

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

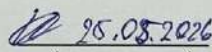
Направление подготовки: 09.03.01 «Информатика и вычислительная техника»
Профиль подготовки: «Программное обеспечение средств вычислительной техники и автоматизированных систем»

БАКАЛАВРСКАЯ РАБОТА

на тему

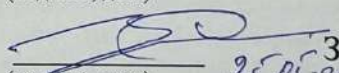
ОБУЧЕНИЕ АЛГОРИТМА РЕКОМЕНДАЦИЙ
СОЦИАЛЬНЫХ СЕТЕЙ НА
ВЫЧИСЛИТЕЛЬНОМ КЛАСТЕРЕ
ПОСРЕДСТВОМ MPI

Студент


(подпись, дата)

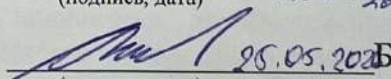
Кулахметов Сабир Исхакович
(ФИО полностью)

Руководитель


(подпись, дата)

Зинкин С.А.
(фамилия, инициалы)

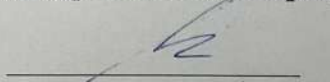
Нормоконтролёр


(подпись, дата)

Бычков А.С.
(фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 22.05.2026 г. № 14)

Заведующий кафедрой


(подпись)

М.А. Митрохин
(фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 02 июня 2026 г. № 4)

Секретарь ГЭК


(подпись)

Е.И. Гурин
(фамилия, инициалы)

Пенза 2026

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
М.А. Митрохин
« 07 » 04 20 26 г.

**ЗАДАНИЕ
НА БАКАЛАВРСКУЮ РАБОТУ**

1. Студент Кулахметов Сабир Исхакович гр. 22ВВП2 факультета вычислительной техники

2. Тема работы: Обучение алгоритма рекомендаций социальных сетей на вычислительном кластере посредством MPI

Тема утверждена приказом ПГУ № 347/о от "7" апреля 2026 г.

3. Руководитель бакалаврской работы д.т.н., профессор кафедры ВТ ПГУ Зинкин С.А.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

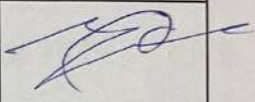
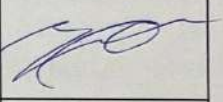
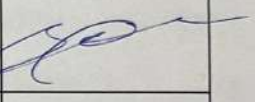
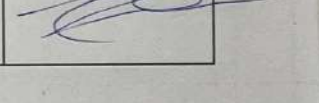
Разработать алгоритм рекомендации публикаций пользователям социальной сети, в качестве основы использующий метод машинного обучения Transformer. Поднять вычислительный кластер и обучить используемую в качестве основы алгоритма нейронную сеть, руководствуясь принципами распараллеленного обучения и используя для межпроцессной коммуникации технологию MPI. Сравнить разработанное решение с имеющимися аналогами. Язык программирования Python, среда разработки Spyder. Операционная система узлов кластера Linux Alt Workstation 10.4. Программа должна использовать следующие технологии:

1. PyTorch
2. Mpi4py
3. Numpy

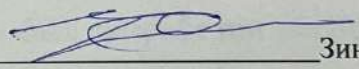
5. Перечень подлежащих разработке вопросов

Требуется произвести анализ предметной области: изучить существующие аналоги, способы проектирования и разработки программ для распределённых вычислительных систем. Изучить способы удалённого запуска процессов ЭВМ с ОС на базе ядра Linux. Выбрать язык программирования, метод машинного обучения и среду разработки.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Изучение предметной области	25	7.04.2026 – 13.04.2026	
2	Анализ требований	15	14.04.2026 – 18.04.2026	
3	Выбор технологий, применяемых при проектировании и разработке	45	19.04.2026 – 24.04.2026	
4	Разработка алгоритма	75	24.04.2026 – 4.05.2026	
5	Поднятие и тестирование работы вычислительного кластера	10	5.05.2026 – 7.05.2026	
6	Обучение алгоритма и анализ полученных результатов	40	7.05.2026 – 15.05.2026	
7	Оформление пояснительной записки	24	16.05.2026 – 21.05.2026	

Дата выдачи задания "7" апреля 2026 г.

Руководитель бакалаврской работы  Зинкин С.А.
(подпись, дата)

Задание к исполнению принял студент  07.04.2026 Кулахметов С.И.
(подпись, дата)

Работу к защите допустить
Декан факультета  Фионова Л.Р.
(подпись, дата)

23.05.2020

Реферат

Бакалаврская работа содержит 144 листа, 57 рисунков, 16 таблиц, 10 источников, 3 приложения.

ВЫЧИСЛИТЕЛЬНЫЙ КЛАСТЕР, РАСПРЕДЕЛЁННАЯ ВЫЧИСЛИТЕЛЬНАЯ СИСТЕМА, РАСПАРАЛЛЕЛЕННОЕ МАШИННОЕ ОБУЧЕНИЕ, MPI, PYTORCHDDP, DATA PARALLELISM

Цель работы – разработать алгоритм рекомендаций социальных сетей, обучение которого можно распараллелить на вычислительном кластере; произвести сравнение технологий, которые могут использоваться для выполнения данной задачи и определить возможности масштабирования распределённой вычислительной системы.

Разрабатываемый алгоритм должен проводить анализ недавно оценённых пользователем постов, находить закономерности в его интересах и предлагать теги, назначаемые на рекомендуемые публикации. Алгоритм должен реализовывать стратегию обучения Data Parallelism с возможностью обучения как на мультипроцессорной системе, так и на вычислительном кластере, состоящем из нескольких отдельных ЭВМ.

В результате выполнения работы был разработан алгоритм рекомендаций для социальных сетей, который удовлетворяет всем поставленным задачам, а именно: реализация параллельного машинного обучения и возможность развёртывания на вычислительном кластере. Были использованы технологии PyTorch и MPI, поднят вычислительный кластер, состоящий из трёх объединённых в единую вычислительную систему ЭВМ. Алгоритм обучен путём развёртывания вычислений на данном кластере.

					ПГУ1.090301.40.001 ПЗ				
Изм.	Лист	№ докум.	Подпись	Дата					
Разраб.		Кулахметов С.И.		25.08.26	Обучение алгоритма рекомендаций социальных сетей на вычислительном кластере посредством MPI Пояснительная записка	Лит.	Лист	Листов	
Провер.		Зинкин С.А.		25.08.26			2	144	
Н. Контр		Бычков А.С.		25.08.26		ФВТ, каф. ВТ гр. 22ВВП2			
Утверд.		Митрохин М.А.		25.08.26					

Содержание

Перечень сокращений и обозначений	7
Введение	8
1 Обучение алгоритма рекомендаций социальных сетей на вычислительном кластере посредством MPI	9
1.1 Анализ темы	9
1.2 Анализ аналогов	11
1.2.1 Библиотека NCCL (NVIDIA Collective Communications Library)	12
1.2.2 Библиотека GLOO	13
1.2.3 Технология OpenMP	14
1.2.4 Фреймворк PyTorch Distributed	15
1.2.5 Вывод	16
1.3 Стратегии параллельного обучения	17
1.3.1 Стратегия «Data Parallelism»	18
1.3.2 Стратегия «Model Parallelism»	19
1.3.3 Стратегия «Pipeline Parallelism»	19
1.4 Постановка задачи	20
1.5 Требования к программно-аппаратному обеспечению	21
2 Выбор инструментов разработки	23
2.1 Выбор языка программирования	23
2.1.1 Язык программирования «Python»	23
2.1.2 Язык программирования «C++»	24
2.1.3 Вывод	25
2.2 Выбор библиотеки MPI	25
2.2.1 Реализация «Open MPI»	26
2.2.2 Реализация «MPICH»	26
2.2.3 Реализация «Intel MPI»	26
2.2.4 Вывод	27
2.3 Установка и настройка среды разработки	27

2.3.1 Установка дистрибутива «Anaconda» и создание окружения	27
2.3.2 Установка пакетов и библиотек для разработки	29
2.3.3 Установка среды разработки «Spyder»	30
3 Архитектура приложения	32
3.1 Архитектура программы	32
3.1.1 Парадигма «SPMD» (Single Program Multiple Data)	32
3.1.2 Стратегия параллелизации обучения «Data Parallelism»	33
3.2 Архитектура используемой нейронной сети	34
3.2.1 Архитектура Transformer	34
3.2.2 Transformer-encoder	35
3.2.3 Transformer-decoder	38
4 Разработка программы, развёртывание и запуск распределённой вычислительной системы	40
4.1 Разработка модулей программы	40
4.1.1 Разработка модуля обработки тегов	41
4.1.2 Разработка модуля обработки датасета	41
4.1.3 Разработка модуля, реализующего работу с моделью Transformer	42
4.1.4 Реализация распределённого обучения с применением MPI	43
4.1.5 Разработка модуля генерации рекомендаций	45
4.2 Модификация основной программы для использования технологии PyTorch DDP	47
4.2.1 Изменения в инициализации распределённого окружения	47
4.2.2 Применение DistributedDataParallel	48
4.2.3 Упрощение цикла обучения	48
4.2.4 Сравнение этапов выполнения модифицированной программы с исходной версией	48
4.3 Поднятие кластера и запуск распараллеленного обучения	49
4.3.1 Поднятие кластера на базе 3-х ЭВМ с операционной системой Linux Alt Workstation 10.4	49

4.3.2	Запуск распараллеленного обучения на вычислительном кластере	58
4.3.3	Тестирование работы обученного алгоритма	62
5	Методология определения масштабирования системы	65
5.1	Определение времени вычислений при изменении числа процессов и объёма данных	66
5.1.1	Определение времени обработки одного примера	66
5.1.2	Определение общего времени вычислений	69
5.1.3	Определение времени вычислений при масштабировании системы ..	70
5.2	Расчёт процента коллективных операций MPI при изменении числа процессов	72
5.2.1	Определение процента операций синхронизации MPI от общего времени обучения алгоритма	72
5.2.2	Определение процента операций синхронизации MPI от времени обучения алгоритма при изменении числа процессов	73
5.3	Определение предела масштабирования	75
5.4	Сравнение результатов работы реализованного вручную подхода с результатами при использовании технологии PyTorch DDP	77
6	Безопасность жизнедеятельности	81
6.1	Техника безопасности при работе с персональным компьютером	81
6.1.1	Организация рабочего места	81
6.1.2	Электробезопасность и размещение техники	82
6.2	Требования к освещению	82
6.3	Режим отдыха и профилактика переутомления	83
7	Стартап проект	85
7.1	Сегменты целевой аудитории	86
7.2	Ключевые процессы	86
7.3	Каналы взаимодействия	87
7.4	Источники дохода	87
7.5	Ключевые ресурсы	88

7.6 Риски при реализации проекта	89
Список использованных источников	92
Приложение А – UML-диаграммы	93
Приложение Б – Листинг программы	98
Приложение В – Презентация	135

Перечень сокращений и обозначений

В данной работе применяются следующие сокращения и обозначения

ОС – операционная система

ЦОД – центр обработки данных

ЭВМ – электронная вычислительная машина

CPU – (Central Processor Unit) центральный процессор

DDP – (Distributed Data Parallelism) метод рапараллеливания вычислений путём
разделения данных между процессами

GLOO – (Global Operations) кроссплатформенная коммуникационная
библиотека

GPU – (Graphical Processor Unit) графический процессор

MPI – (Message Passing Interface) интерфейс передачи сообщений,
используемый для написания параллельных программ

NCCL – (NVIDIA Collective Communications Library) библиотека для
графических ускорителей Nvidia, которая представляет собой
высокооптимизированную реализацию коллективных операций

UML – (Unified Modeling Language) – унифицированный язык моделирования

Введение

Обучение алгоритма рекомендаций социальных сетей на вычислительном кластере с применением технологии MPI представляет собой нестандартное решение в области машинного обучения, так как для вычислений используется множество потоков на CPU, работающих на нескольких ЭВМ в сети. В отличие от традиционных решений, для организации подобной вычислительной системы не требуется дорогого оборудования, обладающего не всем доступными GPU и большим количеством как оперативной, так и постоянной памяти.

Актуальность разработки обусловлена активным развитием платформ и социальных сетей от средних и малых стартапов, и отдельных энтузиастов, которым необходимо решать аналогичную задачу с ограниченным бюджетом на закупку специального оборудования, но способных организовать локальную сеть из нескольких ЭВМ.

Целью работы является: написание алгоритма рекомендаций социальных сетей, в основе которого лежит модель машинного обучения Transformer, ключевой особенностью которой является распараллеленное обучение на вычислительном кластере, организованное посредством стандарта MPI MPICH, с узлами, на которых установлена операционная система на базе ядра Linux; а также проведение сравнения с существующими технологиями и определения возможностей масштабирования используемой распределённой вычислительной системы.

					ПГУ1.090301.40.001 ПЗ	Лист
						8
Изм.	Лист	№ докум.	Подпись	Дата		

1 Обучение алгоритма рекомендаций социальных сетей на вычислительном кластере посредством MPI

1.1 Анализ темы

В наше время при пользовании социальными сетями возникает необходимость выдачи пользователю нового контента, в случае, если постов с сообществ или страниц, на которые оформлена подписка, недостаточно для заполнения ленты. Что необходимо для удержания внимания пользователя на платформе, с последующей её монетизацией с учётом выявляемых интересов. Изначальные алгоритмы рекомендаций, как правило, весьма примитивны и основаны на принципе: «Если пользователь активно оценивает контент по определённой тематике, то именно на основе данной области и организовывать рекомендательную выдачу». Такой подход приводит к «информационному вакууму», который, в свою очередь, ведёт к выдаче однообразного контента и, в последствие – к снижению заинтересованности пользователя в используемом сервисе.

Однако, в текущих реалиях, есть возможность в качестве алгоритма рекомендаций выдачи постов пользователям использовать не программный код с заданной жёсткой примитивной логикой, а нейронную сеть, которая позволяет решить фундаментальные проблемы подобных алгоритмов.

Далее приведены аргументы в пользу данного подхода:

– масштабируемость и гибкость. Алгоритм на основе нейронной сети можно обучить, предоставив ему историю взаимодействия пользователя с предложенным ранее контентом в требуемом формате. А при необходимости добавить дополнительные категории – просто переобучить модель с новыми данными, что позволит алгоритму брать в учёт не учитываемые ранее области;

– борьба с «информационным пузырьём». Нейронная сеть способна анализировать огромное количество данных о реакциях пользователя в течение большого периода на различные категории предлагаемого контента, что

позволяет составить полную картину интересов человека, необходимую для комплексного анализа предпочитаемых областей и нахождения корреляций в оцениваемых материалах для составления последующей выдачи;

– регулировка точности выдачи. В отличие от жёстко заданного программного алгоритма, при использовании нейронной сети можно настроить точность соответствия выдаваемых постов интересам анализируемого пользователя. Что может быть полезно при ситуации, когда пользователю можно предлагать контент не только по конкретной тематике, но и по смежным областям, которые могут его заинтересовать.

Однако, не смотря на явные преимущества использования нейронной сети в качестве основы для алгоритма рекомендаций постов социальных сетей, подобное решение могут позволить себе, в основном, крупные компании, владеющие соответствующим программными продуктами и с необходимым для закупки оборудования бюджетом. В связи с чем, возникает необходимость в организации решения для малых и средних стартапов и отдельных энтузиастов, которое позволит применять подобные методы в своих продуктах.

Активное развитие цифровых платформ, позволяющих людям делиться результатами своего творчества и иной деятельности, делают внедрение решения проблемы обучения алгоритма рекомендаций на ограниченных ресурсах особо актуальным.

Существует несколько способов реализации подобного решения:

– параллелизация на уровне вычислительных потоков с использованием нескольких GPU. Для его реализации требуется серверное оборудование, обладающее возможностью установки нескольких GPU, между которыми будет распределено обучение модели;

– распараллеливание обучения на уровне вычислительного кластера с применением технологии MPI. Для его реализации требуется два и более компьютеров (узлов), объединённых в единую сеть. Каждый узел может содержать один или несколько GPU. Обучение модели распределяется между

узлами с использованием MPI: каждый узел обрабатывает собственную часть данных, после чего MPI обеспечивает синхронизацию градиентов или параметров модели между всеми узлами. Это позволяет масштабировать обучение на вычислительный кластер без модификации логики самой нейронной сети.

На основе описанных подходов в рамках данной работы разрабатывается система, осуществляющая интеллектуальный подбор рекомендаций в социальных сетях. Система включает в себя следующий функционал:

- запуск обучения модели на нескольких ЭВМ, объединённых в вычислительный кластер;
- формирование рекомендаций на основе оценок предложенного пользователю ранее контента.

Ключевым компонентом системы является алгоритм рекомендаций, реализованный на основе нейронной сети. Обучение модели производится на вычислительном кластере с использованием MPI, что позволяет обрабатывать историю взаимодействий пользователей с постами и формировать персонализированную выдачу даже при ограниченных вычислительных ресурсах.

Целевыми операционными системами для построения кластера, на котором осуществляется обучение нейронной сети, являются ОС на базе ядра Linux и компонентов GNU, в частности, такие дистрибутивы как Debian (и основанные на нём) и Alt Workstation от отечественной компании «Базальт СПО».

1.2 Анализ аналогов

Для решения задачи распараллеленного обучения нейронных сетей на нескольких вычислительных узлах необходимо выбрать подходящую коммуникационную библиотеку или фреймворк. Помимо классического MPI, существует ряд специализированных решений, оптимизированных для задач

глубокого обучения в гетерогенных вычислительных средах. Далее рассмотрены такие библиотеки и технологии как:

- NCCL;
- GLOO;
- OpenMP;
- PyTorch Distributed.

Проведен анализ и выявлены слабые и сильные стороны каждой из технологий.

1.2.1 Библиотека NCCL

Библиотека NCCL разработана компанией NVIDIA и представляет собой высокооптимизированную реализацию коллективных операций (AllReduce, Broadcast, ReduceScatter и т.д.) для графических процессоров собственного производства. Она является стандартом при обучении нейронных сетей на кластерах, оснащённых графическими ускорителями компании NVIDIA. NCCL автоматически выбирает оптимальный маршрут передачи данных: через NVLink (межпроцессорное соединение GPU), PCIe или высокоскоростные сетевые интерфейсы InfiniBand и Ethernet.

Достоинства:

- максимальная производительность. Обеспечивает наивысшую пропускную способность и минимальные задержки при обмене данными между GPU, что критически важно при синхронизации градиентов больших моделей;
- глубокая интеграция с PyTorch и TensorFlow. Используется по умолчанию в качестве коммуникационного бэкенда в DistributedDataParallel (DDP) PyTorch при работе с GPU;
- поддержка современных топологий. Эффективно использует высокоскоростные межсоединения, такие как NVLink и NVSwitch, а также RDMA-сети.

Недостатки:

– аппаратная зависимость. Библиотека ориентирована исключительно на GPU производства компании NVIDIA и не может быть использована для обмена данными между CPU или графическими ускорителями от других производителей;

– сложность отладки. При возникновении проблем на уровне коммуникаций (например, тайм-ауты или ошибки синхронизации) диагностика проблемы затруднена из-за низкоуровневого характера библиотеки;

– лицензирование. Хотя NCCL распространяется на безвозмездной основе, её использование требует принятия лицензионного соглашения компании NVIDIA, что может накладывать ограничения при использовании данной технологии в некоторых средах.

1.2.2 Библиотека GLOO

GLOO – это кроссплатформенная коммуникационная библиотека с открытым исходным кодом. В отличие от рассмотренной ранее NCCL, GLOO ориентирована на аппаратную универсальность и способна работать как на центральных процессорах, так и на графических ускорителях под управлением различных операционных систем. Данная библиотека реализует базовый набор коллективных операций и часто используется в качестве резервного варианта при отсутствии NCCL или для CPU-ориентированных вычислений.

Достоинства:

– кроссплатформенность и универсальность. Поддерживает CPU и GPU (включая даже, в некоторых реализациях, ускорители от компании AMD), работает в Linux и macOS. Является стандартным бэкендом для PyTorch DDP при обучении на CPU;

– простота развёртывания. Не требует специальных драйверов или сложной настройки сетевого оборудования, что упрощает запуск распределённого обучения в гетерогенных вычислительных средах;

– открытый исходный код. Позволяет при необходимости модифицировать библиотеку и обеспечивает прозрачность внутренних механизмов.

Недостатки:

– низкая производительность на GPU. По сравнению с NCCL, пропускная способность GLOO при работе с GPU значительно ниже, особенно в высокоскоростных сетях;

– сравнительно небольшая аптимизация. Алгоритмы коллективных операций не так тщательно оптимизированы под специфическое аппаратное обеспечение, как в проприетарных решениях;

– ограниченная распространённость вне PyTorch. В экосистемах других фреймворков (например, TensorFlow) используется реже или почти не используется.

1.2.3 Технология OpenMP

OpenMP (Open Multi-Processing) – это не распределённая, а многопоточная технология параллельного программирования для систем с общей памятью (один сервер, одна рабочая станция). В отличие от MPI, NCCL или GLOO, которые предназначены для обмена сообщениями между узлами не зависимо от того гомогенная или гетерогенная вычислительная среда, OpenMP позволяет эффективно задействовать все ядра одного многоядерного процессора. Параллелизм достигается путём добавления директив компилятора в код на языках C, C++ или Fortran, что автоматизирует распределение работы цикла или задачи между потоками.

Достоинства:

– чрезвычайно высокая простота использования. Для распараллеливания вычислительного цикла достаточно добавить одну строку директивы, не изменяя логику программы. Идеально подходит для ускорения предобработки данных или выполнения операций на CPU;

– эффективное использование многоядерных CPU. Позволяет линейно масштабировать производительность на задачах, интенсивно использующих один процессор;

– отсутствие накладных расходов на сетевой обмен. Работа в рамках общей памяти исключает задержки и проблемы, связанные с сетевым взаимодействием.

Недостатки:

– сложность масштабирования на несколько узлов. Как правило, не подходит для распределённых вычислений на кластере. При использовании утилиты `mpirun` часто возникают проблемы с сетевым взаимодействием между узлами. Изначально заточена под работу на одном CPU. Максимальная производительность ограничена ресурсами одного сервера;

– потенциальные проблемы с конкуренцией за память. При интенсивной работе с данными производительность может упирается в пропускную способность оперативной памяти, что приводит к снижению эффективности при большом числе потоков;

– ограниченная поддержка в Python. Не смотря на существование библиотеки `mpi4py`, для использования OpenMP в экосистеме Python часто приходится прибегать к написанию модулей-расширений на C/C++.

1.2.4 Фреймворк PyTorch Distributed

PyTorch Distributed – это не просто коммуникационная библиотека как NCCL или GLOO, а встроенный в фреймворк PyTorch высокоуровневый модуль, предназначенный для распределённого обучения нейросетей. Его ключевым компонентом является `DistributedDataParallel (DDP)`. DDP абстрагирует низкоуровневые детали коммуникации, позволяя разработчику с минимальными изменениями кода запустить обучение на нескольких GPU или CPU как в пределах одного узла, так и на вычислительном кластере. В качестве бэкенда DDP может использовать NCCL (для GPU), GLOO (для CPU) или MPI.

Достоинства:

					ПГУ1.090301.40.001 ПЗ	Лист
						15
Изм.	Лист	№ докум.	Подпись	Дата		

- бесшовная интеграция с PyTorch. DDP является рекомендуемым и наиболее поддерживаемым способом распределённого обучения в экосистеме PyTorch. Код, написанный для одной GPU, может быть легко адаптирован для работы на множестве устройств;

- высокая эффективность. В DDP реализован механизм асинхронной коммуникации (перекрытие вычислений обратного распространения ошибки и передачи градиентов), что минимизирует простои ускорителей;

- гибкость и масштабируемость. Поддерживает обучение на огромном числе GPU и CPU с использованием различных стратегий синхронизации и предоставляет инструменты для запуска.

Недостатки:

- привязка к экосистеме PyTorch. Фреймворк не может быть напрямую использован для обучения моделей, написанных на TensorFlow или JAX;

- необходимость настройки окружения. Для запуска на нескольких узлах требуется дополнительная настройка переменных окружения (MASTER_ADDR, MASTER_PORT, WORLD_SIZE, RANK) или использование планировщиков задач (SLURM, Kubernetes);

- относительная сложность начальной настройки. По сравнению с однопоточным кодом, DDP требует явного управления распределённым сэмплером данных и синхронизацией состояния модели.

1.2.5 Вывод

Таблица 1 – Достоинства и недостатки рассмотренных технологий

Возможности технологий	NCCL	GLOO	OpenMP	PyTorch Distributed
Возможность производить вычисления на нескольких ЭВМ в сети	+	+	-	+
Поддержка вычислений на CPU	-	+	+	+
Поддержка вычислений на GPU	+	+	-	+
Кроссплатформенность	-	-	+	+

Возможности технологий	NCCL	GLOO	OpenMP	PyTorch Distributed
Возможность интеграции в существующий код	-	-	+	+
Применимость в других сферах вычислений	-	-	+	-

Также стоит отметить, что производительность и масштабируемость различных технологий не одинаковы. Например, NCCL обеспечивает максимальную пропускную способность на кластерах с GPU от компании NVIDIA, но бесполезна при обучении на CPU. В свою очередь, PyTorch Distributed, выступая в роли высокоуровневой надстройки, позволяет абстрагироваться от низкоуровневых деталей коммуникации, но при этом жёстко привязан к конкретному фреймворку глубокого обучения.

Важно понимать, что выбор той или иной библиотеки или технологии параллельных вычислений не является универсальным. Он должен определяться конкретными задачами: необходимостью масштабирования за пределы одного узла, типом доступного аппаратного обеспечения (CPU или GPU), а также требованиями к простоте внедрения и отладки. Таким образом, разработчик вправе выбирать наиболее подходящий инструмент, балансируя между производительностью, гибкостью и сложностью реализации распределённой системы.

1.3 Стратегии параллельного обучения

Для решения задачи эффективного распараллеленного обучения нейронных сетей, помимо выбора конкретной коммуникационной библиотеки, принципиальное значение имеет стратегия параллелизации — способ разделения вычислений и данных между узлами. Существуют 3 основные стратегии параллельного обучения — Data Parallelism (параллелизм данных), Model Parallelism (модельный параллелизм) и Pipeline Parallelism (конвейерный

параллелизм) — покрывают большинство современных нужд. Далее проведен их анализ и выявлены слабые и сильные стороны каждого из подходов.

1.3.1 Стратегия «Data Parallelism»

Стратегия Data Parallelism является наиболее распространённой стратегией распределённого машинного обучения. Её суть заключается в том, что каждый узел в распределённой вычислительной системе хранит полную копию модели, но обрабатывает собственное непересекающееся подмножество обучающих примеров. После вычисления локальных градиентов выполняется коллективная операция синхронизации, усредняющая градиенты всех устройств, и каждый узел обновляет свою копию весов. Таким образом, на каждой итерации все копии модели остаются идентичными.

Достоинства:

- простота реализации и отладки. Архитектура модели не требует изменений, распараллеливание может достигаться, как ручным разбиением на процессы, так и за счёт обёрток типа DistributedDataParallel, автоматизирующих синхронизацию градиентов;

- масштабируемость. Пока модель целиком помещается в память одного ускорителя, добавление устройств обеспечивает почти линейный прирост производительности;

- универсальность. Подходит для подавляющего большинства современных архитектур (например Transformer, CNN, ResNet), легко реализуема и поддерживается всеми основными фреймворками.

Недостатки:

- избыточное потребление памяти. Каждый узел хранит полную копию модели, оптимизатора и промежуточных активаций, что делает стратегию неприменимой для очень больших моделей;

- чувствительность к пропускной способности сети. Синхронизация градиентов может стать узким местом при медленных межсоединениях в вычислительной среде, ограничивая масштабируемость.

1.3.2 Стратегия «Model Parallelism»

Стратегия Model Parallelism применяется, когда модель не помещается в память одного устройства. В этой стратегии различные части модели (слои, блоки и т.д.) размещаются на разных узлах, а входные данные последовательно продвигаются через эти элементы. Промежуточные активации передаются по сети между устройствами. Более тонкая разновидность — Tensor Parallelism — разделяет отдельные матричные операции (например, умножения в self-attention) между устройствами, что позволяет равномерно распределить нагрузку на узлы.

Достоинства:

- возможность обучать модели очень больших размеров. Единственный способ работать с моделями, чей объём параметров превышает память одного GPU или CPU;
- гибкое распределение ресурсов. Можно размещать очень большие слои на более мощных устройствах, балансируя вычислительную нагрузку;
- более низкие требования к пропускной способности сети по сравнению с частой синхронизацией градиентов в Data Parallelism.

Недостатки:

- большая сложность реализации. Требуется вручную проектировать схему разделения модели и управлять передачей тензоров между устройствами;
- последовательная зависимость и простои. Пока первое устройство обрабатывает свой фрагмент, остальные простаивают;
- рост коммуникационных задержек. При большом числе разделённых слоёв затраты на передачу активаций могут превысить выгоду от данного метода параллелизма.

1.3.3 Стратегия «Pipeline Parallelism»

Стратегия Pipeline Parallelism представляет собой комбинацию идей Data и Model Parallelism, стремясь уменьшить простои устройств, характерные для модельного параллелизма. Модель по-прежнему делится на группы слоёв,

размещаемые на разных устройствах, но обучение организуется конвейерно: входной мини-батч разбивается на микро-батчи, которые последовательно подаются в конвейер. Пока вторая стадия обрабатывает первый микро-батч, первая уже приступает ко второму. Это позволяет существенно повысить загрузку оборудования.

Достоинства:

- высокая степень использования устройств. За счёт перекрытия вычислений и передачи данных простои сокращаются до минимума, особенно при большом количестве микро-батчей;

- применимость к огромным моделям. Позволяет обучать модели, которые не вмещаются в память одного узла, с эффективностью, близкой к стратегии Data Parallelism;

- гибкость размера фрагментов. Можно адаптировать число стадий и микро-батчей под характеристики аппаратного обеспечения.

Недостатки:

- усложнение отладки и настройки. Требуется тщательно балансировать стадии по вычислительной сложности, чтобы избежать перекосов загруженности оборудования;

- наличие «пузырей» (bubble overhead). Даже в идеальном конвейере часть времени устройства неизбежно простаивают, что снижает общую эффективность;

- дополнительная потребность в памяти. Необходимость хранить активации для нескольких микро-батчей одновременно увеличивает пиковое потребление памяти на устройстве.

1.4 Постановка задачи

Необходимо разработать алгоритм рекомендаций для социальных сетей, в основе которого будет лежать использование тегов, назначаемых на публикации с соответствующей тематикой. В качестве модели для данной

					<i>ПГУ1.090301.40.001 ПЗ</i>	Лист
						20
Изм.	Лист	№ докум.	Подпись	Дата		

нейросети необходимо использовать Transformer, так как она позволяет обрабатывать весь набор данных (история оценённых пользователем постов) и искать в нём корреляции, чтобы затем сгенерировать теги, которые будут назначены посту, максимально соответствующему области интересов человека.

Обучение нейронной сети, необходимо производить на вычислительном кластере, обмен информацией между узлами которого организован при помощи технологии MPI, обеспечивающей синхронизацию градиентов или параметров модели между всеми узлами.

Основная цель проекта – организовать обучения алгоритма рекомендаций социальных сетей на вычислительном кластере посредством MPI и продемонстрировать его работоспособность на реальной вычислительной системе.

1.5 Требования к программно-аппаратному обеспечению

В таблице 2 указаны минимальная и рекомендуемая конфигурации аппаратного и программного обеспечения, необходимые и достаточные для корректного функционирования вычислительной системы.

Таблица 2 – Минимальная и рекомендуемая конфигурации аппаратного и программного обеспечения

Параметры оборудования	Минимальная конфигурация	Рекомендуемая конфигурация
Процессор	Intel Core I3-1115G4 (2 ядра, 4.1 ГГц)	Intel Core I5-1135G7 (4 ядра, 4.2 ГГц)
Объём оперативной памяти	4 Гб	8 Гб
Объём постоянной памяти	35 Гб	128 Гб
Скорость сетевого интерфейса	100 Мбит/с	1 Гбит/с
Количество ЭВМ, объединённых в сеть	2	8

Параметры оборудования	Минимальная конфигурация	Рекомендуемая конфигурация
Операционная система	Linux Debian 11	Linux Debian 11, Linux Ubuntu 20.04, Red Hat Enterprise Linux 9, Alt Workstation 10.4

2 Выбор инструментов разработки

Для того чтобы разработать качественную систему рекомендаций, способную эффективно обрабатывать запросы и обучаться на больших объёмах данных, следует выбрать правильные инструменты на всех этапах цикла разработки. Это помогает избежать проблем с производительностью, масштабируемостью и совместимостью компонентов как на этапе разработки, так и при последующей эксплуатации системы. Грамотно подобранный стек технологий позволяет сократить временные затраты на реализацию прототипа, оптимизировать использование вычислительных ресурсов и обеспечить возможность дальнейшего расширения функционала без кардинальной переработки архитектуры.

2.1 Выбор языка программирования

Для решения задачи построения распределённой рекомендательной системы на основе нейронных сетей критически важно выбрать правильный язык программирования. От этого выбора зависит не только скорость разработки и доступность необходимых библиотек, но и эффективность выполнения ресурсоёмких вычислений, а также удобство организации параллельной обработки данных на вычислительном кластере. Наиболее распространёнными языками в области высокопроизводительных параллельных вычислений являются:

- Python;
- C++.

Чтобы определить, какой из них в большей степени соответствует поставленным целям, проведём анализ и выявим особенности каждого.

2.1.1 Язык программирования «Python»

Python де-факто является стандартом в области анализа данных и машинного обучения. Подавляющее большинство современных фреймворков глубокого обучения (PyTorch, TensorFlow, JAX) предоставляют в первую

очередь Python-интерфейс, что позволяет быстро прототипировать сложные архитектуры нейронных сетей с минимальными временными затратами на разработку. Кроме того, Python обладает богатейшей экосистемой библиотек для предобработки данных, их визуализации и научных вычислений. Для организации параллельных и распределённых вычислений в Python существуют высокоуровневые обёртки над MPI, а также встроенные средства фреймворков, что значительно упрощает масштабирование обучения на несколько узлов по сравнению с низкоуровневой реализацией на C++.

