#### СОДЕРЖАНИЕ

**Введение** ?

[**Лабораторная работа № 1**](#_ЛАБОРАТОРНАЯ_РАБОТА_№)Создание первой MPI-программы 5

[**Лабораторная работа № 2**](#_ЛАБОРАТОРНАЯ_РАБОТА_№_1) Основные языковые конструкции C#. Линейный

вычислительный процесс 20

**Лабораторная работа № 3** Управляющие операторы разветвляющегося

вычислительного процесса 37

**Приложения** 133

**Литература** 162

#### ВВЕДЕНИЕ

Лабораторный практикум …

Для выполнения лабораторных работ необходимо установить на ПК перечисленные ниже программы и зарегистрироваться на указанных ниже ресурсах. Процесс установки и регистрации описан в Приложении 1.

Программы для установки:

* Git (<https://git-scm.com> );
* Visual Studio Code (<https://code.visualstudio.com> );
* MobaXterm (<https://mobaxterm.mobatek.net/download.html> ).

Ресурсы для регистрации:

* GitHub (<https://github.com> ).

#### ЛАБОРАТОРНАЯ РАБОТА № 1

Тема: **создание первой MPI-программы**

**1 ЦЕЛЬ РАБОТЫ**

Познакомиться:

* с интерфейсом интегрированной среды разработки Ms Visual Studio 2010;
* с основными возможностями Ms Visual Studio 2010.

Научиться:

* создавать и сохранять проекты различных типов;
* выполнять отладку проектов.

**2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ.  
ТЕХНОЛОГИЯ ВЫПОЛНЕНИЯ РАБОТЫ**

**2.1 Создание репозитория на GitHub**

**Git** (произносится «гит») – распределённая система управления версиями файлов. Система спроектирована как набор программ, специально разработанных с учётом их использования в скриптах. Это позволяет удобно создавать специализированные системы контроля версий на базе Git или пользовательские интерфейсы.

Git поддерживает быстрое разделение и слияние версий, включает инструменты для визуализации и навигации по нелинейной истории разработки. Git предоставляет каждому разработчику локальную копию всей истории разработки; изменения копируются из одного репозитория в другой.

Удалённый доступ к репозиториям Git обеспечивается git-daemon, gitosis, SSH или HTTP-сервером. TCP-сервис git-daemon входит в дистрибутив Git и является наряду с SSH наиболее распространённым и надёжным методом доступа. Метод доступа по HTTP, несмотря на ряд ограничений, очень популярен в контролируемых сетях, потому что позволяет использовать существующие конфигурации сетевых фильтров.

***Упражнение 1***

В этом упражнении вы создадите удаленный репозиторий на сайте <https://github.com> и создадите его копию на ПК.

1. Зайдите в свой аккаунт на сайте <https://github.com>.
2. В правом верхнем углу находится иконка с фотографией пользователя и рядом раскрывающийся список с командами меню пользователя (рисунок 1.1).

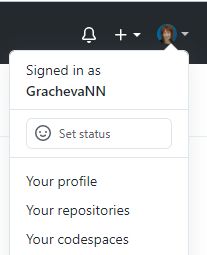


Рисунок 1.1 – Фрагмент меню пользователя

1. Щелкните ЛКМ по пункту Your repositories. На экране появится страница со списком репозиториев пользователя (у Вас этот список пустой).
2. Для создания нового репозитория щелкните по кнопке New  . На экране появится страница для создания нового репозитория (рисунок 1.2).

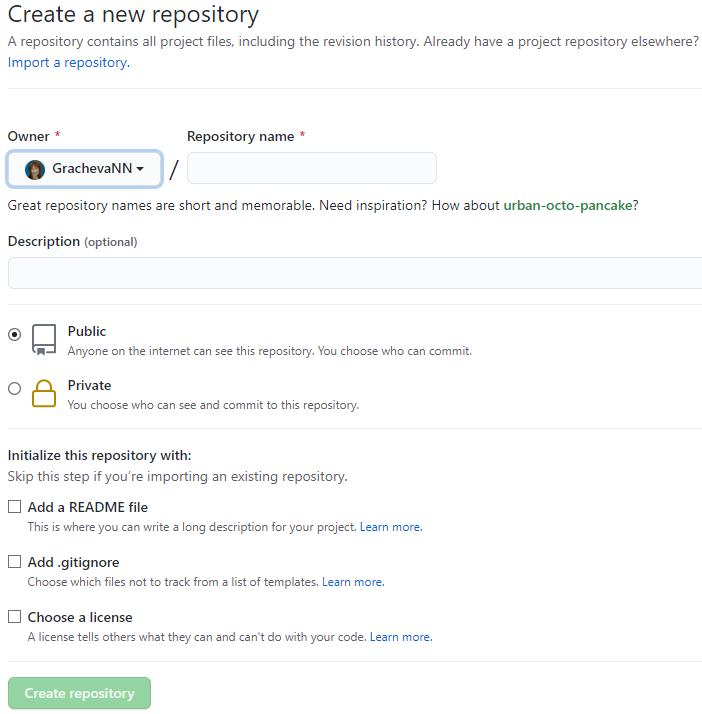


Рисунок 1.2 − Страница для создания нового репозитория

1. Заполните открытую страницу как показано на рисунке 1.3. Щелкните по кнопке Create repository.

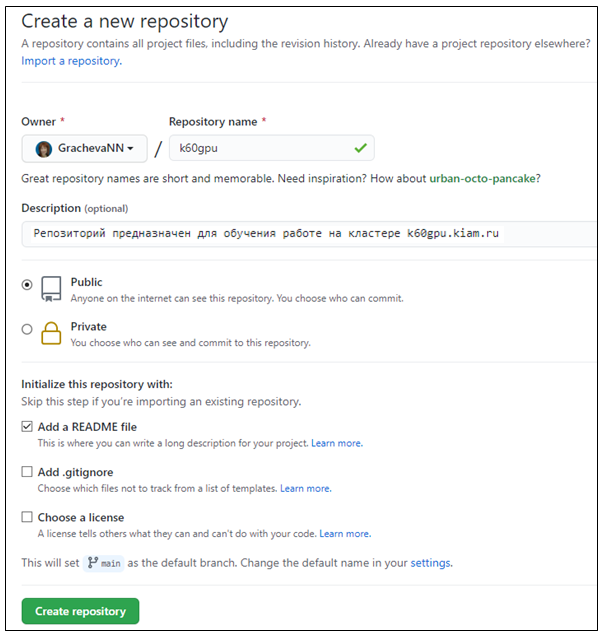


Рисунок 1.3 − Страница для создания нового репозитория с настройками

1. На экране появится страница с вновь созданным репозиторием (рисунок 1.4), будем называть *удаленным репозиторием*. На странице отображено название репозитория, файл README.md и меню для работы в репозитории. Файл README.md содержит описание проекта, указанное при создании, на языке разметки Markdown.

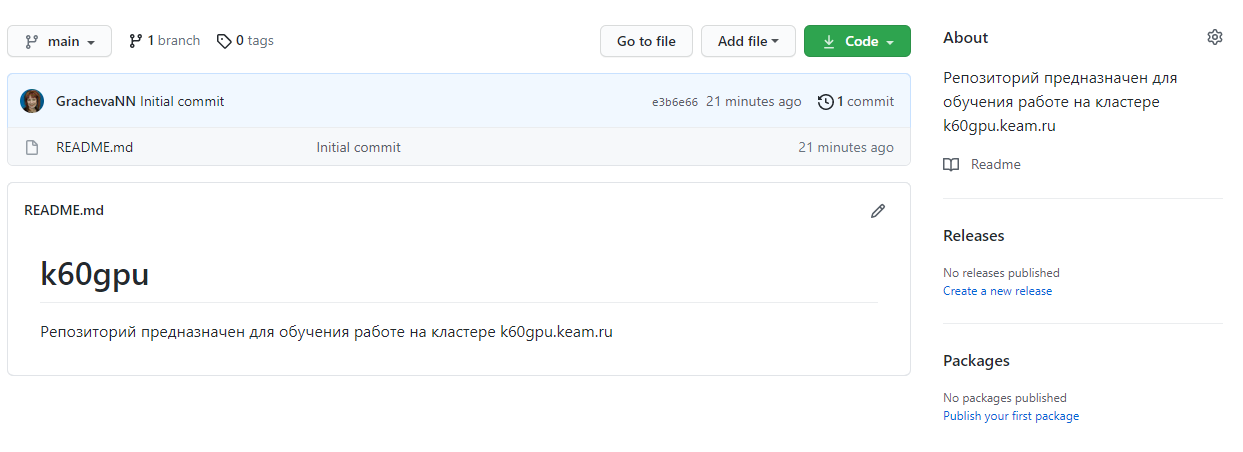


Рисунок 1.4 – Страница созданного репозитория

1. Выберите на локальном ПК место для клонирования репозитория. Щелкните ПКМ по пустому полю содержимого открытого диска (или папки) и в появившемся контекстном меню выберите команду Git Bash Here (рисунок 1.5). Запустится программа Git и на экране появится окно программы для ввода команд (рисунок 1.6).

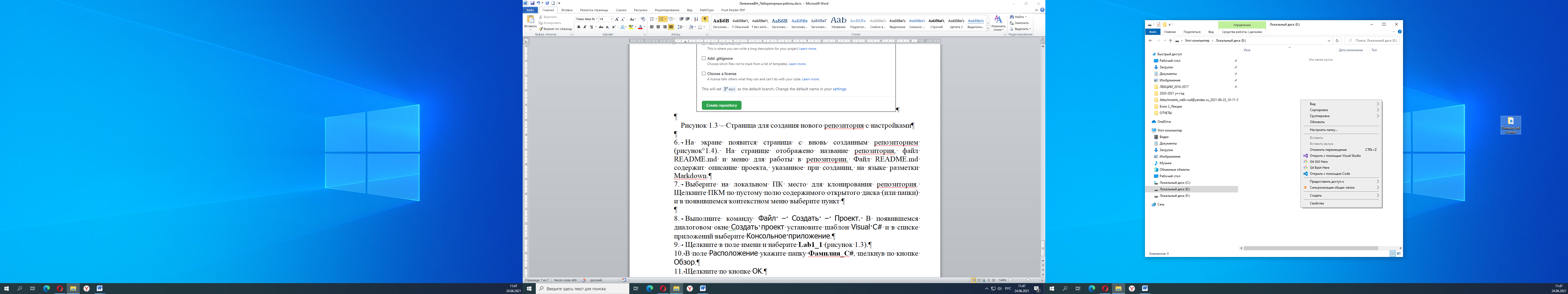


Рисунок 1.5 − Контекстное меню

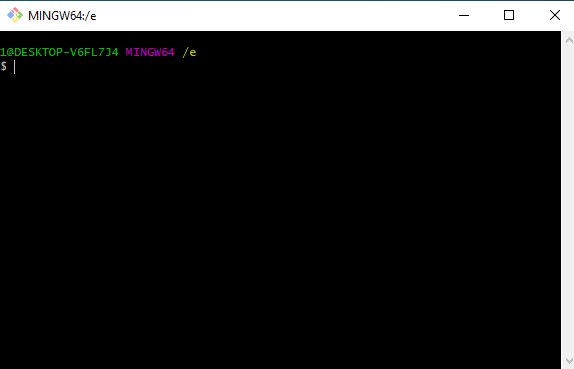


Рисунок 1.6 − Окно программы Git

*Вариант 1 (с использованием программы Git)*

1. Для копирования ссылки на репозиторий перейдите на сайте <https://github.com> к открытой Вами странице нового репозитория k60gpu и щелкните по кнопке  (см. рисунок 1.4). На экране появится меню, в котором необходимо скопировать ссылку для клонирования репозитория. Щелкните по кнопке копирования ссылки (рисунок 1.7). Ссылка скопирована в буфер обмена.

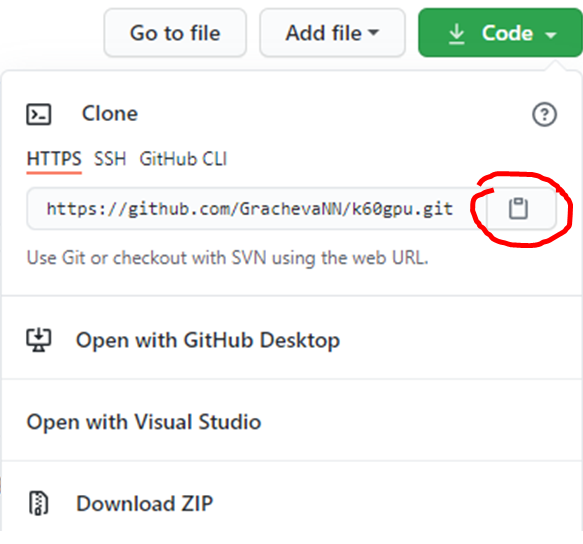


Рисунок 1.7 – Меню и кнопка для копирования ссылки

1. Вернитесь к открытому окну программы Git (см. рисунок 1.6). Наберите с клавиатуры команду git clone, нажмите клавишу **Пробел** и затем в этом же окне после набранной команды щелкните ПКМ и выберите в контекстном меню команду Paste. В результате в окне программы Git появится команда (рисунок 1.8). Учтите, что в Вашей команде имя пользователя будет другое. Нажмите клавишу **Enter**. Сравните результат работы команды с рисунком 1.9.

