**Федеральное государственное автономное образовательное учреждение   
высшего образования**

**КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ**

**ВЫСШАЯ ШКОЛА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И   
ИНФОРМАЦИОННЫХ СИСТЕМ**

Направление подготовки: 09.03.03 Прикладная информатика

Профиль: Прикладная информатика в экономике

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**Разработка клиент-серверного фреймворка для высоконагруженных вычислений на базе Linux и MPI**

**Работа завершена:**

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2017 г.

Студент группы 11-308 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_И.А. Габидуллин

**Работа допущена к защите:**

Научный руководитель

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2017 г. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.С. Тощев

Директор Высшей школы ИТИС

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2017 г. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.Ф. Хасьянов

Казань – 2017 г.

Содержание

[Введение 3](#_Toc485151071)

[1 Постановка задачи 5](#_Toc485151072)

[2 Обзор существующих технологий и решений 7](#_Toc485151073)

[2.1 Принципы параллельных вычислений 7](#_Toc485151074)

[2.2 Технологии параллельных вычислений 8](#_Toc485151075)

[2.3 Обзор средств управления параллельными вычислениями 10](#_Toc485151076)

[3 Проектирование и разработка фреймворка 12](#_Toc485151077)

[3.1 Технологии разработки 12](#_Toc485151078)

[3.1.1 Язык программирования 12](#_Toc485151079)

[3.1.2 Средства разработки 13](#_Toc485151080)

[3.1.3 Средства проектирования 14](#_Toc485151081)

[3.1.4 Разработка серверной части 15](#_Toc485151082)

[3.1.5 Разработка клиентской части 17](#_Toc485151083)

[3.2 Проектирование фреймворка 17](#_Toc485151084)

[3.3 Разработка серверной части 20](#_Toc485151085)

[3.3.1 Разработка сервиса параллельных задач 21](#_Toc485151086)

[3.3.2 Разработка сервера управления 24](#_Toc485151087)

[3.4 Разработка клиентской части 26](#_Toc485151088)

[3.5 Взаимодействие клиента с сервером 28](#_Toc485151089)

[4 Экспериментальные исследования 31](#_Toc485151090)

[4.1 Разработка параллельной MPI программы 31](#_Toc485151091)

[4.2 Работа фреймворка на разных операционных системах 31](#_Toc485151092)

[Заключение 33](#_Toc485151093)

[Список использованных источников 35](#_Toc485151094)

[Приложение 37](#_Toc485151095)

# Введение

В мире существует множество задач, которые можно разделить на мелкие подзадачи и решать их параллельно. Особенно часто это встречается в науке. Разнообразные математические преобразования и вычисления. В настоящее время даже на мобильных устройствах по несколько процессоров, и было бы глупо не использовать возможности параллельных вычислений.

Параллельные вычисления уже давно стали важной частью Computer Science и активно используются на практике. Во многих научных организациях, например, в Казанском Федеральном Университете существует кластер электронно-вычислительных машин для выполнения параллельных вычислений на базе технологии MPI. [[[1]](#footnote-1)] Но не существует простого, удобного и недорогого способа управлять вычислениями: разворачивать программный код, запускать, отслеживать, получать результаты или данные о скорости работы. Либо система является сложной, либо очень дорогой, либо же всё это выполняется вручную прямо на кластере, пользователь подключается к кластеру, загружает нужные ему файлы и запускает процесс.

Исходя из вышеизложенного, было принято решение создать клиент-серверный фреймворк (программная платформа) для работы с параллельными вычислениями. Данный фреймворк должен позволять гибко настраивать взаимодействие с сервером параллельных вычислений.

Единожды развернув и настроив фреймворк, пользователь получает удобный способ управления своими вычислениями. Пользователь получает возможность организовать работу извне, не подключаясь напрямую к высокопроизводительному кластеру.

Так как фреймворк подразумевает дальнейшее использование и развитие другими разработчиками, то одной из задач является разработать проект таким образом, чтобы его структура была понятна без больших усилий.

Даная работа позволит улучшить результативность работы разных научных групп, которые используют в большом количестве параллельные вычисления. Таким образом, данная работа несет в себе практическую пользу в научной сфере.

Итогом данной дипломной работы является готовый к использованию клиент-серверный фреймворк для управления параллельными MPI задачами, функционирующий на базе операционной среды Linux.

# Постановка задачи

Основной целью данной дипломной работы является создание гибкого для изменений клиент-серверного фреймворка для управления высокопроизводительными вычислениями.

Рассмотрим базовые возможности, которые должен поддерживать фреймворка. Фреймворк должен поддерживать такие вещи как:

1. загрузка программы на сервер;
2. выполнение параллельной программы;
3. возможность управлять ходом работы программы;
4. возможность легко дополняться и изменяться.

Для достижения цели дипломной работы необходимо выполнить следующие задачи:

1. изучить парадигмы и технологии параллельных вычислений;
2. исследовать существующие системы управления параллельными вычислениями;
3. спроектировать архитектуру фреймворка, клиентскую и серверную части;
4. спроектировать архитектуру взаимодействия клиентской и серверной части;
5. разработать серверную часть;
6. разработать клиентскую часть;
7. исследовать работу фреймворка на разных операционных системах.

Теперь же рассмотрим в подробности функции, которые должен выполнять фреймворк в базовой версии:

1. создать задачу для выполнения параллельных вычислений, при этом может загружаться программа и данные из клиента на сервер или же программа и данные будут сразу находиться на сервере;
2. начать выполнение задачи с указанным количеством потоков;
3. остановить запущенную ранее задачу;
4. получить статус задачи, выполняется она или нет;
5. удалить задачу с сервера, то есть удаление параллельной программы, а также всех её данных.

Возможные дополнительные функции для дальнейшего развития проекта:

1. получить результаты работы параллельной программы;
2. получить статистику (загруженность системы, время выполнения) выполнения параллельной программы;
3. загрузить задачу через более быстрый протокол передачи данных.

Базовый функционал будет покрывать большинство потребностей в управлении задачами параллельных вычислений, а дополнительные функции в дальнейшем лишь улучшат условия использования фреймворка. Также фреймворк не обязательно использовать лишь с параллельными задачами, можно запускать обычные трудоёмкие задачи на отдельном сервере, произведя лишь небольшую настройку.

# Обзор существующих технологий и решений

## Принципы параллельных вычислений

В 1966 году профессором Стэндфордского университета Майклом Финном была предложена, так называемая «Таксономия Флинна» (Flynn's taxonomy). В ней электронно-вычислительные машины были разделены по принципу параллелизма в потоках команд и потоках данных. Рассмотрим её:

* ОКОД — система, которая работает с одиночным потоком команд и одиночным потоком данных (Single Instruction stream over a Single Data stream);
* ОКМД — система, которая работает с одиночным потоком команд и множественным потоком данных (Single Instruction, Multiple Data);
* МКОД — система, которая работает со множественным потоком команд и одиночным потоком данных (Multiple Instruction Single Data);
* МКМД — система, которая работает со множественным потоком команд и множественным потоком данных (Multiple Instruction Multiple Data). [[[2]](#footnote-2)]

Архитектура ОКОД — это традиционный компьютер [фон-Неймановской архитектуры](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D1%84%D0%BE%D0%BD_%D0%9D%D0%B5%D0%B9%D0%BC%D0%B0%D0%BD%D0%B0) с одним процессором, который выполняет последовательно одну инструкцию за другой, работая с одним потоком данных. ОКОД не является параллельной архитектурой. МКОД же является очень редко используемой архитектурой. На практике чаще всего используются такие архитектуры как ОКМД и МКМД.

