г.

Руководитель к. т. н., доцент	_____	А.А. Пазников
Студент	_____	Д.П. Державин

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

Направление: 09.04.01 “Информатика и вычислительная техника”
Магистерская программа: “Распределенные интеллектуальные системы и технологии”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
М. С. Куприянов
“___” _____ 202__г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы

Тема: Распределённый стек без использования блокировок модели удалённого доступа к памяти

Студент Д.П. Державин Группа № 7307

№ этапа	Наименование работ	Срок выполнения
1	Обзор литературы по теме работы	06.02-16.02
2	Изучение стандарта MPI и библиотеки MPICH	17.02-01.03
3	Проектирование структуры данных	02.03-27.03
4	Реализация структуры данных в коде на языке C++	28.03.-29.04
5	Проведение экспериментов и анализ результатов	30.04-02.05
6	Оформление пояснительной записки	03.05-12.05
7	Предварительное рассмотрение работы	22.05
8	Представление работы к защите	26.05

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

Студент _____ Д.П. Державин

РЕФЕРАТ

Пояснительная записка: 84 стр., 14 рис., 5 таблиц, прил. 1

Магистерская диссертация посвящена разработке распределённого стека без использования блокировок в модели удалённого доступа к памяти. Объектом исследования являются распределённые структуры данных, предметом исследования – распределённый стек без блокировок.

Результат проделанной работы включает в себя описание алгоритма распределённого стека без блокировок, исходный код проекта на языке программирования C++ с использованием библиотеки MPICH и реализованной в ней модели удалённого доступа к памяти (RMA) стандарта MPI. Работа сопровождается приведением экспериментальных результатов производительности структуры данных. Реализованный стек обладает важными свойствами по сравнению с аналогами: он не подвержен блокировкам и корректно и эффективно исполняется на распределённой вычислительной системе.

Описанный в данной работе алгоритм стека может найти применение в научных расчётах на вычислительных кластерах, а также быть использован в качестве вспомогательной структуры данных, например, пула распределённой памяти, для реализации более сложных структур данных.

ABSTRACT

The master's thesis is devoted to the development of a distributed stack without the use of locks in a remote memory access model. The object of research is distributed data structures, the subject of research is a distributed stack without locks.

The result of the work done includes a description of the distributed stack algorithm without locks, the source code of the project in the C++ programming language using the MPICH library and the MPI remote memory access (RMA) model implemented in it. The work is accompanied by the presentation of experimental data structure performance results. The implemented stack has important properties compared to its analogues: it is not subject to locks and is executed correctly and efficiently on a distributed computing system.

The stack algorithm described in this paper can be used in scientific calculations on computing clusters, and can also be used as an auxiliary data structure, for example, a distributed memory pool, to implement more complex data structures.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ABA – проблема незамеченного обновления данных потоков по причине многочисленных CAS-операций.

Back-off – отступление, общее название для стратегий поведения структуры данных в случае обнаружения высокой конкурентности за общий ресурс.

BCL – Berkeley Core Library.

CAS – сравнение с обменом, compare-and-set, compare-and-swap, одна из операций RMW.

DMA – Direct Memory Access, прямой доступ к памяти

DMM – Distributed Memory Machine, система с распределённой памятью.

EBS – Elimination back-off stack, стек со стратегией отступления в виде массива исключения

Exponential back-off – экспоненциальное отступление, заключается в приостанавливании исполнения текущего потока на промежуток времени, в некоторое количество раз, превышающий предыдущий промежуток времени ожидания.

HCL – Hermes Container Library.

HP – Hazard Pointer, указатель опасности.

Lock-free – без блокировок, уровень неблокирующей синхронизации алгоритма, гарантирующий прогресс хотя бы одного потока.

MIMD – Multiple Instruction stream, Multiple Data stream, многопроцессорные вычислительные системы.

MPI – Message Passing Interface, стандарт для обмена данными с помощью сообщений в распределенных компьютерных сетях.

MPICH – одна из библиотек, в которой реализован стадарт MPI.

NUMA – система с неоднородным доступом к памяти.

Obstruction-free – без препятствий, уровень неблокирующей синхронизации алгоритма, гарантирующий одного потока, если все остальные будут приостановлены.

RDMA – Remote Direct Memory Access, удалённый прямой доступ к памяти с аппаратной поддержкой.

RMA – Remote Access Memory, модель удалённого доступа к памяти.

RMW – read-modify-write, категория атомарных операций, позволяющих читать значение из ячейки памяти и одновременно записывать в неё новое значение.

SMM – Shared Memory Machine, система с общей памятью.

SMR – Safe Memory Reclamation, безопасное освобождение памяти

TS – Treiber Stack, стек Трейбера.

TSS – Time-Stamped Stack, стек с временными метками.

Wait-free – без ожиданий, уровень неблокирующей синхронизации алгоритма, гарантирующий прогресс всех потоков.

ВВЕДЕНИЕ

В 2021 году был введён в эксплуатацию первый в мире суперкомпьютер с эксафлопсной мощностью – Hewlett Packard Enterprise Frontier, также известный как OLCF-5. Его количество вычислительных ядер составило целых 8 730 112! Не меньше поражает цена за такое чрезвычайно мощное оборудование – 600 млн. долларов США. Удивительно, но страны по всему миру не собираются на этом останавливаться и продолжают инвестировать в создание всё новых, и новых суперкомпьютеров. На первый взгляд это может показаться абсурдом. Однако, как указывают многочисленные факты и исследования, суперкомпьютеры решают ряд сложнейших вычислительных задач, связанных с прогнозированием метеорологических явлений, освоением космоса, созданием высокопрочных материалов и испытанием медицинских препаратов.

«Под капотом» суперкомпьютеры заметно отличаются от домашних компьютеров. Системы с общей памятью (SMM), те же самые домашние компьютеры, плохо масштабируются: поддержка совместной работы нескольких процессоров требует сложного дополнительного аппаратного и программного обеспечения. По этой причине отдельные вычислительные узлы суперкомпьютеров, являющихся уже системами с распределённой памятью (DMM), объединяются высокоскоростной коммуникационной сетью.

Распределённая память приводит к созданию распределённых структур данных, которые так же, как и данные (разделяемые данные) на SMM, могут параллельно обрабатываться вычислительными ядрами. Корректное параллельное выполнение многих программ требует синхронизации доступа к изменяемым данным: данные должны оставаться согласованными.

Существует блокирующий подход к построению алгоритмов параллельной обработки данных. Блокирующий подход подразумевает защиту общих ресурсов с помощью блокирующих примитивов синхронизации, например, мьютексов. Такой подход является простым для

понимания, но считается ненадёжным, так как страдает от типичных для него проблем: взаимные блокировки, динамические взаимные блокировки и инверсии приоритетов выполнения задач, решение которых может предложить неблокирующий подход. Неблокирующий подход основан на следующих принципах: 1) структуры данных должны быть открыты для одновременного доступа со стороны нескольких потоков и 2) задержка одного потока не должна приостанавливать прогресс выполнения программы. Последний принцип является причиной, по которой с увеличением числа потоков производительность обработки неблокирующих структур данных теоретически в целом должна быть выше, чем с блокировками.

Проектирование неблокирующих структур данных считается очень трудной задачей. В отличие от SMM исследований для DMM в этой области крайне недостаточно. Одной из таких мало изученных структур данных является стек.

Цель работы – реализовать распределённый стек без использования блокировок в модели удалённого доступа к памяти (RMA). RMA (Remote Memory Access) – реализованная в стандарте MPI модель памяти, позволяющая процессу указать параметры взаимодействия для обеих сторон, участвующих в обмене сообщениями. *Объектом исследования* являются распределённые структуры данных. *Предмет исследования* – распределённый стек без использования блокировок. Для достижения поставленной цели необходимо решить следующие *задачи*:

- провести обзор существующих реализаций распределённого стека;
- спроектировать и реализовать распределённый стек в коде на языке программирования C++;
- провести анализ производительности полученного распределённого стека.

В первом разделе рассматривается конкурентный доступ к общим данным. Во втором разделе рассматривается обмен сообщениями между

процессами в системах с распределённой памятью, стандарт MPI. В третьем разделе проводится обзор реализаций неблокирующего стека для SMM и DMM, решений проблем ABA и SMR и библиотек распределённых структур данных. В четвёртом разделе излагается собственный подход к реализации стека без блокировок с распределённой памятью. В пятом разделе приводятся экспериментальные результаты и анализ разработанной структуры данных. В шестом разделе приведена оценка и защита результатов интеллектуальной деятельности.

1 Доступ к конкурентным данным

Потоки программ могут иметь доступ к одним и тем же данным. Такой доступ называется конкурентным, а данные – конкурентными [1]. Конкурентность – способность системы исполнять несколько вычислительных задач в перекрывающиеся промежутки времени. В отличие от параллелизма одновременное выполнение нескольких задач на аппаратном уровне не является строгим условием для конкурентности.

Если обращение к конкурентным данным происходит из нескольких потоков, то необходимо соблюдать правила, согласно которым все потоки будут видеть только корректные изменения данных, то есть данные должны оставаться согласованными. В случае с нарушением этих правил говорят о состоянии гонки (race condition) [2]. Если речь идёт о неправильной модификации одного объекта, то ситуация также именуется гонкой за данные (data race) [2]. Действия, направленные на сохранение согласованности данных называются синхронизацией, а используемые при этом механизмы управления конкурентностью – примитивами синхронизации.

1.1 Атомарные операции

Атомарная операция [1] – операция, которая либо выполняется полностью, либо небо не выполняется. Атомарность операций может обеспечиваться как на аппаратном уровне с помощью специальных процессорных инструкций, так и на программном. Далее под атомарными операциями будет подразумеваться только первый случай. Не все, но многие примитивы синхронизации строятся на основе атомарных операций. Атомарные операции также являются атомарными примитивами синхронизации.

Среди атомарных операций выделяют операции чтения-модификации-записи (read-modify-write – RMW) [1]. К ним относятся:

- *atomic read* – операция чтения;
- *atomic write* – операция записи;
- *test-and-set* – операция записи нового значения и возврата старого;
- *fetch-and-add* – операция увеличения содержимого ячейки памяти на заданное значение и возврат её предыдущего значения;
- *compare-and-swap (CAS)* – операция сравнения содержимого ячейки памяти с одним из аргументов и записи значения второго аргумента в случае успеха сравнения.

Совместное единовременное принятие решения потоками, то есть синхронизацию, принято называть консенсусом [1]. Число консенсуса – максимальное число потоков, для которого можно решить задачу консенсуса. Среди представленных RMW-операций самой важной является CAS, так как она в отличие от остальных операций позволяет решить задачу консенсуса для бесконечного числа потоков.

1.2 Противодействие состоянию гонки

На сегодняшний день выделяют три основных подхода к противодействию состоянию гонки [2]: блокирующий, неблокирующий и транзакционный подходы. Идея транзакционного подхода заключается в управлении конкурентностью аналогично тому, как это делается по отношению к транзакциям в базах данных. Транзакционный подход выходит за рамки данной выпускной квалификационной работы. В следующих пунктах будут рассмотрены первые два подхода и будет проведено их сравнение.

1.3 Блокирующий подход

Блокирующий подход [2] заключается в защите участков кода, в которых происходит конкурентный доступ к данным, примитивами

блокирующей синхронизации. Самыми известными из них являются мьютекс, семафор и спинлок.

Мьютекс (mutex) [2] – примитив синхронизации, гарантирующий исполнение участка кода только одним потоком. Слово «mutex» является акронимом и происходит от сочетания «**mut**ual **ex**clusion», что буквально переводится как «взаимное исключение», то есть мьютекс взаимно исключает контролируемые им участки кода, которые называются критическими секциями. Для работы с мьютексом обязательно должны быть предоставлены две базовые функции: захвата мьютекса (lock, acquire) и его освобождения (unlock, release).

Семафор (semaphore) [1] является обобщением мьютекса. В отличие от мьютекса семафор позволяет находиться в защищаемом участке кода нескольким потокам, их максимальное количество в произвольный момент времени определяется параметром, который называется ёмкостью семафора. Изнутри семафор устроен, как атомарный счётчик, отслеживающий количество потоков, получивших разрешение на вход.

Спинлок (spinlock) [1] имеет точно такой же пользовательский интерфейс, как и мьютекс. Однако в отличие от мьютекса и семафора в спинлоке по-другому реализован механизм блокирования. Вместо того, чтобы позволить планировщику операционной системы (ОС) вытеснить поток с исполнения на процессоре и перевести его в ожидающий режим, спинлок использует циклическую проверку некоторой переменной на доступность атомарными операциями.

Вытеснение потока планировщиком ОС с последующим его пробуждением – очень затратное по времени действие. Если необходимый интервал блокировки потока небольшой, то выгодно использовать спинлок, в противном случае – мьютекс или семафор, так как длительное циклическое ожидание в спинлоке будет тратить слишком много процессорного времени – было бы разумнее, если бы поток вовсе не исполнялся, то есть был вытеснен

планировщиком ОС. Также на вычислительных системах, состоящих из одного вычислительного ядра, в операционной системе без вытесняющей многозадачности решения на основе спинлоков не работают по причине отсутствия потока, который мог бы модифицировать состояние атомарной переменной, за изменением которой в цикле ожидания спинлока наблюдает другой поток.

С целью повышения эффективности блокировки были разработаны несколько других примитивов блокирующей синхронизации: блокировка чтения-записи и условная переменная.

Блокировка чтения-записи (readers–writer lock) [1] – это примитив синхронизации, который предоставляет доступ к некоторым конкурентным данным либо на чтение несколькими потоками одновременно, либо на их эксклюзивное изменение, тем самым отделяя блокировку чтения от блокировки записи.

Условная переменная (condition variable) [2] – это примитив синхронизации, который гарантирует, что один или несколько потоков будут заблокированы до тех пор, пока не поступит сигнал о выполнении некоторого условия от другого потока или пока не истечет максимальное время ожидания. Условная переменная всегда работает в паре с мьютексом.

Блокирующий подход в целом является очень простым для понимания и применения. Однако с ним связано много проблемных ситуаций, одними из которых являются взаимная блокировка, динамическая взаимная блокировка, инверсия приоритетов, сопровождение, потерянное пробуждение и ложное пробуждение.

Взаимная блокировка (deadlock) [2] – это проблемная ситуация, которая возникает, когда для выполнения какой-либо операции потокам необходимо захватить несколько мьютексов, но сложилось так, что потоки захватили не все мьютексы, а только часть, и в результате ни один из потоков не может

продолжить исполнение, так как вынужден ожидать, пока другой поток не освободит нужный ему мьютекс.

Динамическая взаимная блокировка (livelock) [1] – это проблемная ситуация, при которой несколько потоков выполняют бесполезную работу, попадая в цикл в момент попытки захватить ресурсы, и при этом состояния потоков постоянно меняется в зависимости друг от друга.

Инверсия приоритетов [1] возникает, когда поток с более низким приоритетом вытесняется, но все еще удерживает блокировку, необходимую потоку с более высоким приоритетом.

Сопровождением (convoing) [1] называют проблемную ситуацию, которая происходит, когда поток, удерживающий блокировку, вытесняется с процессора из-за окончания выделенного времени, ошибок чтения страниц из памяти или других видов прерываний.

Ситуации потерянного и ложного пробуждения связаны с использованием условных переменных.

Потерянное пробуждение (lost wake-up) [1] возникает, когда один или несколько потоков находятся в ожидании несмотря на то, что значение условной переменной стало логически верным. Одной из причиной такого поведения может стать изменение значения условной переменной без удерживания соответствующего мьютекса.

Ложным пробуждением (spurious wake-up) [2] называют ситуацию, когда ожидающий поток, захвативший мьютекс, был пробужден не по причине сигнала от условной переменной. Одним из возможных решений этой проблемы может стать повторная проверка условия после пробуждения.

1.4 Неблокирующий подход

Неблокирующий подход [1] опирается на свойство линеаризуемости [3].

Свойство линеаризуемости заключается в следующем: результат работы функции должен вступать в силу мгновенно в момент между началом и

концом её исполнения. Этот результат должен быть эквивалентен одному из правильных последовательных шагов алгоритма, как если бы он выполнялся не параллельно. Момент наступления результата называется точкой линеаризации. Свойство линеаризуемости также очень интересно тем, что композиция двух линеаризуемых функций тоже является линеаризуемой функцией. Это позволяет проектировать линеаризуемые параллельные алгоритмы.

В неблокирующем подходе свойство линеаризуемости поддерживается с помощью атомарных операций, важнейшей из которых является CAS, так как именно она позволяет синхронизировать бесконечное количество потоков.

Синхронизация в неблокирующем подходе является неблокирующей. Выделяют три уровня неблокирующей синхронизации: свободный от препятствий, свободный от блокировок и свободный от ожиданий. Уровни перечислены по порядку от уровня с самой слабой гарантией прогресса алгоритма до уровня с самой сильной гарантией. Уровень с более сильной гарантией также обеспечивает прогресс уровней, слабее его. Например, свободный от блокировок алгоритм является свободным от препятствий, но не наоборот.

Уровень *свободный от препятствий* (obstruction-free) [1] означает, что каждый поток алгоритма, запущенный в любой момент времени, завершится за конечное число шагов, если он не встретит препятствий со стороны других потоков. То есть системный прогресс алгоритма гарантируется, если поток исполняется в изоляции: одновременно с ним другие потоки не работают.

Уровень *свободный от блокировок* (lock-free) [1] означает, что с каждой итерацией алгоритма хотя бы один поток совершает прогресс независимо от успеха остальных потоков, и если шаги алгоритма будут вызываться бесконечно часто, то алгоритм завершится за конечное число шагов.