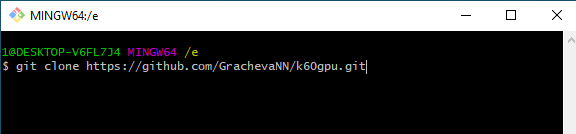


Рисунок 1.8 – Меню для копирования ссылки

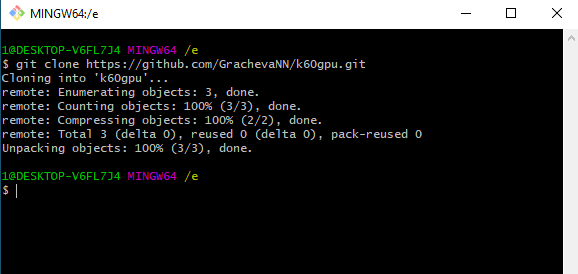


Рисунок 1.9 – Результат работы команды

1. Закройте окно программы Git. Перейдите в выбранную Вами папку (диск) и убедитесь, что в ней появилась папка k60gpu – *локальный репозиторий* на ПК. Откройте указанную папку. В ней расположены папка .git и файл **README.md**.

*Вариант 2 (с использованием командной строки Windows)*

1. Для клонирования репозитория выберите на локальном ПК место для клонирования репозитория. В проводнике откройте выбранный диск (или папку). В адресной строке проводника наберите команду cmd и нажмите клавишу **Enter**.
2. На экране появится окно программы **cmd.exe** (терминал). Далее скопируйте ссылку на репозиторий с сайта **GitHub** (пункт 8). В терминале наберите команду git clone, нажмите клавишу Пробел и щелкните ПКМ. Вставится ссылка на репозиторий. Нажмите клавишу Enter. После выполнения команды убедитесь, что репозиторий склонировался в выбранное Вами место.
3. Закройте окно программы Git. Перейдите в выбранную Вами папку (диск) и убедитесь, что в ней появилась папка k60gpu – *локальный репозиторий* на ПК. Откройте указанную папку. В ней расположены папка .git и файл **README.md**.

**2.2 Создание файла gitignore**

В процессе работы с программным кодом в Visual Studio Code создаётся папка .vs, которая содержит вспомогательный файлы. Необходимо сделать папку .vs невидимой для программы Git. Для этого создадим файл **.gitignore**.

***Упражнение 2***

В этом упражнении Вы научитесь создавать файл **.gitignore**.

1. В локальном репозитории k60gpu создайте файл **.gitignore**.
2. Откройте созданный файл с помощью программы Блокнот и наберите следующий текст (рисунок 1.10). Сохраните файл и закройте.

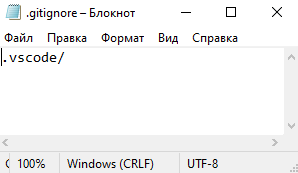


Рисунок 1.10 – Содержимое файла **.gitignore**

**2.3 Создание исходного кода программы**

***Упражнение 3***

В этом упражнении Вы научитесь создавать исходный код программы на языке С с использованием библиотеки MPI.

*Исходный код* (также исходный текст) – текст компьютерной программы на каком-либо языке программирования или языке разметки, который может быть прочтён человеком. В обобщённом смысле – любые входные данные для транслятора. Исходный код транслируется в исполняемый код целиком до запуска программы при помощи компилятора или может исполняться сразу при помощи интерпретатора (<https://ru.wikipedia.org/wiki/Исходный_код>).

1. В локальном репозитории k60gpu создайте папку lab01 и откройте её.
2. Вызовите на экран консоль (через cmd или Git Bash Here).
3. В консоли выполните команду code . (*code пробел точка*). На экране откроется окно программы Visual Studio Code. В появившемся окне в левой части экрана находится проводник (EXPLORER), в правой части – окно редактора кода.
4. Для быстрого запуска Visual Studio Code в текущей папке рекомендуется создать файл **vscode.bat**, содержащий команду code . (*code пробел точка*). Как создать этот файл смотрите в ***Упражнении 4***.
5. Для создания файла, содержащего исходный код программы, в Visual Studio Code выполните команду File – New File. В окне редактора кода откроется файл с именем **Untitled-1**. Сохраните файл с именем **main.c**, выполнив команду File – Save as…
6. Наберите текст программы (рисунок 1.11).

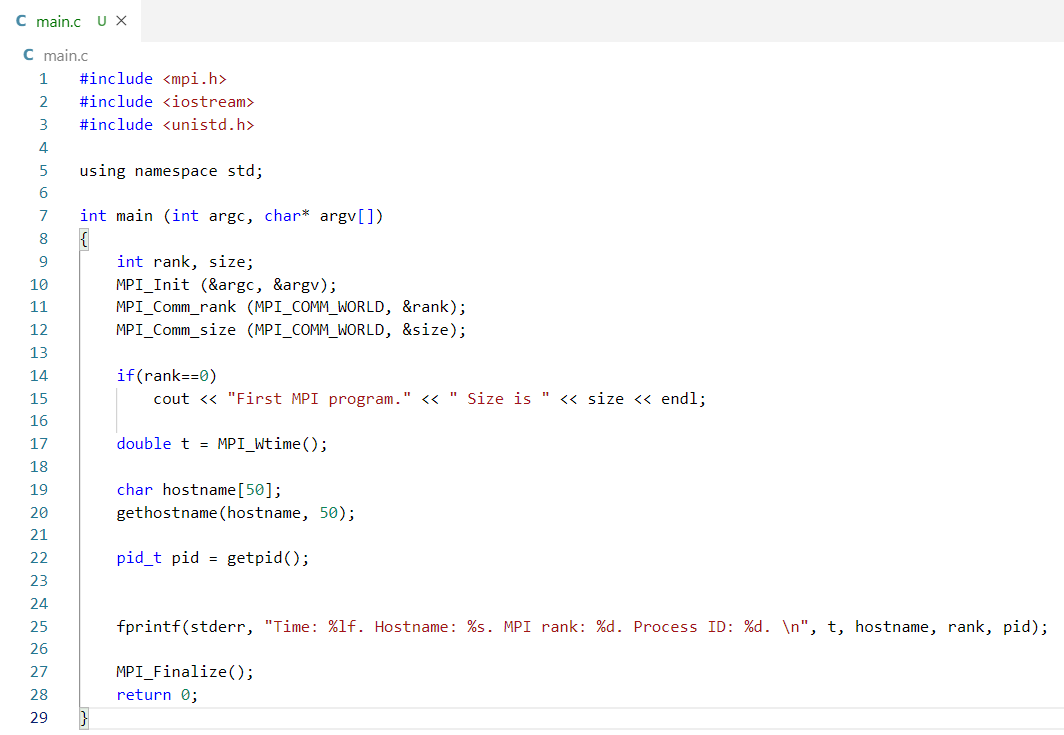


Рисунок 1.11 – Текст программы **main.c**

В тексте программы используются заголовочные файлы: **mpi.h**, **unistd.h**.

В программировании *заголовочный файл* (англ. *header file*) или *подключаемый файл* – файл, содержимое которого автоматически добавляется препроцессором в исходный текст в том месте, где располагается некоторая директива.

В языках программирования С и C++ заголовочные файлы – основной способ подключить к программе типы данных, структуры, прототипы функций, перечисляемые типы и макросы, используемые в другом модуле. По умолчанию используется расширение **.h**; иногда для заголовочных файлов языка C++ используют расширение **.hpp**.

Заголовочный файл в общем случае может содержать любые конструкции языка программирования, но на практике исполняемый код (за исключением inline-функций в C++) в заголовочные файлы не помещают. Например, идентификаторы, которые должны быть объявлены более чем в одном файле, удобно описать в заголовочном файле, а затем его подключать по мере надобности. Подобным же образом работает модульность и в большинстве ассемблеров. По сложившейся традиции, в заголовочных файлах объявляют функции стандартной библиотеки С и Си++ (<https://ru.wikipedia.org/wiki/Заголовочный_файл> ).

1. Поясним код программы построчно.

#include <mpi.h> – основная директива препроцессора #include, добавляющая содержимое файла **mpi.h** в программу.

mpi.h – заголовочный файл, содержащий определения констант и функций библиотеки **MPI**.

#include <iostream> – основная директива препроцессора #include, добавляющая содержимое файла **iostream** в программу.

iostream – заголовочный файл, содержащий определения констант и функций библиотеки, поддерживающей ввод и вывод;

#include <unistd.h> – основная директива препроцессора #include, добавляющая содержимое файла **unistd.h** в программу.

unistd.h – заголовочный файл, необходимый для обеспечения доступа к операционным системам POSIX.

using namespace std – директива using, которая делает доступными все имена в пространстве имён std.

int main (int argc, char\* argv[]) – заголовок функции main, содержащий имя функции main, аргументы функции argc, argv[] и тип возвращаемого значения int. Аргументы функции main: int argc – количество параметров, переданных исполняемому файлу при вызове; char\* argv[] – указатель на первый элемент символьного массива argv[].

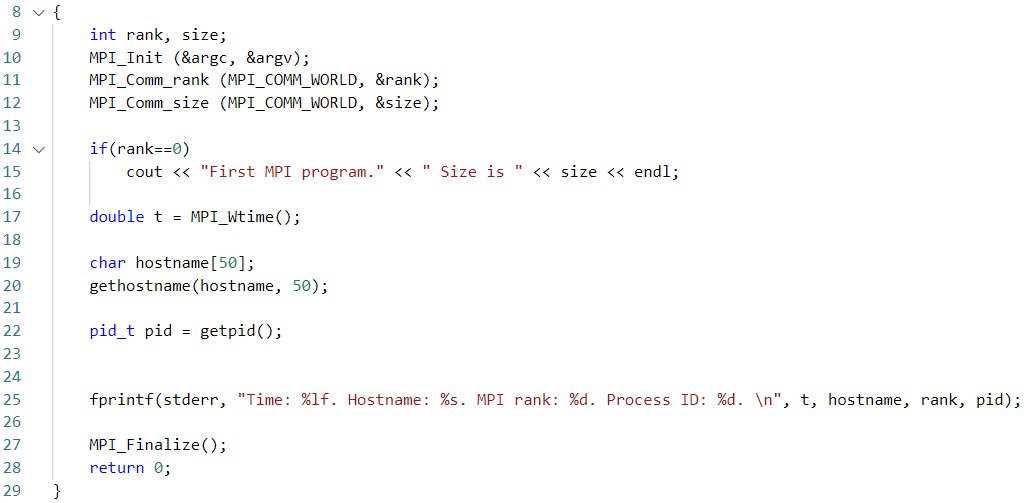


Рисунок 1.12 – Фрагмент кода

Строки 8-29 кода (рисунок 1.12) содержат определение функции main. Ниже разберем тело функции построчно.

int rank, size; – объявление переменных rank, size без их инициализации. Переменная rank будет содержать индекс текущего потока MPI, переменная size будет содержать общее количество потоков MPI.

MPI\_Init (&argc, &argv); – процедура инициализации параллельной части программы.

MPI\_Init (&argc, &argv) – общая процедура MPI. Инициализация параллельной части программы. Все другие процедуры MPI могут быть вызваны только после вызова MPI\_Init. Инициализация параллельной части для каждого приложения должна выполняться только один раз. В языке C функции MPI\_Init передаются указатели на аргументы командной строки программы argc и argv, из которых системой могут извлекаться и передаваться в параллельные процессы некоторые параметры запуска программы. (Антонов, 2004 г.)

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &rank); – процедура, которая возвращает номер процесса в коммуникаторе MPI\_COMM\_WORLD.

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &rank) – общая процедура MPI. В аргументе rank процедура возвращает номер процесса в коммуникаторе MPI\_COMM\_WORLD. (Антонов, 2004 г.)

MPI\_COMM\_WORLD – коммуникатор, включающий все процессы параллельного приложения.

MPI\_Comm\_size (MPI\_COMM\_WORLD, &size); – процедура, которая возвращает число параллельных процессов в коммуникаторе MPI\_COMM\_WORLD.

MPI\_Comm\_size (MPI\_COMM\_WORLD, &size) – общая процедура MPI. В аргументе size процедура возвращает число параллельных процессов в коммуникаторе MPI\_COMM\_WORLD. (Антонов, 2004 г.)

if(rank==0)

cout << "First MPI program." << " Size is " << size << endl;

Строки 14-15 кода содержат вывод в стандартный поток вывода количества потоков (переменная size) только для потока MPI с номером 0 (rank = 0).

double t = MPI\_Wtime(); – текущее значение времени.