Однако стоит отметить, что сам Python является интерпретируемым языком, и его выполнение «в чистом виде» может быть медленным. В контексте машинного обучения этот недостаток нивелируется тем, что все вычислительно ёмкие операции (матричные умножения, свёртки и т.д.) выполняются внутри библиотек, написанных на C/C++ и CUDA, а Python выступает лишь в роли удобного управляющего интерфейса.

2.1.2 Язык программирования «C++»

Язык C++ широко применяется в высокопроизводительных вычислениях благодаря возможности тонкого контроля над памятью и минимальным накладным расходам на выполнение кода. Все ключевые библиотеки глубокого обучения, включая PyTorch и TensorFlow, имеют низкоуровневое ядро, написанное на C++ и CUDA C. Это позволяет добиться максимальной эффективности при вычислениях на GPU и процессорах. Кроме того, классические реализации MPI изначально ориентированы на C/C++, что даёт разработчику полный контроль над коммуникацией между узлами и позволяет оптимизировать передачу данных вплоть до отдельных байтов.

Тем не менее, разработка полноценной нейросетевой модели непосредственно на C++ сопряжена со значительными трудностями. Отсутствие гибкости, свойственной Python, замедляет процесс экспериментирования с архитектурой и гиперпараметрами. Необходимость ручного управления памятью и более сложный синтаксис увеличивают время

разработки и повышают риск возникновения трудноуловимых ошибок. Интеграция с современными инструментами экосистемы машинного обучения, которая реализована в первую очередь для Python.

2.1.3 Вывод

Проанализировав возможности обоих языков программирования, было принято решение выбрать Python в качестве основного языка для разработки системы распределённого обучения рекомендательной модели. Такой выбор обусловлен тем, что Python предоставляет наиболее богатую экосистему для машинного обучения, позволяет быстро реализовать прототип выбранной модели Transformer и легко масштабировать его на кластер с использованием MPI. При этом критически важная производительность вычислений обеспечивается за счёт вызовов низкоуровневых C++/CUDA-ядер фреймворка PyTorch. В итоге использование Python позволяет сконцентрироваться на алгоритмической составляющей задачи и логике параллельных вычислений, избегая избыточных затрат времени на низкоуровневую реализацию, которая была бы неизбежна при разработке на C++.

2.2 Выбор библиотеки MPI

После определения языка программирования и общего подхода к распараллеливанию необходимо выбрать конкретную реализацию интерфейса передачи сообщений (MPI) между узлами вычислительного кластера. От этого выбора зависят стабильность работы распределённой системы, эффективность использования сетевых ресурсов и сложность настройки вычислительного кластера. Наиболее распространёнными реализациями MPI в научной и инженерной среде являются:

- Open MPI;
- MPICH;
- Intel MPI.

Рассмотрим особенности каждой из них применительно к задаче обучения нейросетевых моделей в гетерогенной вычислительной среде.

2.2.1 Реализация «Open MPI»

Open MPI представляет собой современную, активно развиваемую реализацию стандарта MPI, объединившую наработки нескольких более ранних проектов (FT-MPI, LA-MPI, LAM/MPI). Она отличается модульной архитектурой, позволяющей гибко подключать различные транспортные уровни (BTL — Byte Transfer Layer) для обмена данными: общую память (shared memory), TCP/IP, InfiniBand, RoCE и др. Благодаря широкой поддержке высокоскоростных сетей и оптимизациям под многопроцессорные системы Open MPI часто используется в крупных вычислительных центрах и на суперкомпьютерах.

Тем не менее, богатство возможностей Open MPI имеет и обратную сторону. Для стабильной работы в сетях на базе простого TCP/IP, особенно в виртуализированных средах, часто требуется тонкая ручная настройка параметров BTL и исключение неподдерживаемых интерфейсов. Кроме того, диагностика коммуникационных сбоев в Open MPI может быть затруднена из-за сложной внутренней структуры и большого количества неявных параметров.

2.2.2 Реализация «MPICH»

MPICH является одной из старейших и наиболее широко портируемых реализаций MPI. Её главными достоинствами считаются высокая надёжность и предсказуемость работы, особенно в стандартных TCP/IP-сетях. MPICH имеет более простую и прозрачную архитектуру по сравнению с Open MPI, что упрощает развёртывание и отладку распределённых приложений. Многие коммерческие варианты MPI (Intel MPI, Microsoft MPI) основаны именно на кодовой базе MPICH.

2.2.3 Реализация «Intel MPI»

Intel MPI — это коммерческая реализация, основанная на MPICH и оптимизированная для процессоров Intel и высокоскоростных сетей (InfiniBand,

Omni-Path). Она обеспечивает наивысшую производительность на центральных процессорах от компании Intel, но требует приобретения лицензии и избыточна для задач, не требующих экстремально низких задержек.

2.2.4 Вывод

Проанализировав особенности доступных реализаций MPI, в рамках данной работы было принято решение использовать MPICH. Такой выбор обусловлен несколькими факторами. Во-первых, MPICH продемонстрировал стабильную работу в тестовой среде, состоящей из виртуальных машин VirtualBox, соединённых стандартной сетью TCP/IP, в то время как настройка Open MPI в аналогичных условиях потребовала бы дополнительных усилий и ручной коррекции параметров BTL. Во-вторых, MPICH предоставляет достаточную для целей обучения модели пропускную способность, полностью покрывая потребности в синхронизации градиентов нейронной сети. В-третьих, открытая лицензия и широкая распространённость MPICH гарантируют лёгкость воспроизведения результатов на других вычислительных установках. Таким образом, выбор MPICH позволяет сосредоточиться на решении прикладной задачи, минимизируя затраты времени на администрирование и отладку распределённой вычислительной среды.

2.3 Установка и настройка среды разработки

2.3.1 Установка дистрибутива «Anaconda» и создание окружения

Anaconda представляет собой дистрибутив и систему управления пакетами для языка программирования Python, включающую в себя набор библиотек, используемых для обработки данных и машинного обучения. Помимо управления пакетами Anaconda используется для развёртывания окружений, используемых для тонкой конфигурации пакетов и библиотек в конкретном проекте.

Прежде чем установить Anaconda, необходимо скачать установочный скрипт из репозитория при помощи команды «wget» (см. рисунок 1). Затем в

интерпретаторе «bash» нужно выпустить выполнение скачанного на предыдущем шаге скрипта (см. рисунок 2).

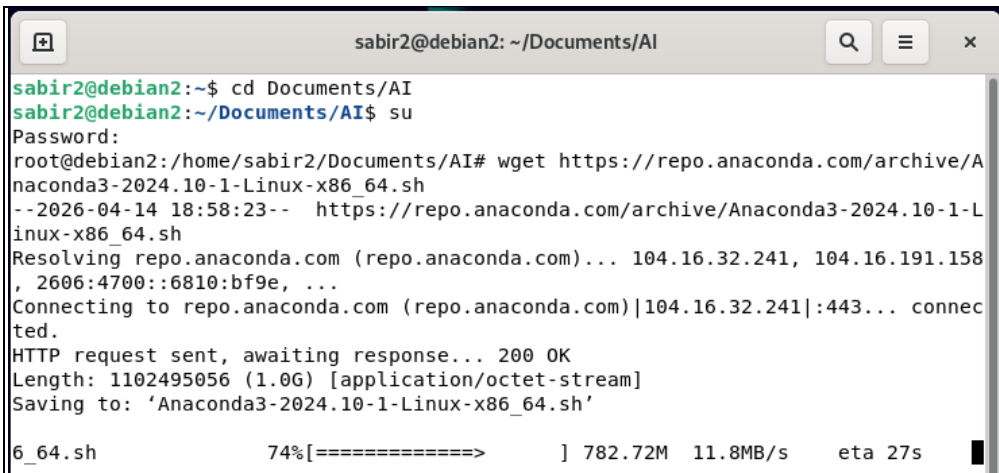


Рисунок 1 – Скачивание установочного скрипта для Anaconda

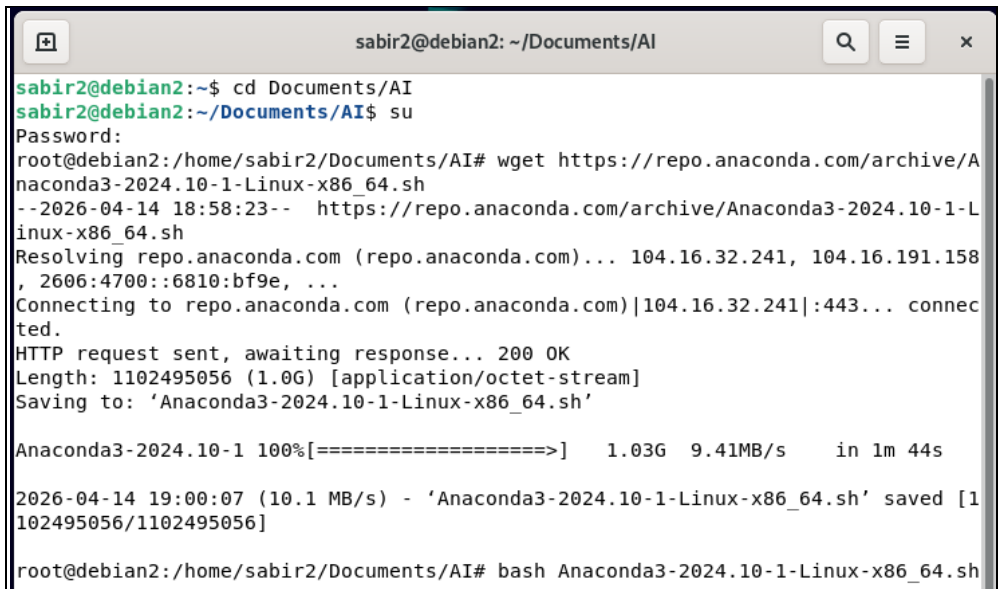


Рисунок 2 – Запуск установочного скрипта командой «bash»

После установки дистрибутива необходимо командой «conda create» создать окружение с указанием его названия и версии используемого языка программирования Python для реализуемого проекта (см. рисунок 3).

Перед началом настройки окружения проекта необходимо активировать его командой «conda activate» с указанием названия нужного окружения (см. рисунок 4).

```
sabir2@debian2: ~/Documents/AI
[no] >>> yes
no change      /root/anaconda3/condabin/conda
no change      /root/anaconda3/bin/conda
no change      /root/anaconda3/bin/conda-env
no change      /root/anaconda3/bin/activate
no change      /root/anaconda3/bin/deactivate
no change      /root/anaconda3/etc/profile.d/conda.sh
no change      /root/anaconda3/etc/fish/conf.d/conda.fish
no change      /root/anaconda3/shell/condabin/Conda.psm1
no change      /root/anaconda3/shell/condabin/conda-hook.ps1
no change      /root/anaconda3/lib/python3.12/site-packages/xontrib/conda.xsh
no change      /root/anaconda3/etc/profile.d/conda.csh
modified       /root/.bashrc

==> For changes to take effect, close and re-open your current shell. <==

Thank you for installing Anaconda3!
root@debian2:/home/sabir2/Documents/AI# source ~/.bashrc
(base) root@debian2:/home/sabir2/Documents/AI# conda create -n mpi_project python=3.9
Channels:
- defaults
Platform: linux-64
Collecting package metadata (repodata.json): |
```

Рисунок 3 – Создание conda-окружения с именем «mpi_project» и версией языка программирования Python 3.9

```
sabir2@debian2: ~/Documents/AI
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate mpi_project
#
# To deactivate an active environment, use
#
#     $ conda deactivate
#
(base) root@debian2:/home/sabir2/Documents/AI# conda activate mpi_project
```

Рисунок 4 – Активация conda-окружения с именем «mpi_project»

2.3.2 Установка пакетов и библиотек для разработки

После активации окружения при помощи пакетного менеджера `pip` и команды «`pip install`» нужно установить библиотеки, применяющиеся в текущем проекте, необходимые для обучения алгоритма и организации вычислений на кластере (см. рисунки 5 и 6). Такими библиотеками являются: «`torch`», «`pnumru`» и «`mpi4py`».

Для сборки программы и взаимодействия между узлами кластера, при помощи пакетного менеджера «apt» нужно установить следующие пакеты: «mpich», «libmpich-dev», «gcc», «g++» и «make».

```
sabir2@debian2: ~/Documents/AI

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate mpi_project
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) root@debian2:/home/sabir2/Documents/AI# conda activate mpi_project
(mpi_project) root@debian2:/home/sabir2/Documents/AI# pip install torch numpy
```

Рисунок 5 – Установка библиотек «torch» и «numpy» в conda-окружении

```
sabir2@debian2: ~/Documents/AI

Downloading markupsafe-3.0.3-cp39-cp39-manylinux2014_x86_64.manylinux_2_17_x86_64.manylinux_2_28_x86_64.whl (20 kB)
Downloading networkx-3.2.1-py3-none-any.whl (1.6 MB)
1.6/1.6 MB 4.5 MB/s 0:00:00
Installing collected packages: nvidia-cusparse-cu12, mpmath, zipp, typing-extensions, sympy, nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia-nccl-cu12, nvidia-curand-cu12, nvidia-cufile-cu12, nvidia-cuda-runtime-cu12, nvidia-cuda-nvrtc-cu12, nvidia-cuda-cupti-cu12, nvidia-cublas-cu12, numpy, networkx, MarkupSafe, fsspec, filelock, nvidia-cusparse-cu12, nvidia-cufft-cu12, nvidia-cudnn-cu12, Jinja2, importlib-metadata, triton, nvidia-cusolver-cu12, torch
Successfully installed MarkupSafe-3.0.3 filelock-3.19.1 fsspec-2025.10.0 importlib-metadata-8.7.1 Jinja2-3.1.6 mpmath-1.3.0 networkx-3.2.1 numpy-2.0.2 nvidia-cublas-cu12-12.8.4.1 nvidia-cuda-cupti-cu12-12.8.90 nvidia-cuda-nvrtc-cu12-12.8.93 nvidia-cuda-runtime-cu12-12.8.90 nvidia-cudnn-cu12-9.10.2.21 nvidia-cufft-cu12-11.3.3.83 nvidia-cufile-cu12-1.13.1.3 nvidia-curand-cu12-10.3.9.90 nvidia-cusolver-cu12-11.7.3.90 nvidia-cusparse-cu12-12.5.8.93 nvidia-cusparselt-cu12-0.7.1 nvidia-nccl-cu12-2.27.3 nvidia-nvjitlink-cu12-12.8.93 nvidia-nvtx-cu12-12.8.90 sympy-1.14.0 torch-2.8.0 triton-3.4.0 typing-extensions-4.15.0 zipp-3.23.1
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager, possibly rendering your system unusable. It is recommended to use a virtual environment instead: https://pip.pypa.io/warnings/venv. Use the --root-user-action option if you know what you are doing and want to suppress this warning.
(mpi_project) root@debian2:/home/sabir2/Documents/AI# pip install mpi4py
```

Рисунок 6 – Установка библиотеки «mpi4py» в conda-окружении

2.3.3 Установка среды разработки «Spyder»

Spyder (Scientific Python Development Environment) — это кроссплатформенная среда разработки с открытым исходным кодом,

специально созданная для анализа данных, научных исследований и инженерных расчетов на языке программирования Python.

Установить данную среду разработки можно из официальных репозиториях используемого дистрибутива операционной системы на базе ядра Linux (см. рисунок 7). После запуска среды разработки открывается её главное окно, в котором можно открыть текущий проект или создать новый (см. рисунок 8).

```
sabir@debian2:~$ su
Password:
(mpi_project) root@debian2:/home/sabir# apt update
Hit:1 http://security.debian.org/debian-security bullseye-security InRelease
Hit:2 http://ftp.de.debian.org/debian bullseye InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
291 packages can be upgraded. Run 'apt list --upgradable' to see them.
(mpi_project) root@debian2:/home/sabir# apt install spyder
```

Рисунок 7 – Установка среды разработки Spyder

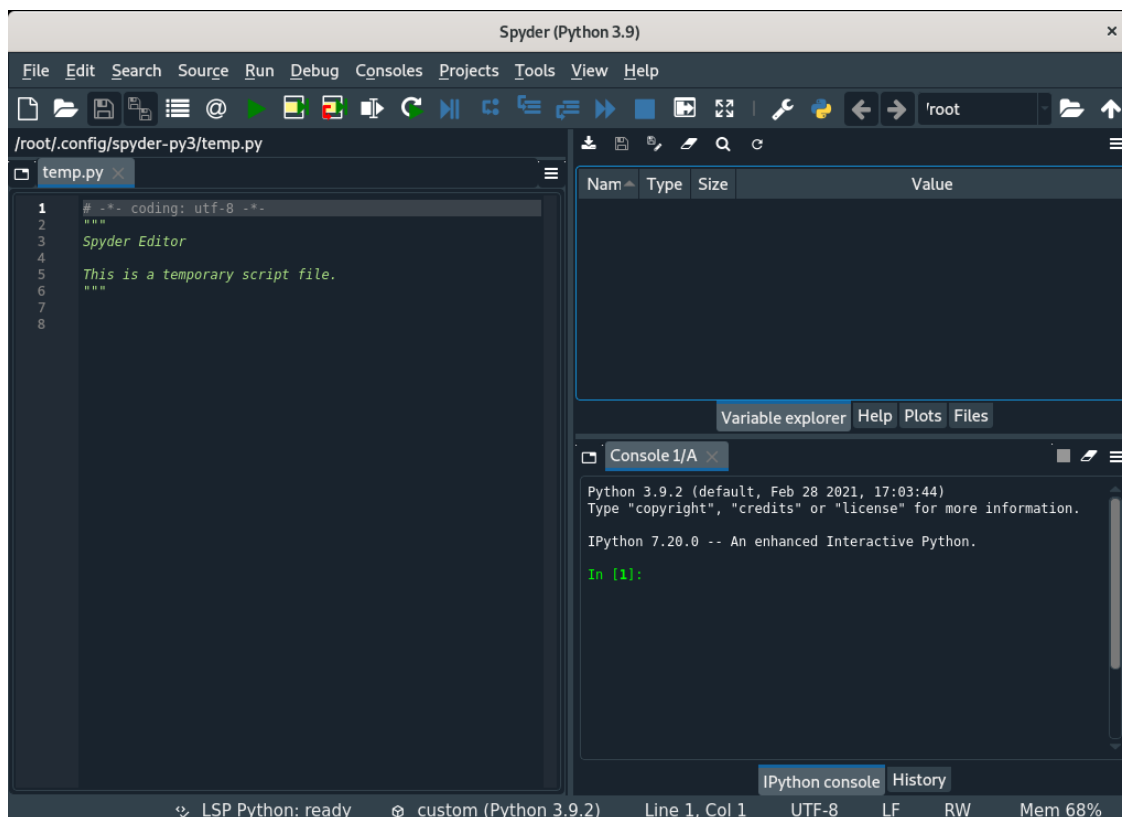


Рисунок 8 – Запущенная среда разработки Spyder

3 Архитектура приложения

3.1 Архитектура программы

Архитектура данного приложения представляет собой распределённую систему обучения нейронной сети, использующую технологию MPI для параллельной обработки данных и синхронизации градиентов. В основе системы лежит модель глубокого машинного обучения Transformer для генерации тегов, назначаемых на посты в социальных сетях. Система разбита на несколько логических компонентов, взаимодействующих через специальный механизм обмена сообщениями MPI.

3.1.1 Парадигма «SPMD» (Single Program Multiple Data)

SPMD – это модель параллельного программирования, в которой один и тот же экземпляр программы выполняется на нескольких вычислительных процессах, но каждый процесс обрабатывает свою часть данных и может иметь собственную логическую роль, например: master или worker. Схема парадигмы SPMD представлена на рисунке 9.

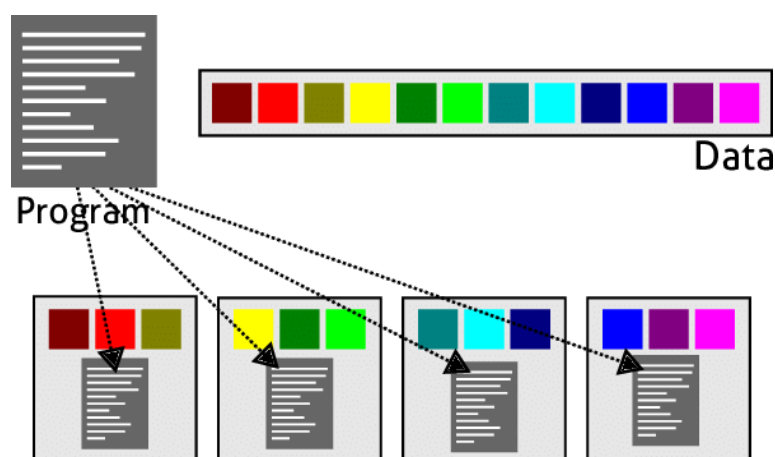


Рисунок 9 – Схема парадигмы SPMD

Парадигму SPMD обычно относят к программированию через обмен сообщениями для архитектур с распределенной памятью. Распределенная память состоит из набора независимых компьютеров, называемых «узлами».

Каждый узел запускает свой собственный экземпляр программы и взаимодействует с другими узлами посредством отправки и получения сообщений, вызывая для этого процедуры отправки и приема. Барьерная синхронизация также может быть реализована при помощи сообщений. Последовательные участки программ реализуются одинаковыми вычислениями на всех узлах, а не с помощью вычисления результата на одном узле и отправки его результата на другие.

В разработанной программе данная парадигма отражена в том, что один и тот же обучающий скрипт выполняется на каждом узле MPI-кластера. Код не изменяется для разных ролей – все процессы запускают идентичную программу. Однако, несмотря на единую кодовую базу, процессы получают разные роли через назначение ранга. Процесс с рангом 0 (именуемый master) выполняет дополнительные обязанности: загружает исходный JSON-файл, строит словарь тегов, собирает локальные частоты от всех процессов, рассылает готовый словарь, выполняет валидацию модели и сохраняет лучшую версию. Процессы с рангами от 1 до N-1 (именуемые workers) только получают словарь от мастера, загружают свою часть данных и участвуют в вычислении градиентов.

Каждый процесс работает со своей частью данных: DistributedSampler разбивает индексы обучающего датасета на world_size непересекающихся частей.

А для взаимодействия между процессами используются коллективные операции MPI, которые, по сути, являются обменом сообщениями: comm.gather, comm.bcast и comm.Allreduce.

3.1.2 Стратегия параллелизации обучения «Data Parallelism»

Data Parallelism – это метод распределённого машинного обучения, при котором данные разбиваются на части между вычислительными процессами, а модель полностью реплицируется на каждом процессе. Градиенты, вычисленные на разных частях данных, усредняются (синхронизируются)

коллективными операциями, после чего веса модели обновляются одинаково на всех процессах.

В программе данная стратегия реализована следующим образом: DistributedSampler разбивает обучающий датасет на world_size частей (по числу процессов), каждый процесс получает свой уникальный набор индексов; DataLoader с этим сэмплером загружает батчи только из своей части данных; Каждый процесс создаёт собственную копию модели ImprovedTagTransformer с одинаковой начальной инициализацией.

3.2 Архитектура используемой нейронной сети

В качестве нейронной сети для алгоритма рекомендаций было принято решение использовать модель глубокого машинного обучения Transformer, так как её задача – это обработка последовательностей данных. Главное преимущество модели Transformer заключается в способности обрабатывать длительные цепочки зависимостей в последовательностях. Кроме того, она способна обрабатывать последовательности данных параллельно.

3.2.1 Архитектура Transformer

Устройство модели Transformer состоит из кодирующего (энкодер) и декодирующего (декодер) компонентов (см. рисунок 10). На вход принимается последовательность данных, создается её векторное представление (эмбендинг), прибавляется вектор позиционного кодирования, после чего набор элементов без учета порядка в последовательности поступает в кодирующий компонент, а затем декодирующий компонент получает на вход часть этой последовательности и выход кодирующего. В результате получается новая выходная последовательность.

Transformer — это поставленные друг за другом модели внимания, которые позволяют исходную последовательность векторов перевести в новую, вектора в которой кодируют информацию о контексте каждого элемента. Transformer-кодировщик переводит исходные векторы в скрытые, которые

правильно сохраняют в себе информацию о контексте каждого элемента. Далее Transformer-декодировщик декодирует результат кодировщика в новую последовательность, состоящую из эмбедингов элементов выходного языка.

В результате по эмбедингам генерируются сами итоговые элементы с помощью вероятностной языковой модели.

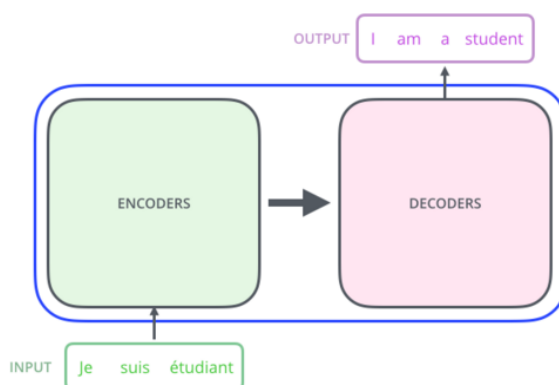


Рисунок 10 – Упрощённая архитектура модели Transformer

3.2.2 Transformer-encoder

Схема архитектуры Transformer-encoder представлена на рисунке 11.

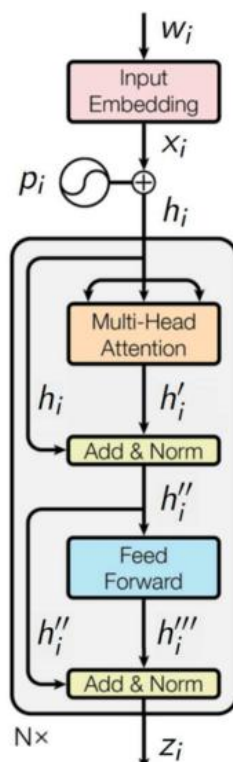


Рисунок 11 – Transformer-encoder

Работа кодировщика определяется последовательностью из следующих семи шагов:

а) на вход поступает обрабатываемая последовательность элементов w_i , по которой создается последовательность эмбедингов, где каждый x_i - это векторное представление элемента w_i ;

б) далее добавляются векторы позиционного кодирования p_i : $h_i = x_i + p_i$, $H = (h_1, \dots, h_n)$. Основное свойство позиционного кодирования — чем дальше вектора будут стоять друг от друга в последовательности, тем больше между ними будет расстояние;

с) полученный вектор h_i подается на вход в блок многомерного самовнимания (multi-headed self-attention). $h_{ji} = \text{Attn}(Q_j h_i, K_j H, V_j H)$, где обрабатываются обучаемые матрицы: Q для запроса, K для ключа, V для значения;

д) затем необходима конкатенация, чтобы вернуться в исходную размерность: $h'_i = \text{MHj}(h_{ji}) = [h_{1i} \dots h_{ji}]$;

е) добавление сквозной связи (skip connection) — добавление из входного вектора к выходному ($h'_i + h_i$). Затем производится нормализация слоя (layer normalization): $h''_i = \text{LN}(h'_i + h_i; \mu_1, \sigma_1)$. У нее два обучаемых параметра, для каждой размерности вектора вычисляется среднее и дисперсия;

ф) теперь добавляется преобразование, которое будет обучаемым — полносвязную двухслойную нейронную сеть: $h'''_i = W_2 \text{ReLU}(W_1 h''_i + b_1) + b_2$;

г) после всего этого снова происходит повторение пункта 5: добавляется сквозная связь и нормализация слоя: $z_i = \text{LN}(h'''_i + h''_i; \mu_2, \sigma_2)$;

После, в кодирующем компоненте пункты кодировщика с - г повторяются еще несколько раз, преобразовывая друг за другом из контекста контекст. Тем самым происходит обогащение модели и увеличение количества её параметров.

Так как в архитектуре Transformer обработка последовательности заменяется на обработку множества, возникает необходимость в хранении информации о порядке элементов последовательности. Чтобы отобразить

информацию о позиции элемента в исходной последовательности используется позиционное кодирование. Позиционное кодирование (positional encoding) — позволяет модели получить информацию о порядке элементов в последовательности путем прибавления специальных меток к вектору входных элементов.

Самый важный механизм в архитектуре Transformer — это внимание (attention). Он позволяет нейронной сети понять, какая часть входной последовательности данных наиболее подходит задаче. Механизм внимания определяет для каждого токена последовательности, какие другие токены необходимы для его понимания в данном контексте. Процесс вычисления внимания представлен схемой, отображённой на рисунке 12.

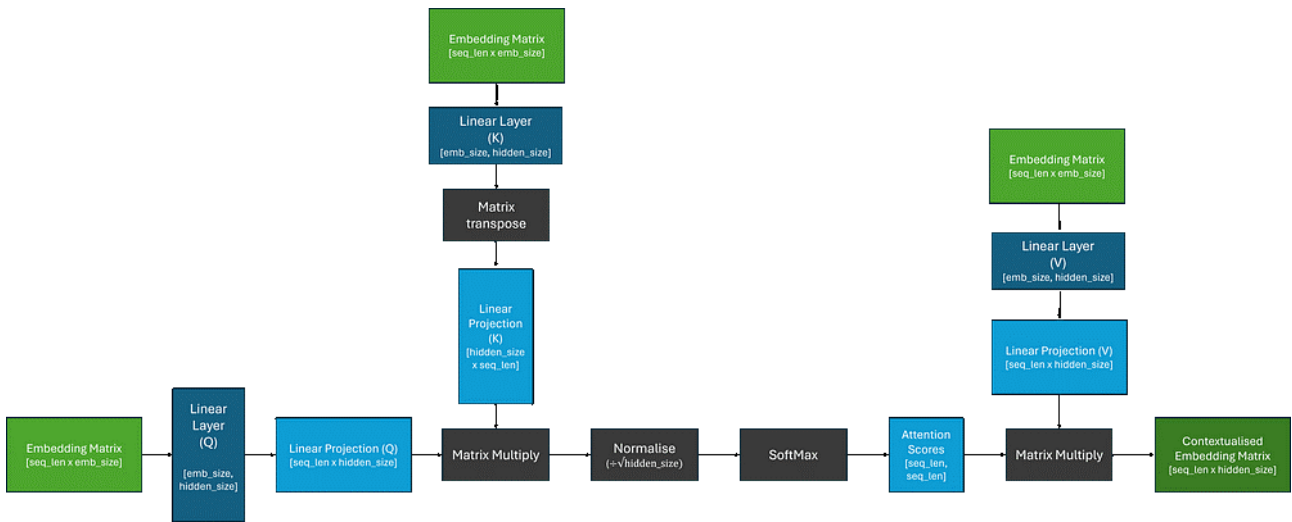


Рисунок 12 – Схема вычисления внимания

Как правило, параллельно запускается несколько блоков внимания, чтобы Transformer одновременно обрабатывал разные части входной последовательности данных — это называют multi-head attention. Идея данного подхода заключается в том, что выходы нескольких независимых блоков self-attention конкатенируются и передаются через линейный слой. Он позволяет модели комбинировать контекстную информацию из каждого блока внимания.

3.2.3 Transformer-decoder

На вход декодировщика подается выход кодировщика. Главное отличие архитектуры декодировщика заключается в том, что дополнительно имеется attention к вектору, который получен из последнего блока кодирующего компонента. Компонент декодировщика тоже многослойный и каждому блоку компонента на вход подается вектор именно с последнего блока кодирующего компонента.

Схема архитектуры Transformer-decoder представлена на рисунке 13.

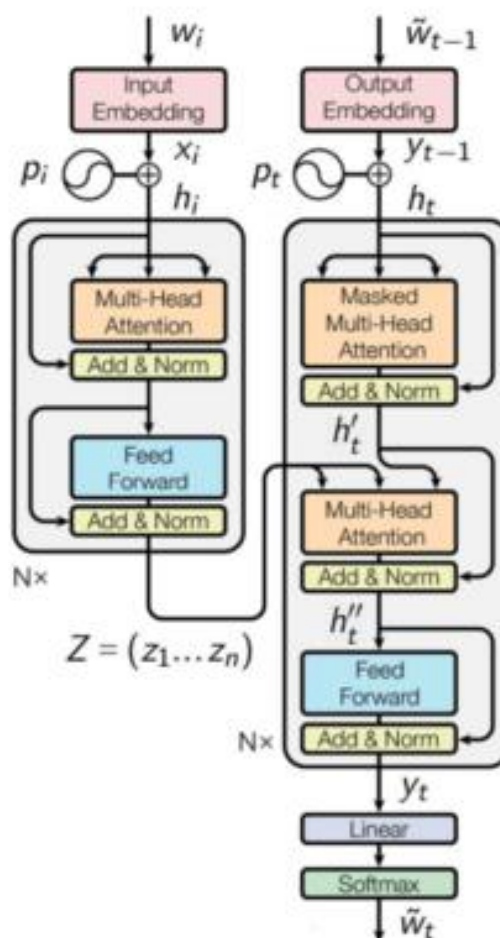


Рисунок 13 – Transformer-decoder

Работа декодировщика определяется следующими шагами:

а) для того, чтобы распараллелить декодировщик и уйти от рекуррентности, но также генерировать элементы друг за другом, используется

прием маскирования данных из будущего. Принцип этого метода заключается в том, что мы запрещаем модели подглядывать в те элементы, которые еще не сгенерированы с учетом порядка. Когда генерируем элемент под номером t , имеем право смотреть только первые $t-1$ элементов: $h_t = y_{t-1} + p_t$; $H_t = (h_1, \dots, h_t)$;

b) далее идет этап многомерного самовнимания: линейная нормализация и multi-headed self-attention. Особенность в том, что в attention ключи и значения применяются не ко всем векторам, а только к тем, значения которых уже синтезировали (H_t): $h'_t = \text{LN} \circ \text{MHj} \circ \text{Attn}(Q_j h_t, K_j H_t, V_j H_t)$, где $\circ$ — композиция;

c) на следующем этапе производится многомерное внимание на кодировку Z , результат работы компонента кодировщика: $h''_t = \text{LN} \circ \text{MHj} \circ \text{Attn}(Q_j h_t, K_j Z, V_j Z)$;

d) линейная полносвязная сеть (по аналогии с кодировщиком): $y_t = \text{LN} \circ \text{FNN}(h''_t)$;

e) в самом конце получится вероятностная порождающая модель для элементов. Результат (индекс слова с наибольшей вероятностью): $\text{SoftArgMax}(W y_t + b_y)$, где W_y , b_y — обучаемые параметры линейного преобразования. Для каждой позиции t выходной последовательности строится вероятностная модель языка, то есть все элементы из выходного словаря получают значение вероятности. Эти значения как раз получаются из векторов y_t из предыдущего пункта, которые берутся с последнего блока трансформера-декодировщика.

