Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра ПМиК
Допустить к защите

Зав.каф. _____ Фионов А. Н.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Разработка инструментария для планирования и анализа расчётов на гибридных кластерах течений разреженного газа.

Магистерская диссертация

по направлению 09.04.01 «Информатика и вычислительная техника»

Студент Немцев М. С. / /

Руководитель Малков Е. А. / /

Новосибирск 2016 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Для решения научных, технических и промышленных задач, требующих значительных ресурсов, широкое распространение получили кластерные системы [12]. Применение вычислительных комплексов позволяет эффективно решать и моделировать сложные задачи. В качестве решения и тестирования реализованного метода, за основу был взят гибридный кластер ИВЦ НГУ [13].

Выполняющиеся на гибридном кластере задачи, имеют большое количество выходных данных. Данные в свою очередь ежедневно записываются, методом моментального снимка « Snapshot » на жесткий диск. Время на выполнение задачи связанной с вычислениями может занимать от месяца и более.

Большое количество программного обеспечения, написанного для вычислительной техники, не обладает необходимым запасом вычислительной мощности, производительность остается низкой, время на обработку и вычисление результатов какой либо задачи, по прежнему очень высоко [12]. А главное нельзя сказать, в какой момент времени нужно прекращать выполнение той или иной задачи, достигнув установившегося результата в вычислениях.

Основой всех проектов связанных с вычислениями является: точность, время обработки данных, и доступ к обработанным данным. В качестве примера можно привести задачу, решение которой основано на вычислении уравнения с использованием больших вычислительных ресурсов. Используя ресурс графических карт размещённых на узлах гибридного кластера, для решения сложной задачи, я смогу получить хороший прирост скорости вычисления данных, по отношению к использованию только CPU для решения той же задачи.

Рассматривая метод использования моментального снимка «snapshot» возникла идея, а можно ли, обойтись без выгрузки большого объёма данных на жёсткий диск. В следствии чего, было решено разработать и реализовать метод,

возможности которого позволили бы динамически влиять на данные задачи во время её выполнения.

Основной целью дипломного проекта, является разработка и реализация метода динамического доступа к графической памяти, который поможет оценить ход решения любой задачи связанной с расчётами.

Главными этапами работы являются:

1. разработка наиболее подходящего средства для обмена данными между задачами;
2. реализация и тестирование разработанного средства на гибридном кластере;
3. реализация программы на гибридном кластере, по расчёту уравнения гидродинамики, используя ресурс трёх графических карт (GPU);
4. проведение тестирования разработанного метода в совместном использовании с программой разработанной на этапе 3.

Работа посвящена разработке инструментария, который в свою очередь содержал функционал, позволяющий быстро обрабатывать и моделировать данные, с помощью разработанной и реализованной модели «подключения и сканирования видеопамяти».

Структура предполагаемого инструментария строилась на основе использования современных средств, как совмещение технологий Qt [11], CUDA [7], OpenMPI [3].

На основе совмещения технологий OpenMPI и Qt разработан и реализован графический интерфейс пользователя для управления расчётами течений разреженного газа, выполняемыми на гибридном кластере. Также на основе совмещения технологий CUDA и OpenMPI разработан и реализован основной модуль, предназначенный для вычислений, обработки и передачи данных между MPI процессами и несколькими GPU.

Программная разработка основана на сканировании графической памяти и передачи данных между MPI процессами двух различных задач, также представления результатов в графическом виде. Она позволит более

эффективно планировать и проводить специализированные вычислительные эксперименты в различных областях.

1 Описание технологий

В данной дипломной работе рассматривалось множество различных подходов к реализации, а именно, что использовать в качестве пересылки данных, с помощью какого средства производить вычисления, и с помощью какой технологии представлять в графическом виде полученные результаты.

Отталкиваясь от основного задания и разработки методов его решения, были рассмотрены разные технологии, языки программирования и также платформы.

В работе были применены следующие технологии:

- технология Open MPI [2];
- технология CUDA [9];
- технология Qt [11].

Подробнее о каждой технологии представлено ниже.

1.1 Описание технологии Open MPI

Коммуникационная библиотека MPI является стандартом в параллельном программировании с использованием механизма передачи сообщений.

MPI-программа представляет собой набор независимых процессов, каждый из которых выполняет одинаковую программу написанную на языке C или FORTRAN. Процессы MPI-программы взаимодействуют друг с другом посредством вызова коммуникационных процедур. Каждый процесс выполняется в своем собственном адресном пространстве [2].

Все функции распределения процессов по вычислительным узлам и для запуска на исполнение переносятся на операционную систему.

1.1.1 Краткое описание MPI

MPI - это библиотека функций, в которой взаимодействие параллельных процессов обеспечивается по средствам передачи сообщений [19].

Это обширная, и в тоже время сложная библиотека, состоящая из более чем 120 функций, в число которых входят:

- функции инициализации и закрытия MPI процессов;
- функции, реализующие коммуникационные операции типа точка-точка;
- функции, реализующие коллективные операции;
- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

Любая параллельная программа может быть написана с использованием 6 MPI функций, а достаточно полную среду составляет набор из 24 функций [21].

Каждая из MPI функций характеризуется способом выполнения:

- локальная функция - выполняется внутри вызывающего процесса. Ее завершение не требует коммуникаций;
- нелокальная функция - для ее завершения требуется выполнение MPI процедуры другим процессом;
- глобальная функция - процедуру должны выполнять все процессы группы. Несоблюдение этого условия может приводить к зависанию задачи;
- блокирующая функция - возврат управления из процедуры гарантирует возможность повторного использования параметров, участвующих в вызове. Никаких изменений в состоянии процесса, вызвавшего блокирующий запрос, до выхода из процедуры не может происходить;
- неблокирующая функция - возврат из процедуры происходит немедленно, без ожидания окончания операции и до того, как будет разрешено повторное использование параметров, участвующих в запросе. Завершение неблокирующих операций осуществляется специальными функциями.

Все процедуры языка «С» являются функциями, и большинство из них возвращают код ошибки [22].

Логические переменные представляются типом int (true соответствует 1, а false - 0).

Определение всех именованных констант, прототипов функций и определение типов выполняется подключением файла mpi.h. MPI допускает возможность запуска процессов параллельной программы на компьютерах различных платформ, обеспечивая при этом автоматическое преобразование данных при пересылках [3].

В таблице 1.1 - приведено соответствие предопределенных в MPI типов стандартным типам языка С.

Константа MPI	**Тип данных**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_LONG	unsigned long_int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	Байт
MPI_PACKED	Упакованные данные

1.1.2 Составляющая программы на MPI

- Программа состоит из N параллельных процессов;
- Процессы порождаются один раз при запуске программы;
- Каждый процесс имеет отдельное адресное пространство, общей памяти нет;
- Процессы взаимодействуют путем отправки и получения сообщений;
- Процессы могут образовывать группы.

Общая схема MPI программы:

```
#include<mpi.h>

 int main(int argc,char **argv)
{
int rank, size;
//Инициализация работы с MPI
MPI_Init(&argc, &argv);

//Завершение работы с MPI
MPI_Finalize();
return 0;
}
```

1.1.3 Сообщения в MPI

- Передача данных между процессами осуществляется путем отправки и приема сообщений [4];
- Требует совместная работа отправляющего и принимающего процесса [1];
- Изменение памяти принимающего процесса происходит при его явном участии.

Общая схема MPI сообщений приём \ передача:

```
if (rank == 0) {

// Программа процесса #0
int data_send[L];

      // Отправить массив data_send на процесс #7
     MPI_Send(data_send, L, MPI_INT, 7 , 1, MPI_COMM_WORLD);

 } else if (rank == 7) {

// Программа процесса #7

int data_resv[L];
MPI_Status status;
```

```
    // Принять массив длины L с любого процесса в data_resv
    MPI_Recv(data_resv, L, MPI_INT, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD, &status);
}
```

1.1.4 Способы отправки данных

- Стандартные прием/передача (MPI_Send). Блокирует работу. Возврат после окончания передачи. Абсолютно безопасная передача данных [14];
- Передача с буферизацией (MPI_Bsend). Возврат после записи сообщения в системный буфер. Не требует вызова функции приема сообщения;
- Передача с синхронизацией (MPI_Ssend). Возврат после начала приема сообщения процессом-получателем. Гарантирует безопасное использование буфера [14];
- Передача по готовности (MPI_Rsend). Процесс-получатель должен инициировать прием сообщения. Уменьшает накладные расходы на организацию передачи.[20].

1.2 Описание технологии QT

Qt — это кроссплатформенный инструментарий разработки приложений на C++ [11]. Инструментарий Qt, значительно расширяет стандартную библиотеку языка:

- программирования графического пользовательского интерфейса;
- сетевого программирования сокетов, работа с СУБД, HTTP, XML, JSON;
- работы с мультимедийными данными, такими как аудио и видео;
- программирования под мобильную платформу (сенсоры, позиционирование, Bluetooth, коммуникация);

- интернационализации приложений, поддержка Unicode и локализации;
- рефлексивного программирования, поддержки динамической типизации, получения информации о типах, создания объектов по имени класса и изменения их свойств.

Возможности Qt начинаются с ядра библиотеки, на основе которого построена большая часть остальных компонентов. Также рассматривается подход к созданию графического пользовательского интерфейса.

Qt содержит богатый набор инструментов, большинство их которых построено на основе возможностей, предоставляемых ядром библиотеки. Принципы, заложенные в ядро Qt, повсеместно используются во всех остальных компонентах каркаса, так что о них следует иметь представление независимо от того, стоит ли задача использовать имеющие компоненты Qt или создавать свои собственные [11].

При помощи Qt можно разрабатывать приложения с графическим интерфейсом, приложения, работающие с сетью, приложения, работающие с базами данных и мультимедийные приложения, работать с XML - структурами и 3D-графикой, осуществлять рисование и доступ к сетевым ресурсам. Поскольку поддерживается ряд платформ, Qt может работать на Linux, Mac OS, Windows.

1.2.1 Инструментальная поддержка

В составе каркаса Qt поставляется ряд утилит и программ, делающих разработку Qt-приложений простой.

1.2.1.1 Qmake

Утилита qmake упрощает сборку проектов на различных платформах. Данная утилита автоматически генерирует Makefile на основе .pro - файла, описывающего проект, так что далее проект может быть собран стандартной

утилитой make. Файлы проектов обычно устроены довольно просто, но могут содержать и более сложную информацию, если это необходимо.

Файл проекта обычно содержит перечисление заголовочных файлов, файлов реализации, файлов форм, ресурсов, модулей Qt, сторонних библиотек и конфигураций. Можно опционально указывать различные наборы для различных платформ. При использовании интегрированной среды разработки Qt Creator большая часть содержимого файла проекта генерируется автоматически.

1.2.1.2 Qt Creator

Qt Creator — это кроссплатформенная интегрированная среда разработки Qt - приложений, поддерживающая подсветку и проверку синтаксиса, автодополнение кода, управление проектами, автоматическую сборку проектов, запуск, отладку и профилирование, системы контроля версий и др.

В Qt Creator входит визуальный редактор пользовательских графических интерфейсов.

Основная задача Qt Creator — упростить разработку приложения с помощью фреймворка Qt на разных платформах. Поэтому среди возможностей, присущих любой среде разработки, есть и специфичные, такие как отладка приложений на QML и отображение в отладчике данных из контейнеров Qt, встроенный дизайнер интерфейсов как на QML, так и на QtWidgets.

1.2.1.3 Qt Designer

Qt Designer – инструмент предназначенный для проектирования и создания графических пользовательских интерфейсов (GUI) из компонентов Qt. Есть возможность создавать и настраивать свои виджеты или диалоги в режиме "что видишь, то и получишь" (what-you-see-is-what-you-get, WYSIWYG), и проверять их, используя разные стили и разрешающие способности.

С помощью Qt Designer возможна «сборка» интерфейса из готовых компонентов виджетов и менеджеров размещения, настойка их свойств, редактирование меню приложения, панелей инструментов, доступных пользователю действий, связей между определёнными сигналами и слотами.

Характерной особенностью Qt Designer является поддержка визуального редактирования сигналов и слотов. Так, например, можно связать сигнал, генерируемый по переключению состояния виджета CheckBox со слотом отвечающим за доступность другого виджета.

Qt Designer может быть запущен как отдельное приложение, так и во встроенном в IDE Qt Creator виде [11].

Разработанный интерфейс сохраняется в файл с расширением ui, который подключается к создаваемой программе с помощью специальных методов библиотеки Qt. Этот файл имеет xml-формат, и может, в случае необходимости, редактироваться в любом текстовом редакторе.

1.2.2 Виджеты

Qt Widgets — это традиционные элементы графического пользовательского интерфейса, которые обычно используются в окружениях рабочего стола. Qt подстраивает внешний вид виджетов под используемое окружение, так что приложения Qt выглядят одинаково под Windows, Linux и Mac OSX. Виджеты Qt подходят для статичного пользовательского интерфейса без сложной анимации, что является стандартной ситуацией, например, для приложений офисного типа.

Виджеты Qt могут отображать данные и состояние приложения, принимать обрабатывать пользовательские события и являться контейнерами для других виджетов, которые должны быть сгруппированы вместе. Виджеты, не вложенные в другие виджеты, называются окнами window. Виджеты Qt представляются классами, производными от QWidget. Сам QWidget является классом, производным от QObject, так что виджеты Qt прекрасно вписываются в объектную модель каркаса.

1.2.2.1 Виджеты и события

Класс QWidget предоставляет базовую реализацию функции event() для обработки событий приложения. Данная реализация вызывает предопределённые функции - обработчики для различных типов ситуаций:

- Обработчик paintEvent() вызывается всякий раз, когда виджет должен быть перерисован. При этом переданное событие позволяет получить информацию о том, в какой области экрана следует осуществить рисование;
- Обработчик resizeEvent() вызывается в случае изменения размера виджета. При этом доступна информация о старом и новом размере;
- Обработчики mousePressEvent() и mouseReleaseEvent() вызываются в качестве реакции на использование кнопок мыши в случае, когда курсор находится над виджетом. Первый обработчик будет вызван после нажатия кнопки, а второй — после её «отжатия». Так же предусмотрена обработка события «двойной клик» функцией mouseDoubleClickEvent();
- Для виджетов, которые предназначенны для обработки событий ввода с клавиатуры, вызывается обработчик keyPressEvent(). Перед началом пользовательского ввода вызывается обработчик focusInEvent(), а после focusOutEvent().

1.2.3 Схема QT для представления данных в графическом виде

Схема Qt, изображена на рисунке 1.1.

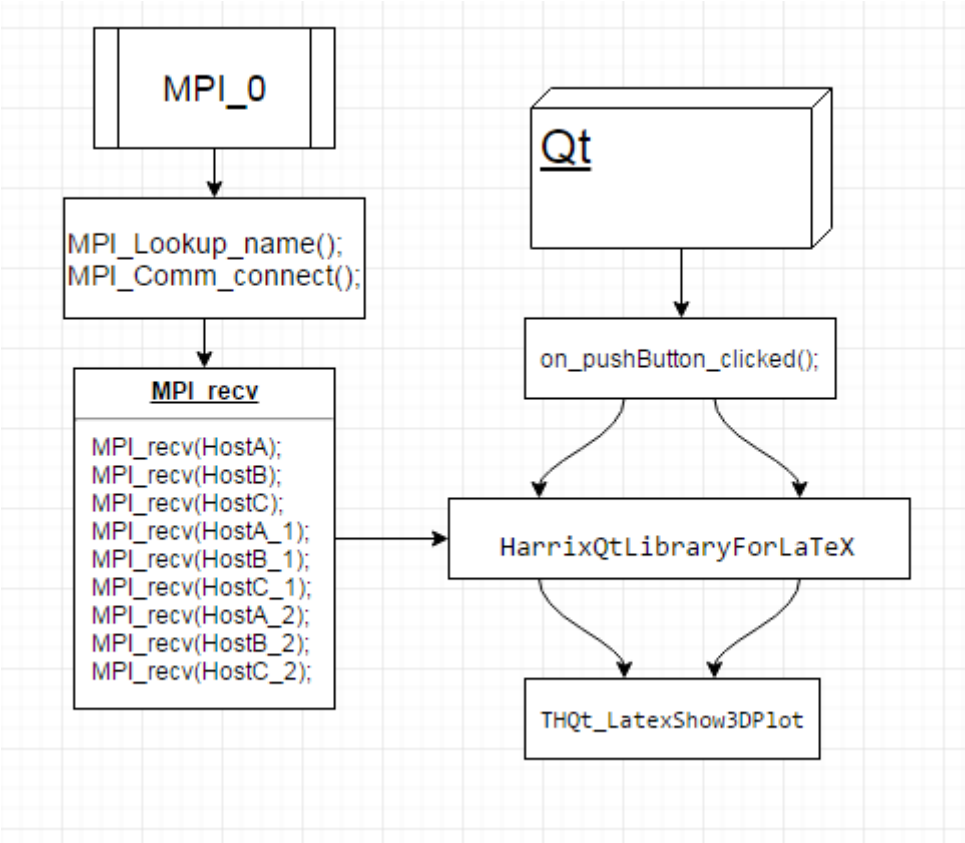

Рисунок 1.1. – Схема Qt

Вся часть программы, написанная на Qt, даже та часть, что отвечает за MPI, выполняется в теле функции void MainWindow::on_pushButton_clicked().