Уровень *свободный от ожиданий* (wait-free) [1] означает, что каждый поток завершит шаги алгоритма за конечное число шагов, не зависящее от других потоков.

Важно подчеркнуть, из того, что алгоритм неблокирующий ещё не следует, что он является свободным от блокировок. Это лишь означает, что в алгоритме не применяются примитивы блокирующей синхронизации. Свободный от блокировок алгоритм – алгоритм, который не приводит к ситуации блокировки. Так, Obstruction-free алгоритмы не являются свободными от блокировок и могут страдать от проблемных ситуаций livelock и deadlock. Однако на практике это случается редко, и в большинстве случаев с этим справляются с помощью стратегии отступления (back-off) [1]. Идея back-off заключается в том, что, когда поток обнаруживает конкурентность за ресурс с другими потоками, он временно приостанавливает своё выполнение, чтобы дать им возможность беспрепятственно завершить свою операцию.

Lock-free алгоритмы по определению не приводят к проблемным ситуациям блокировки. Тем не менее, частые циклы операций сравнения с обменом (CAS) могут приводить к ситуации *голодания потоков* (starvation) или как её ещё называют *застревание потоков*. Данная ситуация характеризуется тем, что одному потоку постоянно удаётся успешно выполнить некоторую операцию, в то время как другой поток вынужден из-за него повторять одни и те же шаги.

Неблокирующим алгоритмам свойственны проблема ABA и проблема безопасного освобождения памяти, которые обычно возникают совместно.

Проблема ABA [4] возникает, когда из ячейки памяти читается два раза одинаковое значение, и то, что значение никак не изменилось, интерпретируется как то, что состояние ячейки никак не менялось между эти чтениями несмотря на то, что другой поток в этот промежуток времени успел модифицировать значение, выполнить шаг алгоритма и восстановить старое значение. Это проблема связана с использованием многочисленных сравнений

с обменом. Ниже приведён пример последовательности событий, ведущих к проблеме ABA:

1. поток *P* читает значение *a* из общей памяти;
2. поток *P* собирается изменить значение из общей памяти, но вытесняется, позволяя исполняться потоку *Q*;
3. поток *Q* меняет значение с *a* на *b* и обратно на *a* и вытесняется;
4. поток *P* возобновляется и видит, что значение не изменилось.

Результат: поток *P* не заметил, что поток *Q* дважды поменял значение и что скорее поток Q, возможно, сделал какой-нибудь шаг алгоритма, от которого зависит поведение потока *P*.

Проблема безопасного освобождения памяти (Safe Memory Reclamation – SMR) [4] заключается в том, что к некоторой области памяти, например, отделённому от структуры данных узлу, во время восстановления этой области памяти могут обращаться параллельные потоки, обращавшиеся к узлу до отделения. Когда узел полностью отсоединился, может случиться так, что какой-либо поток получит доступ к его освобожденной памяти, что может ошибкам сегментации и логическим нарушениям шагов алгоритма. Проблема характерна для языков программирования без автоматической сборки мусора (garbage collection), например, C++. Причиной этой проблемы чаще всего является проблема ABA.

1.5 Сравнение блокирующего и неблокирующего подходов

В предыдущих двух пунктах были рассмотрены принципы, лежащие в основе блокирующего и неблокирующего подходов, а также сопутствующие им проблемы. В этом пункте будут окончательно выделены сильные и слабые стороны двух подходов и даны рекомендации, в каких случаях стоит применять каждый из них.

Достоинствами блокирующего подхода являются:

- простота построения алгоритмов.

Недостатками блокирующего подхода являются:

- низкая отказоустойчивость по причине блокировок;
- низкая масштабируемость по процессорам, так как в алгоритмах с блокировками любой поток может быть приостановлен в произвольный момент времени.

Достоинствами неблокирующего подхода являются:

- *высокая отказоустойчивость*, так как блокировки либо полностью отсутствуют (lock-free), либо возникают очень редко и с ними вполне успешно справляются (obstruction-free);
- *высокая масштабируемость* по процессорам, так как в случае lock-free хотя бы один поток совершает прогресс на каждом шаге алгоритма.

Недостатками неблокирующего подхода являются:

- сложность построения алгоритмов;
- подверженность проблеме ABA;
- подверженность проблеме SMR;
- большое количество атомарных операций, которые выполняются гораздо медленнее, чем неатомарные, что оказывает существенное влияние на производительность.

Исходя из приведённых выше достоинств и недостатков двух подходов можно прийти к следующим выводам:

1. если разрабатываемая система должна быть надёжной и масштабируемой, то выбор стоит сделать в пользу неблокирующего подхода;
2. если важна высокая скорость и простота разработки системы, то стоит прибегнуть к неблокирующему подходу, так как из многих последовательных алгоритмов несложно сделать параллельные без значительного изменения шагов исходного алгоритма путём

взаимного исключения критических участков кода с помощью мьютексов;

3. чем чаще используются атомарные операции в алгоритме с неблокирующим подходом, тем ниже его производительность.

А что показывает практика? Многие исследователи ссылаются на данные, приведённые в статье [5]. Авторы сравнили блокирующие и неблокирующие реализации структур данных: стек, очередь, куча, и алгоритмов: Quicksort и решение задачи коммивояжёра (Travelling Salesman Problem – TSP), и пришли к выводу, что неблокирующие аналоги алгоритмов являются более производительными и масштабируемыми, чем неблокирующие. Также существует общее наблюдение, что с каждой новой версией ядра Linux разработчики всё чаще отказываются от семафоров в пользу неблокирующих алгоритмов.

2 Обмен сообщениями в системе с распределённой памятью

Согласно классификации Флинна к категории MIMD относятся многопроцессорные вычислительные системы. Эндрю Таненбаум в книге «Архитектура компьютера» [6] предлагает разбить MIMD на две большие группы (рисунок 2.1): мультипроцессоры и мультикомпьютеры.

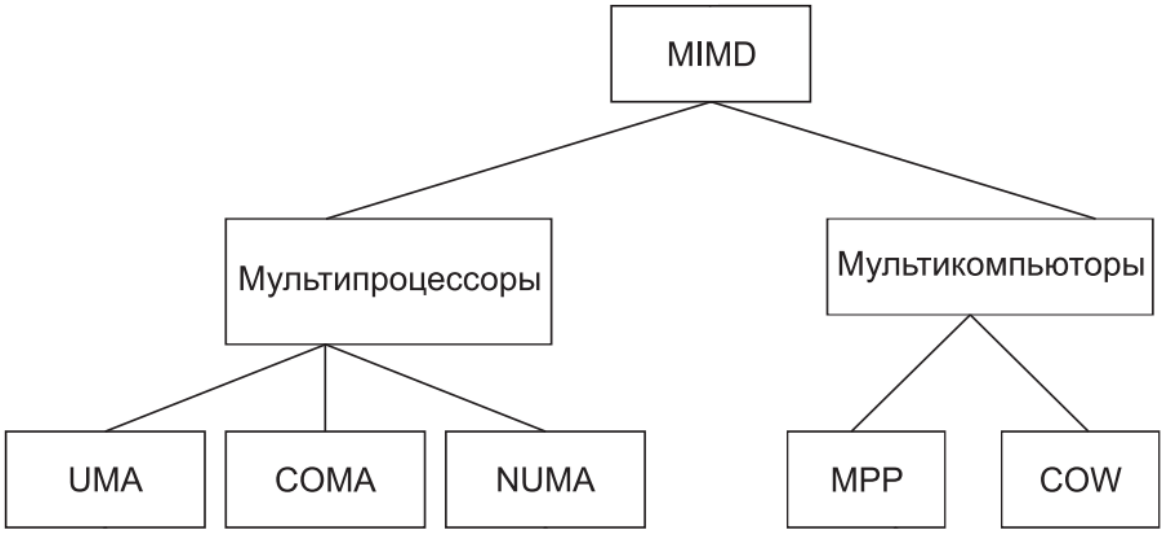

Рисунок 2.1 – Категория вычислительный машин MIMD

Мультипроцессорами называются *системы с общей памятью* (Shared Memory Machines – SMM). SMM могут масштабироваться по процессорам, однако чем больше становится процессоров, тем сильнее они начинают конкурировать за ресурсы вычислительной системы, и в первую очередь за оперативную память (RAM). Именно этот фактор главным образом сдерживает наращивание вычислительных мощностей привычным методом – подключением очередного процессора к шине с однородным доступом к RAM. Переход от *систем с однородным доступом к памяти* (Unified Memory Access – UMA) к *системам с неоднородным доступом* (Non Unified Memory Access) и *системам с доступом только кэш-памяти* (Cache Only Memory Access – COMA) требует сложного аппаратного и программного обеспечения и позволяет вести речь только о сотнях процессоров.

Мультикомпьютерами называются системы с распределённой памятью (Distributed Memory Machines – DMM). К *кластерным рабочим станциям* (Cluster Of Workstations – COW) относятся персональные компьютеры, связанные обыкновенной коммуникационной сетью. Наибольший интерес с точки зрения производительности вычислений представляют *процессоры с массовым параллелизмом* (Massively Parallel Processor – MPP), которыми и являются *суперкомпьютеры*, обладающие тысячами, а то и миллионами вычислительных ядер. MPP отличаются от COW главным образом дорогостоящей высокоскоростной коммуникационной сетью.

2.1 Аппаратный доступ к памяти процесса по коммуникационной сети

Аппаратные технологии доступа к памяти процесса некоторой программы по коммуникационной сети делятся на две большие группы: с двух- (two-sided) и односторонней (one-sided) коммуникацией [7-9]. Двухстороннюю коммуникацию ещё называют Message Passing (MP) и Send/Recv, а одностороннюю – удалённый прямой доступ к памяти (Remote Direct Memory Access – RDMA). В первом случае процесс-отправитель должен явно указать вызову Send, какие данные передавать, а процесс-получатель в соответствующем вызове Recv – куда эти данные сохранить. Во втором случае все действия по предоставлению необходимой информации о передаче данных перекладываются на отправителя.

RDMA лучше всего поддерживается высокоскоростными коммуникационными сетями, такими как InfiniBand, PERCS, Gemini и Aries. В большинстве случаев RDMA оказывается более эффективной технологией, чем MP, за счёт реализации подхода с нулевым копированием (zero-copy). Данный подход позволяет сетевому адаптеру напрямую обращаться к памяти удалённого процесса некоторой программы без участия самого процесса и системных вызовов операционной системы, что позволяет значительно

сэкономить такты вычислительного узла удалённого процесса. Также технология RDMA доступна поверх обыкновенной коммуникационной сети Ethernet по протоколам RoCE и iWarp.

2.2 Стандарт MPI

Написание программ для мультикомпьютеров требует специального программного обеспечения, примерами которого являются UPC, Chapel и MPI. MPI является самым известным среди них.

MPI (Message Passing Interface) [10] – это стандарт, описывающий обмен сообщениями между процессами параллельной программы с распределённой памятью. Первая версия стандарта (MPI-1) вышла в 1994 году. На данный момент последней версией стандарта является MPI-4, вышедшая в 2021 году, и ведётся работа над пятой версией. MPI опирается на аппаратную поддержку технологий доступа к памяти процесса по коммуникационной сети, описанных в предыдущем пункте, и функционирует на всех вычислительных машинах категории MIMD. Доступ к MPI может осуществляться из языков программирования Fortran, C, C++ и Java. Существует множество библиотек, реализующих стандарт MPI, например, OpenMPI [11], Intel MPI [12] и MPICH [13].

MPICH – одна из самых первых библиотек MPI. Она является свободной, портируемой и быстро развивается. В данной выпускной квалификационной работе используется библиотека MPICH с версией стандарта MPI-4.1.

Стандарт MPI-4 [14] описывает:

- двухстороннюю коммуникацию;
- частичную коммуникацию;
- типы данных;
- коллективные операции;
- группы процессов;

- контексты коммуникаций;
- топологии процессов;
- объект Info;
- инициализацию, создание и управление процессами;
- одностороннюю коммуникацию;
- инструментальную поддержку;
- управление окружением и запросами;
- параллельный файловый ввод-вывод;
- функции обращения из языков программирования Fortran и C.

Центральными понятиями в MPI являются группа и коммуникатор.

Группой в MPI называется упорядоченное множество идентификаторов процессов. Каждый процесс в группе ассоциирован с некоторым целым числом rank (ранг).

Коммуникатором в MPI называется объект, который соединяет группы процессов. Коммуникатор выдаёт индивидуальный идентификатор каждому процессу.

2.2.1 Виды коммуникаций

В MPI-4 существует 4 вида коммуникаций [14]: двухсторонние, коллективные, частичные и односторонние.

Двухсторонние коммуникации (рисунок 2.2) удобны в применении и чаще всего используются в простых случаях. Они выполняются между двумя процессами и требуют вызова с обеих сторон таких функций, как MPI_Send и MPI_Recv. MPI_Send – функция отправки сообщения, а MPI_Recv – приёма. Процессы блокируются в функциях MPI_Send и MPI_Recv до тех пор, пока вызов соответствующей функции не завершится на противоположной стороне обмена сообщениями.

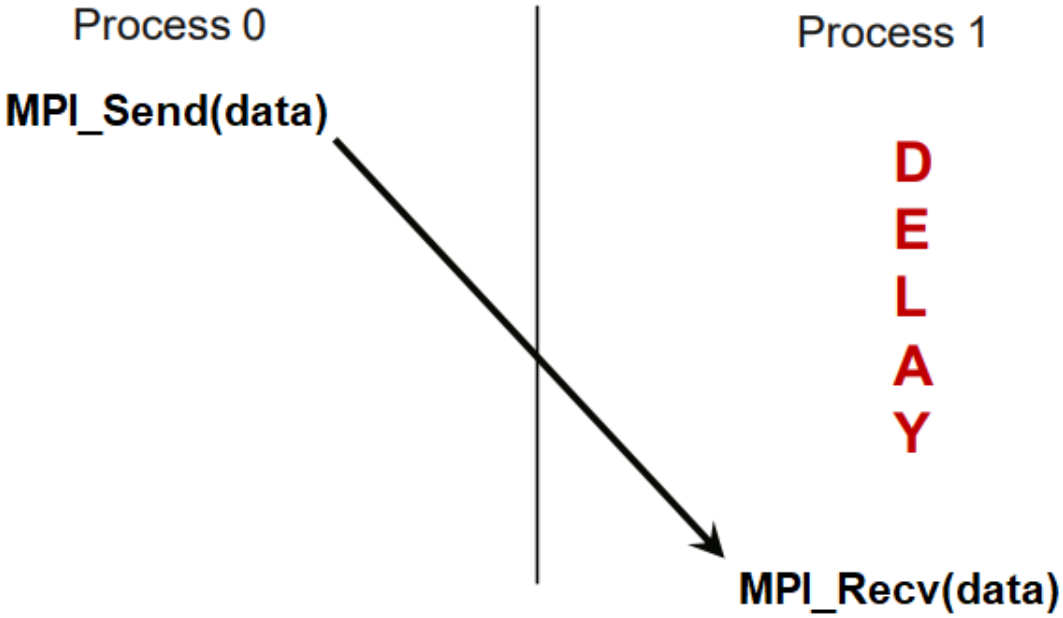

Рисунок 2.2 –Пример двухсторонней коммуникации

Коллективные коммуникации отличаются от двухсторонних тем, что в них должны участвовать все процессы, связанные с коммуникатором. В противном случае процессы зависают или аварийно завершаются. Коллективные коммуникации включают в себя:

- рассылку данных от одного процесса всем остальным (MPI_Bcast);
- сбор данных со всех процессов на одном или нескольких процессах (MPI_Gather, MPI_Allgather);
- разбиение массива данных и рассылка его фрагментов всем процессам (MPI_Scatter);
- вычислительные операции редукции (MPI_Reduce, MPI_Scan);
- барьерную синхронизацию (MPI_Barrier).

В качестве примера рассмотрим функцию MPI_Reduce (рисунок 2.3). В данном примере функция MPI_Reduce вызывается с главным процессом ранга 1 и использует операцию MPI_SUM, чтобы сложить числа со всех процессов. Результат сложения помещается на процесс с рангом 1.