Также ОКМД и МКМД разделяются на подклассы:

* системы с общей памятью (shared memory, SM);
* системы с распределенной памятью (distributed memory, DM).

Таким образом, фреймворк должен поддерживать архитектуры ОКМД и МКМД. Или же иметь возможность добавить поддержку новой архитектуры, не изменяя общей структуры проекта.

## Технологии параллельных вычислений

Существуют несколько крупных и используемых технологий для параллельных вычислений. Рассмотрим некоторые из них:

* MPI (Message Passing Interface) — программный интерфейс для передачи [информации](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D1%8F). Реализация данной технология позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Существуют реализации для языков Fortran, Java, C и C++;
* OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках C, C++ и Fortran. Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью; [[[3]](#footnote-3)]
* CUDA (Compute Unified Device Architecture) — программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia. Для разработки используется диалект языка программирования С.

Каждая из данных технологий направлена на решение своих специфических задач.

MPI технология нацелена на использование в системах с распределенной памятью. В тех случаях, когда затраты на передачу данных велики. Процессы общаются между собой при помощи передачи сообщений друг другу. Таким образом, они могут обмениваться результатами, либо данными между собой, а также отправлять какие-либо команды для выполнения на другом процессоре.

OpenMP технология используется же для систем, которые имеют общую память. Реализация данной технологии имеет несколько потоков (thread), один из них главный (master), остальные подчиненные (slave). Главный поток создает задачи, которые подчиненные выполняют. Программирование осуществляется при помощи специальных директив.

CUDA использует мощь графических процессоров для произведения разнообразных вычислений. Данная технология является самой молодой. Архитектура CUDA даёт разработчику возможность управлять инструкциями и памятью графического ускорителя.

Самой распространенной технологий из указанных трёх является MPI, так как она позволяет наиболее гибко распределять задачи между процессорами и потоками, также при помощи неё можно использовать систему с распределённой памятью, что очень важно в кластерах. Но оставшиеся две технологии также являются сильными технологиями для своих задач, и не стоит забывать о них.

## Обзор средств управления параллельными вычислениями

Есть несколько проектов, которые позволяют управлять параллельными вычислениями. Большинство из них являются коммерческими проектами. Так как разрабатываемый проект не является коммерческим, рассмотрим некоторые некоммерческие программы.

PBS (Portable Batch System) — система управления [распределенными вычислениями](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F). Основная функция PBS — запуск вычислительных задач в вычислительной среде по [расписанию](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D0%B8%D1%81%D0%B0%D0%BD%D0%B8%D0%B5). Наиболее часто используется для управления вычислительным процессом в [кластерах](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%82%D0%B5%D1%80_(%D0%B3%D1%80%D1%83%D0%BF%D0%BF%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BE%D0%B2)). Является одной из самых старых систем. На основе неё создано более современные системы управления, имеющие большее количество возможностей.

TORQUE (Terascale Open-Source Resource and QUEue Manager) — менеджер [распределенных ресурсов](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) для вычислительных [кластеров](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%82%D0%B5%D1%80_(%D0%B3%D1%80%D1%83%D0%BF%D0%BF%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BE%D0%B2)) из [машин](https://ru.wikipedia.org/wiki/%D0%A5%D0%BE%D1%81%D1%82) под управлением [Linux](https://ru.wikipedia.org/wiki/Linux) и других [Unix-подобных](https://ru.wikipedia.org/wiki/UNIX-%D0%BF%D0%BE%D0%B4%D0%BE%D0%B1%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) [операционных систем](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). Является современной версией PBS. Одна из самых развитых систем управления распределенными вычислениями, поддерживается большим организаций и научных лабораторий. [[[4]](#footnote-4)]

В вычислительном кластере на базе Казанского (Приволжского) Федерального университета для управления вычислениями используется такая система как IBM Platform LSF. Данная система является мощной платформой управления нагрузок для распределенных сред высокопроизводительных вычислений. Платформа поддерживает огромное количество функций управления параллельными вычислениями. Но имеет несколько минусов, являясь коммерческой и зарубежной разработкой.

Данные проекты развивались долгое время и имеют большое количество возможностей. Но также они являются сложными в установке и настройке системами.

Разрабатываемый фреймворк же должен быть лёгок как в настройке, так и установке, иметь возможность легко подвергаться модификациям. А также не исключать возможности совместного использования вместе с указанными системами.

# Проектирование и разработка фреймворка

## Технологии разработки

Важной частью в разработке какого-либо проекта является выбор инструментов и технологий, которые будут использоваться.

Выбирать следует исходя из целей и задач, так как неверно выбранная технологическая база в дальнейшем может помешать разработке. Или же в худшем случае придется начать работу сначала, используя уже подходящие технологии.

В сфере разработки программного обеспечения следует выделять следующие аспекты, в которых нужно выбирать те или иные инструменты:

1. язык программирования и программная платформа (фреймворк);
2. средства разработки;
3. средства проектирования.

В данной главе будет рассмотрен выбор тех или иных инструментов и технологий для каждого аспекта разработки и проектирования, которые будут использоваться на данном проекте.

### Язык программирования

В качестве языка разработки был выбран C# 7-ой версии. Данный язык программирования подходит для разработки разнообразных видов приложений: приложений для мобильной платформы, веб приложений, бизнес приложений, а также высоконагруженных систем. По этой причине данный язык как нельзя подходит для данного проекта, так как в нём сочетается разные виды приложений. Используя данный язык можно написать, как клиентскую часть, так и серверную.

Также данный язык выбран из-за того, что он достаточно популярный и распространенный. Разработчиков, которые знакомы с ним и работают, используя указанный язык, очень большое количество и найти таких специалистов не составляет труда. По индексу TIOBE (индекс, оценивающий популярность языков программирования) язык программирования C# на май 2017 года занимает 5 место по популярности. [[[5]](#footnote-5)]

В каждой новой версии C# добавляются новые возможности для разработчика и становится всё больше способов сделать программный код более структурированным и читабельным. Например, в последних версиях языка добавились такие возможности как интерполяция строк и локальные функции. [[[6]](#footnote-6)]

Язык C# поддерживается несколькими фреймворками, в дальнейшем будет описано какие именно фреймворки были выбраны для клиентской и серверной части проекта.

### Средства разработки

Для того, чтобы воспользоваться самыми новыми возможностями языка программирования, следует воспользоваться самой новой средой разработки. В данном случае выбор пал на Visual Studio 2017 Community.

Данная среда позволяет быстро и качественно писать код, так как в ней существует большое количество подсказок, которые могут помочь разработчику быстрее и качественнее создавать программный код. В ней удобно следить за исполнением кода. Visual Studio предоставляет множество данных о производительности создаваемой программы, что очень важно в разработке. Также данная среда поддерживает большое количество языков программирования, а значит программисту не надо осваивать большое количество сред разработки, а достаточно лишь изучить одну.

При помощи Visual Studio можно легко собирать проекты, а также разворачивать, либо настраивать развёртывание при необходимости.

Для тестирования веб-сервера использовалось расширение для веб-браузера Opera – RestMan. Плагин легко устанавливается из магазина расширений веб-браузера. А благодаря интуитивному и удобному интерфейсу, расширением достаточно легко научиться пользоваться.

Данное расширение позволяет отправлять REST веб запросы любого вида к указанному веб серверу и получать ответы. Ответы же отображаются в удобном для анализа виде.

### Средства проектирования

Проектирование – это очень важная часть разработки. Надо иметь способ легко создавать нужные для проекта диаграммы. Средой проектирования была выбрана компьютерную программу Visual Paradigm Community Edition.