Обработка и построение 3D поверхности происходит непосредственно в теле условия if (my_rank == 0), после того как осуществится непосредственное подключение и приём данных с задачи сервера [18].

Для построения поверхности была использована библиотека «HarrixQtLibraryForLaTeX», за основу использования данной библиотекой был взят готовый проект и переделан путём небольших манипуляций под свой.

Список использованных функций библиотеки «HarrixQtLibraryForLaTeX» [15]:

- QString HQt_LatexBegin();

Возвращает начало для полноценного Latex файла для шаблона.

- QString HQt_LatexBeginArticle();

Возвращает начало для полноценного Latex файла в виде статьи для шаблона.

- QString HQt_LatexBeginArticleWithPgfplots();

Возвращает начало для полноценного Latex файла в виде статьи для шаблона.

- QString HQt_LatexBeginWithPgfplots();

Возвращает начало для полноценного Latex файла для шаблона.

- QString HQt_LatexEnd();

Возвращает концовку для полноценного Latex файла для шаблона.

- QString THQt_LatexShow3DPlot() [15].

Функция возвращает строку с выводом некоторого 3D графика в виде поверхности.

1.3 Описание технологии CUDA

CUDA (Compute Unified Device Architecture) — программно - аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia.

Основными плюсами CUDA являются ее простота программирования, которая ведется на "расширенном языке «C»" и гибкость, также она бесплатная.

1.3.1 Вычислительная модель GPU

С появление CUDA были полностью сняты все ограничения, предложив GPGPU простую и удобную модель. Вычислительная модель GPU рассматривается как специализированное вычислительное устройство, называемое device:[6].

- является сопроцессором к CPU, называемым host;
- обладает собственной памятью DRAM;

- обладает возможностью параллельного выполнения огромного количества отдельных нитей threads.

При этом между нитями на CPU и нитями на GPU есть принципиальные различия:

- нити на GPU обладают крайне «небольшой стоимостью» - их создание и управление требует минимальных ресурсов, в отличие от CPU;
- для эффективного удаления возможностей GPU нужно использовать многие тысячи отдельных нитей, а для CPU обычно нужно не более 10-20 нитей.

По сравнению с традиционным подходом к организации вычислений общего назначения посредством возможностей графических API, у архитектуры CUDA отмечают следующие преимущества в этой области [8]:

- интерфейс программирования приложений CUDA (CUDA API) основан на стандартном языке программирования Си с некоторыми ограничениями. По мнению разработчиков, это должно упростить и сгладить процесс изучения архитектуры CUDA [5];
- разделяемая между потоками память «shared memory» размером в 16 Кб может быть использована под организованный пользователем кэш с более широкой полосой пропускания, чем при выборке из обычных текстур;
- более эффективные транзакции между памятью центрального процессора и видеопамятью;
- полная аппаратная поддержка целочисленных и побитовых операций;
- поддержка компиляции GPU кода средствами открытого LLVM.

CUDA использует большое число отдельных нитей для вычислений, часто каждому вычисляемому элементами соответствует одна нить. Все нити группируются в иерархию – «grid/block/thread», данная схема изображена на рисунке 1.2.

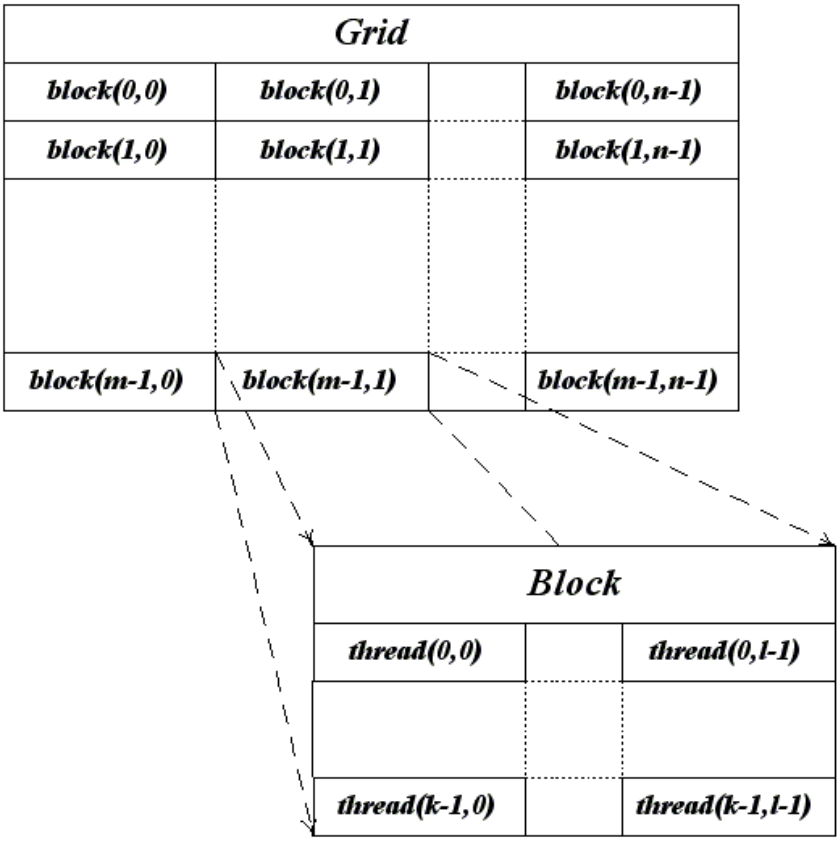

Рисунок 1.2 – Иерархия нитей в CUDA

Верхний уровень «grid» - соответствует ядру и объединяет все нити, выполняющие данное ядро. «grid» представляет собой одномерный или двухмерный массив блоков «block». Каждый блок «block» представляет из себя одномерный, двухмерный, трехмерный массив нитей «threads».[9].

При этом каждый блок представляет собой полностью независимый набор взаимодействующих между собой нитей, нити из разных блоков не могут между собой взаимодействовать.

Фактически блок соответствует независимо решаемой подзадаче, так например если нужно найти произведение двух матриц, то результирующую матрицу можно разбить на отдельные подматрицы одинакового размера. Нахождение каждой такой подматрицы может происходить абсолютно независимо от нахождения остальных подматриц. Нахождение такой

подматрицы - это задача отдельного блока, внутри блока каждому элементу подматрицы соответствует отдельная нить.

При этом нити внутри блока могут взаимодействовать между собой через:

- общую память «shared memory»;
- функцию синхронизации всех нитей блока «__synchronize».

Подобная иерархия довольно естественна. С одной стороны, хочется иметь возможность взаимодействия между отдельными нитями, а с другой, чем больше таких нитей, тем выше оказывается цена подобного взаимодействия.

Поэтому исходная задача, применения ядра к входным данным, разбивается на ряд подзадач, каждая из которых, решается абсолютно независимо, т.е. никакого взаимодействия между подзадачами нет, и в произвольном порядке.[7].

Сама же подзадача решается при помощи набора взаимодействующих между собой нитей.

С аппаратной части, все нити разбиваются на так называемые «warp» - блоки подряд идущих нитей, которые одновременно выполняются и могут взаимодействовать друг с другом. Каждый блок нитей разбивается на несколько «warp», размер «warp» для всех GPU равен 32.

Важным моментом является то, что нити фактически выполняют одну и ту же команды, но каждая со своими данными. Поэтому если внутри «warp» происходит ветвление, например: в результате выполнения оператора if, то все нити «warp» выполняют все возникающие при этом ветви. Поэтому крайне желательно уменьшить ветвление в пределах каждого отдельного «warp».

Также используется понятие «half-warp» - это первая или вторая половина «warp». Подобное разбиение «warp» на половины связано с тем, что обычно обращение к памяти делаются отдельно для каждого «half-warp».

Кроме иерархии нитей существует также несколько различных типов памяти. Быстродействие приложения очень сильно зависит от скорости работы с памятью. Именно поэтому в CPU большую часть кристалла занимают

различные кэши, предназначенные для ускорения работы с памятью, в то время как для GPU основную часть кристалла занимают ALU.

В CUDA для GPU существует несколько различных типов памяти, доступных нитям, сильно различающихся между собой, такая схема приведена в таблице 1.2.

Тип памяти	Уровень	Скорость работы
registers	per-thread	высокая (on chip)
local	per-thread	низкая (DRAM)
shared	per-block	высокая (on-chip)
global	per-grid	низкая(DRAM)
constant	per-grid	высокая(on chip L1 cache)
texture	per-grid	высокая(on chip L1 cache)

1.3.2 CUDA и язык C

Сама технология CUDA вводит ряд дополнительных расширений для языка C, которые необходимы для написания кода для GPU:

- спецификаторы функций, которые показывают, как и откуда буду выполняться функции;
- спецификаторы переменных, которые служат для указания типа используемой памяти GPU;
- спецификаторы запуска ядра GPU;
- встроенные переменные для идентификации нитей, блоков и других параметров при исполнении кода в ядре GPU;
- дополнительные типы переменных.

Спецификаторы функций определяют, как и откуда буду вызываться функции. Всего в CUDA три спецификатора:[10].

- __host__ — выполнятся на CPU, вызывается с CPU;
- __global__ — выполняется на GPU, вызывается с CPU;
- __device__ — выполняется на GPU, вызывается с GPU.

Спецификаторы запуска ядра служат для описания количества блоков, нитей и памяти, которые хотите выделить при расчете на GPU. Синтаксис запуска ядра имеет следующий вид:

KernelFunc <<< gridSize, blockSize, sharedMemSize, cudaStream >>>(float* param1, float* param2);

Где,

- gridSize – размерность сетки блоков (dim3), выделенную для расчетов;
- blockSize – размер блока (dim3), выделенного для расчетов;
- sharedMemSize – размер дополнительной памяти, выделяемой при запуске ядра;
- cudaStream – переменная cudaStream_t, задающая поток, в котором будет произведен вызов.

KernelFunc – функция ядра, спецификатор __global__. Некоторые переменные при вызове ядра можно опускать, например sharedMemSize и cudaStream.

Так же стоит упомянуть о встроенных переменных:

- gridDim – размерность грида, имеет тип dim3. Позволяет узнать размер грида, выделенного при текущем вызове ядра;
- blockDim – размерность блока, так же имеет тип dim3. Позволяет узнать размер блока, выделенного при текущем вызове ядра;
- blockIdx – индекс текущего блока в вычислении на GPU, имеет тип uint3;
- threadIdx – индекс текущей нити в вычислении на GPU, имеет тип uint3;
- warpSize – размер warp’a, имеет тип int.

Добавленные типы данных такие как:

В язык добавляются 1/2/3/4-мерные вектора из базовых типов - char1, char2, char3, char4, uchar1, uchar2, uchar3, uchar4, short1, short2, short3, sho rt4, ushort1, ushort2,ushort3, ushort4, int1, int2, int3, int4, uint1, uint2, uint3, uint4, l

ong1, long2, long3, long4, ulong1, ulong2, ulong3, ulong4, float1, float2, float3, float 2, и double2.

Обращение к компонентам вектора идет по именам - x, y, z и w. Для создания значений-векторов заданного типа служит конструкция вида make_<typeName>.

- int2 a = make_int2 (1, 7);
- float3 u = make_float3 (1, 2, 3.4f).

Также для задания размерности служит тип dim3, основанный на типе uint3, но обладающий нормальным конструктором, инициализирующим все не заданные компоненты единицами.

1.3.3 Директива вызова ядра

Для запуска ядра на GPU используется следующая конструкция:

- kernelName <<<Dg,Db,Ns,S>>> (args).

Здесь «kernelName» - это имя соответствующей __global__ функции, «Dg» - переменная типа dim3, задающая размерность и размер grid в блоках, «Db» - переменная типа dim3, задающая размерность и размер блока, в нитях, «Ns» - переменная типа size_t, задающая дополнительный объем shared-памяти, которая должна быть динамически выделена, к уже статически выделенной shared-памяти, «S» - переменная типа cudaStream_t, задает поток CUDA stream, в котором должен произойти вызов, по умолчанию используется поток 0. Через args обозначены аргументы вызова функции kernelName.

Также в язык C добавлена функция __syncthreads, осуществляющая синхронизацию всех нитей блока. Управление из нее будет возвращено только тогда, когда все нити данного блока вызовут эту функцию. Т.е. когда весь код, идущий перед этим вызовом, уже выполнен. Эта функция очень удобная для организации без конфликтной работы с shared-памятью.[9].

Также CUDA поддерживает все математические функции из стандартной библиотеки C, однако с точки зрения быстродействия лучше использовать их float-аналоги, а не double - например sinf. Кроме этого CUDA предоставляет

дополнительный набор математических функций __sinf, __powf и т.д. обеспечивающие более низкую точность, но заметно более высокое быстродействие, чем sinf, powf и т.п.

2 Описание гибридного кластера

2.1 Доступ к ресурсам ИВЦ

Доступ к вычислительному комплексу осуществляется с использованием протоколов SSH и SFTP (SCP):[13].

- протоколы SFTP и SCP позволяют передавать файлы между персональной рабочей станцией пользователя и сервером ИВЦ;
- протокол SSH позволяет работать на удалённом сервере, т.е. выполнять на нём команды.

К примеру, компилировать свои программы и затем ставить их в очередь на выполнение. Пользователям, работающим в ОС Microsoft Windows, предлагается использовать следующее открытое программное обеспечение: SSH-клиент PuTTY и SFTP-клиент WinSCP. Пользователи, работающие в Unix-системах, уже имеют в их составе утилиты 'scp' и 'ssh'.

Для авторизации пользователей используются только RSA/DSA-ключи, при этом открытая (public) часть ключа сообщается администрации ИВЦ в процессе регистрации и в дальнейшем пользователь авторизуется закрытой (private) частью ключа.

2.2 Аппаратное обеспечение

Имеющиеся сервера можно разделить на типы по следующим критериям:

1. используемые вычислительные элементы. Если кроме CPU для проведения вычислений используются сопроцессоры или GPU, архитектура таких серверов называется «гибридной» или «гетерогенной».[13].
2. самодостаточность. Для удешевления и более компактного расположения используются так называемые блейд-сервера (от английского «blade» - «лезвие»), содержащие только основные компоненты - материнскую плату, процессор, ОЗУ, …, но не

имеющие индивидуальной системы охлаждения или электропитания. Такие сервера не могут работать самостоятельно, они должны быть установлены в специальную серверную полку (также называемую «шасси» или «корзина»), обеспечивающую всем узлам централизованное отказоустойчивое электропитание, охлаждение, а иногда также управление и подключение к сетям передачи данных.