Double precision MPI\_Wtime() – эта функция возвращает на вызвавшем процессе астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. (Антонов, 2004 г.)

char hostname[50]; – объявление символьного массива hostname.

gethostname(hostname, 50); – извлекает стандартное имя хоста (локального компьютера) для локального компьютера с именем hostname длиной 50 элементов типа char.

int gethostname(char \*name, size\_t len) – возвращает имя узла с null на конце в массиве символов name длиной len байт. Если имя узла, оканчивающееся null, не помещается, то имя обрезается и ошибки не происходит. В POSIX.1 сказано, что если обрезание произошло, то неясно, будет ли буфер содержать завершающий байт с null. (<http://ru.manpages.org/gethostname/2> )

pid\_t pid = getpid(); – переменная, возвращающая ID текущего процесса.

Pid\_t тип данных – целое число со знаком, который способен представить ID процесса.

Getpid – функция возвращает ID текущего процесса. (<https://www.opennet.ru/docs/RUS/glibc/glibc-23.html> )

fprintf(stderr, "Time: %lf. Hostname: %s. MPI rank: %d. Process ID: %d. \n", t, hostname, rank, pid); – функция, которая выводит в поток stderr значения переменных t, hostname, rank, pid.

int fprintf(FILE \*stream, const char \* format,...); – функция осуществляет форматированный вывод в специфицированный поток.

MPI\_Finalize(); – завершение параллельной части программы.

MPI\_Finalize () – общая процедура MPI. Завершение параллельной части приложения. Все последующие обращения к любым процедурам MPI, в том числе к MPI\_Init, запрещены. К моменту вызова MPI\_Finalize каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены. (Антонов, 2004 г.)

return 0; – возвращает результат работы функции main.

1. Сохраните файл и закройте программу.

***Упражнение 4***

В этом упражнении Вы научитесь создавать пакетный файл vscode.bat.

*Пакетный файл* (англ. batch file) – текстовый файл в MS-DOS, OS/2 или Windows, содержащий последовательность команд, предназначенных для исполнения командным интерпретатором. После запуска пакетного файла программа-интерпретатор (как правило, COMMAND.COM или cmd.exe) читает его строка за строкой и последовательно исполняет команды. Пакетный файл – аналог скриптовых файлов командной строки (shell script) в Unix-подобных операционных системах

(<https://ru.wikipedia.org/wiki/Пакетный_файл> ).

1. Запустите программу Visual Studio Code. В папке lab01 создайте файл **vscode.bat**, содержащий команду code . (*code пробел точка*).
2. Сохраните файл и закройте его. В проводнике в текущей папке появится созданный файл.

***Упражнение 5***

В этом упражнении Вы научитесь проверять состояние локального репозитория на ПК (git status), вносить в него изменения (git add), фиксировать (подтверждать) внесенные изменения (git commit) и вносить изменения в удаленный репозиторий (git push).

1. В программе Visual Studio Code выполните команду Terminal – New Terminal. В окне редактора кода в нижней части появится панель терминала.
2. В терминале выполните команду

git add ./

В результате git добавил в отслеживание папку lab01 вместе с её содержимым.

1. Проверьте состояние локального репозитория. Для этого в терминале выполните команду

git status

Результаты работы команды показаны на рисунке 1.13.

1. Для фиксирования (подтверждения) внесенных изменений в локальном репозитории в терминале выполните команду

git commit –m"added main.c, vscode.bat".

На экране появится сообщение о том, что изменения внесены (рисунок 1.14).

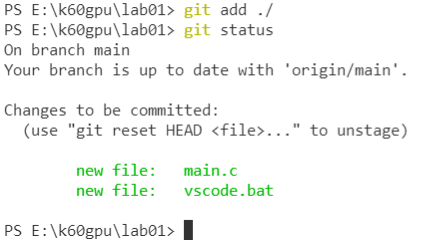


Рисунок 1.13 – Результат работы команды git status

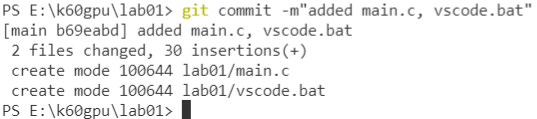


Рисунок 1.14 – Результат работы команды git commit

1. Для внесения изменения в удаленный репозиторий в терминале выполните команду

git push

На экране появится следующая информация (рисунок 1.15). Если при первой отправке изменений в удаленный репозиторий на экране появится окно с запросом логина и пароля для входа в Ваш аккаунт **GitHub**, введите логин и пароль и щелкните по кнопке **Login**. Далее выполните процедуру авторизации программы Visual Studio Code для доступа на **GitHub**.

1. Закройте программу Visual Studio Code.
2. Перейдите на **GitHub**, обновите страницу и убедитесь, что в удаленном репозитории k60gpu появилась папка lab01 с файлами.

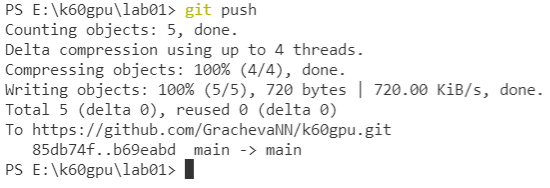


Рисунок 1.15 – Результат работы команды git push

**2.4 Работа с кластером**

Вычислительный комплекс K-60 соcтоит из двух секций – одна без графических ускорителей k60.kiam.ru, другая с графическими ускорителями k60gpu.kiam.ru. Это главный вычислительный ресурс ИПМ РАН.

Аппаратное обеспечение без GPU состоит из 85 вычислительных узлов. Каждый узел представляет собой двухпроцессорный сервер со следующими характеристиками:

* процессоры 2 х Intel Xeon E5-2690 v4;
* оперативная память 256GB;
* диск 1TB;
* 2 сетевых карты Gigabit Ethernet;
* 2 сетевых карты 56G Infiniband;
* корпус: 19" 2U.

Аппаратное обеспечение c GPU состоит из 10 вычислительных узлов. Каждый узел представляет собой двухпроцессорный сервер со следующими характеристиками:

* процессоры 2 х Intel Xeon Gold 6142 v4(16 x ядер);
* GPU 4 x nVidia Volta GV100GL;
* оперативная память 768GB;
* диск 2TB;
* 2 сетевых карты Gigabit Ethernet;
* 2 сетевых карты 56G Infiniband;
* корпус: 19" 4U.

Управляющая машина имеет следующие характеристики:

* 2 х Intel Xeon E5-2690 v4;
* оперативная память 256GB;
* диск 1TB;
* корпус 19" 2U;
* 2 сетевых карты Gigabit Ethernet;
* 1 сетевая карта 10G Ethernet;
* 1 сетевая карта 56G Infiniband.

Система хранения данных, общая для обеих секций.

* /nethome на 38TB;
* /tmphome на 241TB.

Характеристики сети:

* внутренняя сеть Gigabit Ethernet для управления вычислительными узлами;
* внутренняя сеть Infiniband для взаимодействия вычислительных узлов при счете параллельной программы;
* внутренняя сеть Infiniband для монтирования общей файловой системы **/nethome**.

Программное обеспечение кластера состоит из:

* операционная система CentOS 7.3;
* Система Управления Прохождением Пользовательских Задач (СУППЗ).
* распараллеливание вычислений [MPI](http://www2.sscc.ru/Litera/liter2.htm), OpenMP для обеих секций, CUDA для секции с GPU.
* компиляторы С/C++, Fortran для обеих секций, CUDA для секции с GPU;
* сервер удаленного доступа по протоколам SSH/SCP.

**2.3.1 Создание исполняемого файла**

*Исполняемый файл* (executable file) – это файл, который может быть загружен в память загрузчиком операционной системы и затем исполнен ([https://intuit.ru/studies/courses/89/89/lecture/28301#:~:text=Исполняемый%20файл%20(executable%20file)%20-,имеют%20расширения%20".exe"%20и%20".dll"](https://intuit.ru/studies/courses/89/89/lecture/28301#:~:text=Исполняемый%20файл%20(executable%20file)%20-,имеют%20расширения%20".exe"%20и%20".dll" ) ).

***Упражнение 6***

В этом упражнении Вы научитесь создавать новое подключение к кластеру k60gpu в программе MobaXterm.

1. Откройте программу MobaXterm (рисунок 1.16).

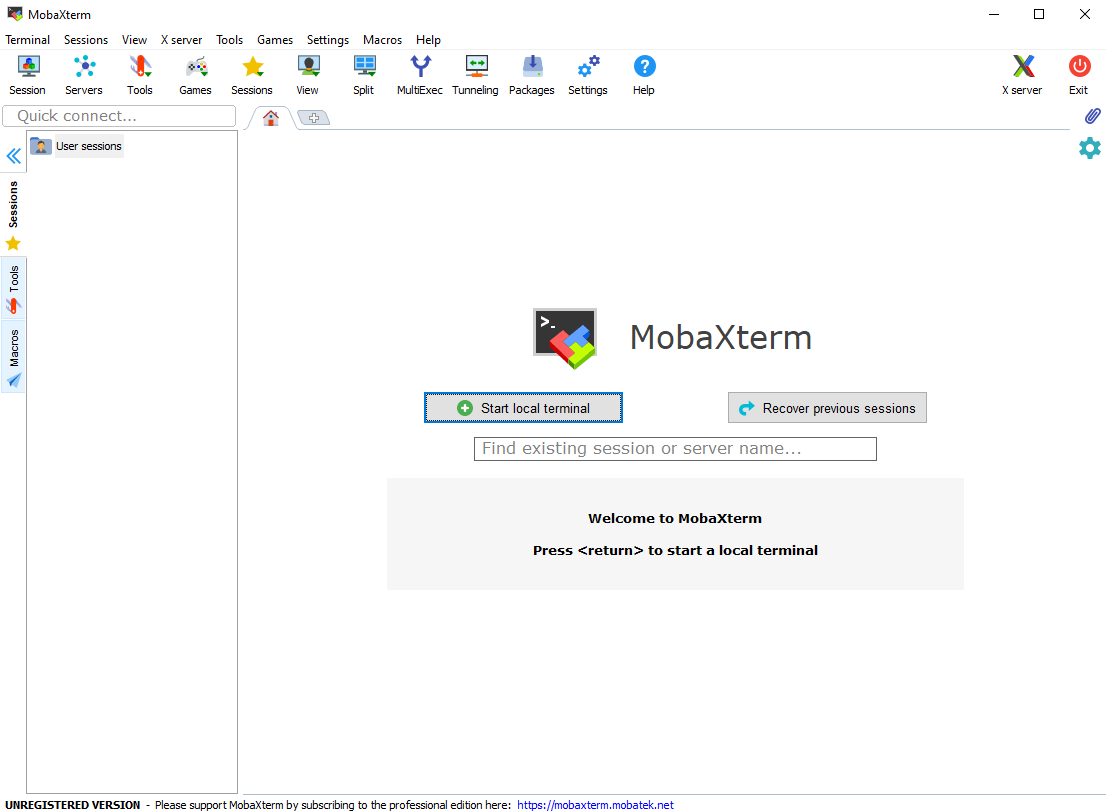


Рисунок 1.16 – Окно программы MobaXterm

1. На панели инструментов щелкните по кнопке Session . На экране откроется окно Session settings (рисунок 1.17).
2. На панели инструментов щелкните по кнопке SSH . В окне Session settings появятся настройки SSH. Заполните настройки, как показано на рисунке 1.18. Обратите внимание, что в поле Specify username Вы должны ввести свой логин (адрес сервера кластера, логин и пароль доступа необходимо получить заранее). Щелкните по кнопке OK.