Последний этап выполняется только после того, когда повторились пункты a-d для всех декодировщиков. На выходе получаются вероятности классов, по факту для каждой позиции решается задача многоклассовой классификации, для того, чтобы понять какие элементы лучше поставить на каждые позиции.

4 Разработка программы, развёртывание и запуск распределённой вычислительной системы

4.1 Разработка модулей программы

Разработанная программа состоит из двух файлов: первый – main.py, отвечающий за распараллеленное обучение алгоритма рекомендаций социальных сетей; второй – predict.py, задача которого – генерация тегов, назначаемых на рекомендуемые посты при помощи обученной модели. UML-диаграмма классов основной программы представлена на рисунке 14.

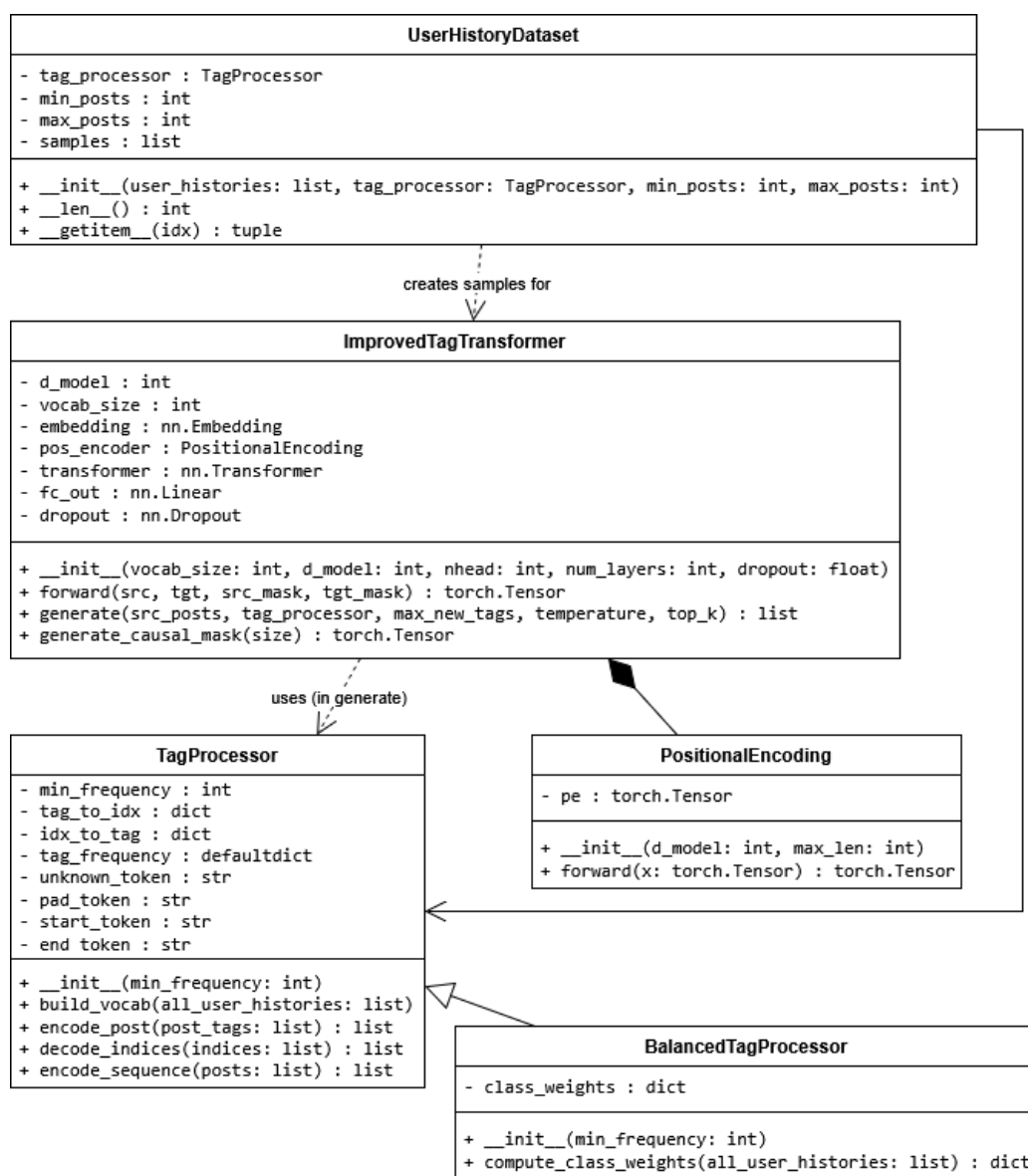


Рисунок 14 – UML-диаграмма классов основной программы main.py

4.1.1 Разработка модуля обработки тегов

Модуль обработки тегов реализован в классах TagProcessor и BalancedTagProcessor файла main.py. Его назначение – преобразование текстовых тегов в числовые индексы и вычисление весов для борьбы с дисбалансом классов.

Класс TagProcessor содержит следующие методы:

- build_vocab – сканирует все посты загруженных пользователей, подсчитывает частоту тегов, создаёт словарь tag_to_idx и его обратное отображение idx_to_tag. Помимо отсканированных индексов, добавляет также служебные токены: <PAD> (индекс 0), <START> (1), <END> (2), <UNK> (3);
- encode_post – принимает список тегов, а возвращает список их индексов, где неизвестные теги заменяются на <UNK>;
- encode_sequence – склеивает несколько постов в один плоский список индексов (используется для формирования входной последовательности модели);
- decode_indices – обратное преобразование, отбрасывает все служебные токены.

Класс BalancedTagProcessor унаследован с TagProcessor и реализует данный метод compute_class_weights, который по собранной частоте тегов вычисляет веса для функции потерь: $weight = total / (num_classes * freq)$. Редкие теги получают больший вес, что вынуждает модель уделять им больше внимания.

Данные модули позволяют идентичным образом работать с тегами на всех MPI-процессах, а также корректно обрабатывать новые теги, не встреченные в обучающей выборке через добавление токена <UNK>.

4.1.2 Разработка модуля обработки датасета

Класс UserHistoryDataset формирует обучающие пары из истории активности пользователя. Далее представлен реализуемый алгоритм:

- берётся последовательность оценённых постов каждого пользователя;

- скользящим окном (в диапазоне от `min_posts` до `max_posts`) генерируются примеры: вход = `num_input_posts` предыдущих постов, цель = следующий пост;

- количество входных постов выбирается случайным образом в заданном диапазоне, что повышает обобщающую способность модели;

- входная последовательность (теги нескольких постов) склеивается в один плоский список индексов;

- цель индексов дополняется токенами `<START>` и `<END>` для авторегрессивной генерации.

Отдельно выделена функция `collate_batch`, которая принимает список примеров разной длины; дополняет все последовательности нулями (индекс `<PAD>`) до максимальной длины в батче; создаёт маску, указывающую, где находятся реальные токены, а где паддинг (для игнорирования в `loss`).

Использование `DistributedSampler` с параметрами `num_replicas=world_size` и `rank=rank` обеспечивает равное разбиение датасета между MPI-процессами. Каждый процесс получает свой уникальный набор индексов, для определения границ используемой области датасета.

4.1.3 Разработка модуля, реализующего работу с моделью Transformer

Функционал класса `ImprovedTagTransformer` построен на основе использования стандартного `nn.Transformer` из PyTorch. Структура организации модуля:

- входной эмбединг (`nn.Embedding`) – преобразует индексы тегов в векторы размерности `d_model`;

- позиционное кодирование (`PositionalEncoding`) – добавляет информацию о позиции токена в последовательности через синусоидальные функции, чтобы модель могла различать порядок тегов;

- непосредственно сам `Transformer` (`nn.Transformer`) – включает энкодер и декодер с механизмом многоголового самовнимания. Переключение `batch_first=True` упрощает работу с батчами;

- выходной линейный слой (fc_out) – проецирует выход декодера на размерность словаря (количество возможных тегов);
- dropout – регулировка для предотвращения переобучения.

Метод forward: вызывает эмбединг и позиционное кодирование для входной и целевой последовательностей; передаёт последовательности в nn.Transformer с масками для игнорирования паддингов; возвращает логиты (не нормализованные вероятности) для всех позиций.

Метод generate для авторегрессивной генерации: инициализирует выход с токеном <START>; на каждом шаге подаёт текущий выход в декодер, предсказывает следующий токен; применяет top-k sampling (ограничение числа кандидатов) и температуру для контроля разнообразия выдачи; запрещает повторение уже сгенерированных тегов и раннее завершение (первые два шага запрещён <END>); при преждевременной остановке возвращает наиболее частые теги из словаря.

4.1.4 Реализация распределённого обучения с применением MPI

Распределённое обучение реализовано в функции distributed_train, которая вызывается из основного блока if __name__ == "__main__". Архитектура следует парадигме SPMD, описанной ранее.

Работа программы для обучения алгоритма происходит в шесть этапов:

- инициализация MPI: каждый процесс получает свой ранг (comm.Get_rank()) и размер мира (comm.Get_size()); мастер-процесс (rank 0) отвечает за загрузку данных, валидацию и сохранение модели; воркеры (rank ≥ 1) участвуют в вычислении градиентов;
- сбор частот тегов и построение словаря: только мастер читает JSON-файл и строит user_histories; все процессы вычисляют локальную частоту тегов на своих данных; мастер объединяет частоты, строит глобальный словарь и рассылает его через comm.bcast;

– распределение данных: DistributedSampler разбивает индексы датасета между процессами; DataLoader с этим сэмплером загружает только свою часть данных;

– цикл обучения: в каждой эпохе вызывается sampler.set_epoch(epoch) для корректного перемешивания; проводятся операции над каждым батчем (прямой проход, вычисление loss, обратное распространение, ручная синхронизация градиентов – каждый процесс усредняет свои градиенты с градиентами всех остальных через comm.Allreduce, шаг оптимизатора);

– валидация и ранняя остановка: валидация выполняется только на мастере; если валидационная ошибка улучшилась, мастер сохраняет модель; при отсутствии улучшений в течение patience эпох мастер рассылает сигнал остановки через comm.bcast, воркеры завершают цикл;

– профилирование: замеряется время чистых вычислений (forward+backward) и время синхронизации MPI для каждого батча и каждой эпохи; в конце работы мастер выводит детальную статистику: общее время, долю вычислений и коммуникации, среднее время эпохи.

Все этапы работы основной программы отображены на UML-диаграмме последовательности, продемонстрированной на рисунке 15.

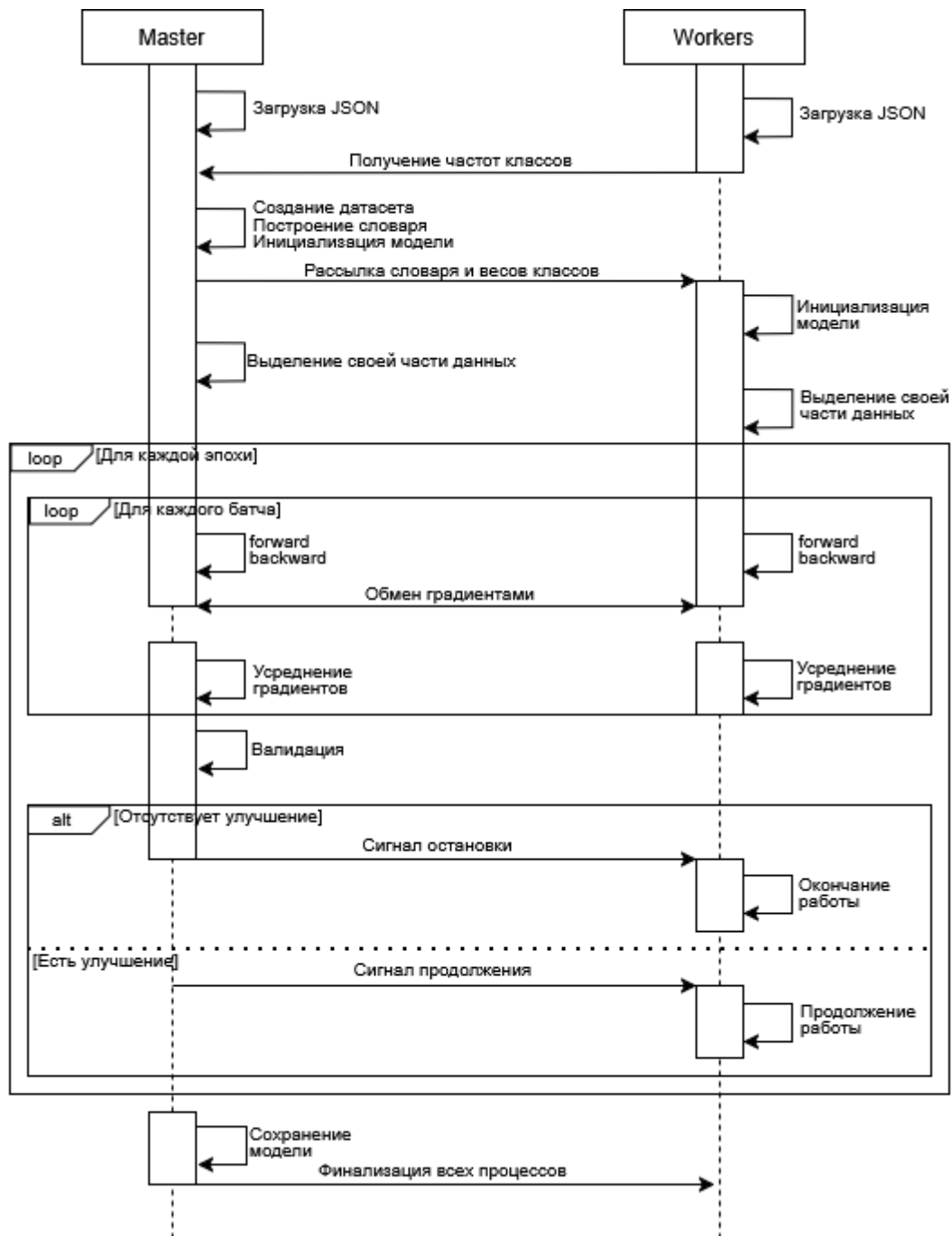


Рисунок 15 – UML-диаграмма последовательности основной программы main.py

4.1.5 Разработка модуля генерации рекомендаций

Файл predict.py предназначен для загрузки обученной модели и генерации тегов для следующего предлагаемого поста на основе истории предыдущих постов конкретного пользователя. Это позволяет использовать модель в реальном приложении без необходимости повторного обучения. Модуль

работает полностью независимо от обучающей программы, используя только сохранённые файлы модели и словаря.

Основные функции модуля генерации:

- `load_model()` – загружает сохранённую модель и объект `TagProcessor` из файла `*.pt`;
- `load_input_from_json()` – читает входные данные пользователя, содержащие теги с оценённых им недавно постов, из JSON-файла;
- `generate_tags()` – обёртка для вызова метода `generate` модели;
- `save_result_to_json()` – сохраняет сгенерированные теги в JSON-файл с полями: `recommended_tags` – список тегов; `status` – статус выполнения; `num_tags` – количество тегов;
- `main()` – организует полный конвейер: разбор аргументов командной строки; проверка существования модели и входного файла; загрузка модели и входных постов; вызов `generate_tags` с заданными гиперпараметрами; вывод результата и его сохранение в JSON.

`predict.py` импортирует классы `ImprovedTagTransformer` и `BalancedTagProcessor` из `main.py`, что гарантирует полную совместимость структуры модели и обработчика тегов. Загрузка модели происходит через `torch.load`, причём словари восстанавливаются точно такими же, как при обучении (благодаря сохранению их в `checkpoint`). Это обеспечивает корректное отображение индексов в теги и обратно. UML-диаграмма классов модуля генерации представлена на рисунке 16.

Скрипт не требует повторного обучения и может запускаться на любом компьютере с установленной библиотекой `PyTorch`. Результат генерации сохраняется в структурированном JSON-формате, что упрощает интеграцию с веб-приложениями или другими сервисами. Наличие аргументов командной строки позволяет гибко указывать пути к файлам, что автоматизирует пакетную обработку.

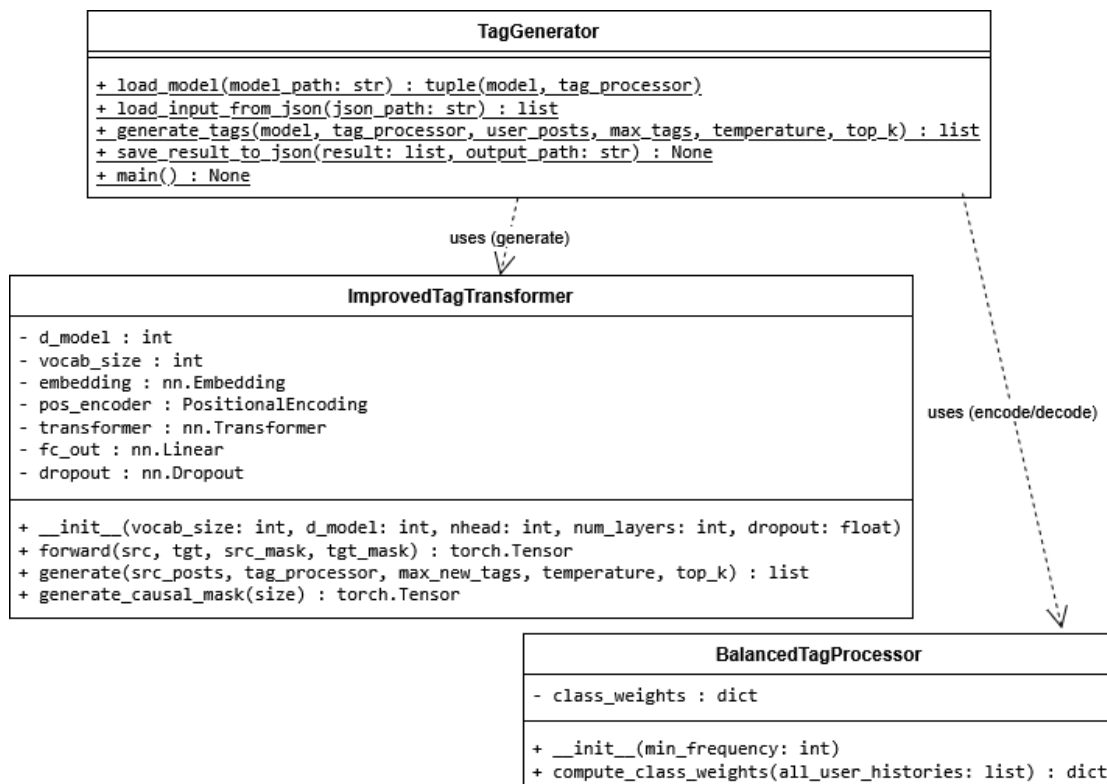


Рисунок 16 – UML-диаграмма классов модуля генерации predict.py

4.2 Модификация основной программы для использования технологии PyTorch DDP

4.2.1 Изменения в инициализации распределённого окружения

В отличие от оригинального варианта, где использовался mpi4py и вызовы MPI.COMM_WORLD, модифицированная версия получает параметры распределённого запуска через переменные окружения RANK, WORLD_SIZE и LOCAL_RANK. Инициализация группы процессов выполняется вызовом `dist.init_process_group(backend='gloo')`. Выбор бэкенда `gloo` обусловлен ориентацией на CPU-вычисления, для которых этот бэкенд является стандартным.

Так как все процессы запускаются с одинаковыми аргументами, словарь тегов теперь строится каждым процессом независимо по полной копии данных, что избавляет от необходимости операций `gather` и `bcast`, характерных для MPI-решения. Для обеспечения идентичности разбиения данных на обучающую и

валидационную выборки применяется фиксированный seed(42) для генераторов псевдослучайных чисел.

4.2.2 Применение DistributedDataParallel

Центральным изменением является обёртка модели в DistributedDataParallel (DDP) вместо ручной реализации коллективных операций синхронизации градиентов.

Это позволило исключить ручной цикл синхронизации градиентов через comm.Allreduce. Теперь сразу после прямого и обратного проходов (loss.backward()) DDP в фоновом режиме выполняет усреднение градиентов с использованием наиболее эффективных для CPU алгоритмов коллективных операций. Шаг оптимизатора optimizer.step() завершает все незаконченные коммуникации и обновляет веса модели.

Обратной стороной модификации является необходимость обращаться к атрибутам оригинальной модели через model.module.

4.2.3 Упрощение цикла обучения

Функция distributed_train_ddp сохранила общую логику обучения, но лишилась явного блока синхронизации градиентов.

Для обеспечения одинакового числа шагов на каждой итерации загрузчик тренировочных данных использует параметр drop_last=True, отбрасывающий неполный последний батч.

4.2.4 Сравнение этапов выполнения модифицированной программы с исходной версией

Последовательность операций модифицированной программы выглядит следующим образом:

- инициализация: чтение переменных окружения RANK, WORLD_SIZE, LOCAL_RANK, создание группы процессов с бэкендом gloo;
- подготовка данных: каждый процесс загружает JSON, строит словарь и создаёт датасет независимо (в изначальной версии только мастер собирал статистику, а затем рассылал словарь);

- распределение данных: DistributedSampler как и ранее разбивает индексы между процессами;
- цикл обучения: прямой проход → вычисление потерь → обратный проход → асинхронная синхронизация градиентов (внутри DDP) → шаг оптимизатора;
- валидация и ранняя остановка: выполняются только на процессе с рангом 0; сигнал остановки не рассылается явно, так как при использовании DDP достаточно выйти из цикла на мастере, а остальные процессы будут автоматически завершены;
- профилирование: собираются времена вычислений и DDP-синхронизации, выводится итоговая статистика, аналогичная MPI-версии.

4.3 Поднятие кластера и запуск распараллеленного обучения

4.3.1 Поднятие кластера на базе трёх ЭВМ под управлением операционной системы Linux Alt Workstation 10.4

Основная идея кластера – запуск одного и того же экземпляра программы на нескольких узлах. Как правило, для реализации подобной модели требуется доступ по ssh-соединению между всеми узлами кластера, для удалённого запуска распределённой программы с мастера. Поэтому, прежде чем производить поднятие и запуск необходимо убедиться, что все компьютеры в сети видят друг друга и на каждом из них свободен порт 22, по которому происходит коммуникация по ssh-соединению. Также, особенностью подобных распределённых систем является то, что окружение Anaconda и все необходимые пакеты, в том числе исходники программы и данные, должны быть доступны по одинаковому локальному пути на каждой ЭВМ.

UML-диаграмма развёртывания кластера показана на рисунке 17.

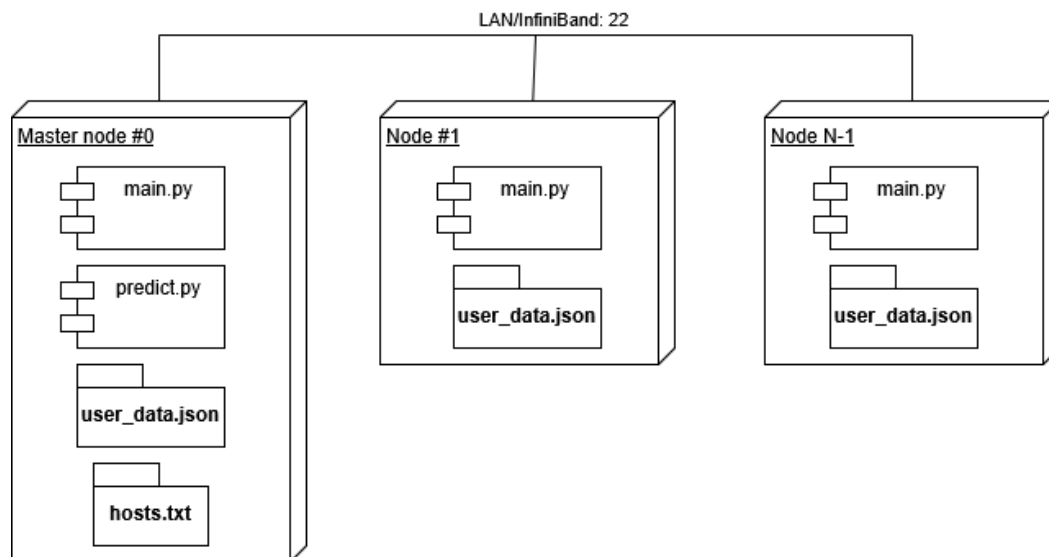


Рисунок 17 – UML-диаграмма развёртывания кластера

Первым этапом необходимо определить ip-адреса, назначенные машинам в сети, которые будут объединены в кластер:

```
[adminmpi@vt316-12u ~]$ hostname -i
192.168.10.72
```

```
[adminmpi@vt316-11u ~]$ hostname -i
192.168.10.71
```

```
[adminmpi@vt316-10u ~]$ hostname -i
192.168.10.70
```

Затем – определяется доступность всех узлов в сети (см. рисунки 18-20).

```
[adminmpi@vt316-12u ~]$ ping -c 4 192.168.10.71
PING 192.168.10.71 (192.168.10.71) 56(84) bytes of data.
64 bytes from 192.168.10.71: icmp_seq=1 ttl=64 time=0.206 ms
64 bytes from 192.168.10.71: icmp_seq=2 ttl=64 time=0.262 ms
64 bytes from 192.168.10.71: icmp_seq=3 ttl=64 time=0.261 ms
64 bytes from 192.168.10.71: icmp_seq=4 ttl=64 time=0.265 ms

--- 192.168.10.71 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3074ms
rtt min/avg/max/mdev = 0.206/0.248/0.265/0.024 ms
[adminmpi@vt316-12u ~]$ ping -c 4 192.168.10.70
PING 192.168.10.70 (192.168.10.70) 56(84) bytes of data.
64 bytes from 192.168.10.70: icmp_seq=1 ttl=64 time=0.219 ms
64 bytes from 192.168.10.70: icmp_seq=2 ttl=64 time=0.263 ms
64 bytes from 192.168.10.70: icmp_seq=3 ttl=64 time=0.258 ms
64 bytes from 192.168.10.70: icmp_seq=4 ttl=64 time=0.260 ms

--- 192.168.10.70 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3059ms
rtt min/avg/max/mdev = 0.219/0.250/0.263/0.018 ms
[adminmpi@vt316-12u ~]$
```

Рисунок 18 – Перекрёстный ping между ЭВМ в сети

Изм.	Лист	№ докум.	Подпись	Дата

```
[adminmpi@vt316-11u ~]$ ping -c 4 192.168.10.72
PING 192.168.10.72 (192.168.10.72) 56(84) bytes of data.
64 bytes from 192.168.10.72: icmp_seq=1 ttl=64 time=0.223 ms
64 bytes from 192.168.10.72: icmp_seq=2 ttl=64 time=0.217 ms
64 bytes from 192.168.10.72: icmp_seq=3 ttl=64 time=0.108 ms
64 bytes from 192.168.10.72: icmp_seq=4 ttl=64 time=0.278 ms

--- 192.168.10.72 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3064ms
rtt min/avg/max/mdev = 0.108/0.206/0.278/0.061 ms
[adminmpi@vt316-11u ~]$ ping -c 4 192.168.10.70
PING 192.168.10.70 (192.168.10.70) 56(84) bytes of data.
64 bytes from 192.168.10.70: icmp_seq=1 ttl=64 time=0.366 ms
64 bytes from 192.168.10.70: icmp_seq=2 ttl=64 time=0.250 ms
64 bytes from 192.168.10.70: icmp_seq=3 ttl=64 time=0.201 ms
64 bytes from 192.168.10.70: icmp_seq=4 ttl=64 time=0.211 ms

--- 192.168.10.70 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3096ms
rtt min/avg/max/mdev = 0.201/0.257/0.366/0.065 ms
```

Рисунок 19 – Перекрёстный ping между ЭВМ в сети

```
[adminmpi@vt316-10u ~]$ ping -c 4 192.168.10.72
PING 192.168.10.72 (192.168.10.72) 56(84) bytes of data.
64 bytes from 192.168.10.72: icmp_seq=1 ttl=64 time=0.216 ms
64 bytes from 192.168.10.72: icmp_seq=2 ttl=64 time=0.263 ms
64 bytes from 192.168.10.72: icmp_seq=3 ttl=64 time=0.257 ms
64 bytes from 192.168.10.72: icmp_seq=4 ttl=64 time=0.256 ms

--- 192.168.10.72 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3051ms
rtt min/avg/max/mdev = 0.216/0.248/0.263/0.018 ms
[adminmpi@vt316-10u ~]$ ping -c 4 192.168.10.71
PING 192.168.10.71 (192.168.10.71) 56(84) bytes of data.
64 bytes from 192.168.10.71: icmp_seq=1 ttl=64 time=0.216 ms
64 bytes from 192.168.10.71: icmp_seq=2 ttl=64 time=0.250 ms
64 bytes from 192.168.10.71: icmp_seq=3 ttl=64 time=0.276 ms
64 bytes from 192.168.10.71: icmp_seq=4 ttl=64 time=0.262 ms

--- 192.168.10.71 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3112ms
rtt min/avg/max/mdev = 0.216/0.251/0.276/0.022 ms
```

Рисунок 20 – Перекрёстный ping между ЭВМ в сети

Первым этапом в настройке ssh-соединения является генерация публичного ключа. Для этого можно воспользоваться утилитой `ssh-keygen`. Стоит отметить, что при настройке необходимо отключить удалённый доступ по паролю и по кодовой фразе к компьютеру, на котором будет выполняться экземпляр программы. Записать сгенерированный ssh-ключ на будущие узлы можно утилитой `ssh-copy-id` (см. рисунок 21). Операции создания и копирования ключа должны быть произведены на всех компьютерах, чтобы любой узел имел публичные ssh-ключи всех остальных ЭВМ в вычислительном кластере.

```
[adminmpi@vt316-12u ~]$ ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa
Generating public/private rsa key pair.
Your identification has been saved in /home/adminmpi/.ssh/id_rsa.
Your public key has been saved in /home/adminmpi/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:XoHSXvav1LcJDYPOBqJLmVH+lraxszQ03EgucNRrP48 adminmpi@vt316-12u.bt.local
The key's randomart image is:
+---[RSA 2048]-----+
|
|.. .
|..oo +
|.oo.o o.
|.o *S....o
|o X.*.= o+
|* =.% +..o..|
|.o *.X. ...o|
|.Eo.. o
+---[SHA256]-----+
[adminmpi@vt316-12u ~]$ ssh-copy-id adminmpi@192.168.10.71
The authenticity of host '192.168.10.71 (192.168.10.71)' can't be established.
ED25519 key fingerprint is SHA256:u8WxM0hRKkPgA0zB/rMCgLhj0uR2xKC/jiW18CfKUzY.
Are you sure you want to continue connecting (yes/no)?
```

Рисунок 21 – Создание и копирование ssh-ключа для беспарольного доступа к остальным компьютерам в сети

Проверка ssh-соединения продемонстрирована на рисунке 22.

```
[adminmpi@vt316-12u ~]$ ssh adminmpi@192.168.10.71 hostname -i
192.168.10.71
[adminmpi@vt316-12u ~]$
```

Рисунок 22 – Удалённое выполнение программы для проверки ssh-соединения

В ходе поднятия вычислительного кластера на ЭВМ под управлением ОС Linux Alt Workstation 10.4 выяснилось, что в репозиториях данного дистрибутива отсутствует пакет MPICH, необходимый для установки утилиты mpiexec, при помощи которой запускаются распределённые вычисления. Для сборки недостающего пакета из исходного кода необходимо установить нужные для компиляции пакеты через пакетный менеджер apt (см. рисунок 23), а затем – скачать архива с исходным кодом пакета MPICH (см. рисунки 23 и 24). После скачивания архива с исходным кодом его необходимо распаковать (см. рисунок 25), переместиться в полученный каталог и осуществить конфигурирование, которое продемонстрировано на иллюстрации 26.

```
[adminmpi@vt316-12u Downloads]$ sudo apt-get install gcc gcc-c++ make -y
Чтение списков пакетов... Завершено
Построение дерева зависимостей... Завершено
Последняя версия gcc уже установлена.
Последняя версия gcc-c++ уже установлена.
Последняя версия make уже установлена.
0 будет обновлено, 0 новых установлено, 0 пакетов будет удалено и 431 не будет о
бновлено.
[adminmpi@vt316-12u Downloads]$ wget https://www.mpich.org/static/downloads/4.2.
1/mpich-4.2.1.tar.gz
--2026-04-24 10:48:51-- https://www.mpich.org/static/downloads/4.2.1/mpich-4.2.
1.tar.gz
Resolving www.mpich.org (www.mpich.org)... 172.65.90.25, 172.65.90.27, 172.65.90
.26, ...
Connecting to www.mpich.org (www.mpich.org)|172.65.90.25|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 39915201 (38M) [application/x-gzip]
Saving to: 'mpich-4.2.1.tar.gz'

mpich-4.2.1.tar.gz  0%[                ] 24.06K  ---KB/s
```

Рисунок 23 – Установка пакетов и скачивание исходного кода

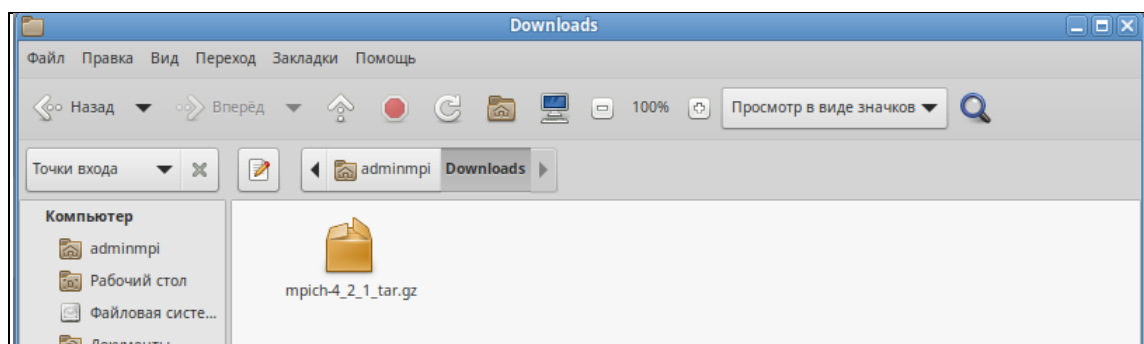


Рисунок 24 – Архив успешно скачан

```
[adminmpi@vt316-12u Downloads]$ tar -xzf mpich-4_2_1_tar.gz
[adminmpi@vt316-12u Downloads]$
```

Рисунок 25 – Распаковка архива с исходным кодом

```
[adminmpi@vt316-12u mpich-4.2.1]$ ./configure --disable-fortran --prefix=/usr/lo
cal
Configuring MPICH version 4.2.1 with '--disable-fortran' '--prefix=/usr/local'
Running on system: Linux vt316-12u.bt.local 6.1.121-un-def-alt1 #1 SMP PREEMPT_D
YNAMIC Sat Dec 28 13:03:47 UTC 2024 x86_64 GNU/Linux
checking build system type... x86_64-pc-linux-gnu
```

Рисунок 26 – Конфигурирование

Как только конфигурирование завершилось, можно приступить к непосредственной сборке при помощи утилиты make (см. рисунок 27). А затем – при помощи той же утилиты осуществить установку пакета (см. рисунок 28).