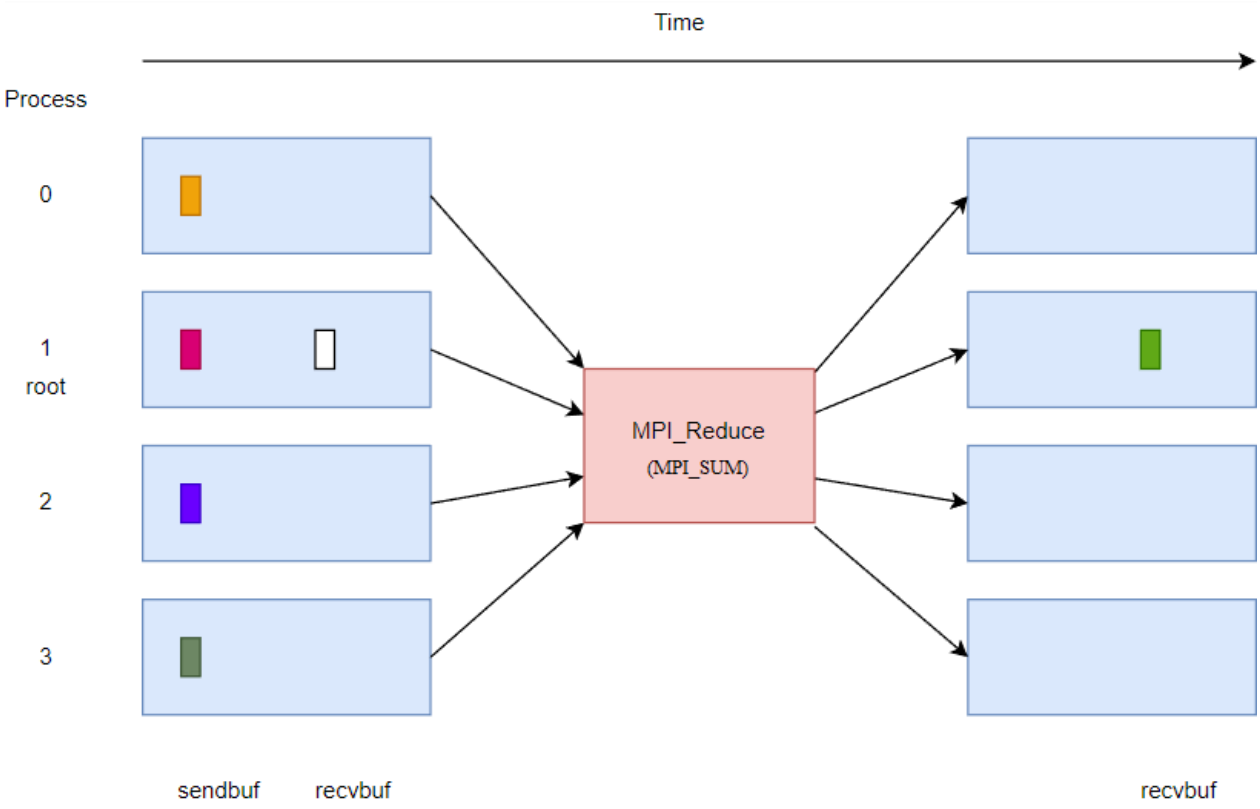

Рисунок 2.3 – Пример коллективной коммуникации с MPI_Reduce

Частичные коммуникации появились в MPI начиная с версии MPI-4. Эта новая модель коммуникаций позволяет вносить данные из нескольких потоков с меньшими накладными расходами, чем двухсторонняя коммуникация.

Односторонние коммуникации (рисунок 2.4) отличаются от двухсторонних тем, что параметры обмена сообщениями между двумя процессами указываются только одним из них. Таких обменов может быть много, но все они должны совершаться в пределах промежутков выполнения, называемых эпохами (epoches). Односторонние коммуникации опираются на аппаратную технологию RDMA и чаще всего оказываются более производительными, чем двухсторонние коммуникации. Более подробно односторонние коммуникации будут рассмотрены в следующих пунктах.

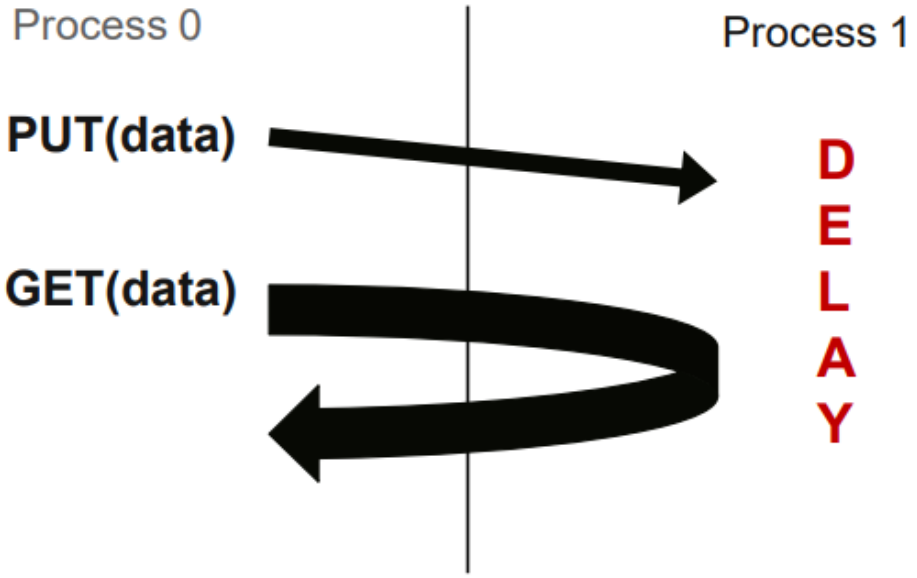

Рисунок 2.4 – Общая схема односторонних коммуникаций

2.2.2 Модель памяти RMA

Удаленный доступ к памяти (Remote Memory Access – RMA) [14] – это дополнение к механизму обмена сообщениями MPI, позволяющее задавать все коммуникационные параметры отправителя и получателя в одном процессе. Этот вид коммуникации облегчает написание программ с динамически меняющимися шаблонами доступа к данным, когда распределение данных либо фиксировано, либо меняется не слишком часто. В таком случае каждый процесс может указать, какие данные должны быть доступны или обновлены другими процессами. Таким образом, параметры передачи данных указываются только одной стороной обмена сообщениями. При обычной двухсторонней коммуникации отправитель и получатель должны явно вызывать функции отправки и получения данных. Чтобы выполнить эту операцию, программе необходимо распределить параметры передачи данных между процессами. Такое распределение данных может потребовать участия всех процессов в сложных совместных вычислениях, отнимающих много времени, или опросах и периодической обработки запросов на обмен сообщениями. RMA позволяет избежать этих избыточных операций.

Двухсторонняя коммуникация объединяет в себе передачу данных и их синхронизацию между отправителем и получателем. Идея подхода RMA заключается в разделении этих процессов.

RMA позволяет использовать механизмы быстрой или асинхронной связи, предоставляемой различными платформами: когерентная или некогерентная общая память, аппаратно поддерживаемые операции, механизмы DMA.

MPI поддерживает две принципиально разные модели памяти: *разделённую* (separate) и *объединенную* (unified).

Разделённая модель памяти (рисунок 2.5) не располагает информацией о согласованности памяти и отличается высокой переносимостью. Эта модель похожа на слабо когерентную систему памяти и требует от пользователя правильной последовательности обращений к памяти с помощью синхронных вызовов.

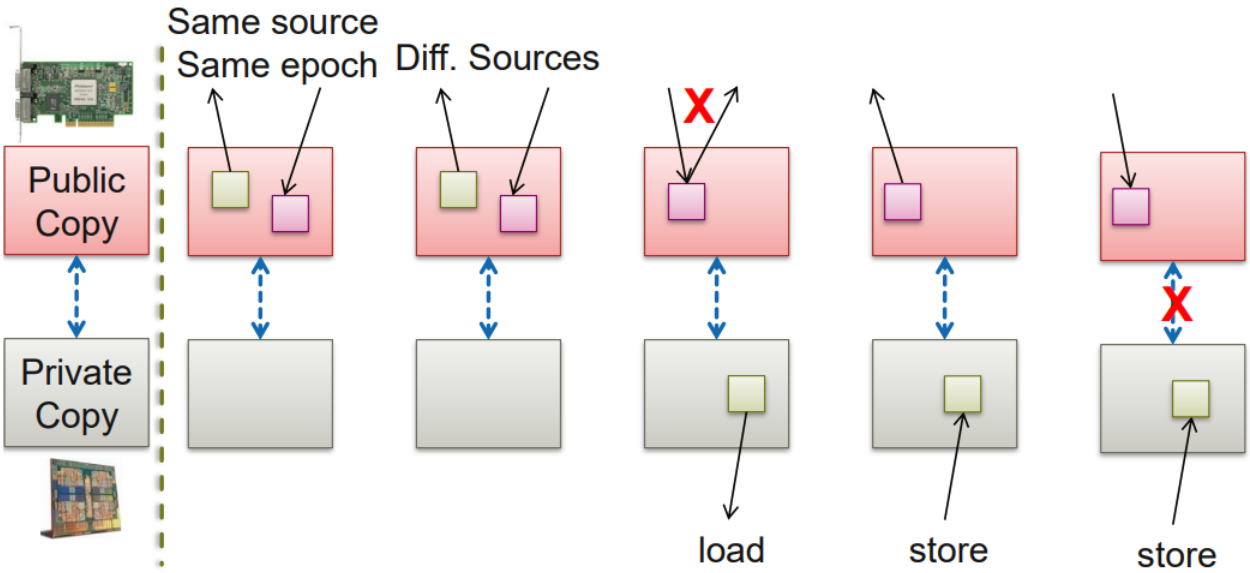

Рисунок 2.5 – Схема разделённой памяти

Объединённая модель памяти (рисунок 2.6) может использовать преимущества аппаратного обеспечения когерентности кэша и аппаратно ускоренных односторонних операций, обычно используемых в высокопроизводительных системах.

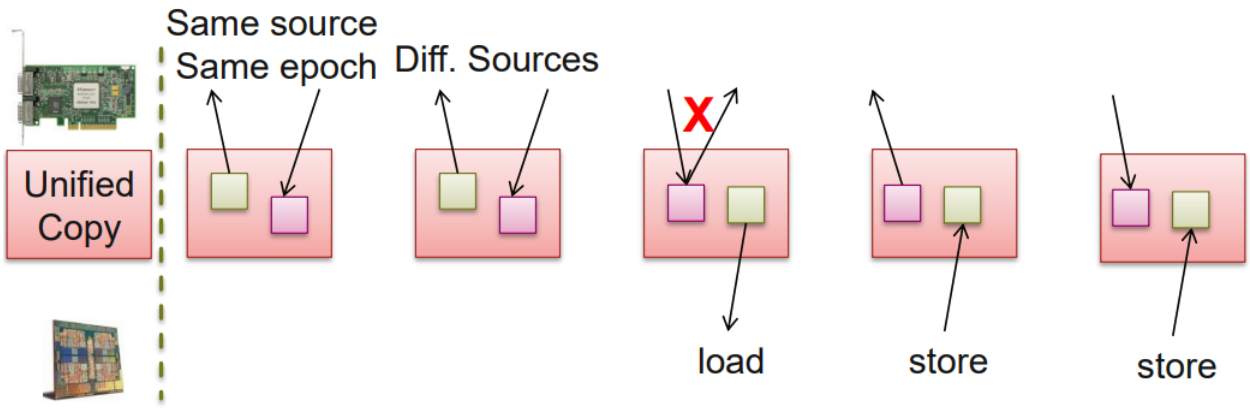

Рисунок 2.6 – Схема объединённой памяти

Модель памяти RMA опирается на концепцию публичных и частных копий окон [14]. Окно – это область памяти процессов, открытая для удалённого доступа. Некоторые вычислительные системы имеют общедоступную область памяти: память на системах с общей памятью и основная память на системах с распределенной памятью, доступная всем процессам. Также на многих вычислительных машинах копии данных из основной памяти сохраняются в локальные частные буферы каждого процесса с целью ускорения доступа. Эти буферы могут быть как когерентными, так и некогерентными. В первом случае во всех частных копиях последовательно отражаются изменения в основной памяти. Во втором случае должны быть синхронизированы и явно обновлены во всех частных копиях конкурирующие запросы к основной памяти. В когерентной системе обновления могут быть сделаны непосредственно в удаленной памяти без участия удаленного процесса. Однако для отражения изменения открытого окна в частной памяти в некогерентной системе необходимо вызывать функции RMA. Открытое и закрытое окна в когерентной памяти логически совпадают, а в некогерентных системах они остаются логически разделенными. Если публичное и частное окна логически совпадают, то модель памяти является объединённой, в противном случае – разделённой.

2.2.3 Функции обмена сообщениями RMA

Среди функций одностороннего обмена MPI RMA можно выделить основные [14]:

- MPI_Put;
- MPI_Get;
- MPI_Compare_and_swap;
- MPI_Fetch_and_op;
- MPI_Accumulate;
- MPI_Get_accumulate.

Последние 4 функции являются атомарными и неблокирующими.

Функция *MPI_Compare_and_swap* является реализацией атомарной RMW-операции CAS, а *MPI_Fetch_and_op* – atomic write, atomic read и fetch-and-add. Эти функции могут иметь аппаратную поддержку, и поэтому MPI_Fetch_and_op может оказаться быстрее, например, той же самой функции MPI_Get_accumulate, но у них есть ограничение на размер передаваемых данных – не больше 64 бит.

Функция *MPI_Put* (рисунок 2.7) очень похожа на функцию MPI_Send. Она отправляет данные из памяти процесса, инициировавшего обмен сообщениями, в память удалённого процесса.

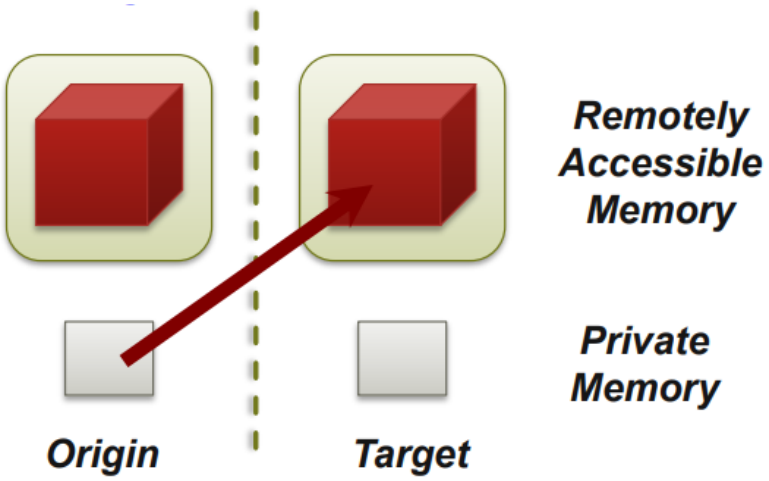

Рисунок 2.7 – Схема работы функции MPI_Put

Функция *MPI_Get* (рисунок 2.8) похожа на функцию MPI_Recv. Она получает данные из памяти удалённого процесса и сохраняет их в память процесса, инициировавшего обмен сообщениями.

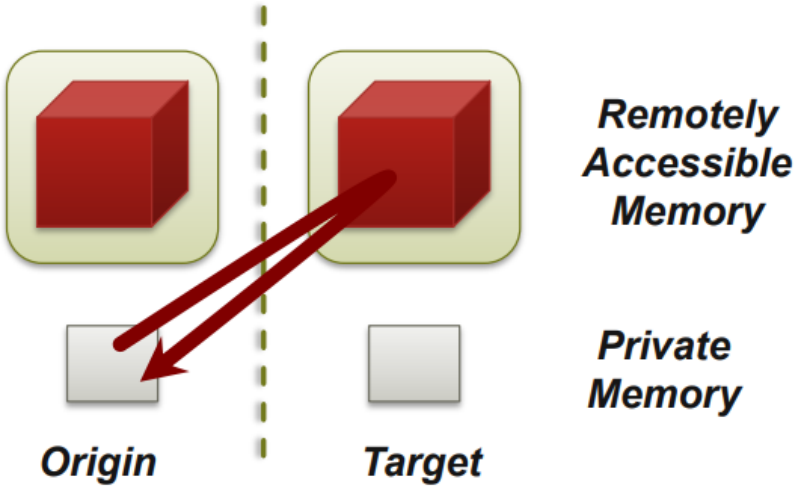

Рисунок 2.8 – Схема работы функции MPI_Get

Функция *MPI_Accumulate* (рисунок 2.9) является редукционной. Она применяет операцию объединения данных, например, MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, к содержимому инициировавшего обмен сообщениями и удалённого процессов и отправляет результат в буфер удалённого процесса.

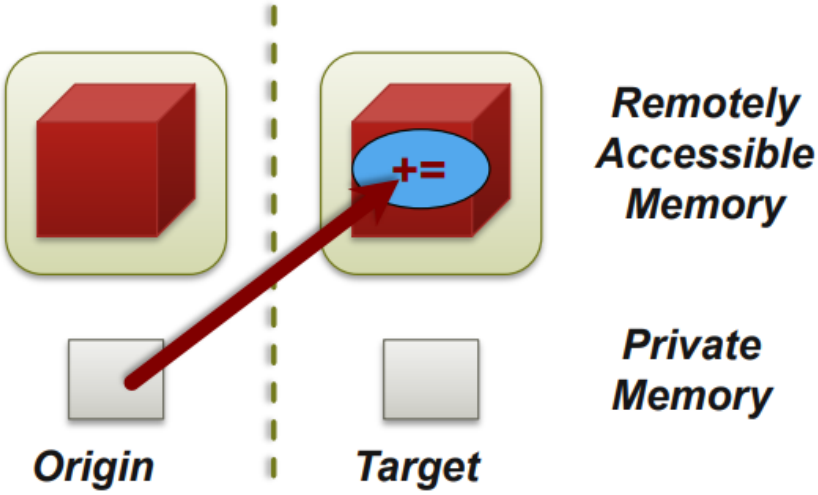

Рисунок 2.9 – Схема работы функции MPI_Accumulate

Функция *MPI_Get_accumulate* (рисунок 2.10), работает так же, как и MPI_Accumulate, но помещает результат редукции в буфер процесса, инициировавшего обмен сообщениями.