Данная программа является бесплатной для некоммерческого использования. Программа поддерживает все виды UML диаграмм. Программа позволяет сохранять диаграммы в одном проекте, в следствии чего очень удобно создавать большое количество диаграмм для одного проекта.

К недостаткам можно отнести не совсем очевидный пользовательский интерфейс.

### Разработка серверной части

Так как языком разработки был выбран C#, то выбор фреймворка ограничивается разными видами .NET. В свою очередь .NET Framework имеет множество версий, между которыми можно выбирать:

1. .NET Framework – классический Windows фреймворк. Является старейшей платформой среди .NET платформ. Поддерживает большое количество типов приложений (Windows приложение, веб-приложение, мобильное приложение). Также платформа имеет большие возможности взаимодействия с системой. Но данная платформа поддерживает лишь семейство операционных систем Microsoft Windows. Поэтому описанная программная платформа не подходит для данного проекта;
2. Mono – некоммерческий кроссплатформенный (Linux, MacOS, Windows) фреймворк. Является перенесенной на другие операционные системы независимыми разработчикам .NET Framework. Но с некоторыми недоработками;
3. .NET Core – является разработанной компанией Microsoft кроссплатформенной программной платформой. Является самой молодой из данных проектов. В данной платформе используются наработки Mono. Данная платформа активно развивается и добавляет всё большее количество функций из классической .NET Framework, но для большего количества платформ. [[[7]](#footnote-7)]

Таким образом, стоит выбор между .NET Core и Mono. Все зависит от проекта. Вот какие факторы нужно учесть при выборе:

1. .NET Core занимает гораздо меньше места по сравнению Mono;
2. .NET Core на данный момент содержит меньше функциональности, чем Mono. Но в перспективе будет более развитым чем Mono. Для разрабатываемого проекта функциональности .NET Core более чем достаточно;
3. компилятор .NET Core более современный и поддерживает все новые возможности. Здесь уже Mono является менее развитой технологией;
4. поддержка Microsoft. .NET Core является разработкой Microsoft, а значит развиваться данная платформа будет быстрее и качественнее чем Mono. А также, учитывая, что Mono является конкурентом .NET Core, есть вероятность что платформа Mono в скором времени исчезнет совсем.

Исходя из этого, логично будет ориентироваться на .NET Core. Mono подойдет для случая, когда уже сейчас необходимы возможности, которых еще нет в .NET Core или разработка велась уже некоторое время для данной платформы.

Таким образом, имея новый современный проект, будет выбран .NET Core. Данная платформа поддерживает все нужные для проекта возможности, позволяет создать кроссплатформенный код. Также с данной платформой легче начать работу, так как Visual Studio 2017 сразу после установки поддерживает разработку для этой платформы.

### Разработка клиентской части

Предполагается то, что клиентскую часть можно создать, используя любую платформу, так как взаимодействие с серверной частью происходит через протокол HTTP (Web API).

В данной работе будет создано консольное приложение, которое взаимодействует с сервером по протоколу HTTP. Разработка будет происходить на платформе .NET Core для кроссплатформенности.

Предполагается, что, пользователь данного проекта может с лёгкостью добавить свои новые функции или создать абсолютно новый клиент. Например, сайт, который будет выполнять роль клиента. Благодаря выбранному протоколу взаимодействия и открытому исходному коду серверной части, такой клиент можно легко создать.

## Проектирование фреймворка

В основу архитектуры проекта положена модель слоёв, описанная Мартином Фаулером в книге «Архитектура корпоративных программных приложений». [[[8]](#footnote-8)] Архитектура заключается в том, что проект разбивается на несколько независимых слоёв (layer). Таким образом достигается гибкая для модернизации и изменений структура, что в случае с фреймворком очень важно.

В проекте существует несколько слоёв:

* Engine Layer – движок – базовый слой, в котором проводятся все минимальные, простые манипуляции;
* Manager Layer – слой который управляет Engine Layer, а также предоставляет доступ к его вычислениям и результатам;
* Presentation Layer – слой который отвечает за отображение данных в любом виде, будь то консольное или веб приложение.

Рассмотрим Presentation Layer. Presentation Layer представляет из себя консольное кроссплатформенное приложения которое общается с сервером посредством HTTP запросов. Консольное приложение генерирует специфический HTTP запросы по указанному адресу сервера.

Сервер представляет из себя Manager Layer. Его можно назвать Task Manger, то есть он отвечает за создание и управление задачами. Сам Task Manager общается с другими существующими менеджерами:

* Process Manager – взаимодействует с запущенными задачами, может их останавливать и получать статус задачи;
* Parallel Manager – запускает параллельные задачи (параллельную программу) с нужным количеством потоков.

По аналогии с менеджерами существует два движка, Process Engine и Parallel Engine, они выполняют те же функции что и менеджеры, но на более низком уровне, менеджеры управляют ими.

Presentation Layer взаимодействует с Task Manager через протокол HTTP, в то время как Task Manager управляет двумя другими менеджерами напрямую через их интерфейсы.

Разбивку на слои можно отобразить при помощи диаграммы. На рисунке 1 можно наглядно увидеть каким именно образом и на какие слои разбивается проект.

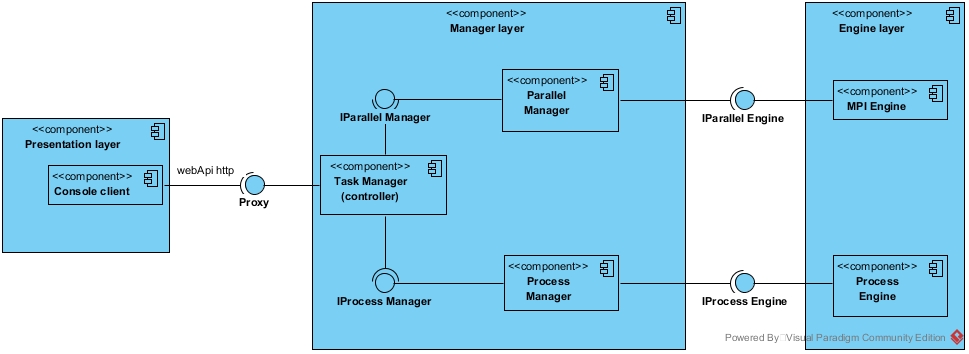


Рисунок 1. Структура слоёв фреймворка

На рисунке 2 можно изучить структуру решения фреймворка, на какие именно под проекты решение разбивается и как происходит взаимодействие между ними.

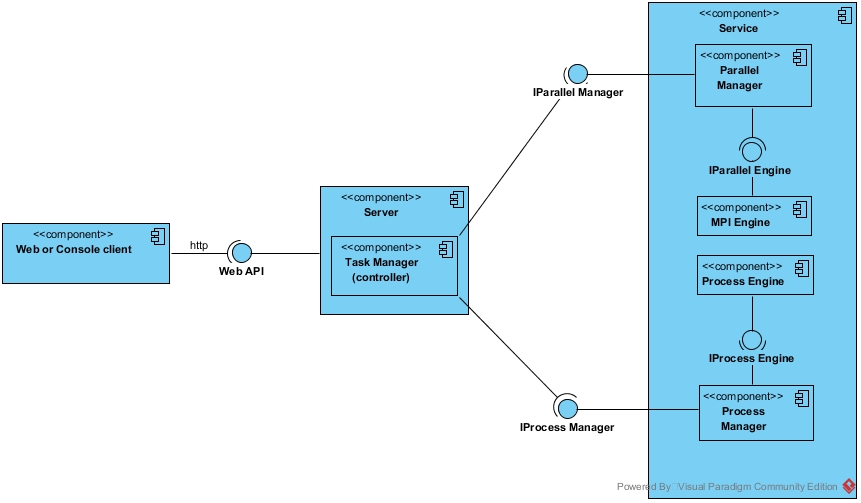


Рисунок 2. Структура решения фреймворка

Данная диаграмма будет полезна разработчикам, которым нужно внести какие-либо изменения в проект, так как на ней наглядно отображено где и какие модули проекта располагаются.

