Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«КАЗАНСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. А.Н. ТУПОЛЕВА-КАИ»

Р.Ш. МИНЯЗЕВ

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ (MPI, OpenMP, CUDA)

Учебно-методическое пособие

Рекомендовано к изданию Учебно-методическим управлением КНИТУ-КАИ

Казань 2021

УДК 681.3
ББК 32.973.26-018.2
М 57

Рецензенты:
кафедра «Системы автоматизации и управления технологическими процессами»
(Казанский национальный исследовательский технологический университет);

кандидат технических наук, доцент М.Ю. Перухин
(Казанский национальный исследовательский технологический университет).

Минязев Р.Ш.

М 57 Параллельное программирование (MPI, OpenMP, CUDA): учебно-методи-ческое пособие / Р.Ш. Минязев. – Казань: Изд-во КНИТУ-КАИ, 2021. – 68 с.

ISBN 978-5-7579-2567-7

Содержит теоретический материал и практические инструкции по выполнению лабораторных работ в рамках дисциплины «Параллельное программирование» для студентов направления 09.03.01 «Информатика и вычислительная техника». Рассмотрены основные технологии написания параллельных программ для платформ вычислительных кластеров: MPI, OpenMP, CUDA. Представлен теоретический материал по этим технологиям и практические задания по выполнению лабораторного практикума.

Табл.1. Ил. 11. Библиогр.: 14 назв.

УДК 681.3
ББК 32.973.26-018.2

ISBN 978-5-7579-2567-7

ВВЕДЕНИЕ

В рамках учебного курса рассматриваются несколько основных технологий, используемых для написания параллельных программ [1, 2]:

MPI [3] (*Message Passing Interface, http://www.mpi-forum.org, стандарт*) – в этом случае программы запускаются на множестве (десятки, сотни) процессов – параллельное многопроцессное программирование, передача данных между процессами в ходе выполнения программы осуществляется высокоуровневыми сетевыми функциями библиотеки MPI. Такие программы исполняются на вычислительном кластере, где имеется множество независимых узлов, объединенных высокоскоростной локальной сетью (GigabitEthernet, Infiniband);

OpenMP [4] (*Open Multi-Processing, http://openmp.org/wp/, стандарт*) – программы запускаются на множестве (десятки) потоков – параллельное многопоточное программирование, обрабатываемые данные хранятся в общей памяти компьютера (куче), потоки имеют прямой доступ к данным друг друга. Такие программы ориентированы на запуск на многоядерных серверах;

CUDA [5] (*Compute Unified Device Architecture,* http://www.nvidia.ru/object/cuda-parallel-computing-ru.html, *технология компании NVIDIA*) – программы запускаются на боль-

шом множестве (сотни, тысячи, миллионы) потоков – параллельное мультипоточное программирование, один поток на CPU запускает kernel (функцию), исполняющуюся на множестве потоков на GPU (видеокарте) [6], обрабатываемые данные хранятся в памяти видеокарты. Такие программы запускаются на вычислительных серверах с видеокартами NVIDIA.

Наиболее серьезная проблема при написании параллельных программ – разработка параллельного алгоритма, предусматривающего распределение работ, данных для обработки, между множеством исполнителей (потоки, процессы). Наибольшую сложность в этом случае представляет решение задачи организации эффективного взаимодействия между исполнителями (организация обмена промежуточными данными).

1. ЗНАКОМСТВО С БИБЛИОТЕКОЙ MPI. ЗАПУСК ПРОСТЫХ MPI ПРОГРАММ

Аббревиатура MPI – название стандарта и технологии параллельного программирования одновременно (актуальная версия 2012 г. 3.0), при этом имеются различные библиотеки как платные, так и бесплатные, реализующие этот стандарт, наиболее популярная бесплатная реализация стандарта – библиотека Mpich. Она есть как под Windows, так и под Linux (https://mpich.org/, https://ru.wikipedia.org/wiki/Message_Passing_Interface). Это один из наиболее популярных инструментов для написания параллельных программ. Как правило, та или иная реализация MPI установлена на вычислительном кластере сразу после его поставки, многие промышленные программные комплексы (Ansys, FlowVision) реализованы с использованием библиотеки MPI.

Параллельные MPI программы запускаются на множестве процессов и исполняются на вычислительном кластере.

Однако ничто не мешает установить библиотеку MPI локально и запускать множество процессов на одном компьютере, разработка и отладка логики взаимодействия между множеством процессов параллельной программы происходит локально на одном компьютере, после чего отлаженную программу запускают на множестве узлов на вычислительном кластере для проведения вычислительных экспериментов или решения прикладной задачи.

Область использования MPI – решение задач моделирования газо-гидродинамики, дифференциальных уравнений в частных производных, сеточные методы вычислений и многие другие [13]. Значительная нагрузка при использований MPI ложится на локальную сеть, большая часть функции библиотеки связана с передачей данных между множеством процессов, в качестве сети зачастую используется высокоскоростная сеть Infiniband.

Библиотека MPI включает в себя средства разработки: компилятор, библиотеку, набор заголовочных файлов и утилит для исполнения параллельных программ: mpiexec – утилита для запуска исполняемого файла на множестве процессов, mpd – сервисная служба, запускающаяся на всех узлах кластера в фоновом режиме при загрузке оперативной системы и ожидающая запуска MPI процессов через утилиту mpiexec в процессе исполнения программ.

Основное преимущество использования данной библиотеки при разработке параллельных программ – легкая портируемость MPI программ для работы на различных аппаратных платформах (разных кластерах).

Кроме сетевых функций в состав библиотеки MPI входят специальные вызовы для обработки исключительных

ситуаций и отладки. Реализация MPI допускает продолжение работы приложения после возникновения ошибки. Полномасштабная отладка MPI-программ достаточно сложна вследствие одновременного исполнения нескольких программных ветвей, при этом традиционная отладка методом включения в исходный текст операторов печати затруднена вследствие смешивания в файле выдачи информации от различных ветвей MPI-программы. Одним из наиболее мощных отладчиков считается TotalView (http://www.dolphinics.com). В функции подобных отладчиков входит трассировка программы и профилирование – выдача (часто в графическом виде) информации об обменах в параллельной программе [14].

Пример простейшей MPI программы (MPI Hellow World):

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

int main(int argc, char** argv) {
  // Initialize the MPI environment
  MPI_Init(&argc, &argv);

  // Get the number of processes
  int numprocs;
  MPI_Comm_size(MPI_COMM_WORLD, & numprocs);

  // Get the rank of the process
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // Get the name of the processor
  char processor_name[MPI_MAX_PROCESSOR_NAME];
```

```
  int name_len;
  MPI_Get_processor_name(processor_name,
&name_len);

  // Print off a hello world message
  printf("Hello world from processor %s, rank %d
out of %d processors\n", processor_name, rank,
numprocs);