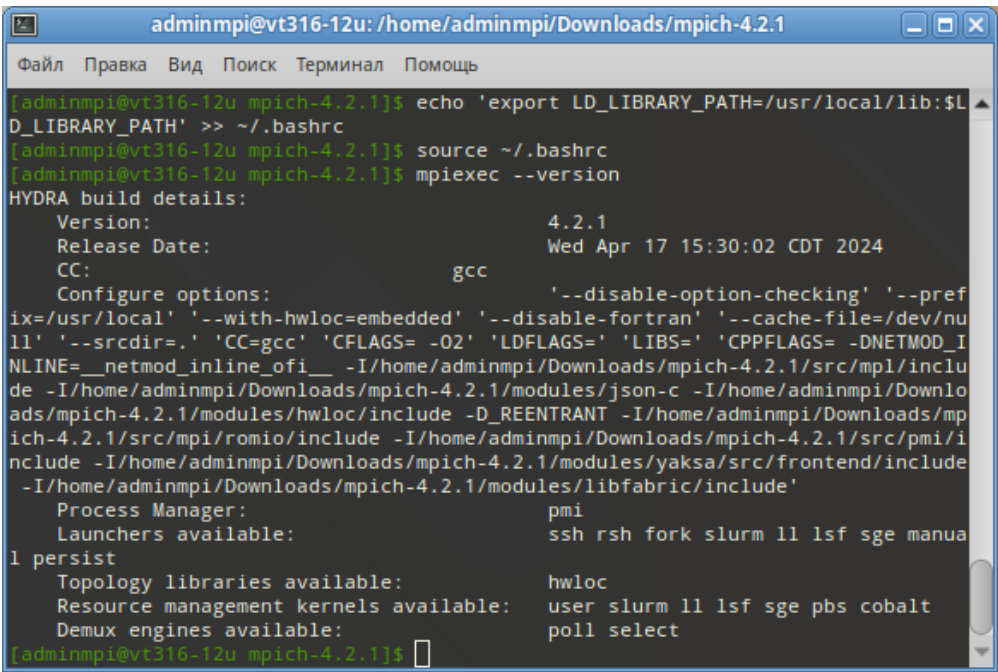
Как только пакет будет установлен, проводится настройка переменных окружения и проверка его распознавания системой (см. рисунок 29).

```
is also available.
*****
Configuration completed.
[adminmpi@vt316-12u mpich-4.2.1]$ make -j$(nproc)
```

Рисунок 27 – Сборка при помощи утилиты make

```
make[2]: выход из каталога «/home/adminmpi/Downloads/mpich-4.2.1/examples»
make[1]: выход из каталога «/home/adminmpi/Downloads/mpich-4.2.1»
[adminmpi@vt316-12u mpich-4.2.1]$ sudo make install
```

Рисунок 28 – Установка пакета после сборки



```
adminmpi@vt316-12u: /home/adminmpi/Downloads/mpich-4.2.1
Файл Правка Вид Поиск Терминал Помощь
[adminmpi@vt316-12u mpich-4.2.1]$ echo 'export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH' >> ~/.bashrc
[adminmpi@vt316-12u mpich-4.2.1]$ source ~/.bashrc
[adminmpi@vt316-12u mpich-4.2.1]$ mpiexec --version
HYDRA build details:
  Version: 4.2.1
  Release Date: Wed Apr 17 15:30:02 CDT 2024
  CC: gcc
  Configure options: '--disable-option-checking' '--prefix=/usr/local' '--with-hwloc=embedded' '--disable-fortran' '--cache-file=/dev/null' '--srcdir=.' 'CC=gcc' 'CFLAGS=-O2' 'LDFLAGS=' 'LIBS=' 'CPPFLAGS=-DNETMOD_INLINE=__netmod_inline_ofi__ -I/home/adminmpi/Downloads/mpich-4.2.1/src/mpi/include -I/home/adminmpi/Downloads/mpich-4.2.1/modules/hwloc/include -D_REENTRANT -I/home/adminmpi/Downloads/mpich-4.2.1/src/mpi/romio/include -I/home/adminmpi/Downloads/mpich-4.2.1/src/pmi/include -I/home/adminmpi/Downloads/mpich-4.2.1/modules/yaksa/src/frontend/include -I/home/adminmpi/Downloads/mpich-4.2.1/modules/libfabric/include'
  Process Manager: pmi
  Launchers available: ssh rsh fork slurm ll lsf sge manua
  1 persist
  Topology libraries available: hwloc
  Resource management kernels available: user slurm ll lsf sge pbs cobalt
  Demux engines available: poll select
[adminmpi@vt316-12u mpich-4.2.1]$
```

Рисунок 29 – Настройка переменных окружения

После установки утилиты `mpirun` требуется установка дистрибутива Anaconda для создания окружения и управления пакетами для языка программирования Python. Прежде чем установить Anaconda, необходимо скачать установочный скрипт из репозитория при помощи команды `wget` (см. рисунок 30). Затем в интерпретаторе `bash` нужно выпустить выполнение скачанного на предыдущем шаге скрипта (см. рисунок 31).

```
[adminmpi@vt316-12u Downloads]$ wget https://repo.anaconda.com/archive/Anaconda3-2024.10-1-Linux-x86_64.sh
--2026-04-24 12:03:59-- https://repo.anaconda.com/archive/Anaconda3-2024.10-1-Linux-x86_64.sh
Resolving repo.anaconda.com (repo.anaconda.com)... 104.16.32.241, 104.16.191.158, 2606:4700::6810:bf9e, ...
Connecting to repo.anaconda.com (repo.anaconda.com)|104.16.32.241|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1102495056 (1.0G) [application/octet-stream]
Saving to: 'Anaconda3-2024.10-1-Linux-x86_64.sh'

-2024.10-1-Linux-x86 17%[==>] 179.74M 51.3MB/s eta 20s
```

Рисунок 30 – Скачивание установочного скрипта для Anaconda

```
[adminmpi@vt316-12u Downloads]$ bash Anaconda3-2024.10-1-Linux-x86_64.sh

Welcome to Anaconda3 2024.10-1

In order to continue the installation process, please review the license agreement.
Please, press ENTER to continue
>>>
ANACONDA TERMS OF SERVICE
Please read these Terms of Service carefully before purchasing, using, accessing, or downloading any Anaconda Offerings (the "Offerings"). These Anaconda Terms of Service ("TOS") are between Anaconda, Inc. ("Anaconda") and you ("You"), the
```

Рисунок 31 – Запуск установочного скрипта интерпретатором bash

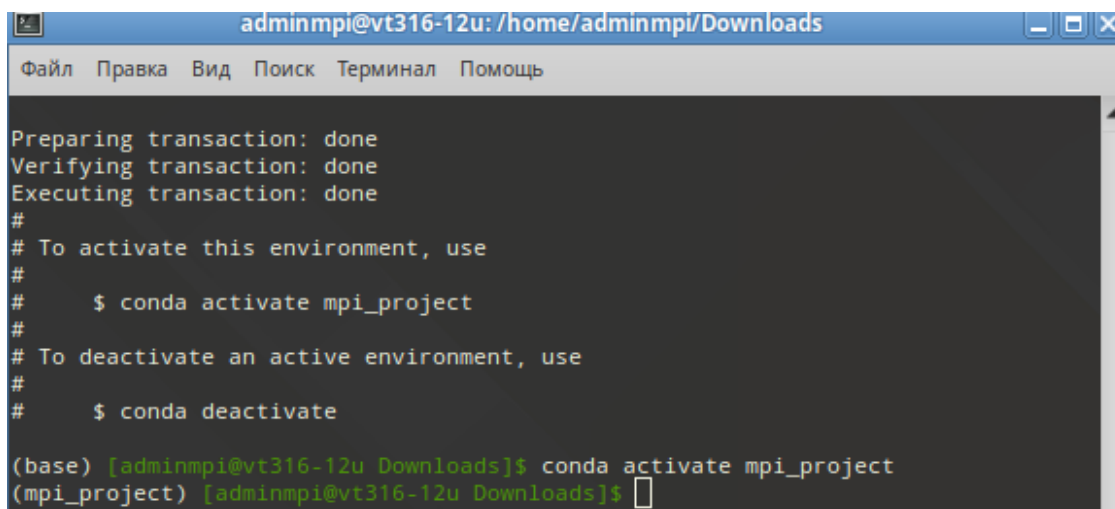
После установки дистрибутива необходимо командой «conda create» создать окружение с указанием его названия и версии используемого языка программирования Python для запускаемого проекта (см. рисунок 32).

```
(base) [adminmpi@vt316-12u Downloads]$ conda create -n mpi_project python=3.9
Channels:
- defaults
Platform: linux-64
Collecting package metadata (repodata.json): \
```

Рисунок 32 – Создание conda-окружения

Затем следует активировать окружение командой «conda activate» (см. рисунок 33). Для того, чтобы conda-окружение автоматически активировалось при логине текущего пользователя, из-под которого производится работа на кластере, нужно модифицировать файл ~/.bashrc

следующим образом: добавить в конец файла команду активации требуемого окружения (см. рисунок 34).

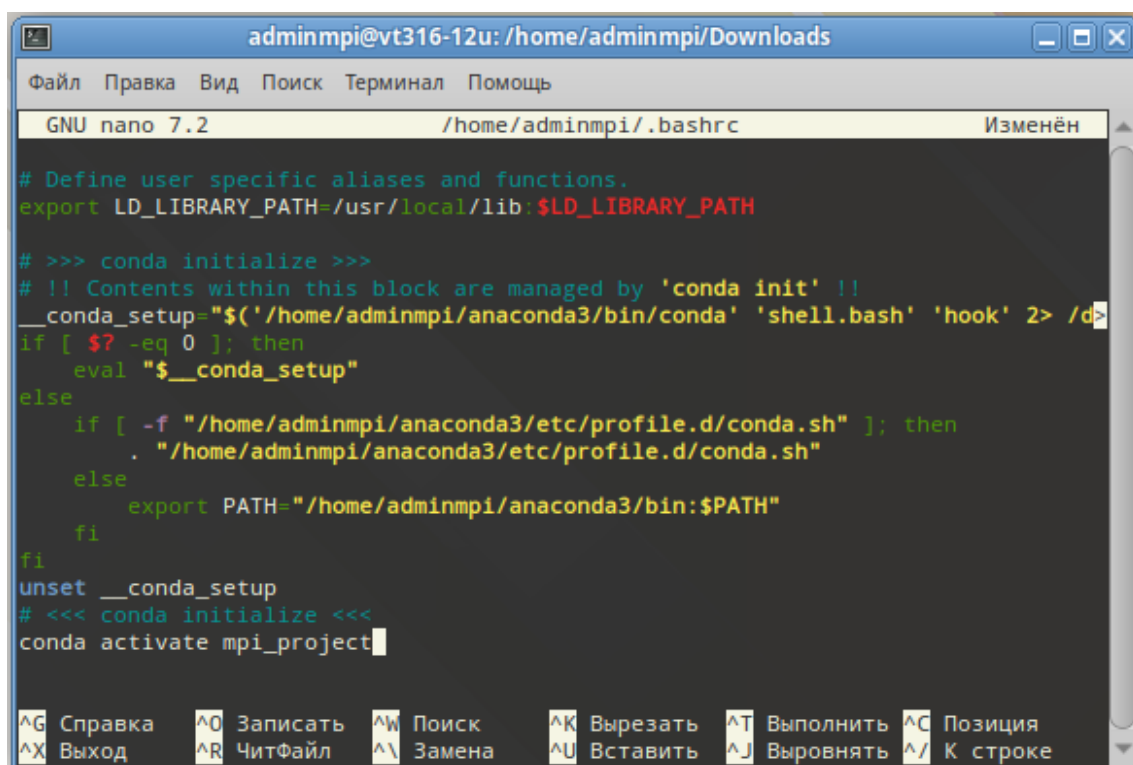


```
adminmpi@vt316-12u: /home/adminmpi/Downloads
Файл Правка Вид Поиск Терминал Помощь

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#   $ conda activate mpi_project
#
# To deactivate an active environment, use
#
#   $ conda deactivate

(base) [adminmpi@vt316-12u Downloads]$ conda activate mpi_project
(mpi_project) [adminmpi@vt316-12u Downloads]$
```

Рисунок 33 – Активация conda-окружения запускаемого проекта



```
adminmpi@vt316-12u: /home/adminmpi/Downloads
Файл Правка Вид Поиск Терминал Помощь
GNU nano 7.2 /home/adminmpi/.bashrc Изменён

# Define user specific aliases and functions.
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH

# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/home/adminmpi/anaconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/home/adminmpi/anaconda3/etc/profile.d/conda.sh" ]; then
        . "/home/adminmpi/anaconda3/etc/profile.d/conda.sh"
    else
        export PATH="/home/adminmpi/anaconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<
conda activate mpi_project
```

Рисунок 34 – Настройки для автоматической активации conda-окружения

После активации окружения нужно установить библиотеки, применяющиеся в текущем проекте: torch, numpy и mpi4py (см. рисунок 35).

```
(mpi_project) [adminmpi@vt316-12u Downloads]$ pip install torch numpy mpi4py
```

Рисунок 35 – Установка библиотек языка Python

Как только `mpirun` и пакеты в Anaconda будут установлены на каждом узле, необходимо командой `which` проверить расположение утилиты `mpirun` и интерпретатора Python (см. рисунки 36 и 37). На всех машинах оно должно быть одинаковым.

```
[adminmpi@vt316-12u mpich-4.2.1]$ which mpirun  
/usr/local/bin/mpirun
```

Рисунок 36 – Расположение утилиты `mpirun`

```
(mpi_project) [adminmpi@vt316-12u Downloads]$ which python  
/home/adminmpi/anaconda3/envs/mpi_project/bin/python  
(mpi_project) [adminmpi@vt316-12u Downloads]$
```

Рисунок 37 – Расположение интерпретатора Python

Перед запуском распределённых вычислений необходимо на каждый узел кластера в один и тот же каталог скопировать программу для обучения алгоритма и данные для неё. Затем, на узле-мастере создаётся файл `hosts.txt`, в котором прописаны `ip`-адреса других узлов (в том числе и свой) и через двоеточие – количество процессов, которое можно на них запустить. Содержание файла `hosts.txt` для текущего примера представлено на рисунке 38.

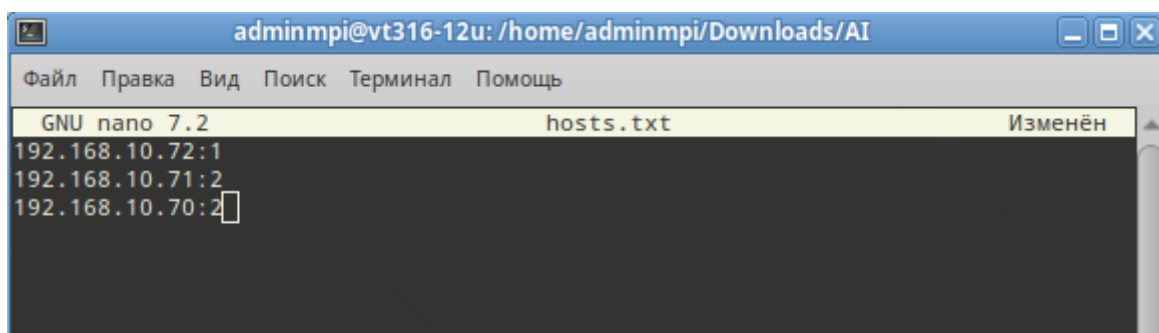


Рисунок 38 – Файл `hosts.txt`

4.3.2 Запуск распараллеленного обучения на вычислительном кластере

После того, как вычислительный кластер был поднят, можно запустить распараллеленное обучение разработанного алгоритма. Для этого, на каждом узле кластера в директории по одному и тому же пути, необходимо положить файл `main.py` и данные об активности пользователей в файле `user_data.json` (см. рисунок 39).

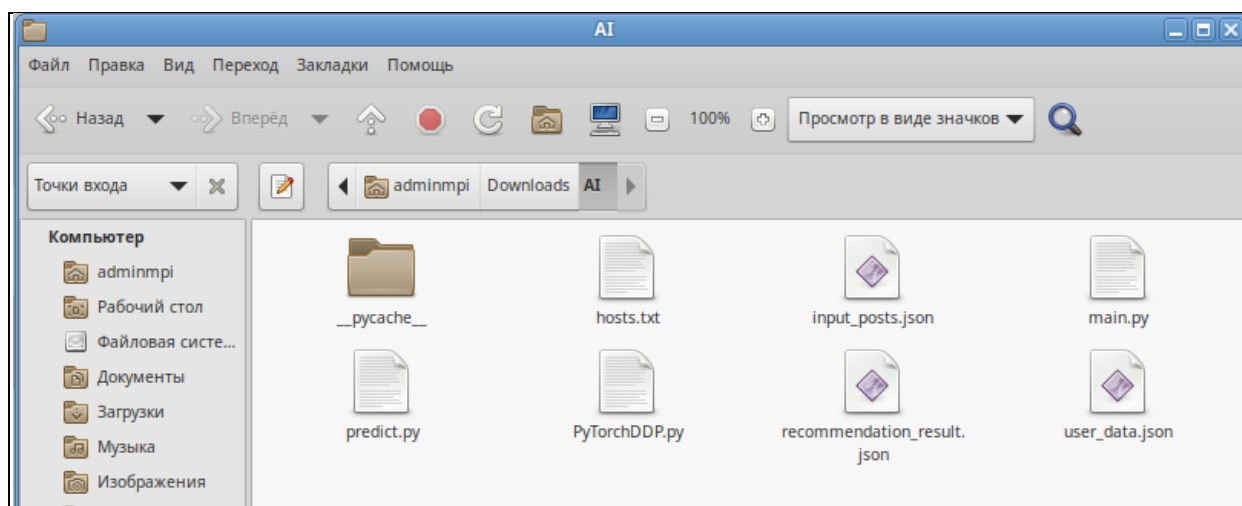


Рисунок 39 – Файлы `main.py` и `user_data.json` в рабочей директории узла-мастера

Для запуска версии с MPI используется пакет `mpiexec`. Формат команды следующий: `mpiexec -f <файл_с_узлами.txt> -np <общее_число_процессов> <интерпретатор_python> <запускаемая_программа.py>`. Пример запуска MPI-версии программы показан на рисунке 40.

```

adminmpi@vt316-12u: /home/adminmpi/Downloads/AI
Файл Правка Вид Поиск Терминал Помощь
(mpi_project) [adminmpi@vt316-12u AI]$ mpiexec -f hosts.txt -np 24 python main.py
[vt316-10u.bt.local:16/24] Процесс запущен
[vt316-10u.bt.local:21/24] Процесс запущен
[vt316-10u.bt.local:22/24] Процесс запущен
[vt316-10u.bt.local:23/24] Процесс запущен
[vt316-10u.bt.local:17/24] Процесс запущен
[vt316-10u.bt.local:18/24] Процесс запущен
[vt316-10u.bt.local:19/24] Процесс запущен
[vt316-10u.bt.local:20/24] Процесс запущен
[vt316-12u.bt.local:6/24] Процесс запущен
[vt316-12u.bt.local:1/24] Процесс запущен
[vt316-12u.bt.local:3/24] Процесс запущен
[vt316-12u.bt.local:4/24] Процесс запущен
[vt316-12u.bt.local:5/24] Процесс запущен
[vt316-12u.bt.local:7/24] Процесс запущен

```

Рисунок 40 – Запуск распараллеленного обучения через MPI

Логирование процесса обучения алгоритма продемонстрировано на иллюстрациях 41 и 42.

```

adminmpi@vt316-12u: /home/adminmpi/Downloads/AI
Файл Правка Вид Поиск Терминал Помощь

1: ЗАГРУЗКА ДАННЫХ
[vt316-11u.bt.local:12/24] Процесс запущен
[vt316-12u.bt.local:0] Загрузка данных из user_data.json
[vt316-11u.bt.local:13/24] Процесс запущен
[vt316-12u.bt.local:0] Загружено 570 пользователей

Сбор частот тегов со всех процессов...

2: ПОСТРОЕНИЕ СЛОВАРЯ
[vt316-12u.bt.local:0] Словарь построен: 390 токенов
[vt316-12u.bt.local:0] - Тегов: 386

3: ПОДГОТОВКА ДАННЫХ
[vt316-12u.bt.local:0] Создано 31302 обучающих примеров
Размер обучающего датасета: 28172
Размер валидационного датасета: 3130

4: СОЗДАНИЕ МОДЕЛИ
Параметров модели: 8,882,566
[vt316-12u.bt.local:5] Получен словарь от мастера, размер 390
[vt316-12u.bt.local:6] Получен словарь от мастера, размер 390
[vt316-10u.bt.local:22] Получен словарь от мастера, размер 390
[vt316-12u.bt.local:4] Получен словарь от мастера, размер 390

```

Рисунок 41 – Процесс обучения

```

adminmpi@vt316-12u: /home/adminmpi/Downloads/AI
Файл Правка Вид Поиск Терминал Помощь

[vt316-12u.bt.local:4] Sampler: replicas=24, rank=4, dataset size=28172
[vt316-12u.bt.local:7] Sampler: replicas=24, rank=7, dataset size=28172
[vt316-10u.bt.local:18] Sampler: replicas=24, rank=18, dataset size=28172
[vt316-10u.bt.local:21] Sampler: replicas=24, rank=21, dataset size=28172
[vt316-11u.bt.local:14] Sampler: replicas=24, rank=14, dataset size=28172
[vt316-11u.bt.local:11] Sampler: replicas=24, rank=11, dataset size=28172
[vt316-11u.bt.local:12] Sampler: replicas=24, rank=12, dataset size=28172
[vt316-11u.bt.local:9] Sampler: replicas=24, rank=9, dataset size=28172

5: РАСПРЕДЕЛЁННОЕ ОБУЧЕНИЕ
Количество процессов: 24
=====
/home/adminmpi/anaconda3/envs/mpi_project/lib/python3.9/site-packages/torch/nn/modules/transformer.py:515: UserWarning: The PyTorch API of nested tensors is in prototype stage and will change in the near future. We recommend specifying layout=torch.jagged when constructing a nested tensor, as this layout receives active development, has better operator coverage, and works with torch.compile. (Triggered internally at /pytorch/aten/src/ATen/NestedTensorImpl.cpp:178.)
  output = torch._nested_tensor_from_mask(
[vt316-12u.bt.local:0] Epoch 1/5, Train Loss: 5.8003, Val Loss: 5.3200
Время эпохи: 32.40с (вычисления: 21.86с, MPI: 4.39с)
[vt316-12u.bt.local:0] Модель сохранена в improved_tag_recommender.pt
[vt316-12u.bt.local:0] Новый лучший результат! Val Loss: 5.3200 (эпоха 1)

```

Рисунок 42 – Процесс обучения

После завершения обучения (см. рисунок 43) в директории с проектом на узле-мастере появится файл `improved_tag_recommender.pt`, содержащий словарь и состояние обученной модели (см. рисунок 44).

```

adminmpi@vt316-12u: /home/adminmpi/Downloads/AI
Файл Правка Вид Поиск Терминал Помощь

=====
СТАТИСТИКА ВРЕМЕНИ ОБУЧЕНИЯ
=====
Общее время работы программы:      0ч 5м 6.47с
Время обучения (все эпохи):        0ч 5м 5.33с

Из них:
[vt316-12u.bt.local:2] Эпоха 5 завершена, train_loss=5.2522
[vt316-12u.bt.local:4] Эпоха 5 завершена, train_loss=5.2783
[vt316-12u.bt.local:6] Эпоха 5 завершена, train_loss=5.2724
Чистые вычисления (forward+backward): 2м 6.71с (41.5%)
Синхронизация MPI (Allreduce):        2м 10.77с (42.8%)
Прочее (загрузка, ожидание):          0м 47.85с (15.7%)

Среднее время эпохи: 61.07с
Число процессов: 24
[vt316-12u.bt.local:1] Эпоха 5 завершена, train_loss=5.2736
[vt316-12u.bt.local:3] Эпоха 5 завершена, train_loss=5.2742
[vt316-12u.bt.local:5] Эпоха 5 завершена, train_loss=5.2584
[vt316-12u.bt.local:7] Эпоха 5 завершена, train_loss=5.2484

Подготовка и сохранение:             0ч 0м 1.14с
=====

```

Рисунок 43 – Окончание обучения

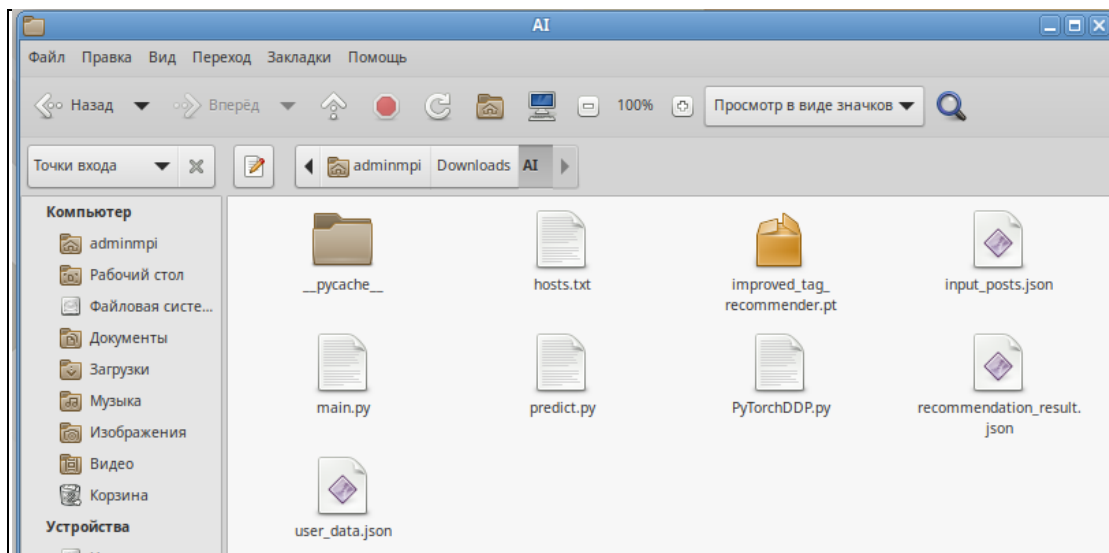


Рисунок 44 – Файл improved_tag_recommender.pt с состоянием модели и словарём

Прежде чем запустить распараллеленное обучение, в основе которого лежит PyTorch DDP, необходимо узнать имя основного сетевого интерфейса узла (см. рисунок 45) и задать его в переменной окружения через export GLOO_SOCKET_IFNAME=<имя_сетевого_интерфейса>.

```
adminmpi@vt316-12u: /home/adminmpi/Downloads/AI
Файл Правка Вид Поиск Терминал Помощь
(mpi_project) [adminmpi@vt316-12u AI]$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 10:ff:e0:46:c0:aa brd ff:ff:ff:ff:ff:ff
    inet 192.168.10.72/24 brd 192.168.10.255 scope global enp3s0
        valid_lft forever preferred_lft forever
    inet6 fe80::12ff:e0ff:fe46:c0aa/64 scope link
        valid_lft forever preferred_lft forever
(mpi_project) [adminmpi@vt316-12u AI]$
```

Рисунок 45 – Имя сетевого интерфейса узла

Для запуска версии, основанной на PyTorch DDP, используется пакет torchrun. Формат команды следующий: torchrun --nnodes=<число_узлов> --nproc-

per-node=<число_процессов_на_узле> --node-rank=<ранк_узла> --master-addr=<ip-адрес_узла-мастера> --master-port=<порт_мастера> <файл_программы.py>. Пример запуска PyTorchDDP-версии программы показан на рисунке 46. Важным отличием является то, что MPI сам запускает процессы через ssh-соединение. Для PyTorch DDP указанную ранее команду запуска необходимо выполнить на всех узлах в сети. Причём, первой выполняется команда на узле-мастере. В результате обучения в директории с проектом на узле-мастере также появится файл improved_tag_recommender.pt.

```

adminmpi@vt316-12u: /home/adminmpi/Downloads/AI
Файл Правка Вид Поиск Терминал Помощь
(mpi_project) [adminmpi@vt316-12u AI]$ export GLOO_SOCKET_IFNAME=enp3s0
(mpi_project) [adminmpi@vt316-12u AI]$ torchrun --nnodes=3 --nproc-per-node=5 --
node-rank=0 --master-addr=192.168.10.72 --master-port=29000 PyTorchDDP.py
W0504 10:02:40.224637 7766 site-packages/torch/distributed/run.py:774]
W0504 10:02:40.224637 7766 site-packages/torch/distributed/run.py:774] *****
*****
W0504 10:02:40.224637 7766 site-packages/torch/distributed/run.py:774] Setting O
MP_NUM_THREADS environment variable for each process to be 1 in default, to avoi
d your system being overloaded, please further tune the variable for optimal per
formance in your application as needed.
W0504 10:02:40.224637 7766 site-packages/torch/distributed/run.py:774] *****
*****
[Gloo] Rank [Gloo] Rank [Gloo] Rank [Gloo] Rank 10 is connected to is connected
to 314214 is connected to peer ranks. is connected to peer ranks. 14Expected
number of connected peer ranks is : 14Expected number of connected peer ranks i
s : peer ranks. 14 peer ranks.
[Gloo] Rank 14Expected number of connected peer ranks is : Expected number of co
nected peer ranks is :
1414
4 is connected to 14 peer ranks. Expected number of connected peer ranks is : 14
[vt316-12u.bt.local:3/15] Процесс запущен
[vt316-12u.bt.local:1/15] Процесс запущен
[vt316-12u.bt.local:4/15] Процесс запущен

```

Рисунок 46 – Запуск обучения

4.3.3 Тестирование работы обученного алгоритма

Полученный в результате обучения файл необходим для работы скрипта predict.py, генерирующего теги для рекомендуемого пользователю поста на основе записей о недавно оценённых им публикаций, описанных в файле input_posts.json. Для грамотной работы скрипта необходимо поместить данный файл в каталог с ним же и запустить программу. В результате в файл recommendation_result.json будут записаны теги, которые также будут

продублированы выводом в терминал. Результат работы скрипта генерации с обученной моделью представлен на рисунках 47-50.

```
=====
ВХОДНЫЕ ДАННЫЕ
=====
Пост 1: ['технологии', 'искусственный интеллект', 'нейросеть']
Пост 2: ['нейросеть', 'обучение', 'данные']
Пост 3: ['технологии', 'робототехника', 'автоматизация']
Пост 4: ['робот', 'умный_дом', 'будущее']
Пост 5: ['технологии', 'программирование', 'код']
Пост 6: ['программирование', 'алгоритм', 'разработка']

=====
ГЕНЕРАЦИЯ РЕКОМЕНДАЦИЙ
=====
/home/sabir/anaconda3/lib/python3.12/site-packages/torch/nn/modules/transformer.py:531:
nd specifying layout=torch.jagged when constructing a nested tensor, as this layout rec
rch/aten/src/ATen/NestedTensorImpl.cpp:178.)
  output = torch._nested_tensor_from_mask(

Рекомендуемые теги для следующего поста: ['программирование', 'квантовые вычисления']

Результат сохранён в recommendation_result.json
```

Рисунок 47 – Результат генерации тегов для рекомендуемого пользователю поста

```
=====
ВХОДНЫЕ ДАННЫЕ
=====
Пост 1: ['sport', 'спорт', 'football', 'футбол']
Пост 2: ['football', 'футбол', 'goal', 'гол']
Пост 3: ['basketball', 'баскетбол', 'nba', 'нба']
Пост 4: ['tennis', 'теннис', 'tournament', 'турнир']
Пост 5: ['boxing', 'бокс', 'fight', 'бой']
Пост 6: ['hockey', 'хоккей', 'puck', 'шайба']
Пост 7: ['skateboarding', 'скейтбординг', 'trick', 'трюк']

=====
ГЕНЕРАЦИЯ РЕКОМЕНДАЦИЙ
=====
/home/sabir/anaconda3/lib/python3.12/site-packages/torch/nn/modules/transformer.py:531:
nd specifying layout=torch.jagged when constructing a nested tensor, as this layo
rch/aten/src/ATen/NestedTensorImpl.cpp:178.)
  output = torch._nested_tensor_from_mask(

Рекомендуемые теги для следующего поста: ['спорт', 'medicine']

Результат сохранён в recommendation_result.json
```

Рисунок 48 – Результат генерации тегов для рекомендуемого пользователю поста

```

=====
ВХОДНЫЕ ДАННЫЕ
=====
Пост 1: ['science', 'наука', 'physics', 'физика']
Пост 2: ['physics', 'физика', 'quantum', 'кванты']
Пост 3: ['biology', 'биология', 'genetics', 'генетика']
Пост 4: ['astronomy', 'астрономия', 'space', 'космос']
Пост 5: ['chemistry', 'химия', 'reaction', 'реакция']
Пост 6: ['psychology', 'психология', 'mind', 'сознание']
Пост 7: ['anthropology', 'антропология', 'culture', 'культура']

=====
ГЕНЕРАЦИЯ РЕКОМЕНДАЦИЙ
=====
/home/sabir/anaconda3/lib/python3.12/site-packages/torch/nn/modules/transformer.py:
nd specifying layout=torch.jagged when constructing a nested tensor, as this layout is not
rch/aten/src/ATen/NestedTensorImpl.cpp:178.)
  output = torch._nested_tensor_from_mask(
Рекомендуемые теги для следующего поста: ['mathematics', 'astronomy']

Результат сохранён в recommendation_result.json

```

Рисунок 49 – Результат генерации тегов для рекомендуемого
пользователю поста

```

=====
ВХОДНЫЕ ДАННЫЕ
=====
Пост 1: ['travel', 'путешествия', 'mountains', 'горы']
Пост 2: ['hiking', 'походы', 'nature', 'природа']
Пост 3: ['beach', 'пляж', 'sea', 'море']
Пост 4: ['diving', 'дайвинг', 'underwater', 'подводный']
Пост 5: ['camping', 'кемпинг', 'tent', 'палатка']
Пост 6: ['excursion', 'экскурсия', 'museum', 'музей']
Пост 7: ['tour', 'тур', 'adventure', 'приключение']

=====
ГЕНЕРАЦИЯ РЕКОМЕНДАЦИЙ
=====
/home/sabir/anaconda3/lib/python3.12/site-packages/torch/nn/modules/transformer.py:
nd specifying layout=torch.jagged when constructing a nested tensor, as this layout is not
rch/aten/src/ATen/NestedTensorImpl.cpp:178.)
  output = torch._nested_tensor_from_mask(
Рекомендуемые теги для следующего поста: ['туризм', 'mountaineering']

```

Рисунок 50 – Результат генерации тегов для рекомендуемого
пользователю поста

5 Методология определения масштабирования системы

Для определения поведения модели при обучении в рамках данной дипломной работы было проведено обучение описанного ранее алгоритма на оборудовании следующей конфигурации:

- количество узлов кластера = 3;
- количество физических ядер процессора каждого узла = 6;
- количество оперативной памяти каждого узла = 32 Гб;
- операционная система Linux Alt Workstation 10.4.