Также по диаграмме с рисунка 2 можно изучить развёртывание системы. Сервер управления, сервис параллельных вычислений, и клиент могут как располагаться на одной машине, так и располагаться на разных машинах. По задумке сервер управления и сервис параллельных вычислений располагаются на одной машине, но при желании их можно разнести. Также, если есть необходимость иметь несколько сервисов параллельных вычислений можно модифицировать сервер управления и создать несколько сервисов на нескольких машинах.

Таким образом, получается структура, в которой легко можно разобраться и добавить нужный функционал. Например, движки создаются через программный интерфейс и для того, чтобы подменить вызов MPI задач на вызов OpenMP задач, надо всего лишь реализовать интерфейс IParallel Engine для OpenMP задач и вызывать данную реализацию в реализации Parallel manager.

Из-за универсальности Web API легко можно создать новые клиенты для сервера. Например, веб сайт, который облегчит управление сервером, а также позволит детально и наглядно изучать данные, которые будут возвращаться сервером. Причем клиент можно создать на любой технологии, главное, чтобы была возможность генерировать HTTP запросы.

## Разработка серверной части

Выше были рассмотрены архитектура и интерфейсы взаимодействия проектов, теперь же рассмотрим реализацию, изучим исходный код проекта, а также глубже изучим программные интерфейсы фреймворка.

При разработке использовались всевозможные курсы и пособия, созданные компанией Microsoft для разработчиков. [[[9]](#footnote-9)]

### Разработка сервиса параллельных задач

Данный сервис отвечает за взаимодействие с параллельными задачами. Сервис включает в себя два менеджера Process Manager и Parallel Manager. Менеджер представляет из себя программный интерфейс и его реализацию.

Рассмотрим Process Manager и его интерфейс. Интерфейс IProcess Manager имеет два метода:

* bool IsRunningProcess(List<string> files) – данный метод принимает на вход список файлов задач, а возвращает булевское значения, которое означает в процесс ли сейчас указанная задача или нет;
* void KillProcess(List<string> files) – данный метод завершает все процессы указанной задачи.

Process Manager, который наследуется от данного интерфейса реализует оба этих метода и внутри себя вызывают Process Engine, что можно увидеть на рисунке 3.

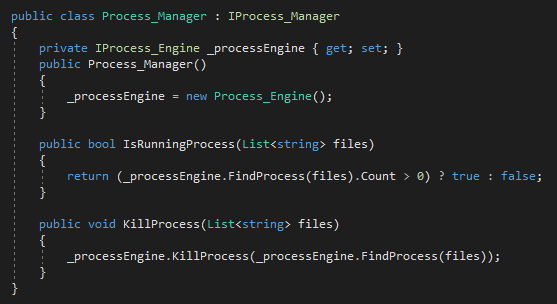


Рисунок 3. Класс Process Manager

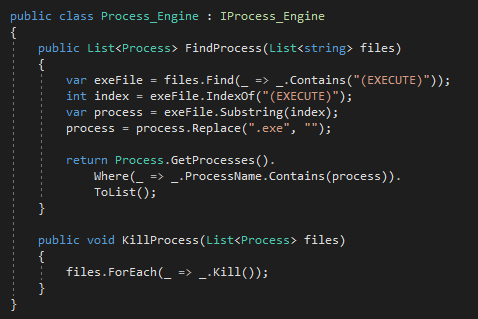
Рассмотрим класс Parallel Manager. В базовой реализации он включает в себя лишь один метод – void StartParallelProcess(string file, int threadCount). Из названия метода ясно, что данный метод запускает параллельную задачу, с указанным количеством потоков. Данный менеджер вызывает в себе реализацию IParallel Engine.

Теперь рассмотрим Engine Layer, который включает в себя IParallel Engine, IProcess Engine и их реализации.

IProcess Engine включает в себя два метода:

* List<Process> FindProcess(List<string> files) – находит рабочие процессы для определенной задачи;
* void KillProcess(List<Process> files) – завершает указанные процессы для задачи;

Реализации данных методов в классе Parallel Engine можно рассмотреть на рисунке 4.

Рисунок 4. Класс Process Engine

Для разделения исполняемой программы от данных идущих с ней, в название файла с программной добавляется тэг «(EXECUTE)», при помощи этого тэга Process Engine находит по названию файла запущенные экземпляры.

Реализация IParallel Engine называется MPI Engine, так как запускает MPI параллельные программы. Включает в себя лишь один публичный метод – Process StartParallelProcess(string file, int threadCount). Данный метод запускает MPI процесс по определенному принципу в зависимости от того на какой системе (Linux или Windows) размещен сервис. Если же пользователю нужен будет не MPI процесс, то как раз класс MPI Engine можно будет поменять на реализацию для определенной другой технологии, например, для OpenMP или же CUDA.

С данным сервисом будет взаимодействовать сервер управления, который будет рассмотрен дальше.

### Разработка сервера управления

Сервер управления, как уже говорилось выше, является ASP NET Core Web API приложением. Благодаря Visual Studio весь код нужный для создания сервера уже есть, остается лишь создать нужный контроллер.

Таким контроллером будет являться Task Controller. Это главный управляющий элемент фреймворка. Task Controller имеет функции:

* public async Task<bool> CreateTask(string taskName, IFormFileCollection files) – создает параллельную задачу с указанным названием, также через этот метод загружаются файлы для это задачи;
* public bool StartTask(string taskName, int threadCount) – запускает указанную задачу;
* public bool StopTask(string taskName) – останавливает указанную задачу;
* public bool GetTaskStatus(string taskName) – возвращает статус задачи, True, если задача выполняется, False, если задача не выполняется.

Внутри себя данные методы вызывают нужные для задачи менеджеры. Для каждой задачи создается своя папка с исполняемой параллельной программой, а также данными, которые были загружены. Для работы с файлами создан класс File Helper, который может выполнять все нужные для проекта манипуляции с файлами и папками. Такие как создать папку, удалить папку, создать путь к файлу, получить все файлы по адресу. Данный класс можно рассмотреть на рисунке 5.

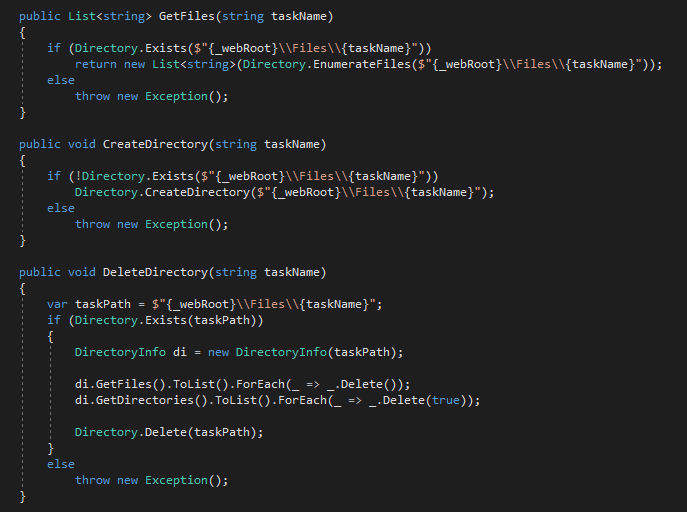


Рисунок 5. Часть класса File Helper

Данный класс позволяет сделать Task Controller более легким и читаемым и заниматься в нём лишь тем, что действительно относится к его задачам.