  // Finalize the MPI environment.
  MPI_Finalize();
}
```

Первым в каждой MPI программе должен быть вызов MPI_Init, который должен присутствовать в каждой программе MPI и предшествует всем другим вызовам функции MPI. Он устанавливает «среду» (environment) MPI. Только одно обращение к MPI_Init допускается в исполнении программы. Его аргументами являются количество аргументов командной строки процесса и собственно командная строка процесса. Почти каждая функция MPI возвращает код ошибки, который выражен либо как MPI_SUCCESS, либо как зависимый от реализации код ошибки.

Запускаемые процессы могут объединяться в различные группы (коммуникаторы), по умолчанию при запуске все процессы входят в одну общую глобальную группу (коммуникатор MPI_COMM_WORLD). При использовании функции передачи данных, как правило, в параметрах вызова функции нужно указывать используемый коммуникатор.

Функция MPI_Comm_size возвращает (в numprocs) число запущенных для данной программы процессов. Значение numprocs - это размер группы процессов, связанных с

коммуникатором MPI_COMM_WORLD. Процессы каждой группы пронумерованы целыми числами, начиная с 0, которые называются рангами (rank). Каждый процесс определяет свой номер в группе, связанной с данным коммуникатором, с помощью MPI_Comm_rank. Таким образом, каждый процесс получает одно и то же число в numprocs, но разные числа в rank. Процесс печатает свой ранг и общее количество запущенных процессов, затем процессы выполняют MPI_Finalize. Эта функция должна быть выполнена каждым процессом MPI и приводит к ликвидации «среды» MPI. Никакие вызовы MPI не могут быть осуществлены процессом после MPI_Finalize [7].

Установка MPI, настройка проекта Visual Studio

Для установки MPI на локальном компьютере нужно скачать дистрибутив под конкретную операционную систему. В данном примере скачивали дистрибутив под Windows 7 (http://www.mpich.org/static/downloads/1.4.1p1/ – библиотека mpich2). Установка вопросов не вызывает, после установки нужно добавить в переменную окружения в path путь к установленному mpich: C:\Program Files (x86)\MPICH2\bin, если он туда уже не добавился (рис. 1.1).

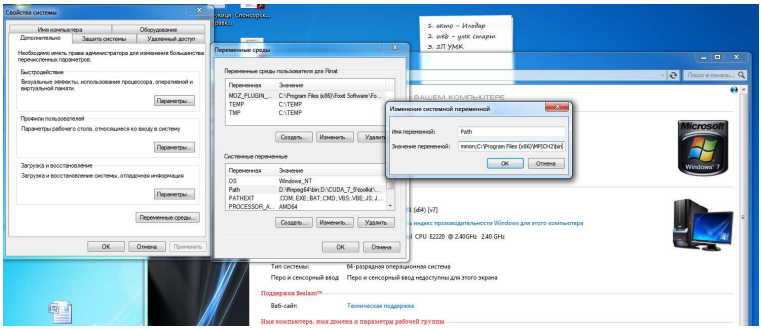

Рис. 1.1. Добавление переменной окружения Windows 7

Для настройки проекта, чтобы он видел библиотеку mpich, нужно создать обычный проект VisualStudio (рис. 1.2), при создании проекта необходимо снять галочки: предварительно скомпилированный заголовок и проверка жизненного цикла (рис. 1.3).

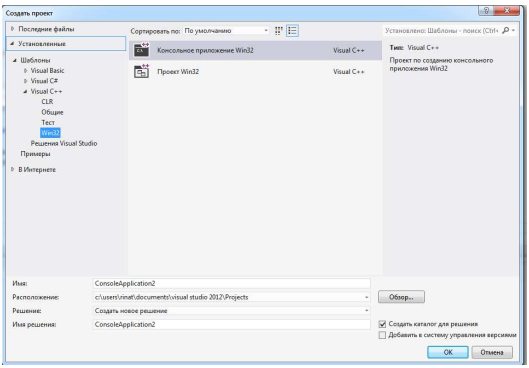

Рис. 1.2. Создание проекта VisualStudio

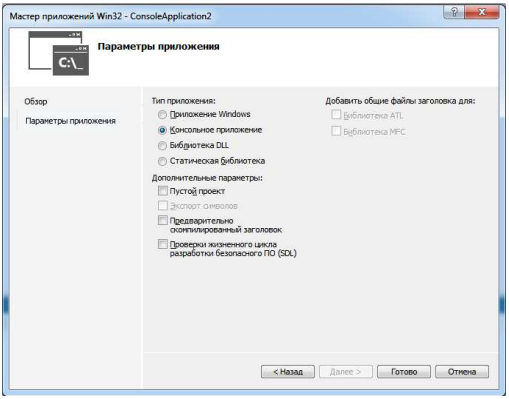

Рис. 1.3. Снятие лишних галочек в настройках проекта

Необходимо заменить сгенерированную заготовку кода программой MPI Hellow World. После чего в свойствах проекта прописываются пути для поиска и подключения заголо-

вочных файлов MPI и библиотек MPI (рис. 1.4): Проект-> свойства «название проекта» -> каталоги VC++ -> каталоги включения (C:\Program Files (x86)\MPICH2\include) и каталоги библиотек (C:\Program Files (x86)\MPICH2\lib), также необходимо прописать дополнительные библиотеки для использования (рис. 1.5): Проект-> свойства «название проекта» -> Компоновщик -> Ввод ->дополнительные зависимости -> Изменить (добавьте mpi.lib, cxx.lib, mpe.lib), после чего нажимается **Ctrl +F5** для компиляции и запуска программы. Программа должна запустится на одном процессе (рис. 1.6).

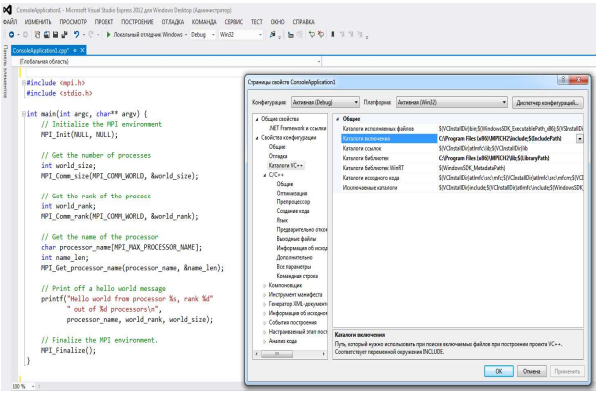

Рис. 1.4. Добавление путей для поиска заголовочных файлов и библиотек

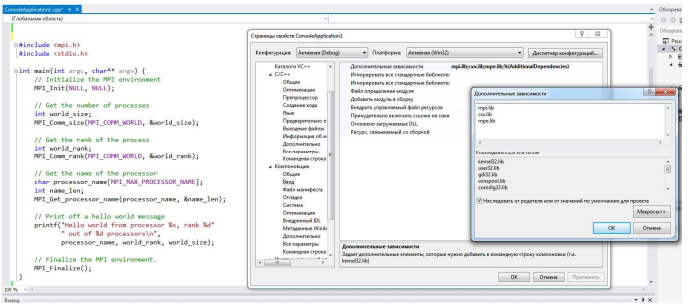

Рис. 1.5. Подключение дополнительных библиотек к линковщику проекта

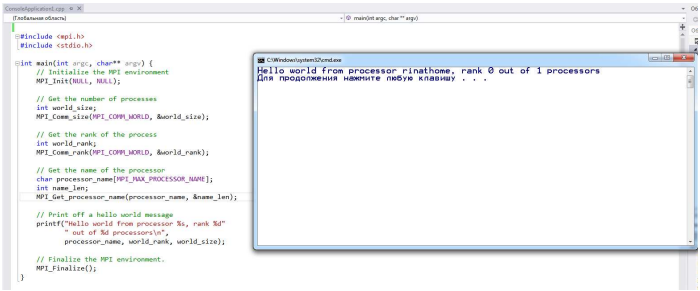

Рис. 1.6. Запуск MPI программы на одном процессе

Для запуска на множестве процессов надо запустить программу из консоли CMD Windows через утилиту mpiexec (рис. 1.7).

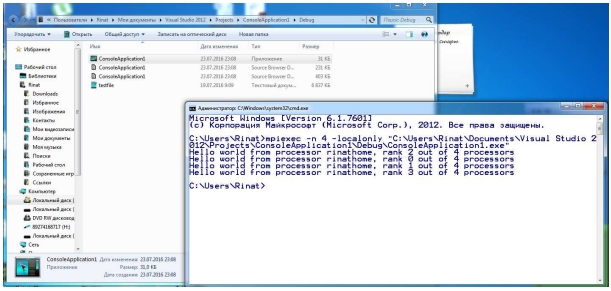

Рис. 1.7. Запуск MPI программы локально на четырех процессах

Для этого нужно открыть консоль CMD и прописать команду для запуска на четырех процессах локально **mpiexec -n 4 –localonly.** Далее нужно указать путь к скомпилированному исполняемому файлу, например: C:\Users\Rinat\ Documents\Visual Studio 2012\Projects\Console Application1\ Debug\ConsoleApplication1.exe. После чего нужно нажать Enter для запуска. Чтобы не набирать весь путь к файлу вручную, необходимо открыть папку с исполняемым файлом через проводник Windows (например: C:\Users\ Rinat\Documents\ Visual Studio 2012\Projects\Console Applic ation1\ Debug) и

просто мышкой перетащить исполняемый файл в консоль Windows, весь путь вставится автоматически. Если все хорошо, то можно увидеть, что программа запустилась на нескольких процессах.

Про детальную установку и запуск на нескольких компьютерах можно прочитать на сайте: http://iproc.ru/programming/mpich-windows/, но для выполнения лабораторных работ дома это не обязательно.

Лабораторная работа № 1

Цель работы: ознакомление с библиотекой MPI, запуск простейших MPI программ на множестве процессов локально и на вычислительном ПК-кластере, знакомство с основными функциями библиотеки MPI.

Необходимое оборудование – лабораторные работы выполняются на вычислительном ПК-кластере КНИТУ-КАИ – аудитория с 12 рабочими компьютерами под Windows 7 x 32 и файловым сервером SLES 10.0 x 32. Компьютеры соединены сетью GigabitEthernet через коммутатор dlink (ПК-кластер). Сетевое хранилище данных файлового сервера доступно компьютерам через службу Samba. На каждом узле установлена библиотека mpich 2.

Параллельные mpi программы разрабатываются на языке C\C++ в Visual Studio 2012 с подключенными необходимыми MPI библиотеками, на сервере имеется папка с настроенным проектом TestMPI.

Запуск MPI программ на ПК-кластере осуществляется с любого компьютера через консоль с помощью утилиты mpiexec, при запуске указывается число процессов, на которых нужно запустить mpi программу. Также указывается тек-

стовый файл (hosts), содержащий ip адреса узлов кластера, на которых будет запускаться MPI программа.

В принципе работы могут выполняться дома на одном компьютере с установленным mpich (множество процессов запускаются локально). В этом случае нельзя получить прирост скорости решения задачи, но можно разобраться в логике функционирования параллельных многопроцессных MPI программ, запрограммировать их и отладить.

Пример запуска MPI программы на множестве процессов локально на одном компьютере через консоль CMD:

C:\Users\Rinat>mpiexec –n 4 –localonly TestMPI.exe

Пример запуска MPI программы на множестве процессов на множестве узлов кластера через консоль CMD:

C:\Users\Rinat>mpiexec –n 4 –machinefile C:\hosts.txt TestMPI.exe

Порядок проведения работы: студент подготавливает исходные тексты MPI-программ; компилирует их в исполняемое приложение; запускает их с помощью утилиты mpiexec на множестве процессов; анализирует выходные данные программы, заполняет «Журнал исследований» по лабораторной работе; вставляет принтскрины экрана там, где требуется; заполняет таблицы с результатами вычислительных экспериментов; строит графики; заполняет графу выводы. Отвечает на контрольные вопросы, имеющиеся в некоторых заданиях.

Задание 1. Первой задачей является компиляция простейшей MPI программы HellowMPI (параллельный Hellow-World), программа запускается локально на множестве (12 и более) процессов, после этого на ПК-кластере на нескольких

узлах на множестве (12 и более) процессов. Исходник программы HellowMPI.cpp находится на сервере. После прогона программы студент отвечает на вопросы первого задания.

Задание 2. Компиляция тестового примера MPI программы, вычисляющей число Pi. Используются функции MPI_Bcast(), MPI_Reduce(), MPI_Wtime(), более подробно они будут рассмотрены позднее.

Задание 3. Изменение исходной программы таким образом, чтобы точность вычислений определял пользователь, для этого надо посмотреть закомментированные участки и там, где нужно, снять комментарии, чтобы все процессы кроме главного ждали, пока будет введена точность вычислений функцией scanf().

Задание 4. Скомпилировать и запустить программу simpleSendMPI (запускается на двух процессах), в которой один процесс генерирует данные (случайные числа генерируются функцией rand ()) и отправляет их второму процессу. Используются функции MPI_Send(), MPI_Recv(). Данные передаются как типы данных MPI: MPI_INT, MPI_FLOAT.

2. ПРИЕМ-ПЕРЕДАЧА СООБЩЕНИЙ ФУНКЦИЯМИ БИБЛИОТЕКИ MPI

Наиболее простые функции для обмена данными между процессами – функции «*точка-точка*», в таких взаимодействиях участвуют два процесса из всех, один процесс является отправителем сообщения, а другой – получателем. Процесс-отправитель вызывает функцию передачи данных (MPI_send) и указывает номер процесса-получателя (rank), а процесс-получатель должен вызвать функцию приема

(MPI_Recv) с указанием номера процесса отправителя (rank), чтобы получить данные.

Эти функции являются блокирующими, иначе говоря с синхронизацией. Они приостанавливают работу вызвавшего их процесса до тех пор, пока не будут завершены отправка или прием данных. На практике это может привести к возникновению тупиковых ситуаций – дедлоков.

Прототипы функции

Основная функция для оправки данных от одного процесса другому:

```
int  MPI_Send (void* buf, int count, MPI_Datatype
datatype, int dest, int msgtag, MPI_Comm comm);
```

buf – адрес начала буфера посылки сообщения;

count – число передаваемых в сообщении элементов (*не байт* – см. далее);

datatype – тип передаваемых элементов;

dest – номер процесса-получателя;

msgtag – идентификатор сообщения (целое число в интервале 0 до 32767, должно совпадать в парных send и recv);

comm – идентификатор группы (MPI_COMM_WORLD по умолчанию).

Функция MPI_Send осуществляет *блокирующую* посылку сообщения с идентификатором msgtag процессу с номером dest, причем сообщение состоит из count элементов типа datatype (все элементы сообщения расположены подряд в буфере buf, значение count может быть нулем). Тип передаваемых элементов datatype должен указываться с помощью

предопределенных констант типа. В MPI не используются привычные для C типы данных (int, char и др.), вместо них удобно применять определенные для данной платформы константы MPI_INT, MPI_CHAR и другие (таблица).

Предопределенные в MPI константы типов данных

Константы MPL	Соответствующий тип в C
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_SHORT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_SHORT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_UNSIGNED_CHAR	unsigned char
MPI_CHAR	signed char

Основная функция для приема данных одним процессом от другого:

