

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

**«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ
ПЕТРА ВЕЛИКОГО»**

институт компьютерных наук и кибербезопасности

высшая школа технологий искусственного интеллекта

направление 02.03.01 математика и компьютерные науки

Отчет по дисциплине:

по теме:

«Архитектура суперкомпьютерных систем»

Обучающийся: _____ Черепанов Михаил Дмитриевич

Руководитель: _____ Чуватов Михаил Владимирович

«_____» _____ 20__ г.

Санкт-Петербург 2024

Содержание

Перечень сокращений и обозначений	3
Введение	4
1 Основная часть	7
1.1 Создание узлов кластера	7
1.2 Установка необходимого программного обеспечения	9
1.3 Настройка сети между виртуальными машинами	10
1.4 Конфигурация ресурсов GPU	14
1.5 Конфигурация NFS	20
1.6 Конфигурация slurm	22
1.7 Конфигурация MUNGE	25
1.8 Настройка доступа по ssh без пароля	25
1.9 Решение практической задачи с помощью созданного кластера . . .	27
1.9.1 Разработка решения задачи №1	28
1.9.2 Разработка решения задачи №2	30
1.10 Проведение экспериментов	33
2 Заключение	35
Список источников	36
Приложение А	37
Приложение В	40
Приложение С	43
Приложение D	48
Приложение Е	49

Перечень сокращений и обозначений

- ОС - операционная система;
- ВМ - виртуальная машина;
- СКЦ - суперкомпьютерный центр;
- CPU - central processing unit (центральный процессор);
- GPU - graphics processing unit (графический процессор);
- HSA - Heterogeneous System Architecture(гетерогенная системная архитектура);
- MPI - Message Passing Interface(интерфейс передачи сообщений);
- CUDA - Compute Unified Device Architecture;
- SLURM - Simple Linux Utility for Resource Management;

Введение

В современном мире объем данных и сложность вычислений растут экспоненциально. Научные исследования, моделирование, искусственный интеллект — все эти задачи требуют огромных вычислительных ресурсов. Чтобы справиться с этим, всё чаще используют вычислительные кластеры — объединения множества независимых вычислительных узлов, работающих вместе как единое целое и управляемых централизованно.

Архитектура суперкомпьютера строится на идеи распараллеливания вычислений и распределения задач между доступными узлами, соединенными между собой сетью. В суперкомпьютерных системах количество таких вычислительных узлов может достигать до миллионов.

Так, например суперкомпьютер «El Capitan» разработанный в национальной лабораторией Министерства энергетики США, который возглавил список ТОП 500 в 2024 году имеет более 11 миллионов вычислительных узлов. Данный суперкомпьютер имеет вычислительную скорость 1.742 ЭФлопс/с [1]. Один эксафлоп = 10^{18} операций в секунду.

Вычислительный кластер СКЦ «Политехнический» Торнадо, доступный студентам СПбПУ содержит 612 вычислительных узлов и имеет производительность около 1.2 Пфлопс/с [2]. Один петафлоп = 10^{15} операций в секунду.

В суперкомпьютерных кластерах используют два основных вида процессоров: CPU и GPU.

CPU — главный компонент компьютера, который выполняет последовательные вычисления и управляет другими системами. Есть в любом компьютере. CPU отвечает за выполнение последовательных операций, обработку логики программ, управление потоками данных. Он включает в себя несколько ядер, каждое из которых может выполнять задачи независимо.

GPU — специализированный компонент, изначально созданный для обработки графики, но теперь широко используемый для ускорения параллельных вычислений. Он содержит от сотен до тысяч ядер, которые могут одновременно выполнять однородные операции над большими массивами данных. GPU эффективен в задачах машинного обучения, научного моделирования, рендеринга и обработки больших данных.

На сегодняшний день архитектура, в которой вместе используются CPU и GPU стала стандартной в мире суперкомпьютеров и называется **гетерогенной системной архитектурой**.

Операционные системы в распределенных вычислительных системах

В большинстве распределенных вычислительных систем, как и в суперкомпьютерах, в частности, самой популярной операционной системой является Linux. Так с 2015 года Linux вытеснил все остальные ОС из списка топ 500.

Преимущества Linux перед другими ОС для подобных задач заключается в следующем:

- Существование множества дистрибутивов позволяет выбрать конкретную си-

систему под требуемые задачи. Кроме того, открытый исходный код позволяет доработать систему под специфичные нужды и подразумевает бесплатное использование ОС.

- Ядро Linux является монолитным и хранит в себе все необходимые компоненты — драйверы, планировщик задач, файловую систему. Отсутствие необходимости передавать данные между компонентами распределенного ядра позволяет увеличить скорость работы системы в целом.

При этом kernel-сервисы выполняются в адресном пространстве ядра, что повышает общую производительность.

- Linux поддерживает такие технологии, как OpenMPI и SLURM, для эффективного распределения задач между узлами в кластере, распределенные файловые системы;

Общие сведения о средах виртуализации

Виртуализация — это создание изолированной программной среды (или нескольких таких сред) в рамках одного физического устройства.

Логическая схема виртуализации включает три составляющих:

Хост-система — это основная операционная система, в рамках которой происходят создание и функционирование изолированной виртуальной среды.

«Гостевая» система — это операционная система (или иная программа, процесс), которая работает внутри изолированной виртуальной среды.

Гипервизор — это программа, с помощью которой осуществляются и управление виртуальной средой. Основные функции гипервизора:

- Создание, запуск, приостановка и завершение работы виртуальных машин;
- Изоляция между виртуальными машинами;
- Управление доступом виртуальных машин к процессору, памяти, дисковому пространству и другим аппаратным ресурсам.

Выделяют два типа гипервизоров:

Гипервизор первого типа: устанавливается непосредственно на железо и работает без какой-либо хостовой операционной системы. Он выполняет функции операционной системы для виртуальных машин, управляющей их ресурсами. Гипервизоры такого типа являются более эффективными по скорости, изоляции виртуальных машин, стабильности (не зависят от ОС). Примеры гипервизоров первого типа: Microsoft Hyper-V для Windows, KVM (Kernel-based Virtual Machine) для Linux.

Гипервизор второго типа: устанавливается на хостовую операционную систему, которая уже работает на физическом оборудовании. Виртуальные машины управляются через гипервизор, но гипервизор работает в контексте хостовой ОС. Преимущества таких гипервизоров: простота установки, гибкость настройки. Однако они уступают первому типу по скорости работы. Примеры: Oracle VirtualBox, VMware Workstation.

Данная работа заключается в создании виртуального гетерогенного распределенного вычислительного кластера с помощью гипервизора Microsoft Hyper-V, узлами которого являются виртуальные машины, гостевая система - Ubuntu Server. На части виртуальных машин доступны графические ускорители хоста - nvidia.

Количество узлов значительно меньше, чем в реальных распределенных вычислительных системах, однако принципы вычисления на них, используемое программное обеспечение, настройка узлов и сети между узлами повторяют принципы работы с реальными системами.

После создания вычислительного кластера произведено тестирование системы с использованием технологий CUDA и MPI.

Тестирование кластера производится путем обработки датасета, предоставленного СКЦ «Политехнический» в соответствии с выбранным вариантом.

Итак, постановка задачи:

1. Создать виртуальный гетерогенный вычислительный кластер состоящий из 4х виртуальных машин: 2 CPU узла, 2 GPU узла. Установить необходимое программное обеспечение, настроить сетевое взаимодействие как между узлами, так и между узлами и хостом.
2. С использованием созданного кластера решить следующие задачи:
 - (a) Отранжировать задачи по убыванию доли успешно завершенных заданий;
 - (b) Нормализовать время ожидания заданий в очереди;

1 Основная часть

1.1 Создание узлов кластера

В качестве общей конфигурации виртуальной машины выбраны следующие параметры:

- Выделенное ОЗУ: 4096 МВ;
- Число виртуальных процессоров: 2;
- Имя виртуальной машины: `cpunodeXX`, `gpunodeXX`, где `XX` - порядковый номер машины.

Для упрощения работы при создании узлов была выбрана следующая стратегия: первая из виртуальных машин создается в диспетчере Hyper-V, на ней производятся все необходимые настройки, последующие создаются путем копирования основной машины.

Для создания ВМ с необходимыми характеристиками требуется выполнить следующие действия: Создать виртуальную машину в диспетчере Hyper-V, настроить ее параметры, установить ОС. Алгоритмы этих действий приведены ниже.

Процесс создания виртуальной машины

- Открыть диспетчер Hyper-V → «Создать» → «Виртуальная машина».
- В разделе «Укажите имя и местоположение»:
Имя: `cpunode01`; Местоположение: `D:VMs`;
- В разделе «Укажите поколение»:
Выбрать поколение №2;
- В разделе «Выделить память»:
Указать 4096 мб;
- В разделе «Настройка сети»:
Выбрать сеть «Default Switch»;
- В разделе «Подключить виртуальный жесткий диск»:
Имя: `cpunode01.vhdx`;
Расположение: `D:VMs`;
Размер: 127 ГБ;
Параметры установки: «Установить позднее»;
- Нажать «Далее» → «Готово»;

Настройка параметров виртуальной машины

- Открыть диспетчер Hyper-V → перейти в параметры созданной виртуальной машины;
- В разделе оборудование → «Процессор»:
Число виртуальных процессоров: 2;
- В разделе «SCSI-контроллер»:
Добавить виртуальный DVD-дисковод;
В разделе «DVD-дисковод» - указать образ операционной системы: (ubuntu.iso);
- В разделе «Безопасность»: Отключить безопасную загрузку;
- В разделе «Управление»:
Автоматическое действие при запуске: Ничего;
Автоматическое действие при завершении: Ничего;
- Нажать «Ок» → «Применить»;

Установка ОС на виртуальную машину

Установка ОС на виртуальную машину не отличается от стандартной установки ОС на компьютер.