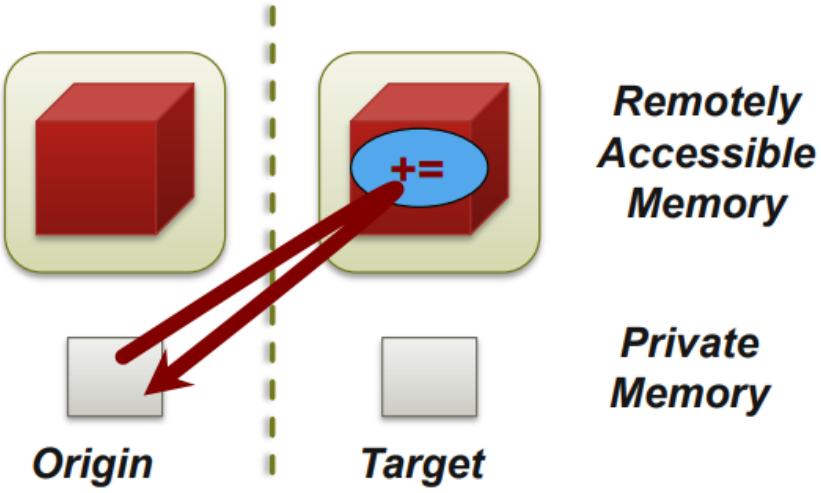

Рисунок 2.10 – Схема работы функции MPI_Get_accumulate

2.2.4 Синхронизация обменов в RMA

В MPI RMA односторонняя коммуникация делится на два вида: активная и пассивная [15-19].

При *активной коммуникации* (active one-sided communication) данные из одного процесса перемещаются в память другого, причём оба процесса участвуют в коммуникации. Этот вид коммуникации похож на двухстороннюю, но отличается тем, что все параметры передачи данных предоставляются одним процессом, а второй процесс участвует только в синхронизации. Активная коммуникация делится на два типа по способу синхронизации: простая (active target) и обобщённая (generalized active target).

При *пассивной коммуникации* (passive one-sided communication) в передаче явно участвует только процесс, инициировавший обмен сообщениями, причём несколько процессов могут обращаться к одному и тому же месту в удалённом окне. Синхронизация при пассивной коммуникации тоже называется пассивной.

Вызовы функций RMA обмена сообщениями должны происходить только в рамках эпох – промежутков выполнения передачи данных. Обмены сообщениями должны начинаться и завершаться вызовами функций синхронизации окна (MPI_Win).

При активной коммуникации с простой синхронизацией (рисунок 2.11) эпохи открываются и закрываются с помощью коллективного вызова функции MPI_Win_fence, все операции обмена сообщениями завершаются на втором её вызове. Этот вид коммуникации наиболее полезен для слабо синхронизированных алгоритмов, где граф взаимодействующих процессов меняется очень часто, или где каждый процесс взаимодействует со многими другими процессами.

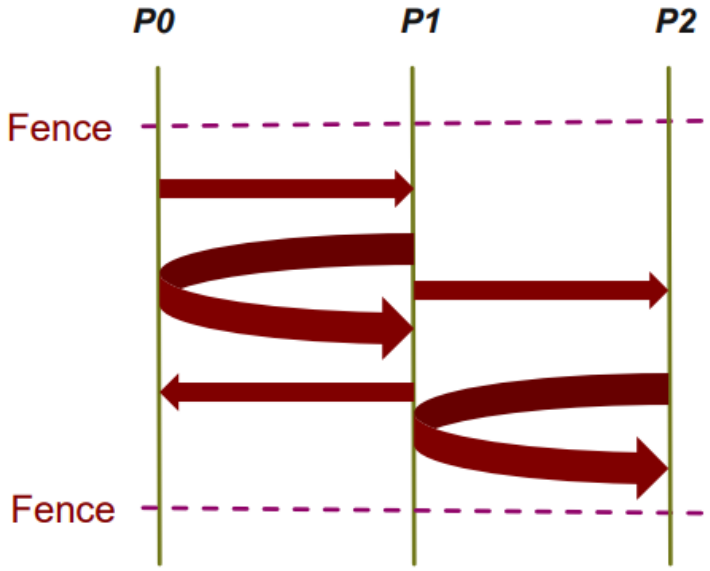

Рисунок 2.11 – Схема активной коммуникации с простой синхронизацией

При активной коммуникации с обобщённой синхронизацией (рисунок 2.12) в отличие от простой коммуникации процессы указывают, с кем совершают обмен данными. В этом виде коммуникации процессы могут меняться ролями. Инициирующий процесс открывает и закрывает эпоху с помощью функций MPI_Win_post и MPI_Win_wait, а удалённый процесс – MPI_Win_start и MPI_Win_complete. Данный вид коммуникации эффективен,

когда каждый процесс взаимодействует с небольшим числом соседей и когда граф коммуникаций фиксирован или меняется нечасто.

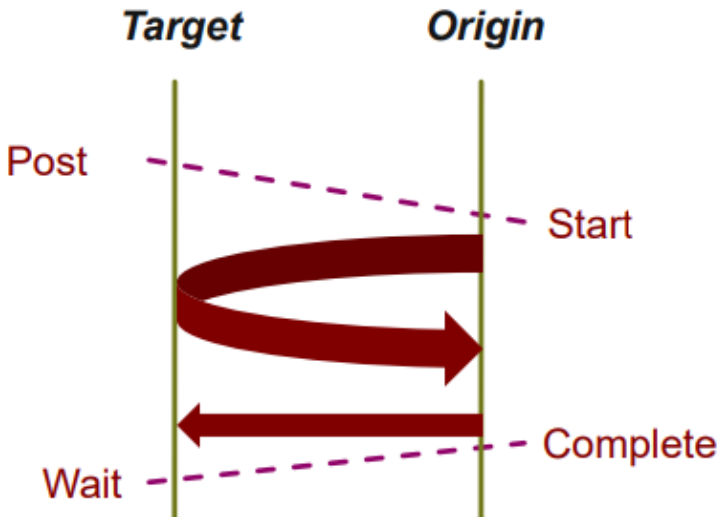

Рисунок 2.12 – Схема активной коммуникации с обобщённой синхронизацией

При пассивной коммуникации (рисунок 2.13) эпоха открывается для одного процесса с помощью функции MPI_Win_lock и для всех – с помощью MPI_Win_local_all, а закрывается с помощью MPI_Win_unlock и MPI_Winlock_all. После вызова функций MPI_Win_flush и MPI_Win_flush_all все обмены сообщениями завершаются с одним процессом и с несколькими процессами, соответственно.

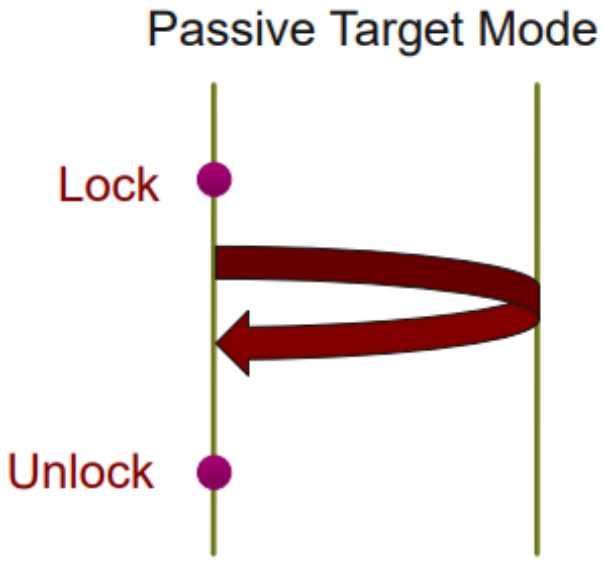

Рисунок 2.13 – Схема пассивной коммуникации

Выводы

Подведём итоги. Односторонние коммуникации представляют собой мощный механизм повышения производительности обменов сообщениями в MPI. Особое место занимает пассивная коммуникация как самая эффективная. По этой причине многие исследователи опираются именно на неё при реализации собственных алгоритмов и структур данных. Что касается активной коммуникации, то в последнее время она больше критикуется [20] и её предлагается исключить из стандарта в связи с тем, что этот вид коммуникации необоснованно сложен и приводит к накладным расходам на вычисления.

3 Обзор алгоритмов стека

Несмотря на то, что существует исчерпывающее число исследований в области неблокирующих структур данных и алгоритмов для SMM, научных работ для DMM крайне не хватает. Структура данных стек не является исключением. Более того, не трудно заметить, что на протяжении 10 лет исследователи мало уделяли внимания созданию распределённого стека, в основном занимаясь портированием на DMM таких структур данных, как очередь, множество и хеш-таблица.

Стек – очень важная и необходимая структура данных. Он используется в планировании обработки задач на основе LIFO, реализации пула объектов и, конечно, при обходе графа в глубину. Обход графа в глубину является основой поиска с возвратом (backtracking) и таких алгоритмов на графах, как поиск компонент связности, поиск максимального потока, топологическая сортировка и многих других. На вход перечисленным алгоритмам чаще всего подаются огромные массивы данных, обработка которых на одной вычислительной машине заняла бы слишком много времени.

В связи с тем, что существующие малочисленные алгоритмы неблокирующего стека для DMM пока что опираются на классические подходы для SMM, помимо реализаций для DMM будут рассмотрены реализации неблокирующего стека и решения проблем ABA и SMR для SMM. Также по причине недостатка работ для распределённого стека будет уделено внимание другим структурам данных, портированным на DMM.

3.1 Реализации неблокирующего стека для SMM

Считается, что первый lock-free стек для SMM, был описан в статье [21]. Он получил название *стек Трейбера* (Traiber Stack – TS) по фамилии автора статьи. Типичная реализация TS представляет собой односвязный список, в

котором используется операция CAS для атомарной модификации вершины стека.

С тех пор TS претерпел различные изменения. Появились такие его модификации, как TS с экспоненциальной стратегией отступления, Elimination back-off, Time-Stamped Stack.

В *TS с экспоненциальной стратегией отступления* (Exponential back-off [1]) после неудачной попытки изменить вершину стека поток приостанавливает своё выполнение на промежуток времени, в некоторое количество раз, как правило 2, превышающий предыдущий промежуток времени ожидания по той же самой причине. Идея состоит в том, что чем выше конкурентность за вершину стека, тем дольше потоку нужно ждать, пока она не снизится. Майкл и Скотт в статье [5] продемонстрировали, что такой стек является более производительным, чем блокирующий стек, но всё ещё недостаточно масштабируемым.

Идея *Elimination back-off stack* (EBS) [22] заключается в том, что операции над стеком push и pop являются комплементарными. Push и pop отменяют друг друга, если pop выполняется сразу после push. Таким образом, состояние стека остается неизменным, как будто эти две операции никогда не выполнялись. В Elimination back-off stack потоки устраняют друг друга с помощью массива исключения (elimination array), в котором они стараются ответить на противоположные вызовы путём обмена значениями, выбирая случайные элементы массива. Потоки, которые не могут найти пару или находят пару с непарным вызовом и тем самым не могут исключить вызов, могут повторить исключение в новом месте или обратиться к общей вершине стека. Комбинация структур данных, массивов и разделяемого стека является линеаризуемой, поскольку общий стек является линеаризуемым, а исключенные вызовы могут быть отсортированы таким образом, как будто они произошли в тех точках, где произошел обмен значениями.

В *Time-Stamped Stack* (стек с метками времени) [23] предлагают избавиться от упорядоченной схемы расположения узлов стека в памяти, потому что она сдерживает параллелизм. Авторы предлагают подход на основе временных меток, чтобы не упорядочивать узлы структуры данных. Идея состоит в том, что пары элементов можно оставить неупорядоченными, если связанные с ними операции вставки выполнялись одновременно, и порядок накладывается по мере необходимости при окончательном удалении. Результаты, приведённые в статье, указывают на более высокую масштабируемость TSS, чем EBS.

3.2 Решения проблем ABA и SMR на SMM

Описанные в предыдущем пункте стратегии снижения конкурентности за вершину стека способствуют увеличению масштабируемости, но не решают две главные проблемы неблокирующих алгоритмов: ABA и SMR. Среди различных методов решения этих проблем наибольшую известность получили меченые указатели, указатели опасности и подсчёт ссылок.

Меченые указатели (Tagged pointers) были предложены IBM [24]. Идея заключается в том, чтобы дополнить каждый указатель неким тегом, который может только инкрементироваться. Это позволяет сделать такой меченый указатель уникальным. По причине выравнивания размер меченого указателя будет составлять два машинных слова, и на 64-битной машине понадобится 128-битный CAS для атомарного обновления меченого указателя. Также стоит отметить, что подход на основе меченых указателей позволяет решить только проблему ABA, но не GMR.

Указатели опасности (Hazard Pointers – HP) были предложены в исследовании [4]. Подход HP заключается в том, что, когда некоторый поток желает получить доступ к объекту, который другой поток собирается удалить, он сначала устанавливает указатель опасности, ссылающийся на этот объект, тем самым сообщая другому потоку, что на данный момент этот объект ещё

используется и удалять его опасно. Когда объект перестаёт быть нужным, указатель опасности снимается. Поток, который хочет освободить память под некоторый узел, помещает его в специальный список на освобождение. Очищается этот список в основном двумя способами: в отдельном потоке или по вызову pop. Недостаток подхода HP заключается в том, что при каждом удалении узла необходимо просканировать массив, равный количеству потоков. Этот процесс можно немного оптимизировать, вместо удаления помещая узлы в некоторый список. Когда в списке накопится достаточное количество узлов, его можно очистить за один раз. Также существует подход, похожий на HP, но по какой-то причине не получивший широкого применения – Pass-the-back.

Подсчёт ссылок упоминается в различных источниках [2, 25] и встречается в нескольких вариантах. Подход сводится к подсчёту количества потоков, обращающихся к узлу. В одном из вариантов, описанном в книге [2], используется два счётчика ссылок на узел: внешний и внутренний, сумма которых равна количеству ссылок на узел. Внешний счётчик хранится вместе с указателем на узел и увеличивается на единицу при каждом чтении указателя. Когда поток завершает обработку узла, он уменьшает внутренний счётчик на единицу. Когда связь между внешним счётчиком и указателем становится больше не нужна, внутренний счётчик инкрементируется на значение внешнего минус один. Если внутренний счётчик обращается в ноль, это означает, что никаких ссылок на узел извне больше не осталось, и его можно освободить.

3.3 Реализации неблокирующего стека для DMM

Известна одно исследование [26], касающееся реализации неблокирующего распределённого стека. Авторы статьи предложили общий подход к портированию большинства известных структур данных,

разработанных для SMM, на DMM и демонстрируют его эффективность на примере lock-free TS, EBS и TSS.

Предложенный общий подход сводится к следующим принципам:

1. использование специального 64-битного глобального указателя, одна часть бит которого отводится под ранг запущенного процесса (в оригинале 24), а другая – под смещение в памяти вычислительного узла (в оригинале 48);
2. использование некоторого узкого множества атомарных примитивов синхронизации, доступных на многих платформах, достаточного для того, чтобы портировать SMM на DMM;
3. использование HP с его алгоритмом освобождения памяти, адаптированным для DMM.

Глобальный указатель из первого принципа специально был разработан 64-битным, чтобы его изменение поддерживалось аппаратно атомарными 64-битными операциями. Авторы продемонстрировали, что такого указателя вполне достаточно, чтобы адресовать память даже на самом мощном в мире суперкомпьютере из рейтинга TOP500.

Указатели опасности из третьего принципа отличаются от традиционных тем, что они реализованы на основе глобальных указателей из первого принципа.

Авторы заметили, что подходы на основе меченых указателей нельзя портировать на DMM, так как для такого решения необходим CAS над двойным словом.

Авторы предупредили, из того, что какая-то одна реализация структуры данных более быстрая и масштабируемая, чем другая, на SMM, ещё не следует, что она будет производительнее на DMM. Например, TS оказался быстрее, чем EBS и TSS на DMM несмотря на то, что на SMM TS медленнее. Причина заключается в разнице в количестве обменов между структурами данных. При портировании на DMM обмены становятся удалёнными между

процессами. Авторы показали это, продемонстрировав результаты на реализованных ими двух стеках с локальными массивами исключения (Elimination back-off node-aware – EBS NA), то есть если представляется такая возможность, взаимные обмены push и pop совершаются локально на одном вычислительном узле. Первый стек получил название EBS_NA, а второй – EBS2_NA. Разница между этими двумя стеками заключается в том, что EBS_NA сначала обращается к вершине стека, а лишь потом к массиву исключения, а EBS2_NA выполняет эти действия в обратном порядке. Оба варианта стека оказались производительнее, чем TS, причём самым производительным из этих двух оказался EBS2_NA.

Структуры данных, изложенные авторами в статье, были реализованы на основе библиотеки BCL с собственными расширениями к ней и опираются на односторонние коммуникации, в том числе MPI RMA. Исходный код проекта доступен на GitHub.

Результаты, приведённые авторами в статье, указывают на то, что их подход к портированию структур данных на DMM действительно работает. Однако стоит напомнить, что важным недостатком HP, который был взят за основу авторами для решения ABA и GMR является частое сканирование массива HP, которое у них ещё и сопровождается применением сортировки кучей к массиву HP. Возможно, HP – не самый масштабируемый подход.

3.4 Реализации отличных от стека неблокирующих структур данных для DMM

Существует несколько работ по реализации небольших библиотек распределённых структур данных. Им стоит уделить внимание, так как они нередко предоставляют базовые средства, необходимые при разработке собственных структур данных.

В библиотеке *BCL (Berkeley Core Library)* [27] реализованы очередь, хеш-таблица, массив и фильтр Блума. В библиотеке доступны односторонние

операции в том числе и для MPI RMA. Библиотека доступна к расширению, и её можно дополнить своей структурой данных. Однако на данный момент в BCL используется 128-битный глобальный указатель. Поскольку базовые высокоскоростные коммуникационные сети в настоящее время поддерживают RMW-примитивы, ограниченные 64 битами, то портировать существующие неблокирующие структуры данных, в том числе стек, с SMM на DMM с помощью BCL не представляется возможным. Очереди, реализованные в BCL, являются централизованными, то есть все процессы, запущенные на вычислительных узлах, должны хранить элементы структуры данных в адресном пространстве некоторого главного вычислительного узла. Во-первых, это создаёт узкое место обращения к структуре данных. Во-вторых, это решение ограничивает память, выделяемую под структуру данных, памятью только одного вычислительного узла. Также стоит отметить, что аллокаторы памяти из BCL используют блокировки.