```
int MPI_Recv (void* buf, int count, MPI_Datatype
datatype, int source, int msgtag, MPI_Comm comm,
MPI_Status *status);
```

buf – адрес начала буфера приема сообщения (возвращаемое значение);

count – максимальное число элементов в принимаемом сообщении;

datatyp – тип элементов принимаемого сообщения;

source – номер процесса-отправителя (если неизвестно от кого придут данные, указывают константу MPI_ANY_SOURCE);

msgtag – идентификатор принимаемого сообщения (целое число в интервале 0 до 32767 должно совпадать в парных send и recv), если заранее его не знают указывают константу MPI_ANY_TAG;

comm – идентификатор группы;

status – структура хранит информацию о том, как завершилась работа функции, была ли ошибка, от кого фактически пришли данные (rank, tag) и в каком объеме (возвращаемое значение).

Функция MPI_Recv осуществляет прием сообщения с идентификатором msgtag от процесса source с блокировкой (блокировка гарантирует, что после возврата из функции все элементы сообщения приняты и расположены в буфере buf). Число элементов в принимаемом сообщении *не должно* превосходить значения count, если число принятых элементов меньше count, то гарантируется, что в буфере buf изменятся только элементы, соответствующие элементам принятого сообщения. Если процесс посылает два сообщения другому процессу и оба сообщения соответствуют одному и тому же вызову MPI_Recv, первым будет принято то сообщение, которое было отправлено раньше [1].

С помощью пары функций MPI_Send/MPI_Recv осуществляется надежная (но не слишком эффективная) передача данных между процессами. Однако в некоторых случаях (например, когда принимающая сторона ожидает приема сообщений, но априори не знает длины и их тип) удобно использовать блокирующую функцию **MPI_Probe**, позволяющую определить характеристики сообщения *до того*, как оно будет помещено в приемный пользовательский буфер (гарантируется, что следующая вызванная функция MPI_Recv прочитает именно протестированное MPI_Probe сообщение):

```
int MPI_Probe (int source, int msgtag, MPI_Comm
comm, MPI_Status *status);
```

source – номер процесса-отправителя (или MPI_ANY_SOURCE);

msgtag – идентификатор ожидаемого сообщения (или MPI_ANY_TAG);

comm – идентификатор группы;

status – параметры обнаруженного сообщения (возвращаемое значение).

Очень часто функции MPI_Send/MPI_Recv используются совместно и именно в таком порядке, поэтому в MPI специально введены две функции, осуществляющие одновременно посылку одних данных и прием других. Первая из них – **MPI_Sendrecv**, у нее первые пять формальных параметров такие же, как у MPI_Send, остальные семь параметров аналогичны MPI_Recv. Следует учесть, что:

– как при приеме, так и при передаче используется один и тот же коммуникатор;

– порядок приема и передачи данных MPI_Sendrecv выбирает автоматически; при этом гарантировано отсутствие deadlock'а;

– MPI_Sendrecv совместима с MPI_Send и MPI_Recv.

Операция **send-receive** комбинирует в одном обращении посылку сообщения одному получателю и прием сообщения от другого отправителя. Буфера отправителя и получателя должны быть разделены и могут иметь различную длину и типы данных. Получателем и отправителем может быть тот же самый процесс. Эта операция весьма полезна для выполнения сдвига данных по цепи процессов.

Функция **MPI_Sendrecv_replace** помимо общего коммуникатора использует еще и общий для приема-передачи буфер. Применять MPI_Sendrecv_replace удобно с учетом условий:

– принимаемые данные должны быть заведомо *не длиннее* отправляемых;

– принимаемые и отправляемые данные должны иметь одинаковый тип;

– принимаемые данные записываются на место отправляемых;

– MPI_Sendrecv_replace так же гарантированно не вызывает deadlock'а.

Функция MPI_SENDRECV_REPLACE выполняет блокирующие передачи и приемы. Тот же самый буфер используется для отправки и получения, так что посланное сообщение замещается полученным.

Имеются следующие модификации процедуры передачи данных с блокировкой MPI_Send:

MPI_Bsend – передача сообщения *с буферизацией*. Если прием посылаемого сообщения еще не был инициализирован процессом-получателем, то сообщение будет записано в специальный буфер, и произойдет немедленный возврат из процедуры. Выполнение этой процедуры никоим образом не зависит от соответствующего вызова процедуры приема сообщения. Тем не менее процедура может вернуть код ошибки, если места под буфер недостаточно (о выделении места для буферизации должен озаботиться пользователь);

MPI_Ssend – передача сообщения *с синхронизацией*. Выход из этой процедуры произойдет только тогда, когда прием посылаемого сообщения будет инициализирован процессом-получателем. Таким образом, завершение передачи с синхронизацией говорит не только о возможности повторного использования буфера посылки, но и о гарантированном достижении процессом-получателем точки приема сообщения в программе. Использование передачи сообщений с синхронизацией может замедлить выполнение программы, но

позволяет избежать наличия в системе большого количества не принятых буферизованных сообщений;

MPI_Rsend – передача сообщения *по готовности.* Этой процедурой можно пользоваться только тогда, когда процесс-получатель уже инициировал прием сообщения. В противном случае вызов процедуры является ошибочным и результат ее выполнения не определен. Гарантировать инициализацию приема сообщения перед вызовом процедуры MPI_Rsend можно с помощью операций, осуществляющих явную или неявную синхронизацию процессов (например, MPI_Barrier или MPI_Ssend). Во многих реализациях процедура MPI_Rsend сокращает протокол взаимодействия между отправителем и получателем, уменьшая накладные расходы на организацию передачи данных.

В MPI предусмотрен набор процедур для осуществления *асинхронной передачи данных.* В отличие от блокирующих процедур, возврат из процедур данной группы происходит сразу после вызова без какой-либо остановки работы процессов, на фоне дальнейшего выполнения программы одновременно происходит и обработка асинхронно запущенной операции. Эта возможность исключительно полезна для создания эффективных программ – во многих случаях совершенно не обязательно дожидаться окончания посылки сообщения для выполнения последующих вычислений.

Асинхронным аналогом процедуры MPI_Send является **MPI_Isend,** для MPI_Recv, соответственно **MPI_Irecv**. Аналогично трем модификациям процедуры MPI_Send предусмотрены три дополнительных варианта процедуры MPI_Isend:

MPI_Ibsend – неблокирующая передача сообщения с буферизацией;

MPI_Issend – неблокирующая передача сообщения с синхронизацией;

MPI_Irsend – неблокирующая передача сообщения по готовности.

Лабораторная работа № 2

Цель работы: ознакомление с функциями библиотеки MPI (блокирующими), предназначенными для передачи данных между процессами, взаимодействие типа точка-точка. Решение переборной задачи (брутфорс) с использованием MPI на множестве узлов кластера.

Порядок проведения работы: студент подготавливает исходные тексты MPI-программ; компилирует их в исполняемое приложение; запускает их с помощью утилиты mpiexec на множестве процессов; анализирует выходные данные программы, заполняет «Журнал исследований» по лабораторной работе; вставляет принтскрины экрана там, где требуется; заполняет таблицы с результатами вычислительных экспериментов; строит графики; заполняет графу выводы. Отвечает на контрольные вопросы, имеющиеся в некоторых заданиях.

Задание 1. Написать программу передачи сообщения по кольцу (подробнее в презентации учебного курса). Программа должна запускаться на любом числе процессов (n>1), выполняются десять циклов приема-передачи маркера по кольцу, на каждом цикле генерируется новый маркер (число в интервале от 1 до 1000). За образец берут исходник ringSendMPI (он не рабочий, в нем надо исправить ошибки). Нужно запустить программу, вставить в журнал исследований принтскрин, на котором видно, что маркер случайное число и выполняется несколько циклов с разными сгенерированными маркерами.

Задание 2. Написать *последовательную* программу, которая будет искать ключ к Игре **Saw3** (число в интервале от 1 до 1 000 000) последовательным брутфорсом. За образец берут исходник брутфорсера **check_game.cpp,** в нем перебира-

ются первые десять ключей. В данной задаче изнутри одной программы запускается другая, при этом результат работы внутренней программы записывается в файл, если файл имеет размер 8 байт (содержит текст You Win!), то значит найден правильный ключ. Необходимо оценить теоретическое время исполнения программы в секундах (нужно обратить внимание на наличие sleep), сам *ключ находить необязательно* (будет слишком долго).

Задание 3. Написать *параллельную реализацию* программы, которая будет искать ключ к Игре **Saw3** (число в интервале от 1 до 100 000) на множестве MPI процессов (нужно обратить внимание на одновременный доступ к файлу по записи). За образец следует взять исходник последовательной программы, написанной ранее. Вставьте в журнал принтскрин с кодом программы. Нужно запустить программу на разном числе процессов на нескольких машинах и *найти ключ*.

3. ПАРАЛЛЕЛЬНЫЙ ВВОД-ВЫВОД ДАННЫХ С ИСПОЛЬЗОВАНИЕМ MPI ФУНКЦИИ

При работе с файлами в MPI программе можно завести свой уникальный файл (входной/выходной) для каждого MPI процесса, добавив к названию файла номер процесса (rank), но это будет не совсем параллельный ввод/вывод. В библиотеке MPI имеется ряд функций, предназначенных для параллельного чтения из файла и параллельной записи в файл.

Сложно сказать, насколько это дает повышение производительности в реальных приложениях, эти программы запускаются из сетевого хранилища, примонтированного ко всем узлам кластера (служба samba), сетевое хранилище – это диск SATA3, пропускная способность до 600 Мбайт/с, RAID

массив не используется. Тем не менее, можно попытаться написать программы MPI, использующие эти возможности, и провести оценку производительности параллельной работы с файлами как при записи, так и при чтении. Для замеров времени нужно использовать специальные функции MPI.

Прототипы функции

Разберем наиболее важные функции MPI для работы с файлами. Коллективная функция для открытия файла всеми процессами [3]:

```
int MPI_File_open(MPI_Comm comm, char *filename,
int amode, MPI_Info info, MPI_File *fh);
```

comm – коммуникатор (дескриптор);

filename – имя открываемого файла (строка);
amode – тип доступа к файлу (целое);
info – информационный объект (дескриптор);
fh – новый дескриптор файла (дескриптор).

Коллективная подпрограмма MPI_FILE_OPEN открывает файл с именем filename для всех процессов из группы коммуникатора comm. В ней все процессы должны обеспечивать одинаковое значение amode и имена файлов, указывающие на один и тот же файл. Значения info могут быть различны. Ошибки в MPI_FILE_OPEN генерируются при помощи стандартного дескриптора ошибок работы с файлами. Процесс может открыть файл независимо от других процессов, используя коммуникатор MPI_COMM_ SELF (в этом случае каждый процесс открывает свой персональный файл для работы). Возвращаемый дескриптор файла fh может быть использован для доступа до тех пор, пока файл не закроется при помощи MPI_FILE_CLOSE. Перед вызовом MPI_FINALIZE пользователь обязан закрыть (посредством MPI_FILE_ CLOSE) все файлы, открытые с помощью MPI_ FILE_OPEN.

Поддерживаются следующие типы доступа (задаваемые в amode, получаемом при применении *OR* к следующим целым константам):

MPI_MODE_RDONLY – только чтение;

MPI_MODE_RDWR – чтение и запись;

MPI_MODE_WRONLY – только запись;

MPI_MODE_CREATE – создать файл, если он не существует;

MPI_MODE_EXCL – ошибка, если создаваемый файл уже существует;

MPI_MODE_DELETE_ON_CLOSE – удалять файл при закрытии;

MPI_MODE_UNIQUE_OPEN – файл не будет параллельно открыт где-либо еще;

MPI_MODE_SEQUENTIAL – файл будет доступен лишь последовательно;

MPI_MODE_APPEND – установить начальную позицию всех файловых указателей на конец файла.

Изначально все процессы просматривают файл как линейный поток байтов, и каждый процесс просматривает данные в своем собственном представлении (не производится никаких преобразований представления). Файлы *POSIX* являются линейными потоками байтов в родном представлении. Вид файла может быть изменен посредством подпрограммы MPI_FILE_SET_VIEW.

Коллективная функция для установки вида файла, вид (file view) – набор данных, видимый и доступный из открытого файла как упорядоченный набор е-типов. Каждый процесс имеет свой вид файла, определенный тремя параметрами: смещением, е-типом, файловым типом. Шаблон, описанный

в файловом типе, повторяется, начиная со смещения. Смещение – это позиция в файле относительно текущего вида, представленная как число е-типов. «Дыры» в файловом типе вида пропускаются при подсчете номера этой позиции. Нулевое смещение – это позиция первого видимого е-типа в виде (после пропуска смещения и начальных «дыр» в виде).