При установке выбираем следующие параметры:

- Выбор языка: "English".
- Для раскладки выбираем предлагаемую английскую раскладку.
- Тип установки: "Ubuntu Server".
- Настройки пространства памяти:
 1. Выбираем опцию: "Custom Storage Layout".
 2. Для свободного пространства: заводим swap(50 гб) и основное пространство памяти(4 гб);
- Настраиваем параметры профиля системы:
 - "Your name": вводим имя(mikhail);
 - "Your server's name": cpunode01 (как у виртуальной машины);
 - "Pick a username": user1
 - Выбрать пароль.
- Если предложено установить «Ubuntu Pro» - отказываемся от данной опции;
- Согласиться с установкой OpenSSH сервера;
- Перезапустить ВМ для завершения установки;

1.2 Установка необходимого программного обеспечения

Для выполнения работы необходимо установить следующие пакеты:

- libopenmpi3 - пакет для библиотеки Open MPI;
- slurmd - пакет для управляющего задачами демона;
- openssh-client - клиент OpenSSH;
- openssh-server - сервер OpenSSH;

Установка происходит с помощью следующих команд:

```
1 sudo apt update
2 sudo apt upgrade
3 sudo apt install libopenmpi3
4 sudo apt install slurmd
5 sudo apt install openssh-client
6 sudo apt install openssh-server
7 sudo apt clean
```

1. apt update - обновляет локальный индекс пакетов в системе, скачивая актуальную информацию о доступных пакетах из репозитория, указанных в файлах /etc/apt/sources.list и /etc/apt/sources.list.d/. Это нужно для того, чтобы ОС знала о новых версиях пакетов и их зависимостях.
2. apt upgrade - обновляет установленные пакеты в системе.
3. apt install <name> - устанавливает в системе пакет с именем <name>.
4. apt clean - удаляет все загруженные архивы пакетов из кеша АРТ, освобождая место на диске.

1.3 Настройка сети между виртуальными машинами

Для настройки сети необходимо:

1. Настроить конфигурацию сети на виртуальных машинах;
2. Настроить проброс портов на системе хоста;

Настройка конфигурации сети на виртуальных машинах

Для настройки виртуальных машин по умолчанию используется *cloud-init* - инструмент для автоматической конфигурации виртуальных машин при запуске. Для корректной конфигурации сети нам нужно, чтобы cloud-init не вмешивался в настройки сети.

Для этого необходимо изменить следующие файлы:

1. `/etc/cloud/cloud.cfg.d/99-disable-network-config.cfg` - файл используется для отключения автоматического управления сетевыми настройками, которое выполняет cloud-init. Нам нужно отключить это управление. Для этого запишем в файл следующее содержимое:

```
1 network: {config: disabled}
```

2. `/etc/netplan/50-cloud-init.yaml` - файл конфигурации сети для инструмента Netplan, который используется для настройки сетевых интерфейсов в дистрибутивах Linux. Когда cloud-init активен для конфигурации сети, он генерирует этот файл автоматически, основываясь на предоставленных метаданных. В нашем случае содержимое файла нужно записать в ручную.

Запишем в файл следующее содержимое:

```
1 network:
2   ethernets:
3     eth0:
4       dhcp4: false
5       addresses: [10.0.108.1/24]
6       routes:
7         - to: default
8           via: 10.0.108.254
9       nameservers:
10        addresses: [8.8.4.4, 8.8.8.8]
11 version: 2
```

Содержимое файла означает:

- (a) `network:` - начинается секция описания сетевых настроек.
- (b) `ethernets:` - Раздел для настройки проводных (Ethernet) сетевых интерфейсов.
- (c) `eth0:` - Имя сетевого интерфейса. `eth0` - Это стандартное имя для Ethernet-адаптера.

- (d) `dhcp4: false` - Указывает, что DHCP (динамическое получение IP-адреса) отключён. Система будет использовать статическую IP-конфигурацию, заданную в поле `addresses`.
- (e) `addresses: [10.0.108.1/24]` - Статический IP-адрес, назначаемый интерфейсу `eth0`;
- (f) `routes`:
 - `to: default` — задаёт маршрут по умолчанию;
 - `via: 10.0.108.254` - маршрутизатор, через который будут отправляться такие пакеты.
- (g) `nameservers`:
 - `addresses: [8.8.4.4, 8.8.8.8]` - настройка DNS серверов. Используем публичные серверы Google DNS.
- (h) `version: 2` - Указывает версию формата Netplan. На данный момент версия 2 является стандартной для Ubuntu.

Настройка проброса портов в host системе

Для взаимодействия с виртуальными машинами с помощью хост-узла, необходимо пробросить порты на настроенные ранее адреса. Для этого необходимо воспользоваться следующими PowerShell командами:

1. `New-VMSwitch` - Создает новый виртуальный сетевой адаптер для виртуальных машин [3].

Принимаемые параметры:

- `SwitchName` - имя для адаптера. Обязательный параметр;
- `SwitchType` - Определяет тип создаваемого адаптера. Разрешенные типы:
 - `Internal` - позволяет виртуальной машине обмениваться данными с хостовой машиной и другими виртуальными машинами на том же хосте;
 - `External` - позволяет виртуальной машине общаться только с другими виртуальными машинами, но не имеет доступа к хосту или внешней сети;

Создадим сетевой адаптер следующей командой:

```
1 New-VMSwitch -SwitchName "cluster_switch" -SwitchType Internal
```

2. `New-NetIPAddress` - Создает новый IP-адрес и привязывает его к указанному сетевому интерфейсу [3].

Принимаемые параметры:

- `InterfaceAlias` - Указывает имя сетевого интерфейса, к которому будет привязан новый IP-адрес. Обязательный параметр;

- IPAddress - Указывает IP-адрес, который нужно настроить. Обязательный параметр;
- PrefixLength - Указывает длину префикса подсети для IP-адреса. Например, для маски подсети 255.255.255.0 длина префикса равна 24. Обязательный параметр;
- DefaultGateway - Указывает адрес шлюза по умолчанию, который будет использоваться для указанного IP-адреса.

Пример настройки нового IP-адреса в PowerShell:

```
1 New-NetIPAddress -InterfaceAlias "vEthernet (cluster_switch)" -IPAddress
  ↪ "10.0.108.254" -PrefixLength 24
```

3. New-NetNat - Создает новый NAT (Network Address Translation) для указанного интерфейса внутренней сети [3].

Принимаемые параметры:

- Name - Указывает имя NAT-объекта. Обязательный параметр;
- InternalIPInterfaceAddressPrefix - Указывает адресный префикс внутренней сети, который будет использоваться для NAT. Обязательный параметр;
- ExternalIPInterfaceAddressPrefix - Указывает адресный префикс внешней сети.
- Description - Добавляет описание к NAT-объекту.

```
1 New-NetNat -Name ClusterNAT -InternalIPInterfaceAddressPrefix
  ↪ 10.0.108.0/24
```

4. Add-NetNatStaticMapping - Добавляет статическое сопоставление портов между внешними и внутренними адресами для NAT (Network Address Translation). Это позволяет направлять трафик, поступающий на внешний адрес и порт, к определенному внутреннему адресу и порту [3].

Принимаемые параметры:

- NatName - Указывает имя существующего объекта NAT, к которому применяется статическое сопоставление. Обязательный параметр;
- ExternalIPAddress - Указывает внешний IP-адрес, на который поступает входящий трафик. Для разрешения любых IP-адресов можно использовать 0.0.0.0/0. Обязательный параметр;
- ExternalPort - Указывает порт внешнего IP-адреса, на который будет перенаправляться трафик. Обязательный параметр;
- InternalIPAddress - Указывает IP-адрес устройства внутри сети, к которому будет перенаправляться трафик. Обязательный параметр;

- InternalPort - Указывает порт устройства внутри сети, который будет использоваться для трафика. Обязательный параметр;
- Protocol - Указывает протокол (например, TCP или UDP), который будет использоваться для сопоставления. Обязательный параметр.