Для отслеживания динамики обучения применялись датасеты с 28172, 50404 и 111987 обучающими примерами.

Произведены замеры времени вычислений и времени синхронизаций на числе процессов, кратном количеству узлов в кластере для равномерной нагрузки на каждый из них: 3, 6, 9, 12, 15, 18, 21 и 24.

Результаты замеров динамики времени синхронных и асинхронных операций при обучении алгоритма, представлены в таблицах 3-5. В таблице 3 продемонстрированы замеры времени синхронных и асинхронных операций при обучении алгоритма на датасете из 28172 обучающих примеров. В таблице 4 – замеры при обучении алгоритма на датасете из 50404 обучающих примеров. В таблице 5 – замеры при обучении алгоритма на датасете из 111987 обучающих примеров.

Таблица 3 – Динамика времени вычислений и синхронизаций

Число процессов	Время вычислений	Время синхронизаций	Число батчей на процесс	%MPI
3	488	147	18	23%
6	252	92	9	27%
9	173	71	6	29%
12	132	58	4	31%
15	109	53	3	33%
18	92	53	3	37%
21	80	89	2	53%
24	107	63	2	37%

Таблица 4 – Динамика времени вычислений и синхронизаций

Число процессов	Время вычислений	Время синхронизаций	Число батчей на процесс	%MPI
3	869	246	36	22%
6	443	141	18	24%
9	306	101	12	25%
12	238	87	9	27%
15	195	75	7	28%
18	166	67	6	29%
21	142	140	5	50%
24	127	131	4	51%

Таблица 5 – Динамика времени вычислений и синхронизаций

Число процессов	Время вычислений	Время синхронизаций	Число батчей на процесс	%MPI
3	1924	522	72	21%
6	984	271	36	22%
9	679	203	24	23%
12	521	168	18	24%
15	437	131	14	23%
18	370	119	12	24%
21	318	278	10	47%
24	280	256	9	48%

5.1 Определение времени вычислений при изменении числа процессов и объёма данных

5.1.1 Определение времени обработки одного примера

Первым шагом в определении времени, затрачиваемого на вычисления, в целом, является вычисление времени, уделяемого на обработку одного примера из предоставленного датасета. Время обработки одного примера $t(p)$ вычисляется по следующей формуле

$$t(p) = \frac{T \times p}{N}, \quad (1)$$

где T – суммарное время вычислений;

p – общее число процессов в вычислительной системе;

N – число обучающих примеров в датасете.

Время обработки одного примера при изменении числа процессов и неизменяемом наборе обучающих примеров, не постоянно и может быть рассчитано при помощи линейной регрессии вида

$$t(p) = a \times p + b,$$

где a , b – изменяемые параметры, вычисляемые для текущего объёма датасета;

p – общее число процессов в вычислительной системе.

В таблице 6 приведены вычисления времени обработки одного примера, соответствующего определённому числу процессов для датасета из 28172 обучающих примеров по формуле (1).

Таблица 6 – Время обработки одного примера $t(p)$ для разного количества процессов p

Число процессов (p)	Общее время обработки (T)	Время обработки одного примера $t(p)$
3	488	0,05197
6	252	0,05367
9	173	0,05527
12	132	0,05623
15	109	0,05804
18	92	0,05878

После определения времени обработки одного примера для известного числа процессов составляется таблица, которая определяет параметры для поиска линейной регрессии, задача которой – определение времени обработки одного примера для последующих количеств процессов (см. таблицу 7).

Таблица 7 – Параметры для нахождения линейной регрессии

Число процессов (p)	Время обработки одного примера $t(p)$	Квадрат числа процессов (p^2)	Произведение времени обработки одного примера и числа процессов
3	0,05197	9	0,1559
6	0,05367	36	0,32202

Число процессов (p)	Время обработки одного примера t(p)	Квадрат числа процессов (p ²)	Произведение времени обработки одного примера и числа процессов
9	0,05527	81	0,49741
12	0,05623	144	0,67471
15	0,05804	225	0,87055
18	0,05878	324	1,05807
Σ 63	Σ 0,33395	Σ 819	Σ 3,57866

Далее составляется система уравнений с суммами из таблицы 7 в качестве коэффициентов:

$$\begin{cases} 819a + 63b = 3,57866 \\ 63a + 6b = 0,33395 \end{cases}$$

Затем, производится поиск параметров a и b методом Крамера.

Для нахождения параметра a необходимо вычислить коэффициент Δ :

$$\Delta = \begin{vmatrix} 819 & 63 \\ 63 & 6 \end{vmatrix},$$

отсюда $\Delta = 945$. Далее определяется коэффициент Δ_a :

$$\Delta_a = \begin{vmatrix} 3,57866 & 63 \\ 0,33395 & 6 \end{vmatrix},$$

следовательно $\Delta_a = 0,43311$. Затем находится сам параметр a как отношение Δ_a к Δ :

$$a = \frac{\Delta_a}{\Delta},$$

отсюда получаем $a = 0,0004583$. Для нахождения параметра b необходимо вычислить коэффициент Δ_b :

$$\Delta_b = \begin{vmatrix} 819 & 3,57866 \\ 63 & 0,33395 \end{vmatrix},$$

отсюда $\Delta_b = 48,04947$. Сам параметр b как отношение Δ_b к Δ :

$$b = \frac{\Delta_b}{\Delta},$$

следовательно $b = 0,050846$.

Отсюда, получена линейная регрессия для определения времени обработки одного примера в зависимости от числа процессов, выглядящая следующим образом

$$t(p) = 0,0004583 \times p + 0,050846, \quad (2)$$

где p – общее число процессов в вычислительной системе.

5.1.2 Определение общего времени вычислений

После нахождения времени, затрачиваемого на обработку одного примера из датасета, можно вычислить время T , затрачиваемое на общие вычисления при заданном количестве процессов.

Опираясь на формулы (1) и (2) из прошлого пункта, получаем

$$T = t(p) \times \frac{N}{p},$$

где T – суммарное время вычислений;

$t(p)$ – время обработки одного примера;

p – общее число процессов в вычислительной системе;

N – число обучающих примеров в датасете.

Следовательно

$$T = \frac{(a \times p + b) \times N}{p}, \quad (3)$$

где T – суммарное время вычислений;

a, b – изменяемые параметры, вычисляемые для текущего объёма датасета;

p – общее число процессов в вычислительной системе;

N – число обучающих примеров в датасете.

При помощи полученной формулы можно произвести вычисление времени обработки данных на известном количестве процессов и сверить его с результатами, полученными при замерах на реальном оборудовании. Сравнение времени вычислений, полученного в результате замеров, с временем, рассчитанным по формуле (3), приведено в таблице 8.

Таблица 8 – Сравнение результатов замеров с расчётами

р (число процессов)	Замеренное общее время вычислений	Вычисленное по формуле (3) общее время вычислений
3	488	490,389
6	252	251,65
9	173	172,071
12	132	132,281
15	109	108,407
18	92	92,4909

5.1.3 Определение времени вычислений при масштабировании системы

Для демонстрации динамики изменения времени вычислений от изменения количества обучающих примеров в датасете и от количества процессов в вычислительной системе построен график, продемонстрированный на рисунке 51.

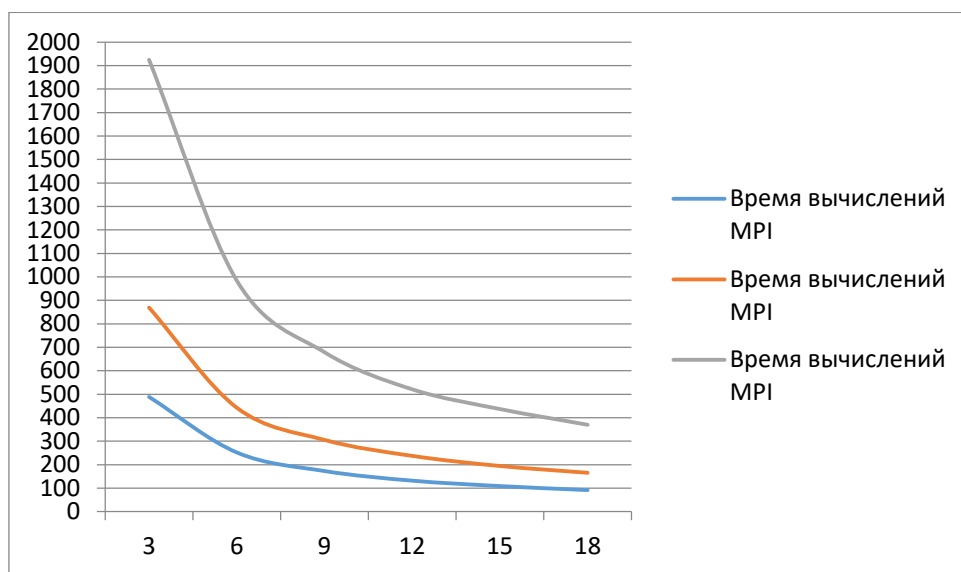


Рисунок 51 – Результаты изменения времени вычислений при изменении размера датасета и числа процессов (Ось Y – время вычислений, Ось X – число процессов)

Опираясь на полученный материал, а также, проанализировав время вычислений на одном и том же количестве процессов для разных объёмов данных, прослеживается следующая динамика: время вычислений на равном

числе процессов изменяется пропорционально изменению объёма набора данных. В таблице 9 представлены коэффициенты изменения времени вычислений при изменении размера датасета при одном и том же числе процессов, где N – размер датасета, T – время вычислений, p – общее число процессов.

Таблица 9 – Коэффициенты изменения времени вычислений

p	T_1	T_2	T_3	T_2/T_1	T_3/T_1	T_3/T_2
3	488	869	1924	1,78	3,94	2,21
6	252	443	984	1,76	3,90	2,22
9	173	306	679	1,77	3,92	2,22
12	132	238	521	1,80	3,95	2,19
15	109	195	437	1,79	4,01	2,24
18	92	166	370	1,80	4,02	2,23

Можно заметить, что отношение размеров датасетов приблизительно равно отношению общего времени, затрачиваемого на обработку, при $N_1 = 28172$, $N_2 = 50404$; $N_3 = 111987$: $N_2/N_1 = 1,79$; $N_3/N_1 = 3,98$; $N_3/N_2 = 2,22$.

Опираясь на вышеописанное, для определения времени вычислений при изменении числа процессов и объёма датасета можно модифицировать формулу (3), добавив туда параметр K , который определяется отношением нового объёма датасета к старому

$$T = K \times \frac{(a \times p + b) \times N}{p}, \quad (4)$$

где T – суммарное время вычислений;

a , b – изменяемые параметры, вычисляемые для текущего объёма датасета;

p – общее число процессов в вычислительной системе;

N – число обучающих примеров в датасете;

K – коэффициент отношения нового объёма датасета к старому;

5.2 Расчёт процента коллективных операций MPI при изменении числа процессов

5.2.1 Определение процента операций синхронизации MPI от общего времени обучения алгоритма

Чтобы определить, какой процент операций занимает синхронизация градиентов через MPI в рамках одного набора данных, необходимо произвести вычисления по данной формуле

$$\%MPI = \frac{T_{\text{вычисл}} + T_{\text{синхр}}}{T_{\text{вычисл}}} \times 100\%, \quad (5)$$

где $\%MPI$ – процент коллективных операций синхронизации;

$T_{\text{вычисл}}$ – общее время вычислений;

$T_{\text{синхр}}$ – общее время коллективных операций синхронизации.

По таблицам 1-3 и формуле (5) построен график, отражающий динамику изменения процента времени синхронизаций MPI в зависимости от числа процессов (см. рисунок 52).

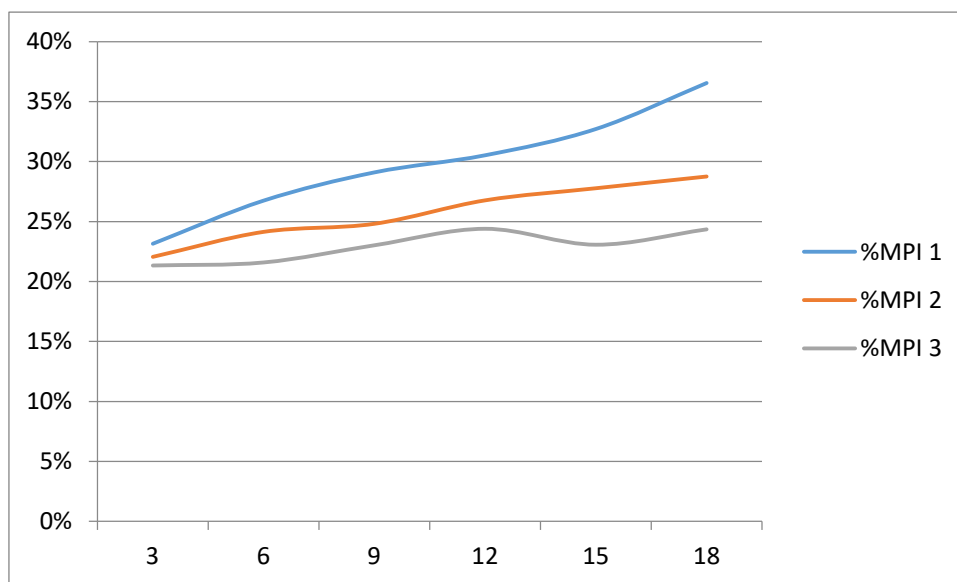


Рисунок 52 – График динамики изменения процента времени синхронизаций от общего времени обучения (Ось Y – процент времени синхронизации от общего времени обучения, Ось X – число процессов)

На представленном графике заметно, что изменение процента времени синхронизаций в рамках одного и того же объёма данных идёт почти линейно. Вследствие чего, для грубой оценки, можно также применить линейную регрессию для определения динамики значений в зависимости от изменения числа процессов.

5.2.2 Определение процента операций синхронизации MPI от времени обучения алгоритма при изменении числа процессов

Как было описано ранее, изменение процента времени синхронизаций через MPI может быть рассчитано при помощи линейной регрессии вида

$$\%MPI(p) = a \times p + b,$$

где $\%MPI$ – процент коллективных операций синхронизации;

a и b – изменяемые параметры, вычисляемые для текущего объёма датасета;

p – общее число процессов в вычислительной системе.

В таблице 10 приведены параметры для расчёта линейной регрессии, определяющей, процент времени синхронизаций через MPI для обучения на датасете из 50404 обучающих примеров.

Таблица 10 – Параметры для нахождения линейной регрессии, определяющей, процент времени синхронизаций через MPI

Число процессов (p)	Процент коллективных операций синхронизации ($\%MPI$)	Квадрат числа процессов (p^2)	Произведение числа процессов и процента коллективных операций синхронизации $p \times \%MPI$
3	22	9	66
6	24	36	144
9	25	81	225
12	27	144	324
15	28	225	420
18	29	324	522
$\Sigma 63$	$\Sigma 155$	$\Sigma 819$	$\Sigma 1701$

Далее составляется следующая система уравнений с суммами из таблицы 10 в качестве коэффициентов:

$$\begin{cases} 819a + 63b = 1701 \\ 63a + 6b = 155 \end{cases}$$

Затем, производится поиск параметров a и b методом Крамера.

Для нахождения параметра a необходимо вычислить коэффициент Δ :

$$\Delta = \begin{vmatrix} 819 & 63 \\ 63 & 6 \end{vmatrix},$$

отсюда $\Delta = 945$. Далее определяется коэффициент Δ_a :

$$\Delta_a = \begin{vmatrix} 1701 & 63 \\ 1555 & 6 \end{vmatrix},$$

следовательно $\Delta_a = 441$. Затем находится сам параметр a как отношение Δ_a к Δ :

$$a = \frac{\Delta_a}{\Delta},$$

отсюда получаем $a = 0,4666667$. Для нахождения параметра b необходимо вычислить коэффициент Δ_b :

$$\Delta_b = \begin{vmatrix} 819 & 1701 \\ 63 & 155 \end{vmatrix},$$

отсюда $\Delta_b = 19782$. Сам параметр b как отношение Δ_b к Δ :

$$b = \frac{\Delta_b}{\Delta},$$

следовательно $b = 20,9333333$.

Отсюда, получена линейная регрессия для определения процента времени синхронизаций через MPI от общего времени обучения для набора данных из 50404 обучающих примеров, выглядящая следующим образом

$$\%MPI(p) = 0,4666667 \times p + 20,9333333. \quad (6)$$

где $\%MPI$ – процент коллективных операций синхронизации;

p – общее число процессов в вычислительной системе.

При помощи полученной формулы можно произвести вычисление процента времени синхронизаций через MPI от общего времени обучения на известном количестве процессов и сверить его с результатами, полученными

при замерах на реальном оборудовании. Результаты вычислений приведены в таблице 11.

Таблица 11 – Сравнение времени синхронизации, полученного в результате замеров, с временем, рассчитанным по формуле (6)

Число процессов (p)	Замеренный процент коллективных операций синхронизации	Рассчитанный по формуле (6) процент коллективных операций синхронизации
3	22	22,3333
6	24	23,7333
9	25	25,1333
12	27	26,5333
15	28	27,9333
18	29	29,3333

5.3 Определение предела масштабирования

Обратив внимание на данные в таблицах 1-3, при числе процессов больше 18, можно заметить, что время синхронизации скачкообразно увеличивается. А также, в некоторых моментах нарушается динамика времени вычислений. Наглядно такое поведение системы отражено на графиках, представленных на рисунках 53-55, где рисунок 53 отражает динамику для таблицы 3, рисунок 54 отражает динамику для таблицы 4, а рисунок 55 – для таблицы 5.

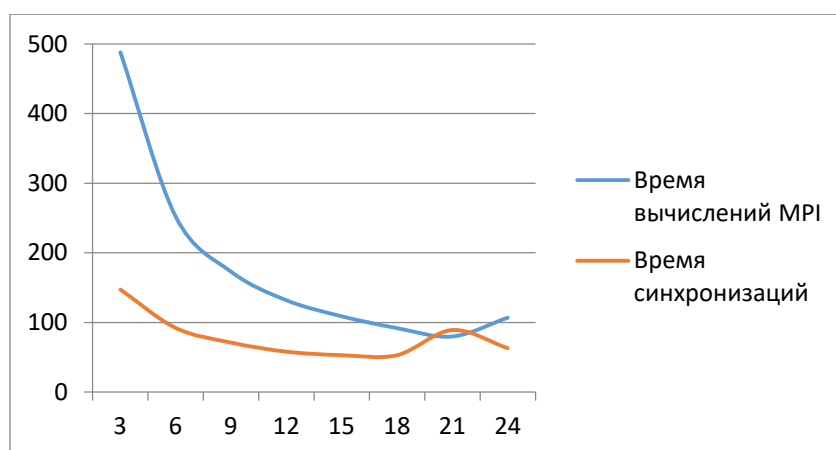


Рисунок 53 – Динамика времени вычислений и времени синхронизаций (Ось Y – время вычислений и синхронизаций, Ось X – число процессов)

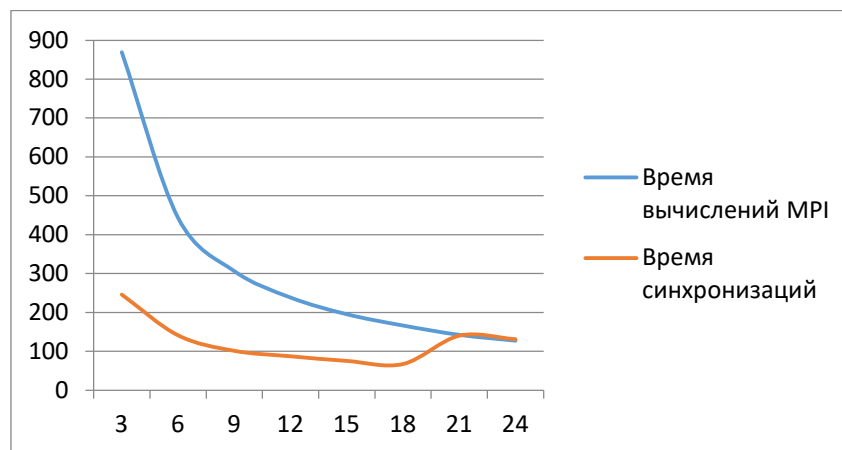


Рисунок 54 – Динамика времени вычислений и времени синхронизаций (Ось Y – время вычислений и синхронизаций, Ось X – число процессов)

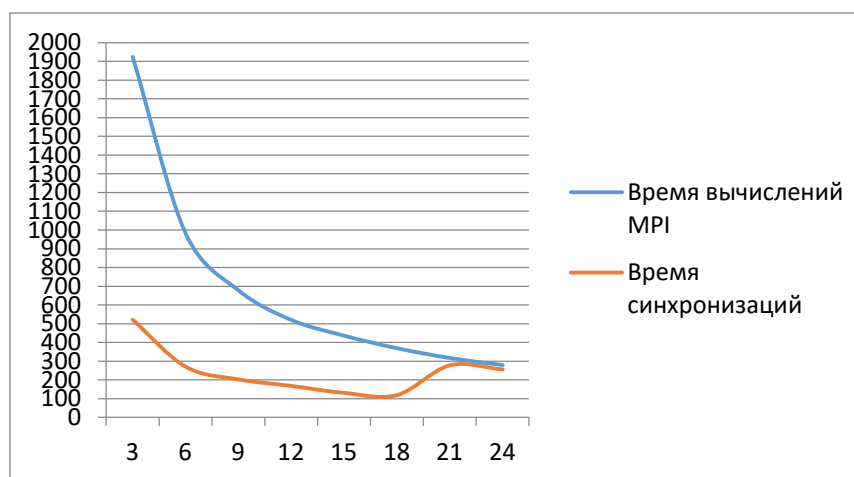


Рисунок 55 – Динамика времени вычислений и времени синхронизаций (Ось Y – время вычислений и синхронизаций, Ось X – число процессов)

Данное поведение связано со спецификой работы подобных вычислительных систем, в которых каждый экземпляр модели запускается на отдельном процессе, получая свою собственную часть данных.

Ранее было описано, что количество узлов в системе равно 3, а число физических ядер процессора на каждой ЭВМ – составляет 6. Следовательно, в текущей конфигурации оборудования безболезненно можно запустить 18 процессов для обучения алгоритма. При превышении данного порога на узлах возникает ситуация, когда число процессов превышает количество физических ядер, из-за чего между ними возникает конкуренция за вычислительные и

сетевые ресурсы. В результате, наблюдается рост накладных расходов на синхронизацию и переключение контекста между процессами при вычислениях.

5.4 Сравнение результатов работы реализованного вручную подхода с результатами при использовании технологии PyTorch DDP

Далее можно сравнить результат работы с модифицированной версией программы. Модифицированная версия использует технологию `torch.nn.parallel`, которая реализует аналогичный подход в распределённом обучении, однако PyTorch DDP использует специальный бекенд GLOO, отвечающий за оптимизацию и организацию распараллеленного обучения, а также выполняет операции синхронизации не явно.

Для отслеживания динамики обучения, как и в первой версии с MPI, применялись датасеты с 28172 и 50404 обучающими примерами. Произведены замеры времени вычислений на числе процессов, кратном количеству узлов в кластере для равномерной нагрузки на каждый из них: 3, 6, 9, 12, 15, 18, 21 и 24. В таблице 12 продемонстрирована динамика времени вычислений при обучении алгоритма на датасете из 28172 обучающих примеров. В таблице 13 – динамика времени вычислений при обучении алгоритма на датасете из 50404 обучающих примеров.

Таблица 12 – Динамика времени вычислений при обучении алгоритма

Число процессов	Время вычислений	Число батчей на процесс
3	205	18
6	261	9
9	180	6
12	123	4
15	94	3
18	94	3
21	74	2
24	76	2

Таблица 13 – Динамика времени вычислений при обучении алгоритма

Число процессов	Время вычислений	Число батчей на процесс
3	366	36
6	460	18
9	295	12
12	247	9
15	190	7
18	160	6
21	159	5
24	161	4

Далее, на рисунках 56 и 57 наглядно показано изменение времени работы обоих вариантов алгоритма в зависимости от числа процессов. Рисунок 56 отражает динамику данных таблицы 12, а рисунок 57 – таблицы 13, соответственно.

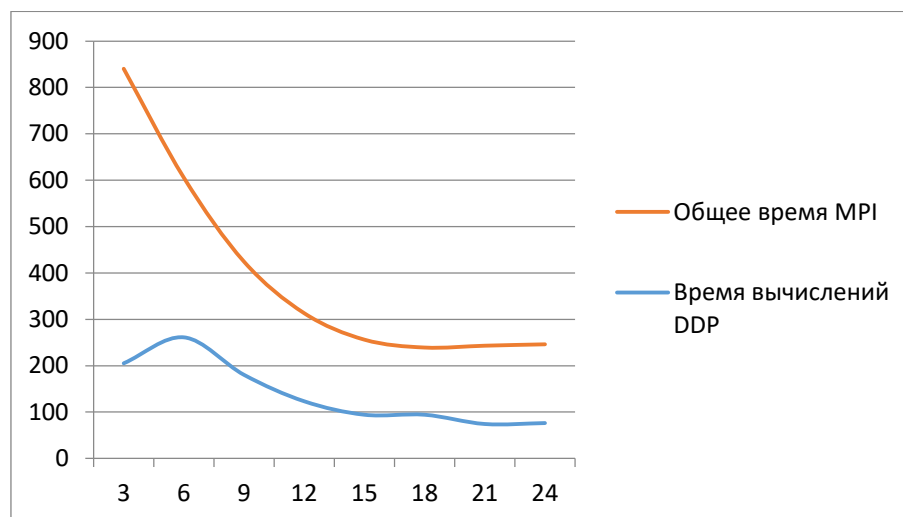


Рисунок 56 – Динамика времени работы обоих вариантов алгоритма на одних и тех же данных (Ось Y – время работы, Ось X – число процессов)

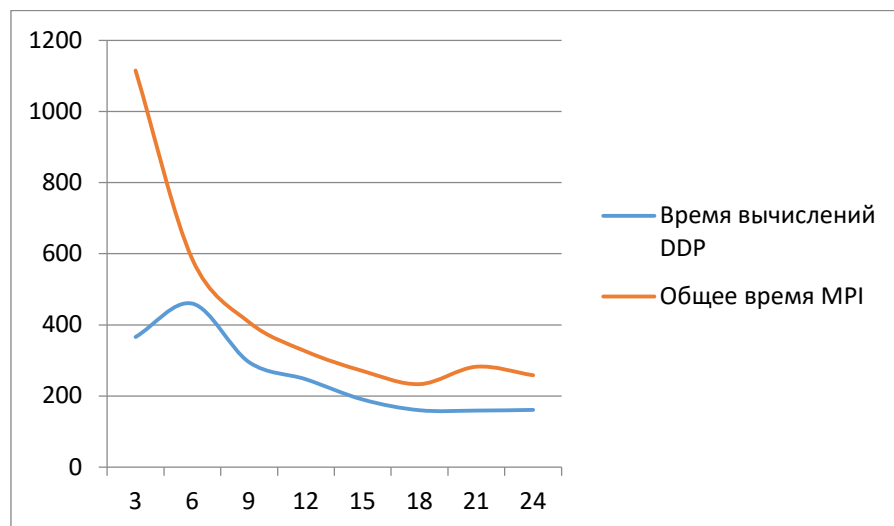


Рисунок 57 – Динамика времени работы обоих вариантов алгоритма на одних и тех же данных (Ось Y – время работы, Ось X – число процессов)

Опираясь на полученные данные, можно прийти к выводу, что версия, основанная на PyTorch DDP, показывает лучший результат по времени обучения, нежели вариант, распараллеленный посредством MPI. Связано это со следующими факторами:

- GLOO автоматически объединяет градиенты в один буфер и выполняет одну операцию Allreduce на группу градиентов, а не на каждый параметр по отдельности;
- GLOO использует перекрытие, стараясь выполнять усреднение градиентов асинхронно, на сколько это возможно.

Здесь также наблюдается аналогичный феномен, касаемо времени вычислений, что и в предыдущем пункте: при достижении 18 процессов, время вычислений не изменяется в лучшую сторону, так как применение встроенного метода из PyTorch для обеспечения распараллеленного обучения алгоритма на параллельных данных, не освобождает вычислительную систему от конкуренции процессов за вычислительные ресурсы узла. В том числе, изменение времени обучения прямо пропорционально изменению объёма данных, что делает возможным прогнозирование затрачиваемых на распараллеленное обучение ресурсов, при $N1 = 28172$ и $N2 = 50404$:

$N2/N1 = 1,79$. В таблице 14 продемонстрированы коэффициенты изменения времени вычислений при изменении размера датасета при одном и том же числе процессов, где N – размер датасета, T – время вычислений, p – общее число процессов.

Таблица 14 – Коэффициенты изменения времени вычислений

p	$T1$	$T2$	$T2/T1$
3	205	366	1,79
6	261	460	1,76
9	180	295	1,64
12	123	247	2,01
15	94	190	2,02
18	94	160	1,70

Также стоит отметить, что хоть бекенд GLOO в связке с технологией PyTorch DDP специально высокооптимизирован для задач машинного обучения – однако, это единственная сфера, в которой он применим. В отличие от MPI – задача которого, написание распределённых программ любого спектра задач, в том числе, с возможностью реализации динамической балансировки нагрузки между процессами.

6 Безопасность жизнедеятельности

В современном мире работа за персональным компьютером стала неотъемлемой частью профессиональной деятельности и повседневной жизни человека. Длительное взаимодействие с вычислительной техникой может приводить к негативным последствиям для здоровья, таким как: перенапряжение глаз, нарушение осанки, туннельный синдром, общее переутомление. Для минимизации подобных рисков разработаны специальные правила.

В данном разделе рассмотрены основные требования к обеспечению безопасной работы за компьютером, организации освещения на рабочем месте и режиму отдыха.

6.1 Техника безопасности при работе с персональным компьютером

При работе с персональным компьютером человек может подвергаться воздействию следующих вредных факторов:

- поражение электрическим током;
- электромагнитное излучение;
- повышенный риск возникновения пожара;
- повышенный уровень шума;
- механические повреждения (при неисправном оборудовании).

Для снижения влияния этих факторов и обеспечения комфортного труда необходимо соблюдать ряд правил при организации рабочего процесса и места.

6.1.1 Организация рабочего места

Рабочее место должно быть оснащено удобной и эргономичной мебелью. Стул рекомендуется выбирать с регулировкой высоты и угла наклона спинки – это позволяет сохранять правильную осанку. Ноги при сидении должны быть

выпрямлены вперёд, ступни полностью стоять на полу. Спина должна иметь опору на спинку стула.

Высота рабочего стола подбирается так, чтобы при работе с клавиатурой и мышью руки сгибались в локтях под углом около 90 градусов. Монитор располагается на расстоянии вытянутой руки. Верхняя часть экрана должна находиться на уровне глаз или чуть ниже, сам монитор рекомендуется слегка наклонять для уменьшения бликов. Такое положение снижает нагрузку на позвоночник, шейный отдел и зрительные органы человека.

6.1.2 Электробезопасность и размещение техники

Не следует использовать компьютер или периферийные устройства при наличии внешних повреждений (трещины на корпусе, оголённые провода). Не рекомендуется оставлять технику включённой на длительное время без присмотра. Для уменьшения воздействия электромагнитного излучения рабочее место целесообразно располагать вдоль стен таким образом, чтобы задняя стенка монитора была обращена к стене (или к соседнему монитору – экраном, а не тыльной частью). Вычислительную и иную электронную технику следует заземлять в соответствии с правилами эксплуатации.

6.2 Требования к освещению

Одна из главных проблем при длительной работе за компьютером – избыточная нагрузка на зрение. Неправильное освещение помещения и неверная настройка яркости монитора могут привести к усталости глаз, сухости, покраснению, временному ухудшению зрительного восприятия и головной боли. Для предотвращения этих явлений необходимо соблюдать следующие требования:

– наличие естественного освещения. По возможности следует использовать естественный свет, который снижает напряжение глаз и улучшает общее самочувствие. При этом необходимо исключать попадание прямых

солнечных лучей на экран монитора – они создают блики и мешают восприятию информации;

– искусственное освещение. Для работы в вечернее или тёмное время суток применяется искусственное освещение (настольные лампы, комнатные светильники и пр.). Освещение должно быть достаточно ярким, чтобы избежать напряжения глаз, но не чрезмерно интенсивным, чтобы не создавать бликов на экране. Рекомендуется комбинировать общее и местное освещение;

– расположение источников света. Локальный источник света (например, настольная лампа) должен устанавливаться сбоку или сзади от монитора для создания грамотного светового баланса, и так чтобы свет не падал прямо на экран и не создавал отражений;

– настройка монитора. Правильная регулировка яркости и контрастности экрана уменьшает утомление глаз. В вечернее время полезно активировать «ночной режим» (тёплые цвета), который снижает нагрузку и не нарушает выработку мелатонина, что положительно сказывается на качестве сна;

Соблюдение этих правил обеспечивает зрительный комфорт и позволяет дольше сохранять работоспособность.