```
int MPI_File_set_view (MPI_File fh, MPI_Offset
disp,MPI_Datatype etype, MPI_Datatype filetype,
char *datarep, MPI_Info info) [10];
```

INOUT fh –дескриптор файла (дескриптор);

IN disp – смещение (целое);

IN etype – элементарный тип данных (дескриптор);

IN filetype – тип файла (дескриптор);

IN datarep – представление данных (строка);

IN info – информационный объект (дескриптор).

Подпрограмма MPI_FILE_SET_VIEW изменяет вид данных файла для процесса. Начало вида устанавливается в disp; тип данных устанавливается в etype; распределение данных по процессам в filetype и представление данных устанавливается в datarep. Кроме того, MPI_FILE_SET_VIEW сбрасывает все индивидуальные и общие файловые указатели в 0. MPI_FILE_SET_VIEW – коллективная; значения datarep и размеры е-типов в представлении данных должны совпадать во всех процессах в группе; значения disp, filetype и info могут быть различны. Типы данных, передаваемые в etype и filetype, должны согласовываться.

Е-тип всегда определяет расположение данных в файле. Если etype – это портируемый тип данных, размер е-типа вычисляется масштабированием какого-либо смещения в типе данных до соответствия представлению данных в файле. Если

etype не является портируемым типом данных, то при вычислении размера е-типа масштабирование не производится. Пользователь должен быть осторожен при использовании непортируемых е-типов в гетерогенных средах. Если при открытии файла был определен тип доступа MPI_MODE_SEQUENTIAL, специальное смещение MPI_DISPLACEMENT_CURRENT должно передаваться в качестве disp. Это установит смещение в текущую позицию общего файлового указателя.

Коллективная функция для чтения из открытого файла всеми процессами, читает файл, используя индивидуальный файловый указатель.

```
int MPI_File_read(MPI_File fh, void *buf, int
count, MPI_Datatype data-type, MPI_Status
*status);
```

INOUT fh –дескриптор файла (дескриптор);

OUT buf – начальный адрес буфера (выбор);

N count – количество элементов в буфере (целое);

IN datatype – тип данных каждого элемента буфера (дескриптор);

OUT status – объект состояния (Status).

Коллективная функция для записи в открытый файл всеми процессами, используя индивидуальный файловый указатель.

```
int MPI_File_write(MPI_File fh, ROMIO_CONST void
*buf, int count, MPI_Datatype datatype,
MPI_Status *status);
```

INOUT fh– дескриптор файла (дескриптор);

IN buf – начальный адрес буфера (выбор);

IN count – количество элементов в буфере (целое);

IN datatype – тип данных каждого элемента буфера (дескриптор);

OUT status – объект состояния (Status).

Лабораторная работа № 3

Цель работы: ознакомление с функциями MPI, предназначенными для параллельного чтения/записи в файл. Написание программы, находящей секретный пароль по известному хэшу md5

Порядок проведения работы – студент подготавливает исходные тексты MPI-программ; компилирует их в исполняемое приложение; запускает с помощью утилиты mpiexec на множестве процессов; анализирует выходные данные программы; заполняет «Журнал исследований» по лабораторной работе; вставляет принтскрины экрана там, где требуется; заполняет таблицы с результатами вычислительных экспериментов; строит графики; заполняет графу выводы. Отвечает на контрольные вопросы, имеющиеся в некоторых заданиях.

Задание 1. Скомпилировать и запустить программу md5.cpp, которая находит хеш md5 от введенной строки (используется системная библиотека `Wincrypt.h`). Вставить в журнал принтскрин, на котором видно, что вычислили значение md5 от заданной строки.

Задание 2. У студента есть инсайдерская информация о том, сколько символов в пароле и из каких символов он состоит (шесть неповторяющихся символов: a, i, m, n, r, t). Задача написать MPI программу, которая будет генерировать все возможные комбинации символов (возможные пароли) и записывать их в разные файлы (на каждый MPI процесс свой файл). В качестве заготовки программы нужно взять GeneratePasswords.cpp. В этой программе показано, как работать со

множеством файлов в MPI процессах (нужно обратить внимание, что в примере производится запись данных типа int, при этом фактически числа воспринимаются как коды символов, у студентов будут наборы символов char фиксированной длины шесть). Благодаря параметру MPI_COMM_SELF каждый процесс открывает свой файл.

Необходимо рассчитать сколько всего будет паролей; вставить в отчет формулу для подсчета и результат; запустить свою программу, сгенерировать все комбинации, это понадобится для следующего задания. Нужно вставить принтскрин, наиболее полно показывающий проделанную студентом работу и результат.

Задание 3. Объединить все файлы, найденные на предыдущем этапе, в один. Есть хеш значение пароля (E964 9925BF97756FA03D6423B04D6255), нужно написать MPI программу, которая будет параллельно считывать из файла все сгенерированные пароли, вычислять их значение md5 и сравнивать с исходным. В качестве заготовки программы необходимо взять Find_MD5_MPI.cpp. Следует обратить внимание, здесь каждый процесс проверяет первые десять паролей и выходит из программы, нулевой процесс печатает NO, остальные не печатают ничего, чтобы не загромождать вывод. Нужно найти пароль, вставить в отчет принтскрин, наиболее полно показывающий проделанную работу и результат.

4. ИСПОЛЬЗОВАНИЕ КОЛЛЕКТИВНЫХ ФУНКЦИЙ ПЕРЕДАЧИ СООБЩЕНИЙ MPI

Наиболее простые функции для обмена данными между процессами – функции «*точка-точка*», в таких взаимодействи-

ях участвуют два процесса из всех, один процесс является отправителем сообщения, а другой – получателем. Однако MPI поддерживает более сложные типы взаимодействий, которые предполагают участие нескольких (возможно более двух) процессов. Такие операции называют коллективными. Коллективные операции широко используются при написании программ MPI. Это связано не только с естественностью таких операций (часто используемые алгоритмы предполагают рассылку данных всем вычислительным процессам, или наоборот, сборку рассчитанных данных на корневом процессе), но и высокой эффективностью таких операций.

Передача данных от одного процесса всем, широковещательная рассылка: при программировании параллельных задач часто возникает необходимость разослать какую-то порцию данных *всем процессам сразу*. Очевидно, что для решения этой задачи можно воспользоваться рассмотренными ранее операциями двупроцессного обмена.

```
MPI_Comm_size(MPI_COMM_WORLD,&size);
for (i=1; i<size; i++)

  PI_Send(&buf,buflen,MPI_INT,i,0,MPI_COMM_WORLD);
        ...
```

Однако такое решение неэффективно вследствие значительных затрат на синхронизацию процессов. Поэтому в MPI появилась специальная операция – операция широковещательной рассылки:

```
int MPI_Bcast (void *buffer, int count,
MPI_Datatype datatype, int root, MPI_Comm comm).
```

Операция предполагает рассылку данных из буфера buffer, содержащего count элементов типа datatype с процесса, имеющего номер root, всем процессам, входящим в коммуникатор comm.

***Передача данных от всех процессов одному. Операция редукции [9]*:** обратная по отношению к широковещательной рассылке операция – *операция сбора данных*, или *редукция*. Операция редукции позволяет, собрав на одном из узлов данные, посланные остальными узлами, выполнить над ними какую-либо из групповых операций – типа сложения, поиска максимума, минимума, среднего значения и т.д.:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm);
```

sendbuf – данные, которые посылаются на каждом из процессов;

recvbuf – адрес буфера, в который будет помещен результат;

count – количество элементов в буфере sendbuf;

datatype – тип элементов в буфере sendbuf;

op – коллективная операция, которая выполняется над данными;

root– ранг процесса, на котором должен быть собран результат;

comm – коммуникатор, в рамках которого выполняется операция.

В качестве коллективных операций можно использовать предопределенные в MPI операции, такие как MPI_SUM, MPI_MIN, MPI_MAX и другие (всего их 12). Кроме того, имеется возможность определить свою операцию.

Распределение и сбор данных

При программировании часто возникает задача распределения массива данных по процессам некоторыми регулярными «кусками». Например, распределение матрицы, нарезанной вертикальными лентами. Возникает и обратная

задача – сбор на некотором выделенном процессе некоторого набора данных, распределенного по всем процессам.

Распределение и сбор данных осуществляется с помощью вызовов процедур MPI_Scatter и MPI_Gather:

```
int MPI_Scatter(void* sendbuf, int sentcount,
MPI_Datatype senddatatype, void* recbuf, int rec-
count, MPI_Datatype recdatatype,int root,MPI_Comm
comm);
```

sendbuf – адрес буфера для передачи данных;

sentcount – количество элементов, передаваемых на каждый процесс (общее количество элементов в буфере равно произведению sentcount на количество процессов в коммуникаторе);

senddatatype – тип передаваемых данных;

recbuf – буфер для приема данных;

reccount – размер буфера recbuf;

recdatatype – тип данных для приемки;

root – ранг процесса, с которого рассылаются данные;

comm – коммуникатор.

При вызове этой процедуры произойдет следующее. Процесс с рангом root произведет передачу данных всем другим процессам в коммуникаторе. Каждому процессу будет отправлено sentcount элементов. Процесс с рангом 0 получит порцию из sendbuf, начиная с нулевого и заканчивая sentcount-1 элементом. Процесс с рангом 1 получит порцию, начиная с sentcount, заканчивая 2* sentcount-1 и т.д.

Подпрограмма MPI_Gather собирает данные от остальных процессов.

```
int MPI_Gather(void* sendbuf, int sentcount,
MPI_Datatype senddatatype, void* recbuf, int rec-
```

```
count, MPI_Datatype recdatatype,int root,MPI_Comm
comm);
```

sendbuf – адрес буфера для передачи данных;

sentcount– количество элементов, передаваемое на главный процесс;

senddatatype – тип передаваемых данных;

recbuf – буфер для приема данных;

reccount – размер буфера recbuf;

recdatatype – тип данных для приемки;

root – ранг процесса, на котором собираются данные;

comm – коммуникатор.

Посредством MPI_Gather каждый процесс в коммуникаторе передает данные из буфера sendbuf на процесс с рангом root. Этот «ведущий» процесс осуществляет склейку поступающих данных в буфере recbuf. Склейка данных осуществляется линейно, положение пришедшего фрагмента данных определяется рангом процесса, его приславшего. В целом процедура MPI_Gather обратна по своему действию процедуре MPI_Scatter.

Следует заметить, что при использовании MPI_Gather сборка осуществляется только на одном процессе. Во всех остальных процессах заполнение буфера recbuf не определено. Для некоторых задач необходимо, чтобы данные, рассчитанные на каждом из процессов, были собраны в единый объект опять же на каждом процессе. В таком случае вместо функции MPI_Gather следует использовать функцию MPI_Allgather. При использовании функции MPI_Allgather на всех процессах в буфере recbuf будут собраны одинаковые данные – «большой» объект, полученный как объединение фрагментов, переданных с каждого из процессов.