Также данный контроллер очень легко расширить для новых функций или переопределить ранее созданные функции, так как нет сложных взаимосвязей, из-за которых при изменении кода может появиться ошибка в абсолютно другом месте.

Если же пользователю фреймворка нужна будет функциональность вроде пользователей и их прав, то ему нужно будет создать новый контроллер, который будет выполнять данную функциональность (регистрация, вход), а в данном контроллере лишь проверять имеет ли пользователь использовать данный функционал или нет.

Таким образом, получился довольно простой в использовании сервер управления, который, во-первых, имеет весь базовый задуманный функционал, а во-вторых при необходимости может легко модифицироваться.

## Разработка клиентской части

Пользователь фреймворка может создать свой клиент на основе данных, которые указаны выше, а также полученных из изучения исходного кода фреймворка. В рамках данной работы было создано консольное приложение, которое позволяет взаимодействовать с сервером управления вычислениями.

С точки зрения разработки консольное приложение представляет из себя обычное консольное приложение, на базе .NET Core.

Для реализации меню программы был создан внутренний (internal) класс Menu, которое отображает доступные команды на экран пользователя. После выбора команды, запускается метод, который соотнесен с данной командой, это реализовано при помощи делегатов. Класс имеет делегат с названием CallDelegate и используется это таким образом:

private CallDelegate GetAssociatedDelegate(int numberOfRow)

{

switch (numberOfRow)

{

case 1:

return CreateTask;

case 2:

return StartTask;

case 3:

return StopTask;

case 4:

return GetStatus;

case 5:

return DeleteTask;

default:

return Exit;

}

}

GetAssociatedDelegate возвращает функцию в виде делагата и в вызывающем методе данная функция вызывается таким образом – GetAssociatedDelegate(Convert.ToInt32(key)).Invoke();

Для пользователя консольное приложение представляет из себя меню командной строки в котором по номеру команды выбирается команда и запускается процесс. Это можно увидеть на рисунках 6 и 7.

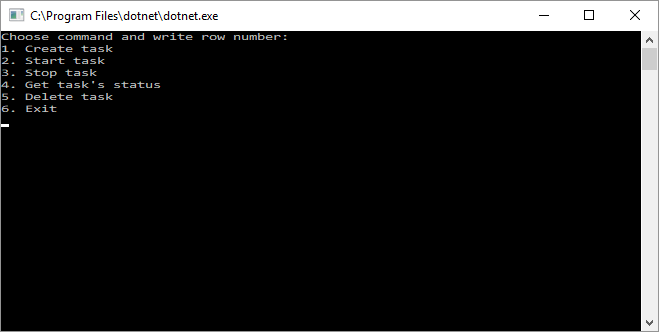


Рисунок 6. Главное меню приложения

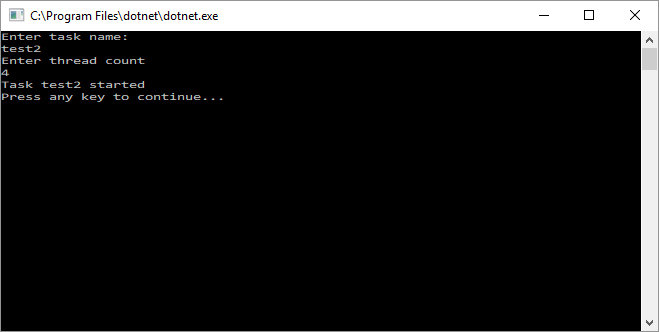


Рисунок 7. Меню запуска задачи

Данное консольное приложение не затрачивает большое количество ресурсов, не требует ничего, кроме установленного .NET Core, может вызывать любую команду на сервере, а также легко расширяться.

Для взаимодействия с сервером клиент использует класс Proxy, который рассмотрим в следующей главе.

## Взаимодействие клиента с сервером

Клиент взаимодействует с сервером по протоколу HTTP. Для реализации клиента в .NET существует класс HttpClient, который позволяет совершать любого вида запросу по указанному адресу. Помимо этого, данный может отправлять как синхронные, так и асинхронные запросы.

Для отделения логики взаимодействия от логики клиента был описан класс Proxy. Данный класс использует HttpClient для выполнения нужных для работы запросов.

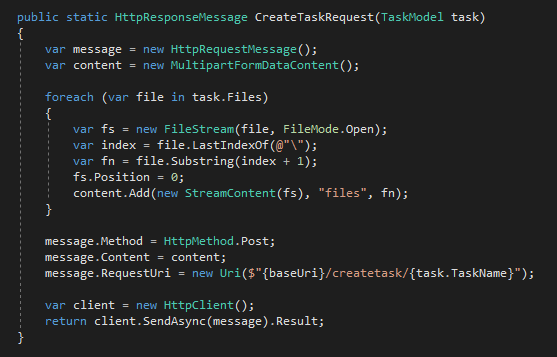
При помощи данного класса, а также классов HttpRequestMessage и MultipartFormDataContent можно гибко настраивать то, что отправится на сервер. Это используется в методе CreateTaskRequest на рисунке 8.

Рисунок 8. Описание метода Create Task Request

Можно увидеть каким образом генерируется запрос с файлами. Файлы считываются с жесткого диска, упаковываются в экземпляр класса MultipartFormDataContent. После этого эти данные помещаются в переменную message, которая имеет тип HttpRequestMessage. После чего готовое сообщение отправляется через HttpClient.

Остальные методы класса Proxy практически идентичны, меняются лишь адреса, по которому отправляются запросы, что видно на рисунке 9.

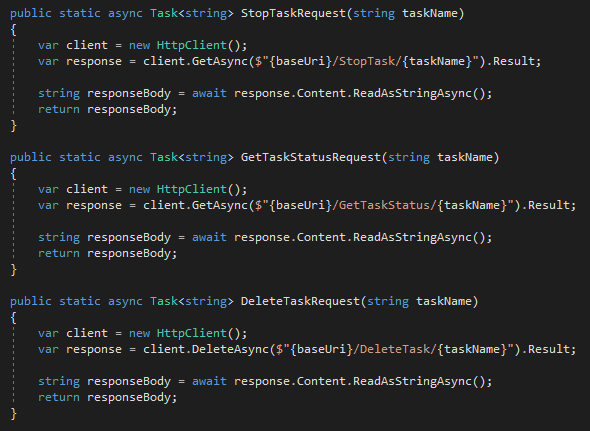


Рисунок 9. Оставшиеся методы класса Proxy

В некоторых методах, при удачно выполненном запросе, сервер отвечает булевской переменной. Класс Proxy расшифровывает ответ и возвращает клиенту ответ в виде строки, которая используется уже клиентом, для того, чтобы отобразить пользователю удачно прошла процедура или нет.

Данный класс можно использовать в любой реализации клиента на данном языке. Для клиентов на других языках программирования придется создавать свои реализации, копирующие данную функциональность.

# Экспериментальные исследования

## Разработка параллельной MPI программы

Для проверки разработанной системы было решено создать параллельную программу на базе технологии MPI.