2.2.1 Сервера с GPU

12 блейд-серверов HP SL390s G7 гибридной архитектуры, каждый из которых содержит:

- Два 6-ядерных процессора Xeon X5670 с тактовой частотой 2933 МГц;
- 96 ГБ ОЗУ;
- Три карты NVIDIA Tesla M2090 на архитектуре Fermi (compute capability 2.0), у каждой из которых:
  - 1 GPU с 512 ядрами;
  - 6 ГБ памяти GDDR5 с пропускной способностью 177 ГБ/сек при выключенном контроле чётности (при включении ECC некая часть будет тратиться для обеспечения контроля);
  - 665 Гфлопс пиковой производительности для вычислений двойной точности. 1331 Гфлопс для одинарной.

2.3 Сети передачи данных

Коммуникационная сеть: Infiniband 4x QDR и DDR с пропускной способностью 40 и 20 Гбит/с соответственно и латентностью порядка 1-7 мкс, предназначена для доступа к сетевым системам хранения данных и для взаимодействия параллельных процессов, работающих на разных серверах кластера (например, для передачи сообщений MPI).

- Транспортная сеть: Gigabit Ethernet, используется для управления операционными системами серверов и работающими процессами.

- Сервисная сеть: Fast Ethernet, служит для доступа к интерфейсам администрирования, например, к HP Integrated Lights-Out или коммутационному оборудованию.

2.4 Система хранения данных

Для хранения данных используются:

1. сетевая система хранения данных с параллельной архитектурой Panasas ActiveStor 18. Полная ёмкость - 312 ТБ;
2. сетевая система хранения (NAS) HP IBRIX Storage Systems (X9000) на основе масштабируемой параллельной файловой системы IBRIX полной ёмкостью 186 ТБ;
3. NAS HP StorageWorks SFS на базе параллельной файловой системы Scalable File Share, основанной на технологии Lustre. Полная ёмкость системы составляет 24 ТБ;
4. HP Enterprise Virtual Array (EVA) 4100 на Fibre Channel дисках. Полная ёмкость - 4 ТБ.

Для обеспечения отказоустойчивости используются дисковые массивы RAID уровней 1, 5 и 6.

2.5 Производительность

Производительность серверов разных типов:

- пиковая производительность серверов HP BL2x220c G6 и G7 - 21,2 Тфлопс. Полученная на тесте Linpack - 17,3 Тфлопс, что составляет 81.6% от пиковой;
- пиковая производительность NVIDIA Tesla M2090 серверов HP SL390s G7 - 23,9 Тфлопс. Полученная на тесте Linpack - 11,9 Тфлопс.

2.6 Планировщик задач Altair PBS Pro

Программы, выполняемые на комплексе, не запускаются пользователями сразу на выполнение - вместо этого они помещаются в очередь ожидающих задач. Для этого используется команда qsub из пакета PBS, которой в качестве параметра передается скрипт на языке bash, содержащий всю необходимую информацию - какую программу надо запустить и какие ресурсы ей потребуются (количество процессов, оперативной памяти, максимальное время работы, …).[13].

Проверив скрипт и поместив его в очередь, qsub выводит на экран строку вида «XXXXX.hpc-suvir.hpc», где XXXXX - некое число, являющееся уникальным идентификатором задачи.

Посмотреть статус задачи в очереди можно при помощи команды qstat, в качестве параметра которой опционально можно указать идентификатор интересующей задачи. Удалить свою задачу из очереди можно командой qdel с идентификатором в качестве параметра.

Работающий на сервере планировщик задач контролирует текущее состояние комплекса и при наличии необходимых свободных ресурсов запускает очередной скрипт из очереди. Скрипт запускается только на одном из узлов (называемом «Mother Superior»), задача распараллеливания возлагается на пользователя и его программу. (В свою очередь, использование в программе стандартов MPI или OpenMP позволяет данную задачу облегчить). Названия узлов, выделенных задаче, и ряд других дополнительных параметров передаются скрипту через переменные окружения.

После завершения скрипта его стандартные потоки вывода и ошибок (stdout и stderr) сохраняются в той же директории в файлы с названиями вида «Название скрипта.oXXXXX» и «Название скрипта.eXXXXX» соответственно. Подробнее описано на этой странице. Следует отметить, что при одновременном выводе (например, в stdout) несколькими потоками порядок

сообщений в файле будет неопределённым, никакого упорядочивания не производится.

С момента постановки задачи в очередь и до её завершения нельзя изменять файлы, относящиеся к данной задаче, ни скрипт для qsub, ни запускаемое приложение, ни исходные данные. Именно эти самые файлы, а не их копии, будут использоваться при работе задачи, поэтому их модификация с большой вероятностью прервёт работу либо приведёт к некорректным результатам.

2.6.1 Запрос ресурсов

Ресурсы могут относиться к серверу PBS, к очередям или к узлам (виртуальным узлам). Задача запрашивает необходимые ей ресурсы. Ресурсы выделяются, при этом некоторые из них (например, процессорные ядра или оперативная память узла) при выполнении задачи потребляются и освобождаются только после её завершения. Виртуальные узлы, находящиеся на одном сервере, могут разделять свои ресурсы друг с другом.[13].

Ресурсы, относящиеся к серверу PBS или к очереди, используются всей задачей и в запросе указываются в следующем виде: «-l Ресурс=Значение». Пример такого ресурса – «walltime», время, необходимое задаче для работы. Если по истечении этого времени задача не завершится сама, то будет прервана принудительно. Указывается в виде «часы:минуты:секунды». Например, следующий запрос означает, что задаче достаточно полторы минуты работы:

- -l walltime=00:01:30.

Ресурсы, относящиеся к узлам (или виртуальным узлам), запрашиваются и выделяются в виде блоков (chunk). Запрос в большинстве случаев можно представить в следующем виде: «-l select=N:chunk», где строка «chunk» имеет вид «Ресурс1=Значение1[:Ресурс2=Значение2 …]», а число «N» - сколько таких блоков требуется. Если нужны блоки разных типов, можно использовать следующую форму запроса: «-l select=N1:chunk1[+N2:chunk2 …]».

В частности, именно так запрашивается необходимое количество ядер (параметр «ncpus», хотя логичнее было бы «ncores») и оперативной памяти («mem»). При использовании MPI или OpenMP также надо указать, сколько потоков должно быть запущено в этом блоке, для этого используются параметры «mpiprocs» и «ompthreads» соответственно. Например, следующий запрос означает, что задаче необходимы два блока, каждый из которых содержит 4 ядра и 8000 МБ памяти, и на каждом из этих блоков будут запущены 4 MPI процесса:

- -l select=2:ncpus=4:mpiprocs=4:mem=8000m.

Для указания нужного расположения выделяемых блоков на узлах используется запрос «-l place=[arrangement][:sharing]».

1. «Arrangement» имеет смысл только при запросе более одного блока. Может иметь одно из значений:
   - «pack» - все выделенные блоки должны быть на одном узле;
   - «scatter» - все блоки должны быть на разных серверах;
   - «free» - произвольным образом.
2. «Sharing» может иметь одно из значений:
   - «exclhost»- только эта задача может использовать выделенный узел. Неиспользуемые на узле ресурсы будут недоступны другим задачам;
   - «excl» - только эта задача может использовать выделенный виртуальный узел;
   - «shared» - значение по умолчанию. Задача может разделять ресурсы узла (или виртуального узла) с другими задачами.

2.6.2 Пример задачи

На рисунке 2.1представлен скрипт «main.sh».

```
#!/bin/bash

#PBS -l select=2:ncpus=1:mem=2000m,place=scatter
#PBS -l walltime=00:01:00
#PBS -m n

cd $PBS_O_WORKDIR
echo "I run on node: `uname -n`"
echo "My working directory is: $PBS_O_WORKDIR"
echo "Assigned to me nodes are:"
cat $PBS_NODEFILE
sleep 15
```

Рисунок 2.1 – скрипт «main.sh»

Поставим в очередь скрипт приведённый на рисунке 2.1. командой

- qsub main.sh

Спустя некоторое время в текущей директории появляются файлы «main.e12312» и «main.o12312». Первый из них содержит сообщения об ошибках и в случае корректной работы, как правило, должен быть пустым. Второй файл содержит вывод скрипта, он представлен на рисунке 2.2.

```
I run on node: cn15
My working directory is: /mnt/storage/home/admin/examples/pbs
Assigned to me nodes are:
cn15
cn16
```

Рисунок 2.2 – скрипт «main.o12312»

Для того чтобы просмотреть состояние выполняющейся задачи нужно выполнить команду:

- qstat 12312.

Где 12312 - идентификатор получившейся задачи.

На рисунке 2.3. приведено состояние выполняющейся задачи.

```
Job id            Name             User              Time Use S Queue
----------------  ---------------- ----------------  -------- - -----
12312.hpc-suvir1  main.sh          admin                    0 Q workq
```

Рисунок 2.3 – состояние задачи «main.sh»

2.6.3 Интерактивный режим

В случае, если необходимо интерактивное взаимодействие с работающей программой, используется соответствующий режим запуска задач. В этом режиме после постановки задачи в очередь команда qsub не завершается, а ждёт, когда планировщик выделит запрошенные ресурсы, после чего предоставляет shell на выделенном узле. Если запрошено несколько узлов, доступ предоставляется к «Mother Superior».

Вторая важная особенность интерактивного режима заключается в том, что из скрипта, передаваемого qsub, используются только опции PBS (строки «#PBS …»), содержимое скрипта не выполняется. Поэтому после получения shell необходимую программу надо запустить самостоятельно, после чего с ней можно работать в интерактивном режиме.

Для запроса у планировщика интерактивного режима надо при вызове qsub дополнительно указать параметр «-I»:

- qsub -I submit.sh

Либо можно указать этот параметр внутри скрипта submit.sh:

- #PBS -I

Может быть удобным отказаться от использования скрипта и все параметры передавать qsub через коммандную строку:

- qsub -I -l select=1:ncpus=1:mem=2000m,walltime=1:0:0

В случае, если будет использоваться система X Window для запуска на узлах программ с графическим интерфейсом, необходимо добавить ключ «-X»:

- qsub -I -X -l select=1:ncpus=1:mem=2000m,walltime=1:0:0

Если программа отработала и завершилась до истечения walltime, то для завершения самой интерактивной задачи выполните команду «exit».

3 Описание проекта

В момент разработки и реализации инструментария, рассмотрены подходы «методы», связанные с передачей, пересылкой данных между задачами, такие как:

- использование именованных каналов;
- использование сокетов;
- использование Open MPI.

В результате изучения и написания, и применения каждого из методов было приято решение, остановиться на последнем «с использованием Open MPI» [23].

Главная суть использования «Open MPI» - это применение реализованного метода, который устанавливает соединение между двумя группами процессов MPI, не разделяющими единый коммуникатор [23].

Разработка включает в себя следующие возможности:

- Взаимодействие двух частей приложения, запущенных независимо друг от друга;
- Сканирование памяти клиентом с сервера, установив соединение между задачами;
- Сервер может принимать соединения от нескольких клиентов. И клиенты, и сервер могут быть параллельными программами.

В процессе разработки программного пакета пользователя, задача основанная на реализации и проектировании, разбита на три стадии.

3.1 Стадии проекта

В программном плане реализованная «модель» на первой стадии выглядит следующем образом, она приведена на рисунке 3.1.

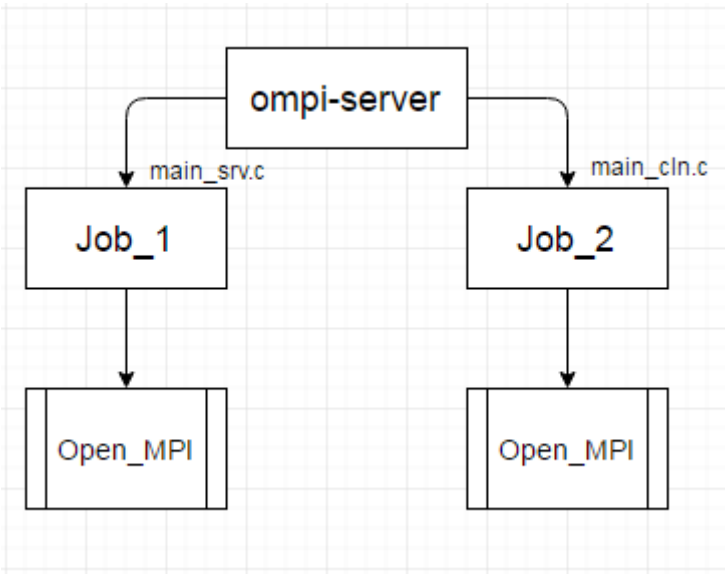

Рисунок 3.1 – программная модель на первой стадии

Данная разработка содержит клиентскую и серверную часть, каждая из которых реализована только на «Open MPI».

Модель установления соединения между двумя MPI задачами представлена на рисунке 3.2.

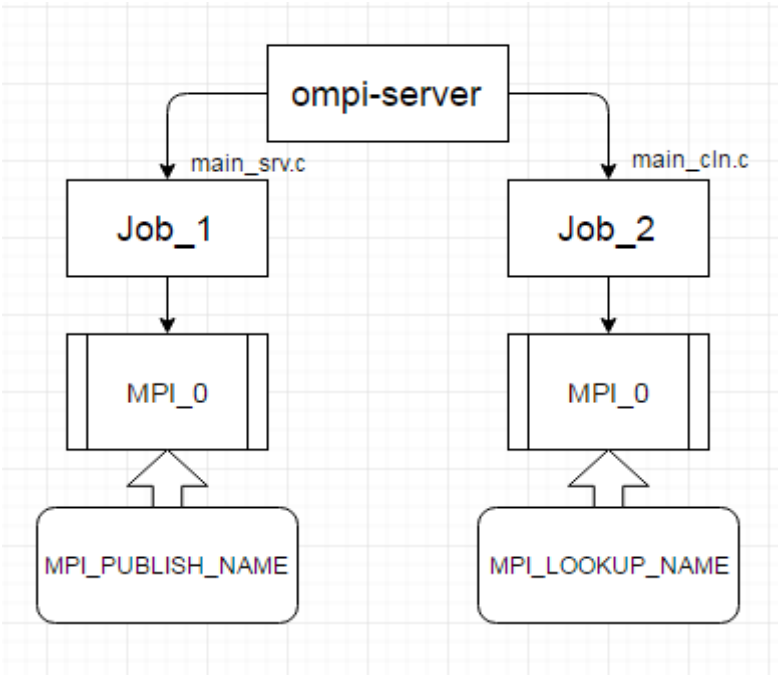

Рисунок 3.2 – соединение между двумя MPI задачами

Описание разработки на стороне сервера.

Серверу доступны следующие процедуры. Первой процедурой, он вызывает MPI_OPEN_PORT для установки порта, по которому к нему можно получить доступ. Второй процедурой, он должен вызывать MPI_COMM_ACCEPT для приема соединения.

Рассмотрим подробнее каждую из процедур.

- MPI_OPEN_PORT (info, port_name);

Таблица 3. 1 – параметры функции MPI_OPEN_PORT

info	Информация, специфичная для реализации об установке адреса (дескриптор)
port_name	Новый установленный порт (строка)

Функция устанавливает сетевой адрес, кодируя его в строку port_name, по которому сервер в состоянии принимать соединения от клиентов. port_name поддерживается системой, используя информацию аргумента info.

MPI копирует имя порта, поддерживаемое системой, в port_name. Аргумент port_name определяет вновь открываемый порт и может использоваться клиентом для контакта с сервером. С помощью MPI_MAX_PORT_NAME определяется максимальный размер строки, которая может поддерживаться системой.

- MPI_COMM_ACCEPT (port_name, info, root, comm, newcomm).

Таблица 3.2 – параметры функции MPI_COMM_ACCEPT

port_name	Имя порта (строка, используется только root)
info	Информация, зависящая от реализации (дескриптор, используется только root)
root	Ранг в comm для узла root (целое)
comm	Интракоммуникатор, внутри которого выполняется коллективный вызов (дескриптор)
newcomm	Интеркоммуникатор с клиентом в качестве удаленной группы (дескриптор)

MPI_COMM_ACCEPT устанавливает соединение с клиентом. Это коллективная операция посредством вызывающего коммуникатора. Она возвращает интеркоммуникатор, позволяющий установить связь с клиентом.