1

```
Add-NetNatStaticMapping -NatName ClusterNAT -ExternalIPAddress 0.0.0.0/24  
↪ -ExternalPort 40122 -InternalIPAddress 10.0.108.1 -InternalPort 22  
↪ -Protocol TCP
```

1.4 Конфигурация ресурсов GPU

Для конфигурации ресурсов GPU для узлов, необходимо воспользоваться следующими PowerShell командами:

1. Get-VMHostPartitionableGpu - используется в средах, работающих с виртуализацией на базе Hyper-V. Она позволяет получить информацию о графических процессорах (GPU), которые поддерживают функцию разделяемого GPU (GPU Partitioning).[\[3\]](#)

Принимаемые параметры:

- -CimSession (необязательный) - Позволяет указать удаленную сессию CIM (Common Information Model) для выполнения команды на другом компьютере. Если параметр не указан, команда выполняется локально.
- -ThrottleLimit (необязательный) - Ограничивает количество одновременных операций. Если параметр не указан, используется системное значение по умолчанию.

```
1 Get-VMHostPartitionableGpu
```

Результат выполнения:

```
1 Get-VMHostPartitionableGpu
2
3
4 Name : \\?\PCI#VEN_10DE&DEV_
5 2487&SUBSYS_40741458&REV_A1#4&d0467e6&0&0008#
6 {064092b3-625e-43bf-9eb5-dc
7 845897dd59}\GPUPARAV
8 ValidPartitionCounts : {32}
9 PartitionCount : 32
10 TotalVRAM : 1000000000
11 AvailableVRAM : 1000000000
12 MinPartitionVRAM : 0
13 MaxPartitionVRAM : 1000000000
14 OptimalPartitionVRAM : 1000000000
15 TotalEncode : 18446744073709551615
16 AvailableEncode : 18446744073709551615
17 MinPartitionEncode : 0
18 MaxPartitionEncode : 18446744073709551615
19 OptimalPartitionEncode : 18446744073709551615
20 TotalDecode : 1000000000
21 AvailableDecode : 1000000000
22 MinPartitionDecode : 0
23 MaxPartitionDecode : 1000000000
24 OptimalPartitionDecode : 1000000000
25 TotalCompute : 1000000000
26 AvailableCompute : 1000000000
27 MinPartitionCompute : 0
28 MaxPartitionCompute : 1000000000
```

```

29     OptimalPartitionCompute : 1000000000
30     CimSession              : CimSession: .
31     ComputerName            : 4E305-10
32     IsDeleted                : False
33
34     Name                    : \\?\PCI#VEN_8086&DEV
35     _4C8A&SUBSYS_7D171462&REV_04#3&11583659&0&10#
36     {064092b3-625e-43bf-9eb5-dc8
37     45897dd59}\GPUPARAV
38     ValidPartitionCounts    : {32}
39     PartitionCount           : 32
40     TotalVRAM                : 1000000000
41     AvailableVRAM            : 1000000000
42     MinPartitionVRAM        : 0
43     MaxPartitionVRAM        : 1000000000
44     OptimalPartitionVRAM    : 1000000000
45     TotalEncode              : 18446744073709551615
46     AvailableEncode          : 18446744073709551615
47     MinPartitionEncode      : 0
48     MaxPartitionEncode      : 18446744073709551615
49     OptimalPartitionEncode   : 18446744073709551615
50     TotalDecode              : 1000000000
51     AvailableDecode          : 1000000000
52     MinPartitionDecode      : 0
53     MaxPartitionDecode      : 1000000000
54     OptimalPartitionDecode   : 1000000000
55     TotalCompute             : 1000000000
56     AvailableCompute         : 1000000000
57     MinPartitionCompute     : 0
58     MaxPartitionCompute     : 1000000000
59     OptimalPartitionCompute  : 1000000000
60     CimSession              : CimSession: .
61     ComputerName            : 4E305-10
62     IsDeleted                : False

```

Результат выполнения команды `Get-VMHostPartitionableGpu` указывает на два доступных графических процессора, которые поддерживают GPU Partitioning.

Первый GPU:

```

1     PCI#VEN\_10DE&DEV\_2487&SUBSYS\_40741458&REV\_A1#4&d0467e6&0&0008#

```

- графическое устройство с ядром Nvidia 3060 - будем использовать его.

2. `Add-VMGpuPartitionAdapter` — добавляет GPU-адаптер к виртуальной машине, чтобы она могла использовать часть ресурсов физического GPU. Этот адаптер предоставляет доступ к вычислительным, графическим и другим возможностям GPU [3].

Основные параметры:

- `-VMName` — имя виртуальной машины, к которой добавляется GPU-адаптер (обязательный параметр).

- **-InstancePath** — путь к конкретному GPU, который будет разделён и назначен виртуальной машине (обязательный параметр).
- **-MinPartitionVRAM**, **-MaxPartitionVRAM**, **-OptimalPartitionVRAM** — задают минимальный, максимальный и оптимальный объём видеопамати (VRAM), выделяемый виртуальной машине.
- **-MinPartitionEncode**, **-MaxPartitionEncode**, **-OptimalPartitionEncode** — определяют минимальное, максимальное и оптимальное количество ресурсов для кодирования видео.
- **-MinPartitionDecode**, **-MaxPartitionDecode**, **-OptimalPartitionDecode** — настраивают минимальное, максимальное и оптимальное количество ресурсов для декодирования видео.
- **-MinPartitionCompute**, **-MaxPartitionCompute**, **-OptimalPartitionCompute** — задают минимальный, максимальный и оптимальный объём вычислительных ресурсов (Compute), доступных виртуальной машине.

Введем следующую команду:

```

1 Add-VMGpuPartitionAdapter -VMName gpunode01 -InstancePath
   ↳ "\\?\PCI#VEN_10DE&DEV_2487&SUBSYS_40741458&
2 REV_A1#4&d0467e6&0&0008#{064092b3-625e-43bf-9eb5-dc8
3 45897dd59}\GPUPARAV" -MinPartitionVRAM 0 -MaxPartitionVRAM 1000000000
   ↳ -OptimalPartitionVRAM 1000000000 -MinPartitionEncode 0
   ↳ -MaxPartitionEncode 18446744073709551615 -OptimalPartitionEncode
   ↳ 18446744073709551615 -MinPartitionDecode 0 -MaxPartitionDecode
   ↳ 1000000000 -OptimalPartitionDecode 1000000000 -MinPartitionCompute 0
   ↳ -MaxPartitionCompute 1000000000 -OptimalPartitionCompute 1000000000

```

3. **Set-VM** — команда для изменения настроек виртуальной машины (VM), таких как параметры процессора, памяти и других ресурсов [3].

Принимаемые параметры:

- **-VMName** — имя виртуальной машины, параметры которой нужно изменить (обязательный параметр).
- **-GuestControlledCacheTypes** — задаёт возможность управления типами кэширования гостевой операционной системе. Значение **\$true** активирует эту функцию.
- **-LowMemoryMappedIoSpace** — определяет объём адресного пространства для работы устройств с низкоуровневой памятью (например, 3GB).
- **-HighMemoryMappedIoSpace** — задаёт объём адресного пространства для устройств с высокоуровневой памятью (например, 32GB).
- **-ProcessorCount** — задаёт количество процессоров, выделяемых виртуальной машине.
- **-DynamicMemory** — включает возможность использования динамической памяти для виртуальной машины.

Введем следующую команду:

```
1 PS C:\Windows\system32> Set-VM -VMName gpunode01
   ↪ -GuestControlledCacheTypes $true -LowMemoryMappedIoSpace 3GB
   ↪ -HighMemoryMappedIoSpace 32GB
```

Установка необходимого ПО для GPU

Для установки драйверов графического устройства необходимо скопировать их с хост-устройства с помощью scp.

Выполнение копирования драйверов GPU:

```
1 # Копируем содержимое папки с драйверами с хоста на виртуальную машину через SCP
2 # -r: копирование рекурсивно
3 # -P 40124: указание порта
4 scp -r -P 40124 C:\Windows\System32\DriverStore\FileRepository\
5 nv_dispig.inf_amd64_0afec3f2050014a0 user1@127.0.0.1:/tmp/
6
7 # Создаем директорию для драйверов
8 # -p - создавать поддиректории, если они не созданы
9 root@gpunode02:/tmp# mkdir -p /usr/lib/wsl/drivers
10
11 # Перемещаем скачанную папку драйвера из /tmp в директорию drivers
12 root@gpunode02:/usr/lib/wsl# mv /tmp/nv_dispig.inf_amd64_0afec3f2050014a0/
   ↪ /usr/lib/wsl/drivers/
13
```

Указанные драйвера обязательно должны лежать в директории `/usr/lib/wsl/drivers`, иначе они не будут найдены.

Конфигурация драйверов на виртуальной машине:

```
1 # Копируем библиотеку lib из Windows на виртуальную машину через SCP
2
3 PS C:\Windows\system32> scp -r -P 40124 C:\Windows\System32\lxss\lib
   ↪ user1@127.0.0.1:/tmp/
4
5 # Перемещаем скопированную библиотеку в директорию WSL
6 root@gpunode02:/usr/lib/wsl# mv /tmp/lib/ /usr/lib/wsl/
7
8 # Устанавливаем права доступа на директорию WSL: только чтение и выполнение (555)
9 root@gpunode02:/usr/lib# chmod -R 555 wsl/
10
11 # Устанавливаем владельцем директории WSL пользователя root и группу root
12 root@gpunode02:/usr/lib# chown -R root:root wsl/
```

Установка dxgkrnl с использованием готового shell скрипта.

dxgkrnl — это системный драйвер Windows, связанный с графической подсистемой. Его полное название — DirectX Graphics Kernel, и он отвечает за взаимодействие между операционной системой и аппаратным обеспечением GPU.

Установить его можно следующим образом:

1. Скачаем скрипт `install.sh` с использованием `curl` по пути: <https://content.staralt.dev/dxgkrnl-dkms/main/install.sh>;
2. Передадим скрипт на исполнение `bash` интерпретатору:

```
1 curl -fsSL https://content.staralt.dev/dxgkrnl-dkms/main/install.sh | sudo bash
   ↪ -es
```

Используемые флаги команды curl:

```
# -f: завершить с ошибкой, если произошел сбой HTTP-запроса.
# -s: режим, скрывающий прогресс загрузки.
# -S: отображение ошибок.
# -L: автоматическое следование за перенаправлениями.
```

Используемые флаги bash:

```
# -e: завершает выполнение скрипта при любой ошибке.
# -s: интерпретирует данные, подаваемые через стандартный ввод, как сценарий bash.
```

Проверка успешной установки CUDA

Для проверки успешной установки используются команды `lspci` и `nvidia-smi`.

1. `lspci` - Инструмент для отображения списка устройств PCI (Peripheral Component Interconnect), таких как видеокарты, сетевые карты, звуковые карты и другие компоненты.

Флаг `-v` - подробный вывод.