В библиотеке HCL (Hermes Container Library) [28] реализованы упорядоченное и неупорядоченное множество, простая очередь и очередь с приоритетами. HCL опирается на новую технологию RPC over RDMA, поддерживаемую некоторыми сетевыми адаптерами. RPC (Remote Procedure Call) – протокол удалённого вызова процедур, а сама технология RPC over RDMA означает аппаратную поддержку удалённого прямого доступа к памяти для протокола RPC. Авторы HCL указывают на производительность от 2 до 12 раз превосходящую BCL. Тем не менее, структуры данных в HCL являются централизованными.

Выводы

Во-первых, известно лишь одно исследование [26] по реализации неблокирующего стека для DMM, в котором предлагается несколько интересных идей, но масштабируемость подхода HP вызывает сомнения.

Во-вторых, все авторы полагаются на односторонние коммуникации и их аппаратную поддержку. Также стоит отметить, что технология RDMA развивается, и чем теснее алгоритмы и структуры данных связаны с ней, как в библиотеке HCL, тем выше их производительность. Причём, чтобы иметь наилучшую аппаратную поддержку атомарных примитивов синхронизации, структуры данных, используемые для решения проблем ABA и SMR, и защищаемые ими поля целевой структуры данных, например, стека, должны укладываться в 64 бита.

В-третьих, при проектировании структуры данных стоит заранее предусмотреть её децентрализованность, чтобы память под структуру данных не ограничивалась памятью одного вычислительного узла и сам узел не стал узким местом в вычислительной системе.

В-четвёртых, делать точные прогнозы производительности структуры данных на DMM, опираясь только на данные её быстродействия на SMM, очень сложно, результат может быть совершенно иным.

4 Описание реализованного стека без блокировок

Центральными проблемами неблокирующих структур данных являются ABA и SMR. По этой причине проектирование собственной неблокирующей структуры данных стоит начинать с их решения. В работе [26] было продемонстрировано, что HP может быть успешно перенесён на DMM. Тем не менее, применение HP связано с высокими накладными расходами, описанными в предыдущем разделе. Существует ли другой, возможно, более эффективный способ решить проблемы ABA и SMR, который можно перенести с SMM на DMM? В данной выпускной квалификационной работе ответ на этот вопрос является утвердительным, а способ решения проблем – упомянутый в предыдущем разделе подсчёт ссылок [2].

В книге [2] Энтони Уильямс на примере неблокирующего стека для SMM описывает, как можно применить механизм подсчёта внешних и внутренних ссылок на узел структуры данных, чтобы решить проблемы ABA и SMR. Этот стек, являющийся TS, был взят за основу для реализации распределённого стека без блокировок. Самым простым средством повышения производительности такого TS является добавление к нему exponential back-off.

В данном разделе описывается реализация распределённого TS с exponential back-off, который имеет две версии: централизованную и децентрализованную.

4.1 Вспомогательные структуры данных

Чтобы применить схему управления неблокирующим стеком, описанную в [2], необходимо портировать структуры данных counted_node_ptr и node с SMM на DMM. На DMM эти структуры данных получили названия CountedNodePtr и Node и были преобразованы в классы. Роль этих структур данных заключается в реализации узлов односвязного списка, на котором

основан стек, хранении указателя на данные пользователя и подсчёте количества потоков, ссылающихся на данные.

На распределённой системе при децентрализации данных недостаточно смещения от некоторого начального адреса, чтобы определить местоположение этих данных в памяти. Необходимо знать, на каком вычислительном узле они находятся. По этой причине в распределённом стеке в роли указателя выступают две переменные: номер вычислительного узла (rank) и смещение от некоторого начального адреса (offset).

Класс CountedNodePtr содержит битовые поля m_externalCounter, m_rank и m_offset. m_externalCounter является внешним счётчиком ссылок, а m_rank и m_offset отвечают за адресацию в памяти структуры данных Node. Структура CountedNodePtr также является атомарной головой односвязного списка и вершиной стека, соответственно. Размер структуры данных CountedNodePtr составляет 8 байт.

Клас Node содержит поля m_acquired, m_reserved, m_internalCounter и m_countedNodePtrNext. m_acquired является индикатором захвата ресурса и принимает значения 0 (свободен) и 1 (захвачен). m_reserved служит для выравнивания до 4 байт. m_internalCounter – 32-битный внутренний счётчик ссылок. m_countedNodePtrNext указывает на следующую структуру Node и хранит количество потоков, обратившихся к ней. Размер структуры данных Node составляет 16 байт.

Причиной, по которой размер CountedNodePtr составляет 8 байт, является необходимость в аппаратной поддержке связанных с этой структурой атомарных изменений функциями MPI_Fetch_and_op и MPI_Compare_and_swap, чтобы достичь более высокой производительности. Однако такое решение также влияет на максимальное количество параллельных процессов и адресуемых в памяти элементов стека. В итоге было введено ограничение на максимальное количество процессов – 8190 (число 8191 было зарезервировано), и внешний счётчик ссылок занял 13 бит.

Отсюда поле m_rank тоже получило размер 13 бит. Оставшиеся 38 бит были отведены под поле m_offset, и поэтому максимальное количество элементов, которое может хранить стек составило 2^{38}. Учитывая, что в настоящий момент в разных исследованиях распределённые стеки и очереди масштабируются в среднем не больше, чем на два десятка процессов, введённые в данной работе ограничения не являются слишком строгими. При желании можно выбрать другие параметры.

4.2 Общее описание распределённого стека

При добавлении очередного элемента в стек должно существовать пространство в памяти, куда этот элемент можно сохранить, даже если это указатель. Это пространство может быть получено с помощью аллокатора памяти либо во время обращения к стеку, либо, заранее выделив память, при создании стека.

В первом случае на SMM можно просто вызвать функцию выделения памяти средствами используемого языка программирования. Однако для DMM такой вызов аллокатора должен быть удалённым. Средство MPI для выделения памяти на удалённом вычислительном узле без участия удалённого процесса самой программы на MPI не известно, и тогда возникает необходимость в постоянно запущенном потоке, отвечающем за выделение памяти по запросу удалённого процесса, на вычислительном узле. По причине сложности реализации данного подхода без блокировок было решено от него отказаться.

Во втором случае как на SMM, так и на DMM можно заранее выделить память необходимого объёма. При добавлении очередного элемента в стек данные сохраняются в свободную ячейку уже выделенной памяти. Именно такой подход был выбран в данной работе.

Разработанный стек был разбит на два класса: внутренний и внешний стек.

Внутренний стек оперирует структурами данных CountedNodePtr и Node и основан на стеке без блокировок из источника [2]. Элементами, над которыми внутренний стек совершает операции push и pop, являются две переменные описания местоположения пользовательских данных в распределённой памяти. Внутренний стек по отношению к внешнему выступает в качестве аллокатора свободных адресов не занятых ячеек памяти. При операции push внутренний стек выдаёт внешнему стеку свободный адрес, который становится занятым, и помещает этот адрес на вершину стека. Непосредственно перед помещением адреса на вершину стека внешний стек по этому адресу размещает данные пользователя. При выполнении операции pop внутренний стек снимает с вершины стека занятый адрес, который снова становится свободным. Непосредственно перед тем, как пометить адрес свободным, внутренний стек даёт внешнему стеку возможность забрать данные пользователя по этому адресу. Изначально вершина внутреннего стека инициализируется структурой CountedNodePtr с информацией об отсутствии данных – значение поля m_rank равняется максимальному положительному числу, которое можно составить из отведённого количества бит, минус 1.

Внешний стек реализует пользовательский интерфейс для работы со стеком, состоящий из функций push и pop, отвечает за сохранение пользовательских данных, управляет внутренним стеком и применяет exponential back-off, если это необходимо.

Внутренний стек заранее выделяет объём памяти равный $16 \cdot N$ байт, внешний стек – $M \cdot N$, где N – максимально возможное количество элементов в стеке, а M – размер элемента пользователя.

Атомарные операции над структурами данных, которые предполагает исходная реализация неблокирующего стека для SMM совершались с пассивной синхронизацией MPI RMA для достижения более высокой производительности. Операция CAS на DMM выполнялась с помощью функции MPI_Compare_and_swap, атомарное чтение – MPI_Fetch_and_op с

аргументом NO_OP, атомарная запись – MPI_Fetch_and_op с аргументом MPI_REPLACE, атомарный инкремент (fetch-and-add) – MPI_Fetch_and_op с аргументом MPI_SUM.

В централизованной версии стека весь массив объектов класса Node и пользовательских данных хранится на центральном процессе, и операция push помещает «указатель» на объект из этого массива. В децентрализованной версии каждый процесс имеет свой массив объектов класса Node и свой массив пользовательских данных, и при операции push поиск свободного объекта класса Node и размещение данных пользователя осуществляются в локальной памяти процесса.

Основная логика работы стека сосредоточена во внутреннем стеке. Исходный код внутреннего стека приводится в приложении А.

4.3 Инициализация и деинициализация стека

Инициализация внутреннего стека состоит из инициализации массива памяти под свободные адреса (массив Node) и вершины стека.

Инициализация внешнего стека состоит из инициализации массива пользовательских данных.

Инициализация массива Node в централизованной версии стека происходит следующим образом:

1. с помощью функции MPI_Win_create_dynamic создаётся RMA-окно к массиву Node;
2. с помощью функции MPI_Alloc_mem выделяется память под массив Node;
3. массив Node заполняется значениями по умолчанию;
4. с помощью функции MPI_Win_attach окно и память массива Node становится видна всем процессам;
5. с помощью функции MPI_Get_address извлекается адрес начала массива Node;

6. с помощью функции MPI_Bcast адрес начала массива Node передаётся всем процессам.

Значение по умолчанию для Node – свободная ячейка с обнулёнными счётчиками ссылок и указатель (m_rank и m_offset) на пустое значение в памяти. Адрес начала массива Node используется другими процессами при обращении к нему с помощью функций односторонней коммуникации.

Аналогичным образом происходит инициализация массива пользовательских данных внешнего стека в централизованной версии. Инициализация вершины стека внутреннего стека в централизованной версии отличается от инициализации предыдущих структур тем, что память выделяется только под один элемент класса CountedNodePtr.

Инициализация вершины стека в централизованной и децентрализованной версиях стека совпадают. В децентрализованной версии каждый процесс выделяет память под массив объектов класса Node и пользовательские данные. С помощью функции MPI_Bcast каждый процесс получает адреса массивов объектов класса Node и пользовательских данных в RMA-окнах других процессов и сохраняет их в свои локальные массивы адресов m_pBaseNodeArrAddresses и m_pUserDataBaseAddresses, которые затем использует для совершения односторонних коммуникаций.

В ходе деинициализации стека с помощью функции MPI_Free_mem освобождается память под пользовательские данные, массив Node и голову внутреннего стека, с помощью MPI_Win_free закрываются RMA-окна.

4.4 Описание операции push

При вызове пользовательской функции push внешний стек вызывает функцию push внутреннего стека и передаёт ему функцию обратного вызова (callback) для размещения данных пользователя по выданному внутренним стеком адресу. Если внутренний стек не смог выделить свободный адрес, так как они закончились, то внутренний стек передаёт в функцию обратного

вызова адрес, «указывающий на пустоту». Если внутреннему стеку не удалось заменить вершину стека, то он вызывает функцию обратного вызова для выполнения back-off.

Операция push внутреннего стека происходит следующим образом:

1. функция acquireNode ищет свободный адрес;
2. если свободных адресов нет, то вызывается функция обратного вызова внешнего стека с аргументом в виде адреса, «указывающего на пустоту», и функция push внутреннего стека завершается;
3. вызывается функция обратного вызова внешнего стека с аргументом в виде свободного адреса (с этим адресом также ассоциирован объект класса Node) для размещения данных пользователя;
4. в объект resHeadCountedNodePtr атомарно считывается (atomic read) текущее значение вершины стека;
5. создаётся объект countedNodePtrNext, ссылающийся на выделенный объект класса Node, с внешним счётчиком ссылок, равным единице.
6. в объект класса Node записывается адрес следующего такого объекта, полученный из считанной текущей вершины стека;
7. Операция CAS пытается заменить вершину стека на объект countedNodePtrNext;
8. Если операции CAS не удалось произвести замену, то выполнить функцию обратного вызова back-off внешнего стека и перейти к шагу 6.

Функция acquireNode из 1-го номера списка обходит массив объектов класса Node и с помощью операции CAS пытается заменить 32-битное поле очередного такого объекта со свободного (m_acquired = 0) на занятое (m_acquired = 1). Если за весь проход массива не удалось провести ни одну замену, функция возвращает адрес, «указывающий на пустоту».

4.5 Описание операции pop

При вызове пользовательской функции pop внешний стек вызывает функцию pop внутреннего стека и передаёт ему функцию обратного вызова для получения данных пользователя по снимаемому внутренним стеком адресу. Если в функцию обратного вызова в качестве аргумента передаётся адрес, «указывающий на пустоту», то внешний стек возвращает пользователю значение по умолчанию.

Операция pop внутреннего стека происходит следующим образом:

1. В объект oldHeadCountedNodePtr атомарно считывается текущее значение вершины стека;
2. Функция increaseHeadCount увеличивает количество потоков, обратившихся к вершине стека на чтение, на единицу и записывает результат в объект oldHeadCountedNodePtr;
3. Если адрес на объект класса Node в объекте oldHeadCountedNodePtr «указывает на пустоту», значит стек пуст, вызывается функция обратного вызова внешнего стека с аргументом в виде «пустого адреса» и функция pop завершается;
4. В объект node записывается значение объекта класса Node, находящееся по адресу из объекта oldHeadCountedNodePtr;
5. Создаётся объект countedNodePtrNext, в который записывается адрес следующего за объектом node объекта класса Node из объекта node;
6. Операция CAS пытается заменить вершину стека на объект countedNodePtrNext;
7. Если операции CAS не удалось произвести замену, то атомарно уменьшить внутренний счётчик ссылок на единицу, и если значение счётчика до атомарного уменьшения было равно единице, то адрес помечается снова свободным вызывается функция обратного вызова back-off внешнего стека и осуществляется переход к шагу 2. Иначе перейти к шагу 8;

8. Вызывается функция пользователя с аргументом в виде снимаемого со стека адреса, по которому внешний стек забирает данные пользователя;
9. Внутренний счётчик ссылок по адресу из объекта oldHeadCountedNodePtr плюс 4 (смещение внутреннего счётчика внутри класса Node) атомарно уменьшается на значение внешнего счётчика минус 2;
10. Если значение внутреннего счётчика ссылок до атомарного уменьшения было равно минус значение уменьшения, то адрес помечается свободным.

Функция increaseHeadCount работает следующим образом:

1. Создаётся объект newCountedNodePtr;
2. Создаётся объект resCountedNodePtr, который инициализируется значением olHeadCountedNodePtr;
3. Объектам newCountedNodePtr и oldHeadCountedNodePtr присваивается значение resCountedNodePtr;
4. Внешний счётчик ссылок объекта newCountedNodePtr увеличивается на 1;
5. Операция CAS пытается заменить вершину стека на объект newCountedNodePtr;
6. Если операции CAS не удалось произвести замену, то перейти на шаг 3, иначе перейти на шаг 7;
7. Присвоить внешнему счётчику ссылок объекта oldHeadCountedNodePtr значение внешнего счётчика ссылок объекта newCountedNodePtr.

Увеличение внешнего счётчика ссылок на вершину стека в функции increaseHeadCount необходимо, чтобы показать остальным процессам, что текущий процесс ссылается на эту вершину, и освобождать память под неё нельзя.

Неуспешная замена вершины стека на шаге 6 операции pop означает, что другой процесс либо уже удалил вершину, либо начал её читать. По этой причине текущему процессу нужно начинать удаление вершины снова, но сначала нужно атомарно уменьшить счётчик на текущий узел, так как текущий процесс перестаёт его читать. Если внутренний счётчик обнулился, тогда узел можно удалить из односвязного списка. Операция fetch-and-add возвращает значение до инкремента, поэтому для проверки обнуления сравнивается предыдущее значение с 1.

В случае успешной замены вершины стека на шаге 6 операции pop предыдущая вершина переходит во владение текущего процесса, и данные можно извлечь. Уменьшение внутреннего счётчика ссылок на значение минус 2 связано с тем, что вершина стека сначала была исключена из односвязного списка, а затем текущий процесс к ней больше не обращается.

5 Проведение экспериментов

С целью исследования масштабируемости и скорости работы распределённого стека были проведены эксперименты для его централизованной и децентрализованной версии.

Эксперименты проводились на вычислительной машине с шестиядерным процессором Intel i5-11400. Базовая частота процессора составляет 2,6 ГГц, максимальная – 4,4 ГГц. Размер кэш-памяти L1 составляет 288 Кб для данных и 192 Кб для аппаратных инструкций, L2 – 3 Мб, L3 – 12 Мб. В качестве операционной системы использовалась Linux Ubuntu 20.04.

Для экспериментов был выбран один бенчмарк, который заключался в исполнении 15000 случайных операций push и pop в общем на все процессы. Операции выбирались равновероятно. Были выбраны следующие параметры exponential back-off: минимальный – 1 нс, максимальный – 100 нс. По завершению очередного испытания сохранялось максимальное время исполнения бенчмарка среди всех процессов. Это время было использовано для вычисления пропускной способности распределённого стека – количества случайных операций push и pop в секунду. Бенчмарк проводился для количества процессов от 1 до 6.