Аргумент info является строкой, определяемой реализацией, позволяющей точный контроль над вызовом MPI_COMM_ACCEPT.

MPI_COMM_ACCEPT является блокирующим вызовом. Можно реализовать не блокирующий доступ, поместив MPI_COMM_ACCEPT в отдельный поток.

Описание разработки на стороне клиента.

Клиенту доступна следующая процедура – это MPI_COMM_CONNECT, служащая для установки соединения с сервером.

- MPI_COMM_CONNECT (port_name, info, root, comm, newcomm);

Таблица 3.3 – параметры функции MPI_COMM_CONNECT

port_name	Сетевой адрес (строка, используется только root)
info	Информация, зависящая от реализации (дескриптор, используется только root)
root	Ранг в comm для узла root (целое)
comm	Интракоммуникатор, внутри которого выполняется коллективный вызов (дескриптор)
newcomm	Интеркоммуникатор с сервером в качестве удаленной группы (дескриптор)

Эта процедура устанавливает соединение с сервером, указанным port_name. Она коллективна для вызывающего коммуникатора и возвращает интеркоммуникатор, в котором удаленная группа участвует в MPI_COMM_ACCEPT.

Если названный порт не существует (или был закрыт), MPI_COMM_CONNECT возвращает ошибку класса MPI_ERR_PORT.

Если порт существует, но не имеет незавершенных MPI_COMM_ACCEPT, соединение пытается в конечном итоге выждать таймаут, определяемый реализацией, или завершиться, если сервер вызвал MPI_COMM_ACCEPT. В случае наступления таймаута MPI_COMM_CONNECT возвращает ошибку класса MPI_ERR_PORT.

Описание процесса взаимодействия двух MPI процессов независимых задач.

Процедуры предоставляют механизм опубликования имен. Пара (service_name, port_name) публикуется сервером и могут восстанавливаться клиентом при использовании только service_name. Реализация MPI определяет границы service_name, то есть область, в которой service_name может быть восстановлен. Если область представляет собой пустое множество (т.е., ни один клиент не может восстановить информацию), то опубликование имен не поддерживается. В реализации должны описываться то, как определяются эти границы. Высококачественные реализации могут дать определенный контроль над функциями опубликования имен через аргумент info.

На стороне сервера для взаимодействия MPI процесса с MPI процессом клиента используется функция MPI_PUBLISH_NAME, и для решения проблемы с одинаковыми именами используется функция MPI_UNPUBLISH_NAME.

Немного подробнее об MPI_PUBLISH_NAME и MPI_UNPUBLISH_NAME.

- MPI_PUBLISH_NAME (service_name, info, port_name);

Таблица 3.4 - параметры функции MPI_PUBLISH_NAME

service_name	Имя сервиса для ассоциации с портом (строка)
info	Информация, зависящая от реализации (дескриптор)
port_name	Имя порта (строка)

Процедура публикует пару (port_name, service_name), чтобы приложение могло восстановить поддерживаемое системой port_name, используя service_name.[17].

Реализация определяет границы опубликованного имени сервиса, то есть область, в которой имя сервиса уникально и, с другой стороны, область, в которой может быть восстановлена пара (port_name, service_name). В частности, имя сервиса может быть уникально для задачи (когда задача определяется распределенной операционной системой или пакетным планировщиком), уникально для машины или уникально в области Kerberos. Граница может зависеть от аргумента info для MPI_PUBLISH_NAME.

MPI запрещает опубликование более чем одного service_name для одного port_name. С другой стороны, если service_name уже опубликовано внутри границ, определенных info, поведение MPI_PUBLISH_NAME не известно. Реализация MPI может через механизм аргумента info для MPI_PUBLISH_NAME предоставить способ, разрешающий существование нескольких серверов с одним и тем же сервисом в одних и тех же границах. В этом случае, политика, определяемая реализацией, будет определять, какое из нескольких имен портов будет возвращаться MPI_LOOKUP_NAME.

Хотя service_name имеет ограниченное пространство действия, определяемое реализацией, port_name всегда имеет глобальные границы внутри коммуникационного пространства, используемого реализацией (т.е., оно является уникальным глобально).[17].

Аргумент port_name должен быть именем порта, установленного MPI_OPEN_PORT и еще не закрытого MPI_CLOSE_PORT.

- MPI_UNPUBLISH_NAME (service_name, info, port_name);

Таблица 3.5 - параметры функции MPI_UNPUBLISH_NAME

service_name	Имя сервиса (строка)
info	Информация, зависящая от реализации (дескриптор)
port_name	Имя порта (строка)

Эта процедура отменяет опубликование имени сервиса, которое было ранее опубликовано. Попытка отмены опубликования имени, которое еще не было опубликовано или уже было отменено, вызывает ошибку класса MPI_ERR_SERVICE.

Все опубликованные имена должны отменяться, прежде чем будет закрыт соответствующий порт и завершен процесс публикации. Поведение MPI_UNPUBLISH_NAME зависит от реализации в случае, если процесс пытается отменить опубликование имени, которое не было опубликовано.

На стороне клиента для взаимодействия MPI процесса с MPI процессом сервера используется функция MPI_LOOKUP_NAME.

- MPI_LOOKUP_NAME (service_name, info, port_name);

Таблица 3.6 - параметры функции MPI_LOOKUP_NAME

service_name	Имя сервиса (строка)
info	Информация, зависящая от реализации (дескриптор)
port_name	Имя порта (строка)

Функция восстанавливает port_name, опубликованное через MPI_PUBLISH_NAME, по service_name. Если service_name еще не было опубликовано, оно вызывает ошибку класса MPI_ERR_NAME. Приложение должно иметь для port_name буфер достаточной величины, чтобы хранить максимально возможное имя.

Если реализация позволяет несколько вхождений одного и того же service_name с теми же границами действия, определенное port_name выбирается способом, определяемым реализацией.

Если аргумент info был использован вместе с MPI_PUBLISH_NAME, чтобы сообщить реализации, как опубликовывать имена, такой же аргумент info может потребоваться MPI_LOOKUP_NAME.

В программном плане реализованная «модель» на второй стадии выглядит также как и в первом случае, она изображена на рисунке 3.3.

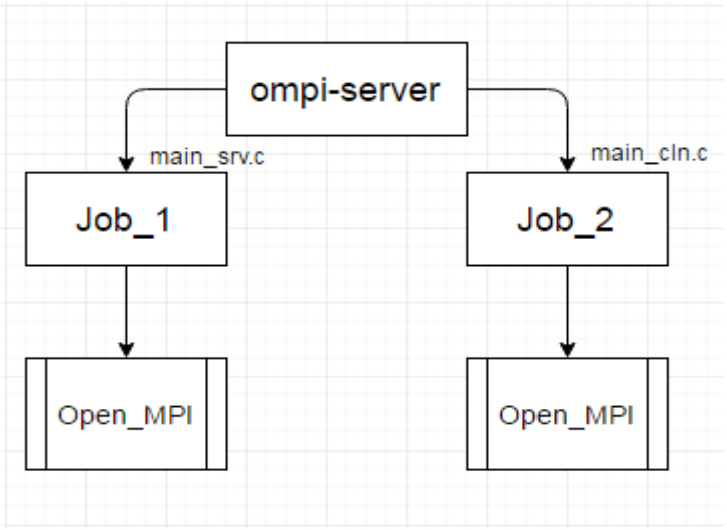

Рисунок 3.3 – Программная модель на второй стадии

Данная разработка содержит клиентскую и серверную часть, каждая из которых реализована на «Open MPI», с обработкой множества MPI процессов

на каждой задаче, и с запросом на передачу данных клиентом, от сервера клиенту.

Модель передачи данных между двумя MPI задачами, где на стороне сервера реализована часть по обработке данных с использованием множества MPI процессов, также реализован приём и передача данных между задачами, а на стороне клиента реализована часть по приёму и запросу данных от сервера.

Такая модель представлена на рисунке 3.4.

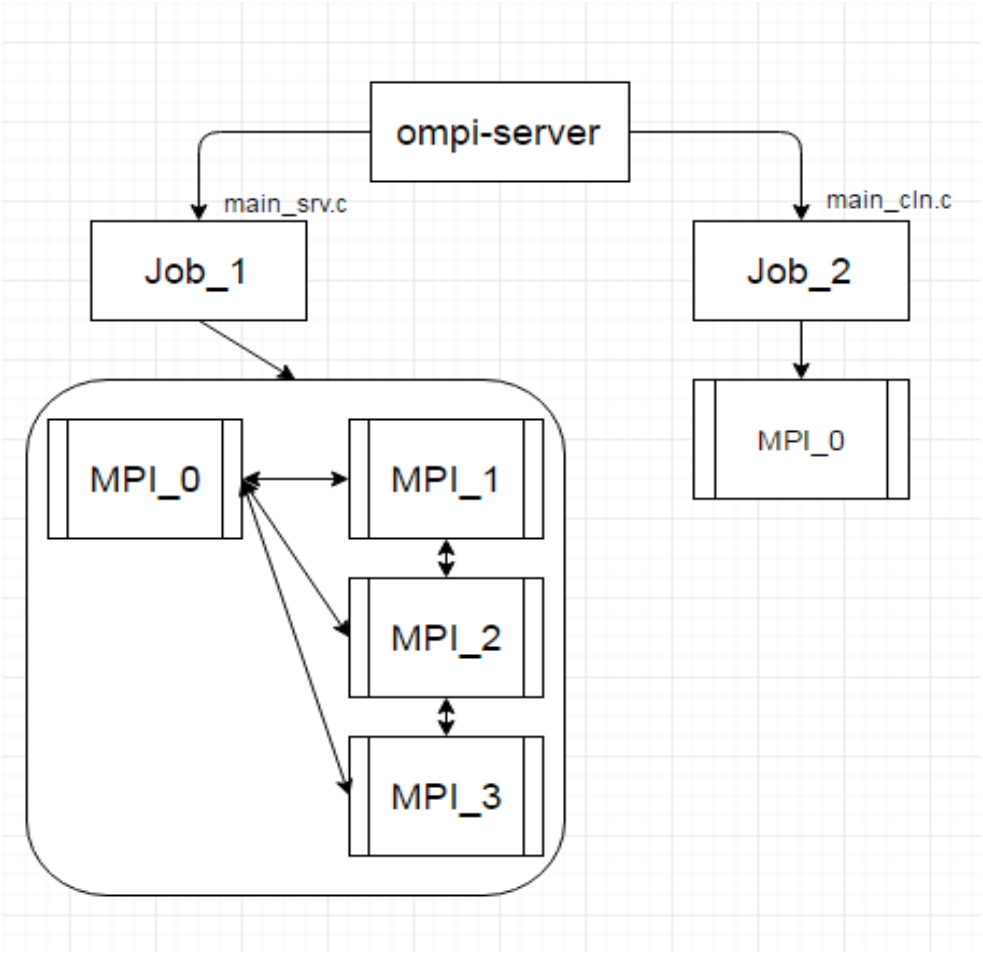

Рисунок 3.4 – Обмен данными между двумя MPI задачами

Обмен данными между клиентом и сервером осуществляется двумя функциями, первая MPI_SEND - функция передачи данных, вторая MPI_RECV – функция приёма данных.

- MPI_ SEND (buf, count, MPI_Datatype, dest, tag, comm);

Таблица 3.7 - параметры функции MPI_ SEND

buf	Адрес начала расположения пересылаемых данных;
count	Число пересылаемых элементов;
datatype	Тип посылаемых элементов;
dest	Номер процесса-получателя в группе, связанной с коммуникатором comm;
tag	Идентификатор сообщения (аналог типа сообщения функций nread и nwrite PSE nCUBE2);
comm	Коммуникатор области связи.

Функция выполняет посылку count элементов типа datatype сообщения с идентификатором tag процессу dest в области связи коммуникатора comm.

Переменная buf - это, как правило, массив или скалярная переменная.

- MPI_RECV (buf, count, datatype, source, tag, comm, status);

Таблица 3.8 - параметры функции MPI_RECV

buf	Адрес начала расположения принимаемого сообщения;
count	Максимальное число принимаемых элементов;
datatype	Тип элементов принимаемого сообщения;
source	Номер процесса отправителя;
tag	Идентификатор сообщения;
comm	Коммуникатор области связи;
status	Атрибуты принятого сообщения.

Функция выполняет прием count элементов типа datatype сообщения с идентификатором tag от процесса source в области связи коммуникатора comm.

Для использования множества MPI процессов в одной задаче используются следующие функции:

- MPI_COMM_SIZE - Функция определения числа процессов в области связи;
    - MPI_COMM_SIZE (MPI_Comm comm, int *size);

Таблица 3.9 - параметры функции MPI_COMM_SIZE

comm	Коммуникатор;
size	Число процессов в области связи коммуникатора comm.

- MPI_COMM_RANK - Функция определения номера процесса;

- MPI_COMM_RANK (MPI_Comm comm, int *rank);

Таблица 3.10 - параметры функции MPI_COMM_RANK

comm	Коммуникатор;
size	Номер процесса, вызвавщего функцию.

В программном плане реализованная «модель» на третьей стадии выглядит следующем образом, она представлена на рисунке 3.5.

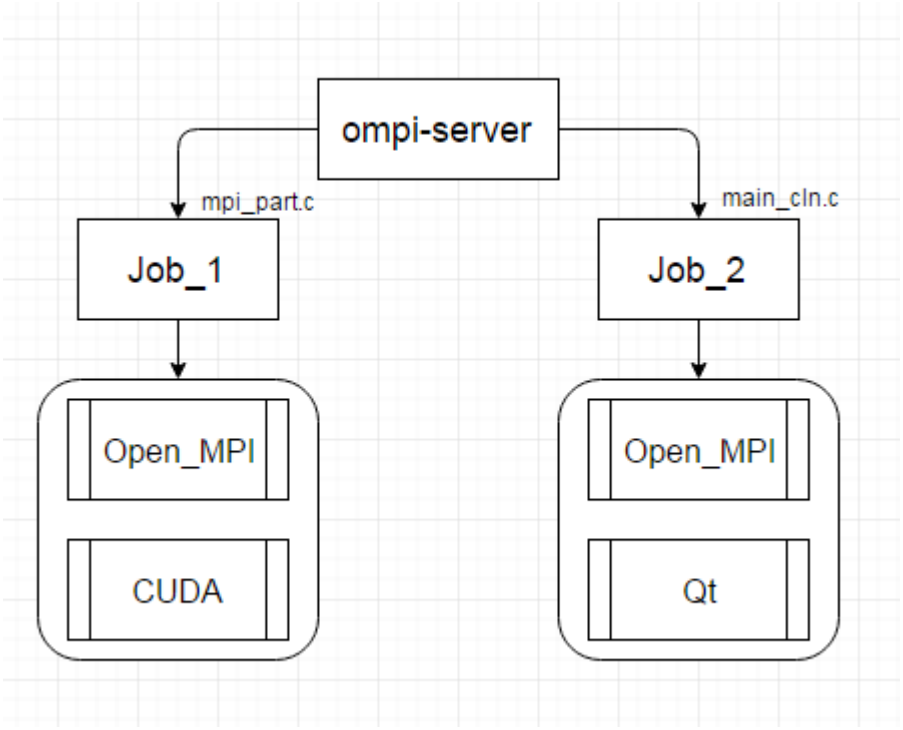

Рисунок 3.5 – Программная модель на третьей стадии

Данная разработка содержит клиентскую и серверную часть. Клиентская часть реализована на основе совмещения технологий OpenMPI и Qt . Серверная часть реализована на основе совмещения технологий CUDA и OpenMPI.[21].

Финальная модель, установления соединения между двумя MPI задачами, также осуществляется сканирование графической памяти сервера, клиентом. Такая модель представлена на рисунке 3.6.