6.3 Режим отдыха и профилактика переутомления

Длительная непрерывная работа за компьютером ведёт к мышечному напряжению, усталости глаз и снижению концентрации внимания. Чтобы избежать негативных последствий от долгой работы, необходимо придерживаться рационального режима труда и отдыха.

– наличие перерывов. При 8-часовой рабочей смене общая продолжительность перерывов должна составлять от 30 до 90 минут. Рекомендуется делать короткие (5–10 минут) перерывы каждый час работы. В это время следует отойти от компьютера, размяться, выполнить несколько упражнений для шеи, плеч, кистей рук и спины – это улучшает кровообращение и снимает мышечное напряжение;

– гимнастика для глаз. Полезно выполнять специальные упражнения: круговые движения открытыми глазами, частые моргания, смену фокусировки взгляда с близкого предмета (например, кончик ручки) на дальний (объект за окном или в дальнем углу комнаты). Такая зарядка предотвращает развитие синдрома «сухого глаза» и возникновение зрительной усталости;

– график работы и отдыха. Следует избегать систематических переработок, соблюдать чёткое расписание и планировать задачи для оптимизации времени. Работа в дни отдыха и постоянные сверхурочные негативно сказываются как на физическом, так и на ментальном здоровье, увеличивая риск хронической усталости, нарушения сна и снижения иммунитета;

– профилактика зависимости от устройств. Чрезмерное использование смартфонов и персональных компьютеров вне выполнения рабочих обязанностей может приводить к формированию зависимости, нарушению сна и снижению концентрации внимания. Поэтому рекомендуется ограничивать время вне рабочего использования гаджетов, в особенности перед сном.

Соблюдение перечисленных требований к рабочему месту, освещению и режиму отдыха позволяет существенно снизить риск профессиональных заболеваний, сохранить высокую работоспособность и обеспечить комфортные и безопасные условия труда для разработчиков программных продуктов и обычных пользователей цифровой техники.

7 Стартап проект

В качестве дальнейшего развития идеи, реализованной в рамках выпускной квалификационной работы, планируется реализовать собственную платформу для поддержки социальной и творческой активности её будущих пользователей.

Ключевым моментом будет переработка реализованного алгоритма: задача распараллеленного машинного обучения будет полностью делегирована варианту программы, реализованному на PyTorch DDP, с последующим внедрением бекенда NCCL при условии обеспечения сервиса видеоускорителями от компании Nvidia; распределённая вычислительная система, написанная с применением технологии MPI, сместит свой фокус в обучении нейронной сети с обучения алгоритма, в целом, на анализ и обработку больших данных, с составлением аналитики и высокопроизводительными вычислениями для последующей исследовательской работы.

Для работы сервиса планируется реализация Web-клиента, а также мобильных приложений под операционные системы Android и iOS. Обеспечение работы алгоритма и backend-части приложений ляжет на ЦОД, основу которого составит вычислительный кластер в связке с сервером.

По достижении необходимого уровня качества и производительности алгоритма, его лицензированные копии могут распространяться на коммерческой основе. А также, при получении должного финансирования на масштабирование мощностей собственного ЦОД, возможна организация предоставления его мощностей на договорной основе для развёртывания и работы алгоритма, лицензия которого приобретена заказчиком.

Целевой аудиторией данного проекта будут являться малые и средние стартапы, а также отдельные энтузиасты, реализующие собственные проекты по созданию социальных сетей, модель которых определена их потребностями, которые требуют гибкий рекомендательный алгоритм, без необходимости чрезмерных финансовых и временных затрат на его разработку.

					<i>ПГУ1.090301.40.001 ПЗ</i>	Лист
						85
Изм.	Лист	№ докум.	Подпись	Дата		

7.1 Сегменты целевой аудитории

Как было отмечено ранее: целевой аудиторией проекта являются малые и средние стартапы, а также отдельные энтузиасты, реализующие собственные проекты по созданию социальных сетей, которым может потребоваться развёртывание рекомендательной модели без необходимости осуществления крупных финансовых и временных затрат на её разработку. Прежде чем составить, коммерческое предложение для каждого из сегментов целевой аудитории, необходимо определить их потребности.

Малые и средние стартапы, как правило, имеют некоторое внешнее финансирование своей деятельности, что во многих случаях уже подразумевает наличие у них личных или арендованных серверных мощностей и хостингов. Их может заинтересовать приобретение лицензионной копии алгоритма и технического сопровождения для развёртывания распределённой вычислительной системы, вместо поиска и расширения штата сотрудников для реализации собственного решения.

Что же касается энтузиастов, разрабатывающих проект в одиночку и имеющих из финансирования только личные средства, то для них сформировано следующее решение: будет предложена аренда части ЦОД, на которой развёрнут экземпляр алгоритма рекомендаций, и предоставлен к нему API. Это позволит в рамках ограниченного финансового и аппаратного обеспечения заказчика не покупать отдельную лицензию и оборудование.

7.2 Ключевые процессы

Далее представлены ключевые процессы при разработке стартап-проекта:

- создание команды.
- поиск средств.
- разработка платформы и развёртывание вычислительных мощностей.
- поиск покупателей продукта.
- поддержка программного продукта.

При составлении команды, на начальном этапе необходимо наличие персонала следующих спецификаций: разработчики, бухгалтер, менеджер по продажам. Разработчики набраны из числа единомышленников. Позиции бухгалтера и менеджера по продажам могут быть закрыты при помощи специализированных сайтов по поиску работы и найма сотрудников.

Процесс реализации алгоритма рекомендаций и развёртывания кластера продемонстрирован в пункте 4.

Поиск покупателей продукта может осуществляться посредством внедрения рекламных интеграций и специализированного раздела на собственной web-платформе, с последующей обработкой поступивших откликов от заказчиков менеджером по продажам, который в условиях плотной коммуникации с командой разработчиков способен сформировать грамотное коммерческое предложение для покупателя программного продукта.

7.3 Каналы взаимодействия

Для эффективной демонстрации работы алгоритма рекомендаций нужно разработать собственную платформу, выполняющую функции социальной сети, и использующую данный алгоритм для обеспечения собственной работоспособности. Охват и монетизация подобной платформы будут лучшей рекламой продукта для потенциальных заказчиков и партнёров.

На разработанном ресурсе в специальном разделе будет располагаться контактная информация, для связи с поддержкой и менеджером по продажам, при помощи которой будет осуществляться дистанционная коммуникация с партнёрами или потенциальными заказчиками, а также координация личных встреч для презентации продукта и подписания договоров, при наличии такой необходимости.

7.4 Источники дохода

Продукт будет иметь следующие источники дохода:

					ПГУ1.090301.40.001 ПЗ	Лист
						87
Изм.	Лист	№ докум.	Подпись	Дата		

- партнёрская программа для продажи места под рекламные интеграции на реализованной платформе малому и среднему бизнесу, а также самозанятым;
- продажа лицензионных копий алгоритма и услуг по его установке на распределённую вычислительную систему;
- аренда выделенных мощностей ЦОД заказчиком для развёртывания алгоритма;
- доработка алгоритма под нужды заказчика.

7.5 Ключевые ресурсы

Для запуска и сопровождения стартап проекта потребуются следующие ключевые ресурсы:

- разработчики ПО для iOS и Android, разработчики web-клиента и серверной части, специалист по машинному обучению и обработке больших данных;
- менеджер по продажам и бухгалтер, решающие финансовые и организационные вопросы в проекте;
- техническое оборудование: компьютеры, сервер и программное обеспечение;
- финансовые ресурсы.

Финансирование стартап-проекта планируется осуществить как за счет собственных средств, так и за счет государственной программы «Гранты для молодых предпринимателей».

В таблицах 15 и 16 отображены плановые финансовые затраты на обеспечение ключевых ресурсов проекта.

Таблица 12 – Затраты на материальные ресурсы

Направление использования средств	Цена, руб.	Сумма, руб
Аренда хостинга и регистрация доменного имени (6 мес.)	1000	6 000
Закупка комплектующих для сервера	150 000	150 000
Закупка комплектующих для ЦОД	300 000	300 000
Аренда помещения для серверов (6 мес)	60 000	60 000
Комунальные услуги (6 мес)	10 000	60 000
Закупка персональных компьютеров	50 000	100 000

Таблица 13 – Затраты на людские ключевые ресурсы

Направление использования средств	Кол-во	Цена, руб.	Сумма, руб
З/п разработчику (6 мес)	3	15 000	270 000
З/п бухгалтеру (6 мес)	1	30 000	180 000
З/п менеджеру (6 мес)	1	30 000	180 000

7.6 Риски при реализации проекта

Прежде чем приступить к реализации стартап-проекта, требуется провести анализ рисков, которые могут негативно повлиять на его реализацию и сопровождение.

К таким рискам можно отнести: низкие темпы закупки рекламных интеграций в демонстративном продукте, недостаточный спрос на предлагаемое решение, проблемы с финансированием проекта, технические трудности при внедрении проекта, проблемы с доступностью требуемого оборудования и низкая стабильность сложившейся экономической ситуации.

Заключение

В результате выполнения выпускной квалификационной работы был разработан и обучен алгоритм рекомендаций для социальных сетей, в основе которого лежит принцип генерации тегов, назначаемых на рекомендуемый пользователю пост в зависимости от недавно оценённых им публикаций. А также модель машинного обучения Transformer, позволяющая грамотно анализировать контекст выдачи не только интересуемых пользователя тем, но и смежных с ними областей.

В качестве стратегии обучения был реализован «Data Parallelism» при помощи 2-х подходов: разработка распараллеленной системы обучения, обмен сообщениями между процессами в которой осуществляется посредством технологии MPI; применение готового решения PyTorch DDP и специализированного бекенда GLOO, которые берут на себя задачу оптимизации распараллеленных вычислений и коммуникацию между процессами.

В рамках выпускной квалификационной работы был поднят вычислительный кластер из 3-х ЭВМ, объединённых в сеть, с установленной на них операционной системой Linux Alt Workstation 10.4, на котором проведено распараллеленное обучение алгоритма рекомендаций социальных сетей с применением и последующим сравнением скорости работы обоих подходов. В результате проведена аналитика полученных данных, выявлены закономерности и предложена методология для определения масштабируемости вычислительной системы для распределённого машинного обучения.

В дальнейшей перспективе возможно улучшение разработанной системы для распределённого обучения алгоритма рекомендаций социальных сетей в трёх направлениях:

а) при добавлении в алгоритм модулей, не связанных с машинным обучением, но выполняющих иную аналитическую или вычислительную

работу, возможна доработка программной реализации с применением MPI в сторону: внедрения динамической балансировки нагрузки на узлы вычислительного кластера; использования асинхронных коллективных операций вместо барьерной синхронизации процессов; и установки специальных сетей InfiniBand, рассчитанных специально на объединение узлов кластера;

б) при переходе на вычислительную систему, основным вычислительным модулем которой является один или несколько GPU компании Nvidia, можно с минимальными затратами программно модифицировать реализацию алгоритма рекомендаций, использующую для распараллеленного обучения технологию PyTorch DDP и бекенд GLOO, на применение бекенда NCCL для более эффективного развёртывания вычислений на видеокартах;

с) при необходимости, можно объединить оба предыдущих примера а и д, для достижения максимальной эффективности при комбинировании задач, которое обязательно возникнет в дальнейшем, при увеличении числа данных, усовершенствовании и дополнении алгоритма.

Список использованных источников

1. Макмахан Брайан, Рао Делип. Знакомство с PyTorch: глубокое обучение при обработке естественного языка. — СПб.: Питер, 2020. — 256 с.: ил. — (Серия «Бестселлеры O'Reilly»)
2. habr.com/ru: [Электронный ресурс]. URL: <https://habr.com/ru/articles/753276/> (дата обращения: 10.03.2026)
3. notissimus.com: [Электронный ресурс]. URL: <https://notissimus.com/25-frejmworkov-python-kotorymi-nuzhno-ovladet-v-2023-godu> (дата обращения: 11.03.2026)
4. Бёрнс Б. Распределенные системы. Паттерны проектирования. — СПб.: Питер, 2019. — 224 с.: ил. — (Серия «Бестселлеры O'Reilly»)
5. Гафаров Ф.М. Нейронные сети в PyTorch / Ф.М. Гафаров, А.Ф. Гилемзянов. — Казань: Казанский федеральный университет, 2024. — 106 с
6. habr.com/ru: [Электронный ресурс]. URL: <https://habr.com/ru/companies/mws/articles/770202/> (дата обращения: 06.04.2024)
7. neerc.ifmo.ru: [Электронный ресурс]. URL: <https://neerc.ifmo.ru/wiki/index.php?title=%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D1%84%D0%BE%D1%80%D0%BC%D0%B5%D1%80> (дата обращения: 06.04.2024)
8. mpi4py.readthedocs.io: [Электронный ресурс]. URL: <https://mpi4py.readthedocs.io/en/stable/> (дата обращения: 03.03.2026)
9. docs.pytorch.org: [Электронный ресурс]. URL: <https://docs.pytorch.org/docs/2.5/notes/ddp.html#torchdynamo-ddpoptimizer> (дата обращения: 07.04.2026)
10. systems-analysis.ru: [Электронный ресурс]. URL: https://systems-analysis.ru/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_Transformer (дата обращения: 29.03.2026)

Приложение А

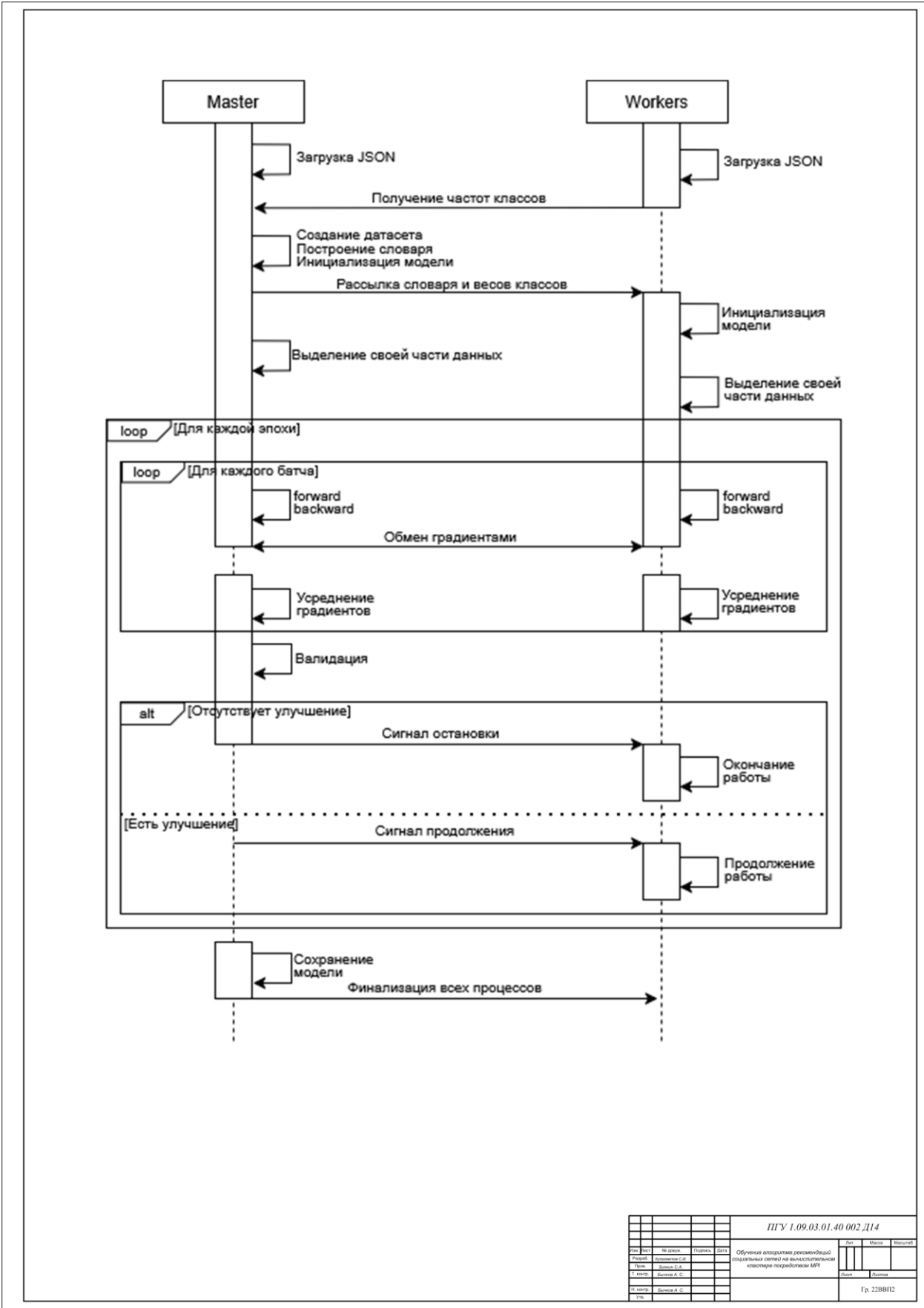
(обязательное)

Обучение алгоритма рекомендаций социальных сетей на вычислительном
кластере посредством MPI

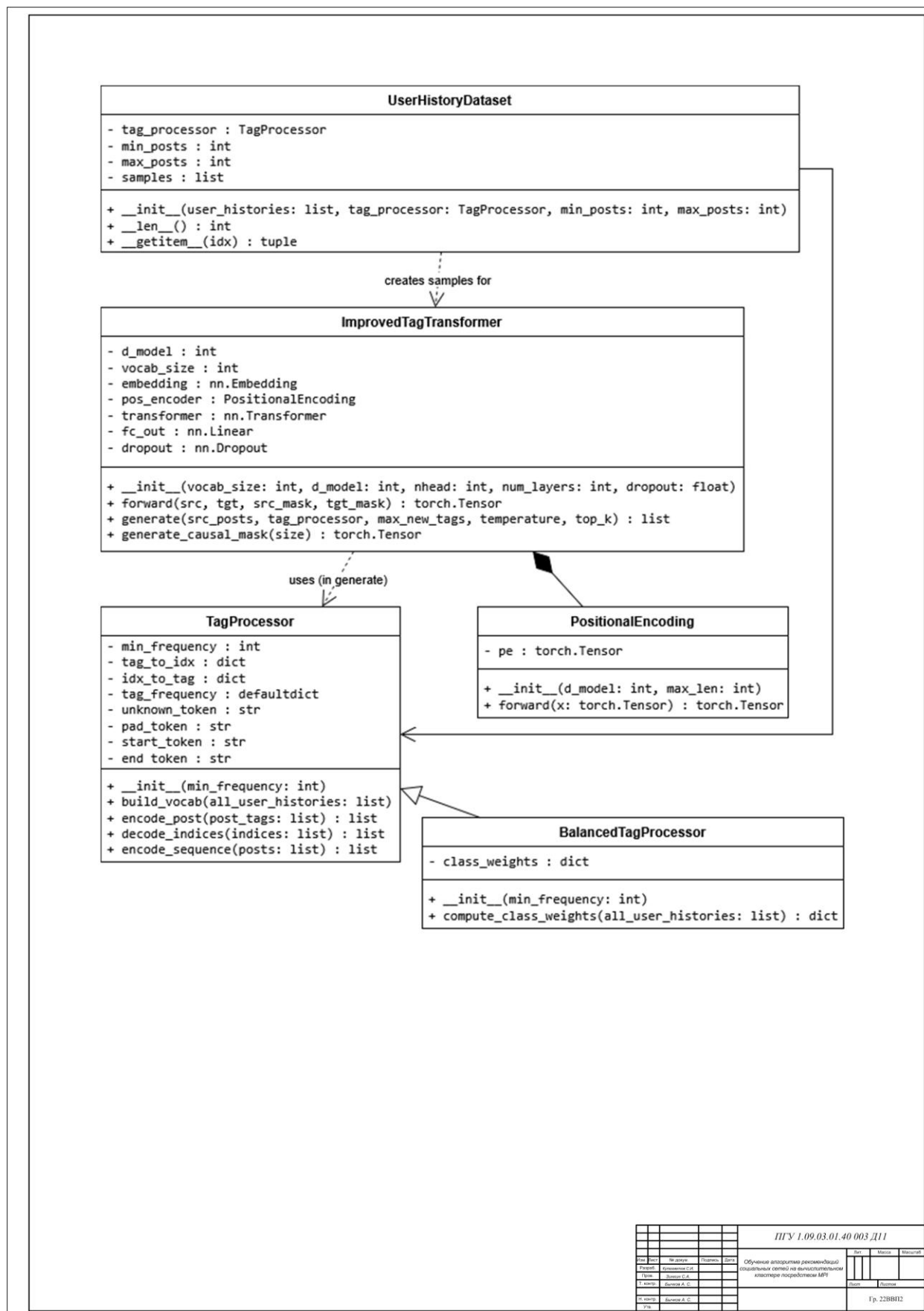
UML– диаграммы

					ПГУ1.090301.40.001 ПЗ	Лист
						93
Изм.	Лист	№ докум.	Подпись	Дата		

Приложение А.1-Диаграмма последовательности



Приложение А.2-Диаграмма классов модуля main.py



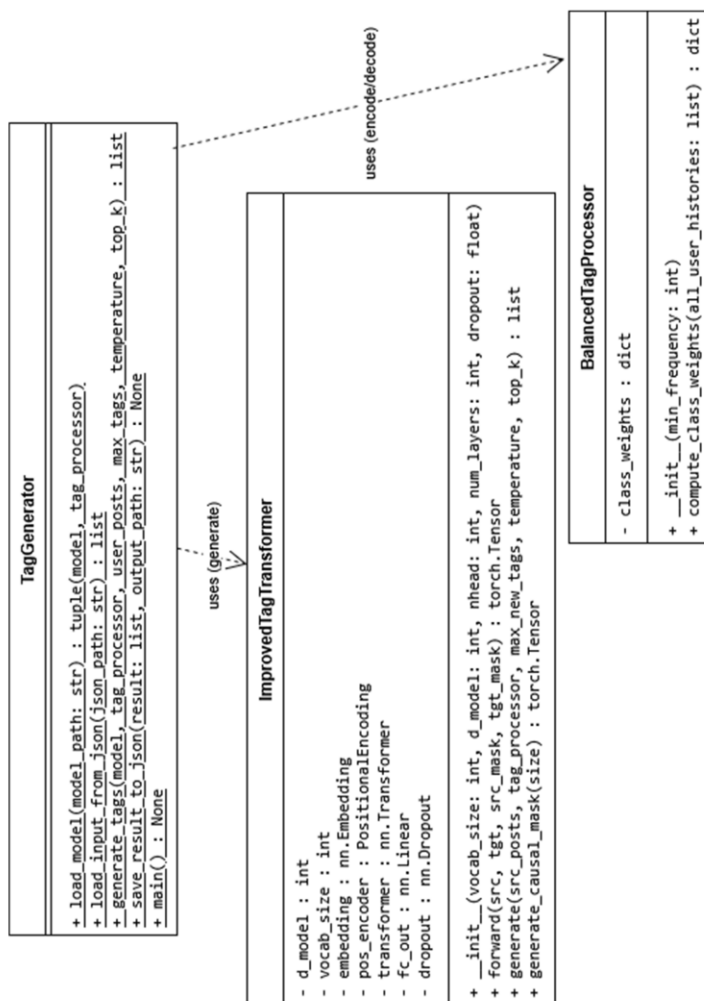
				ПГУ 1.09.03.01.40.003 Д11			
Имя	Фамилия	Имя	Фамилия	Имя	Фамилия	Имя	Фамилия
Степанов	Александр С.С.	Павлов	Д.И.	Обучение алгоритма рекомендаций социальных сетей на вычислительном кластере персональных ИИУ			
Иван	Александр С.С.	Васильев	А.С.				
И.И.И.	Васильев А.С.			Гр. 2288012			

Изм.	Лист	№ докум.	Подпись	Дата

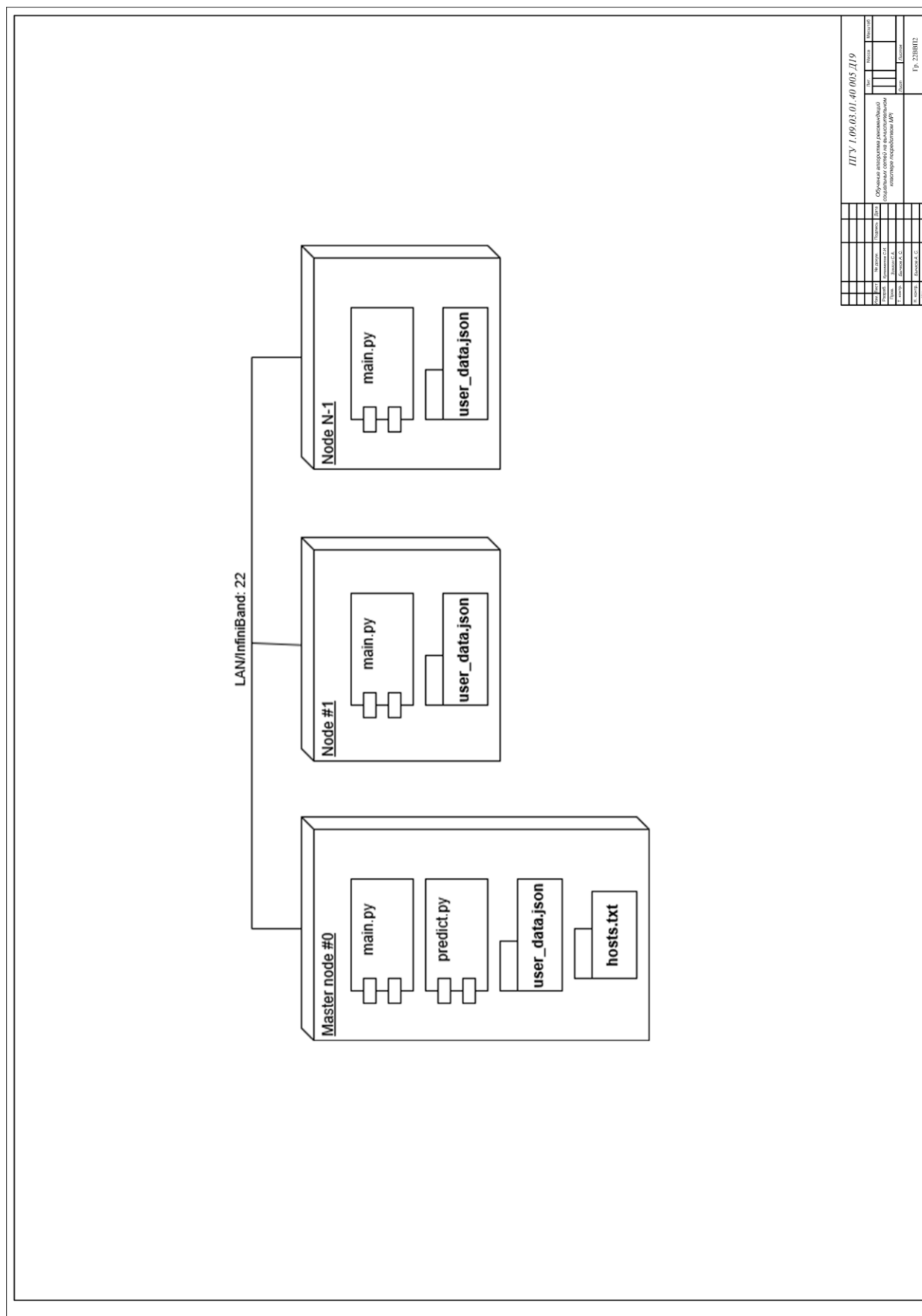
ПГУ1.090301.40.001 ПЗ

Лист
95

ПриложениеА.3-Диаграмма классов модуля predict.py

[illegible]

Приложение А.4-Диаграмма развертывания

[illegible]

Приложение Б

(обязательное)

Обучение алгоритма рекомендаций социальных сетей на вычислительном
кластере посредством MPI

Листинг программы

					ПГУ1.090301.40.001 ПЗ	Лист
						98
Изм.	Лист	№ докум.	Подпись	Дата		

Файл «main.py»

```
import time
import math
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, Subset
from torch.utils.data.distributed import DistributedSampler
from collections import defaultdict
import random
import numpy as np
import json
import os
import socket
from mpi4py import MPI

# =====
# ИНИЦИАЛИЗАЦИЯ MPI
# =====

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
world_size = comm.Get_size()
is_master = (rank == 0)
hostname = socket.gethostname()

if is_master:
    start_time = time.time()
    print(f"[{hostname}:{rank}] Начало выполнения программы: {time.strftime('%Y-%m-%d %H:%M:%S')}", flush=True)

print(f"[{hostname}:{rank}/{world_size}] Процесс запущен", flush=True)

# =====
# 1. БАЗОВЫЙ ОБРАБОТЧИК ТЕГОВ
# =====

class TagProcessor:
    """
    Обрабатывает теги, включая новые, которых не было в обучении
    """
    def __init__(self, min_frequency=2):
        self.min_frequency = min_frequency
        self.tag_to_idx = {}
        self.idx_to_tag = {}
        self.tag_frequency = defaultdict(int)
        self.unknown_token = "<UNK>"
        self.pad_token = "<PAD>"
        self.start_token = "<START>"
        self.end_token = "<END>"
```

```

def build_vocab(self, all_user_histories):
    """
    Строит словарь на основе всех историй реакций пользователей
    на посты
    """
    # Сбор частоты тегов
    for history in all_user_histories:
        for post in history:
            for tag in post:
                self.tag_frequency[tag] += 1

    # Создание словаря
    special_tokens = [self.pad_token, self.start_token,
self.end_token, self.unknown_token]
    for i, token in enumerate(special_tokens):
        self.tag_to_idx[token] = i
        self.idx_to_tag[i] = token

    # Добавление частых тегов
    idx = len(special_tokens)
    for tag, freq in sorted(self.tag_frequency.items(),
key=lambda x: x[1], reverse=True):
        if freq >= self.min_frequency:
            self.tag_to_idx[tag] = idx
            self.idx_to_tag[idx] = tag
            idx += 1

    if is_master:
        print(f"[{hostname}:{rank}] Словарь построен:
{len(self.tag_to_idx)} токенов", flush=True)
        print(f"[{hostname}:{rank}] - Тегов:
{len(self.tag_to_idx) - 4}", flush=True)

def encode_post(self, post_tags):
    """
    Превращает пост в индексы (неизвестные теги → UNK)
    """
    return [self.tag_to_idx.get(tag,
self.tag_to_idx[self.unknown_token])
            for tag in post_tags]

def decode_indices(self, indices):
    """
    Превращает индексы обратно в теги
    """
    tags = []
    for idx in indices:
        tag = self.idx_to_tag.get(idx, self.unknown_token)
        if tag not in [self.pad_token, self.start_token,
self.end_token]:
            tags.append(tag)

```

```

        return tags

def encode_sequence(self, posts):
    """
    Превращает последовательность постов в один плоский список
индексов
    """
    encoded = []
    for post in posts:
        encoded.extend(self.encode_post(post))
    return encoded

# =====
# 2. РАСШИРЕННЫЙ ОБРАБОТЧИК С ВЕСАМИ
# =====

class BalancedTagProcessor(TagProcessor):
    """Расширенный процессор с весами для редких тегов"""

    def __init__(self, min_frequency=1):
        super().__init__(min_frequency)
        self.class_weights = None

    def compute_class_weights(self, all_user_histories):
        """Вычисляет веса для борьбы с дисбалансом классов"""
        # Сбор частоты тегов
        for history in all_user_histories:
            for post in history:
                for tag in post:
                    self.tag_frequency[tag] += 1

        # Вычисление весов (обратная частота)
        total = sum(self.tag_frequency.values())
        num_classes = len(self.tag_frequency)

        self.class_weights = {}
        for tag, freq in self.tag_frequency.items():
            # Вес = общее_количество / (количество_классов *
частота_тега)
            weight = total / (num_classes * freq)
            self.class_weights[tag] = min(weight, 5.0) #
Ограничение максимальным весом 5

        return self.class_weights

# =====
# 3. ПОЗИЦИОННОЕ КОДИРОВАНИЕ
# =====

class PositionalEncoding(nn.Module):

```

```

def __init__(self, d_model, max_len=500):
    super().__init__()
    pe = torch.zeros(max_len, d_model)
    position = torch.arange(0, max_len,
dtype=torch.float).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, d_model, 2).float() *
(-math.log(10000.0) / d_model))
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    self.register_buffer('pe', pe)

def forward(self, x):
    return x + self.pe[:x.size(1)]

# =====
# 4. ОСНОВНАЯ МОДЕЛЬ TRANSFORMER
# =====

class ImprovedTagTransformer(nn.Module):
    def __init__(self, vocab_size, d_model=128, nhead=4,
num_layers=3, dropout=0.3):
        super().__init__()

        self.d_model = d_model
        self.vocab_size = vocab_size

        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_encoder = PositionalEncoding(d_model)

        self.transformer = nn.Transformer(
            d_model=d_model,
            nhead=nhead,
            num_encoder_layers=num_layers,
            num_decoder_layers=num_layers,
            batch_first=True,
            dropout=dropout
        )

        self.fc_out = nn.Linear(d_model, vocab_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src, tgt, src_mask=None, tgt_mask=None):
        src_emb = self.embedding(src) * math.sqrt(self.d_model)
        tgt_emb = self.embedding(tgt) * math.sqrt(self.d_model)

        src_emb = self.pos_encoder(src_emb)
        tgt_emb = self.pos_encoder(tgt_emb)

        src_emb = self.dropout(src_emb)
        tgt_emb = self.dropout(tgt_emb)