Полученные результаты для централизованной версии стека были сведены в таблицу (таблица 5.1).

Таблица 5.1 – Результаты для централизованной версии стека

Время, с	Операции/с
222,3803	67
34,5492	434
29,2377	513
27,6017	543
30,3646	493
27,0042	555

Полученные результаты для децентрализованной версии стека были сведены в таблицу (таблица 5.2).

Таблица 5.2 – Результаты для децентрализованной версии стека

Время, с	Операции/с
220,3682	68
8,2242	1823
4,0486	3704
2,4642	6087
1,6790	8933
1,4777	10150

Также по результатам бенчмарка были построены графики пропускной способности централизованной и децентрализованной версии распределённого стека (рисунок 5.1).

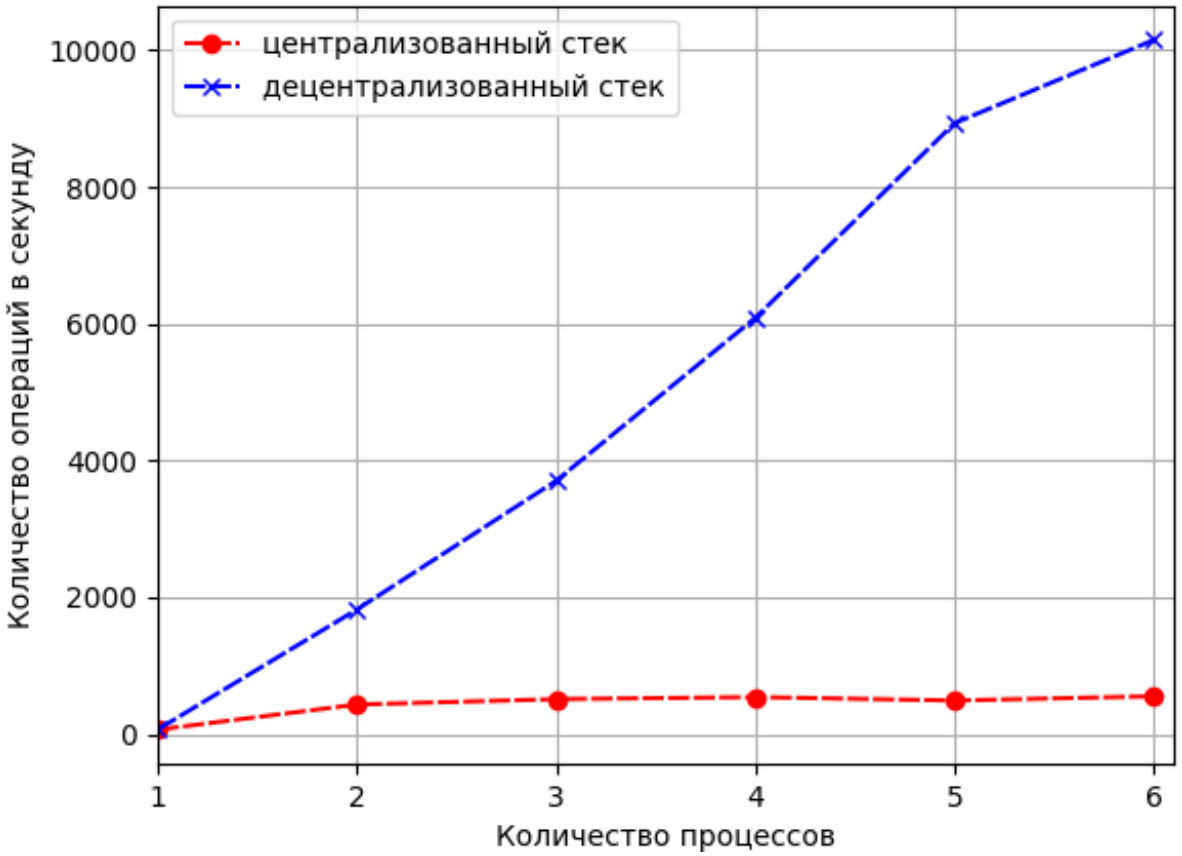

Рисунок 5.1 – Графики централизованной и децентрализованной версий стека

По результатам графиков (рисунок 5.1) можно сделать вывод, что централизованная версия распределённого стека масштабируется только до двух процессов, децентрализованная версия является более производительной и масштабируемой, чем централизованная, но её производительность на 6 процессах начинает замедляться.

6 Оценка и защита результатов интеллектуальной деятельности

Защита законных прав и интересов в сфере интеллектуальной собственности является одной из приоритетных задач правового государства. Одним из важнейших условий их вовлечения в хозяйственный оборот и формирования инновационных рынков является обеспечение правовой охраны результатов научно-технической деятельности. Правовая охрана – это основа защиты интересов авторов, инвесторов и производителей от недобросовестной конкуренции при распространении этих результатов.

Важность исследований в области оценки и охраны результатов интеллектуальной деятельности связана с тем, что переход к рыночной экономике и развитие предпринимательства в России привели к существенным изменениям в области правового регулирования, увеличению доли наукоемкой продукции и необходимости её оценки и правовой охраны.

Главной задачей этого раздела является получение практических навыков в области оценки и защиты результатов интеллектуальной деятельности. Для достижения поставленной цели необходимо решить следующие задачи:

Сформулировать объект исследования дополнительного раздела на основании полученных результатов в основной части ВКР.

Оценить рыночную стоимость объекта исследования (результата интеллектуальной деятельности) на основании предложенной методики.

Исследовать законодательные основы правовой защиты результатов интеллектуальной деятельности.

Методика проведения оценки и защиты результатов интеллектуальной деятельности осуществлялась в соответствии с методическими указаниями по оценке и защите результатов интеллектуальной деятельности выпускных квалификационных работ [29].

6.1 Описание результата интеллектуальной деятельности

Целью выпускной квалификационной работы (ВКР) является реализация распределённого стека без использования блокировок в модели удаленного доступа к памяти.

Стек является одной из самых часто применяемых структур данных. Он используется в планировании обработки задач на основе LIFO, реализации пула объектов и, конечно, при обходе графа в глубину. На вход алгоритмам обработки графов чаще всего подаются огромные массивы данных, обработка которых на одной вычислительной машине заняла бы слишком много времени.

Разработанный на языке программирования C++ алгоритм стека может стать востребованным в области параллельных вычислений на распределённых вычислительных машинах, например, суперкомпьютерах. Разработанный стек может быть использован в качестве вспомогательной структуры данных, например, пула распределённой памяти, для реализации более сложных структур данных. Замечательным свойством алгоритма является отсутствие блокировок, что позволяет достичь более высокой масштабируемости и, соответственно, производительности на вычислительных кластерах, чем с блокировками.

Выбор объекта исследования должен осуществляться на основании части четвёртой Гражданского кодекса Российской Федерации (ГК РФ) [30], согласно которому, результатами интеллектуальной деятельности и приравненными к ним средствами индивидуализации юридических лиц, товаров, работ, услуг и предприятий, которым предоставляется правовая охрана (интеллектуальной собственностью), являются:

- произведения науки, литературы и искусства;
- программы для электронных вычислительных машин (программы для ЭВМ);
- базы данных;

- исполнения;
- фонограммы;
- сообщение в эфир или по кабелю радио- или телепередач (вещание организаций эфирного или кабельного вещания);
- изобретения;
- полезные модели;
- промышленные образцы;
- селекционные достижения;
- топологии интегральных микросхем;
- секреты производства (ноу-хау);
- фирменные наименования;
- товарные знаки и знаки обслуживания;
- наименования мест происхождения товаров;
- коммерческие обозначения.

Из представленного списка объектов, подлежащих оценке результатов интеллектуальной деятельности, результаты, полученные в процессе выполнения выпускной квалификационной работы, относятся к объекту «программы для электронных вычислительных машин (программы для ЭВМ)».

6.2 Оценка рыночной стоимости результата интеллектуальной деятельности

При проведении оценки рыночной стоимости результата интеллектуальной деятельности использую в основном доходный, сравнительный и затратный подходы.

Доходный подход рекомендуется применять, когда существует достоверная информация, позволяющая прогнозировать будущие доходы,

которые объект оценки способен приносить, а также связанные с объектом оценки расходы.

Сравнительный подход рекомендуется применять, когда доступна достоверная и достаточная для анализа информация о ценах и характеристиках объектов-аналогов. При этом могут применяться как цены совершенных сделок, так и цены предложений.

Затратный подход преимущественно применяется в тех случаях, когда существует достоверная информация, позволяющая определить затраты на приобретение, воспроизводство либо замещение объекта оценки.

Область распределённых структур данных и алгоритмов является достаточно перспективной, так как открывает возможности эффективной утилизации гораздо больших вычислительных ресурсов, чем вычислительные системы с общей памятью. Тем не менее, исследований в этой области крайне мало, и аналогичные прикладные решения встречаются крайне редко. По этой причине рынок не представлен коммерческими продуктами, имеющими сходства с объектом оценки, и поэтому спрогнозировать будущие доходы, которые он способен принести, невозможно. Следовательно, доходный и сравнительный подходы не могут быть использованы в данном случае. Однако имеющиеся сведения о расходах на разработку программного обеспечения (ПО) позволяют применить затратный подход.

Методика проведения затратного подхода осуществлялась согласно методическим указаниям по экономическому обоснованию выпускных квалификационных работ [31].

Ниже перечислены основные статьи затрат на разработку ПО:

- расходы на оплату труда;
- отчисления на социальные нужды;
- накладные расходы;
- амортизация оборудования:
- услуги сторонних организаций.

6.2.1 Расчет затрат на оплату труда

Для расчёта затрат на оплату труда был сформирован план проводимых работ.

В связи с тем, что характер проводимых работ предполагает глубокие знания в области параллельных вычислений и компьютерной математики, а также свободное владение языками программирования низкого уровня, данные о средней заработной плате студента были собраны для C++-программиста уровня middle, а для научного руководителя – senior. По оценке «Хабр Карьера» медианная заработная плата таких специалистов в РФ во второй половине 2022 года составила 140 тыс. рублей для уровня middle и 238 тыс. рублей – для senior [32].

Таким образом, если принять, что в месяце двадцать один рабочий день, то дневная ставка студента и научного руководителя составят:

$$С_{студ} = \frac{140000}{21\ день} = 6666{,}67 \text{ руб./день},$$

$$С_{рук} = \frac{238000}{21\ день} = 11333{,}33 \text{ руб./день}.$$

Был определён план работ студента (таблица 6.1).

Таблица 6.1 – План работ студента

Этапы и содержание выполняемых работ	Трудоемкость, чел./день	Ставка, руб./день
Составление ТЗ	1	6666,67
Изучение технической литературы	44	6666,67
Обзор аналогов	15	6666,67
Разработка распределённого стека	61	6666,67
Тестирование и отладка	25	6666,67
Согласование с руководителем и исправление ошибок	7	6666,67
Оформление дипломного проекта	14	6666,67

Был определён план работ научного руководителя (таблица 6.2).

Таблица 6.2 – План работ научного руководителя

Этапы и содержание выполняемых работ	Трудоемкость, чел./день	Ставка, руб./день
Составление ТЗ	1	11333,33
Обзор аналогов	15	11333,33
Согласование с руководителем и исправление ошибок	7	11333,33

Трудозатраты научного руководителя составят:

$$\mathrm{Т}_{\text{рук}} = 23 \text{ чел./дней.}$$

Трудозатраты студента составят:

$$\mathrm{Т}_{\text{студ}} = 167 \text{ чел./дней.}$$

Общая формула расчёта заработной платы:

$$\mathrm{З}_{\text{осн.з./пл.}} = \sum_{i=1}^{k} T_i \cdot C_i,$$

где T_i – время, затраченное i-ым исполнителем на проведение работ в днях; $\mathrm{С}_i$ – ставка i-го исполнителя в рублях в день.

Применительно к конкретно нашему случаю основная заработная плата рассчитывается по формуле:

$$\mathrm{З}_{\text{осн.з./пл.}} = \mathrm{Т}_{\text{рук.}} \cdot \mathrm{С}_{\text{рук.}} + \mathrm{Т}_{\text{студ.}} \cdot \mathrm{С}_{\text{студ.}}$$

Основная заработная плата составит:

$$\mathrm{З}_{\text{осн.з./пл.}} = 23 \cdot 11333{,}33 + 167 \cdot 6666{,}67 = 1374000{,}00 \text{ руб.}$$

Дополнительная заработная плата рассчитывается по формуле:

$$\mathrm{З}_{\text{доп.з./пл.}} = \mathrm{З}_{\text{осн.з./пл.}} \cdot \frac{\mathrm{Н}_{\text{доп.}}}{100},$$

где $\mathrm{Н}_{\text{доп.}}$ – норматив дополнительной заработной платы. Дополнительная заработная плата составит:

$$\mathrm{З}_{\text{доп.з./пл.}} = 1374000 \cdot 8{,}3 \,/100 = 114042 \text{ руб.}$$

Формула расчёта отчислений на социальные нужды:

$$З_{соц.} = (З_{осн.з./пл.} + З_{доп.з./пл.}) \cdot \frac{Н_{соц..}}{100},$$

где $Н_{соц.}$ – норматив отчислений на страховые взносы на обязательное социальное, пенсионное и медицинское страхование. Отчисления на социальные нужды от основной и дополнительной заработной платы равны:

$$З_{соц} = (1374000 + 114042) \cdot 30{,}2/100 = 449338{,}68 \text{ руб.}$$

6.2.2 Расчет накладных расходов

В данной работе норматив накладных расходов равен 20% от суммы основной и дополнительной заработной платы. Накладные расходы рассчитываются по формуле:

$$C_{нр} = (З_{осн.з./пл.} + З_{доп.з./пл.}) \cdot \frac{Н_{нак.}}{100},$$

где $Н_{нак.}$ – норматив накладных расходов.

Накладные расходы равны:

$$C_{нр} = (1374000 + 114042) \cdot \frac{20}{100} = 297608{,}40 \text{ руб.}$$

6.2.3 Издержки на амортизацию оборудования

В процессе разработке ПО использовался ноутбук. Стоимость ноутбука ASUS Vivobook Pro 15 – 73 490 рублей [33].

Согласно постановлению правительства РФ от 01.01.2002 №1 (ред. От 28.04.2018) «О Классификации основных средств, включаемых в амортизационные группы» [34] персональные компьютеры и печатающие устройства к ним, относятся ко второй группе. Срок полезного использования оборудования равен от двух до трех лет включительно. Срок эксплуатации три года. Норма амортизации $Н_{АМ}$= 33,3%.

Годовая амортизация рассчитывается по формуле:

$$А_{г} = К_{об.} \cdot \frac{Н_{АМ}}{100},$$

где $К_{об.}$ − стоимость оборудования; $Н_{АМ}$ − норма амортизации.

Годовая амортизация составит:

$$А_г = 73490 \cdot \frac{33,33}{100} = 24472,17 \text{ руб.}$$

Сумма амортизации за рабочий день рассчитывается по формуле:

$$А_д = \frac{А_г}{N}.$$

В 2023 году 247 рабочих дней, соответственно сумма амортизации за один рабочий день составит:

$$А_д = 24472,17/247 = 99 \text{ руб. } 8 \text{ коп.}$$

Таким образом, амортизация оборудования за всё время на разработку ПО составит:

$$А_{вкр} = 99 \text{ руб.} 17 \text{ коп.} \cdot 167 = 16545 \text{ руб.} 97 \text{ коп.}$$

6.2.4 Расходы на услуги сторонних организаций

В процессе разработки ПО был необходим доступ к интернету. Оператором связи выступил МТС, интернет 500 Мбит/с – 650 руб. в месяц в рамках тарифа «ФИТ Интернет 500 Мбит/с», ссылка на тариф [35]. Интернет использовался в течение 6 месяцев. Таким образом, затраты на услуги сторонних организаций составили 3250 руб.

6.2.5 Себестоимость разработки программного обеспечения

Была определена себестоимость разработки программного обеспечения по статьям расхода и в целом (таблица 6.3).

Таблица 6.3 – Смета затрат на разработку ПО

Наименование статьи	Сумма, руб.	Структура себестоимости, (%)
Расходы на услуги сторонних организаций	3250	0,14
Издержки на амортизацию оборудования	16545,97	0,73

Продолжение таблицы 6.3

Наименование статьи	Сумма, руб.	Структура себестоимости, (%)
Расходы на оплату труда	1488042,00	66,03
Накладные расходы	297608,40	13,21
Отчисления на социальные нужды	449388,68	19,94
ИТОГО:	2254835,05	100,00

В итоге, себестоимость разработки программного обеспечения оценивается в 2254835,05 рублей.

6.3 Правовая защита результатов интеллектуальной деятельности

Правовая охрана объектов интеллектуальной собственности обеспечивается на основе патентного законодательства, законодательства по защите недобросовестной конкуренции, авторского права, законодательства о средствах индивидуализации.

Нарушение прав интеллектуальной собственности наказывается в соответствии с гражданским, уголовным и административным законодательством.

Технические новшества могут быть защищены от использования другими лицами с помощью патентов на изобретения, патентов на промышленные образцы, свидетельств на полезные модели и свидетельств на ноу-хау.

Согласно пункту 1 статьи 1259 части 4 ГК РФ программы для ЭВМ относятся к объектам авторских прав, которые охраняются как литературные произведения. Следовательно, в данной работе результат интеллектуальной деятельности защищается авторским правом. Авторское право распространяется на все компьютерные программы и базы данных,

независимо от того, находятся ли они в общественном достоянии и независимо от их материального носителя, цели или полезности.