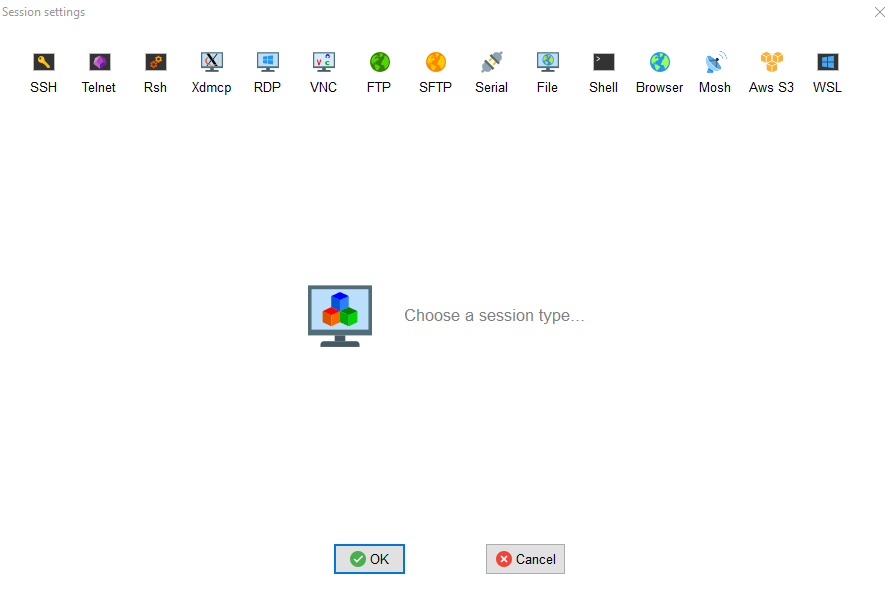


Рисунок 1.17 – Окно настроек Session settings

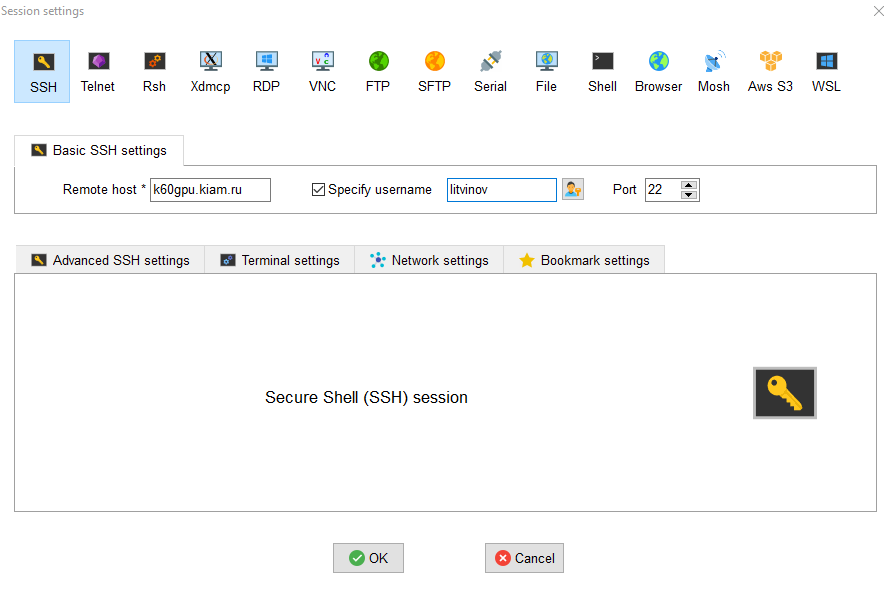


Рисунок 1.18 – Окно настроек Session settings с настройками SSH

1. На вкладке Sessions проводника программы MobaXterm появилось новое подключение (рисунок 1.19).

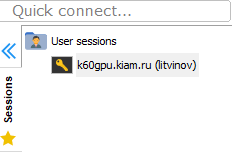


Рисунок 1.19 – Новое подключение

1. Подключитесь к кластеру k60gpu, щелкнув дважды ЛКМ по новому подключению. Откроется окно с новой сессией, где для первого подключения необходимо ввести пароль.
2. После входа на кластер откроется окно терминала с приглашением к работе вида [litvinov@k60gpu ~]$ (рисунок 1.20). Оно несет в себе информацию об имени пользователя (litvinov), имени компьютера (k60gpu) и текущей папке (каталоге) (~). Тильда (~) обозначает домашнюю папку. Если на кластер зайдет другой пользователь, то изменится первая строка (до знака @). Если поменять текущий каталог, то знак тильды будет заменен на имя нового текущего каталога.



Рисунок 1.20 – Приглашение к работе

**2.3.2 Клонирование данных с удаленного репозитория  
на вычислительный кластер**

***Упражнение 7***

В этом упражнении Вы научитесь клонировать удаленный репозиторий с GitHub на вычислительный кластер k60gpu.

1. На кластере в домашней папке создайте папку work. Для этого в терминале выполните команду

mkdir work

1. Сделайте созданный каталог текущим, выполнив команду

cd work

1. Скопируйте ссылку на удаленный репозиторий с GitHub (упражнение 1, п. 8). Клонируйте удаленный репозиторий в папку work (упражнение 1, п. 9).
2. Сделайте текущей папку k60gpu (команда cd k60gpu).
3. Для упрощения работы со структурой папок можно воспользоваться оболочкой Midnight Commander. Чтобы открыть оболочку, наберите команду mc и нажмите клавишу **Enter.** На экране появится окно оболочки с содержимым текущей папки (в данном случае, папки k60gpu) (рисунок 1.21).

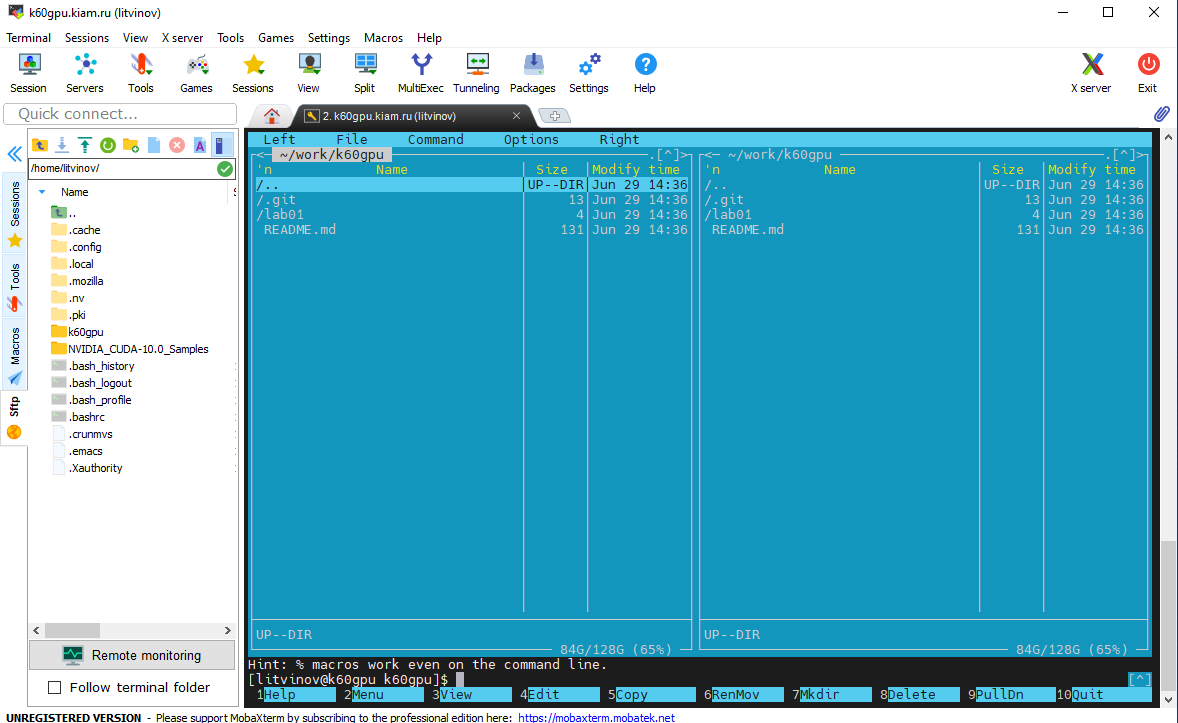


Рисунок 1.21 – Окно с оболочкой mc

**2.3.2 Компилирование исходного кода программы в исполняемый файл**

***Упражнение 8***

В этом упражнении Вы научитесь компилировать исходный код программы в исполняемый файл.

1. Откройте папку lab01. Для этого в оболочке mc щелкните ЛКМ по папке lab01. На экране откроется содержимое указанной папки (рисунок 1.22).

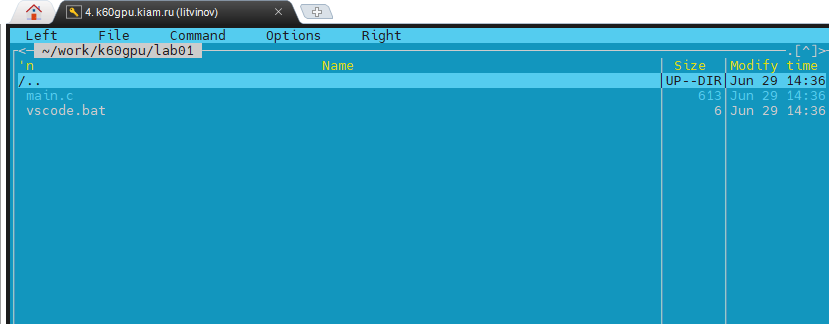


Рисунок 1.22 – Содержимое папки lab01

1. Нажмите комбинацию клавиш **Ctrl + O**. Окно оболочки свернется, на экране будет виден терминал. Для просмотра содержимого в папке в оболочке снова нажмите **Ctrl + O**.
2. Для компилирования файла с исходным кодом **main.c** в командной строке терминала выполните команду

mpicxx -O3 -c main.c

Параметр -c указывает компилятору, что необходимо создать объектный файл, в данном случае файл **main.o**. После выполнения последней команды в папке lab01 появится файл **main.o** (рисунок 1.23).

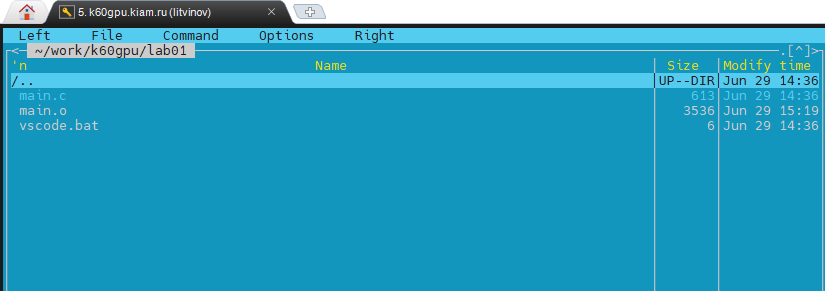


Рисунок 1.23 – Содержимое папки lab01 с объектным файлом **main.o**

1. Для создания исполняемого файла **myapp** из объектного файла **main.o** в командной строке терминала выполните команду

mpicxx -o myapp main.o

В папке lab01 появится исполняемый файл **myapp** (рисунок 1.24).

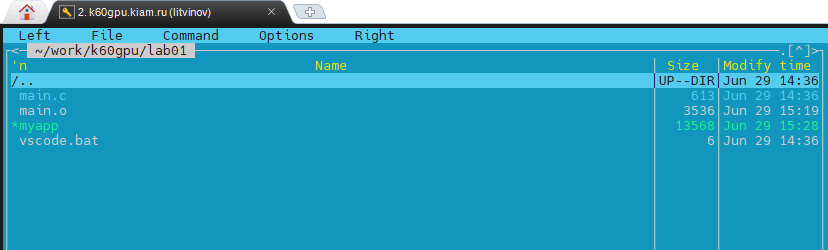


Рисунок 1.24 – Содержимое папки lab01 с исполняемым файлом **myapp**

**2.3.3 Запуск исполняемого файла на кластере**

***Упражнение 9***

В этом упражнении Вы научитесь запускать исполняемый файл на вычислительном кластере k60gpu под управлением MPI.

1. Для запуска исполняемого файла **myapp** в командной строке терминала выполните команду

mpirun -np 4 ./myapp

При запуске программы на исполнение после имени команды mpirun указывается два параметра: первый параметр –np 4 указывает на число необходимых MPI-процессов – 4, второй параметр ./myapp указывает путь к исполняемому файлу. В терминале появится сообщение следующего содержания (рисунок 1.25):

* задаче присвоен идентификатор **myapp.1** и она поставлена в очередь на исполнение;
* указано название вычислительного узла, на котором запланировано исполнение задачи, в данном случае, узел **gpu09**;
* задача **myapp.1** успешно стартовала и присвоен **pid**, в данном случае, **1395**.

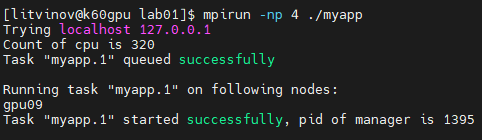


Рисунок 1.25 – Результат выполнения команды

1. Для проверки списка исполняющихся в данный момент задач пользователя в терминале выполните команду

mps

На экране появится сообщение со списком запущенных задач (рисунок 1.26). В данном случае, список пуст.

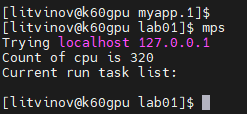


Рисунок 1.26 – Результат выполнения команды mps

1. Для проверки статуса выполнения задачи **myapp.1** в терминале выполните команду

mqtest myapp.1

На экране появится сообщение, что задача **myapp.1** выполнена (рисунок 1.27). Это говорит о том, что запущенная ранее на исполнение задача выполнена. Поэтому можно посмотреть результаты работы программы.

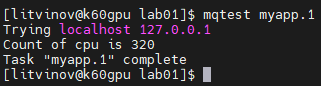


Рисунок 1.27 – Результат выполнения команды mqtest

1. Результаты работы программы располагаются в папке, имя которой соответствует имени задачи – myapp.1 (рисунок 1.28).