Введем команду:

```
1 lspci -v
```

Получим в выводе:

```
1 Kernel driver in use: dxgkrnl
```

Видим драйвер `dxgkrnl` - значит он корректно установился.

2. `nvidia-smi` - консольная утилита, предоставляемая NVIDIA, которая позволяет управлять и мониторить графические процессоры NVIDIA на уровне системы.

Введем команду:

```
1 nvidia-smi
```

Полученный вывод, представленный на рис. [1](#), говорит о корректной установке `nvidia cuda`.

```
user1@gpunode01:/usr/lib/wsl/drivers/nv_dispig.inf_amd64_0afec3f2050014a0$ nvidia-smi
Thu Nov 28 09:10:02 2024

+-----+
| NVIDIA-SMI 560.35.02                Driver Version: 560.94        CUDA Version: 12.6   |
+-----+-----+-----+
| GPU   Name                               Persistence-M   Bus-Id        Disp.A   Volatile Uncorr. ECC |
| Fan  Temp  Perf              Pwr:Usage/Cap     Memory-Usage   GPU-Util  Compute M. |
|=====+=====+=====+
|  0  NVIDIA GeForce RTX 3060              On          00000000:01:00:0  Off      N/A       |
|  0%   32C   P8               13W / 170W        0MiB / 12288MiB   0%      Default |
|                                           MIG M.       N/A       |
+-----+-----+-----+

+-----+
| Processes:                               GPU Memory |
| GPU   GI   CI        PID   Type   Process name                        Usage |
|=====+=====+=====+
| No running processes found |
+-----+

user1@gpunode01:/usr/lib/wsl/drivers/nv_dispig.inf_amd64_0afec3f2050014a0$
```

Рис. 1: Результат запуска команды nvidia-smi

1.5 Конфигурация NFS

NFS (Network File System) — это протокол для удалённого доступа к файлам по сети. Он позволяет компьютерам в сети обмениваться данными и работать с файлами, как если бы они находились на локальной файловой системе.

Для параллельного исполнения задач на нескольких узлах, нужно чтобы программы файлы находились на каждом из узлов в одной и той же директории. Вместо копирования файлов через scp, можно выделить один главный узел и директорию в нем, а к остальным узлам примонтировать этот каталог. Тогда файлы из выбранной машины будут видны на всех остальных машинах. Для этого и используется NFS. Основным узлом был выбран gpunode02.

Для настройки NFS нужно:

1. На каждой из виртуальных машин подготовить файл `/etc/hosts` для упрощения определений доменных имен.

В файле данные указываются в следующем формате:

```
1 IP-адрес Основное имя хоста Полное доменное имя хоста
```

Таким образом нужно, чтобы на каждом из узлов содержимое файла выглядело так:

```
1 127.0.0.1 localhost
2
3 10.0.108.1 cpunode01 cpunode01.hpc
4
5 10.0.108.2 cpunode02 cpunode02.hpc
6
7 10.0.108.3 gpunode01 gpunode01.hpc
8
9 10.0.108.4 gpunode02 gpunode02.hpc
```

После этого на главном узле необходимо добавить в файл `/etc/export` следующую строку:

```
1 /home/user1/mpi *(rw,nohide,no_subtree_check)
```

Содержимое обозначает:

`/home/user1/mpi` - каталог для экспорта, будет виден всем узлам;

`rw` - у всех узлов есть разрешения на чтение и на запись в каталог;

`nohide` - разрешает видимость подкаталогов;

`no_subtree_check` - отключение проверок подкаталогов для ускорения выполнения;

Далее необходимо применить экспорт план командой:

```
1 exportfs -avr
```

флаги которой значат:

-a - копировать все каталоги, указанные в плане;

-v - подробный вывод;

-r - перезагружает или пересчитывает все экспортированные каталоги, обновляя состояние экспортов;

После этого на всех остальных машинах, кроме основной нужно выполнить команду:

```
1 sudo mount -t nfs gpunode02:/home/user1/mpi /home/user1/mpi
```

Команда примонтирует экспортированный каталог в тот же путь на соответствующей машине.

Стоит обратить внимание, что после каждой перезагрузки машин - монтирование прерывается, и последние две команды требуется ввести еще раз.

1.6 Конфигурация slurm

SLURM - это высокопроизводительная система управления ресурсами и заданиями для вычислительных кластеров.

Параметры конфигурации SLURM задаются в файле `/etc/slurm/slurm.conf`. Для автоматического создания этого файла используется html конфигуратор: `/usr/share/doc/slurmctld/slurm-wlm-configurator.easy.html`.

Для получения файла в браузере воспользуемся python http сервером:

```
1 python3 -m http.server --directory /usr/share/doc/slurmctld/
```

Теперь конфигуратор доступен по адресу: `http://10.0.108.4:8000/slurm-wlm-configurator.easy.htm`.

10.0.108.4 - ip машины, на которой запущен сервер.

При открытии файла в браузере требуется поменять следующие поля:

- ClusterName = hpc- имя кластера;
- SlurmctldHost=gpunode02 - доменное имя контролирующего узла;
- PartitionName=production — название создаваемого раздела;
- CPUs=2 — количество процессоров на каждом узле;

Остальные параметры оставим без изменений:

- NodeAddr - адреса вычислительных узлов;
- MaxTime - ограничение на максимальное время выполнения задачи;
- Sockets - количество сокетов у одного узла;
- CoresPerSocket - количество физических ядер на одном сокете;
- ThreadsPerCore - количество логических потоков на одном физическом ядре;
- RealMemory - действительный объем доступной RAM;
- SlurmUser - имя Linux-пользователя, от имени которого будут выполняться службы Slurm;
- StateSaveLocation - путь, по которому slurmctld сохраняет состояние;
- SlurmdSpoolDir - путь, по которому slurmd сохраняет состояние;
- ReturnToService - определяет, по какому условию не отвечающий узел (DOWN) возвращается в работу (по умолчанию - в момент регистрации с верной конфигурацией только в случае, если узел отмечен DOWN из-за того, что он не отвечал на запросы);

- SchedulerType - механизм, контролирующий порядок исполнения задач. По умолчанию - Backfill, очередь задач (FIFO) с применением оптимизации: если имеется менее приоритетная задача и свободные ресурсы, то ставим эту задачу раньше;
- SwitchType - интерконнект между узлами. По умолчанию - None, не требуется специальной обработки (InfiniBand, Myrinet, Ethernet, и т.д.);
- MpiDefault - тип MPI по умолчанию (по этому параметру Slurm установит соответствующие переменные окружения). Этот тип можно переопределить с помощью параметра srun. По умолчанию - None (без поддержки PMI2 или PMIx);
- ProctrackType - алгоритм ассоциации процесса ОС и задачи Slurm. По умолчанию - Cgroup, используется Linux-утилиты cgroup для создания контейнера для задач и отслеживания процессов;
- SelectType - алгоритм выбора узла для задачи. По умолчанию - cons_tres, выделяются отдельные процессоры, память, графические процессоры и другие отслеживаемые ресурсы;
- SelectTypeParameters - указывает какие ресурсы отслеживаются у узлов для алгоритма выбора узла. По умолчанию - CR_Core, количество ядер;
- TaskPlugin - плагин для запуска задач. По умолчанию - Affinity, поддерживается привязка процессов к ядрам, например, по параметру srun -cpu-bind;
- SlurmctldLogFile - путь к файлу лога slurmctld;
- SlurmdLogFile - путь к файлу лога slurmd;

Полученный файл конфигурации необходимо скопировать на каждую машину, путь: /etc/slurm/slurm.conf.

Содержимое файла:

```

1      # slurm.conf file generated by configurator easy.html.
2      # Put this file on all nodes of your cluster.
3      # See the slurm.conf man page for more information.
4      #
5      ClusterName=hpc
6      SlurmctldHost=gpunode02
7      #
8      #MailProg=/bin/mail
9      MpiDefault=none
10     #MpiParams=ports=-#-#
11     ProctrackType=proctrack/cgroup
12     ReturnToService=1
13     SlurmctldPidFile=/run/slurmctld.pid
14     #SlurmctldPort=6817
15     SlurmdPidFile=/run/slurmd.pid
16     #SlurmdPort=6818

```

```

17 SlurmdSpoolDir=/var/lib/slurm/slurmd
18 SlurmUser=slurm
19 #SlurmdUser=root
20 StateSaveLocation=/var/lib/slurm/slurmctld
21 SwitchType=switch/none
22 TaskPlugin=task/affinity
23 #
24 #
25 # TIMERS
26 #KillWait=30
27 #MinJobAge=300
28 #SlurmctldTimeout=120
29 #SlurmdTimeout=300
30 #
31 #
32 # SCHEDULING
33 SchedulerType=sched/backfill
34 SelectType=select/cons_tres
35 SelectTypeParameters=CR_Core
36 #
37 #
38 # LOGGING AND ACCOUNTING
39 AccountingStorageType=accounting_storage/none
40 #JobAcctGatherFrequency=30
41 JobAcctGatherType=jobacct_gather/none
42 #SlurmctldDebug=info
43 SlurmctldLogFile=/var/log/slurm/slurmctld.log
44 #SlurmdDebug=info
45 SlurmdLogFile=/var/log/slurm/slurmd.log
46 #
47 #
48 # COMPUTE NODES
49 NodeName=cpunode[01-02],gpunode01 CPUs=2 State=UNKNOWN
50 PartitionName=production Nodes=ALL Default=YES MaxTime=INFINITE State=UP
51

```


1.7 Конфигурация MUNGE

MUNGE — это инструмент для аутентификации, который используется для обеспечения безопасности в кластерах под управлением SLURM.