Согласно пункту 2 статьи 1255 части 4 ГК РФ автору произведения принадлежат следующие права:

- исключительное право на произведение;
- право авторства;
- право автора на имя;
- право на неприкосновенность произведения;
- право на обнародование произведения.

В процессе разработки ПО студента консультировал научный руководитель. В связи с этим согласно пункту 1 статьи 1258 части 4 ГК РФ, студент и научный руководитель признаются соавторами.

Согласно пункту 1 статьи 1281 части 4 ГК РФ исключительное право на произведение, созданное в соавторстве, действует в течение всей жизни автора, пережившего других соавторов, и 70 лет, считая с 1 января года, следующего за годом его смерти.

ЗАКЛЮЧЕНИЕ

В работе были рассмотрены достоинства и недостатки блокирующего и неблокирующего подходов при организации конкурентного доступа к общим данным, проблемы, связанные с каждым из этих двух подходов, были даны рекомендации по поводу применения этих подходов, была рассмотрена аппаратная и программная поддержка технологии односторонних коммуникаций, включая передовой стандарт MPI, используемый для обмена сообщениями между процессами в системе распределённой памятью, был проведён обзор текущего состояния исследований в области распределённых структур данных, отмечены их сильные и слабые стороны.

В результате выполнения выпускной квалификационной работы был предложен распределённый стек без блокировок на основе подсчёта ссылок, реализованный на языке программирования C++ и использующий модель удалённого доступа к памяти библиотеки MPICH стандарта MPI. Было предложено две версии разработанного стека: централизованная и децентрализованная. Были получены экспериментальные результаты и проведён анализ, в ходе которого было установлено, что централизованная версия стека масштабируется только до 2 процессов, в то время как децентрализованная версия масштабируется как минимум до 6 процессов. Полученный стек может найти применение на вычислительных системах с распределённой памятью в качестве вспомогательной структуры данных, например, для алгоритмов на графах, или в составе других более сложных структур данных. Была дана оценка работы с экономической и правовой точек зрения: посчитаны статьи расходов на выполнение работы, описаны правовые нормы защиты полученных результатов интеллектуальной деятельности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Herlihy, M. The Art of Multiprocessor Programming = Искусство мультипроцессорного программирования / Herlihy M., Shavit N. – Elsevier Inc., 2008. – 529 с.
2. Уильямс, Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. / Уильямс С. ; пер. с англ. Слинкин А. А. – М. : ДМК Пресс, 2012. – 672 с.
3. Herlihy M. Linearizability: A Correctness Condition for Concurrent Object = Линеаризуемость: Доказательство Условие корректности для конкурентного объекта / M. Herlihy, J. M. Wing // ACM Transactions on Programming Languages and Systems. – 1990. – Vol. 12. – С. 463–492.
4. Michael M. Hazard pointers: safe memory reclamation for lock-free objects = Указатели опасности: безопасное освобождение памяти на параллельных и распределённых системах // IEEE Transactions on Parallel and Distributed Systems. – 2004. – Vol. 15. – С. 491–504.
5. Michael M. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors = Неблокирующие алгоритмы и упреждающая безопасная блокировка на мультипроцессорах с разделяемой памятью / Michael M. M., Scott M. L. // Journal of Parallel and Distributed Computing. – 1998. – Vol. 51. – С. 1–26.
6. Таненбаум, Э. Архитектура компьютера. / Таненбаум Э., Остин Т. – 6- е изд. – СПб.: Питер, 2013. — 816 с.
7. Сравнение RDMA и Message Passing [Электронный ресурс]. – URL: https://www.hpcwire.com/2006/10/06/why_compromise-1/ (дата обращения 10.05.23)

8. Критика RDMA [Электронный ресурс]. – URL: https://www.hpcwire.com/2006/08/18/a_critique_of_rdma-1/ (дата обращения 10.05.23)
9. Обзор RDMA [Электронный ресурс]. – URL: https://www.hpcwire.com/2006/09/15/a_tutorial_of_the_rdma_model-1/ (дата обращения 10.05.23)
10. MPI Forum [Электронный ресурс]. – URL: https://www.mpi-forum.org/ (дата обращения 10.05.23)
11. OpenMPI [Электронный ресурс]. – URL: https://www.open-mpi.org/ (дата обращения 10.05.23)
12. IntelMPI [Электронный ресурс]. – URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html (дата обращения 10.05.23)
13. MPICH Documentation [Электронный ресурс]. – URL: https://www.mpich.org (дата обращения 10.05.23)
14. Стандарт MPI-4 [Электронный ресурс]. – URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (дата обращения 10.05.23)
15. Schuchart J. Using MPI-3 RMA for Active Messages = Использование MPI-3 RMA для активных сообщений / J. Schuchart, A. Bouteiller, G. Bosilca // IEEE. – 2019. – С. 47–56.
16. Gerstenberger R. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided = Обеспечение масштабируемого программирования удалённого доступа к памяти с помощью одностороннего MPI-3/ R. Gerstenberger, M. Besta, T. Hoefler // Communications of the ACM. – 2013. –Vol. 61. – С. 106–113.
17. Hoefler T. Remote Memory Access Programming in MPI-3 = Программирование удалённого доступа к памяти в MPI-3 / T. Hoefler, J. Dinan, R. Thakur [и др.] // ACM Transactions on Parallel Computing. – 2013. – Vol. 2. –С. 1–26.

18. MPI Remote Memory Access Programming (MPI3-RMA) and Advanced MPI Programming https://spcl.inf.ethz.ch/Publications/.pdf/MPI_RMA_and_advanced_MPI.pdf (дата обращения 10.05.23)
19. Dinan J. An implementation and evaluation of the MPI 3.0 one-sided communication interface = Реализация и оценка интерфейса односторонней связи MPI 3.0 / J. Dinan, P. Balaji, D. Buntinas [и др.] // Wiley InterScience. – 2016. – Vol. 28. – С. 4385–4404.
20. Schuchart J. Quo Vadis MPI RMA? Towards a More Efficient Use of MPI One-Sided Communication = Куда движется MPI? На пути к более эффективному использованию односторонней коммуникации MPI / J. Schuchart, C. Niethammer, J. Gracia [и др.] // EuroMPI: конференция. – 2021.
21. Treiber R. Systems programming: Coping with parallelism = Программирование систем: копирование с параллелизмом // Technical Report RJ 5118, IBM Almaden Research Center. – 1986.
22. Hendler D. A scalable lock-free stack algorithm = Алгоритм масштабируемого стека без блокировок / D. Hendler, N. Shavit, L. Yerushalmi // Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures. – 2004. – С. 206–215.
23. Dodds M. A scalable, correct time-stamped stack = Масштабируемый корректный стек с метками времени / M. Dodds, A. Haas, C.M. Kirsch // ACM SIGPLAN. – 2015. – С. 233–246.
24. IBM Systems/38 [Электронный ресурс]. – URL: https://homes.cs.washington.edu/~levy/capabook/Chapter8.pdf (дата обращения 10.03.23)
25. Detlefs D. Lock-Free Reference Counting / D. Detlefs, P. Martin, M. Moir // Distributed Computing. – 2002. – С. 255–271.

26. Diep D. A general approach for supporting nonblocking data structures on distributed-memory systems = Общий подход к поддержке неблокирующих структур данных на системах с распределённой памятью / D. Dang, P. H. Ha, K. Fürlinger // Elseivier Inc. – 2023. – Vol. 173 – С. 48–60.
27. Brock B. / BCL: A Cross-Platform Distributed Data Structures Library = BCL: Кроссплатформенная библиотека распределённых структур данных / B. Brock, A. Buluç, K. Yelick // Proceedings of the 48th International Conference. –2019. – Article No. 102. – С. 1–10.
28. Devarajan H. HCL: distributing parallel data structures in extreme scales = HCL: распределение параллельных структур данных в экстремальных масштабах / H. Devarajan, A. Kougkas, K. Bateman // IEEE International Conference on Cluster Computing (CLUSTER): конференция. – 2020. – С. 248–258.
29. Магомедов, М. Оценка и защита результатов интеллектуальной деятельности: учебно-методическое пособие по выполнению дополнительного раздела выпускных квалификационных работ магистров / Магомедов М. Н., Чигирь М. В. – СПб.: СПбГЭТУ “ЛЭТИ”, 2019. – 26 с.
30. Гражданский кодекс Российской Федерации часть 4 [Электронный ресурс]. – URL: http://www.consultant.ru/document/Cons_doc_LAW_64629 (дата обращения 10.05.23)
31. Алексеева, О. Методические указания по экономическому обоснованию выпускных квалификационных работ бакалавров: Метод. Указания / Алексеева О. Г. – СПб.: СПбГЭТУ “ЛЭТИ”, 2013. – 17 с.
32. Хабр Карьера Зарплаты разработчиков во второй половине 2022: языки и квалификации [Электронный ресурс]. – URL:

https://habr.com/ru/companies/habr_career/articles/719730/ (дата обращения 10.05.23)

33. Стоимость ноутбука ASUS Vivobook Pro 15 [Электронный ресурс]. – URL: https://www.citilink.ru/product/noutbuk-asus-vivobook-pro-m6500qh-hn038-ryzen-5-5600h-16gb-ssd512gb-gt-1886447/ (дата обращения 10.05.23)
34. О классификации основных средств, включаемых в амортизационные группы: постановление Правительства Рос. Федерации от 01.01.2002 №1 (ред. от 18.11.2022) // Собрание законодательства Российской Федерации, 01.01.2002. – №1.
35. Тариф «ФИТ Интернет 500 Мбит/с [Электронный ресурс]. – URL: https://internet.gde-luchshe.ru/provider/mts/ (дата обращения 10.05.23)