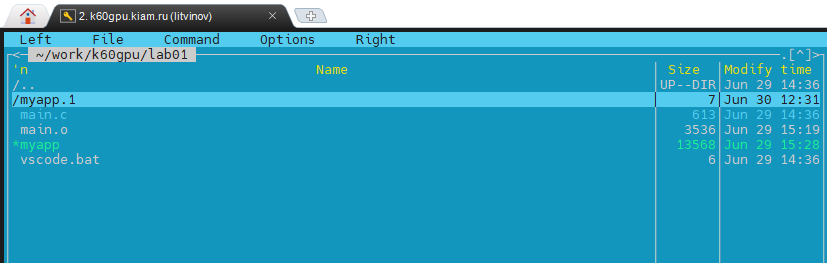


Рисунок 1.28 – Содержимое папки lab01 с новой папкой myapp.1

1. Откройте папку myapp.1. В ней находятся 5 файлов. Рассмотрите содержимое файлов **output** и **errors**. Выберите файл **output** и откройте его, щелкнув по кнопке **F3**.
2. На экране появится содержимое файла **output** (рисунок 1.29).

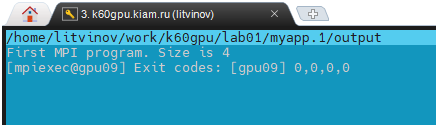


Рисунок 1.29 – Содержимое файла **output**

В указанном файле находятся данные, которые направляются приложением в стандартный поток вывода. В файле исходного кода **main.c** стандартный поток вывода передается с помощью функции cout. Оператор cout выводит сообщение (**“First MPI program”)** и общее количество потоков size (**4**). Далее в файле следует служебная информация. Закройте файл **output**, щелкнув по кнопке **F3**.

1. Выберите файл **errors** и откройте его. На экране появится содержимое файла **errors** (рисунок 1.30).

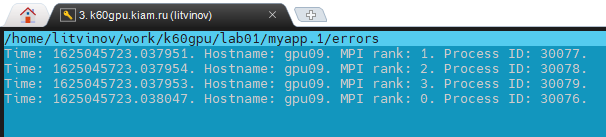


Рисунок 1.30 – Содержимое файла **errors**

В указанном файле находятся данные, которые направляются приложением в стандартный поток ошибки. В файле исходного кода **main.c** стандартный поток ошибки передается с помощью функции fprintf. Оператор fprintf выводит значение переменной t, имя хоста hostname (**gpu09**), номер процесса MPI rank (**0-3**), pid (**30076-30079**) для каждого MPI-потока.

Т.к. при запуске исполняемого файла в команде было указано число необходимых MPI-процессов 4, то в файле **errors** мы видим 4 строки вывода. Обратите внимание, что содержимое Вашего файла **errors** может отличаться от содержимого на рисунке 1.29. Могут измениться значения переменных Time, Hostname и Process ID. Закройте файл **errors**, щелкнув по кнопке **F3**.

1. Сделайте текущей папку lab01. Сверните окно оболочки. Запустите исполняемый файл с новыми параметрами команды mpirun. В терминале выполните команду

mpirun -np 1 -ppn 1 -maxtime 2 ./myapp

При запуске программы на исполнение после имени команды mpirun указывается четыре параметра: первый параметр -np 1 указывает на число необходимых MPI-процессов – 1, второй параметр -ppn 1 указывает на число процессов MPI, запускаемых на каждом вычислительном узле – 1, третий параметр -maxtime 2 указывает на максимальное время выполнения запускаемой этой командой задачи (в минутах), четвертый параметр ./myapp указывает путь к исполняемому файлу. В терминале появится сообщение следующего содержания (рисунок 1.31):

* задаче присвоен идентификатор **myapp.2** и она поставлена в очередь на исполнение;
* указано название вычислительного узла, на котором запланировано исполнение задачи, в данном случае, узел **gpu03**;
* задача **myapp.2** успешно стартовала и присвоен **pid**, в данном случае, **29897**.

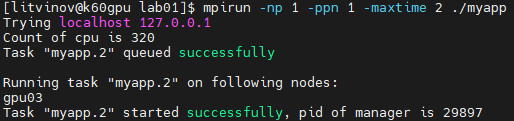


Рисунок 1.31 – Результат выполнения команды

**Обратите внимание!** Если до запуска задачи **myapp.2** выполнение задачи **myapp.1** завершилось, то задача **myapp.2** получит идентификатор **myapp.1** и результаты выполнения этой задачи запишутся в папку myapp.1.

1. Результаты работы программы располагаются в папке, имя которой соответствует имени задачи – myapp.2 (рисунок 1.32).

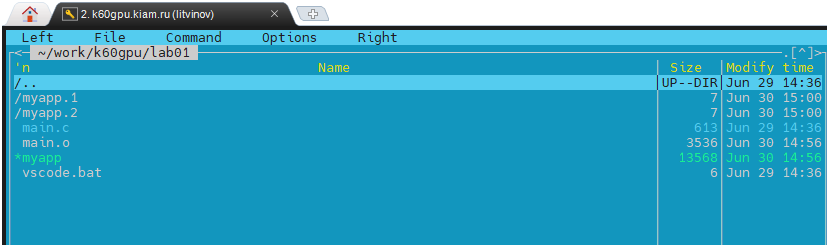


Рисунок 1.32 – Содержимое папки lab01 с новой папкой myapp.2

1. Откройте папку myapp.2. Рассмотрите содержимое файла **errors** (рисунок 1.33).

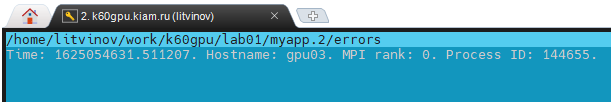


Рисунок 1.33 – Содержимое файла **errors**

При запуске исполняемого файла в команде было указано число необходимых MPI-процессов 1, поэтому в файле **errors** мы видим 1 строку вывода. Закройте файл **errors**.

1. Сделайте текущей папку lab01. Сверните окно оболочки. Запустите исполняемый файл с новыми параметрами команды mpirun. В терминале выполните команду

mpirun -np 2 -ppn 2 -maxtime 2 ./myapp

Рассмотрите параметры команды и сообщение, которое появится в терминале в результате выполнения последней команды.

1. Результаты работы программы сравните с рисунком 1.34.

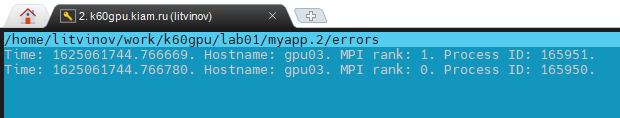


Рисунок 1.34 – Содержимое файла **errors**

1. Откройте папку myapp.2. Рассмотрите содержимое файла **errors** (рисунок 1.33). Закройте файл. Сверните оболочку.
2. Проверьте состояние запущенных на исполнение задач. В терминале выполните команду

mps

Если список запущенных задач не пуст (рисунок 1.34), то отмените выполнение перечисленных в списке задач (в данном случае, задачи **myapp.1**, задачи **myapp.3**). Для этого выполните команду

mcancel <имя задачи>

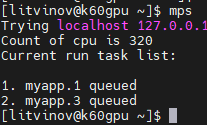


Рисунок 1.35 – Список запущенных задач

Убедитесь, что список запущенных задач пуст. В терминале выполните команду

mps

1. Для просмотра детальной информации обо всех запущенных Вами задачах, количестве исполняемых задач и задач в очереди, в терминале выполните команду

pult t

Результат работы команды показан на рисунке 1.36.

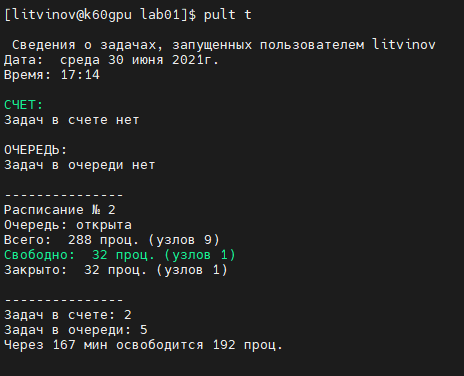


Рисунок 1.36 – Результат работы команды pult t

1. Разверните оболочку. Сделайте текущей папку lab01 и удалите все папки задач **myapp** с их содержимым и файлы **main.o**, **myapp**.
2. Закройте оболочку (клавиша **F10**).

**2.3.4 Создание скриптов и запуск их на выполнение**

В предыдущих упражнениях все команды в терминал Вы вводили вручную. Но при любом изменении исходного кода, Вам необходимо ввести в терминале эти команды заново. Чтобы этого не делать, необходимо создать файл (скрипт), содержащий все необходимые команды. Этот же подход можно применить в случае, когда один и тот же исполняемый файл необходимо запустить с использованием различных параметров.

***Упражнение 10***

В этом упражнении Вы создадите скрипт, который обновляет информацию в локальном репозитории, компилирует проект и запускает его на исполнение с различными параметрами.

1. Запустите программу Visual Studio Code. Для этого в папке lab01 локального репозитория на ПК щелкните дважды ЛКМ по файлу **vscode.bat**.
2. В Visual Studio Code создайте файл **run.sh**. Содержимое указанного файла изображено на рисунке 1.37.

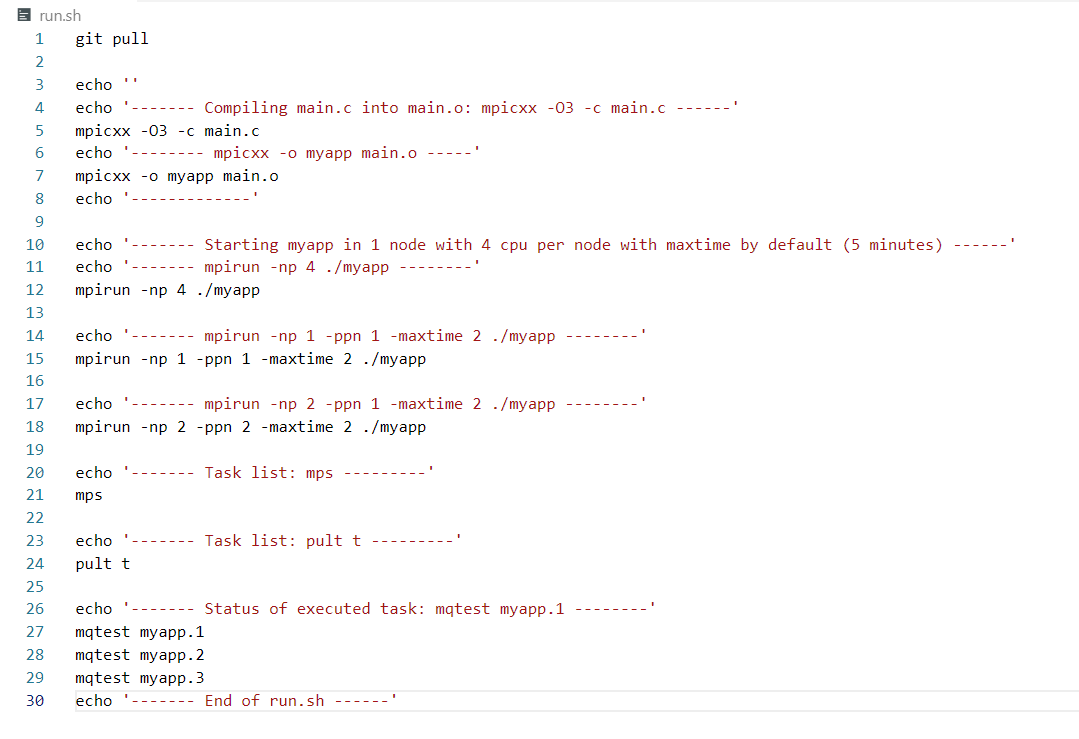


Рисунок 1.37 – Содержимое файла **run.sh**

Для обновления информации в удаленном репозитории на GitHub создадим файл, содержащий команды git.

***Упражнение 11***

В этом упражнении Вы создадите скрипт, который обновляет информацию в удаленном репозитории, и запустите его на выполнение.

1. В Visual Studio Code создайте файл **gitpush.bat**. Содержимое указанного файла изображено на рисунке 1.38.

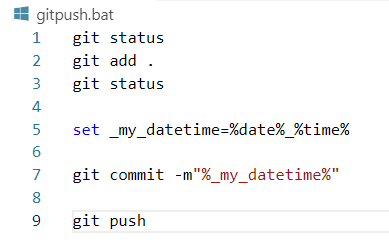


Рисунок 1.38 – Содержимое файла **gitpush.bat**

1. Откройте терминал в программе Visual Studio Code. Выполните команду

./gitpush.bat

1. Перейдите в программу MobaXterm и в терминале выполните команду

git pull

Таким образом, Вы обновили информацию в локальном репозитории кластера. В папке lab01 появились новые файлы: **gitpush.bat**, **run.sh** (рисунок 1.39).