Была разработана программа, которая перемножает матрицы. Шаги в данной математической операции (перемножение матрицы) не зависят друг от друга, поэтому данную задачу можно легко распределить на несколько независящих друг от друга потоков.

Данная программа была написана на языке программирования C++. Для параллельных функций была подключена библиотека MPI.

Данная программа принимает на вход два файла с матрицами, а выходными данными её работы является третий файл, с результатом перемножения считанных с файлов матриц.

## Работа фреймворка на разных операционных системах

Благодаря .NET Core программа, собранная один раз, одинаково запускается на разных платформах. Благодаря этому можно вести разработку на одной операционной системе и благополучно разворачивать запускать программу на другой операционной системе.

Процесс запуска .NET Core приложения хорошо описан разработчиками .NET Core. [[[10]](#footnote-10)] Следуя инструкциям получилось запустить работу фреймворка как на системе Windows, так и на системе Ubuntu, что можно увидеть на рисунке 10 и 11.

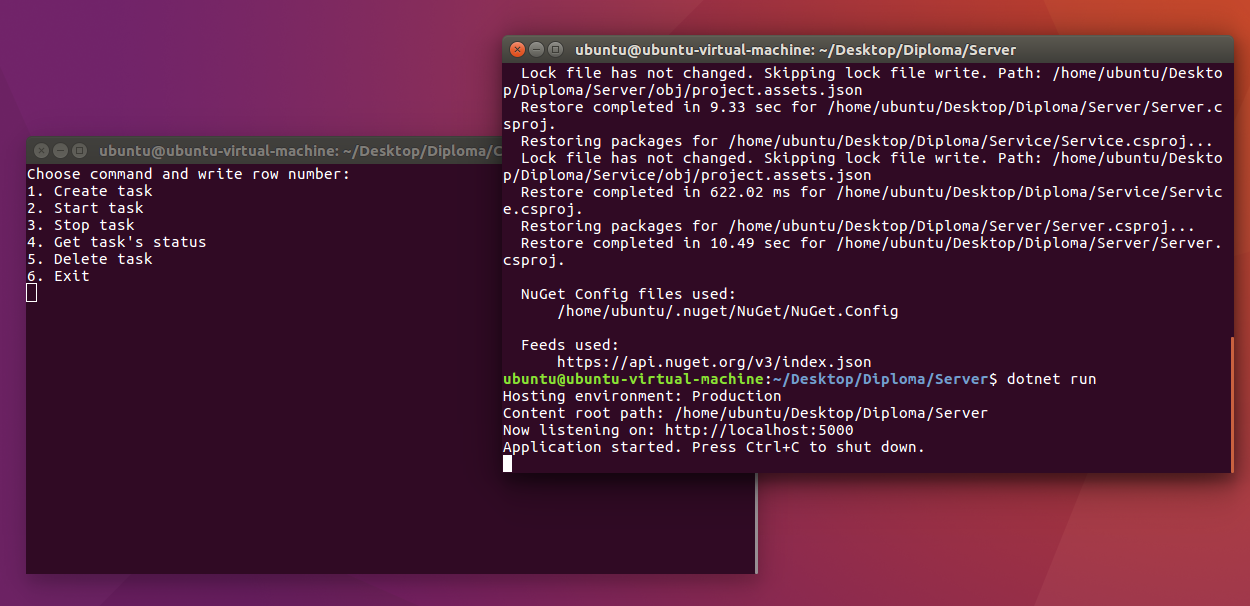


Рисунок 10. Фреймворк запущен и работает в операционной системе Ubuntu.

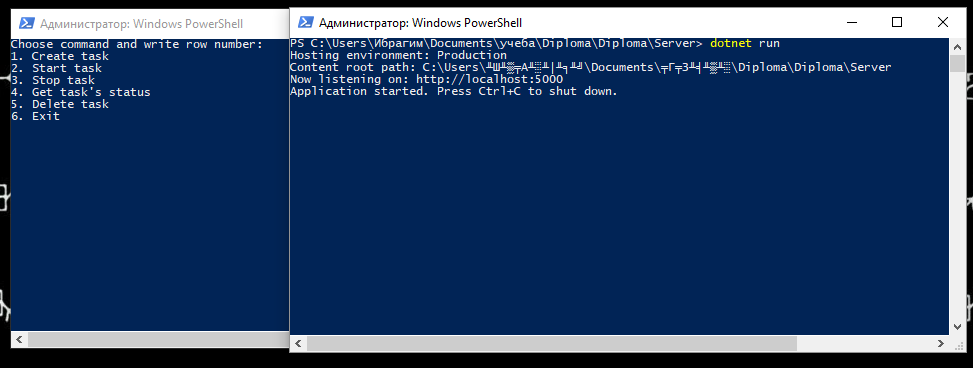


Рисунок 11. Фреймворк запущен и работает в операционной системе Windows 10.

После запуска программ на разных системах была проверена работа, разработанной ранее MPI программы. В фреймворке была создана задача с данной программой. После этого она была запущена. Программа благополучно отработала и вернула результаты своей работы в файл.

Таким образом, можно уверенно сказать, что фреймворк полностью готов к работе в реальной MPI системе.

# Заключение

По итогам работы был разработан кроссплатформенный фреймворк, который упрощает работу с высокопроизводительными параллельными вычислениями. Задач, связанных с параллельными вычислениями с каждым годом, будет становиться всё больше и больше, а значит инструменты, которые повышают удобство, то есть, ускоряют процесс работы с ними, будут очень важны в научной сфере.

Разработанная программная платформа готова к развёртыванию и применению. Помимо этого, фреймворк имеет большие возможности для доработки и добавления нового функционала, так как при создании закладывалось возможности для этого.

В ходе работы пригодились знания, полученные во время учебы в университете. Познания в проектирования программного обеспечения, навыки объектно-ориентированной разработки, знания веб-разработки помогли успешно разработать данный проект.

Во время разработки было получено большое количество новых знаний и умений, как в разработке, так и в проектировании программного обеспечения, самостоятельное проектирование всего приложения позволило узнать многое о том, как спроектировать понятный для разработчиков и гибко настраиваемый фреймворк.

Также были изучены как старые, зарекомендовавшие себя с хорошей стороны, так и самые новые, только развивающиеся, но уже популярные технологии. Были изучены такие технологии как MPI и .NET Core.

В дальнейшем планируется добавление нового функционала в систему. Получение статистики работы параллельной программы, получение результатов работы параллельной задачи, взаимодействие через протокол FTP, запуск вычислений по определенному расписанию. Помимо этого, планируется разработка клиента в виде веб приложения.

Во время выполнения данной работы были разработаны сервис параллельных задачам, сервер управления задачами, клиентское приложение, а также параллельная MPI программа для проверки работы фреймворка. Также данный проект был развернут на нескольких разных системах.

Таким образом, можно с уверенностью сказать, что все поставленные в данной работе задачи были выполнены, а цели достигнуты.