Лабораторная работа № 4

Цель работы: ознакомление с коллективными (групповыми) функциями библиотеки MPI, предназначенными для обмена данными между множеством процессов.

Порядок проведения работы – студент подготавливает исходные тексты MPI-программ; компилирует их в исполняемое приложение; запускает их с помощью утилиты mpiexec на множестве процессов; анализирует выходные данные программы; заполняет «Журнал исследований» по лабораторной работе; вставляет принтскрины экрана там, где требуется; заполняет таблицы с результатами вычислительных экспериментов; строит графики; заполняет графу выводы. Отвечает на контрольные вопросы, имеющиеся в некоторых заданиях.

Задание 1. Скомпилировать и запустить программу **“laba4_example.cpp”**. Ответить на контрольные вопросы.

Задание 2. В исходном тексте программы **“laba4_task1.cpp”** пропущены вызовы процедур широковещательной рассылки (mpi_bcast). Добавить эти вызовы, откомпилировать и запустить программу. Написать в журнал добавленные строки кода, объяснить их назначение. Вставить в журнал принтскрин с результатами работы программы, запуск производится локально на пяти процессах.

Задание 3. Написать программу на основании имеющегося примера **“laba4_task2.cpp”**, реализующую новый оператор для функции редукции. Разрабатываемый оператор должен реализовывать вычисление минимального элемента (ручная реализация **MPI_MIN**). Вставить в журнал принтскрин с результатом работы программы, запуск производится локально на пяти процессах.

Задание 4. Доработать программу, написанную в задании 3. В примере **“laba4_task2.cpp”** каждый процесс посылает в процесс **root** *только один элемент* **(int),** необходимо переделать программу, чтобы это был *массив из трех элементов типа* **float** (массив инициализировать вручную с помощью внешней функции – void MassivInit(…)). Вставить в журнал принтскрин, на котором видны результаты работы программы, запуск производится локально на четырех процессах.

Задание 5. Доработать программу, написанную в задании 4. Необходимо написать программу на основании имеющегося примера, реализующую новый оператор для функции редукции. Разрабатываемый оператор должен реализовывать вычисление среднего арифметического **(MPI_AVG)**. Вставить в журнал принтскрин, на котором видны результаты работы программы, запуск производится локально на четырех процессах.

5. ИСПОЛЬЗОВАНИЕ ВИРТУАЛЬНЫХ ТОПОЛОГИЙ MPI ПРИ РЕШЕНИИ ЗАДАЧ

Топология – дополнительный механизм MPI, который позволяет устанавливать дополнительную систему адресации для процессов. Топология в MPI является логической и никак не связана с топологией физической среды передачи данных. Введение в MPI поддержки топологий связано с тем, что большое число прикладных алгоритмов устроено таким образом, что процессы оказываются упорядоченными в соответствии с некоторой топологией. В MPI поддерживаются два вида топологий – декартова топология и граф. *Декартова*

топология – прямоугольная решетка произвольной размерности и *топология графа* (в этом случае процессы соединены между собой ребрами, показывающими направление обмена) [11].

Над топологиями можно выполнять различные *операции*. Декартовы решетки можно расщеплять на гиперплоскости, удаляя некоторые измерения. Данные можно сдвигать вдоль выбранного измерения декартовой решетки. Сдвигом в этом случае называют пересылку данных между процессами вдоль определенного измерения. Вдоль избранного измерения могут быть организованы коллективные обмены (рис. 5.1).

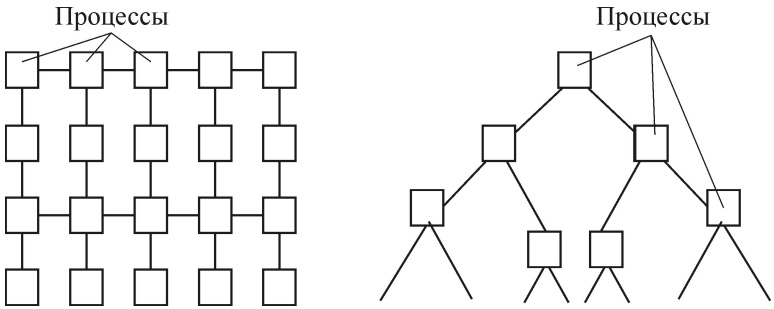

Рис. 5.1. Топологии декартова и граф в MPI

Для того чтобы связать структуру декартовой решетки с коммуникатором MPI_COMM_WORLD, необходимо задать следующие параметры:

размерность решетки (значение 2, например, соответствует плоской решетке);

размер решетки вдоль каждого измерения (размеры {10, 5}, например, соответствуют прямоугольной плоской решетке, протяженность которой вдоль оси X составляет десять узлов-процессов, а вдоль оси Y – пять узлов);

граничные условия вдоль каждого измерения (решетка может быть периодической, если процессы, находящиеся на противоположных концах ряда, взаимодействуют между собой).

MPI дает возможность системе оптимизировать *отображение* виртуальной топологии процессов на физическую с помощью изменения порядка нумерации процессов в группе.

Решетки

Декартовы топологии часто применяются при решении прикладных задач. Известно большое количество алгоритмов, в которых используются «сетки». Один из наиболее широко представленных классов таких задач - сеточные методы решения дифференциальных уравнений в частных производных.

Чтобы создать топологию вида «решетка», используется функция:

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims,
int*dims, int *periods, int reoreder, MPI_Comm
grid_comm);
```

oldcomm – исходный коммуникатор;

ndims – размерность декартовой решетки;

dims – целочисленный массив длины dims, задает количество узлов в каждом измерении;

periods – массив длины ndims, определяет является ли измерение замкнутым;

reoreder – разрешено или нет системе менять нумерацию процессов.

Операция создания топологии является коллективной операцией – ее должны выполнить все процессы коммуникатора. После создания декартовой топологии для определения декартовых координат процесса по его рангу можно воспользоваться функцией:

```
int  MPI_Card_coords (MPI_Comm comm,int rank,int
maxdims,int *coords);
```

comm – коммуникатор;

rank – ранг процесса, для которого нужно определить координаты;

maxdims – длина массива coords;

cords – возвращаемые функцией декартовы координаты процесса.

Функция MPI_Cart_rank наоборот, возвращает ранг процесса по его декартовым координатам.

```
int  MPI_Cart_rank  (MPI_Comm  comm,  int*coords,
int*rank);
```

comm – коммуникатор;

coords – декартовы координаты процесса;

rank– возвращаемый функцией ранг процесса.

Сдвиг в декартовых координатах

Если используется декартова топология, то операцию MPI_SENDRECV можно выполнить путем сдвига данных вдоль направления координаты. В качестве входного параметра MPI_SENDRECV используется номер процесса-отправителя для приема и номер процесса-получателя – для передачи. Если функция MPI_CART_SHIFT выполняется для декартовой группы процессов, то она передает вызывающему процессу эти номера, которые затем могут быть использованы для MPI_SENDRECV. Пользователь определяет направление координаты и величину шага (положительный или отрицательный). Эта функция является локальной, ее вызывает каждый процесс:

```
int  MPI_Cart_shift(MPI_Comm  comm,  int  direction,
int disp, int *rank_source, int *rank_dest);
```

IN comm – коммуникатор с декартовой топологией (дескриптор);

IN direction – координата сдвига (целое);

IN disp – направление смещения (> 0: смещение вверх, < 0: смещение вниз) (целое);

OUT rank_source – номер процесса-отправителя (целое);

OUT rank_dest – номер процесса-получателя (целое).

Аргумент direction указывает размерность, т.е. координату, по которой сдвигаются данные. Координаты маркируются от 0 до ndims-1, где ndims – число размерностей.

В зависимости от периодичности декартовой группы в указанном направлении координаты MPI_CART_SHIFT указывает признаки для кольцевого сдвига или для сдвига без переноса. В случае сдвига без переноса в rank_source или rank_dest может быть возвращено значение MPI_PROC_NULL для указания, что процесс-отправитель или процесс-получатель при сдвиге вышли из диапазона.

Лабораторная работа № 5

Цель работы – ознакомление с виртуальными топологиями библиотеки MPI, их использование при решении задач.

Порядок проведения работы – студент подготавливает исходные тексты MPI-программ; компилирует их в исполняемое приложение; запускает их с помощью утилиты mpiexec на множестве процессов; анализирует выходные данные программы; заполняет «Журнал исследований» по лабораторной работе; вставляет принтскрины экрана там, где требуется; заполняет таблицы с результатами вычислительных экспери-

ментов; строит графики; заполняет графу выводы. Отвечает на контрольные вопросы, имеющиеся в некоторых заданиях.

Задание 1. Необходимо написать программу, моделирующую нагревание двухмерного тора размером 5x8 (40 точек), за образец нужно взять исходник **“laba5_task1.cpp”**. Нагревание идет в пяти точках в течение разных временных промежутков. Программа должна выводить температуру всех точек и среднюю температуру. Нужно скомпилировать и запустить программу на 40 процессах (можно локально, можно на нескольких машинах), заполнить таблицу и построить график *изменения минимальной t точек от времени нагревания***.** График (ось *X* –время нагрева; ось *Y* – время холодных точек).

Задание 2. Написать параллельную программу умножения матрицы на вектор на кольце, используя пример. В примере **“laba5_task2.cpp”** предполагается, что количество строк матрицы *A* и вектора *B* делятся без остатка на количество компьютеров в системе. Исходная матрица A размером 20x20 делится без остатка на количество компьютеров, в данном случае на 4. Векторы *B* и *C* тоже делятся на 4 без остатка.

Программу нужно запустить на четырех компьютерах. Необходимо правильно указать пропущенные параметры вызова функции:

MPI_Cart_create(..., &ring);

MPI_Sendrecv_replace(...).

Необходимо вставить в журнал принтскрин, на котором видны результаты работы программы, запускать локально на четырех процессах. Следует разобраться, где происходит инициализация матрицы и вектора данными.

6. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ С ПРИМЕНЕНИЕМ ТЕХНОЛОГИИ OPENMP

Популярным средством программирования для компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, является технология OpenMP. За основу берётся последовательная программа, а для создания её параллельной многопоточной версии пользователю предоставляется набор директив, функций и переменных окружения. Предполагается, что создаваемая параллельная программа будет переносимой между различными компьютерами с разделяемой памятью, поддерживающими OpenMP API. Технология OpenMP нацелена на то, чтобы пользователь имел один вариант программы для параллельного и последовательного выполнения.

Для использования механизмов OpenMP нужно скомпилировать программу компилятором, поддерживающим OpenMP, с указанием соответствующего ключа (например, в icc/ifort используется ключ компилятора **–openmp**, в gcc/ gfortran **– fopenmp**, Sun Studio **– xopenmp**, в Visual C++ **– /openmp**, в PGI **-mp**). Компилятор интерпретирует директивы OpenMP и создаёт параллельный код. При использовании компиляторов, не поддерживающих OpenMP, директивы OpenMP игнорируются без дополнительных сообщений. Для включения поддержки OpenMP в VisualStudio нужно поставить соответствующую опцию компиляции (рис. 6.1). Распараллеливание в OpenMP выполняется явно при помощи вставки в текст программы специальных директив, а также вызова вспомогательных функций.

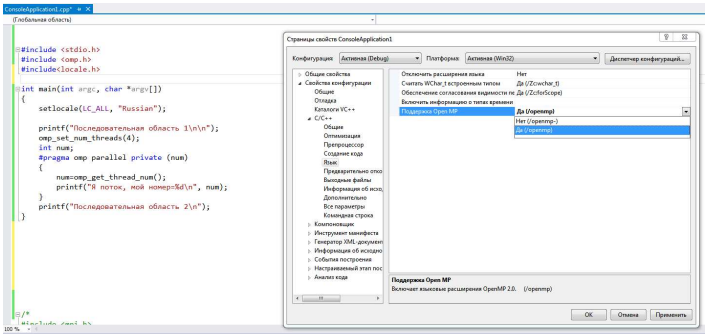

Рис. 6.1. Включение поддержки OpenMP в VisualStudio

При использовании OpenMP предполагается *SPMD*-модель *(Single Program Multiple Data)* параллельного программирования, в рамках которой для всех параллельных потоков используется один и тот же код.

Модель параллельной программы [12]

Программа начинается с последовательной области – сначала работает один поток (нить), при входе в параллельную область порождается ещё некоторое число потоков, между которыми в дальнейшем распределяются части кода. По завершении параллельной области все нити, кроме одной (нити- мастера), завершаются, и начинается последовательная область. В программе может быть любое количество параллельных и последовательных областей. Кроме того, параллельные области могут быть вложенными друг в друга. В отличие от полноценных процессов, порождение потоков является относительно быстрой операцией, поэтому частые порождения и завершения потоков не так сильно влияют на время выполнения программы.

Для написания эффективной параллельной программы необходимо, чтобы все потоки, участвующие в обработке в программе, были равномерно загружены полезной работой.

Это достигается тщательной балансировкой загрузки, для чего предназначены различные механизмы OpenMP.

Существенным моментом является также необходимость синхронизации доступа к общим данным. Само наличие данных, общих для нескольких потоков, приводит к конфликтам при одновременном несогласованном доступе. Поэтому значительная часть функциональности OpenMP предназначена для осуществления различного рода синхронизаций работающих потоков.

OpenMP не выполняет синхронизацию доступа различных потоков к одним и тем же файлам. Если это необходимо для корректности программы, пользователь должен явно применять директивы синхронизации или соответствующие библиотечные функции. При доступе каждого потока к своему файлу никакая синхронизация не требуется.

Пример простой OpenMP программы

Запуск программы происходит прямо из VisualStudio (рис. 6.2).

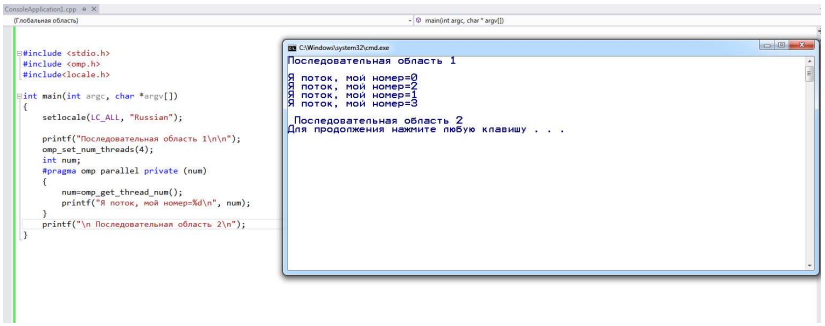

Рис. 6.2. Запуск программы на нескольких потоках openMP