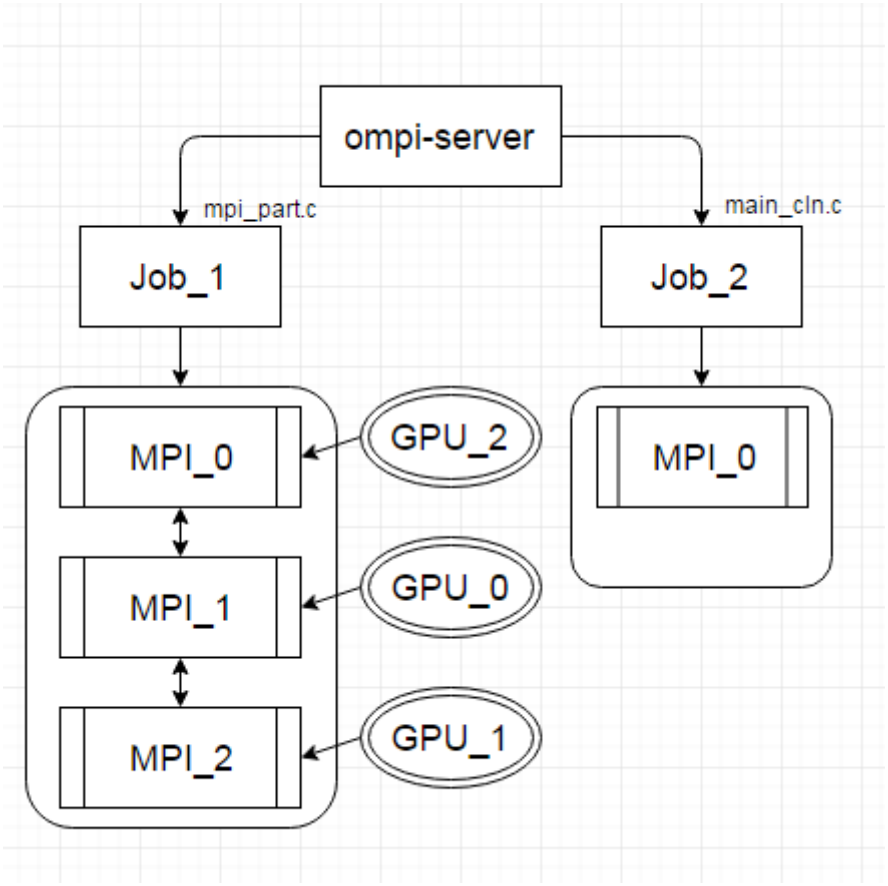

Рисунок.3.6 – Сканирование графической памяти

Задача сервера «mpi_part.c», с использованием CUDA и MPI делится на две, это нужно для того чтобы использовать два компилятора nvcc для кода CUDA и mpicc для MPI. Первая та, что «cuda_part.cu» непосредственно реализует вычислительное ядро на GPU, использует реализацию CUDA под Nvidia. Вторая, та что основная «mpi_part.c», использует основную реализацию MPI, и также добавлена реализация та, что отвечает за каждый MPI процесс, чтобы тот в свою очередь знал, с каким GPU на сервере он должен работать.

Описание работы на стороне сервера:

- «cuda_part.cu» - содержит;

Спецификатор __global__ — выполняется на GPU, вызывается с CPU, реализует функцию AddVectors которая вычисляет заданную формулу. В функции AddVectors используется встроенная переменная threadIdx – это индекс текущей нити в вычислении на GPU, Значения для вычислений

задаются как вектора. «cuda_part.cu» - вместо стандартной функции «main» использует функцию с другим именем и параметрами, описанная как «extern».

- «mpi_part.c» - содержит;

На рисунке 3.7 представлена наглядная схема обслуживания каждым MPI процессом своего GPU. Т.е. у нас есть некий узел Host sl008 с которым мы работаем, далее на данном узле мы формируем три виртуальных узла, на каждом из которых находится по одной GPU.

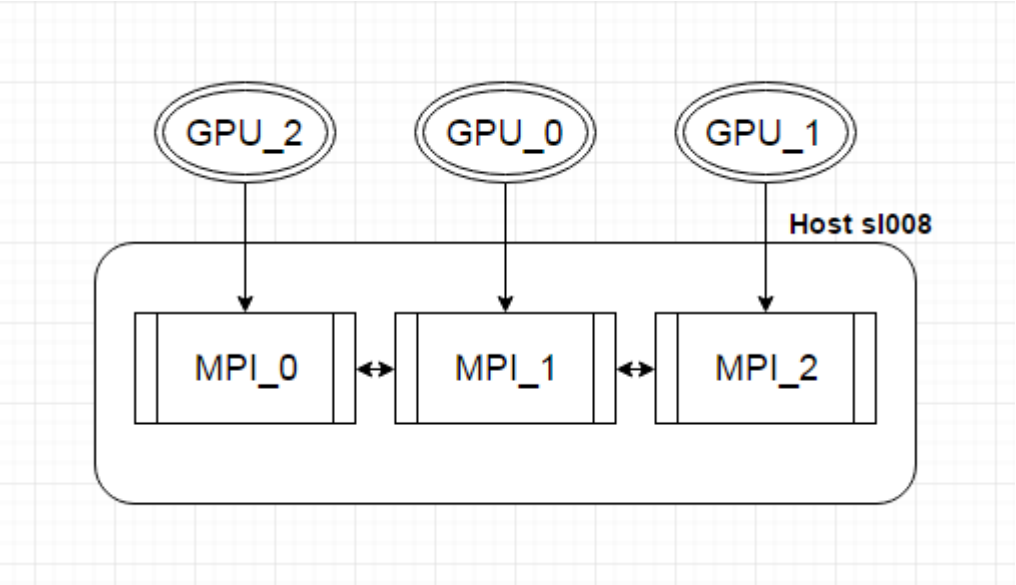

Рисунок 3.7 – Схема обслуживания

Описание работы на стороне клиента:

- «main_cln.c» - содержит;

Функционал совмещения технологии «OpenMPI» и «QT». Содержание «Qt» - это основная реализация интерфейсного окна предназначенное для пользователя, чтобы тот в совою очередь мог наглядно увидеть в графическом представлении полученные данные от сервера. Интерфейсное окно т.е. «Widget», содержит 2 кнопки «Button» и поле «Widget». Первая кнопка «Button1» она же «Загрузить» отвечает за вывод данных в поле «Widget», вторая кнопка «Button2», она же «Очистить» отвечает за очистку поля «Widget». Поле «Widget» - отвечает за саму зарисовку графика. Для создания

графиков в бесплатной версии «Qt» нету инструмента чтобы без проблем нарисовать любой график, но воспользовавшись бесплатной библиотекой «HarrixQtLibraryForLaTeX» - проблема была решена, и на реализацию программной части самого графика ушло минимум усилий и времени.

3.2 Структура проекта

Этапы работы:

1. отработка модуля CUDA;

Для вычисления результатов задачи гидродинамики используется три GPU. Программный модуль приведён в файле «cuda_part.cu», а структура программной реализации наглядно предложена на рисунке 3.8.

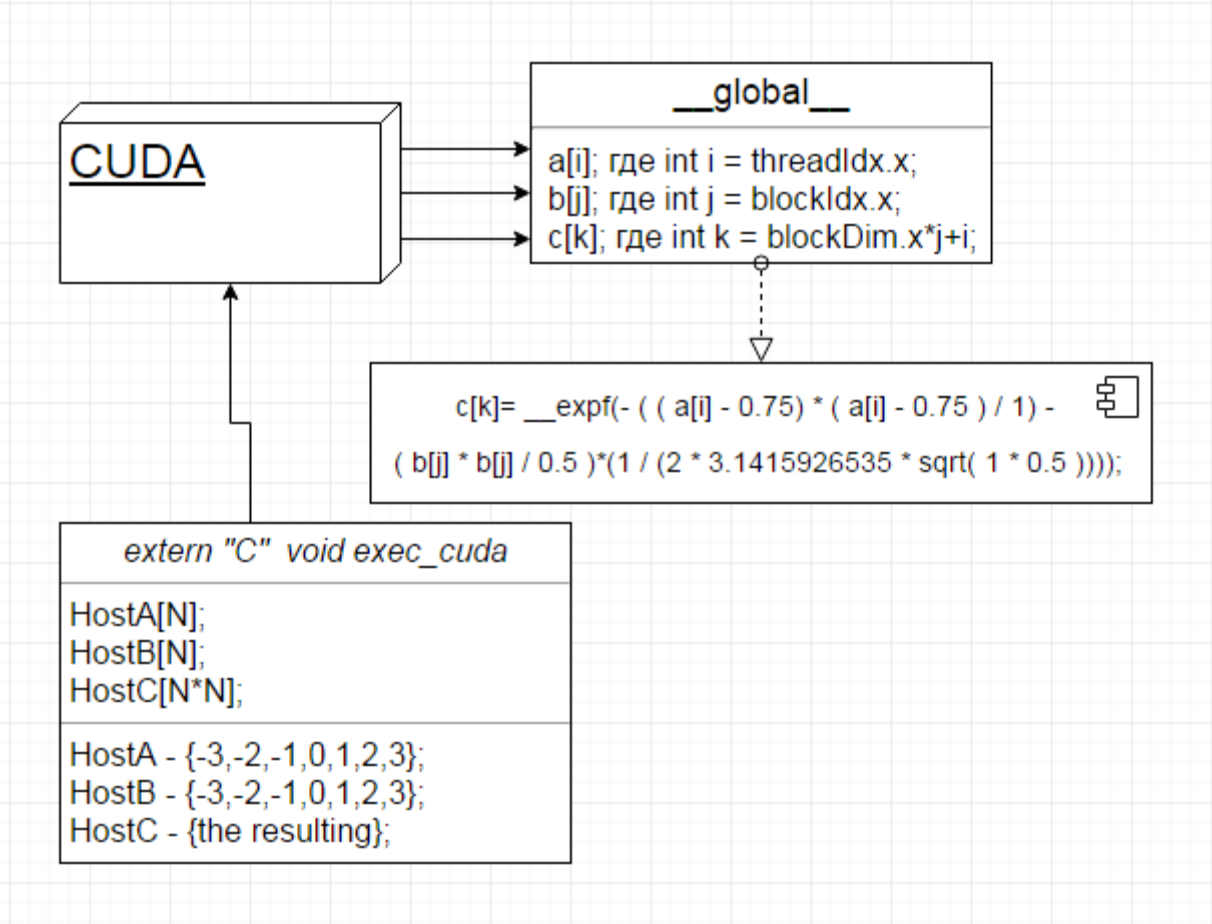

Рисунок 3.8 – Модуль CUDA

2. Программа MPI на стороне задачи сервера;

Сервер служащий для распараллеливания, обработки и пересылки результатов представлен в программе «mpi_part.c», в итоге получается работа, как совмещение двух технологий CUDA + MPI, что в конечном итоге представляет решение задачи гидродинамики и доступ к подсчитанным данным из любой другой задачи в любой момент времени.

На рисунке 3.9 приведена схема работы программы MPI на стороне сервера.

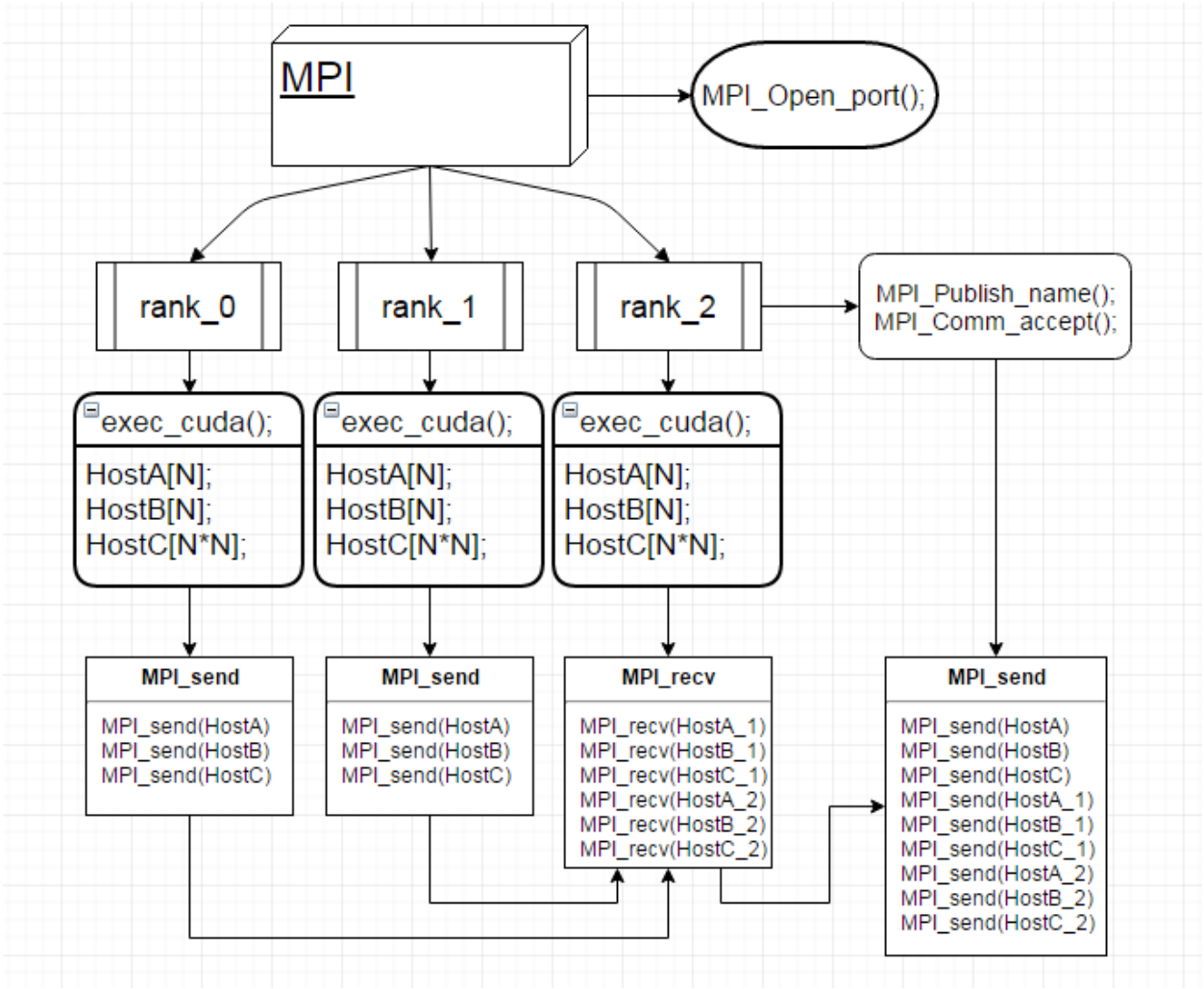

Рисунок 3.9 – Схема работы MPI на стороне сервера

3. Программа MPI на стороне задачи клиента;

Задача клиент «main_cln.c», представляет собой программную реализацию вида запрос принятие результатов от задачи сервера. Т.е. со стороны задачи сервера происходят две команды :

1. MPI_Open_port();
2. MPI_Publish_name().

Как только задача сервер открыла порт и опубликовало имя, по которому будет происходить доступ, задача клиент пытается подключится через заданное имя задачи сервера, это происходит заданием двух команд со стороны задачи клиента:

1. MPI_Lookup_name();
2. MPI_Comm_connect().

Как только мы смогли подключиться к MPI процессу задачи сервера, сервер осуществляет подтверждение:

- MPI_Comm_accept().

После чего осуществляется передача данных от MPI процесса задачи сервера, MPI процессу задачи клиента.

В моём случае, все данные передаются с MPI процесса №2 задачи сервера, на MPI процесс №1 задачи клиента.

На рисунке 3.10 приведена схема работы программы MPI на стороне клиента.