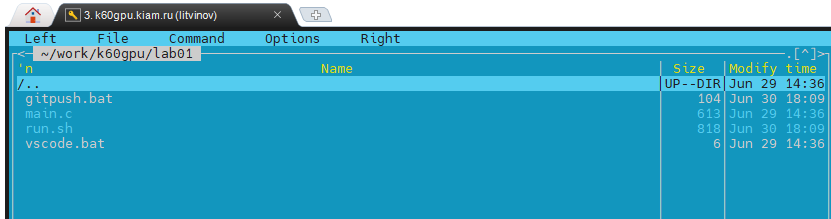


Рисунок 1.39 – Обновленное содержимое папки lab01

***Упражнение 12***

В этом упражнении Вы запустите на выполнение скрипт run.sh.

1. Сверните окно оболочки. В терминале выполните команду

bash run.sh

В результате работы скрипта **run.sh** обновлен локальный репозиторий, скомпилирован проект и запущен на исполнение с различными параметрами.

1. Откройте оболочку и убедитесь, что в папке lab01 появились три папки задач myapp (рисунок 1.40). Количество папок myapp соответствует количеству запускаемых задач в файле **run.sh**.

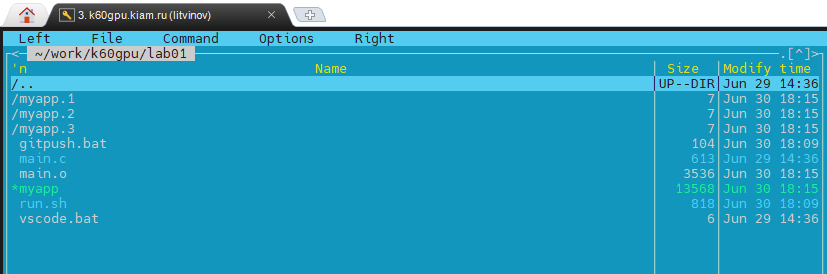


Рисунок 1.40 – Обновленное содержимое папки lab01

***Упражнение 13***

В этом упражнении Вы создадите и запустите на выполнение скрипт **clear.sh**, который удаляет созданные в результате работы папки и файлы.

1. В Visual Studio Code создайте файл **clear.sh**. Содержимое указанного файла изображено на рисунке 1.41.

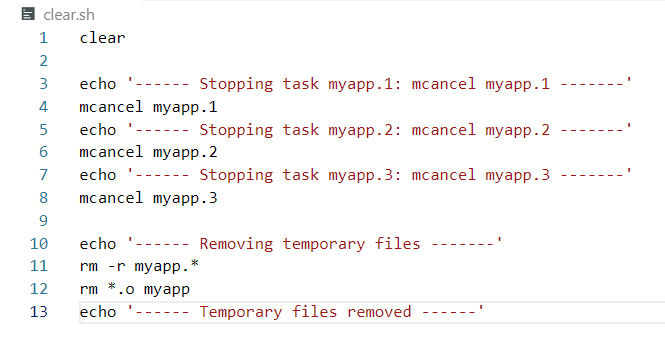


Рисунок 1.41 – Содержимое файла **clear.sh**

1. Откройте терминал в программе Visual Studio Code. Выполните команду

./gitpush.bat

В результате работы команды в удаленный репозиторий будет добавлен созданный файл **clear.sh**.

1. Перейдите в окно программы MobaXterm. Сверните окно оболочки.  
   В терминале выполните команду

git pull

В результате работы команды в локальный репозиторий на кластере будет добавлен созданный файл **clear.sh**. Разверните оболочку и просмотрите содержимое папки lab01 (рисунок 1.42).

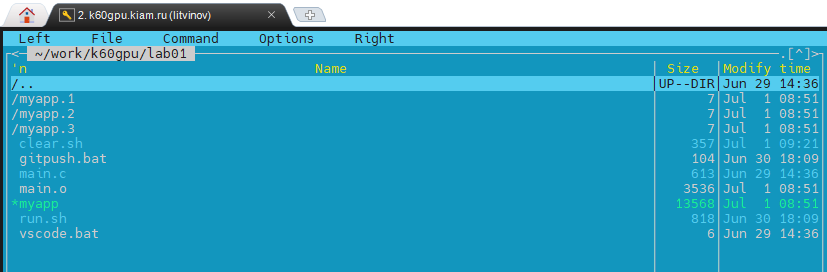


Рисунок 1.42 – Обновленное содержимое папки lab01

1. Сверните оболочку и в терминале выполните команду

bash clear.sh

В результате работы скрипта **clear.sh** будут удалены папки myapp.\* и файлы **main.o**, **myapp**. Разверните оболочку и просмотрите содержимое папки lab01 (рисунок 1.43).

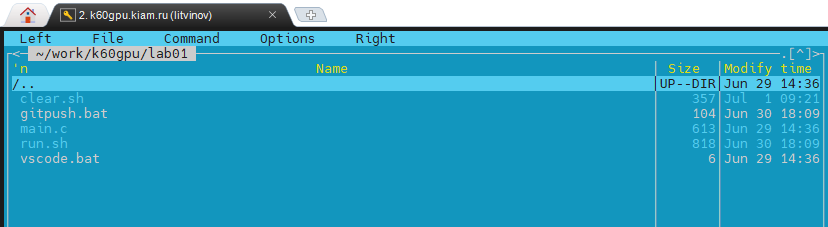


Рисунок 1.43 – Обновленное содержимое папки lab01

В процессе выполнения лабораторной работы Вам пришлось работать с тремя репозиториями: удаленным репозиторием на GitHub и двумя локальными – один локальный репозиторий находится на ПК, второй – на кластере. Любые изменения в локальных репозиториях необходимо вносить в удаленный репозиторий. Обмен изменениями между локальными репозиториями осуществляется через удаленный репозиторий.

На рисунке рисунке 1.44 изображена схема процесса работы с проектом с использованием Git – репозитория.

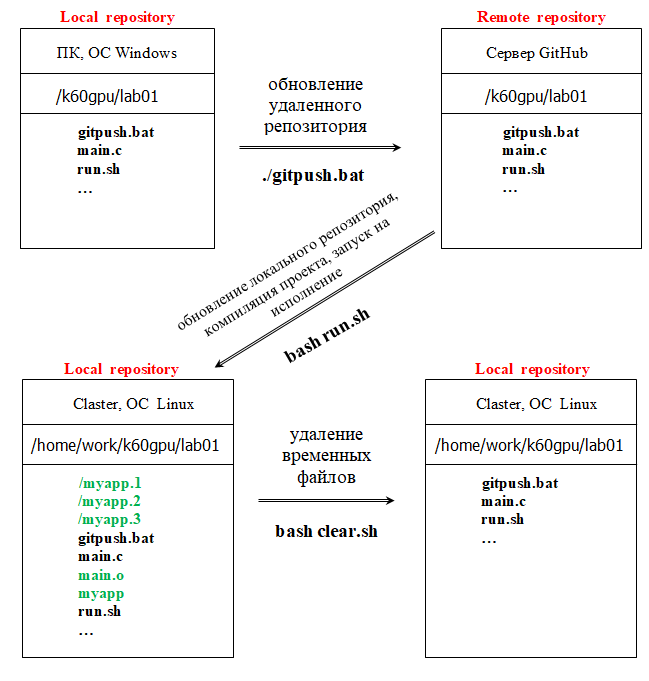


Рисунок 1.44 – Схема процесса работы с проектом с использованием  
Git – репозитория

#### ЛАБОРАТОРНАЯ РАБОТА № 2

Тема: **создание MPI-программы с потоками. Компилирование исполняемого файла из нескольких файлов исходного кода**

**1 ЦЕЛЬ РАБОТЫ**

Познакомиться:

* с интерфейсом интегрированной среды разработки Ms Visual Studio 2010;
* с основными возможностями Ms Visual Studio 2010.

Научиться:

* создавать и сохранять проекты различных типов;
* выполнять отладку проектов.

**2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ.  
ТЕХНОЛОГИЯ ВЫПОЛНЕНИЯ РАБОТЫ**

**2.1 Создание исходного кода программы в одном файле main.cpp**

***Упражнение 1***

В этом упражнении Вы научитесь создавать исходный код программы на языке С++ (файл **main.cpp**).

1. В локальном репозитории k60gpu создайте папку lab02 и откройте её.
2. Скопируйте в текущую папку файлы **gitpush.bat**, **vscode.bat** из папки lab01.
3. Откройте программу Visual Studio Code. Для этого щелкните дважды ЛКМ по файлу **vscode.bat**.
4. Создайте файл **main.cpp**.
5. Наберите текст программы (рисунок 2.1).
6. Поясним код программы построчно.

#include <mpi.h> – основная директива препроцессора #include, добавляющая содержимое файла **mpi.h** в программу.

#include <iostream> – основная директива препроцессора #include, добавляющая содержимое файла **iostream** в программу.

#include <unistd.h> – основная директива препроцессора #include, добавляющая содержимое файла **unistd.h** в программу.

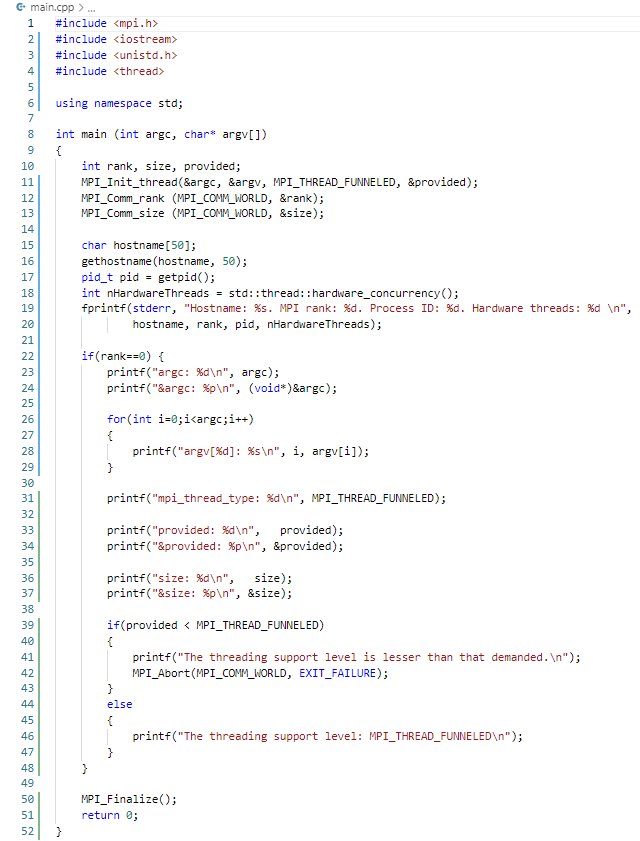


Рисунок 2.1 – Текст программы **main.cpp**

#include <thread> – основная директива препроцессора #include, добавляющая содержимое файла **thread** в программу.

thread – заголовочный файл, содержащий определение класса **thread** и различных вспомогательных функций.

using namespace std – директива using, которая делает доступными все имена в пространстве имён std.

int main (int argc, char\* argv[]) – заголовок функции main, содержащий имя функции main, аргументы функции argc, argv[] и тип возвращаемого значения int. Аргументы функции main: int argc – количество параметров, переданных исполняемому файлу при вызове; char\* argv[] – указатель на первый элемент символьного массива argv[].

Строки 10-51 кода (рисунок 2.1) содержат определение функции main. Ниже разберем тело функции построчно.

int rank, size, provided; – объявление переменных rank, size, provided без их инициализации. Переменная rank будет содержать индекс текущего потока MPI, переменная size будет содержать общее количество потоков MPI, переменная provided будет содержать текущее значение настройки многопоточности, примененное MPI.

Потоковая безопасность в MPI не работает из коробки. Сначала вы должны убедиться, что ваша реализация на самом деле поддерживает несколько потоков, выполняющих вызовы MPI одновременно. В некоторых реализациях MPI, например, в Open MPI, для этой библиотеки необходимо настроить специальные параметры во время сборки. Затем вы должны указать MPI инициализироваться на соответствующем уровне поддержки потоков. В настоящее время стандарт MPI определяет четыре уровня поддержки потоков:

MPI\_THREAD\_SINGLE – означает, что код пользователя является однопоточным. Это уровень по умолчанию, на котором инициализируется MPI, если MPI\_Init() используется;

MPI\_THREAD\_FUNNELED – означает, что пользовательский код является многопоточным, но только основной поток выполняет вызовы MPI. Основной поток - это тот, который инициализирует библиотеку MPI;

MPI\_THREAD\_SERIALIZED – означает, что код пользователя является многопоточным, но вызовы библиотеки MPI сериализуются;

MPI\_THREAD\_MULTIPLE – означает, что код пользователя является многопоточным, и все потоки могут выполнять вызовы MPI в любое время без какой-либо синхронизации.