```
#include <stdio.h>
#include <omp.h>
```

```
#include<locale.h>

int main(int argc, char *argv[]){
 setlocale(LC_ALL, "Russian");
 printf("Последовательная область 1\n\n");
 omp_set_num_threads(4);
 int num;
 #pragma omp parallel private (num) {
       num=omp_get_thread_num();
       printf("Я поток, мой номер=%d\n", num);
 }
 printf("\n Последовательная область 2\n");

}
```

Директивы и функции

Значительная часть функциональности OpenMP реализуется при помощи директив компилятору. Они должны быть явно вставлены пользователем, что позволит выполнять программу в параллельном режиме. Директивы OpenMP в языке Си оформляются указаниями препроцессору, начинающимися с **#pragma omp**. Ключевое слово **omp** используется для того, чтобы исключить случайные совпадения имён директив OpenMP с другими именами.

Формат директивы на Си/Си++:

#pragma omp *directive-name* [опция**[,]** опция**]...]**

Объектом действия большинства директив является один оператор или блок, перед которым расположена директива в исходном тексте программы. В OpenMP такие операторы или блоки называются ассоциированными с директивой. Ассоциированный блок должен иметь одну точку входа

в начале и одну точку выхода в конце. Порядок опций в описании директивы несущественен, в одной директиве большинство опций может встречаться несколько раз. После некоторых опций может следовать список переменных, через запятую.

Все директивы OpenMP можно разделить на три категории:

- определение параллельной области;
- распределение работы;
- синхронизация.

Каждая директива может иметь несколько дополнительных атрибутов – опций (*clause*). Отдельно специфицируются опции для назначения классов переменных, которые могут быть атрибутами различных директив.

Чтобы задействовать функции библиотеки OpenMP в программу, нужно включить заголовочный файл **omp.h**. Если в приложении используют только OpenMP-директивы, включать этот файл не требуется. Функции назначения параметров имеют приоритет над соответствующими переменными окружения, установленными в операционной системе.

Все функции, используемые в OpenMP, начинаются с префикса **omp_**. Если пользователь не будет применять в программе имён, начинающихся с такого префикса, то конфликтов с объектами OpenMP заведомо не будет. В языке Си является существенным регистр символов в названиях функций. Названия функций OpenMP записываются строчными буквами.

Для того чтобы программа, использующая функции OpenMP, могла оставаться корректной для обычного компилятора, можно прилинковать специальную библиотеку, которая определит для каждой функции соответствующую «за-

глушку» (*stub*). Например, в компиляторе Intel соответствующая библиотека подключается заданием ключа компиляции **– openmp-stubs**.

Выполнение программы

После получения выполняемого файла необходимо запустить его на требуемом количестве потоков. Для этого нужно задать количество потоков, выполняющих параллельные области программы, определив значение переменной среды ОС **OMP_NUM_THREADS,** либо указав опции в директивах, либо вызвав функцию OpenMP в коде программы перед директивой.

После запуска начинает работать один поток, а внутри параллельных областей одна и та же программа будет выполняться всем набором потоков. Стандартный вывод программы в зависимости от системы будет выдаваться на терминал или записываться в файл с предопределенным именем.

Замер времени

В OpenMP предусмотрены функции для работы с системным таймером. Функция **omp_get_wtime()** возвращает в вызвавшей нити астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом.

double omp_get_wtime(void);

Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменён за время существования потока. Таймеры разных нитей могут быть не синхронизированы и выдавать различные значения.

Функция **omp_get_wtick()** возвращает в вызвавшей нити разрешение таймера в секундах.Это время можно рассматривать как меру точности таймера.

double omp_get_wtick(void);

Пример иллюстрирует применение функций **omp_get_wtime()** и **omp_get_wtick()** для работы с таймерами в OpenMP. В данном примере производится замер начального времени, затем сразу замер конечного времени. Разность времён даёт время на замер времени. Кроме того, измеряется точность системного таймера:

```
#include "omp.h"
#include <stdio.h>
#include <windows.h>
int main() {
    double start = omp_get_wtime( );
    Sleep(1000);
    double end = omp_get_wtime( );
    double wtick = omp_get_wtick( );

    printf_s("start = %.16g\nend = %.16g\ndiff =
%.16g\n",
             start, end, end - start);