Для корректного общения между машинами нужно, чтобы на всех из них был одинаковый ключ авторизации.

Для этого скопируем munge ключ с основной машины на все остальные:

```
1 scp /etc/munge/munge.key user1@cpunode01:/tmp
2 scp /etc/munge/munge.key user1@cpunode02:/tmp
3 scp /etc/munge/munge.key user1@gpunode01:/tmp
```

После этого на каждой из машин перенесем ключ в директорию /etc/munge/ и установим владельцем и группой - munge:

```
1 chgrp munge /etc/munge/munge.key
2 chown munge /etc/munge/munge.key
```

На каждой из машин перезагрузим сервисы munge и slurmd:

```
1 systemctl restart munge.service
```

На основной машине перезагрузим сервис slurmctld

```
1 sudo systemctl restart slurmctld
```

1.8 Настройка доступа по ssh без пароля

Для корректной работы параллельного исполнения MPI приложения на нескольких узлах - между главным узлом и всеми остальными должен быть настроен доступ по ssh без пароля, на основе ключей авторизации.

Для этого нужно:

1. На каждой машине (на хосте и на каждой виртуальной машине) сгенерировать пару ключей с помощью команды ssh-keygen. Эта команда создаст пару ключей: приватный (id_rsa) и публичный (id_rsa.pub);

2. Передать публичный ключ на ВМ, чтобы хост мог аутентифицироваться на виртуальных машинах без пароля:

```
1 type %USERPROFILE%\\.ssh\id_ed25519.pub |
2 ssh -p 40124 user1@localhost "cat >> .ssh/authorized_keys"
```

В этой команде:

type - вывод содержимого в windows;

%USERPROFILE% - переменная окружения, указывает путь к домашнему каталогу пользователя;

Символ | - передаем вывод следующей команде;

ssh -p 40124 user1@localhost - устанавливаем соединение с основной машиной;
"cat » .ssh/authorized_keys записываем вывод в authorized_keys. В этой папке лежат ключи для авторизации;

3. Передать ключ авторизации с главной машины на все остальные с помощью команды ssh-copy-id - автоматическое копирование публичного ssh ключа.

Введем на главной машине:

```
1 ssh-copy-id cpunode01
2 ssh-copy-id cpunode02
3 ssh-copy-id gpunode01
```

Теперь вход по ssh с хост системы на основную машину и с основной машины на все остальные возможен без пароля.

1.9 Решение практической задачи с помощью созданного кластера

На созданном вычислительном кластере требуется решить две задачи с использованием технологий параллельного программирования. В качестве исходных данных для задач берется предоставленный датасет.

Описание датасета

В датасете представлены логи выполненных задач на суперкомпьютерном кластере.

Содержатся данные о пользователе, который запустил задачу, и параметры запуска задачи.

Полный список данных, представленных в датасете:

1. JobIDRaw - ID задачи;
2. UID - ID пользователя;
3. GID - ID группы;
4. JobName - Название задачи;
5. Partition - Название машины;
6. ReqNodes - Число запрашиваемых узлов;
7. ReqCPUS - Число запрашиваемых CPU узлов;
8. AllocNodes - Реально используемое число узлов;
9. NodeList - Список используемых узлов;
10. Timelimit - Ограничение, заданное пользователем, отведенное под его задачу;
11. Submit - Время, когда задача была принята в очередь;
12. Start - Время, когда задача фактически была принята в очередь;
13. End - Время, когда задача была завершена;
14. Elapsed - Потраченное время;
15. CPUTimeRAW - Затраченное процессорное время CPU;
16. ElapsedRaw - Потраченное время (с);
17. TimelimitRaw - Ограничение, заданное пользователем, отведенное под его задачу (минуты);
18. Priority - Приоритет задачи;
19. State - Состояние выполнения задачи;

Данные разбиты построчно, в одной строке содержатся данные об одной конкретной задаче, разделителем между данными является символ «|».

Датасет представлен в txt формате, содержит порядка 1.5 млн. строк.

Постановка задачи

1. Отранжировать задачи по убыванию доли успешно завершенных заданий;
2. Нормализовать время ожидания заданий в очереди;

1.9.1 Разработка решения задачи №1

Для решения задачи №1 был выбран следующий алгоритм:

1. В процессе с ранком 0 происходит подсчет общего количества строк в файле, в зависимости от общего количества процессов, происходит подсчет строк для каждого из процессов. Каждому из процессов отправляются начало и конец интервала строк, который ему необходимо обработать.
2. В каждом из процессов происходит чтение нужных строк из файла с помощью функции `getline`. В процессе чтения файла происходит заполнение хэш-таблицы `имя_задачи - [количество успешно завершенных задач, количество НЕуспешно завершенных задач]`.
3. После чтения файла данные из хэш-таблиц на каждом процессе записываются в векторы и передаются главному потоку.
4. В главном процессе для каждого из узлов считается доля успешно завершенных заданий и задания сортируются с помощью дерева(контейнера `std::multimap`) по доле успешно завершенных заданий.
5. После сортировки данные выводятся в файл основным потоком.

В алгоритме используются только вычислительные ресурсы CPU.

Обоснование выбора алгоритма

Представленный алгоритм был выбран исходя из принципов уменьшения алгоритмической сложности вычислений:

Максимальная временная сложность описанных операций: $O(N \log(N))$ - при сортировке, значит сложность всего алгоритма: $O(N \log(N))$. Поэтому в первом этапе используется хэш-таблица, сложность вставки в которую $O(N)$ и ее использование не повысит общую вычислительную сложность.

Данное решение не оптимально по памяти: во время решения хранятся три копии данных: в хэш-таблице, векторе и дереве, однако был выбран именно такой способ, поскольку использование `std` контейнеров упрощает реализацию и предоставляет проверенные временем алгоритмы.

Недостатком реализации является использование функции `getline` для чтения данных из файла. Каждому из процессов, все равно придется прочитать каждый символ из файла(для подсчета символов переноса строки), для того, чтобы найти начало нужного ему интервала. Из-за этого преимущества от распараллеливания

задач теряются. Оптимальным решением было бы распределять данные между процессами не построчно, а побайтово.

Основные моменты реализации

1. Алгоритм распределения строк между процессами:

```

1
2 Input: size - количество процессов, amount_str_general - общее количество строк в
   ↪ файле
3 Output: interval_preparing - интервалы строк для каждого процесса
4
5 if rank == 0 then
6     amount_str_for_each ← amount_str_general / size
7     amount_str_for_last ← amount_str_general -(amount_str_for_each * (size - 1))
8
9     for i from 1 to size - 2 do
10         interval_preparing[0] ← (amount_str_for_each * i + 1)
11         interval_preparing[1] ← (amount_str_for_each * (i + 1))
12         MPI_Send(interval_preparing, 2, MPI_LONG, i, TAG, MPI_COMM_WORLD)
13
14     interval_preparing[0] ← (amount_str_for_each * (size - 1) + 1)
15     interval_preparing[1] ← amount_str_general
16     MPI_Send(interval_preparing, 2, MPI_LONG, size - 1, TAG, MPI_COMM_WORLD)
17
18     interval_preparing[0] ← 1
19     interval_preparing[1] ← amount_str_for_each
20 else
21     MPI_Recv(interval_preparing, 2, MPI_LONG, MPI_ANY_SOURCE, MPI_ANY_TAG,
   ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE)
22

```

2. Алгоритм чтения строк из файла для составления хэш таблицы:

```

1
2 Input: file - входной файл, interval_preparing - интервалы строк
3 Output: mp - хэш-таблица: имя_задачи - [количество успешно завершенных задач,
   ↪ количество НЕуспешно завершенных задач]
4
5 while getline(file, curr_line) do
6     if curr_line_number < interval_preparing[0] then
7         continue
8     end
9
10    if curr_line_number <= interval_preparing[1] then
11        p ← getJobNameAndStatusFromString(curr_line)
12        if p.second == COMPLETED then
13            mp[p.first].first ← mp[p.first].first + 1
14        else
15            mp[p.first].second ← mp[p.first].second + 1
16        end
17    else
18        break
19    end

```

20
21
22

end

3. Алгоритм сортировки полученных данных на основном процессе с помощью контейнера `std::multimap`:

1
2
3
4
5
6
7
8
9
10
11
12
13

```
Input: recieved_data - вектор полученных данных
Output: mp - отсортированный словарь с данными

for each it in recieved_data do
    if job_name of it not in mp then
        mp[it.job_name] ← (it.amount_completed, it.amount_uncompleted)
    else
        mp[it.job_name].first ← mp[it.job_name].first + it.amount_completed
        mp[it.job_name].second ← mp[it.job_name].second + it.amount_uncompleted
    end
end
```

Исходный код для решения задачи №1 представлен в приложении А (заголовочный файл - приложение С).

1.9.2 Разработка решения задачи №2

1. В процессе с ранком 0 происходит подсчет общего количества строк в файле, в зависимости от общего количества процессов, происходит подсчет строк для каждого из процессов. Каждому из процессов отправляются начало и конец интервала строк, который ему необходимо обработать.
2. В каждом из процессов происходит чтение нужных строк из файла с помощью функции `getline`. В процессе чтения файла происходит заполнение двух векторов: вектора `id` задач и вектора времени, которое задача была в ожидании.
3. После заполнения векторов каждый из процессов считает максимальное значение времени ожидания для своей выборки данных, отправляет результат в главный поток. Главный процесс вычисляет максимум из представленных значений и отправляет результаты обратно - всем остальным процессам.
4. После расчета максимального значения времени ожидания каждый процесс вызывает функцию нормализации для CPU или GPU узла соответственно. Функция для нормализации на GPU написана с использованием CUDA.
5. После завершения расчетов каждый из процессов записывает результаты вычислений в файл.