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ВНУТРЕННЕГО СТЕКА

InnerStack.h

```
#include <utility>
#include <cstddef>
#include <mpi.h>
#include <spdlog/spdlog.h>
#include <functional>
#include <memory>
#include "CountedNodePtr.h"
#include "Node.h"
namespace rma_stack::ref_counting {
        class InnerStack {
        public:
            static const int HEAD_RANK = 0; // ранг центрального процесса
            InnerStack(MPI_Comm comm,
                               MPI_Info info,
                               bool t_centralized,
                       size_t t_elemsUpLimit,
                       std::shared_ptr<spdlog::logger> t_logger);
            void push(const std::function<void(GlobalAddress)>
            &putDataCallback, const std::function<void()> &backoffCallback);
            void pop(const std::function<void(GlobalAddress)>
            &getDataCallback, const std::function<void()> &backoffCallback);
            void release();
            [[nodiscard]] size_t getElemsUpLimit() const;
        private:
            void initRemoteAccessMemory(MPI_Comm comm, MPI_Info info);
            void increaseHeadCount(CountedNodePtr& oldHeadCountedNodePtr);
            [[nodiscard]] GlobalAddress acquireNode(int rank) const;
            void releaseNode(GlobalAddress nodeAddress) const;
        private:
            size_t m_elemsUpLimit{0}; // ограничение на кол-во элементов
            int m_rank{-1}; // ранг данного процесса
            bool m_centralized; // булевое значения для различения
                  централизованной и децентрализованной версии стека
            MPI_Win m_headWin{MPI_WIN_NULL}; // RMA-окно вершины стека
            CountedNodePtr* m_pHeadCountedNodePtr{nullptr}; // указатель на
                                                              вершину стека
            MPI_Aint m_headAddress{(MPI_Aint)MPI_BOTTOM}; // адрес вершины
                                                 стека на центральном процесса
            MPI_Win m_nodesWin{MPI_WIN_NULL}; // RMA-окно для массива узлов
            Node* m_pNodesArr{nullptr}; // указатель на массив узлов-адресов
            std::unique_ptr<MPI_Aint[]> m_pBaseNodeArrAddresses; // указатель
                        на массив начальных адресов массивов узлов-адресов
            std::shared_ptr<spdlog::logger> m_logger;
        };
} // ref_counting
namespace rma_stack::ref_counting {
    namespace custom_mpi = custom_mpi_extensions;
    void InnerStack::push(const std::function<void(GlobalAddress)>
&putDataCallback, const std::function<void()> &backoffCallback) {
        m_logger->trace("started 'push'");
        auto nodeAddress = acquireNode(m_centralized ? HEAD_RANK : m_rank);
        if (isGlobalAddressDummy(nodeAddress)) {
            m_logger->trace("failed to find free node in 'push'");
            return;
        }
```

```
{
    const auto r = nodeAddress.rank;
    const auto o = nodeAddress.offset;
    m_logger->trace("acquired free node (rank - {}, offset - {}) in
                                                    'push'", r, o);
}
putDataCallback(nodeAddress);
m_logger->trace("put data in 'push'");
CountedNodePtr resHeadCountedNodePtr;
// атомарное чтение вершины стека
MPI_Win_lock(MPI_LOCK_SHARED, HEAD_RANK, 0, m_headWin);
MPI_Fetch_and_op(nullptr,
                 &resHeadCountedNodePtr,
                 MPI_UINT64_T,
                 HEAD_RANK,
                 m_headAddress,
                 MPI_NO_OP,
                 m_headWin);
MPI_Win_flush(HEAD_RANK, m_headWin);
MPI_Win_unlock(HEAD_RANK, m_headWin);

m_logger->trace("fetched head (rank - {}, offset - {})",
    resHeadCountedNodePtr.getRank(),
    resHeadCountedNodePtr.getOffset());

m_logger->trace("started new head pushing in 'push'");

CountedNodePtr newCountedNodePtr;
newCountedNodePtr.setRank(nodeAddress.rank);
newCountedNodePtr.setOffset(nodeAddress.offset);
newCountedNodePtr.incExternalCounter();

CountedNodePtr oldHeadCountedNodePtr;
CountedNodePtr countedNodePtrNext;

constexpr auto nodeSize = sizeof(Node);
const auto countedNodePtrNextDisplacement =
          static_cast<MPI_Aint>(nodeSize * nodeAddress.offset) + 8;
const MPI_Aint countedNodePtrNextOffset =
          MPI_Aint_add(m_pBaseNodeArrAddresses[nodeAddress.rank],
                                  countedNodePtrNextDisplacement);
do {
    countedNodePtrNext = resHeadCountedNodePtr;

    // текущий адрес головы записывается в адрес нового узла, чтобы
    //новый узел мог стать новой головой (вершиной стека
    MPI_Win_lock(MPI_LOCK_SHARED, nodeAddress.rank, 0, m_nodesWin);
    MPI_Put(&countedNodePtrNext,
            1,
            MPI_UINT64_T,
            nodeAddress.rank,
            countedNodePtrNextOffset,
            1,
            MPI_UINT64_T,
            m_nodesWin);
    MPI_Win_flush(nodeAddress.rank, m_nodesWin);
    MPI_Win_unlock(nodeAddress.rank, m_nodesWin);

    oldHeadCountedNodePtr = resHeadCountedNodePtr;
    // замена вершины стека на новую
    MPI_Win_lock(MPI_LOCK_SHARED, HEAD_RANK, 0, m_headWin);
    MPI_Compare_and_swap(&newCountedNodePtr,
```

```
                                &oldHeadCountedNodePtr,
                                &resHeadCountedNodePtr,
                                MPI_UINT64_T,
                                HEAD_RANK,
                                m_headAddress,
                                m_headWin);
            MPI_Win_flush(HEAD_RANK, m_headWin);
            MPI_Win_unlock(HEAD_RANK, m_headWin);

            if (resHeadCountedNodePtr != oldHeadCountedNodePtr){
                m_logger->trace("started to execute backoff callback");
                backoffCallback();
                m_logger->trace("executed backoff callback");
            }
        }
        while (resHeadCountedNodePtr != oldHeadCountedNodePtr);
        m_logger->trace("finished 'push'");
    }
    GlobalAddress InnerStack::acquireNode(int rank) const {
        m_logger->trace("started 'acquireNode'");
        GlobalAddress nodeGlobalAddress = {0, DummyRank, 0};

        if (!isValidRank(rank)) {
            m_logger->trace("finished 'acquireNode'");
            return nodeGlobalAddress;
        }
        uint32_t newAcquiredField{1};
        uint32_t oldAcquiredField{0};
        uint32_t resAcquiredField{0};

        for (MPI_Aint i = 0; i < static_cast<MPI_Aint>(m_elemsUpLimit); ++i)
        {
            constexpr auto nodeSize = static_cast<MPI_Aint>(sizeof(Node));
            const auto nodeDisplacement = i * nodeSize;
            const MPI_Aint nodeOffset =
                  MPI_Aint_add(m_pBaseNodeArrAddresses[rank],
                               nodeDisplacement);

            MPI_Win_lock(MPI_LOCK_SHARED, rank, 0, m_nodesWin);
            MPI_Compare_and_swap(&newAcquiredField,
                                 &oldAcquiredField,
                                 &resAcquiredField,
                                 MPI_UINT32_T,
                                 rank,
                                 nodeOffset,
                                 m_nodesWin);
            MPI_Win_flush(rank, m_nodesWin);
            MPI_Win_unlock(rank, m_nodesWin);

            m_logger->trace("resAcquiredField = {} of (rank - {}, offset -
                  {}) in 'acquireNode'", resAcquiredField, rank, i);
            assert(resAcquiredField == 0 || resAcquiredField == 1);
            if (!resAcquiredField) {
                nodeGlobalAddress.rank = rank;
                nodeGlobalAddress.offset = i;
                break;
            }
        }
        m_logger->trace("finished 'acquireNode'");
        return nodeGlobalAddress;
    }
    void InnerStack::releaseNode(GlobalAddress nodeAddress) const {
```

```
        constexpr auto nodeSize = sizeof(Node);
        const auto nodeDisplacement =
                  static_cast<MPI_Aint>(nodeAddress.offset * nodeSize);
        const MPI_Aint nodeOffset =
                  MPI_Aint_add(m_pBaseNodeArrAddresses[nodeAddress.rank],
                                                 nodeDisplacement);
        int32_t acquiredField{1};
        {
            const auto r = nodeAddress.rank;
            const auto o = nodeAddress.offset;
            m_logger->trace("started to release node (rank - {}, offset -
                                                          {})", r, o);
        }
            // пометка адреса освобождённым
        MPI_Win_lock(MPI_LOCK_SHARED, nodeAddress.rank, 0, m_nodesWin);
        MPI_Fetch_and_op(&acquiredField,
                         &acquiredField,
                         MPI_UINT32_T,
                         nodeAddress.rank,
                         nodeOffset,
                         MPI_REPLACE,
                         m_nodesWin);
        MPI_Win_flush(nodeAddress.rank, m_nodesWin);
        MPI_Win_unlock(nodeAddress.rank, m_nodesWin);
        {
            const auto r = nodeAddress.rank;
            const auto o = nodeAddress.offset;
            m_logger->trace("released node (rank - {}, offset - {})", r, o);
        }
    }
    void InnerStack::pop(const std::function<void(GlobalAddress)>
&getDataCallback, const std::function<void()> &backoffCallback) {
        m_logger->trace("started 'pop'");
        CountedNodePtr oldHeadCountedNodePtr;
        // атомарное чтение текущей головы стека
        MPI_Win_lock(MPI_LOCK_SHARED, HEAD_RANK, 0, m_headWin);
        MPI_Fetch_and_op(nullptr,
                         &oldHeadCountedNodePtr,
                         MPI_UINT64_T,
                         HEAD_RANK,
                         m_headAddress,
                         MPI_NO_OP,
                         m_headWin);
        MPI_Win_flush(HEAD_RANK, m_headWin);
        MPI_Win_unlock(HEAD_RANK, m_headWin);
        {
            const auto r = oldHeadCountedNodePtr.getRank();
            const auto o = oldHeadCountedNodePtr.getOffset();
            const auto e = oldHeadCountedNodePtr.getExternalCounter();
            m_logger->trace("fetched head (rank - {}, offset - {}, ext_cnt -
                                       {}) before loop in 'pop'", r, o, e);
        }
        for (;;) {
            // увеличение внешнего счётчика ссылок на вершину стека на 1,
            // чтобы остальные потоки не удалили узел, пока он читается
            increaseHeadCount(oldHeadCountedNodePtr);
            {
                const auto r = oldHeadCountedNodePtr.getRank();
                const auto o = oldHeadCountedNodePtr.getOffset();
                const auto e = oldHeadCountedNodePtr.getExternalCounter();
                m_logger->trace("head (rank - {}, offset - {}, ext_cnt - {}))
after increaseHeadCount in 'pop'", r, o, e);
```

```
            }
            GlobalAddress nodeAddress = {
                    oldHeadCountedNodePtr.getOffset(),
                    oldHeadCountedNodePtr.getRank(),
                    0
            };
            if (isGlobalAddressDummy(nodeAddress)) {
                  // стек пуст оповещение внешнего стека и завершение pop
                getDataCallback(nodeAddress);
                break;
            }
            Node node;
            constexpr auto nodeSize = sizeof(Node);
            const auto nodeDisplacement =
static_cast<MPI_Aint>(nodeAddress.offset * nodeSize);
            const MPI_Aint nodeOffset =
                 MPI_Aint_add(m_pBaseNodeArrAddresses[nodeAddress.rank],
                                                              nodeDisplacement);
            // получение узла, соответствующего вершине списка, чтобы извлечь
            // из него счётчики для их обновления
            MPI_Win_lock(MPI_LOCK_SHARED, nodeAddress.rank, 0, m_nodesWin);
            MPI_Get(&node,
                    nodeSize,
                    MPI_UNSIGNED_CHAR,
                    nodeAddress.rank,
                    nodeOffset,
                    nodeSize,
                    MPI_UNSIGNED_CHAR,
                    m_nodesWin);
            MPI_Win_flush(nodeAddress.rank, m_nodesWin);
            MPI_Win_unlock(nodeAddress.rank, m_nodesWin);
            auto& countedNodePtrNext = node.getCountedNodePtr();
            {
                const auto r = countedNodePtrNext.getRank();
                const auto o = countedNodePtrNext.getOffset();
                const auto e = countedNodePtrNext.getExternalCounter();
                m_logger->trace("ptr->next (rank - {}, offset - {}, ext_cnt -
{})) in 'pop'", r, o, e);
            }
            CountedNodePtr resHeadCountedNodePtr;
            // удаление текущей вершины стека (головы списка)
            MPI_Win_lock(MPI_LOCK_SHARED, HEAD_RANK, 0, m_headWin);
            MPI_Compare_and_swap(&countedNodePtrNext,
                                 &oldHeadCountedNodePtr,
                                 &resHeadCountedNodePtr,
                                 MPI_UINT64_T,
                                 HEAD_RANK,
                                 m_headAddress,
                                 m_headWin
            );
            MPI_Win_flush(HEAD_RANK, m_headWin);
            MPI_Win_unlock(HEAD_RANK, m_headWin);
            if (resHeadCountedNodePtr == oldHeadCountedNodePtr) {
                  // удаление удалось – внешний стек извлекает данные
                getDataCallback(nodeAddress);
                const auto internalCounterRank =
                             static_cast<int>(nodeAddress.rank);
                const auto internalCounterDisplacement =
                static_cast<MPI_Aint>(nodeAddress.offset * sizeof(Node) +
                             sizeof(int32_t));
                const auto internalCounterOffset =
```

```
                        MPI_Aint_add(m_pBaseNodeArrAddresses[nodeAddress.rank]
                  , internalCounterDisplacement);
                const auto externalCount =
            static_cast<int32_t>(oldHeadCountedNodePtr.getExternalCounter());
                const int32_t countIncrease = externalCount - 2;
                int32_t resInternalCount{0};


                  // атомарное увеличение внутреннего счётчика на значение
                  // внешнего счётчика минус 2
                  MPI_Win_lock(MPI_LOCK_SHARED, internalCounterRank, 0,
                                                          m_nodesWin);
                MPI_Fetch_and_op(
                        &countIncrease,
                        &resInternalCount,
                        MPI_INT32_T,
                        internalCounterRank,
                        internalCounterOffset,
                        MPI_SUM,
                        m_nodesWin);
                MPI_Win_flush(internalCounterRank, m_nodesWin);
                MPI_Win_unlock(internalCounterRank, m_nodesWin);

                if (resInternalCount == -countIncrease)
                    releaseNode(nodeAddress);

                break;
            }
            else {
                const auto internalCounterRank =
                             static_cast<int>(nodeAddress.rank);
                const auto internalCounterDisplacement =
static_cast<MPI_Aint>(nodeAddress.offset * sizeof(Node) + sizeof(int32_t));
                const auto internalCounterOffset =
                  MPI_Aint_add(m_pBaseNodeArrAddresses[nodeAddress.rank],
                             internalCounterDisplacement);
                const int64_t countIncrease{-1};
                int64_t resInternalCount{0};
                  // атомарное уменьшение внутреннего счётчика 1
                MPI_Win_lock(MPI_LOCK_SHARED, internalCounterRank, 0,
                                                              m_nodesWin);
                MPI_Fetch_and_op(
                        &countIncrease,
                        &resInternalCount,
                        MPI_INT32_T,
                        internalCounterRank,
                        internalCounterOffset,
                        MPI_SUM,
                        m_nodesWin);
                MPI_Win_flush(internalCounterRank, m_nodesWin);
                MPI_Win_unlock(internalCounterRank, m_nodesWin);

                if (resInternalCount == 1)
                    releaseNode(nodeAddress);
            }
            m_logger->trace("started to execute backoff callback");
            backoffCallback();
            m_logger->trace("executed backoff callback");
        }
        m_logger->trace("finished 'pop'");
    }
    void InnerStack::increaseHeadCount(CountedNodePtr &oldHeadCountedNodePtr)
```

```
    {
        CountedNodePtr newCountedNodePtr;
        CountedNodePtr resCountedNodePtr = oldHeadCountedNodePtr;
        m_logger->trace("started 'increaseHeadCount'");

        MPI_Win_lock(MPI_LOCK_SHARED, HEAD_RANK, 0, m_headWin);
        Do {
            newCountedNodePtr = oldHeadCountedNodePtr = resCountedNodePtr;
            newCountedNodePtr.incExternalCounter();
            MPI_Compare_and_swap(&newCountedNodePtr,
                                 &oldHeadCountedNodePtr,
                                 &resCountedNodePtr,
                                 MPI_UINT64_T,
                                 HEAD_RANK,
                                 m_headAddress,
                                 m_headWin);
            MPI_Win_flush(HEAD_RANK, m_headWin);
            m_logger->trace("executed CAS in 'increaseHeadCount'");
            m_logger->trace("oldCountedNodePtr is (rank - {}, offset - {},
                                                   ext_cnt - {})",
                            oldHeadCountedNodePtr.getRank(),
                            oldHeadCountedNodePtr.getOffset(),
                            oldHeadCountedNodePtr.getExternalCounter());
            m_logger->trace("resCountedNodePtr is (rank - {}, offset - {},
                                                   ext_cnt - {})",
                            resCountedNodePtr.getRank(),
                            resCountedNodePtr.getOffset(),
                            resCountedNodePtr.getExternalCounter());
        }
        while (oldHeadCountedNodePtr != resCountedNodePtr);
        MPI_Win_unlock(HEAD_RANK, m_headWin);

oldHeadCountedNodePtr.setExternalCounter(newCountedNodePtr.getExternalCounter
());

        m_logger->trace("finished 'increaseHeadCount'");
    }

    InnerStack::InnerStack(MPI_Comm comm, MPI_Info info, bool t_centralized,
                                          size_t t_elemsUpLimit,
                           std::shared_ptr<spdlog::logger> t_logger) :
    m_elemsUpLimit(t_elemsUpLimit),
    m_centralized(t_centralized),
    m_logger(std::move(t_logger)) {
        m_logger->trace("getting rank");
        {
            auto mpiStatus = MPI_Comm_rank(comm, &m_rank);
            if (mpiStatus != MPI_SUCCESS)
                throw custom_mpi::MpiException("failed to get rank",
                        __FILE__, __func__, __LINE__, mpiStatus);
        }
        m_logger->trace("got rank {}", m_rank);

        initRemoteAccessMemory(comm, info);
        MPI_Barrier(comm);
        m_logger->trace("finished InnerStack construction");
    }
    void InnerStack::release() {
        MPI_Free_mem(m_pNodesArr);
        m_pNodesArr = nullptr;
        m_logger->trace("freed up node arr RMA memory");
        MPI_Win_free(&m_nodesWin);
```

```
    m_logger->trace("freed up node win RMA memory");
    MPI_Free_mem(m_pHeadCountedNodePtr);
    m_pHeadCountedNodePtr = nullptr;
    m_logger->trace("freed up head pointer RMA memory");
    MPI_Win_free(&m_headWin);
    m_logger->trace("freed up head win RMA memory");
}

void InnerStack::initRemoteAccessMemory(MPI_Comm comm, MPI_Info info) {
    {
        auto mpiStatus = MPI_Win_create_dynamic(info, comm, &m_nodesWin);
        if (mpiStatus != MPI_SUCCESS)
            throw custom_mpi::MpiException("failed to create RMA window
                    for nodes", __FILE__, __func__, __LINE__, mpiStatus);
    }
    {
        auto mpiStatus = MPI_Win_create_dynamic(info, comm, &m_headWin);
        if (mpiStatus != MPI_SUCCESS)
            throw custom_mpi::MpiException("failed to create RMA window
              for head", __FILE__, __func__, __LINE__, mpiStatus);
    }
    if (m_centralized) {
        m_pBaseNodeArrAddresses = std::make_unique<MPI_Aint[]>(1);

        if (m_rank == HEAD_RANK) {
            m_logger->trace("started to initialize node array");
            constexpr auto nodeSize =
                          static_cast<MPI_Aint>(sizeof(Node));
            const auto nodesSize = static_cast<MPI_Aint>(nodeSize *
                                          m_elemsUpLimit);
            {
                auto mpiStatus = MPI_Alloc_mem(nodesSize, MPI_INFO_NULL,
                                               &m_pNodesArr);
                if (mpiStatus != MPI_SUCCESS)
                    throw custom_mpi::MpiException(
                            "failed to allocate RMA memory",
                            __FILE__,
                            __func__,
                            __LINE__,
                            mpiStatus);
            }

            std::fill_n(m_pNodesArr, m_elemsUpLimit, Node());
            m_logger->trace("initialized node array");
            {
                const auto winAttachSize =
                          static_cast<MPI_Aint>(m_elemsUpLimit);
                auto mpiStatus = MPI_Win_attach(m_nodesWin, m_pNodesArr,
                                        winAttachSize);
                if (mpiStatus != MPI_SUCCESS) {
                    throw custom_mpi::MpiException("failed to attach RMA
                    window", __FILE__, __func__, __LINE__, mpiStatus);
                }
            }
            m_logger->trace("attach nodes RMA window");
            MPI_Get_address(m_pNodesArr, m_pBaseNodeArrAddresses.get());
        }

        m_logger->trace("started to broadcast node array addresses");
        auto mpiStatus = MPI_Bcast(m_pBaseNodeArrAddresses.get(), 1,
                                   MPI_AINT, HEAD_RANK, comm);
        if (mpiStatus != MPI_SUCCESS)
```

```
                throw custom_mpi::MpiException("failed to broadcast node
                  array address", __FILE__, __func__ , __LINE__, mpiStatus);
            m_logger->trace("broadcasted node array addresses");
        }
        else {
            m_logger->trace("started to initialize node array");
            constexpr auto nodeSize = static_cast<MPI_Aint>(sizeof(Node));
            const auto nodesSize    = static_cast<MPI_Aint>(nodeSize *
m_elemsUpLimit);
            {
                auto mpiStatus = MPI_Alloc_mem(nodesSize, MPI_INFO_NULL,
                                               &m_pNodesArr);
                if (mpiStatus != MPI_SUCCESS)
                    throw custom_mpi::MpiException(
                            "failed to allocate RMA memory",
                            __FILE__,
                            __func__,
                            __LINE__,
                            mpiStatus);
            }

            std::fill_n(m_pNodesArr, m_elemsUpLimit, Node());
            m_logger->trace("initialized node array");
            {
                const auto winAttachSize =
                        static_cast<MPI_Aint>(m_elemsUpLimit);
                auto mpiStatus = MPI_Win_attach(m_nodesWin, m_pNodesArr,
                                                winAttachSize);
                if (mpiStatus != MPI_SUCCESS)
                    throw custom_mpi::MpiException("failed to attach RMA
                        window", __FILE__, __func__, __LINE__, mpiStatus);
            }
            m_logger->trace("started to broadcast node array addresses");
            int procNum{0};
            MPI_Comm_size(comm, &procNum);
            m_pBaseNodeArrAddresses = std::make_unique<MPI_Aint[]>(procNum);
            MPI_Get_address(m_pNodesArr, &m_pBaseNodeArrAddresses[m_rank]);

            for (int i = 0; i < procNum; ++i) {
                auto mpiStatus = MPI_Bcast(&m_pBaseNodeArrAddresses[i], 1,
                                           MPI_AINT, i, comm);
                if (mpiStatus != MPI_SUCCESS)
                    throw custom_mpi::MpiException("failed to broadcast node
                                                   array base address",
                                    __FILE__, __func__ , __LINE__, mpiStatus);
                m_logger->trace("m_pNodeArrAddresses[{}] = {}", i,
                                                   m_pBaseNodeArrAddresses[i]);
            }
            m_logger->trace("broadcasted node array addresses");
        }
        if (m_rank == HEAD_RANK) {
            m_logger->trace("started to initialize head");
            {
                auto mpiStatus = MPI_Alloc_mem(sizeof(CountedNodePtr),
                                               MPI_INFO_NULL,
                                               &m_pHeadCountedNodePtr);
                if (mpiStatus != MPI_SUCCESS)
                    throw custom_mpi::MpiException(
                            "failed to allocate RMA memory",
                            __FILE__,
                            __func__,
                            __LINE__,
```

```
                        mpiStatus);
            }
            *m_pHeadCountedNodePtr = CountedNodePtr();
            m_logger->trace("initialized head");
            {
                auto mpiStatus = MPI_Win_attach(m_headWin,
                                                m_pHeadCountedNodePtr, 1);
                if (mpiStatus != MPI_SUCCESS)
                    throw custom_mpi::MpiException("failed to attach RMA
                        window", __FILE__, __func__, __LINE__, mpiStatus);
            }
            m_logger->trace("attached nodes RMA window");
            MPI_Get_address(m_pHeadCountedNodePtr, &m_headAddress);
        }
        {
            m_logger->trace("started to broadcast head address");
            auto mpiStatus = MPI_Bcast(&m_headAddress, 1, MPI_AINT,
                                                    HEAD_RANK, comm);
            if (mpiStatus != MPI_SUCCESS)
                throw custom_mpi::MpiException("failed to broadcast head
                  address", __FILE__, __func__ , __LINE__, mpiStatus);
            m_logger->trace("broadcasted head address");
        }
    }
    size_t InnerStack::getElemsUpLimit() const {
        return m_elemsUpLimit;
    }
} // ref_counting
```