    printf_s("wtick = %.16g\n1/wtick = %.16g\n",
             wtick, 1.0 / wtick);
}
```

Лабораторная работа № 6

Цель работы: ознакомление с основами использовани я библиотеки OpenMP для написания параллельных многопоточных программ: основные директивы компиляторы, опции директив, функции библиотеки OpenMP.

Необходимое оборудование – лабораторные работы выполняются на обычном компьютере с числом ядер не менее двух (в данном случае используется Intel Core i3 с двумя физическими ядрами и поддержкой технологии HyperThreading). Параллельные многопоточные программы разрабатываются на языке C\C++ в Visual Studio 2012 с подключением необходимой опции компилятора OpenMP (см. рис. 6.1). Запуск программы происходит прямо из Visual Studio (Ctrl+F5).

Порядок проведения работы – студент подготавливает исходные тексты программ; компилирует их в исполняемое приложение; запускает их; анализирует выходные данные программы; заполняет «Журнал исследований» по лабораторной работе; вставляет принтскрины экрана там, где требуется; заполняет таблицы с результатами вычислительных экспериментов; строит графики; заполняет графу выводы. Отвечает на контрольные вопросы, имеющиеся в некоторых заданиях.

Задание 1. Скомпилировать и запустить программу Hellow_OpenMP.cpp (нужно включить поддержку OpenMP в настройках проектаVisualStudio). Необходимо вставить в журнал принтскрин, на котором видны результаты работы программы (не забыть добавить вывод своих ФИО в printf).

Задание 2. Откомпилировать программу по вычислению числа Pi (pi.cpp). Нужно разобраться, как она работает, и быть готовым ответить на вопросы по ее программному коду. Следует вставить в журнал принтскрин, на котором видны результаты работы программы (не забыть добавить вывод своих ФИО в printf).

Задание 3. Откомпилировать программу по суммированию двух векторов (**SummVectors.cpp).** Запустить программу

на разном числе потоков и проанализировать время исполнения. Нужно вставить в журнал принтскрин, на котором видны результаты работы программы (не забыть добавить вывод своих ФИО в printf).

Задание 4. Откомпилировать программу по перемножению и суммированию элементов двух векторов **ScalarMult.cpp.** Нужно изменить исходный код программы **ScalarMult.cpp** таким образом, чтобы программа исполнялась на нескольких потоках. Необходимо вставить в журнал принтскрин, на котором видны результаты работы программы (не забыть добавить вывод своих ФИО в printf).

Задание 5. Разработать программу вычисления произведения элементов массива вещественных чисел (числа генерируют случайным образом в интервале от 1 до 1000). Длина массива 800 элементов, нужно использовать оператор reduction. Следует запустить программу на разном числе потоков, оценить время работы, заполнить таблицу, построить график зависимости времени работы от числа потоков, вставить в журнал принтскрин, на котором видны результаты запуска программы для 12 потоков (не забыть добавить вывод своих ФИО в printf).

При этом можно заметить переполнение типа double на определенном этапе. Для решения этой проблемы можно использовать стороннюю библиотеку, например Thrust, но проблема в том, что OpenMP не работает с типами данных этой библиотеки. Следует посмотреть, как можно решить эту проблему.

Задание 6. Разработать программу вычисления произведения матрицы (60*60) на вектор. Элементы матрицы и

вектора – случайные числа типа int, сгенерированные в интервале от 1 до 500. Необходимо запустить программу на разном числе потоков; оценить время работы; заполнить таблицу; построить график зависимости времени работы от числа потоков; вставить в журнал принтскрин, на котором видны результаты запуска программы для 12 потоков (не забыть добавить вывод своих ФИО в printf).

7. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ С ПРИМЕНЕНИЕМ ТЕХНОЛОГИИ CUDA

Технология CUDA (Compute Unified Device Architecture) – это архитектура параллельных вычислений от NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU (Graphics Processing Unit, графических процессоров). Суть подхода в том, что при написании программы выделяется отдельная функция (kernel), которая запускается на множестве потоков, исполняемых на видеокарте [15]. Необходимо учесть, что данная технология используется только на видеокартах компании Nvidia, для программирования под AMD видеокарты нужно использовать технологию OpenCL, но ее в рамках данного курса не рассматривают.

Некоторые основные термины

В технологии CUDA применяют следующие термины:

программный интерфейс приложения – API (сокр. от англ. Application Programming Interface);

набор низкоуровневых программных интерфейсов (API) для создания игр и других высокопроизводительных мульти-

медиаприложений. Включает поддержку высокопроизводительной 2D- и 3D-графики, звука и устройств ввода – DirectX;

интерфейс вывода трехмерных примитивов (геометрических тел). Входит в DirectX – Direct3D (D3D);

спецификация, определяющая независимый от языка программирования кросс-платформенный программный интерфейс для написания приложений, использующих двухмерную и трехмерную компьютерную графику. Включает более 250 функций для рисования сложных трехмерных сцен из простых примитивов. Используется при создании видеоигр, САПР, виртуальной реальности, визуализации в научных исследованиях. На платформе Windows конкурирует с DirectX – OpenGL (от англ. Open Graphics Library, дословно – открытая графическая библиотека);

фреймворк (каркас программной системы) для написания компьютерных программ, связанных с параллельными вычислениями на различных графических (GPU) и центральных процессорах (CPU) – OpenCL (от англ. Open Computing Language, дословно – открытый язык вычислений). В фреймворк OpenCL входят язык программирования и интерфейс программирования приложений (API). OpenCL обеспечивает параллелизм на уровне инструкций и на уровне данных и является реализацией техники GPGPU;

техника использования графического процессора видеокарты (внутри которого множество ядер) для общих вычислений, которые обычно проводит центральный процессор – GPGPU (сокр. от англ. General-Purpose Graphics Processing Units, дословно – GPU общего назначения);

шейдер (англ. shader) – программа построения теней на синтезируемых изображениях, используется в трехмерной

графике для определения окончательных параметров объекта или изображения. Как правило, включает произвольной сложности описание поглощения и рассеяния света, наложения текстуры, отражения и преломления, затенения, смещения поверхности и эффекты постобработки. Сложные поверхности могут быть визуализированы при помощи простых геометрических форм;

рендеринг (англ. rendering) – визуализация, в компьютерной графике процесс получения изображения по геометрической модели с помощью программного обеспечения;

набор инструментальных средств разработки программного обеспечения – SDK (сокр. от англ. Software Development Kit);

центральный микропроцессор; устройство, исполняющее машинные инструкции; часть аппаратного обеспечения ПК, отвечающая за выполнение вычислительных операций (заданных операционной системой и прикладным программным обеспечением) и координирующая работу всех устройств ПК – CPU (сокр. от англ. Central Processing Unit, дословно – центральное/основное/главное вычислительное устройство);

графический процессор; отдельное устройство ПК или игровой приставки, выполняющее графический рендеринг (визуализацию) – GPU (сокр. от англ. Graphic Processing Unit, дословно – графическое вычислительное устройство). Современные графические процессоры очень эффективно обрабатывают и реалистично отображают компьютерную графику. Графический процессор в современных видеоадаптерах применяется в качестве ускорителя трехмерной графики, однако его можно использовать в некоторых случаях и для вычислений общего характера (GPGPU).

Более подробную информацию об этом можно найти в Интернете. Для программирования с использованием технологии CUDA необходимо иметь видеокарту Nvidia, поддерживающую данную технологию. В принципе все модели последних лет ее поддерживают. Для работы нужно скачать CUDA Toolkit с официального сайта компании Nvidia: https://developer.nvidia.com/cuda-downloads (последняя версия на данный момент 7.5). В данный пакет входит специальный драйвер для конкретной видеокарты, пакет разработчика (компилятор, библиотеки) и набор семплов (уже готовых примеров программ в виде проектов под VisualStudio). Настройка чистого проекта VisualStudio для использования компилятора Nvidia и библиотек CUDA достаточно трудоемкая, поэтому лучше брать готовый настроенный проект VidsualStudio из папки Samples и его код менять под конкретную задачу.

Запуск программы производится из Visual Studio нажатием Ctrl+F5.

Пример CUDA программы (элементы вектора возводятся в квадрат)

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

// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
// calculate coordinates of thread
  if (idx<N) a[idx] = a[idx] * a[idx]; // it's wish-
es don't use "if" in code for GPU
```

```
}

int main(void) {
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 10;  // Number of elements in arrays
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);        // Allocate array on host
  cudaMalloc((void **) &a_d, size);   // Allocate array on device
  // Initialize host array and copy it to CUDA device
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  // Do calculation on device:
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1); // if N=3 we have 3 blocks (size 4x4 threads)
  square_array <<< n_blocks, block_size >>> (a_d, N);
  // Retrieve result from device and store it in host array
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // Print results
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  // Cleanup
  free(a_h); cudaFree(a_d);
}
```

Лабораторная работа № 7

Цель работы: ознакомление с технологией CUDA, написание и запуск простых программ на множестве потоков на видеокарте.

Необходимое оборудование – лабораторные работы выполняются на вычислительном кластере КНИТУ-КАИ из семи узлов (Windows Server 2012 R2), шесть из которых рабочие и оснащены двумя GPU картами Nvidia Tesla C2075, один узел управляющий без видеокарт. В принципе работы могут выполняться на домашнем компьютере при наличии видеокарты Nvidia. Использовать будем Visual Studio 2012, а также готовый проект из CUDA SDK, настроенный на Visual Studio 2012 – папка vectorAdd на сервере.

Порядок проведения работы – студент подключается удаленно к одному из узлов кластера под своим пользователем через удаленный доступ Windows (ip адрес узла, логин и пароль пользователя скажет преподаватель). Вся дальнейшая работа ведется на кластере. Студент подготавливает исходные тексты программ; компилирует их в исполняемое приложение; запускает их; анализирует выходные данные программы; заполняет «Журнал исследований» по лабораторной работе; вставляет принтскрины экрана там, где требуется; заполняет таблицы с результатами вычислительных экспериментов; строит графики; заполняет графу выводы. Отвечает на контрольные вопросы, имеющиеся в некоторых заданиях.

Задание 1. Скомпилировать и запустить программу, складывающую между собой два вектора. Изменить программу таким образом, чтобы векторы перемножались с применением технологии CUDA. Вставить в программу операции по замеру времени работы (в данном случае они уже

вставлены в заготовку). Заготовку программы получить у преподавателя. Элементы векторов инициализируются их индексами. Заполнить таблицу. Построить *график изменения времени работы программы от числа потоков***.**

Задание 2. Написать *последовательную* программу, перемножающую матрицу размером **mxn** на вектор **n** (см. справочные материалы). Элементы матрицы и вектора инициализируются их индексами. В качестве заготовки взять предыдущую программу, прогнать ее для разных значений **mxn**, заполнить таблицу. *Исходные данные:* A[m][n] – матрица размерности m×n; b[n] – вектор, состоящий из n элементов. *Результат:* c[n] – вектор из n элементов.

Задание 3. Написать *параллельную реализацию* программы, перемножающей матрицу на вектор на множестве CUDA потоков. В качестве заготовки взять предыдущую программу, прогнать ее для разных значений **mxn**, заполнить таблицу. Построить *график изменения времени работы программы от размера* **mxn.** График (*Ox* – номер опыта, *Oy* – время работы (мс))**.**

Задание 4. Найти ответы на контрольные вопросы (поможет программа devicequery), исходник программы с проектом VisualStudio на сервере.

8. ОБРАБОТКА ГРАФИЧЕСКОЙ ИНФОРМАЦИИ С БИБЛИОТЕКОЙ OPENCV НА МНОЖЕСТВЕ ПОТОКОВ С ТЕХНОЛОГИЕЙ CUDA

Библиотека алгоритмов компьютерного зрения, обработки изображений и численных алгоритмов общего назначения с открытым кодом – OpenCV (англ. Open Source Computer

Vision Library – библиотека компьютерного зрения с открытым исходным кодом). Реализована на C/C++, также разрабатывается для Python, Java, Ruby, Matlab и других языков. Может свободно использоваться в академических и коммерческих целях, распространяется по лицензии BSD.

Настройка проекта VisualStudio для использования библиотеки OpenCV

Для выполнения лабораторных работ понадобится библиотека openCV https://sourceforge.net/projects/opencvlibrary/files/opencv-win/ (последняя версия была 2.14.13). Нужно распаковать скачанную папку и можно начинать работать (например распакова папку в D:\OpenCV), далее – подключать необходимые библиотеки из распакованной папки.

Для настройки проекта, чтобы он видел библиотеку OpenCV, нужно создать обычный проект VisualStudio (см. рис. 1.2), снять галочки: предварительно скомпилированный заголовок и проверка жизненного цикла (см. рис.1.3). Нужно заменить сгенерированную заготовку кода программой OpenCV_hellow_world.cpp.

После этого в свойствах проекта прописать пути для поиска и подключения заголовочных файлов и библиотек (см. рис. 1.4) Проект-> свойства «название проекта» -> каталоги VC++ -> каталоги включения (D:\OpenCV\build\include;) и каталоги библиотек (D:\OpenCV\build\x86\vc11\lib), также необходимо прописать дополнительные библиотеки для использования (см. рис. 1. 5) Проект-> свойства «название проекта» -> Компоновщик -> Ввод ->дополнительные зависимости-> Изменить (нужно добавить opencv_core2413d.lib, opencv_highgui2413d.lib – для данных примеров этих библио-

тек достаточно), после чего нажать F7 для компиляции и сборки программы, запускать программу пока не нужно.

Если все хорошо и сборка прошла удачно, то нужно зайти в папку с данным проектом, где теперь в папке Debug, появилась эта программа. Сюда необходимо скопировать файлы: opencv_highgui2413d.dll, opencv_core2413d.dll из папки D:\OpenCV\build\x86\vc11\bin, также сюда положить какое-нибудь изображение для обработки (например M.jpg). Теперь отсюда надо запустить консоль CMD, для этого, удерживая клавишу Shift, щелкнуть правой кнопкой мыши и выбрать «Открыть командную строку». Теперь в CMD запустить свою программу (нужно указать название своей программы): **OpenCV_1.exe M.jpg**, в результате должен получиться рис. 8.1.

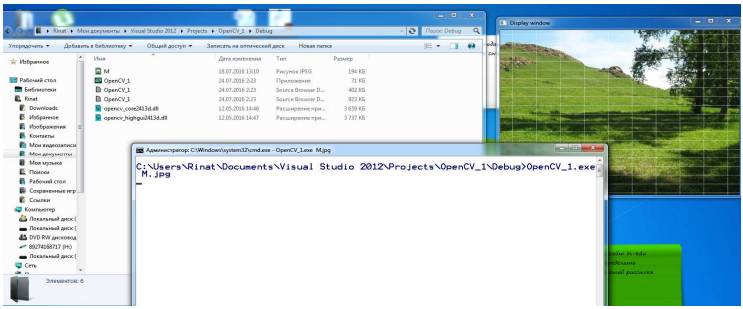

Рис. 8.1. Запуск тестового примера OpenCV

Изображение должно быть с белой сеткой.

Лабораторная работа № 8

Цель работы: ознакомление с библиотекой компьютерного зрения OpenCV, запуск простых программ, совместное использование библиотеки OpenCV и технологии CUDA.