Было написано два варианта реализации функции нормализации чисел, для возможности запуска программы и на GPU и на CPU узлах. Каждая из функций находится в файле динамически разделяемой библиотеки, которые компилируются до вызова программы. Уже скомпилированный код функций вызывается из основного тела программы во время исполнения, в зависимости от типа узла, на котором исполняется процесс. Тип узла записан в переменную окружения каждой из виртуальных машин.

Обоснование выбора алгоритма

GPU узлы используются для оптимизации вычислений большого количества однотипных задач, которые можно выполнять пораздельно. Именно такой задачей и является задача нормализации множества чисел.

Основные моменты реализации

1. Алгоритм распределения строк между процессами - повторяет алгоритм распределения из решения предыдущей задачи.

2. Алгоритм чтения строк из файла для составления векторов(id задачи и время ожидания в очереди):

```

1 Input: file - входной файл, interval_preparing - интервалы строк
2 Output: vec_job_id - вектор id задач
3         vec_time - вектор времён ожидания задач в очереди
4
5 while getline(file, curr_line) do
6     if curr_line_number < interval_preparing[0] then
7         curr_line_number++
8         continue
9     end
10
11    if curr_line_number <= interval_preparing[1] then
12        p ← getIdAndTimeFromString(curr_line) -- получение id и времени ожидания из
13        ↪ строки
14        vec_job_id[index_counter]=p.first
15        vec_time[index_counter]=p.second
16        curr_line_number++
17        index_counter++
18    else
19        break
20    end
end

```

3. Алгоритм нормализации времени ожидания в очереди:

```

1 Input: vec_time - массив значений времени, max_value - максимальное значение
2 ↪ времени ожидания, size - размер массива
3 Output: vec_normal_time - массив нормализованных значений
4
5 for i from 0 to size - 1 do
6     vec_normal_time[i] ← vec_time[i] / max_value
end

```

Исходный код для решения задачи №2 представлен в приложении В(заголовочный файл - приложение С). В приложениях D и E представлены исходные файлы для динамически разделяемых библиотек для CPU и GPU узлов соответственно.

1.10 Проведение экспериментов

Результаты проведения экспериментов показаны в таблицах на рис 2 и 3 для задач №1 и №2 соответственно. В таблицах приведено: общее количество запущенных процессов, количество процессов для каждого из узлов, время выполнения задачи. Количество представленных экспериментов выше для задачи №2, поскольку структура вычислений на гри и сри узлах для этой задачи отличаются, и тесты запуска на сри и гри узлах будут давать различные результаты.

Общее количество потоков	gpunode02	gpunode01	cpunode02	cpunode01	Время выполнения(с)
1	1	0	0	0	7.47
2	1	1	0	0	4.64
4	1	1	1	1	3.62
6	2	2	1	1	3.54
8	2	2	2	2	3.44
16	4	4	4	4	8.56

Рис. 2: Проведение экспериментов для задачи №1.

Общее количество потоков	gpunode02	gpunode01	cpunode02	cpunode01	Время выполнения(с)
1	1	0	0	0	6.51
1	0	0	1	0	6.08
2	1	1	0	0	3.89
2	0	0	1	1	3.54
4	1	1	1	1	3.27
6	1	1	2	2	2.73
6	2	2	1	1	3.04
8	2	2	2	2	3.12
16	4	4	4	4	9.24

Рис. 3: Проведение экспериментов для задачи №2.

По результатам проведенных экспериментов можно сделать следующие выводы:

Для задачи №1: Оптимальное количество процессов для каждого из узлов: 2. При большем увеличении количества процессов - накладные расходы на их создание растут, и преимущества от распараллеливания теряется.

Лучший результат по времени: 3.44 с.

Для задачи №2: Оптимальное количество процессов: по одному на каждый из GPU и по два на каждый из CPU узлов.

Запуск же процессов только на CPU узлах показывает лучшие результаты по времени, чем запуск только на GPU узлах. Такие результаты объясняются тем, что выделение ресурсов на GPU требует затрат по времени, которые не компенсируются скоростью вычислений из-за относительно небольшого объема данных в задаче: общий объем передаваемых на GPU данных - всего около 3ex МБ.

При большем количестве процессов преимущества параллельного исполнения так же теряются.

Лучший результат по времени: 2.73 с.

Общие выводы по проведенным экспериментам:

Для проверки скорости вычислений кластера и нахождения оптимальной конфигурации запуска - объем исследуемого датасета недостаточен. Накладные расходы на создание процессов невеличают преимущества. К тому же возможные оптимизации, применяемые операционной системой и планировщиком задач могут существенно влиять на показываемые результаты при относительно небольшом объеме тестовых данных.

2 Заключение

Итак, в ходе работы был создан виртуальный гетерогенный распределенный вычислительный кластер, состоящий из 2ух GPU и 2ух CPU узлов, с использованием гипервизора Hyper-V.

Для этого было сделано:

- Созданы и настроены узлы кластера в гипервизоре, установлена ОС - ubuntu server;
- Настроена сеть между виртуальными машинами;
- Настроена распределенная файловая система NFS между машинами;
- Настроено программное обеспечение, необходимое для работы с MPI и CUDA.
- Настроена конфигурация SLURM и munge;

Для тестирования кластера были написаны программы с использованием технологий CUDA и MPI для анализа предоставленного датасета.

Были проведены эксперименты с различной конфигурацией запуска программ.

Список источников

- [1] TOP500. Home page.(Дата обращения 16.01.2025)<https://www.top500.org/>
- [2] Суперкомпьютерный центр «Политехнический»(Дата обращения 16.01.2025)<https://research.spbstu.ru/skc/>
- [3] Microsoft. Документация PowerShell.(Дата обращения 17.01.2025)<https://learn.microsoft.com/en-us/powershell/module/hyper-v/?view=windowsserver2025-ps>
- [4] Работа с динамически разделяемыми библиотеками в Linux.(Дата обращения 15.01.2025) <https://pvoid.pro/index.php/articles/41-dl-libraries>
- [5] Microsoft. Документация MPI.(Дата обращения 16.01.2025) <https://learn.microsoft.com/ru-ru/message-passing-interface/microsoft-mpi>

Приложение А

```
1
2 #include "header.h"
3
4
5 void sendDataJobStructureVec(vector<DataJobStructure>& data) {
6     int size = data.size();
7     cout<<"Отправляю "<< size * sizeof(DataJobStructure)<<" байт"<<endl;
8     MPI_Send(&size, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD); // Отправляем размер
        ↳ вектора
9     MPI_Send(data.data(), size * sizeof(DataJobStructure), MPI_BYTE, 0, TAG,
        ↳ MPI_COMM_WORLD); // Отправляем данные
10 }
11
12 void receiveDataJobStructureVec(vector<DataJobStructure>& data, int proccess_id) {
13     int num_elements;
14     MPI_Recv(&num_elements, 1, MPI_INT, proccess_id, 0, MPI_COMM_WORLD,
        ↳ MPI_STATUS_IGNORE); // Принимаем размер данных
15     cout<<"Принимаю "<< num_elements * sizeof(DataJobStructure) <<" байт"<<endl;
16     data.resize(num_elements); // Резервируем память для данных
17     MPI_Recv(data.data(), num_elements * sizeof(DataJobStructure), MPI_BYTE,
        ↳ proccess_id, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // Принимаем данные
18 }
19
20
21 int main(int argc, char *argv[]) {
22     Timer timer;
23     int rank, size;
24     long interval_preparing[2]; // [строка с которой начинаю обрабатывать, строка
        ↳ которой заканчиваем обработку]
25     MPI_Init(&argc, &argv);
26
27     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
28     MPI_Comm_size(MPI_COMM_WORLD, &size);
29     int size_calculate = size-1;
30     //std::cout << "Процесс " << rank << " из " << size << std::endl;
31     if(rank==0){
32         long amount_str_general;
33         long amount_str_for_each;
34         long amount_str_for_last;
35
36         amount_str_general = getAmountString(FILE_NAME);
37         cout<<"amount proccess = "<<size<<endl;
38         cout<<"amount string general = "<<amount_str_general<<endl;
39         amount_str_for_each = amount_str_general/size;
40         amount_str_for_last = amount_str_general - (amount_str_for_each *
            ↳ (size-1));
41         cout<<"Количество строк для каждого процесса кроме последнего:
            ↳ "<<amount_str_for_each<<endl;
42         cout<<"Количество строк для последнего процесса:
            ↳ "<<amount_str_for_last<<endl;
43         for(int i=1;i<size-1;i++){
44             interval_preparing[0] = (amount_str_for_each*i+1);
```

```

45         interval_preparing[1] = (amount_str_for_each*(i+1));
46         MPI_Send(interval_preparing, 2, MPI_LONG, i, TAG, MPI_COMM_WORLD);
47     }
48     interval_preparing[0] = (amount_str_for_each*(size-1)+1);
49     interval_preparing[1] = amount_str_general;
50     MPI_Send(interval_preparing, 2, MPI_LONG, size-1, TAG, MPI_COMM_WORLD); //
    ↳ TAG - тег сообщения, MPI_COMM_WORLD - коммутатор
51     interval_preparing[0] = 1;
52     interval_preparing[1] = amount_str_for_each;
53 }
54 else{
55     MPI_Recv(&interval_preparing, 2, MPI_LONG, MPI_ANY_SOURCE, MPI_ANY_TAG,
    ↳ MPI_COMM_WORLD, MPI_STATUS_IGNORE); // MPI_STATUS_IGNORE - принимаем с
    ↳ любым статусом
56 }
57
58 ifstream file(FILE_NAME);
59 unordered_map<string, pair<long, long>> mp; //(имя_работы) = [кол-во успешных,
    ↳ кол-во неуспешных]
60 long curr_line_number=0;
61 string curr_line;
62
63 //cout<<"Перед циклом " <<"rank proccess = " <<rank<<"
    ↳ "<<interval_preparing[0]<<" "<<interval_preparing[1]<<endl;
64 while(getline(file, curr_line)){
65     if(curr_line_number<interval_preparing[0]){
66         curr_line_number++;
67         continue;
68     }
69     else if(curr_line_number<=interval_preparing[1]){
70         pair<string, string>p = getJobNameAndStatusFromString(curr_line);
71
72         if(p.second==COMPLETED){
73             mp[p.first].first++;
74         }else{
75             mp[p.first].second++;
76         }
77         curr_line_number++;
78     }
79     else{
80         break;
81     }
82 }
83
84
85
86 if(rank!=0){
87     vector<DataJobStructure> vec_data(mp.size());
88     int index_counter=0;
89     for( auto it = mp.begin(); it!=mp.end(); ++it){
90         DataJobStructure djs;
91         strncpy(djs.job_name, it->first.c_str(), sizeof(djs.job_name) - 1); //
    ↳ куда, откуда, макс кол-во символов для копирования
92         djs.job_name[sizeof(djs.job_name) - 1] = '\0';