# Список использованных источников

1. Вычислительный кластер КФУ [электронный ресурс]. Режим доступа http://kpfu.ru/cluster/ свободный. Дата обращения 28.04.2017 [1]
2. Таксономия Флинна [электронный ресурс]. Режим доступа https://ru.wikipedia.org/wiki/Таксономия\_Флинна/ свободный. Дата обращения 03.05.2017 [2]
3. Основы параллельного программирования с использованием технологий MPI и OpenMP: учебное пособие / Жалнин Р.В., Панюшкина Е.Н., Пескова Е.Е., Шаманаев П.А. – Изд-во СВМО, 2013 – 78 с. [3]
4. TORQUE Resource Manager [электронный ресурс] Режим доступа http://www.adaptivecomputing.com/products/open-source/torque/ свободный. Дата обращения 04.05.2017 [4]
5. TIOBE Index [электронный ресурс] Режим доступа https://www.tiobe.com/tiobe-index/ свободный. Дата обращения 23.05.2017 [5]
6. Эндрю Троелсен, Язык программирования C# 6.0 и платформа .NET 4.6 7-е издание: книга для разработчиков С# / Эндрю Троелсен – Изд-во Вильямс, 2016. – 1440 с. [6]
7. Microsoft .NET [электронный ресурс]. Режим доступа https://www.microsoft.com/net/ свободный. Дата обращения 08.05.2017 [7]
8. Мартин Фаулер, Архитектура корпоративных программных приложений / Мартин Фаулер – Изд-во Вильямс, 2006. — 544 c. [8]
9. Microsoft Developer Network [электронный ресурс]. Режим доступа https://msdn.microsoft.com/ свободный. Дата обращения 10.05.2017 [9]
10. Getting Started with ASP.NET Core [электронный ресурс]. Режим доступа https://docs.microsoft.com/ru-ru/aspnet/core/getting-started/ свободный. Дата обращения 19.05.2017 [10]

# Приложение

Исходный код консольного клиента:

using Contracts;

using System;

using System.Collections.Generic;

using System.IO;

using System.Net.Http;

using System.Net.Http.Headers;

namespace ConsoleClient

{

class Program

{

static void Main(string[] args)

{

Menu menu = new Menu();

menu.StartMenu();

}

internal class Menu

{

bool \_exitFlag;

private Dictionary<int, string> \_menu;

private delegate void CallDelegate();

public Menu()

{

\_menu = new Dictionary<int, string>();

\_menu.Add(1, "Create task");

\_menu.Add(2, "Start task");

\_menu.Add(3, "Stop task");

\_menu.Add(4, "Get task's status");

\_menu.Add(5, "Delete task");

\_menu.Add(6, "Exit");

}

public void StartMenu()

{

Console.Clear();

\_exitFlag = false;

while (!\_exitFlag)

{

ShowMenu();

var key = Console.ReadLine();

Console.Clear();

try

{

GetAssociatedDelegate(

Convert.ToInt32(key)).Invoke();

}

catch (Exception exp)

{

Console.WriteLine(exp.Message);

Console.WriteLine(exp.StackTrace);

}

if (!\_exitFlag)

{

Pause();

Console.Clear();

}

}

}

public void ShowMenu()

{

Console.WriteLine("Choose command and write row number:");

foreach (var item in \_menu)

{

Console.WriteLine(item.Key + ". " + item.Value);

}

}

private void Pause()

{

Console.WriteLine("Press any key to continue...");

Console.ReadLine();

}

private CallDelegate GetAssociatedDelegate(int numberOfRow)

{

switch (numberOfRow)

{

case 1:

return CreateTask;

case 2:

return StartTask;

case 3:

return StopTask;

case 4:

return GetStatus;

case 5:

return DeleteTask;

default:

return Exit;

}

}

private void CreateTask()

{

Console.Clear();

try

{

var task = new TaskModel();

Console.WriteLine("Enter task name:");

task.TaskName = Console.ReadLine();

Console.WriteLine("Enter files absoulte paths splitted by ;");

var strFiles = Console.ReadLine();

var files = strFiles.Split(';');

foreach (var file in files)

{

task.Files.Add(file);

}

Console.WriteLine($"Task name: {task.TaskName}");

Console.WriteLine("Task files:");

task.Files.ForEach(\_ => Console.WriteLine(\_));

var result = Proxy.CreateTaskRequest(task);

if (result.StatusCode == System.Net.HttpStatusCode.OK)

Console.WriteLine("Task created!");

else

Console.WriteLine("Something wrong!");

}

catch (Exception exp)

{

Console.WriteLine(exp);

}

}

private void StartTask()

{

Console.Clear();

try

{

Console.WriteLine("Enter task name:");

var task = Console.ReadLine();

Console.WriteLine("Enter thread count");

var threadCount = Convert.ToInt32(Console.ReadLine());

Proxy.StartTaskRequest(task, threadCount);

Console.WriteLine($"Task {task} started");

}

catch (Exception exp)

{

Console.WriteLine(exp);

}

}

private void StopTask()

{

Console.Clear();

try

{

Console.WriteLine("Enter task name:");

var task = Console.ReadLine();

var result = Proxy.StopTaskRequest(task).Result;

if (result == "true")

Console.WriteLine($"Task {task} stopped!");

else

Console.WriteLine("Something is wrong!");

}

catch (Exception exp)

{

Console.WriteLine(exp);

}

}

private void GetStatus()

{

Console.Clear();

Console.WriteLine("Enter task name:");

var task = Console.ReadLine();

var result = Proxy.GetTaskStatusRequest(task).Result;

if (result == "true")

Console.WriteLine($"Task {task} is running");

else

Console.WriteLine($"Task {task} is not running");

try

{

}

catch (Exception exp)

{

Console.WriteLine(exp);

}

}

private void DeleteTask()

{

try

{

Console.Clear();

Console.WriteLine("Enter task name:");

var task = Console.ReadLine();

var result = Proxy.DeleteTaskRequest(task).Result;

if (result == "true")

Console.WriteLine($"Task {task} deleted");

else

Console.WriteLine($"Something is wrong");

}

catch (Exception exp)

{

Console.WriteLine(exp);

}

}

private void Exit()

{

\_exitFlag = true;

}

}

}

}

Интерфейсы:

using System;

using System.Diagnostics;

namespace Contracts

{

public interface IParallel\_Engine

{

Process StartParallelProcess(string file, int threadCount);

}

}

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Text;

namespace Contracts

{

public interface IProcess\_Engine

{

List<Process> FindProcess(List<string> files);

void KillProcess(List<Process> files);

}

}

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Text;

namespace Contracts

{

public interface IProcess\_Manager

{

bool IsRunningProcess(List<string> files);

void KillProcess(List<string> files);

}

}

using System;

using System.Threading.Tasks;

namespace Contracts

{

public interface IParallel\_Manager

{

void StartParallelProcess(string file, int threadCount);

}

}

Класс Proxy:

using System;

using System.Collections.Generic;

using System.IO;

using System.Net.Http;

using System.Text;

using System.Threading.Tasks;

namespace Contracts

{

public static class Proxy

{

static string baseUri = "http://localhost:2926/api/tasks";

public static HttpResponseMessage CreateTaskRequest(TaskModel task)

{

var message = new HttpRequestMessage();

var content = new MultipartFormDataContent();

foreach (var file in task.Files)

{

var fs = new FileStream(file, FileMode.Open);

var index = file.LastIndexOf(@"\");

var fn = file.Substring(index + 1);

fs.Position = 0;

content.Add(new StreamContent(fs), "files", fn);

}

message.Method = HttpMethod.Post;

message.Content = content;

message.RequestUri = new Uri($"{baseUri}/createtask/{task.TaskName}");

var client = new HttpClient();

return client.SendAsync(message).Result;

}

public static void StartTaskRequest(string taskName, int threadCount)

{

var client = new HttpClient();

client.GetAsync($"{baseUri}/StartTask/{taskName}/{threadCount}");

}

public static async Task<string> StopTaskRequest(string taskName)

{

var client = new HttpClient();

var response = client.GetAsync($"{baseUri}/StopTask/{taskName}").Result;

string responseBody = await response.Content.ReadAsStringAsync();

return responseBody;

}

public static async Task<string> GetTaskStatusRequest(string taskName)