MPI\_Init\_thread(&argc, &argv, MPI\_THREAD\_FUNNELED, &provided); – реализация MPI с возможностью задания многопоточности.

MPI\_Comm\_rank  (MPI\_COMM\_WORLD, &rank); – процедура, которая возвращает номер процесса в коммуникаторе MPI\_COMM\_WORLD.

MPI\_Comm\_size  (MPI\_COMM\_WORLD, &size); – процедура, которая возвращает число параллельных процессов в коммуникаторе MPI\_COMM\_WORLD.

char hostname[50]; – объявление символьного массива hostname.

gethostname(hostname, 50); – извлекает стандартное имя хоста (локального компьютера) для локального компьютера с именем hostname длиной 50 элементов типа char.

pid\_t pid = getpid(); – переменная, возвращающая ID текущего процесса.

int nHardwareThreads = std::thread::hardware\_concurrency(); – переменная, содержащая количество задач, которые реально можно выполнять в многопоточном режиме на компьютере.

fprintf(stderr, "Hostname: %s. MPI rank: %d. Process ID: %d. Hardware threads: %d \n", hostname, rank, pid, nHardwareThreads); – функция, которая выводит в поток stderr значения переменных hostname, rank, pid, nHardwareThreads.

Строки 22-48 кода содержат вывод в стандартный поток вывода значения переменных и их адресов только для потока MPI с номером 0 (rank = 0).

MPI\_Finalize(); – завершение параллельной части программы.

return 0; – возвращает результат работы функции main.

1. Сохраните файл.

***Упражнение 2***

В этом упражнении Вы создадите скрипт **run.sh,** необходимый для обновления репозитория, компиляции исходного кода и запуска исполняемого файла.

1. В программе Visual Studio Code создайте файл **run.sh** (рисунок 2.2). Сохраните файл.

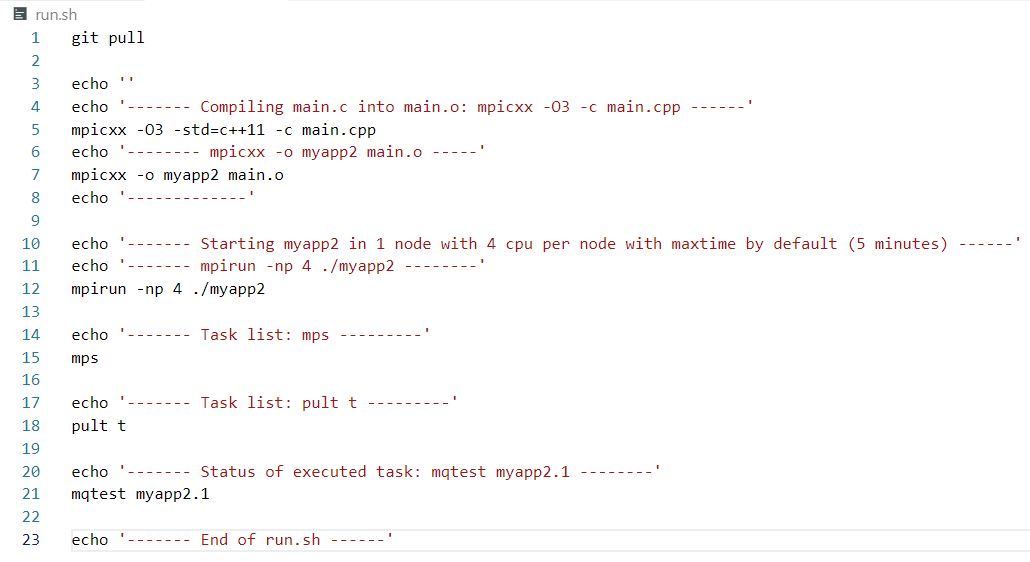


Рисунок 2.2 – Содержимое файла **run.sh**

***Упражнение 3***

В этом упражнении Вы обновите информацию в локальном репозитории k60gpu на кластере.

1. Перейдите в локальный репозиторий k60gpu на ПК. Выполните обновление информации в удаленном репозитории. Для этого запустите на выполнение файл **gitpush.bat**.
2. Откройте программу MobaXterm. Подключитесь к кластеру. Откройте операционную оболочку, выполнив в терминале команду

mc

1. Сделайте текущей папку **~\work\k60gpu**. Сверните окно оболочки (Ctrl+O). Для обновления информации в локальном репозитории на кластере выполните в терминале команду

git pull

1. Разверните окно оболочки, сделайте текущей папку lab02. Сверните окно оболочки (Ctrl+O). Для обновления локального репозитория, компилирования проекта и запуска исполняемого файла, в терминале выполните команду

bash run.sh

Сравните результаты выполнения команды с рисунком 2.3.

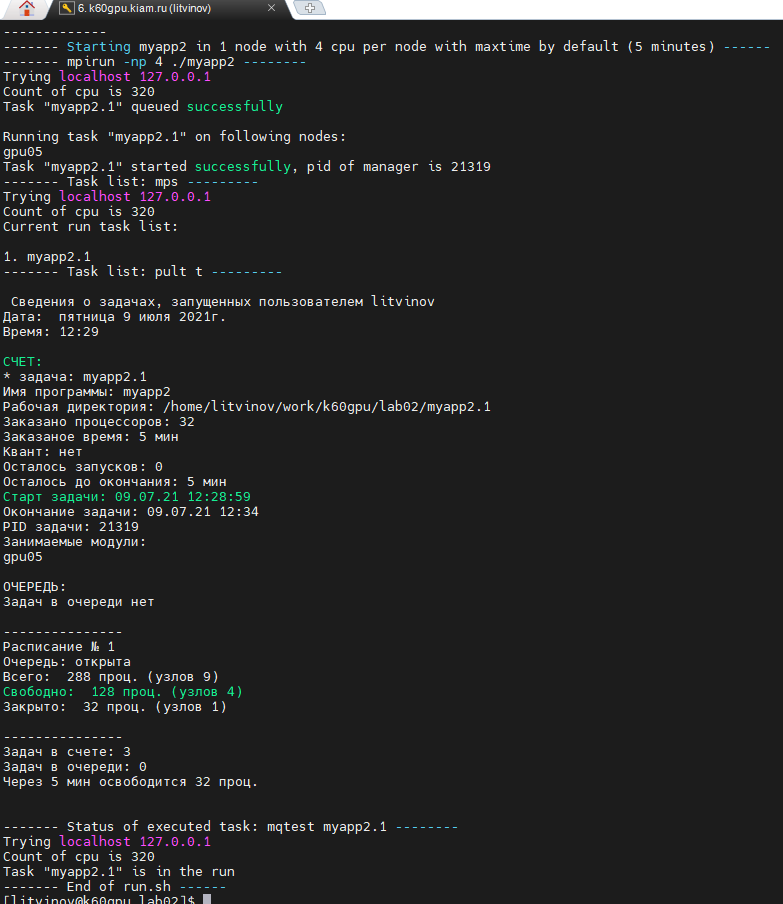


Рисунок 2.3 – Результат выполнения команды

1. Разверните окно операционной оболочки, откройте папку **lab02/myapp2.1**. Откройте файлы errors, output и проанализируйте результаты работы программы.

***Упражнение 4***

В этом упражнении Вы создадите и запустите на выполнение скрипт **clear.sh**, который удаляет созданные в результате работы папки и файлы.

1. В Visual Studio Code создайте файл **clear.sh**. Содержимое указанного файла изображено на рисунке 2.4.

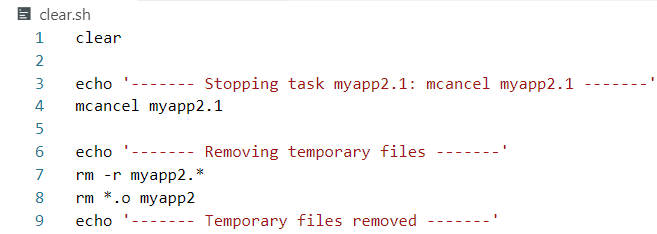


Рисунок 2.4 – Содержимое файла **clear.sh**

1. Откройте терминал в программе Visual Studio Code. Выполните команду

./gitpush.bat

В результате работы команды в удаленный репозиторий будет добавлен созданный файл **clear.sh**.

1. Перейдите в окно программы MobaXterm. Сверните окно оболочки.  
   В терминале выполните команду

git pull

В результате работы команды в локальный репозиторий на кластере будет добавлен созданный файл **clear.sh**. Разверните оболочку и просмотрите содержимое папки lab02 (рисунок 2.5).

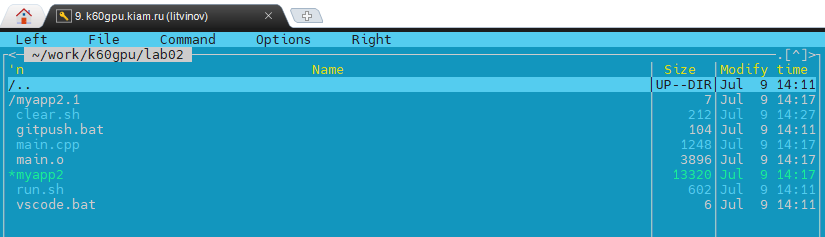


Рисунок 2.5 – Обновленное содержимое папки lab02

1. Сверните оболочку и в терминале выполните команду

bash clear.sh

В результате работы скрипта **clear.sh** будут удалены папки myapp.\* и файлы **main.o**, **myapp**. Разверните оболочку и просмотрите содержимое папки lab02 (рисунок 2.6).

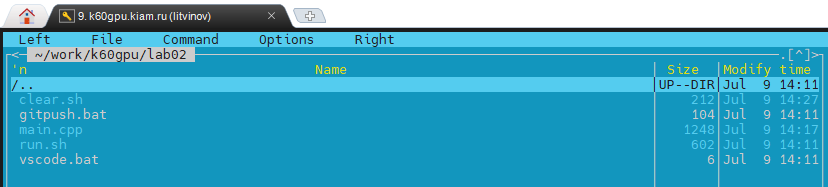


Рисунок 2.6 – Обновленное содержимое папки lab02

**2.2 Перемещение кода инициализации MPI и вывода служебной информации во внешнюю функцию** mpi\_init

***Упражнение 5***

В этом упражнении Вы создадите внешнюю функцию mpi\_init и переместите в неё код инициализации MPI и вывод служебной информации (файл **main.cpp**).

1. В программе Visual Studio Code откройте файл **main.cpp** и внесите изменения в программный код как показано на рисунке 2.7.

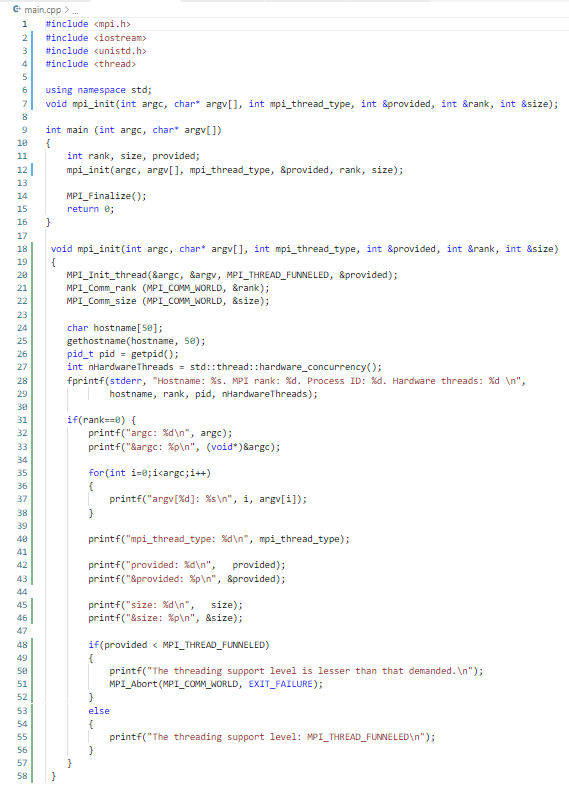


Рисунок 2.7 – Текст программы **main.cpp**

Обратите внимание на следующие изменения программного кода:

* в строке 7 появилось объявление новой функции;
* функция main изменила свой вид (строки 9-16);
* новая функция mpi\_init описана в строках 18-58.

1. Обновите информацию во всех репозиториях и запустите программу на выполнение на кластере, выполнив скрипт **run.sh**. Убедитесь, что результаты работы совпадают.
2. Удалите временные файлы на кластере, выполнив скрипт **clear.sh**.

**2.3 Реализация исходного кода в нескольких файлах**

***Упражнение 6***

В этом упражнении Вы перенесете функцию init\_mpi из файла **main.cpp** в файл **utils.cpp**.

1. В программе Visual Studio Code создайте заголовочный файл **utils.h**. Данный файл необходим для размещения объявления функции init\_mpi, а затем импортировать его для использования. Перенесите строку объявления функции init\_mpi из файла **main.cpp** (строка 7, рисунок 2.7) в файл **utils.h**. Сравните результат с рисунком 2.8. Сохраните и закройте файл.