```

```

        output = self.transformer(
            src_emb, tgt_emb,
            src_key_padding_mask=src_mask,
            tgt_key_padding_mask=None,
            memory_key_padding_mask=src_mask,
            tgt_mask=tgt_mask
        )

        return self.fc_out(output)

    def generate(self, src_posts, tag_processor, max_new_tags=8,
temperature=0.7, top_k=5):
        """
        Генерирует теги для следующего поста (с запретом
        повторов)
        """
        self.eval()

        src_indices = tag_processor.encode_sequence(src_posts)
        src_tensor = torch.tensor([src_indices])
        src_mask = torch.zeros(1, len(src_indices)).bool()

        start_idx =
tag_processor.tag_to_idx[tag_processor.start_token]
        end_idx = tag_processor.tag_to_idx[tag_processor.end_token]

        tgt = torch.tensor([[start_idx]])
        generated_indices = []

        with torch.no_grad():
            for step in range(max_new_tags):
                tgt_mask = self.generate_causal_mask(tgt.size(1))
                logits = self.forward(src_tensor, tgt,
src_mask=src_mask, tgt_mask=tgt_mask)
                next_logits = logits[:, -1, :] / temperature

                # Запрет END токена на первых 2 шагах
                if step < 2:
                    next_logits[0, end_idx] = -float('inf')

                # Запрет на повторение уже сгенерированных тегов
                for idx in generated_indices:
                    next_logits[0, idx] = -float('inf')

                # Запрет START и PAD токенов
                next_logits[0, start_idx] = -float('inf')
                next_logits[0,
tag_processor.tag_to_idx[tag_processor.pad_token]] = -float('inf')

                # Top-k sampling
                if top_k > 0:

```

```

        top_k_values, top_k_indices =
torch.topk(next_logits, min(top_k, next_logits.size(-1)))
        top_k_probs = torch.softmax(top_k_values, dim=-
1)
        chosen_idx = torch.multinomial(top_k_probs,
1).item()
        next_token = top_k_indices[0,
chosen_idx].item()
    else:
        next_probs = torch.softmax(next_logits, dim=-1)
        next_token = torch.multinomial(next_probs,
1).item()

    if next_token == end_idx:
        break

    generated_indices.append(next_token)
    tgt = torch.cat([tgt,
torch.tensor([[next_token]]), dim=1)

    # При отсутствии генерации осуществляется возврат самых
популярных тегов
    if not generated_indices:
        if is_master:
            print("Теги не сгенерированы, используются
популярные метки")
            popular_tags = []
            for tag, freq in
sorted(tag_processor.tag_frequency.items(), key=lambda x: x[1],
reverse=True):
                if tag not in ['<PAD>', '<START>', '<END>',
'<UNK>']:
                    popular_tags.append(tag)
                    if len(popular_tags) >= max_new_tags:
                        break
            return popular_tags

    # Удаление возможных дубликатов
    unique_tags = []
    for tag in tag_processor.decode_indices(generated_indices):
        if tag not in unique_tags:
            unique_tags.append(tag)

    return unique_tags

def generate_causal_mask(self, size):
    mask = torch.triu(torch.ones(size, size),
diagonal=1).bool()
    return mask

# =====

```

```

# 5. DATASET
# =====

class UserHistoryDataset(Dataset):
    def __init__(self, user_histories, tag_processor, min_posts=2,
max_posts=5):
        self.tag_processor = tag_processor
        self.min_posts = min_posts
        self.max_posts = max_posts
        self.samples = []

        for history in user_histories:
            if len(history) < min_posts + 1:
                continue

            for i in range(len(history) - 1):
                if i + 1 < min_posts:
                    continue

                max_possible = min(max_posts, i + 1)
                num_input_posts = random.randint(min_posts,
max_possible)
                input_posts = history[i - num_input_posts + 1 : i +
1]
                target_post = history[i + 1]

                if len(input_posts) >= min_posts:
                    self.samples.append((input_posts, target_post))

        if is_master:
            print(f"[{hostname}:{rank}] Создано {len(self.samples)}
обучающих примеров", flush=True)

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        input_posts, target_post = self.samples[idx]

        src = self.tag_processor.encode_sequence(input_posts)

        start_idx =
self.tag_processor.tag_to_idx[self.tag_processor.start_token]
        end_idx =
self.tag_processor.tag_to_idx[self.tag_processor.end_token]

        target_encoded =
self.tag_processor.encode_post(target_post)
        tgt = [start_idx] + target_encoded + [end_idx]

        return torch.tensor(src), torch.tensor(tgt)

```

```

def collate_batch(batch):
    src_seqs, tgt_seqs = zip(*batch)
    src_padded = nn.utils.rnn.pad_sequence(src_seqs,
batch_first=True, padding_value=0)
    tgt_padded = nn.utils.rnn.pad_sequence(tgt_seqs,
batch_first=True, padding_value=0)
    src_mask = (src_padded == 0)
    return src_padded, tgt_padded, src_mask

# =====
# 6. ЗАГРУЗКА ДАННЫХ ИЗ JSON
# =====

def load_data_from_json(json_path):
    """
    Загружает данные из JSON файла в формате:
    {
        "user_123": {
            "liked_posts": [
                {"tags": ["музыка", "рок"]},
                {"tags": ["рок", "концерт"]}
            ]
        }
    }
    """
    if is_master:
        print(f"[{hostname}:{rank}] Загрузка данных из
{json_path}", flush=True)

    with open(json_path, 'r', encoding='utf-8') as f:
        raw_data = json.load(f)

    user_histories = []

    for user_id, user_data in raw_data.items():
        history = []

        # Извлечение тегов из каждого поста
        for post in user_data.get('liked_posts', []):
            tags = post.get('tags', [])
            if tags: # Пропуск пустых постов
                history.append(tags)

        # Добавление только пользователей с достаточным количеством
постов
        if len(history) >= 3: # Минимум 3 поста для обучения
            user_histories.append(history)

    print(f"[{hostname}:{rank}] Загружено {len(user_histories)}
пользователей", flush=True)

```

```

    return user_histories

# =====
# 7. СОХРАНЕНИЕ МОДЕЛИ (ТОЛЬКО НА МАСТЕРЕ)
# =====

def save_model(model, tag_processor, path="tag_transformer.pt"):
    torch.save({
        'model_state_dict': model.state_dict(),
        'tag_to_idx': tag_processor.tag_to_idx,
        'idx_to_tag': tag_processor.idx_to_tag,
        'tag_frequency': dict(tag_processor.tag_frequency),
    }, path)
    print(f"[{hostname}:{rank}] Модель сохранена в {path}",
          flush=True)

# =====
# 8. РАСПРЕДЕЛЁННОЕ ОБУЧЕНИЕ С MPI
# =====

def distributed_train(model, train_loader, val_loader, optimizer,
                    criterion,
                    scheduler, epochs=5, patience=5,
                    min_delta=0.001):
    best_val_loss = float('inf')
    patience_counter = 0
    best_epoch = 0

    # Для сбора статистики по времени
    total_compute_time = 0.0      # чистое время вычислений
    total_mpi_time = 0.0          # время синхронизации через MPI
    epoch_times = []              # общее время каждой эпохи

    for epoch in range(epochs):
        epoch_start_time = time.time()
        epoch_compute_time = 0.0  # вычисления в этой эпохе
        epoch_mpi_time = 0.0      # MPI в этой эпохе

        # ---- ОБУЧЕНИЕ ----
        model.train()
        total_loss = 0.0
        num_batches = 0

        # Устанавливаем эпоху для DistributedSampler
        train_loader.sampler.set_epoch(epoch)

        for batch_idx, (src, tgt, src_mask) in
            enumerate(train_loader):
                # --- ЧИСТЫЕ ВЫЧИСЛЕНИЯ ---
                compute_start = time.time()

```

```

        logits = model(src, tgt[:, :-1], src_mask=src_mask)
        loss = criterion(logits.reshape(-1, model.vocab_size),
tgt[:, 1:].reshape(-1))

    optimizer.zero_grad()
    loss.backward()

    compute_time = time.time() - compute_start
    epoch_compute_time += compute_time

    # --- СИНХРОНИЗАЦИЯ MPI ---
    mpi_start = time.time()

    for param in model.parameters():
        if param.grad is not None:
            # Allreduce с суммированием и последующим
делением на world_size
            grad_buffer = param.grad.data.clone()
            comm.Allreduce(MPI.IN_PLACE, grad_buffer,
op=MPI.SUM)
            param.grad.data = grad_buffer / world_size

    mpi_time = time.time() - mpi_start
    epoch_mpi_time += mpi_time

    optimizer.step()

    total_loss += loss.item()
    num_batches += 1

    # Периодический вывод
    if batch_idx % 20 == 0 and batch_idx > 0:
        print(f"[{hostname}:{rank}] Epoch {epoch+1}, batch
{batch_idx}, "
            f"loss={loss.item():.4f},
compute={compute_time:.3f}s, mpi={mpi_time:.3f}s",
            flush=True)

    avg_train_loss = total_loss / num_batches if num_batches >
0 else 0

    # Накапливаем за эпоху
    total_compute_time += epoch_compute_time
    total_mpi_time += epoch_mpi_time

    # ---- ВАЛИДАЦИЯ (только мастер) ----
    if is_master:
        model.eval()
        val_loss = 0.0
        num_val_batches = 0
        with torch.no_grad():

```

```

        for src, tgt, src_mask in val_loader:
            logits = model(src, tgt[:, :-1],
src_mask=src_mask)
            loss = criterion(logits.reshape(-1,
model.vocab_size), tgt[:, 1:].reshape(-1))
            val_loss += loss.item()
            num_val_batches += 1
        avg_val_loss = val_loss / num_val_batches if
num_val_batches > 0 else 0

        epoch_total_time = time.time() - epoch_start_time
        epoch_times.append(epoch_total_time)

        if (epoch + 1) % 5 == 0 or epoch == 0:
            print(f"[{hostname}:{rank}] Epoch
{epoch+1:2d}/{epochs}, "
                f"Train Loss: {avg_train_loss:.4f}, Val Loss:
{avg_val_loss:.4f}", flush=True)
            print(f"    Время эпохи: {epoch_total_time:.2f}с "
                f"(вычисления: {epoch_compute_time:.2f}с,
MPI: {epoch_mpi_time:.2f}с)",
                flush=True)

            # Ранняя остановка
            if avg_val_loss < best_val_loss - min_delta:
                best_val_loss = avg_val_loss
                patience_counter = 0
                best_epoch = epoch + 1
                save_model(model, tag_processor,
"improved_tag_recommender.pt")
                print(f"[{hostname}:{rank}]    Новый лучший
результат! Val Loss: {best_val_loss:.4f} (эпоха {best_epoch})",
flush=True)
            else:
                patience_counter += 1

                if patience_counter >= patience:
                    print(f"[{hostname}:{rank}] Ранняя остановка на
эпохе {epoch+1}", flush=True)
                    print(f"[{hostname}:{rank}] Лучшая модель на эпохе
{best_epoch} с Val Loss = {best_val_loss:.4f}", flush=True)
                    stop_signal = 1
                    comm.bcast(stop_signal, root=0)
                    break
                else:
                    stop_signal = 0
                    comm.bcast(stop_signal, root=0)
            else:
                # Не-мастера ждут сигнал
                stop_signal = comm.bcast(None, root=0)
                if stop_signal == 1:

```

```

        print(f"[{hostname}:{rank}] Получен сигнал
остановки от мастера, завершаю обучение", flush=True)
        break

    # Обновление learning rate
    if scheduler is not None:
        scheduler.step()

    # Дополнительный вывод для не-мастеров
    if not is_master and (epoch + 1) % 5 == 0:
        print(f"[{hostname}:{rank}] Эпоха {epoch+1} завершена,
train_loss={avg_train_loss:.4f}", flush=True)

    return best_val_loss, epoch_times, total_compute_time,
total_mpi_time

# =====
# 9. ОСНОВНОЙ ЗАПУСК С MPI
# =====

if __name__ == "__main__":

    DATA_FILE = "user_data.json"
    MODEL_SAVE_PATH = "improved_tag_recommender.pt"
    BATCH_SIZE = 64
    EPOCHS = 10
    PATIENCE = 5
    MIN_DELTA = 0.001
    LEARNING_RATE = 0.0005
    WEIGHT_DECAY = 0.05

    # ---- ЭТАП 1: ЗАГРУЗКА ДАННЫХ (только мастер) ----
    if is_master:
        print("="*60, flush=True)
        print("РАСПРЕДЕЛЁННОЕ ОБУЧЕНИЕ МОДЕЛИ С MPI", flush=True)
        print(f"Количество процессов: {world_size}", flush=True)
        print("="*60, flush=True)

        print("\n1: ЗАГРУЗКА ДАННЫХ", flush=True)
        if not os.path.exists(DATA_FILE):
            print(f"Ошибка: Файл {DATA_FILE} не найден!",
flush=True)
            exit(1)

        user_histories = load_data_from_json(DATA_FILE)
        if len(user_histories) == 0:
            print("Ошибка: Отсутствуют данные для обучения!",
flush=True)
            exit(1)

    # Сбор частот тегов со всех процессов

```

```

        print("\nСбор частот тегов со всех процессов...",
flush=True)
        local_freq = defaultdict(int)
        for history in user_histories:
            for post in history:
                for tag in post:
                    local_freq[tag] += 1

        # Отправляем локальные частоты мастеру
        all_freqs = comm.gather(dict(local_freq), root=0)

        # Мастер объединяет частоты
        global_freq = defaultdict(int)
        for freq_dict in all_freqs:
            for tag, count in freq_dict.items():
                global_freq[tag] += count

        # Мастер строит словарь
        print("\n2: ПОСТРОЕНИЕ СЛОВАРЯ", flush=True)
        tag_processor = BalancedTagProcessor(min_frequency=1)
        # Временно подменяем tag_frequency, чтобы build_vocab
использовал глобальные частоты
        tag_processor.tag_frequency = global_freq
        tag_processor.build_vocab(user_histories) # фактически
использует self.tag_frequency
        class_weights =
tag_processor.compute_class_weights(user_histories)

        print("\n3: ПОДГОТОВКА ДАННЫХ", flush=True)
        full_dataset = UserHistoryDataset(user_histories,
tag_processor, min_posts=2, max_posts=4)
        val_size = max(1, int(0.1 * len(full_dataset)))
        train_size = len(full_dataset) - val_size
        train_dataset, val_dataset =
torch.utils.data.random_split(full_dataset, [train_size, val_size])

        print(f"Размер обучающего датасета: {len(train_dataset)}",
flush=True)
        print(f"Размер валидационного датасета:
{len(val_dataset)}", flush=True)

        vocab_data = {
            'tag_to_idx': tag_processor.tag_to_idx,
            'idx_to_tag': tag_processor.idx_to_tag,
            'tag_frequency': dict(tag_processor.tag_frequency),
            'class_weights': class_weights,
            'vocab_size': len(tag_processor.tag_to_idx)
        }

        print("\n4: СОЗДАНИЕ МОДЕЛИ", flush=True)
        model = ImprovedTagTransformer(
            vocab_size=vocab_data['vocab_size'],

```

```

        d_model=256, nhead=8, num_layers=3, dropout=0.4
    )
    total_params = sum(p.numel() for p in model.parameters())
    print(f"Параметров модели: {total_params:}", flush=True)

    val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE,
shuffle=False, collate_fn=collate_batch)
    else:
        # Не-мастер процессы: загружают данные и отправляют частоты
мастеру
        with open(DATA_FILE, 'r', encoding='utf-8') as f:
            raw_data = json.load(f)
        user_histories = []
        for user_id, user_data in raw_data.items():
            history = []
            for post in user_data.get('liked_posts', []):
                tags = post.get('tags', [])
                if tags:
                    history.append(tags)
            if len(history) >= 3:
                user_histories.append(history)

        local_freq = defaultdict(int)
        for history in user_histories:
            for post in history:
                for tag in post:
                    local_freq[tag] += 1

        # Отправляем локальные частоты мастеру
        comm.gather(dict(local_freq), root=0)

        # Инициализация пустых переменных
        vocab_data = None
        train_dataset = None
        model = None
        val_loader = None
        tag_processor = None
        class_weights = None

        # ---- РАССЫЛКА СЛОВАРЯ ВСЕМ ПРОЦЕССАМ ----
        vocab_data = comm.bcast(vocab_data, root=0)

        if not is_master:
            print(f"[{hostname}:{rank}] Получен словарь от мастера,
размер {vocab_data['vocab_size']}", flush=True)
            tag_processor = BalancedTagProcessor(min_frequency=1)
            tag_processor.tag_to_idx = vocab_data['tag_to_idx']
            # Ключи idx_to_tag приходят как строки из JSON, преобразуем
в int
            tag_processor.idx_to_tag = {int(k): v for k, v in
vocab_data['idx_to_tag'].items()}

```

```

        tag_processor.tag_frequency = defaultdict(int,
vocab_data['tag_frequency'])
        class_weights = vocab_data['class_weights']

        model = ImprovedTagTransformer(
            vocab_size=vocab_data['vocab_size'],
            d_model=256, nhead=8, num_layers=3, dropout=0.4
        )

        # Каждый не-мастер загружает свой датасет заново (можно
        было бы передать через MPI, но проще загрузить)
        with open(DATA_FILE, 'r', encoding='utf-8') as f:
            raw_data = json.load(f)
        user_histories = []
        for user_id, user_data in raw_data.items():
            history = []
            for post in user_data.get('liked_posts', []):
                tags = post.get('tags', [])
                if tags:
                    history.append(tags)
            if len(history) >= 3:
                user_histories.append(history)
        full_dataset = UserHistoryDataset(user_histories,
tag_processor, min_posts=2, max_posts=4)
        val_size = max(1, int(0.1 * len(full_dataset)))
        train_size = len(full_dataset) - val_size
        train_dataset, _ =
torch.utils.data.random_split(full_dataset, [train_size, val_size])

# ---- ПОДГОТОВКА DATALOADER С DISTRIBUTED SAMPLER ----
if is_master:
    train_sampler = DistributedSampler(
        train_dataset,
        num_replicas=world_size,
        rank=rank,
        shuffle=True
    )
else:
    train_sampler = DistributedSampler(
        train_dataset,
        num_replicas=world_size,
        rank=rank,
        shuffle=True
    )

print(f"[{hostname}:{rank}] Sampler: replicas={world_size},
rank={rank}, dataset size={len(train_dataset)}", flush=True)

train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    sampler=train_sampler,

```

```

        collate_fn=collate_batch
    )

    if not is_master:
        val_loader = None

    # ---- СОЗДАНИЕ ОПТИМИЗАТОРА И ФУНКЦИИ ПОТЕРЬ ----
    weights_tensor = torch.ones(len(tag_processor.tag_to_idx))
    for tag, weight in class_weights.items():
        if tag in tag_processor.tag_to_idx:
            idx = tag_processor.tag_to_idx[tag]
            weights_tensor[idx] = weight
    weights_tensor[0] = 0 # PAD токен

    optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE,
weight_decay=WEIGHT_DECAY)
    criterion = nn.CrossEntropyLoss(ignore_index=0,
weight=weights_tensor)
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=EPOCHS)

    # ---- ЗАПУСК ОБУЧЕНИЯ ----
    if is_master:
        print("\n5: РАСПРЕДЕЛЁННОЕ ОБУЧЕНИЕ", flush=True)
        print(f"Количество процессов: {world_size}", flush=True)
        print("="*60, flush=True)

    best_val_loss, epoch_times, total_compute_time, total_mpi_time
= distributed_train(
    model, train_loader, val_loader, optimizer, criterion,
scheduler,
    epochs=EPOCHS, patience=PATIENCE, min_delta=MIN_DELTA
)

    if is_master:
        print("\nОБУЧЕНИЕ ЗАВЕРШЕНО!", flush=True)
        print("="*60, flush=True)

    end_time = time.time()
    elapsed_time = end_time - start_time

    # ---- ПОЛНАЯ СТАТИСТИКА ВРЕМЕНИ ----
    print("\n" + "="*60, flush=True)
    print("СТАТИСТИКА ВРЕМЕНИ ОБУЧЕНИЯ", flush=True)
    print("="*60, flush=True)

    hours = int(elapsed_time // 3600)
    minutes = int((elapsed_time % 3600) // 60)
    seconds = elapsed_time % 60
    print(f"Общее время работы программы:      {hours}ч
{minutes}м {seconds:.2f}с", flush=True)

```

```

# Только время обучения (внутри distributed_train)
total_train_time = sum(epoch_times) if epoch_times else 0
th = int(total_train_time // 3600)
tm = int((total_train_time % 3600) // 60)
ts = total_train_time % 60
print(f"Время обучения (все эпохи):          {th}ч {tm}м
{ts:.2f}с", flush=True)

# Детализация
if total_train_time > 0:
    print(f"\nИз них:", flush=True)
    cm = int(total_compute_time // 60)
    cs = total_compute_time % 60
    print(f"    Чистые вычисления (forward+backward): {cm}м
{cs:.2f}с ({total_compute_time/total_train_time*100:.1f}%)",
flush=True)

    mm = int(total_mpi_time // 60)
    ms = total_mpi_time % 60
    print(f"    Синхронизация MPI (Allreduce):          {mm}м
{ms:.2f}с ({total_mpi_time/total_train_time*100:.1f}%)",
flush=True)

    other_time = total_train_time - total_compute_time -
total_mpi_time
    om = int(other_time // 60)
    os = other_time % 60
    print(f"    Прочее (загрузка, ожидание):          {om}м
{os:.2f}с ({other_time/total_train_time*100:.1f}%)", flush=True)

# Средние показатели
if epoch_times:
    avg_epoch = sum(epoch_times) / len(epoch_times)
    print(f"\nСреднее время эпохи: {avg_epoch:.2f}с",
flush=True)
    print(f"Число процессов: {world_size}", flush=True)

# Время на дополнительную инициализацию
init_time = elapsed_time - total_train_time
ih = int(init_time // 3600)
im = int((init_time % 3600) // 60)
iss = init_time % 60
print(f"\nПодготовка и сохранение:          {ih}ч {im}м
{iss:.2f}с", flush=True)

print("="*60, flush=True)
print(f"[{hostname}:{rank}] Завершение выполнения:
{time.strftime('%Y-%m-%d %H:%M:%S')}", flush=True)

MPI.Finalize()
Файл «PyTorchDDP.py»

```

```

import time
import math
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, Subset
from torch.utils.data.distributed import DistributedSampler
from collections import defaultdict
import random
import numpy as np
import json
import os
import socket
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP

# =====
# ИНИЦИАЛИЗАЦИЯ DDP
# =====

rank = int(os.environ.get('RANK', 0))
world_size = int(os.environ.get('WORLD_SIZE', 1))
local_rank = int(os.environ.get('LOCAL_RANK', 0))
is_master = (rank == 0)
hostname = socket.gethostname()

# Инициализация группы процессов (бэкенд gloo для CPU)
dist.init_process_group(backend='gloo', rank=rank,
world_size=world_size)

if is_master:
    start_time = time.time()
    print(f"[{hostname}:{rank}] Начало выполнения программы:
{time.strftime('%Y-%m-%d %H:%M:%S')}", flush=True)

print(f"[{hostname}:{rank}/{world_size}] Процесс запущен",
flush=True)

# =====
# 1. БАЗОВЫЙ ОБРАБОТЧИК ТЕГОВ
# =====

class TagProcessor:
    def __init__(self, min_frequency=2):
        self.min_frequency = min_frequency
        self.tag_to_idx = {}
        self.idx_to_tag = {}
        self.tag_frequency = defaultdict(int)
        self.unknown_token = "<UNK>"
        self.pad_token = "<PAD>"
        self.start_token = "<START>"
        self.end_token = "<END>"

```

```

def build_vocab(self, all_user_histories):
    for history in all_user_histories:
        for post in history:
            for tag in post:
                self.tag_frequency[tag] += 1

    special_tokens = [self.pad_token, self.start_token,
self.end_token, self.unknown_token]
    for i, token in enumerate(special_tokens):
        self.tag_to_idx[token] = i
        self.idx_to_tag[i] = token

    idx = len(special_tokens)
    for tag, freq in sorted(self.tag_frequency.items(),
key=lambda x: x[1], reverse=True):
        if freq >= self.min_frequency:
            self.tag_to_idx[tag] = idx
            self.idx_to_tag[idx] = tag
            idx += 1

    if is_master:
        print(f"[{hostname}:{rank}] Словарь построен:
{len(self.tag_to_idx)} токенов", flush=True)
        print(f"[{hostname}:{rank}] - Тегов:
{len(self.tag_to_idx) - 4}", flush=True)

    def encode_post(self, post_tags):
        return [self.tag_to_idx.get(tag,
self.tag_to_idx[self.unknown_token]) for tag in post_tags]

    def decode_indices(self, indices):
        tags = []
        for idx in indices:
            tag = self.idx_to_tag.get(idx, self.unknown_token)
            if tag not in [self.pad_token, self.start_token,
self.end_token]:
                tags.append(tag)
        return tags

    def encode_sequence(self, posts):
        encoded = []
        for post in posts:
            encoded.extend(self.encode_post(post))
        return encoded

# =====
# 2. РАСШИРЕННЫЙ ОБРАБОТЧИК С ВЕСАМИ
# =====

class BalancedTagProcessor(TagProcessor):

```

```

def __init__(self, min_frequency=1):
    super().__init__(min_frequency)
    self.class_weights = None

def compute_class_weights(self, all_user_histories):
    for history in all_user_histories:
        for post in history:
            for tag in post:
                self.tag_frequency[tag] += 1

    total = sum(self.tag_frequency.values())
    num_classes = len(self.tag_frequency)

    self.class_weights = {}
    for tag, freq in self.tag_frequency.items():
        weight = total / (num_classes * freq)
        self.class_weights[tag] = min(weight, 5.0)

    return self.class_weights

# =====
# 3. ПОЗИЦИОННОЕ КОДИРОВАНИЕ
# =====

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=500):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len,
dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
(-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x):
        return x + self.pe[:x.size(1)]

# =====
# 4. ОСНОВНАЯ МОДЕЛЬ TRANSFORMER
# =====

class ImprovedTagTransformer(nn.Module):
    def __init__(self, vocab_size, d_model=128, nhead=4,
num_layers=3, dropout=0.3):
        super().__init__()
        self.d_model = d_model
        self.vocab_size = vocab_size
        self.embedding = nn.Embedding(vocab_size, d_model)

```

```

self.pos_encoder = PositionalEncoding(d_model)
self.transformer = nn.Transformer(
    d_model=d_model, nhead=nhead,
    num_encoder_layers=num_layers,
num_decoder_layers=num_layers,
    batch_first=True, dropout=dropout
)
self.fc_out = nn.Linear(d_model, vocab_size)
self.dropout = nn.Dropout(dropout)

def forward(self, src, tgt, src_mask=None, tgt_mask=None):
    src_emb = self.embedding(src) * math.sqrt(self.d_model)
    tgt_emb = self.embedding(tgt) * math.sqrt(self.d_model)
    src_emb = self.pos_encoder(src_emb)
    tgt_emb = self.pos_encoder(tgt_emb)
    src_emb = self.dropout(src_emb)
    tgt_emb = self.dropout(tgt_emb)
    output = self.transformer(
        src_emb, tgt_emb,
        src_key_padding_mask=src_mask,
        tgt_key_padding_mask=None,
        memory_key_padding_mask=src_mask,
        tgt_mask=tgt_mask
    )
    return self.fc_out(output)

def generate(self, src_posts, tag_processor, max_new_tags=8,
temperature=0.7, top_k=5):
    self.eval()
    src_indices = tag_processor.encode_sequence(src_posts)
    src_tensor = torch.tensor([src_indices])
    src_mask = torch.zeros(1, len(src_indices)).bool()
    start_idx =
tag_processor.tag_to_idx[tag_processor.start_token]
    end_idx = tag_processor.tag_to_idx[tag_processor.end_token]
    tgt = torch.tensor([[start_idx]])
    generated_indices = []

    with torch.no_grad():
        for step in range(max_new_tags):
            tgt_mask = self.generate_causal_mask(tgt.size(1))
            logits = self.forward(src_tensor, tgt,
src_mask=src_mask, tgt_mask=tgt_mask)
            next_logits = logits[:, -1, :] / temperature

            if step < 2:
                next_logits[0, end_idx] = -float('inf')
            for idx in generated_indices:
                next_logits[0, idx] = -float('inf')
            next_logits[0, start_idx] = -float('inf')
            next_logits[0,
tag_processor.tag_to_idx[tag_processor.pad_token]] = -float('inf')

```

```

        if top_k > 0:
            top_k_values, top_k_indices =
torch.topk(next_logits, min(top_k, next_logits.size(-1)))
            top_k_probs = torch.softmax(top_k_values, dim=-
1)

            chosen_idx = torch.multinomial(top_k_probs,
1).item()

            next_token = top_k_indices[0,
chosen_idx].item()
        else:
            next_probs = torch.softmax(next_logits, dim=-1)
            next_token = torch.multinomial(next_probs,
1).item()

        if next_token == end_idx:
            break
        generated_indices.append(next_token)
        tgt = torch.cat([tgt,
torch.tensor([[next_token]]), dim=1)

    if not generated_indices:
        popular_tags = []
        for tag, freq in
sorted(tag_processor.tag_frequency.items(), key=lambda x: x[1],
reverse=True):
            if tag not in ['<PAD>', '<START>', '<END>',
'<UNK>']:
                popular_tags.append(tag)
                if len(popular_tags) >= max_new_tags:
                    break
    return popular_tags

unique_tags = []
for tag in tag_processor.decode_indices(generated_indices):
    if tag not in unique_tags:
        unique_tags.append(tag)
return unique_tags

def generate_causal_mask(self, size):
    return torch.triu(torch.ones(size, size),
diagonal=1).bool()

# =====
# 5. DATASET
# =====

class UserHistoryDataset(Dataset):
    def __init__(self, user_histories, tag_processor, min_posts=2,
max_posts=5):
        self.tag_processor = tag_processor

```

```

self.min_posts = min_posts
self.max_posts = max_posts
self.samples = []

for history in user_histories:
    if len(history) < min_posts + 1:
        continue
    for i in range(len(history) - 1):
        if i + 1 < min_posts:
            continue
        max_possible = min(max_posts, i + 1)
        num_input_posts = random.randint(min_posts,
max_possible)
        input_posts = history[i - num_input_posts + 1 : i +
1]

        target_post = history[i + 1]
        if len(input_posts) >= min_posts:
            self.samples.append((input_posts, target_post))

    if is_master:
        print(f"[{hostname}:{rank}] Создано {len(self.samples)}
обучающих примеров", flush=True)

def __len__(self):
    return len(self.samples)

def __getitem__(self, idx):
    input_posts, target_post = self.samples[idx]
    src = self.tag_processor.encode_sequence(input_posts)
    start_idx =
self.tag_processor.tag_to_idx[self.tag_processor.start_token]
    end_idx =
self.tag_processor.tag_to_idx[self.tag_processor.end_token]
    target_encoded =
self.tag_processor.encode_post(target_post)
    tgt = [start_idx] + target_encoded + [end_idx]
    return torch.tensor(src), torch.tensor(tgt)

def collate_batch(batch):
    src_seqs, tgt_seqs = zip(*batch)
    src_padded = nn.utils.rnn.pad_sequence(src_seqs,
batch_first=True, padding_value=0)
    tgt_padded = nn.utils.rnn.pad_sequence(tgt_seqs,
batch_first=True, padding_value=0)
    src_mask = (src_padded == 0)
    return src_padded, tgt_padded, src_mask

# =====
# 6. ЗАГРУЗКА ДАННЫХ ИЗ JSON
# =====

```

```

def load_data_from_json(json_path):
    if is_master:
        print(f"[{hostname}:{rank}] Загрузка данных из {json_path}", flush=True)

    with open(json_path, 'r', encoding='utf-8') as f:
        raw_data = json.load(f)

    user_histories = []
    for user_id, user_data in raw_data.items():
        history = []
        for post in user_data.get('liked_posts', []):
            tags = post.get('tags', [])
            if tags:
                history.append(tags)
        if len(history) >= 3:
            user_histories.append(history)

    print(f"[{hostname}:{rank}] Загружено {len(user_histories)} пользователей", flush=True)
    return user_histories

# =====
# 7. СОХРАНЕНИЕ МОДЕЛИ (ТОЛЬКО НА МАСТЕРЕ)
# =====

def save_model(model, tag_processor, path="tag_transformer.pt"):
    torch.save({
        'model_state_dict': model.state_dict(),
        'tag_to_idx': tag_processor.tag_to_idx,
        'idx_to_tag': tag_processor.idx_to_tag,
        'tag_frequency': dict(tag_processor.tag_frequency),
    }, path)
    print(f"[{hostname}:{rank}] Модель сохранена в {path}", flush=True)

# =====
# 8. РАСПРЕДЕЛЁННОЕ ОБУЧЕНИЕ С DDP
# =====

def distributed_train_ddp(model, train_loader, val_loader,
optimizer, criterion, scheduler, epochs=5, patience=5,
min_delta=0.001):
    best_val_loss = float('inf')
    patience_counter = 0
    best_epoch = 0

    # Для сбора статистики по времени

```

```

total_compute_time = 0.0      # чистое время вычислений
total_ddp_time = 0.0         # время синхронизации DDP
epoch_times = []             # общее время каждой эпохи

for epoch in range(epochs):
    epoch_start_time = time.time()
    epoch_compute_time = 0.0   # вычисления в этой эпохе
    epoch_ddp_time = 0.0       # DDP синхронизация в этой эпохе

    model.train()
    total_loss = 0.0
    num_batches = 0

    train_loader.sampler.set_epoch(epoch)

    for batch_idx, (src, tgt, src_mask) in
enumerate(train_loader):
        # --- ЧИСТЫЕ ВЫЧИСЛЕНИЯ ---
        compute_start = time.time()

        logits = model(src, tgt[:, :-1], src_mask=src_mask)
        loss = criterion(logits.reshape(-1,
model.module.vocab_size), tgt[:, 1:].reshape(-1))

        optimizer.zero_grad()
        loss.backward()

        compute_time = time.time() - compute_start
        epoch_compute_time += compute_time

        # --- СИНХРОНИЗАЦИЯ DDP (происходит внутри
optimizer.step()) ---
        ddp_start = time.time()
        optimizer.step()
        ddp_time = time.time() - ddp_start
        epoch_ddp_time += ddp_time

        total_loss += loss.item()
        num_batches += 1

        if batch_idx % 20 == 0 and batch_idx > 0:
            print(f"[{hostname}:{rank}] Epoch {epoch+1}, batch
{batch_idx}, "
                    f"loss={loss.item():.4f},
compute={compute_time:.3f}s, ddp={ddp_time:.3f}s",
                    flush=True)

        avg_train_loss = total_loss / num_batches if num_batches >
0 else 0

    # Накапливаем за эпоху
    total_compute_time += epoch_compute_time