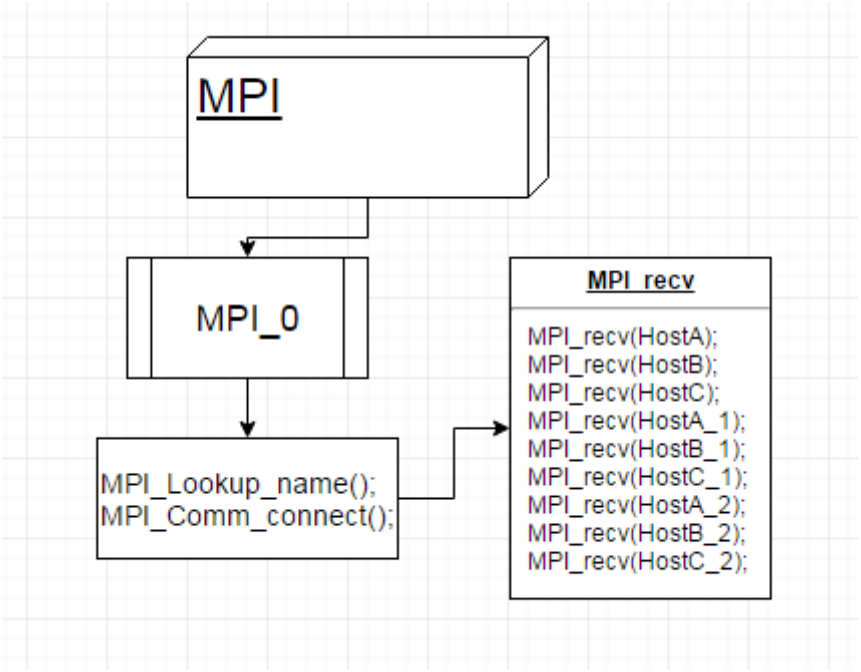

Рисунок 3.10 - Схема работы MPI на стороне клиента

4. Программа MPI+QT на стороне задачи клиента.

На рисунке 3.11 представлена схема MPI+QT, где вся реализация строится на библиотеке HarrixQtLibraryForLaTeX [18], и выводом в трёхмерном пространстве графика как поверхность по координатам, посчитанным в задаче сервера.

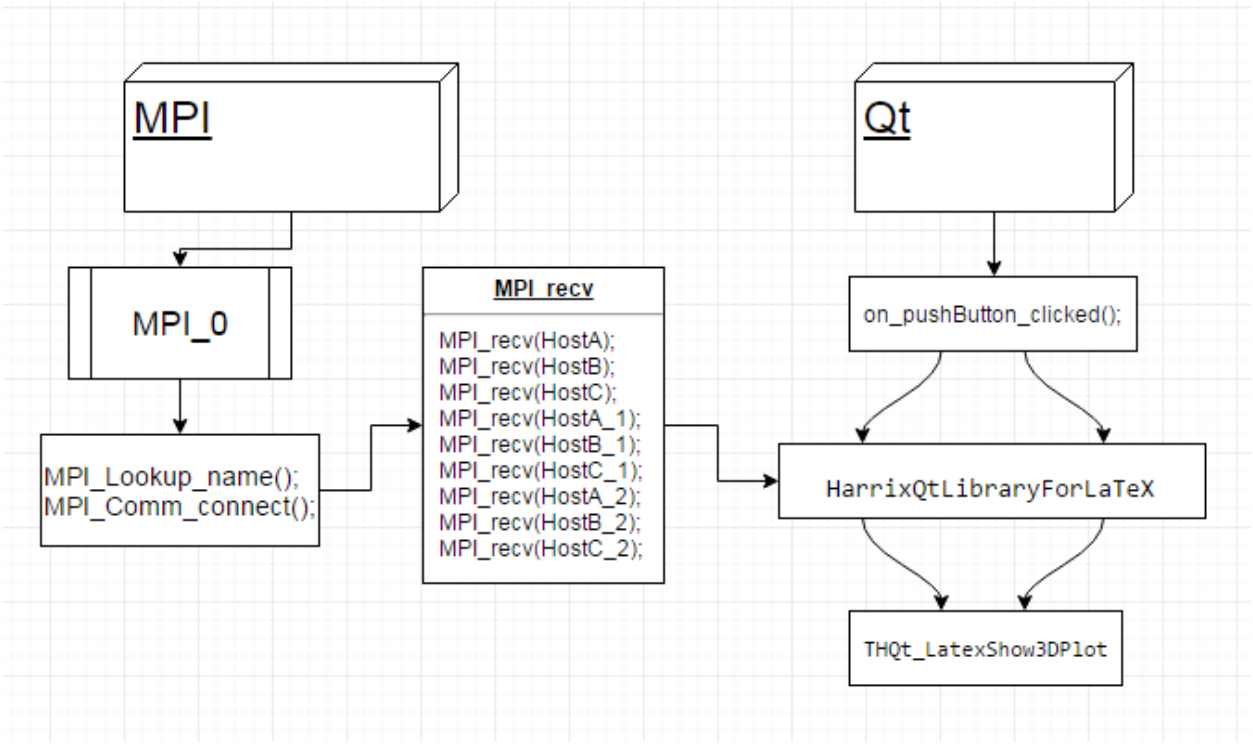

Рисунок 3.11 – Схема MPI+QT

4 Возможности и результаты

4.1 Запуски программы

Главной задачей диссертации являлась, разработка инструментария, с помощью которого можно было бы планировать и анализировать расчёты, которые проводятся на гибридном кластере комплекса ИВЦ НГУ. Все тесты разработанного инструментария проведены как на рабочей станции, так и на реальной задаче, которая была выполнена и протестирована на гибридном кластере.

Весь проект делится на несколько стадий запуска:

- 1 стадия – запуск PBS runIJC_os;

  Данной опцией запускается коммуникатирующий модуль ompi-server.

- 2 стадия – запуск PBS runIJC_srvv;

  Данный модуль несёт в себе два файла, которые компилируются по отдельности, до объектного (.o).

- Для программы с использованием кода написанного под CUDA;
- Для программы с использованием MPI.

Оба по отдельности скомпилированных файла собираются в один исполняемый, который в свою очередь предоставляется PBS на выполнение.

- стадия – запуск PBS runIJC_cln.

  Данный модуль предназначен для коннекта, запроса и принятия данных с модуля описанного под стадией два.

Подробно опишу подготовку к компиляции, саму компиляцию, а также скрипты, с помощью которых выделялся ресурс кластерной системы.

4.1.1 Подготовка к компиляции

Начнем с подготовки к компиляции.

1. Для подключения компилятора CUDA используется команда:
   - export PATH=$PATH:/opt/shared/nvidia/cuda/bin;

- export LD_LIBRARY_PATH=/opt/shared/nvidia/cuda/lib64:/opt/shared/nvidia/cuda/ lib:$LD_LIBRARY_PATH.

Или же без использования миднайт коммандера «mc»:

- module load nvidia/cuda-toolkit.

2. Для подключения компилятора mpicc мною была выбрана самая оптимальная версия openmpi_gcc-1.8.8:

- mpi-selector --set openmpi_gcc-1.8.8;

Обязательно после выполнения выше приведённой команды нужно переподключить сеанс ssh выполнив команду:

- which mpicc.

Примечание:

При каждом сеансе выполняются выше описанные команды однократно – пока не будет завершена работа с кластером.

4.1.2 Компиляция каждой части проекта

Перейдём непосредственно к компиляции каждой части программы.

Описание проведения компиляции каждого из трёх разработанных файлов.

1. Для компиляции программы написанной в совмещении двух технологий CUDA + MPI, назовём её «сервер» необходимо было выполнить три действия:

- Выполнить команду nvcc -arch=compute_20 -c cuda_part.cu, для программы CUDA скомпилировав её до объектного файла (.o);
- Выполнить команду mpicc –c mpi_part.c, для компиляции MPI программы также до объектного файла (.o);
- Далее чтобы собрать два файла в единый требуется выполнить команду mpiCC mpi_part.o cuda_part.o -lm -lcudart -L/opt/shared/nvidia/cuda/lib64 -I/opt/shared/nvidia/cuda/include -o mpi_cuda.

2. Для компиляции программы написанной в совмещении двух технологий MPI + QT, назовём её «клиент», было принято решение на кластере выполнить только часть с использованием MPI, которая отвечала за подключение к MPI процессу задачи «сервера», MPI процесса задачи «клиент», запроса данных, а также приёма данных себе.
   - Для компиляции нужно выполнить команду mpicc main_cln.c -o main_cln.

Примечание к пункту 2.

В связи с использованием Qt5, и методов реализованных в дополнительных библиотеках направленных на построение графиков и диаграмм, не было возможности наглядно протестировать реализованный блок на кластере в полном объёме. Но данную разработку «клиент» опишу, чуть ниже, на тестах с использованием своей рабочей станции.

4.1.3 Запуски PBS скриптов

Для сборки проекта в едино, запускаются 3 стадии PBS. Начну с PBS скриптов в том порядке, как их следует запускать при выполни работы.

1. qsub runIJC_os – запуск ompi-server;

Скрипт runIJC_os приведён на рисунке 4.1.

```
#!/bin/sh
#PBS -q teslaq
#PBS -l walltime=0:05:00
#PBS -l select=1:ncpus=1:mpiprocs=1:mem=6gb,place=scatter

cd $PBS_O_WORKDIR

ompi-server --no-daemonize -d -r mpiuri_2
```

Рисунок 4.1 – Скрипт runIJC_os

2. qsub rinIJC_srvv – запуск программы «сервер», или же CUDA +MPI;

Скрипт runIJC_ srvv приведён на рисунке 4.2.

```
#!/bin/sh

#PBS -q teslaq
#PBS -l walltime=0:03:00
#PBS -l select=1:ngpus=3:ncpus=1:mpiprocs=3:mem=32gb,place=scatter

cd $PBS_O_WORKDIR
#ompi-server --no-daemonize -d -r mpiuri_1
vnodes=$(qstat -f $PBS_JOBID|tr -d '\n'' ''\t'|sed 's/Hold_Types.*//'|sed 's/.*exec_vnode=//'|tr -d \(\)|tr + '\n'|sed 's/:.*//'|sort)
echo "My vnodes:"
for vnode in $vnodes ; do
    node=$(echo $vnode|sed 's/\[.*//')
    gpu=$(echo $vnode|sed 's/.*\[//'|sed 's/\]//')
    echo " $vnode = Node $node, GPU $gpu"
done
echo

## Replace all '[' and ']' by spaces before passing to program:
vnodes=$(echo $vnodes|sed 's/\[/ /g'|sed 's/\]/ /g')

export LD_LIBRARY_PATH=/opt/shared/nvidia/cuda/lib64:/opt/shared/nvidia/cuda/lib:$LD_LIBRARY_PATH
mpirun -x LD_LIBRARY_PATH -hostfile $PBS_NODEFILE --ompi-server file:mpiuri_2 ./mpi_cuda $vnodes
```

Рисунок 4.2 - Скрипт runIJC_ srvv

3. qsub rinIJC_cln – запуск программы «клиент», или же MPI.

Скрипт runIJC_ cln приведён на рисунке 4.3.

```
#!/bin/sh

#PBS -q teslaq
#PBS -l walltime=0:01:00
#PBS -l select=1:ncpus=1:mpiprocs=1:mem=6gb,place=scatter

cd $PBS_O_WORKDIR

mpirun -hostfile $PBS_NODEFILE   --ompi-server file:mpiuri_2  -np 1 ./main_cln
```

Рисунок 4.3. - Скрипт runIJC_ cln

4.2 Результаты выполнения

1. Результаты выполнения после завершения работы PBS скрипта - rinIJC_srvv:
   - Результат: runIJC_srvv.e2680878;

Результат работы задачи «сервер» в исходнике runIJC_srvv.e2680878 , представлен на рисунке 4.4.

```
name is ocean   port is -1462225008 process # 2
```

Рисунок 4.4. - Результат работы задачи «сервер» в исходнике runIJC_srvv.e2680878

- Результат: runIJC_srvv.o2680878.

Результат работы задачи «сервер» в исходнике runIJC_srvv.o2680878 , представлен на рисунке 4.5. – часть рисунка №1.

```
My vnodes:
 sl008[0] = Node sl008, GPU 0
 sl008[1] = Node sl008, GPU 1
 sl008[2] = Node sl008, GPU 2

Finished rank 1
I'm number 1 from 3 and I run on host sl008, gpu 0.
Finished rank 0
I'm number 0 from 3 and I run on host sl008, gpu 2.
Received from the rank 0
-3.000000 + -3.000000 =  0.000000
-3.000000 + -2.000000 =  0.000009
-3.000000 + -1.000000 =  0.000814
-3.000000 + 0.000000 =  0.009913
-3.000000 + 1.000000 =  0.016344
-2.000000 + -3.000000 =  0.000000
-2.000000 + -2.000000 =  0.000086
-2.000000 + -1.000000 =  0.007726
-2.000000 + 0.000000 =  0.094125
-2.000000 + 1.000000 =  0.155186
-1.000000 + -3.000000 =  0.000000
-1.000000 + -2.000000 =  0.000331
-1.000000 + -1.000000 =  0.029818
-1.000000 + 0.000000 =  0.363252
-1.000000 + 1.000000 =  0.598901
0.000000 + -3.000000 =  0.000001
0.000000 + -2.000000 =  0.000520
0.000000 + -1.000000 =  0.046771
0.000000 + 0.000000 =  0.569783
0.000000 + 1.000000 =  0.939413
1.000000 + -3.000000 =  0.000000
1.000000 + -2.000000 =  0.000331
1.000000 + -1.000000 =  0.029818
1.000000 + 0.000000 =  0.363252
1.000000 + 1.000000 =  0.598901
```

Рисунок 4.5. - Результат работы задачи «сервер» в исходнике runIJC_srvv.o2680878. – часть рисунка №1

Результат работы задачи «сервер» в исходнике runIJC_srvv.o2680878 , представлен на рисунке 4.6. – часть рисунка №2.

```
Received from the rank 1
-2.000000 + -2.000000 =  0.000086
-2.000000 + -1.000000 =  0.007726
-2.000000 + 0.000000 =  0.094125
-2.000000 + 1.000000 =  0.155186
-2.000000 + 2.000000 =  0.034627
-1.000000 + -2.000000 =  0.000331
-1.000000 + -1.000000 =  0.029818
-1.000000 + 0.000000 =  0.363252
-1.000000 + 1.000000 =  0.598901
-1.000000 + 2.000000 =  0.133633
0.000000 + -2.000000 =  0.000520
0.000000 + -1.000000 =  0.046771
0.000000 + 0.000000 =  0.569783
0.000000 + 1.000000 =  0.939413
0.000000 + 2.000000 =  0.209611
1.000000 + -2.000000 =  0.000331
1.000000 + -1.000000 =  0.029818
1.000000 + 0.000000 =  0.363252
1.000000 + 1.000000 =  0.598901
1.000000 + 2.000000 =  0.133633
2.000000 + -2.000000 =  0.000086
2.000000 + -1.000000 =  0.007726
2.000000 + 0.000000 =  0.094125
2.000000 + 1.000000 =  0.155186
2.000000 + 2.000000 =  0.034627
```

Рисунок 4.6. - Результат работы задачи «сервер» в исходнике runIJC_srvv.o2680878. – часть рисунка №2

Результат работы задачи «сервер» в исходнике runIJC_srvv.o2680878 , представлен на рисунке 4.7. – часть рисунка №3.

```
I'm number 2 from 3 and I run on host sl008, gpu 1.
rank 2
-1.000000 + -1.000000 =  0.029818
-1.000000 + 0.000000 =  0.363252
-1.000000 + 1.000000 =  0.598901
-1.000000 + 2.000000 =  0.133633
-1.000000 + 3.000000 =  0.004035
0.000000 + -1.000000 =  0.046771
0.000000 + 0.000000 =  0.569783
0.000000 + 1.000000 =  0.939413
0.000000 + 2.000000 =  0.209611
0.000000 + 3.000000 =  0.006330
1.000000 + -1.000000 =  0.029818
1.000000 + 0.000000 =  0.363252
1.000000 + 1.000000 =  0.598901
1.000000 + 2.000000 =  0.133633
1.000000 + 3.000000 =  0.004035
2.000000 + -1.000000 =  0.007726
2.000000 + 0.000000 =  0.094125
2.000000 + 1.000000 =  0.155186
2.000000 + 2.000000 =  0.034627
2.000000 + 3.000000 =  0.001046
3.000000 + -1.000000 =  0.000814
3.000000 + 0.000000 =  0.009913
3.000000 + 1.000000 =  0.016344
3.000000 + 2.000000 =  0.003647
3.000000 + 3.000000 =  0.000110
```

Рисунок 4.7. - Результат работы задачи «сервер» в исходнике runIJC_srvv.o2680878. – часть рисунка №3

2. Результаты выполнения после завершения работы PBS скрипта - rinIJC_cln:

- Результат: runIJC_cln.e2680879;

Результат работы задачи «клиент» в исходнике runIJC_ cln.e2680879, представлен на рисунке 4.8.

```
Before myrank
rank 0
name is ocean   port is -1681783632 process # 0
```

Рисунок 4.8. - Результат работы задачи «клиент» в исходнике runIJC_ cln.e2680879

- Результат: runIJC_cln.o2680879.