```

```

93         djs.amount_completed=it->second.first;
94         djs.amount_uncompleted=it->second.second;
95         vec_data[index_counter]=djs;
96         index_counter++;
97         //cout<<it->first<<" "<< it->first.size()<<" "<<
           ↪ vec_data[index_counter].job_name<<endl;
98     }
99     for(auto e: vec_data){
100         //e.print();
101     }
102     sendDataJobStructureVec(vec_data);
103 }
104
105 if(rank==0){
106     for(int i=1;i<size;i++){
107         vector<DataJobStructure> recieved_data;
108         receiveDataJobStructureVec(recieved_data,i);
109         for(auto it = recieved_data.begin();it!=recieved_data.end();++it){
110             if(mp.find(it->job_name)!=mp.end()){
111                 mp[it->job_name]=make_pair(it->amount_completed,
           ↪ it->amount_uncompleted);
112             }
113             else{
114                 mp[it->job_name].first+=it->amount_completed;
115                 mp[it->job_name].second+=it->amount_uncompleted;
116             }
117         }
118     }
119     multimap<double,string, greater<double>> mp_proporcion_jobName;
120     for(auto it =mp.begin();it!=mp.end();it++){
121         double key = (double)it->second.first/(double)
122         (it->second.first+it->second.second);
123         string val = it->first;
124         mp_proporcion_jobName.insert(make_pair(key,val));
125     }
126     writeOutputData(mp_proporcion_jobName);
127 }
128
129 MPI_Finalize();
130 return 0;
131
132 }
133

```

Приложение В

```
1 #include "header.h"
2 using namespace std;
3 void getGeneralMaxTimeElapsed(long& max_time, int rank, int size){
4     if(rank!=0){
5         MPI_Send(&max_time, 1, MPI_LONG, 0, 2, MPI_COMM_WORLD);
6     }
7     else{
8         vector<long> vec_of_max_val;
9         vec_of_max_val.push_back(max_time);
10        for(int i=1;i<rank;i++){
11            long help_val;
12            MPI_Recv(&help_val, 1, MPI_LONG, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD,
13                ↪ MPI_STATUS_IGNORE);
14            vec_of_max_val.push_back(help_val);
15        }
16        max_time = *max_element(vec_of_max_val.begin(),vec_of_max_val.end());
17    }
18    MPI_Bcast(&max_time, 1, MPI_LONG, 0, MPI_COMM_WORLD);
19 }
20 void writeOutputInOrder(int rank, int size,vector<long>& vec_job_id, vector<long>&
21 ↪ vec_time, vector<double>& vec_normal_time) {
22     // Каждый процесс будет ждать своей очереди
23     for (int i = 0; i < size; ++i) {
24         if (rank == i) {
25             if(rank==0){
26                 outputThreeVecToFile(vec_job_id,vec_time,vec_normal_time,1);
27             }
28             else{
29                 outputThreeVecToFile(vec_job_id,vec_time,vec_normal_time,0);
30             }
31             cout << "Процесс№ " << rank << " записал в файл." << endl;
32         }
33         MPI_Barrier(MPI_COMM_WORLD); // Все процессы ждут друг друга
34     }
35 }
36 int main(int argc, char* argv[]) {
37     int rank, size;
38     Timer *timer =nullptr;
39     int amount_blocks=256;
40     int amount_threads;
41     long interval_preparing[2];// [строка с которой начинаю обрабатывать, строка
42     ↪ которой заканчиваем обработку]
43     MPI_Init(&argc, &argv);
44
45     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
46     MPI_Comm_size(MPI_COMM_WORLD, &size);
47     if(rank==0){
48         timer=new Timer();
49     }
50     int size_calculate = size-1;
```



```

50  if(rank==0){
51      long amount_str_general;
52      long amount_str_for_each;
53      long amount_str_for_last;
54
55      amount_str_general = getAmountString(FILE_NAME);
56      amount_threads = amount_str_general/amount_blocks + 1;
57
58      cout<<"amount  process = "<<size<<endl;
59      cout<<"amount string general = "<<amount_str_general<<endl;
60      amount_str_for_each = amount_str_general/size;
61      amount_str_for_last = amount_str_general - (amount_str_for_each *
        ↳ (size-1));
62      cout<<"Количество строк для каждого процесса кроме последнего:
        ↳ "<<amount_str_for_each<<endl;
63      cout<<"Количество строк для последнего процесса:
        ↳ "<<amount_str_for_last<<endl;
64      for(int i=1;i<size-1;i++){
65          interval_preparing[0] = (amount_str_for_each*i+1);
66          interval_preparing[1] = (amount_str_for_each*(i+1));
67          //cout<<i<<" "<<interval_preparing[0]<<"
        ↳ "<<interval_preparing[1]<<endl;
68          MPI_Send(interval_preparing, 2, MPI_LONG, i, 0, MPI_COMM_WORLD);
69      }
70      interval_preparing[0] = (amount_str_for_each*(size-1)+1);
71      interval_preparing[1] = amount_str_general;
72      MPI_Send(interval_preparing, 2, MPI_LONG, size-1, 0, MPI_COMM_WORLD); //
        ↳ интервал
73      interval_preparing[0] = 1;
74      interval_preparing[1] = amount_str_for_each;
75      for(int i=1;i<size;i++){
76          MPI_Send(&amount_threads, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
77      }
78  }
79  else{
80      MPI_Recv(&interval_preparing, 2, MPI_LONG, MPI_ANY_SOURCE, 0,
        ↳ MPI_COMM_WORLD, MPI_STATUS_IGNORE); // MPI_STATUS_IGNORE - принимаем с
        ↳ любым статусом
81      MPI_Recv(&amount_threads, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
        ↳ MPI_STATUS_IGNORE);
82  }
83  ifstream file(FILE_NAME);
84  string curr_line;
85  long curr_line_number=0;
86  long index_counter=0;
87  long amount_string_for_this_process =
        ↳ interval_preparing[1]-interval_preparing[0]+1;
88  vector<long> vec_job_id(amount_string_for_this_process);
89  vector<long> vec_time(amount_string_for_this_process);
90
91  while(getline(file,curr_line)){
92      if(curr_line_number<interval_preparing[0]){
93          curr_line_number++;
94          continue;

```

```

95     }
96     else if(curr_line_number<=interval_preparing[1]){
97         pair<long, long> p= getSubmitAndStartFromString(curr_line);
98         vec_job_id[index_counter]=p.first;
99         vec_time[index_counter]=p.second;
100         curr_line_number++;
101         index_counter++;
102     }
103     else{
104         break;
105     }
106 }
107
108 long max_time_elapsed = findMaxElement(vec_time);
109 getGeneralMaxTimeElapsed(max_time_elapsed,rank,size);
110
111 if(rank==0){
112     cout<<"Max value for normalization: "<<max_time_elapsed<<endl;
113 }
114
115 void* handle;
116 const char* host_type = std::getenv("HOST_TYPE");
117 if(strcmp(host_type,"GPU")==0){
118     cout<<"Процесс#: "<<rank<<" выполняется на GPU "<<endl;
119     handle = dlopen("/home/user1/mpi/cuda_calculate_shared/gpu_shared.so",
120         ↪ RTLD_LAZY);
121 }
122 else{
123     cout<<"Процесс#: "<<rank<<" выполняется на CPU "<<endl;
124     handle = dlopen("/home/user1/mpi/cuda_calculate_shared/cpu_shared.so",
125         ↪ RTLD_LAZY);
126 }
127 double* (*normalization)(long*,long,long,int,int);
128 normalization = (double* (*)(long*,long,long,int,int)) dlsym(handle,
129     ↪ "normalization");
130
131 double* vec_normal_time_ptr =
132 normalization(vec_time.data(),amount_string_for_this_process,max_time_elapsed
133 ,amount_blocks,amount_threads);//
134 vector<double> vec_normal_time(vec_normal_time_ptr,
135     vec_normal_time_ptr+amount_string_for_this_process);
136 writeOutputInOrder(rank,size,vec_job_id,vec_time,vec_normal_time);
137 free(vec_normal_time_ptr);
138 dlclose(handle);
139 if(rank==0){
140     delete timer;
141 }
142 MPI_Finalize();
143 return 0;
144 }