```

```

total_ddp_time += epoch_ddp_time

if is_master:
    model.eval()
    val_loss = 0.0
    num_val_batches = 0
    with torch.no_grad():
        for src, tgt, src_mask in val_loader:
            logits = model(src, tgt[:, :-1],
src_mask=src_mask)
            loss = criterion(logits.reshape(-1,
model.module.vocab_size), tgt[:, 1:].reshape(-1))
            val_loss += loss.item()
            num_val_batches += 1
        avg_val_loss = val_loss / num_val_batches if
num_val_batches > 0 else 0

    epoch_total_time = time.time() - epoch_start_time
    epoch_times.append(epoch_total_time)

    if (epoch + 1) % 5 == 0 or epoch == 0:
        print(f"[{hostname}:{rank}] Epoch
{epoch+1:2d}/{epochs}, "
            f"Train Loss: {avg_train_loss:.4f}, Val Loss:
{avg_val_loss:.4f}", flush=True)
        print(f"    Время эпохи: {epoch_total_time:.2f}с "
            f"(вычисления: {epoch_compute_time:.2f}с,
DDP: {epoch_ddp_time:.2f}с)",
            flush=True)

        if avg_val_loss < best_val_loss - min_delta:
            best_val_loss = avg_val_loss
            patience_counter = 0
            best_epoch = epoch + 1
            save_model(model.module, tag_processor,
"improved_tag_recommender.pt")
            print(f"[{hostname}:{rank}]    Новый лучший
результат! Val Loss: {best_val_loss:.4f} (эпоха {best_epoch})",
flush=True)
        else:
            patience_counter += 1

        if patience_counter >= patience:
            print(f"[{hostname}:{rank}] Ранняя остановка на
эпохе {epoch+1}", flush=True)
            print(f"[{hostname}:{rank}] Лучшая модель на эпохе
{best_epoch} с Val Loss = {best_val_loss:.4f}", flush=True)
            break

    if scheduler is not None:
        scheduler.step()

```

```

        if not is_master and (epoch + 1) % 5 == 0:
            print(f"[{hostname}:{rank}] Эпоха {epoch+1} завершена,
train_loss={avg_train_loss:.4f}", flush=True)

    return best_val_loss, epoch_times, total_compute_time,
total_ddp_time

# =====
# 9. ОСНОВНОЙ ЗАПУСК С DDP
# =====

if __name__ == "__main__":

    DATA_FILE = "user_data.json"
    MODEL_SAVE_PATH = "improved_tag_recommender.pt"
    BATCH_SIZE = 64
    EPOCHS = 5
    PATIENCE = 5
    MIN_DELTA = 0.001
    LEARNING_RATE = 0.0005
    WEIGHT_DECAY = 0.05

    # Фиксируем seed для воспроизводимости
    seed = 42
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)

    if is_master:
        print("="*60, flush=True)
        print("РАСПРЕДЕЛЁННОЕ ОБУЧЕНИЕ МОДЕЛИ С DDP (GLOO)",
flush=True)
        print(f"Количество процессов: {world_size}", flush=True)
        print("="*60, flush=True)

    # ---- Загрузка данных на всех процессах ----
    if not os.path.exists(DATA_FILE):
        print(f"Ошибка: Файл {DATA_FILE} не найден на процессе
{rank}!", flush=True)
        exit(1)

    user_histories = load_data_from_json(DATA_FILE)
    if len(user_histories) == 0:
        print(f"Ошибка: Отсутствуют данные для обучения на процессе
{rank}!", flush=True)
        exit(1)

    # ---- Построение словаря на каждом процессе независимо ----
    print("\n2: ПОСТРОЕНИЕ СЛОВАРЯ", flush=True)
    tag_processor = BalancedTagProcessor(min_frequency=1)
    tag_processor.build_vocab(user_histories)

```

```

class_weights =
tag_processor.compute_class_weights(user_histories)

# ---- Подготовка датасета на каждом процессе ----
print("\n3: ПОДГОТОВКА ДАННЫХ", flush=True)
full_dataset = UserHistoryDataset(user_histories,
tag_processor, min_posts=2, max_posts=4)
val_size = max(1, int(0.1 * len(full_dataset)))
train_size = len(full_dataset) - val_size
train_dataset, val_dataset =
torch.utils.data.random_split(full_dataset, [train_size, val_size])

if is_master:
    print(f"Размер обучающего датасета: {len(train_dataset)}",
flush=True)
    print(f"Размер валидационного датасета:
{len(val_dataset)}", flush=True)

# ---- Распределённый sampler и DataLoader ----
train_sampler = DistributedSampler(train_dataset,
num_replicas=world_size, rank=rank, shuffle=True)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
sampler=train_sampler, collate_fn=collate_batch, drop_last=True)

if is_master:
    val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE,
shuffle=False, collate_fn=collate_batch)
else:
    val_loader = None

# ---- СОЗДАНИЕ МОДЕЛИ ----
model = ImprovedTagTransformer(
    vocab_size=len(tag_processor.tag_to_idx),
    d_model=256, nhead=8, num_layers=3, dropout=0.4
)
model = DDP(model, broadcast_buffers=False)

if is_master:
    total_params = sum(p.numel() for p in
model.module.parameters())
    print(f"\n4: СОЗДАНИЕ МОДЕЛИ", flush=True)
    print(f"Параметров модели: {total_params:}", flush=True)

# ---- СОЗДАНИЕ ОПТИМИЗАТОРА И ФУНКЦИИ ПОТЕРЬ ----
weights_tensor = torch.ones(len(tag_processor.tag_to_idx))
for tag, weight in class_weights.items():
    if tag in tag_processor.tag_to_idx:
        idx = tag_processor.tag_to_idx[tag]
        weights_tensor[idx] = weight
weights_tensor[0] = 0 # PAD токен

```

```

optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE,
weight_decay=WEIGHT_DECAY)
criterion = nn.CrossEntropyLoss(ignore_index=0,
weight=weights_tensor)
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=EPOCHS)

# ---- ЗАПУСК ОБУЧЕНИЯ ----
if is_master:
    print("\n5: РАСПРЕДЕЛЁННОЕ ОБУЧЕНИЕ", flush=True)
    print(f"Количество процессов: {world_size}", flush=True)
    print("="*60, flush=True)

    best_val_loss, epoch_times, total_compute_time, total_ddp_time
= distributed_train_ddp(
    model, train_loader, val_loader, optimizer, criterion,
scheduler,
    epochs=EPOCHS, patience=PATIENCE, min_delta=MIN_DELTA
)

if is_master:
    print("\nОБУЧЕНИЕ ЗАВЕРШЕНО!", flush=True)
    print("="*60, flush=True)

    end_time = time.time()
    elapsed_time = end_time - start_time

    # ---- ПОЛНАЯ СТАТИСТИКА ВРЕМЕНИ ----
    print("\n" + "="*60, flush=True)
    print("СТАТИСТИКА ВРЕМЕНИ ОБУЧЕНИЯ", flush=True)
    print("="*60, flush=True)

    hours = int(elapsed_time // 3600)
    minutes = int((elapsed_time % 3600) // 60)
    seconds = elapsed_time % 60
    print(f"Общее время работы программы:      {hours}ч
{minutes}м {seconds:.2f}с", flush=True)

    # Только время обучения (внутри distributed_train_ddp)
    total_train_time = sum(epoch_times) if epoch_times else 0
    th = int(total_train_time // 3600)
    tm = int((total_train_time % 3600) // 60)
    ts = total_train_time % 60
    print(f"Время обучения (все эпохи):      {th}ч {tm}м
{ts:.2f}с", flush=True)

    # Детализация
    if total_train_time > 0:
        print(f"\nИз них:", flush=True)
        cm = int(total_compute_time // 60)
        cs = total_compute_time % 60

```

```

        print(f"    Чистые вычисления (forward+backward): {cm}м
{cs:.2f}с ({total_compute_time/total_train_time*100:.1f}%)",
flush=True)

        dm = int(total_ddp_time // 60)
        ds = total_ddp_time % 60
        print(f"    Синхронизация DDP (optimizer.step):      {dm}м
{ds:.2f}с ({total_ddp_time/total_train_time*100:.1f}%)",
flush=True)

        other_time = total_train_time - total_compute_time -
total_ddp_time
        om = int(other_time // 60)
        os = other_time % 60
        print(f"    Прочее (загрузка, ожидание):              {om}м
{os:.2f}с ({other_time/total_train_time*100:.1f}%)", flush=True)

        # Средние показатели
        if epoch_times:
            avg_epoch = sum(epoch_times) / len(epoch_times)
            print(f"\nСреднее время эпохи: {avg_epoch:.2f}с",
flush=True)
            print(f"Число процессов: {world_size}", flush=True)

        # Время на дополнительную инициализацию
        init_time = elapsed_time - total_train_time
        ih = int(init_time // 3600)
        im = int((init_time % 3600) // 60)
        iss = init_time % 60
        print(f"\nПодготовка и сохранение:                    {ih}ч {im}м
{iss:.2f}с", flush=True)

        print("="*60, flush=True)
        print(f"[{hostname}:{rank}] Завершение выполнения:
{time.strftime('%Y-%m-%d %H:%M:%S')}", flush=True)

        dist.destroy_process_group()

Файл «predict.py»

import torch
from collections import defaultdict
import json
import os
import sys

from main import ImprovedTagTransformer, BalancedTagProcessor

def load_model(model_path="improved_tag_recommender.pt"):
    """Загружает сохранённую модель"""
    print("="*60)
    print("ЗАГРУЗКА МОДЕЛИ")
    print("="*60)

```

```

checkpoint = torch.load(model_path, map_location='cpu')

tag_processor = BalancedTagProcessor()
tag_processor.tag_to_idx = checkpoint['tag_to_idx']
tag_processor.idx_to_tag = {int(k): v for k, v in
checkpoint['idx_to_tag'].items()}
tag_processor.tag_frequency = defaultdict(int,
checkpoint['tag_frequency'])

model = ImprovedTagTransformer(
    vocab_size=len(tag_processor.tag_to_idx),
    d_model=256,
    nhead=8,
    num_layers=3,
    dropout=0.4
)
model.load_state_dict(checkpoint['model_state_dict'])
model.eval()

print(f"Модель загружена. Словарь:
{len(tag_processor.tag_to_idx)} токенов")
return model, tag_processor

def load_input_from_json(json_path):
    """
    Загружает входные посты пользователя из JSON файла

    Ожидаемый формат:
    {
        "user_posts": [
            ["музыка", "рок", "концерт"],
            ["рок", "группа"],
            ["музыка", "поп"]
        ]
    }
    """
    print(f"\nЗагрузка входных данных из {json_path}")

    with open(json_path, 'r', encoding='utf-8') as f:
        data = json.load(f)

    user_posts = data['user_posts']

    if not isinstance(user_posts, list):
        raise ValueError("Файл должен содержать массив постов")

    print(f"Загружено {len(user_posts)} постов")
    for i, post in enumerate(user_posts, 1):
        print(f"    Пост {i}: {post}")

```

```

return user_posts

def generate_tags(model, tag_processor, user_posts, max_tags=6,
temperature=0.9, top_k=3):
    """
    Генерирует теги для следующего поста

    Args:
        model: загруженная модель
        tag_processor: процессор тегов
        user_posts: список постов пользователя [['тег1','тег2'],
['тег3','тег4']]
        max_tags: максимальное количество тегов в рекомендации
        temperature: температура генерации (0.5-1.5)
        top_k: количество топ-вариантов для выбора

    Returns:
        list: список рекомендованных тегов
    """
    # Добавление метода generate к модели (если он отсутствует)
    if not hasattr(model, 'generate'):
        # Метод generate из main.py
        def generate(self, src_posts, tag_processor,
max_new_tags=6, temperature=0.9, top_k=5):
            self.eval()

            src_indices = tag_processor.encode_sequence(src_posts)
            src_tensor = torch.tensor([src_indices])
            src_mask = torch.zeros(1, len(src_indices)).bool()

            start_idx =
tag_processor.tag_to_idx[tag_processor.start_token]
            end_idx =
tag_processor.tag_to_idx[tag_processor.end_token]

            tgt = torch.tensor([[start_idx]])
            generated_indices = []

            with torch.no_grad():
                for _ in range(max_new_tags):
                    tgt_mask =
self.generate_causal_mask(tgt.size(1))
                    logits = self.forward(src_tensor, tgt,
src_mask=src_mask, tgt_mask=tgt_mask)
                    next_logits = logits[:, -1, :] / temperature

                    if top_k > 0:
                        top_k_values, top_k_indices =
torch.topk(next_logits, min(top_k, next_logits.size(-1)))
                        top_k_probs = torch.softmax(top_k_values,
dim=-1)

```

```

        chosen_idx = torch.multinomial(top_k_probs,
1).item()
        next_token = top_k_indices[0,
chosen_idx].item()
    else:
        next_probs = torch.softmax(next_logits,
dim=-1)
        next_token = torch.multinomial(next_probs,
1).item()

    if next_token == end_idx:
        break

    generated_indices.append(next_token)
    tgt = torch.cat([tgt,
torch.tensor([[next_token]]), dim=1)

    return tag_processor.decode_indices(generated_indices)

def generate_causal_mask(self, size):
    mask = torch.triu(torch.ones(size, size),
diagonal=1).bool()
    return mask

    model.generate = generate.__get__(model)
    model.generate_causal_mask =
generate_causal_mask.__get__(model)

    return model.generate(user_posts, tag_processor,
max_new_tags=max_tags,
temperature=temperature, top_k=top_k)

def save_result_to_json(result,
output_path="recommendation_result.json"):
    """
    Сохраняет результат рекомендации в JSON файл

    Args:
        result: список рекомендованных тегов
        output_path: путь для сохранения результата
    """
    output_data = {
        "recommended_tags": result,
        "status": "success",
        "num_tags": len(result)
    }

    with open(output_path, 'w', encoding='utf-8') as f:
        json.dump(output_data, f, indent=2, ensure_ascii=False)

    print(f"\nРезультат сохранён в {output_path}")

```

```

return output_data

def main():
    # Параметры по умолчанию
    INPUT_FILE = "input_posts.json" # Файл с входными постами
    OUTPUT_FILE = "recommendation_result.json" # Файл для
сохранения результата
    MODEL_PATH = "improved_tag_recommender.pt"

    # Проверка аргументов командной строки
    if len(sys.argv) > 1:
        INPUT_FILE = sys.argv[1]
    if len(sys.argv) > 2:
        OUTPUT_FILE = sys.argv[2]

    print("="*60)
    print("ГЕНЕРАТОР ТЕГОВ ДЛЯ РЕКОМЕНДУЕМЫХ ПОСТОВ")
    print("="*60)

    # Проверка существования модели
    if not os.path.exists(MODEL_PATH):
        print(f"\nОшибка: Модель {MODEL_PATH} не найдена!")
        return

    # Проверка существования входного файла
    if not os.path.exists(INPUT_FILE):
        print(f"\nОшибка: Файл {INPUT_FILE} не найден!")
        return

    # Загрузка модели
    model, tag_processor = load_model(MODEL_PATH)

    # Загрузка входных данных из JSON
    try:
        user_posts = load_input_from_json(INPUT_FILE)
    except Exception as e:
        print(f"\nОшибка загрузки JSON: {e}")
        return

    if not user_posts:
        print("\nОшибка: Нет данных для рекомендации!")
        return

    # Демонстрация входных данных
    print("\n" + "="*60)
    print("ВХОДНЫЕ ДАННЫЕ")
    print("="*60)
    for i, post in enumerate(user_posts, 1):
        print(f" Пост {i}: {post}")

    # Генерация рекомендаций

```

```

print("\n" + "="*60)
print("ГЕНЕРАЦИЯ РЕКОМЕНДАЦИЙ")
print("="*60)

recommended = generate_tags(
    model,
    tag_processor,
    user_posts,
    max_tags=8,
    temperature=0.7,
    top_k=5
)

print(f"\nРекомендуемые теги для следующего поста:
{recommended}")

# Сохранение результата в JSON
save_result_to_json(recommended, OUTPUT_FILE)

# Демонстрация доступных тегов для справки
available_tags = [tag for tag in
tag_processor.tag_to_idx.keys()
                    if tag not in ['<PAD>', '<START>', '<END>',
'<UNK>']]
print(f"\nДоступные теги (всего {len(available_tags)}):")
print(f"    {' '.join(sorted(available_tags)[:15])}{'...' if
len(available_tags) > 15 else ''}")

print("\n" + "="*60)
print("ГЕНЕРАЦИЯ ЗАВЕРШЕНА!")
print("="*60)

if __name__ == "__main__":
    main()
    Файл «input_posts.json»

{
    "user_posts": [
        ["technology", "технологии", "ai", "искусственный интеллект"],
        ["machine learning", "машинное обучение", "data", "данные"],
        ["programming", "программирование", "python", "питон"],
        ["robotics", "робототехника", "robot", "робот"],
        ["cybersecurity", "кибербезопасность", "security",
"безопасность"],
        ["blockchain", "блокчейн", "crypto", "криптовалюта"],
        ["web development", "веб_разработка", "code", "код"]
    ]
}

    Файл «recommendation_result.json»

```

```
{
  "recommended_tags": [
    "веб_разработка",
    "искусственный_интеллект"
  ],
  "status": "success",
  "num_tags": 2
}
```

					ПГУ1.090301.40.001 ПЗ	Лист
						134
Изм.	Лист	№ докум.	Подпись	Дата		

ПриложениеВ

(обязательно)

Обучение алгоритма рекомендаций социальных сетей на вычислительном
кластере посредством MPI

Презентация

					ПГУ1.090301.40.001 ПЗ	Лист
						135
Изм.	Лист	№ докум.	Подпись	Дата		

Обучение алгоритма рекомендаций социальных сетей на вычислительном кластере посредством MPI

Автор работы: студент группы 22ВВП2 Кулахметов С.И.

Научный руководитель: д.т.н., профессор кафедры ВТ Зинкин С.А.

Пенза, 2026

Актуальность

В современном мире для решения огромного числа различных задач используются нейронные сети. Однако, в сложившейся экономической ситуации закупка специализированного оборудования сопряжена с колоссальными финансовыми затратами.

Ключевой способ решения данной проблемы – использование уже имеющегося оборудования, которое, для достижения максимальной производительности, следует объединить в единый вычислительный кластер.

В результате, возникает необходимость распараллелить обучение используемой нейронной сети на несколько узлов данной вычислительной системы.

2

Цели проекта

- Разработать программу для распараллеленного обучения нейронной сети согласно стратегии «Data Parallelism» на вычислительном кластере, организованном посредством стандарта MPI MPICH.
- Апробирование реализованного метода путём его применения при обучении алгоритма рекомендаций постов для социальных сетей на вычислительном кластере.
- Сравнение реализованного подхода параллелизации с основным аналогом PyTorch DDP и определение возможностей масштабирования вычислительной системы.

3

Обзор аналогов

PyTorch DDP с использованием бекенда NCCL

Достоинства:

- Максимальная производительность на GPU от компании Nvidia.
- Глубокая интеграция с PyTorch и TensorFlow.
- Поддержка современных топологий.

Недостатки:

- Аппаратная зависимость от видеоускорителей компании Nvidia.
- Сложность отладки.
- Лицензирование.

4

Обзор аналогов

PyTorch DDP с использованием бекенда GLOO

Достоинства:

- Кроссплатформенность и универсальность.
- Простота развёртывания.
- Открытый исходный код.

Недостатки:

- Более низкая производительность на GPU.
- Меньшее количество оптимизаций.
- Ограниченное применение вне PyTorch.

5

Стек технологий



6

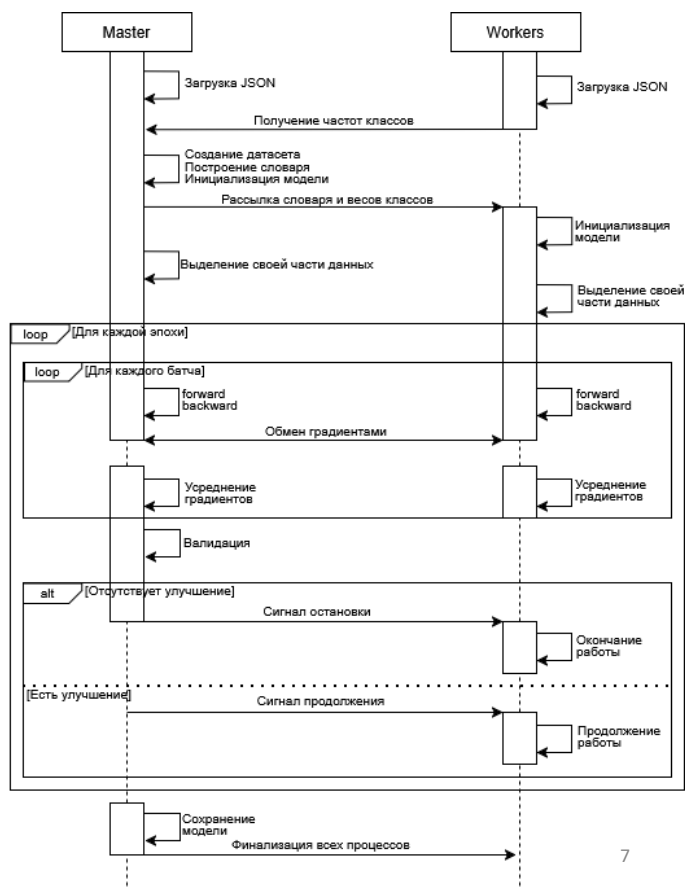
Изм.	Лист	№ докум.	Подпись	Дата

ПГУ1.090301.40.001 ПЗ

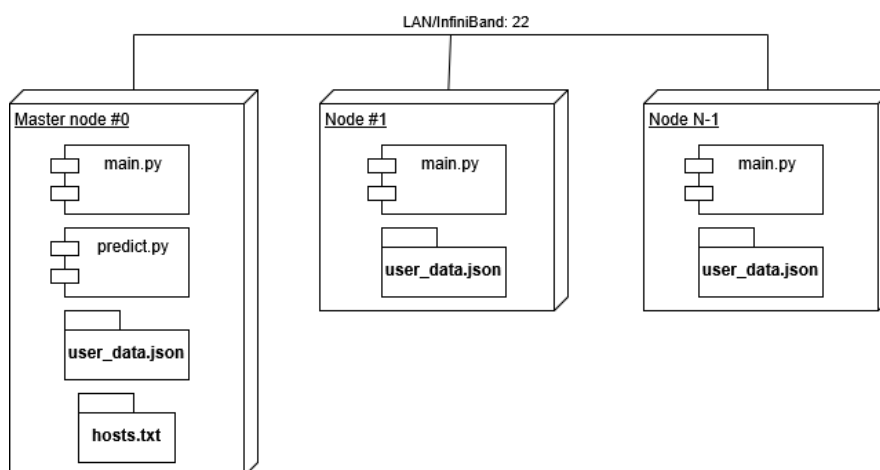
Лист

138

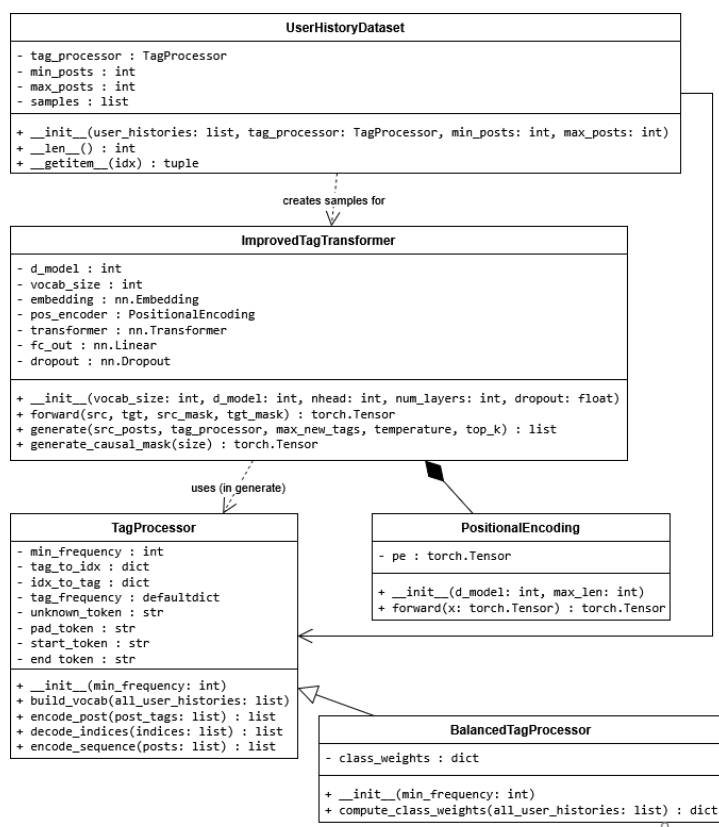
UML-диаграмма последовательности операций при распараллеленном обучении алгоритма



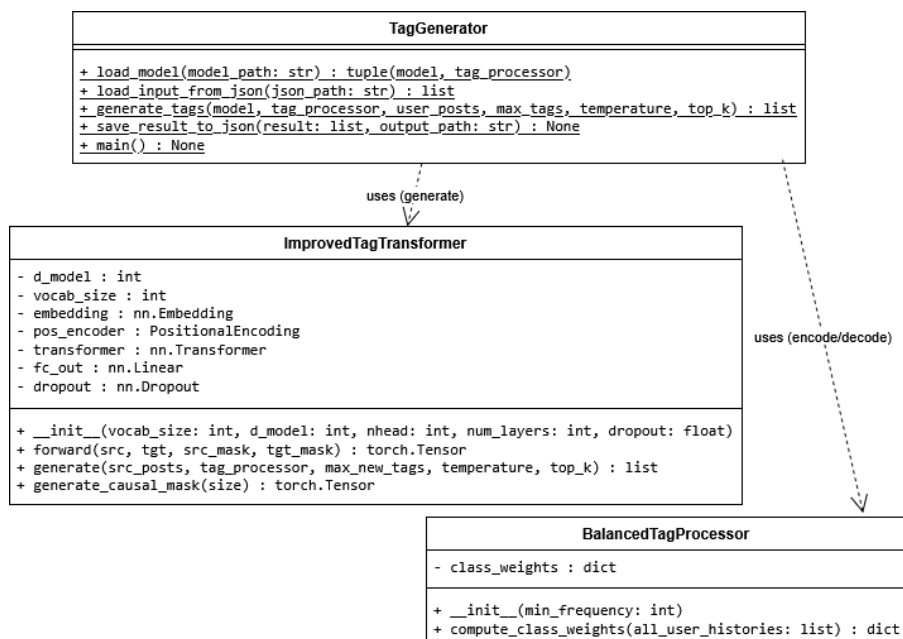
UML-диаграмма развёртывания



UML-диаграмма классов модуля, отвечающего за обучение алгоритма



UML-диаграмма классов модуля генерации тегов для рекомендуемых постов



10

Запуск обучения рекомендательного алгоритма с применением разработанного метода

The image shows two terminal windows. The left window displays the execution of an MPI command: `mpirun -f hosts.txt -np 24 python main.py`. The output shows multiple processes starting on different nodes, including `vt316-10u.bt.local:16/24` through `vt316-11u.bt.local:15/24`. The right window shows the execution of a PyTorch command: `torchrun --nnodes=3 --nproc-per-node=8 --master-addr=192.168.10.72 --master-port=29000 PyTorchDDP.py`. The output shows the process starting on node `W0504` and connecting to 23 peer ranks. It also displays a warning about the `OMP_NUM_THREADS` environment variable.

Результат работы обученного алгоритма рекомендаций постов для социальных сетей

```
=====
ВХОДНЫЕ ДАННЫЕ
=====
Пост 1: ['travel', 'путешествия', 'mountains', 'горы']
Пост 2: ['hiking', 'походы', 'nature', 'природа']
Пост 3: ['beach', 'пляж', 'sea', 'море']
Пост 4: ['diving', 'дайвинг', 'underwater', 'подводный']
Пост 5: ['camping', 'кемпинг', 'tent', 'палатка']
Пост 6: ['excursion', 'экскурсия', 'museum', 'музей']
Пост 7: ['tour', 'тур', 'adventure', 'приключение']

=====
ГЕНЕРАЦИЯ РЕКОМЕНДАЦИЙ
=====
/home/sabir/anaconda3/lib/python3.12/site-packages/torch/nn/modules/transformers.py:178: UserWarning: torch.nn.functional.nested_tensor is deprecated, specifying layout=torch.jagged when constructing a nested tensor, as this is the preferred way. (Triggered at: /home/sabir/anaconda3/lib/python3.12/site-packages/torch/nn/modules/transformers.py:178.)
  output = torch.nested_tensor_from_mask(
Рекомендуемые теги для следующего поста: ['туризм', 'mountaineering']
```

12

Изм.	Лист	№ докум.	Подпись	Дата

ПГУ1.090301.40.001 ПЗ

Лист

141

Масштабирование вычислительной системы

Определение времени обработки одного примера $t(p) = \frac{T \times p}{N}$

Динамика времени обработки одного примера при изменении числа процессов может быть рассчитано при помощи линейной регрессии вида:

$$t(p) = a \times p + b$$

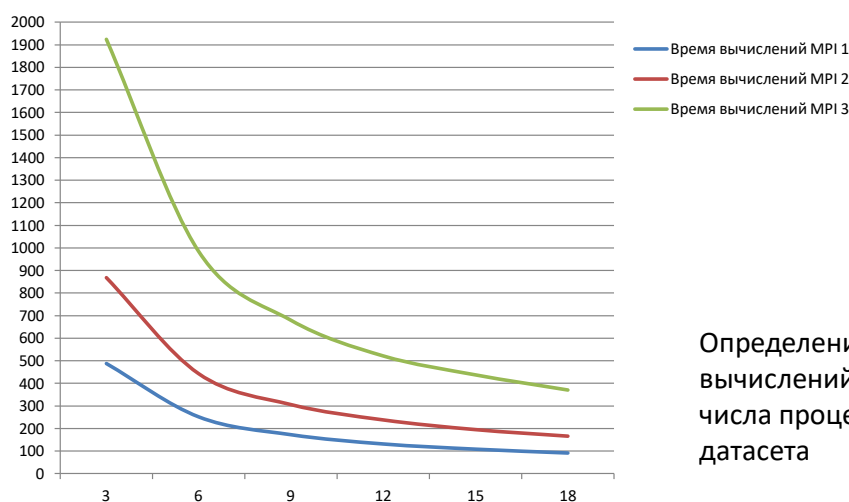
После нахождения времени, затрачиваемого на обработку одного примера, можно вычислить время T , затрачиваемое на общие вычисления при заданном количестве процессов:

$$T = t(p) \times \frac{N}{p} \Rightarrow T = \frac{(a \times p + b) \times N}{p}$$

13

Масштабирование вычислительной системы

Время вычислений на равном числе процессов изменяется пропорционально изменению объёма набора данных

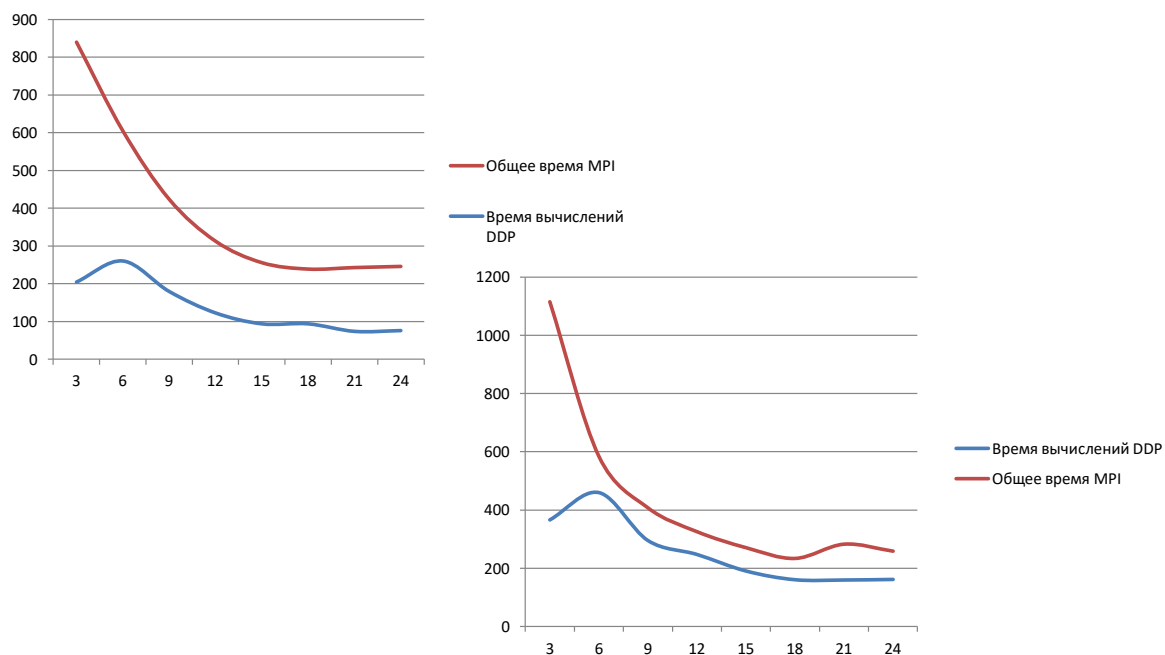


Определения времени вычислений при изменении числа процессов и объёма датасета

$$T = K \times \frac{(a \times p + b) \times N}{p}$$

14

Сравнение результатов работы реализованного вручную подхода с результатами при использовании технологии PyTorch DDP



15

Заключение

В результате выполнения выпускной квалификационной работы успешно разработана программа для реализации распараллеленного обучения по стратегии «Data Parallelism» нейронных сетей и применена для обучения алгоритма рекомендаций постов для социальных сетей на вычислительном кластере, организованном посредством стандарта MPI MPICH.

Результаты тестирования показали, что обученный при помощи реализованного метода алгоритм рекомендаций постов для социальных сетей работает корректно.

16

Изм.	Лист	№ докум.	Подпись	Дата

ПГУ1.090301.40.001 ПЗ

Лист

143

Спасибо за внимание!

17

					ПГУ1.090301.40.001 ПЗ	Лист
						144
Изм.	Лист	№ докум.	Подпись	Дата		