Необходимое оборудование – лабораторные работы выполняются на вычислительном кластере КНИТУ-КАИ из семи узлов (Windows Server 2012 R2), шесть из которых рабочие и оснащены двумя GPU картами Nvidia Tesla C2075, один узел управляющий без видеокарт. В принципе работы могут выполняться на домашнем компьютере при наличии видеокарты Nvidia и скачанной библиотеки OpenCV. Использовать будем VisualStudio 2012.

Порядок проведения работы – студент подключается удаленно к одному из узлов кластера под своим пользователем через удаленный доступ Windows (ip адрес узла, логин и пароль пользователя скажет преподаватель). Вся дальнейшая работа ведется на кластере. Студент подготавливает исходные тексты программ; компилирует их в исполняемое приложение; запускает их; анализирует выходные данные программы; заполняет «Журнал исследований» по лабораторной работе; вставляет принтскрины экрана там, где требуется; заполняет таблицы с результатами вычислительных экспериментов; строит графики; заполняет графу выводы. Отвечает на контрольные вопросы, имеющиеся в некоторых заданиях.

Задание 1. Скомпилировать и запустить программу, накладывающую сетку на изображение. Создать проект VisualStudio, подключить каталоги include, lib, библиотеки OpenCV, прописать additional libraries. Исходник проекта **OpenCV_hellow_world.cpp**. Вставить принтскрин работы данной программы с изображением студента с сеткой, фото должно быть именно его, а не горы.

Задание 2. Изменить программу таким образом, чтобы вместо сетки было изменение цветов гор с зеленого на красный и чтобы результат записывался в новый файл. Фото

должно быть студента, а не горы, вставить принтскрин с результатом работы программы.

Задание 3. Изменить программу таким образом, чтобы красным становилось лицо на фото (фото можно найти в Интернете), вставить принтскрин (используйте paint.NET инструмент пипетка, чтобы понять какого цвета лицо человека).

Задание 4. Взять исходник OpenCV_MPI.cpp, скомпилировать и запустить его. Изменить программу таким образом, чтобы картинка (512 x 512 пикселей) делилась на четыре части, которые рассылались по MPI процессам, нечетные процессы должны поменять цвета с зеленого на красный, после этого надо собрать картинку вместе.

Задание 5. Взять исходник OpenCV_CUDA_Grey.cu, вставить этот код в пример проекта на CUDA, подключить библиотеку OpenCV (описано в пособии), закинуть opencv_highgui2413d.dll, opencv_core2413d.dll и image.jpg в папку с проектом. Скомпилировать и запустить программу. Нужно изменить программу таким образом, чтобы цвета были другие и вставить принтскрин с результатом работы программы.

Задание 6. Взять исходник OpenCV_Video.cpp, вставить этот код в пример проекта на CUDA, подключить библиотеку OpenCV (описано в пособии), закинуть opencv_highgui2413d.dll, opencv_core2413d.dll и M.avi в папку с проектом. Скомпилировать и запустить программу. Нужно изменить программу таким образом, чтобы цвета были другие, и вставить принтскрин с результатом работы программы.

ТЕМЫ КУРСОВЫХ ПРОЕКТОВ

1. Провести исследование на тему «Оценка производительности пакета ANSYS HPC (http://www.cadfem-cis.ru/

products/ansys/hpc/) при функционировании на одном ПК (многоядерном) и на вычислительном GPU кластере». Задачи, требующие решения в рамках задания:

а) провести краткий аналитический обзор пакетов семейства ANSYS;

б) выделить список задач для проведения оценки производительности (benchmark) пакета ANSYS HPC;

в) установить пакет ANSYS HPC на ПК;

г) установить пакет ANSYS Mechanicals на GPU кластере КАИ;

д) провести сравнение производительности на выделенных задачах (скорость решения/размерность задачи) для разных аппаратных платформ;

е) заполнить таблицы, построить сравнительные графики.

2. Создать анимационный фильм «Поступление в КАИ» (или другая тема). Рендеринг осуществить на GPU кластере КАИ (возможное ПО: Blender, Афанасий, 3D MAX, Maya). Задачи, требующие решения:

а) придумать сценарий;

б) нарисовать модели/сцены анимационного фильма;

в) установить ПО для рендеринга на GPU кластере КАИ;

г) отрендерить ролик анимационного фильма.

3. Разработать параллельную программу на CUDA, перебирающей md5 хеши для поиска нужного значения, сгенерированные ключи для перебора хранятся в текстовом файле на отдельных строках (лабораторная работа № 3).

4. Разработать программы (можно последовательную, можно параллельную) на OpenCV с использованием нейросети, принимающей картинку с CAPTCHA и выдающей текст капчи.

5. Разработать программы (параллельная с использованием CUDA) на OpenCV: принимает на вход видеофайл, на

выходе удаляет из видеофайла фрагменты видео, где нет человеческих лиц, и выдает укороченное видео (можно предложить свою модификацию задачи).

6. Необходимо разработать программу (с использованием библиотеки MPI) сортировки числового массива с помощью алгоритма сортировки-слияния (можно предложить другой алгоритм). *Исходные данные*: файл, содержащий целые числа, сгенерированные случайным образом.

7. Нахождение площади пересечения круга и прямоугольника методом Монте-Карло (с использованием библиотеки MPI). Задача решается двумя способами: в прямоугольнике ищется часть круга; в круге ищется часть прямоугольника, после чего производится усреднение. *Исходные данные*: координаты и размеры фигур: прямоугольник (X1, Y1, A, B) и круг (X2, Y2, R). *Результат*: значение площади пересечения фигур.

8. Нахождение объема пересечения шара и параллелепипеда методом Монте-Карло (с использованием библиотеки MPI). Задача решается двумя способами: в параллелепипеде ищется часть шара; в шаре ищется часть параллелепипеда, после чего производится усреднение. *Исходные данные*: координаты и размеры фигур: параллелепипеда (X1, Y1, Z1, A, B, C) и шара (X2, Y2, Z2, R). *Результат*: значение площади пересечения фигур.

9. Решение системы линейных арифметических уравнений методом Гаусса (с использованием библиотеки MPI). *Исходные данные*: матрицы коэффициентов уравнений. *Результат*: вектор коэффициентов результата.

10. Умножение матриц, ленточный алгоритм (с использованием библиотеки MPI). *Исходные данные*: размерность матриц, значения элементов исходных матриц. *Результат*: матрица результата.

11. Реализация реляционных операций выборки и соединения: Select-Project-JOIN (с использованием технологии CUDA). JOIN реализовать с помощью алгоритма вложенных циклов, обмен данных «по кольцу» (можно предложить свой вариант реализации операции JOIN). *Ограничение*: рассматривается случай, когда всего две таблицы (отношения) с небольшим количеством полей (атрибутов). *Исходные данные*: структура двух исходных отношений и сами данные отношений в виде текстовых файлов, условие соединение, атрибуты для соединения. *Результат*: результирующее отношение (в текстовом файле).

12. Цифровая фильтрация изображений (https://habra habr.ru/post/142818). Необходимо выполнить фильтрацию изображения с помощью фильтра, заданного следующим выражением (параллельная реализация программы на MPI с использованием например OpenCV):

0	−1	0
−1	5	−1
0	−1	0

Исходные данные: графический файл, значения коэффициентов фильтра. *Результат*: обработанный графический файл.

13. Моделирование во времени взаимодействия множества частиц вещества (молекул). Моделирование проводится в двухмерном пространстве размером (*A* и *B*). Исходное пространство делится на подобласти. Каждой подобласти выделяется отдельный MPI-процесс. Процесс обсчитывает частицы, находящиеся в его подобласти.

При столкновении с границей или другой частицей происходит изменение траектории с сохранением энергии. *Ис-*

ходные данные: общее количество частиц *N*, радиус частицы *R*, начальное состояние частиц (x, y, v). *Результат:* через фиксированные промежутки времени выводится текстовый файл, содержащий параметры частиц (x, y, v).

14. Реализовать модель цифрового фильтра (http://chaos.sgu.ru/~alexey/lectures/dsp/L10.pdf), можно написать последовательную программу, можно параллельную. *Исходные данные*: параметры FIR фильтров, файл с целочисленными значениями входного сигнала. *Результат:* файл с отфильтрованными данными.

СПИСОК ЛИТЕРАТУРЫ

1. *Воеводин, В.В.* Параллельные вычисления / В.В. Воеводин, Вл.В. Воеводин. – СПб.: БХВ-Петербург, 2004.

2. *Райхлин, В.А.* Начала параллельных вычислений: учебное пособие / В.А. Райхлин. – Казань: Изд-во Казан. гос. техн. ун-та, 2008.

3. *Антонов, А.С.* Технологии параллельного программирования MPI и OpenMP: учебное пособие / А.С. Антонов. – М.: МГУ, 2013.

4. *Левин, М.П.* Параллельное программирование с использованием OpenMP / М.П. Левин. – М.: Интуит, 2012.

5. *Боресков, А.В.* Основы работы с технологией CUDA / А.В. Боресков, А.А. Харламов. – М.: ДМК Пресс, 2012.

6. *Боресков, А.В.* Параллельные вычисления на GPU. Архитектура и программная модель CUDA: учебное пособие / А.В. Боресков. – М.: МГУ, 2012.

7. *Баканов, В.М.* Параллельное программирование в стандарте MPI: учебно-методическое пособие / В.М. Баканов, Д.В. Осипов. – М.: МГУПИ, 2006.

8. *Гришагин, В.А.* Параллельное программирование на основе MPI: учебное пособие / В.А. Гришагин, А.Н. Свистунов. – Н. Новгород: Изд-во ННГУ им.Н.И. Лобачевского, 2005.

9. Документация MPI-2. Открытие файлов. [Электронный ресурс]. – Режим доступа: URL: https://www.opennet .ru/docs/RUS/mpi-2/node166.html.

10. Документация MPI-2. Файловые виды. [Электронный ресурс]. – Режим доступа: URL: https://www.opennet.ru/docs/RUS/mpi-2/node175.html.

11. Академия Intel. Введение в программирование на кластерах. [Электронный ресурс]. – Режим доступа: URL: http://www.intuit.ru/studies/courses/4448/984/lecture/14945?page=2.

12. *Антонов, А.С.* Параллельное программирование с использованием технологии OpenMP: учебное пособие / А.С. Антонов. – М.: Изд-во МГУ, 2009.

13. *Абрамов, Е.В.* Практикум по параллельным вычислениям: учебное пособие / Е.В. Абрамов, И.С. Вершинин, Р.Ф. Гибадуллин, Д.О. Шагеев: под ред. В.А. Райхлина. – Казань: Изд-во Казан. гос. техн. ун-та, 2008.

14. *Сандерс, Дж.* Технология CUDA в примерах: введение в программирование графических процессоров / Дж. Сандерс, Э. Кэндрот. – М.: ДМК Пресс, 2011.

ОГЛАВЛЕНИЕ

МИНЯЗЕВ Ринат Шавкатович

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ (MPI, OpenMP, CUDA)

Учебно-методическое пособие

Редактор Л.М. Самуйлина
Компьютерная верстка – В.Н. Милёхин

Подписано в печать 08.11.21.
Формат 60 × 84 1/16. Бумага офсетная. Печать цифровая.
Усл.печ.л. 3,95. Тираж 50 экз. Заказ А 91.

Издательство КНИТУ-КАИ
420111, Казань, К.Маркса, 10

ISBN 978-5-7579-2567-7