Результат работы задачи «клиент» в исходнике runIJC_ cln.o2680879, представлен на рисунке 4.9. – часть рисунка №1.

```
inform from rank 2
 --------------------
x= -1.000000 y= -1.000000 z= 0.029818
x= -1.000000 y= 0.000000 z= 0.363252
x= -1.000000 y= 1.000000 z= 0.598901
x= -1.000000 y= 2.000000 z= 0.133633
x= -1.000000 y= 3.000000 z= 0.004035
x= 0.000000 y= -1.000000 z= 0.046771
x= 0.000000 y= 0.000000 z= 0.569783
x= 0.000000 y= 1.000000 z= 0.939413
x= 0.000000 y= 2.000000 z= 0.209611
x= 0.000000 y= 3.000000 z= 0.006330
x= 1.000000 y= -1.000000 z= 0.029818
x= 1.000000 y= 0.000000 z= 0.363252
x= 1.000000 y= 1.000000 z= 0.598901
x= 1.000000 y= 2.000000 z= 0.133633
x= 1.000000 y= 3.000000 z= 0.004035
x= 2.000000 y= -1.000000 z= 0.007726
x= 2.000000 y= 0.000000 z= 0.094125
x= 2.000000 y= 1.000000 z= 0.155186
x= 2.000000 y= 2.000000 z= 0.034627
x= 2.000000 y= 3.000000 z= 0.001046
x= 3.000000 y= -1.000000 z= 0.000814
x= 3.000000 y= 0.000000 z= 0.009913
x= 3.000000 y= 1.000000 z= 0.016344
x= 3.000000 y= 2.000000 z= 0.003647
x= 3.000000 y= 3.000000 z= 0.000110
```

Рисунок 4.9. - Результат работы задачи «клиент» в исходнике runIJC_cln.o2680879. – часть рисунка №1

Результат работы задачи «клиент» в исходнике runIJC_ cln.o2680879, представлен на рисунке 4.10. – часть рисунка №2

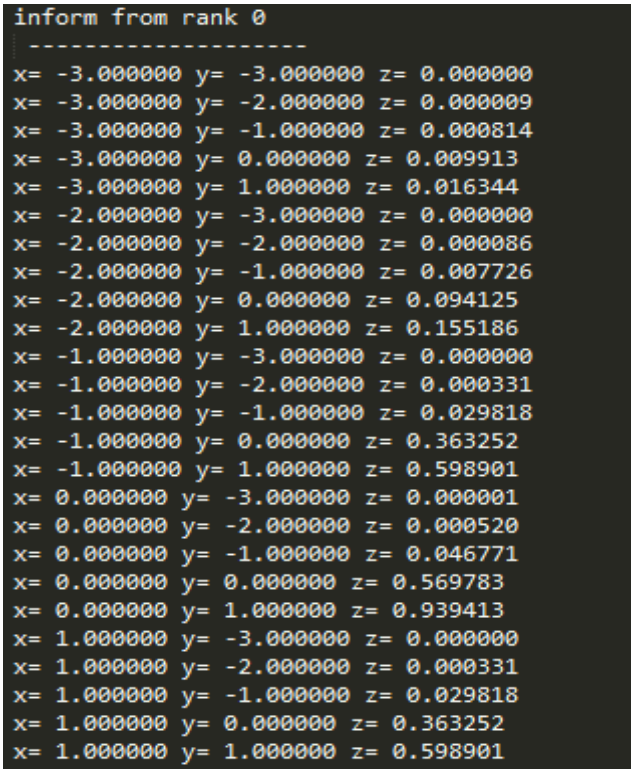

Рисунок 4.10. - Результат работы задачи «клиент» в исходнике runIJC_cln.o2680879. – часть рисунка №2

Результат работы задачи «клиент» в исходнике runIJC_ cln.o2680879, представлен на рисунке 4.11. – часть рисунка №3.

```
inform from rank 1
 --------------------
x= -2.000000 y= -2.000000 z= 0.000086
x= -2.000000 y= -1.000000 z= 0.007726
x= -2.000000 y= 0.000000 z= 0.094125
x= -2.000000 y= 1.000000 z= 0.155186
x= -2.000000 y= 2.000000 z= 0.034627
x= -1.000000 y= -2.000000 z= 0.000331
x= -1.000000 y= -1.000000 z= 0.029818
x= -1.000000 y= 0.000000 z= 0.363252
x= -1.000000 y= 1.000000 z= 0.598901
x= -1.000000 y= 2.000000 z= 0.133633
x= 0.000000 y= -2.000000 z= 0.000520
x= 0.000000 y= -1.000000 z= 0.046771
x= 0.000000 y= 0.000000 z= 0.569783
x= 0.000000 y= 1.000000 z= 0.939413
x= 0.000000 y= 2.000000 z= 0.209611
x= 1.000000 y= -2.000000 z= 0.000331
x= 1.000000 y= -1.000000 z= 0.029818
x= 1.000000 y= 0.000000 z= 0.363252
x= 1.000000 y= 1.000000 z= 0.598901
x= 1.000000 y= 2.000000 z= 0.133633
x= 2.000000 y= -2.000000 z= 0.000086
x= 2.000000 y= -1.000000 z= 0.007726
x= 2.000000 y= 0.000000 z= 0.094125
x= 2.000000 y= 1.000000 z= 0.155186
x= 2.000000 y= 2.000000 z= 0.034627
```

Рисунок 4.11. - Результат работы задачи «клиент» в исходнике runIJC_cln.o2680879. – часть рисунка №3

3. Результаты выполнения после завершения работы PBS скрипта - rinIJC_os:

После запуска PBS скрипта rinIJC_os – создаётся файл mpiuri_2.

Содержимое файла mpiuri_2 представлено на рисунке 4.12.

```
1342767104.0;tcp://10.10.1.205,10.11.1.205:40382
```

Рисунок 4.12. - Содержимое файла mpiuri_2

- Результат: runIJC_os.e2680877;

Результат работы сервера «ompi-server» в исходнике runIJC_os.e2680877, представлен на рисунке 4.13.

```
[sl005:13457] procdir: /var/tmp/pbs.2680877.hpc-suvir1.hpc.nusc.ru/openmpi-sessions-msnemtzev@sl005_0/20489/0/0
[sl005:13457] jobdir: /var/tmp/pbs.2680877.hpc-suvir1.hpc.nusc.ru/openmpi-sessions-msnemtzev@sl005_0/20489/0
[sl005:13457] top: openmpi-sessions-msnemtzev@sl005_0
[sl005:13457] tmp: /var/tmp/pbs.2680877.hpc-suvir1.hpc.nusc.ru
[sl005:13457] sess_dir_cleanup: job session dir does not exist
[sl005:13457] procdir: /var/tmp/pbs.2680877.hpc-suvir1.hpc.nusc.ru/openmpi-sessions-msnemtzev@sl005_0/20489/0/0
[sl005:13457] jobdir: /var/tmp/pbs.2680877.hpc-suvir1.hpc.nusc.ru/openmpi-sessions-msnemtzev@sl005_0/20489/0
[sl005:13457] top: openmpi-sessions-msnemtzev@sl005_0
[sl005:13457] tmp: /var/tmp/pbs.2680877.hpc-suvir1.hpc.nusc.ru
[sl005:13457] [[20489,0],0] orte-server: up and running!
=>> PBS: job killed: walltime 333 exceeded limit 300
[sl005:13457] [[20489,0],0] orte-server: finalizing
[sl005:13457] sess_dir_cleanup: job session dir does not exist
```

Рисунок 4.13. - Результат работы сервера «ompi-server» в исходнике runIJC_os.e2680877

- Результат: runIJC_os.o2680877.

Результат работы сервера ompi-server» в исходнике runIJC_os.o2680877 пустой.

4.3 Запуски клиентской части

Данный подраздел посвящён полному описанию выполнения «клиентской» части, так как задумывалось, т.е. с использованием MPI +QT.[18].

Описание компиляции совмещённой технологии MPI+QT. Для компиляции нужно последовательно выполнить следующие действия:

1. qmake –project –o main_cln.pro;
2. По сути можно считать основным шагом, нужно модернизировать созданный на шаге №1 файл main_cln.pro.

Модернизация файла main_cln.pro, а именно, дописав в него, часть отвечающую за подключение библиотеки #include <mpi.h>, данный участок кода представлен на рисунке 4.14.

```
# MPI Settings/////////////////////////////////////////////////
QMAKE_CXX = mpicxx
QMAKE_CXX_RELEASE = $$QMAKE_CXX
QMAKE_CXX_DEBUG = $$QMAKE_CXX
QMAKE_LINK = $$QMAKE_CXX
QMAKE_CC = mpicc

QMAKE_CFLAGS += $$system(mpicc --showme:compile)
QMAKE_LFLAGS += $$system(mpicxx --showme:link)
QMAKE_CXXFLAGS += $$system(mpicxx --showme:compile) -DMPICH_IGNORE_CXX_SEEK
QMAKE_CXXFLAGS_RELEASE += $$system(mpicxx --showme:compile) -DMPICH_IGNORE_CXX_SEEK
#/////////////////////////////////////////////////////////////
```

Рисунок 4.14. – Код для подключения библиотеки #include <mpi.h> в Qt

3. Выполнить команду: qmake main_cln.pro -o Makefile;

4. Выполнить команду: make;

5. Запустить программу на выполнение: mpirun –np 1 main_cln.

Тестирование «клиентской» части в совмещении технологий MPI+QT, было сделано на основе подсчётов результатов обработки реальной задачи на гибридном кластере комплекса ИВЦ.

Приведу решаемую задачу – в виде тестирования реализованного метода.

Задача:

Нужно рассчитать функцию от трёх переменных.

$$z = f(x, y) \tag{4.1}$$

$$v_z = e^{-\frac{(v_x - V)^2}{T_1} - \frac{(v_y)^2}{T_2}} * \frac{1}{2\pi\sqrt{T_1 * T_2}} \tag{4.2}$$

где

$$\begin{cases} V = 0{,}75 \\ T_1 = 1 \\ T_2 = 0{,}5 \end{cases} \tag{4.3}$$

$$(v_x, v_y) = [-3;3] \times [-3;3] \tag{4.4}$$

Работу «клиентской» части можно увидеть на рисунке 4.15.

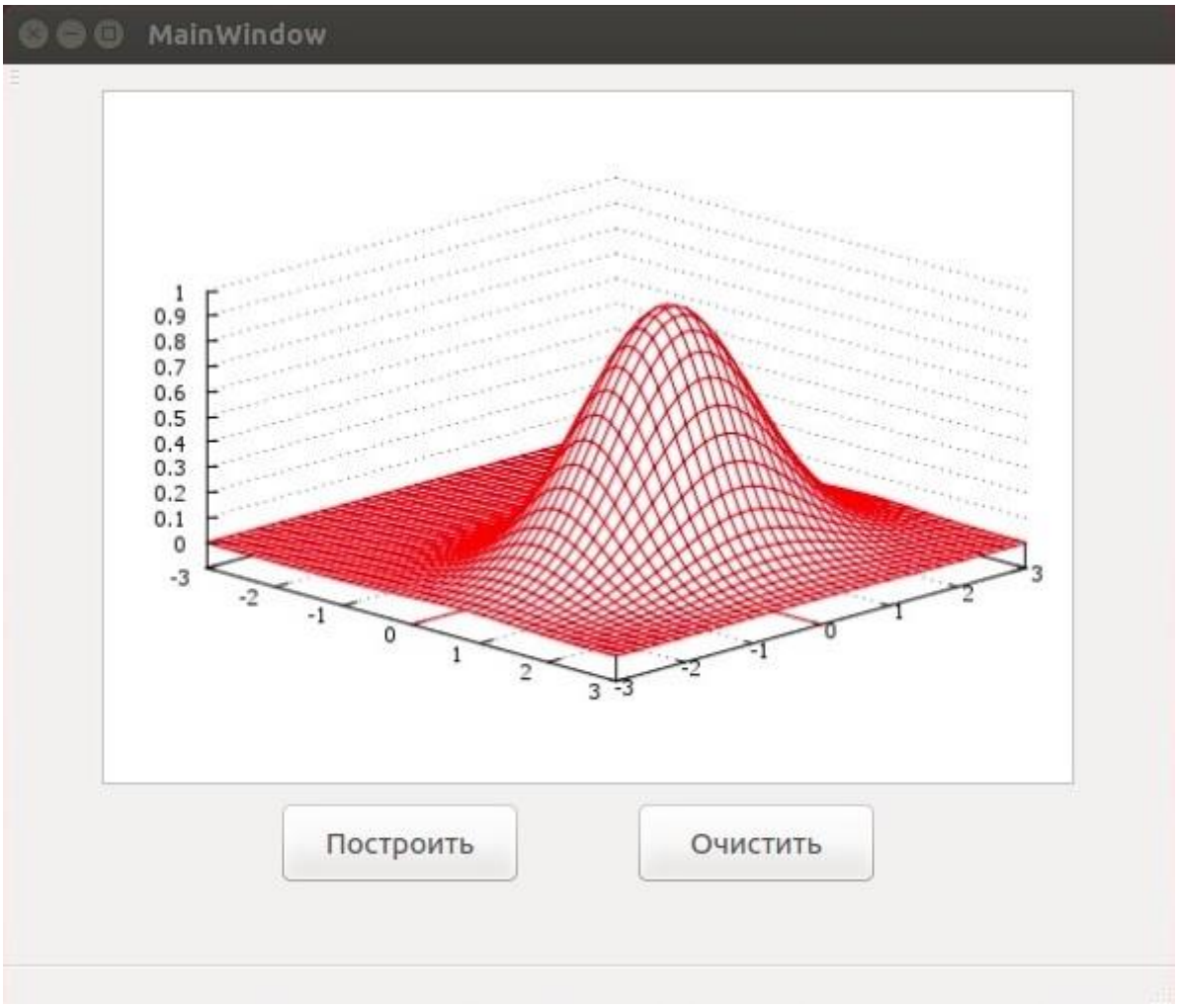

Рисунок 4.15. – Работа «клиентской» части

ЗАКЛЮЧЕНИЕ

Разработанная система показала эффективность в оценке решения сложных задач проводимых на гибридном кластере.

Основная цель дипломного проекта была полностью достигнута путём разработки и реализации метода динамического доступа к графической памяти, который в свою очередь, исходя из проведённых тестов на задаче, воссозданной на гибридном кластере, помог оценить ход решения работы этой задачи.

Метод показал:

- доступ к графической памяти можно проводить как одой задачей так и одновременно несколькими задачами, в любой момент времени, пока запущена задача, к графической памяти которой, нужно получить доступ;
- возможен доступ к графической памяти, как только одной к примеру карте, так и к нескольким на которых проводятся вычисления – по необходимости;
- решение задачи, проводимой на GPU и пересылка, намного эффективнее по времени, чем, если бы мы, решали эту же задачу на CPU.