Рисунок 2.8 – Содержимое файла **utils.h**

1. В программе Visual Studio Code создайте файл **utils.cpp**. В созданный файл перенесите функцию mpi\_init, скопируйте все используемые директивы препроцессора #include (в данном случае, все директивы), перенесите команду using, добавьте директиву #include "utils.h". Результат сравните с рисунком 2.9. Сохраните и закройте файл.

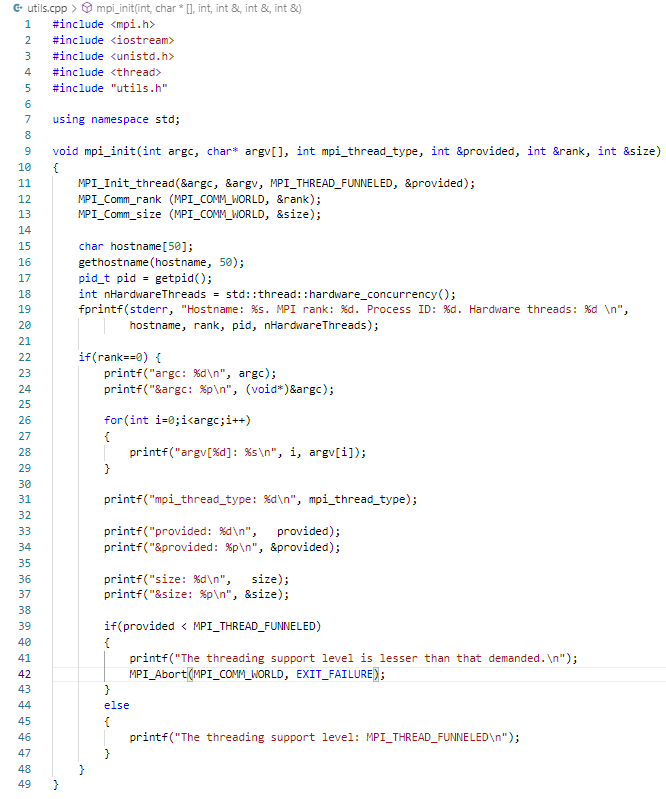


Рисунок 2.9 – Текст программы **utils.cpp**

1. Отредактируйте содержимое файла **main.cpp** как показано на рисунке 2.10.

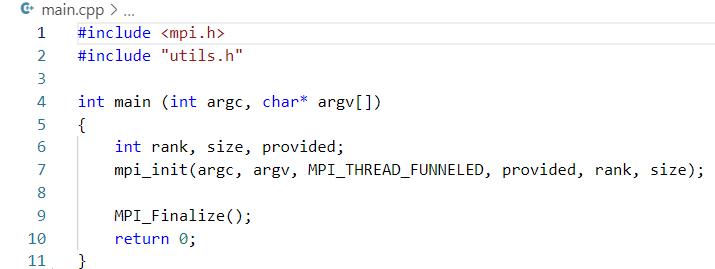


Рисунок 2.10 – Текст программы **main.cpp**

1. Отредактируйте содержимое файла **run.sh** как показано на рисунке 2.11.

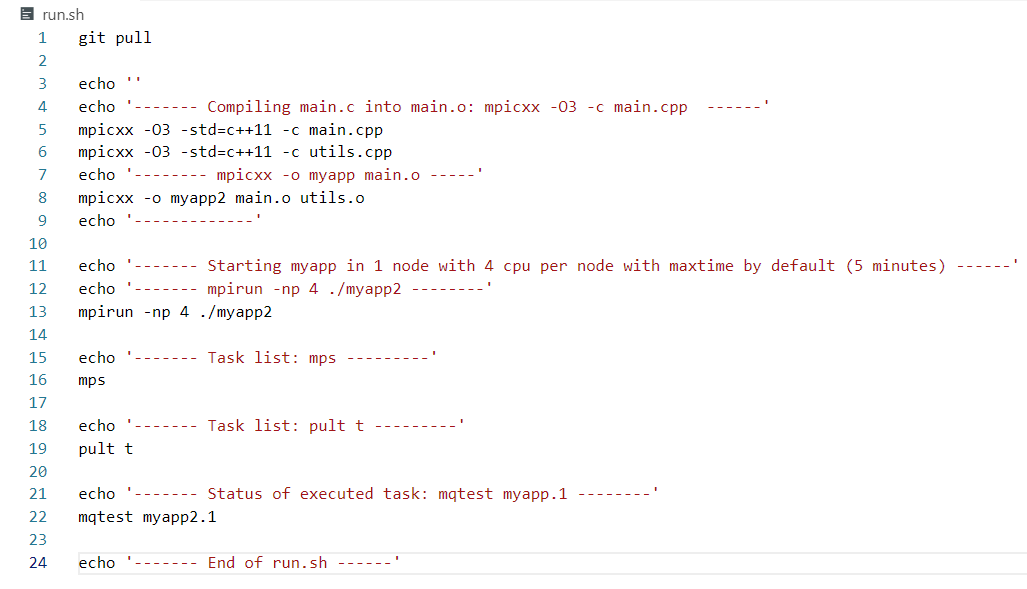


Рисунок 2.11 – Текст скрипта **run.sh**

1. Обновите информацию во всех репозиториях и запустите программу на выполнение на кластере, выполнив скрипт **run.sh**. Убедитесь, что результаты работы совпадают.
2. Удалите временные файлы на кластере, выполнив скрипт **clear.sh**.

ПРИЛОЖЕНИЕ 1

Процесс установки и регистрации ПО.

ПРИЛОЖЕНИЕ 2

**Команды Системы Управления Прохождением Пользовательских Задач (СУППЗ) (**[**https://www.kiam.ru/MVS/documents/k60/commandguide.html**](https://www.kiam.ru/MVS/documents/k60/commandguide.html) **)**

1. **Запуск на исполнение MPI-программы.**

mpirun [параметры\_mpirun...] <имя\_программы> [параметры\_программы...]

Основные параметры команды

|  |  |
| --- | --- |
| -h, --help | Интерактивная подсказка по параметрам команды. |
| -maxtime <максимальное\_время> | Максимальное время счета. От этого времени зависит положение задачи в очереди. После истечения этого времени задача принудительно заканчивается. |
| -np <число\_процессоров> | Число процессоров, требуемое программе. |
| -ppn <число\_процессоров> | Число доступных программе пользователя процессоров на одном узле. Допустимые значения от 1 до 28. Значение по умолчанию – 28. |

mrunf [-fo] <имя\_файла-паспорта>

Команда для запуска программы через паспорт задачи.

При запуске система проверяет корректность данных паспорта. При обнаружении ошибок выдается сообщение пользователю, а запуск останавливается.

Если пользователь задал в паспорте значение any для числа процессоров или номера консольного вывода, то система попросит пользователя явно указать значения этих параметров перед запуском задачи.

Задание ключа -fo (formal output) позволяет получить «формальную» выдачу. При успешном запуске (постановке в очередь) задачи, если задан ключ -fo, на стандартный вывод печатается полное имя задачи, включая полученный номер и имя логической системы.

1. **Получение информации о запущенных задачах.**

mps [имя\_задачи.номер\_задачи]

При отсутствии параметра на экран будет выдан список всех запущенных пользователем и работающих или находящихся в очереди на момент выдачи команды задач. Если задача находится в очереди, это будет отмечено словом queued рядом с именем задачи.

Параметром для команды служит имя задачи и - через точку - ее номер. При задании параметра система выдаст информацию о задаче - время старта (и завершения, если задача уже завершилась), имена узлов, на которых выполняется задача, номер процесса-менеджера задачи.

pult t

Команда выводит на стандартный вывод информацию о задачах, запущенных пользователем.

pult q

Команда выводит на стандартный вывод информацию о задачах, запущенных всеми пользователями.

1. **Завершение запущенной задачи.**

mkill [имя\_задачи.номер\_задачи]

Параметром для команды служит имя задачи и – через точку – ее номер. Данная команда допускает задание в качестве параметра маски Unix-формата с использованием символов-джокеров. По этой команде будут завершены все задачи, имена которых удовлетворят заданной маске.

mkill '\*'

Команда для завершения всех задач пользователя.

При отсутствии параметра пользователю будет выдан список всех запущенных задач и предложено ввести номер (по списку) той задачи, которую нужно завершить. Перед завершением задачи в этом случае будет задан вопрос о полном завершении задачи. Полное завершение фоновой задачи означает ее завершение без возможности повторов. В противном случае завершится только текущая итерация, и, если заказанное время не исчерпалось, задача будет запущена вновь. Это же замечание относится к команде mkill с заданным параметром – команда производит завершение лишь текущей итерации.

mterm [имя\_задачи.номер\_задачи]

Команда для безусловного полного завершения задачи.

Назначение параметра данной команды аналогично mkill, только mterm вызывает полное завершение всех задач, имена которых удовлетворят заданной маске. Вызов mterm без параметров аналогичен вызову mkill без параметров.

Следует отметить, что команды mkill и mterm лишь сообщают системе запуска о необходимости завершить задачу, но реально не дожидаются ее завершения. После выполнения данных команд задача может находиться в системе еще довольно продолжительное время, которое нужно системе для корректного освобождения вычислительных ресурсов.

mcancel <имя\_задачи.номер\_задачи>

Команда для ожидания завершения задачи.

Данная команда безусловно удаляет задачу из системы запуска, вне зависимости от того, находится она на выполнении или в очереди. Команда mcancel не закончит своего выполнения до тех пор, пока задача не покинет систему.

1. **Просмотр стандартного вывод задачи.**

mout [имя\_задачи.номер\_задачи [что\_выдавать]]

При отсутствии параметров будет предложен выбор из списка запущенных задач. При отсутствии параметра что\_выдавать будет предложен вопрос. В качестве ответа можно задать либо out, что означает выдачу стандартного вывода, либо err, что означает просмотр стандартного вывода сообщений об ошибках, либо log. В случае задания log на экран будет выведен стандартный вывод специального системного процесса-менеджера задачи.

Если Вы задали параметры команде mout, то она выдаст вам запрошенную информацию, даже если задача уже завершилась на момент вызова команды.

Прервать выдачу команды mout можно нажатием клавиш Ctrl-C. Снятие вывода НЕ означает снятие задачи – задача продолжит счет.

При запуске задачи система создаст в указанном в паспорте каталоге стандартного ввода/вывода подкаталог, имя которого будет совпадать с именем запущенной задачи (включая номер через точку). В этом подкаталоге и будут непосредственно размещены файлы стандартного ввода/вывода задачи. Файл стандартного вывода имеет имя output, стандартного вывода сообщений об ошибках – errors, стандартного ввода – input. Файл стандартного вывода процесса-менеджера имеет имя manager.log. Система сама не уничтожает эти файлы и подкаталог, зато перезаписывает их, если задача будет запущена повторно с этим же номером. После завершения задачи данные файлы полностью доступны пользователю.

1. **Получение информации о свободных процессорах.**

mfree

Команда, с помощью которой можно узнать, сколько на данный момент времени в системе имеется свободных процессоров.

1. **Команды работы с очередями.**

mps

Команда для просмотра всех запущенных задач, при этом стоящие в очереди задачи будут помечены атрибутом queued.

mqdel <имя\_задачи.номер\_задачи>

или

mcancel <имя\_задачи.номер\_задачи>

Команды для удаления задачи из очереди. Команда mcancel безусловно удаляет задачу из системы запуска, вне зависимости от того, находится она на выполнении или в очереди. Команда mcancel не закончит своего выполнения до тех пор, пока задача не покинет систему.

mqinfol

Команда для просмотра очереди.

pult s

Команда для просмотра текущего расписания.

1. **Определение статуса задачи.**

mqtest <имя\_задачи>

Команда для определения статуса задачи («в очереди», «запущена», «завершилась»).

Данная команда определяет статус задачи и печатает соответствующее сообщение пользователю. Кроме этого, при задании ключа –rcode возможно получить статус задачи через код возврата.

1. **Ожидание завершения задачи.**

mwait <имя\_задачи>

Команда позволяет дождаться завершения запущенной задачи. Данная команда «подвисает» до тех пор, пока задача полностью не завершится и не покинет систему. Для обычной задачи этот момент наступает во время завершения. Для фоновой задачи – во время завершения последнего кванта либо полного снятия со счета по команде mterm. Кроме этого, дождаться завершения задачи можно, задав ключ -wait в команде mpirun.

1. **Работа с интерактивными задачами.**

mwait <имя\_задачи>

Команда позволяет дождаться завершения запущенной задачи. Данная команда «подвисает» до тех пор, пока задача полностью не завершится и не покинет систему. Для обычной задачи этот момент наступает во время завершения. Для фоновой задачи – во время завершения последнего кванта

Литература