```

Приложение С

```
1 #pragma once
2 #include <stdio.h>
3 #include <iostream>
4 #include <sstream>
5 #include <fstream>
6 #include <vector>
7 #include <chrono>
8 #include <iomanip>
9 #include <algorithm>
10 using namespace std;
11 using namespace std::chrono;
12
13 const string FILE_NAME = "test_data";
14 const string OUTPUT_FILE_NAME = "output_result";
15
16 long parseDateTimeToSeconds(const string& dt) {
17     if (dt.size() < 19) return 0;
18     std::tm tmStruct = {};
19     tmStruct.tm_year = std::stoi(dt.substr(0, 4)) - 1900;
20     tmStruct.tm_mon = std::stoi(dt.substr(5, 2)) - 1;
21     tmStruct.tm_mday = std::stoi(dt.substr(8, 2));
22     tmStruct.tm_hour = std::stoi(dt.substr(11, 2));
23     tmStruct.tm_min = std::stoi(dt.substr(14, 2));
24     tmStruct.tm_sec = std::stoi(dt.substr(17, 2));
25     time_t t = std::mktime(&tmStruct);
26     if (t == -1) return 0;
27     return (long long)t;
28 }
29
30 long getAmountString(string file_name){
31     string command = "wc -l < " + file_name;
32
33     FILE* fp = popen(command.c_str(), "r");
34     if (fp == nullptr) {
35         cerr << "Ошибка при выполнении команды!" << endl;
36         return -1;
37     }
38
39     char buffer[128];
40     if (fgets(buffer, sizeof(buffer), fp) != nullptr) {
41         int line_count = std::atoi(buffer);
42         pclose(fp);
43         return line_count;
44     }
45
46     pclose(fp);
47     return -1;
48 }
49 template<typename T>
50 void printVec(vector<T> vec){
51     for(auto e:vec){
52         cout<<e<<" ";
```

```

53     }
54     cout<<endl;
55 }
56
57 long findMaxElement(std::vector<long>& vec) {
58     if (vec.empty()) {
59         throw std::invalid_argument("Вектор пуст. Максимальный элемент невозможно
        ↳ найти.");
60     }
61     long ret = *max_element(vec.begin(),vec.end());
62     return ret;
63 }
64
65 pair<long,long>getSubmitAndStartFromString(const string& str){
66     stringstream job_id;
67     stringstream a;
68     stringstream b;
69     int divider_counter=0;
70     for(int i=0;i<str.size();i++){
71         if(str[i]=='|'){
72             divider_counter++;
73             continue;
74         }
75         if(divider_counter==0){
76             job_id<<str[i];
77         }
78         if(divider_counter==1){
79             a<<str[i];
80         }
81         if(divider_counter==3){
82             b<<str[i];
83         }
84     }
85     long begin_waiting = parseDateTimeToSeconds(a.str());
86     long end_waiting = parseDateTimeToSeconds(b.str());
87     long id;
88     job_id>>id;
89     return make_pair(id,end_waiting-begin_waiting);
90 }
91
92 void outputThreeVecToFile(vector<long> vec_job_name, vector<long>vec_time,
    ↳ vector<double> vec_normal_time, int marker_overwrite){
93     ofstream outFile(OUTPUT_FILE_NAME, marker_overwrite ? ios::trunc : ios::app);
94     ↳ // Если marker_overwrite == 1, файл перезаписывается, иначе данные
95     ↳ добавляются в конец
96
97     if (!outFile) {
98         cerr << "Ошибка при открытии файла: " << OUTPUT_FILE_NAME << endl;
99         return;
100     }
101     long size=vec_job_name.size();
102     if(marker_overwrite){
103         outFile<<"job_id"<<"|"<<"time"<<"|"<<"normal_time"<<endl;
104     }

```

```

103     for(long index=0;index<size;index++){
104         outFile<<vec_job_name[index]<<"|"<<vec_time[index]
105         <<"|"<<vec_normal_time[index]<<endl;
106     }
107     outFile.close();
108 }
109 pair<string,string>getJobNameAndStatusFromString(const string& str){
110     stringstream a;
111     stringstream b;
112     int divider_counter=0;
113     for(int i=0;i<str.size();i++){
114         if(str[i]=='|'){
115             divider_counter++;
116             continue;
117         }
118         if(divider_counter==3){
119             a<<str[i];
120         }
121         if(divider_counter==20){
122             b<<str[i];
123         }
124     }
125     return make_pair(a.str(),b.str());
126 }
127
128 pair<long,string>getJobIdAndStatusFromString(const string& str){
129     stringstream a;
130     stringstream b;
131     int divider_counter=0;
132     for(int i=0;i<str.size();i++){
133         if(str[i]=='|'){
134             divider_counter++;
135             continue;
136         }
137         if(divider_counter==0){
138             a<<str[i];
139         }
140         if(divider_counter==20){
141             b<<str[i];
142         }
143     }
144     long job_id;
145     a>>job_id;
146     return make_pair(job_id,b.str());
147 }
148
149
150 void printPairString_LongLong(pair<string,pair<long,long>> p){
151     cout<<p.first<<" = "<<"["<<p.second.first<<" , "<<p.second.second<<"]"<<endl;
152 }
153 void printMp(unordered_map<string,pair<long,long>> mp){
154     cout<<endl<<"Печатаю хэш таблицу:"<<endl;
155     for(auto e: mp){
156         printPairString_LongLong(e);

```

```

157     }
158     cout<<endl<<endl;
159 }
160
161 long getAmountString(string file_name){
162     string command = "wc -l < " + file_name;
163
164     FILE* fp = popen(command.c_str(), "r");
165     if (fp == nullptr) {
166         cerr << "Ошибка при выполнении команды!" << endl;
167         return -1;
168     }
169
170     char buffer[128];
171     if (fgets(buffer, sizeof(buffer), fp) != nullptr) {
172         int line_count = std::atoi(buffer);
173         pclose(fp);
174         return line_count;
175     }
176
177     pclose(fp);
178     return -1;
179 }
180
181 void DataJobStructure::print(){
182     cout<<"Job id: "<<this->job_name<<" , completed: "<<this->amount_completed<<" ,
183     ↪  uncompleted: "<<this->amount_uncompleted<<endl;
184 }
185
186 void writeOutputData(multimap<double,string,greater<double>>mp){
187     ofstream outFile(OUTPUT_FILE_NAME);
188
189     if (!outFile) {
190         cerr << "Ошибка при открытии файла: " << OUTPUT_FILE_NAME << endl;
191         return;
192     }
193     for(const auto& e:mp){
194         outFile<<e.second<<"|"<<e.first<<endl;
195     }
196     outFile.close();
197 }
198
199 class Timer {
200 public:
201     Timer() {
202         // Сохраняем текущее время в момент создания объекта
203         start_time = high_resolution_clock::now();
204     }
205     ~Timer() {
206         // В момент уничтожения объекта вычисляем разницу
207         auto end_time = high_resolution_clock::now();
208         auto duration = chrono::duration_cast<chrono::microseconds>(end_time -
209         ↪  start_time);

```

```

209         // Переводим разницу в секунды и миллисекунды
210         double seconds = duration.count() / 1000000.0;
211
212         // Выводим результат в формате сек.миллисек
213         cout << "Time elapsed: "
214              << fixed << setprecision(6) << seconds
215              << " seconds" << endl;
216     }
217 private:
218     high_resolution_clock::time_point start_time;
219 };

```

Приложение D

```
1 #include "cpu_shared.h"
2 void normalizationCPU(long* vec_time, double* vec_normal_time, long max_value, long
   ↪ size){
3     for(int i=0;i<size;i++){
4         vec_normal_time[i]=(double)vec_time[i]/(double)max_value;
5     }
6 }
7 extern "C" double* normalization( long* vec_time, long vec_size, long max_value, int
   ↪ amount_threads, int amount_blocks) {
8     long size = vec_size;
9     double *vec_normal_time = (double*)malloc(size * sizeof(double));
10    normalizationCPU(vec_time, vec_normal_time, max_value, size);
11    return vec_normal_time;
12 }
```


Приложение Е

```
1 #include "gpu_shared.h"
2 __global__ void normalizationGPU(long* vec_time, double* vec_normal_time, long
   ↳ max_value, long size) {
3     long idx = blockIdx.x * blockDim.x + threadIdx.x;
4     if (idx < size) {
5         vec_normal_time[idx] = (double)vec_time[idx] / (double)max_value; //
   ↳ Нормализация
6     }
7 }
8 extern "C" double* normalization( long* vec_time, long vec_size, long max_value, int
   ↳ amount_blocks, int amount_threads) {
9     long size = vec_size;
10    double *vec_normal_time = (double*)malloc(size * sizeof(double));
11    long *d_vec_time;
12    double *d_vec_normal_time;
13    cudaMalloc(&d_vec_time, size * sizeof(long));
14    cudaMalloc(&d_vec_normal_time, size * sizeof(double));
15    cudaMemcpy(d_vec_time, vec_time, size * sizeof(long), cudaMemcpyHostToDevice);
16
17    normalizationGPU<<<amount_threads,
   ↳ amount_blocks>>>(d_vec_time, d_vec_normal_time, max_value, size);
18
19    cudaMemcpy(vec_normal_time, d_vec_normal_time, size *
   ↳ sizeof(double), cudaMemcpyDeviceToHost);
20    cudaFree(d_vec_time);
21    cudaFree(d_vec_normal_time);
22    return vec_normal_time;
23 }
24
25
```