Также были выполнены все поставленные во время работы этапы «задачи»:

1. было разработано наиболее подходящее средство для обмена данными между задачами;
2. проведена реализация и тестирование разработанного средства на гибридном кластере – как обычная пересылка заданных значений изначально;
3. реализована программа на гибридном кластере, по расчёту уравнения гидродинамики, используя ресурс трёх графических карт (GPU), также для полноты расширения реализованного метода задача запускалась на двух узлах с использованием шести графических карт;

4. были проведены тесты разработанного метода, в совместном использовании с программой, по расчёту уравнения гидродинамики разработанной на шаге 3.
5. данные запрашиваемые задачей «клиент», представляются с помощью окна Qt, на основе взятой библиотеки «HarrixQtLibraryForLaTeX» [15], для простого построения 3D поверхностей и графиков.

Подводя итог, можно сказать разработанный метод доступа к графической памяти, хорошо применим к любым задачам любой сложности, в том числе задачам гидродинамики, таким как течений разреженных газов.

Метод применим и будет востребован «сказано с моих слов», в различных областях знаний, таких как физика, вычислительная математика, химия и т.д., используя данное средство возможно в любой момент времени просмотреть результаты выполняющейся задачи, тем самым определить на ранних этапах решения нужно ли продолжать выполнять задачу или отталкиваясь на полученные результаты, будет достаточно чтобы определить дальнейший результат.

Также по данным можно не только моделировать данные в виде графика, как представлено в программе, так и по необходимости влиять на результаты проводя дальнейшие вычисления над ними.

Разработанный метод позволит более эффективно планировать и проводить специализированные вычислительные эксперименты в различных областях знаний, облегчит работу специалистов, которые будут применять данный метод на практике для своих задач.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Алексеев А. Г., Йовенко А. Р. Параллельное программирование: учебное пособие. Чебоксары: Чуваш. ун-та, 2014. 200 с.
2. Корнеев В. Д. Параллельное программирование в MPI: 2-е изд., Новосибирск: ИВМиМГ СО РАН, 2002. 215 с.
3. Корнеев В. В. Параллельное программирование в MPI. М.: Ижевск, Институт компьютерных исследований, 2003.
4. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ - Петербург, 2002. 400 с.
5. Параллельные вычисления с CUDA [Электронный ресурс]. URL: http://www.nvidia.ru/object/cuda-parallel-computing-ru.html (дата обращения: 28.05.2016).
6. Nvidia CUDA, неграфические вычисления на графических процессорах [Электронный ресурс]. URL: http://www.ixbt.com/video3/cuda-1.shtml (дата обращения: 24.05.2016).
7. NVidia CUDA: вычисления на видеокарте [Электронный ресурс]. URL: http://www.thg.ru/graphic/nvidia_cuda/print.html (дата обращения: 20.05.2016).
8. Богомолов С. Е. Использование модели массового параллелизма CUDA для разработки программ [Электронный ресурс]. URL: http://www.bog.pp.ru/work/cuda.html (дата обращения: 21.05.2016).
9. Боресков А. В. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: учебное пособие. М.: Перепелт, 2012. 336 с.
10. Сандерс Д., Кэндрот Э. Технология CUDA в примерах. Введение в программирование графических процессоров. М.: ДМК Пресс, 2011. 232с.
11. Шлее М. Профессиональное программирование на C++ Qt 4.8. БХВ Петербург.: 2012. 894 с.

12. Кластерные вычислительные системы [Электронный ресурс]. URL: http://www.intuit.ru/studies/courses/1058/235/lecture/6093 (дата обращения: 07.05.2016).
13. Информационно-вычислительный центр НГУ (ИВЦ) [Электронный ресурс]. URL: http://www.nusc.ru (дата обращения: 04.05.2016).
14. Стивенс У. Р. UNIX: взаимодействие процессов. Питер: 2002. 576с.
15. Антонов А. Технология параллельное программирование MPI и OpenMP. М.: МГУ, 2000. 344с.
16. Сергиенко А. Cpp for Qt. Библиотека для отображения различных данных в LaTeX [Электронный ресурс]. URL: https://github.com/Harrix/Harrix-QtLibraryForLaTeX (дата обращения: 20.05.2016).
17. Establishing Communication [Электронный ресурс]. URL: http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/node100.htm (дата обращения: 06.05.2016).
18. Построение графиков в LaTeX / PGFPlots [Электронный ресурс]. URL: https://habrahabr.ru/post/250997/ (дата обращения: 15.05.2016).
19. Ливак Е.Н. Параллельное программирование в MPI [Электронный ресурс]. URL: http://mf.grsu.by/UchProc/livak/kursoviki/adamin/kurs3.htm/kurs3/part1.htm (дата обращения: 20.05.2016).
20. Организация MPI [Электронный ресурс]. URL: http://cluster.univer.omsk.su/mpi.html (дата обращения: 10.05.2016).
21. Воеводин В. В. Технологии параллельного программирования. Message Passing Interface (MPI) [Электронный ресурс]. URL: http://www.citforum.idknet.com/hardware/ arch/lectures/5.shtml (дата обращения: 21.05.2016).
22. Бабенко Л. К. Параллельные алгоритмы для решения задач защиты информации: 2-е изд., Москва: Горячая линия-Телеком, 2016. 304 с.
23. Немцев М. С. Разработка инструментария для планирования и анализа расчётов на гибридных кластерах течений разреженного газа // РНТК к ДР. Новосибирск: 2016. 6 с.

ПРИЛОЖЕНИЕ А

Программа «cuda_part.cu»

```
#include <stdio.h>
#include "cuda.h"

#define N 5
#define K 5
int assigned_device;
int used_device;

float HostA[N];
float HostB[N];
float HostC[N*N];
float *DeviceA;
float *DeviceB;
float *DeviceC;

__global__ void AddVectors(float* a, float* b, float* c) {
   int i = threadIdx.x;
   int j = blockIdx.x;
   int k = blockDim.x*j+i;
    c[k]= __expf(- ( ( a[i] - 0.75) * ( a[i] - 0.75 ) / 1 ) - ( b[j] * b[j] / 0.5 )*(1 / (2 *
3.1415926535 * sqrt( 1 * 0.5 ))));
}
extern "C" void exec_cuda(int mpi_rank, int assigned_device, float* HostA, float*
HostB, float* HostC) {
if ( cudaSetDevice(assigned_device) != cudaSuccess or
cudaGetDevice(&used_device ) != cudaSuccess or
   used_device != assigned_device ) {
   printf ("Error: unable to set device %d\n", assigned_device);
   return;
 }
   for(int i=0; i<7; i++){
   HostA[i] =i-3+mpi_rank;
   HostB[i] =i-3+mpi_rank;
   }
   cudaMalloc((void**)&DeviceA, N*sizeof(float));
   cudaMalloc((void**)&DeviceB, N*sizeof(float));
   cudaMalloc((void**)&DeviceC, N*N*sizeof(float));
   cudaMemcpy(DeviceA, HostA, N*sizeof(float), cudaMemcpyHostToDevice);
   cudaMemcpy(DeviceB, HostB, N*sizeof(float), cudaMemcpyHostToDevice);
   AddVectors<<<N+1, N>>>(DeviceA, DeviceB, DeviceC);
   cudaMemcpy(HostC, DeviceC, N*N*sizeof(float), cudaMemcpyDeviceToHost);
```

Программа «mpi_part.c»

```
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define N 5
#define K 5

extern void exec_cuda(int, int, float*, float*, float*);

float HostC[N*N];
float HostA[N];
float HostB[N];
float HostA_1[N];
float HostB_1[N];
float HostC_1[N*N];
float HostA_2[N];
float HostB_2[N];
float HostC_2[N*N];
int *gpu_by_rank;
int *vnode_is_used;

int define_gpu (int rank, char *host, int argc, char** argv) {
   int i, gpu = -1;

   for (i=1; i<=(argc-1)/2; i++) {
      if ( strcmp(host,argv[i*2-1]) != 0 )
         continue;
      if ( vnode_is_used[i-1] !=0 )
         continue;
      gpu=atoi(argv[i*2]);
      gpu_by_rank[rank] = gpu;
      vnode_is_used[i-1] = 1;
      break;
   }
   return gpu;
}

int main(int argc, char** argv)
{
   int size, rank, i, k, gpu, tag=0;
   char host[32];
   char* portname;
```

```
    char myport[MPI_MAX_PORT_NAME];
    MPI_Comm  intercomm;
    MPI_Info info;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Info_create(&info);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Open_port(MPI_INFO_NULL, myport);

    if ( size != (argc-1)/2 ) {
       MPI_Finalize();
       printf ("Error: amount of GPUs do not match the MPI size!\n");
       return 100;
    }

    if ( rank == 0 ) {
       gpu_by_rank = (int *)malloc(size*sizeof(int));
       vnode_is_used = (int *)malloc(size*sizeof(int));
       for ( i=0; i<size; i++ ) {
          gpu_by_rank[i]=-1;
          vnode_is_used[i]=0;
       }
       for ( i=1; i<size; i++ ) {
          MPI_Recv(host,32,MPI_CHAR,i,0,MPI_COMM_WORLD, &status);
          gpu = define_gpu(i,host, argc, argv);
          MPI_Send(&gpu,1,MPI_INT,i,0,MPI_COMM_WORLD);
       }
       gethostname(host,32);
       gpu=define_gpu(0,host, argc, argv);
       free(gpu_by_rank);
       free(vnode_is_used);
    } else {
       gethostname(host,32);
       MPI_Send(host,32,MPI_CHAR,0,0,MPI_COMM_WORLD);
       MPI_Recv(&gpu,1,MPI_INT,0,0,MPI_COMM_WORLD, &status);
    }

if (rank==0){
    int i,j;
    printf("Finished rank 0\n");
    printf("I'm number %d from %d and I run on host %s, gpu
%d.\n",rank,size,host,gpu);
    exec_cuda(rank,gpu,HostA,HostB,HostC);
    MPI_Send(HostA, N, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
```

```
   MPI_Send(HostB, N, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
   MPI_Send(HostC, N*N, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
}

   if (rank==1){
   int i,j;
   printf("Finished rank 1\n");
   printf("I'm number %d from %d and I run on host %s, gpu
%d.\n",rank,size,host,gpu);
   exec_cuda(rank,gpu,HostA,HostB,HostC);
   MPI_Send(HostA, N, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
   MPI_Send(HostB, N, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
   MPI_Send(HostC, N*N, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
}

   if (rank==2){
   MPI_Publish_name("ocean", info, myport);
   MPI_Comm_accept(myport, info, 0, MPI_COMM_SELF, &intercomm);
   fprintf(stderr, "name is %s\tport is %d\tprocess # %d\n","ocean", myport, rank);

   int i,j;

   MPI_Recv(HostA_1, N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status );
   MPI_Recv(HostB_1, N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status );
   MPI_Recv(HostC_1, N*N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status );
   printf("Received from the rank 0\n");
   for( i=0;i<K;i++){
   for( j=0;j<K;j++){
   printf("%f + %f =  %f\n",HostA_1[i], HostB_1[j], HostC_1[i*N+j]);
   }
   }

   MPI_Recv(HostA_2, N, MPI_FLOAT, 1, 0, MPI_COMM_WORLD, &status );
   MPI_Recv(HostB_2, N, MPI_FLOAT, 1, 0, MPI_COMM_WORLD, &status );
   MPI_Recv(HostC_2, N*N, MPI_FLOAT, 1, 0, MPI_COMM_WORLD, &status );
   printf("Received from the rank 1\n");
   for( i=0;i<K;i++){
   for( j=0;j<K;j++){
   printf("%f + %f =  %f\n",HostA_2[i], HostB_2[j], HostC_2[i*N+j]);
   }
   }

   printf("I'm number %d from %d and I run on host %s, gpu
%d.\n",rank,size,host,gpu);
   exec_cuda(rank,gpu,HostA,HostB,HostC);
```

```
    printf("rank 2\n");
    for( i=0;i<K;i++){
    for( j=0;j<K;j++){
    printf("%f + %f =  %f\n",HostA[i], HostB[j], HostC[i*N+j]);
    }
    }

    MPI_Send(HostA, N, MPI_FLOAT, 0, 0, intercomm);
    MPI_Send(HostB, N, MPI_FLOAT, 0, 0, intercomm);
    MPI_Send(HostC, N*N, MPI_FLOAT, 0, 0, intercomm);
    MPI_Send(HostA_1, N, MPI_FLOAT, 0, 0, intercomm);
    MPI_Send(HostB_1, N, MPI_FLOAT, 0, 0, intercomm);
    MPI_Send(HostC_1, N*N, MPI_FLOAT, 0, 0, intercomm);
    MPI_Send(HostA_2, N, MPI_FLOAT, 0, 0, intercomm);
    MPI_Send(HostB_2, N, MPI_FLOAT, 0, 0, intercomm);
    MPI_Send(HostC_2, N*N, MPI_FLOAT, 0, 0, intercomm);
}
    MPI_Finalize();
    return 0;
}
```

Программа «main_cln.c»

```
#include <mpi.h>
#include <stdio.h>
#include <malloc.h>

#define N 5
#define K 5

int main(int argc, char** argv)
{
    int size, my_rank, tag;
    float HostA[N];
    float HostB[N];
    float HostC[N*N];
    float HostA_1[N];
    float HostB_1[N];
    float HostC_1[N*N];
    float HostA_2[N];
    float HostB_2[N];
    float HostC_2[N*N];
    char portname[MPI_MAX_PORT_NAME];
    char myport[MPI_MAX_PORT_NAME];
    MPI_Comm  intercomm;
```

```
   MPI_Info info;
   MPI_Status status;

   MPI_Init(&argc,&argv);
   MPI_Info_create(&info);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   fprintf(stderr,"Before myrank\n");
   MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
   fprintf(stderr, "rank %d\n", my_rank);

   if(my_rank==0){
   MPI_Lookup_name("ocean",  MPI_INFO_NULL, portname);
   MPI_Comm_connect(portname, MPI_INFO_NULL, 0, MPI_COMM_SELF,
&intercomm);

   fprintf(stderr, "name is %s\tport is %d\tprocess # %d\n","ocean", portname,
my_rank);
   int i,j;

   MPI_Recv(HostA, N, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
intercomm, &status );
   MPI_Recv(HostB, N, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
intercomm, &status );
   MPI_Recv(HostC, N*N, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
intercomm, &status );
   printf("inform from rank 2 \n ----------\n");
   for(i=0;i<K;i++){
   for(j=0;j<K;j++){
   fprintf(stdout,"x= %f y= %f z= %f\n", HostA[i], HostB[j], HostC[i*N+j]);
   }
   }

   MPI_Recv(HostA_1, N, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
intercomm, &status );
   MPI_Recv(HostB_1, N, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
intercomm, &status );
   MPI_Recv(HostC_1, N*N, MPI_FLOAT, MPI_ANY_SOURCE,
MPI_ANY_TAG, intercomm, &status );
   printf("inform from rank 0 \n ----------\n");
   for(i=0;i<K;i++){
   for(j=0;j<K;j++){
   fprintf(stdout,"x= %f y= %f z= %f\n", HostA_1[i], HostB_1[j], HostC_1[i*N+j]);
   }
   }
```

```
   MPI_Recv(HostA_2, N, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
intercomm, &status );
   MPI_Recv(HostB_2, N, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
intercomm, &status );
   MPI_Recv(HostC_2, N*N, MPI_FLOAT, MPI_ANY_SOURCE,
MPI_ANY_TAG, intercomm, &status );
   printf("inform from rank 1 \n ----------\n");
   for(i=0;i<K;i++){
   for(j=0;j<K;j++){
   fprintf(stdout,"x= %f y= %f z= %f\n", HostA_2[i], HostB_2[j], HostC_2[i*N+j]);
   }
   }
}
   MPI_Finalize();
}
```