{

var client = new HttpClient();

var response = client.GetAsync($"{baseUri}/GetTaskStatus/{taskName}").Result;

string responseBody = await response.Content.ReadAsStringAsync();

return responseBody;

}

public static async Task<string> DeleteTaskRequest(string taskName)

{

var client = new HttpClient();

var response = client.DeleteAsync($"{baseUri}/DeleteTask/{taskName}").Result;

string responseBody = await response.Content.ReadAsStringAsync();

return responseBody;

}

}

}

Классы менеджеры:

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

using System.IO;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Hosting;

using System;

using Contracts;

using Service;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using Server.Helpers;

namespace Server.Controllers

{

[Route("api/[controller]")]

public class TasksController : Controller

{

private IHostingEnvironment \_env;

private IParallel\_Manager \_mpiManager;

private IProcess\_Manager \_processManager;

private FileHelper \_fileHelper;

public TasksController(IHostingEnvironment env)

{

\_env = env;

\_mpiManager = new Parallel\_Manager();

\_processManager = new Process\_Manager();

\_fileHelper = new FileHelper(env.WebRootPath);

}

[HttpPost("CreateTask/{taskName}")]

public async Task<bool> CreateTask(string taskName, IFormFileCollection files)

{

try

{

var webRoot = \_env.WebRootPath;

\_fileHelper.CreateDirectory(taskName);

string file = "";

for (int i = 0; i < files.Count; i++)

{

if (i == 0)

file = \_fileHelper.CreateFile(taskName, "(EXECUTE){files[i].FileName}");

else

file = \_fileHelper.CreateFile(taskName, "{files[i].FileName}");

using (var fileStream = new FileStream(file, FileMode.Create))

{

await files[i].CopyToAsync(fileStream);

}

}

return true;

}

catch

{

return false;

}

}

[HttpGet("StartTask/{taskName}/{threadCount}")]

public void StartTask(string taskName, int threadCount)

{

var files = \_fileHelper.GetFiles(taskName);

var exeFile = files.Find(\_ => \_.Contains("(EXECUTE)"));

\_mpiManager.StartParallelProcess(exeFile, threadCount);

}

[HttpGet("StopTask/{taskName}")]

public bool StopTask(string taskName)

{

try

{

var files = \_fileHelper.GetFiles(taskName);

\_processManager.KillProcess(files);

return true;

}

catch

{

return false;

}

}

[HttpGet("GetTaskStatus/{taskName}")]

public bool GetTaskStatus(string taskName)

{

var files = \_fileHelper.GetFiles(taskName);

return \_processManager.IsRunningProcess(files);

}

[HttpDelete("DeleteTask/{taskName}")]

public bool DeleteTask(string taskName)

{

try

{

\_fileHelper.DeleteDirectory(taskName);

return true;

}

catch

{

return false;

}

}

}

}

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Runtime.InteropServices;

using Contracts;

using System.Linq;

namespace Service

{

public class Process\_Manager : IProcess\_Manager

{

private IProcess\_Engine \_processEngine { get; set; }

public Process\_Manager()

{

\_processEngine = new Process\_Engine();

}

public bool IsRunningProcess(List<string> files)

{

return (\_processEngine.FindProcess(files).Count > 0) ? true : false;

}

public void KillProcess(List<string> files)

{

\_processEngine.KillProcess(\_processEngine.FindProcess(files));

}

}

}

using Contracts;

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Text;

using System.Threading.Tasks;

namespace Service

{

public class Parallel\_Manager : IParallel\_Manager

{

private IParallel\_Engine \_MPIEngine { get; set; }

public Parallel\_Manager()

{

\_MPIEngine = new MPI\_Engine();

}

public void StartParallelProcess(string file, int threadCount)

{

Stopwatch stopWatch = new Stopwatch();

stopWatch.Start();

\_MPIEngine.StartParallelProcess(file, threadCount);

stopWatch.Stop();

TimeSpan ts = stopWatch.Elapsed;

}

}

}

Классы движки:

using System;

using System.Diagnostics;

using System.Runtime.InteropServices;

using Contracts;

namespace Service

{

public class MPI\_Engine : IParallel\_Engine

{

public Process StartParallelProcess(string file, int threadCount)

{

int index = file.LastIndexOf(@"\");

string dir = file.Substring(0, index + 1);

Process result = null;

if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))

result = ExecuteWindowsCommand($"cd {dir} & mpiexec -n {threadCount} {file}");

else if (RuntimeInformation.IsOSPlatform(OSPlatform.Linux))

result = ExecuteLinuxCommand($"cd {dir} ; mpiexec -n {threadCount} {file}");

return result;

}

private Process ExecuteLinuxCommand(string command)

{

ProcessStartInfo processInfo;

Process process;

processInfo = new ProcessStartInfo(command)

{

CreateNoWindow = true,

UseShellExecute = false

};

process = Process.Start(processInfo);

process.WaitForExit();

try

{

process.Kill();

return process;

}

catch

{

return null;

}

}

private Process ExecuteWindowsCommand(string command)

{

ProcessStartInfo processInfo;

Process process;

processInfo = new ProcessStartInfo("cmd.exe", "/c " + command)

{

CreateNoWindow = true,

UseShellExecute = false

};

process = Process.Start(processInfo);

process.WaitForExit();

try

{

process.Kill();

return process;

}

catch

{

return null;

}

}

}

}

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Runtime.InteropServices;

using Contracts;

using System.Linq;

namespace Service

{

public class Process\_Engine : IProcess\_Engine

{

public List<Process> FindProcess(List<string> files)

{

var exeFile = files.Find(\_ => \_.Contains("(EXECUTE)"));

int index = exeFile.IndexOf("(EXECUTE)");

var process = exeFile.Substring(index);

process = process.Replace(".exe", "");

return Process.GetProcesses().

Where(\_ => \_.ProcessName.Contains(process)).

ToList();

}

public void KillProcess(List<Process> files)

{

files.ForEach(\_ => \_.Kill());

}

}

}

1. Вычислительный кластер КФУ [электронный ресурс]. Режим доступа http://kpfu.ru/cluster/ свободный. [↑](#footnote-ref-1)
2. Таксономия Флинна [электронный ресурс]. Режим доступа https://ru.wikipedia.org/wiki/Таксономия\_Флинна/ свободный. [↑](#footnote-ref-2)
3. Основы параллельного программирования с использованием технологий MPI и OpenMP: учебное пособие [↑](#footnote-ref-3)
4. TORQUE Resource Manager [электронный ресурс] Режим доступа http://www.adaptivecomputing.com/products/open-source/torque/ свободный. [↑](#footnote-ref-4)
5. TIOBE Index [электронный ресурс] Режим доступа https://www.tiobe.com/tiobe-index/ свободный. [↑](#footnote-ref-5)
6. Эндрю Троелсен, Язык программирования C# 6.0 и платформа .NET 4.6 7-е издание: книга для разработчиков С# [↑](#footnote-ref-6)
7. Microsoft .NET [электронный ресурс]. Режим доступа https://www.microsoft.com/net/ свободный. [↑](#footnote-ref-7)
8. Мартин Фаулер, Архитектура корпоративных программных приложений [↑](#footnote-ref-8)
9. Microsoft Developer Network [электронный ресурс]. Режим доступа https://msdn.microsoft.com/ свободный. [↑](#footnote-ref-9)
10. Getting Started with ASP.NET Core [электронный ресурс]. Режим доступа https://docs.microsoft.com/ru-ru/aspnet/core/getting-started/ свободный. [↑](#footnote-ref-10)