ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ДИСТАНЦИОННОГО ОБУЧЕНИЯ И ПОВЫШЕНИЯ КВАЛИФИКАЦИИ

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

Учебно-методическое пособие

по дисциплине

«ТЕХНОЛОГИИ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛЕНИЙ»

Автор

Габрельян Б.В.

Ростов-на-Дону, 2018

Аннотация

Учебно-методическое пособие предназначено для студентов очной формы обучения направления 09.04.04 «Программная инженерия».

Авторы

доцент, к.ф.-м.н.,
каф. ПОВТиАС
Габрельян Б.В.

Оглавление

1. ЛАБОРАТОРНАЯ РАБОТА №1: РАСПРЕДЕЛЕНИЕ ВЫЧИСЛИТЕЛЬНОЙ ЗАДАЧИ В СИСТЕМАХ С ЦЕНТРАЛИЗОВАННЫМ УПРАВЛЕНИЕМ

Цель работы: Ознакомление с многопоточностью на уровне языка, реализованной в стандарте C++11 и возможностями, предлагаемыми WinAPI.

1.1. Потоки выполнения по стандарту C++11

Поток выполнения в C++11 представлен классом std::thread. Этот класс и другие (но далеко не все) классы и функции, предназначенные для поддержки потоков, объявлены в файле заголовков <thread>. Конструктор класса thread многократно перегружен и позволяет передавать вновь созданному объекту указатель на функцию, которая будет выполняться в этом потоке. Завершение работы этой функции (функции потока) означает нормальное завершение работы потока. Вызов функции осуществляется из конструктора класса thread. Простейший вариант создания и запуска нового потока выглядит следующим образом:

```
#include <thread>
using namespace std;

void f() { /* тело функции потока */ }

int main() {
  thread   t(f); // создание потока и запуск в нем функции f()
  …
  return 0;
}
```

Функция main запускается в начальном (основном) потоке. Потоки, создаваемые в функции main, по умолчанию становятся рабочими потоками. Такие потоки принудительно завершаются при завершении порождающего их потока. Чтобы заставить порождающий поток ждать завершения порожденного потока нужно вызвать для последнего метод join().

```
  thread   t(f);
  t.join();
```

Теперь функция main не завершится, пока не завершится работа потока t. Вызвать join для потока можно только единожды, после этого поток становится "неприсоединяемым". Узнать, можно ли "присоединить" поток, можно, вызвав для него метод joinable().

Поток не будет принудительно завершаться при завершении порождающего потока и в том случае, если он фоновый (демон в терминологии Unix). Для того чтобы сделать поток фоновым нужно вместо join() вызвать для него метод detach().

```
  thread   t(f);
  t.detach();
```

Функции потока можно передавать аргументы. Например,

```
void fun1(int);

int main() {
  thread   t(fun1,10); // в новом потоке вызывается fun1(10);
  t.join();
  return 0;
}
```

Можно передавать также объекты-функции (объекты классов, в которых переопределена операция вызова функции). Например,

```
class A {
  int a;
public:
  A(int  x = 0) : a(x) {}
  void operator ()() { cout<<++a<<endl; }
void operator ()(int  x) { cout<<(a += x)<<endl; }
};

int main() {
  A x(7);
  thread   t1( x ); // вызов x.operator();
thread   t2( x, 10 ); // вызов x.operator(10);
  t1.join();
  t2.join();
  return 0;}
```

Кроме того, в качестве функции потока можно использовать метод класса, но в этом случае, очевидно, нужно передавать в качестве первого аргумента указатель на объект этого класса. Например,

```
class A {
  int a;
public:
  A(int  x = 0) : a(x) {}
  void add(int  x) { a +=x; }
};

int main() {
  A x(7);
  thread   t(&A::add,&x,10); // вызов x.add(10);
  t.join();
  return 0;
}
```

Последний пример может привести к ошибке, если новый поток t будет фоновым (t.detach() вместо t.join()) т.к. в этом случае локальная переменная x уничтожается при завершении основного потока, а поток t может еще про-должать свою работу.

Если функции потока требуется передать не значение, а ссылку то возникает следующая проблема.

```
class M {…};
void  fun2(M& m);

int main() {
  M  x;
  thread   t( fun2, x );
  t.join();
  return 0;
}
```

Сама функция fun2 ожидает ссылку на объект класса M, но конструктор класса thread получает аргумент x по значению, поэтому в нем создается копия x и эта копия передается fun2 по ссылке. Чтобы fun2 получила не копию, а сам объект нужно явно указать это при передаче x конструктору thread. Для этого используют шаблон функции ref из стандартной библиотеки C++.

```
  thread   t( fun2, std::ref(x) );
```

Количество создаваемых потоков определяется программистом. Их реальную привязку к процессорам или вычислительным ядрам выполняет опера-

ционная система. Но выгоду от распараллеливания задачи можно получить только при эффективном использовании имеющихся аппаратных ресурсов. Статический метод класса tread hardware_concurrency() возвращает количе-ство процессоров или вычислительных ядер доступных в системе. Если этот метод не может получить такую информацию, возвращается значение 0.

```
int  tCount = thread::hardware_concurrency();
```

С каждым потоком связывается идентификатор – объект класса std::thread::id. Для идентификаторов разрешены операции сравнения (== != < <= > >=), присваивание и помещение в поток вывода. Получить идентификатор потока можно либо по ссылке на поток, с помощью метода get_id(), либо с помощью std::this_thread::get_id().

Потоки должны быть уникальными, т.е. нельзя создавать копии потока и нельзя присвоить один поток другому. Для этого в реализации класса thread конструктор копирования и операция присваивания закрыты. Но можно пе-редавать владение потоком от одного объекта класса thread другому объекту этого класса. Тогда первый объект уже не будет связан с потоком, владеть им будет только второй объект. В общем случае передачу владения можно вы-полнить с помощью алгоритма стандартной библиотеки C++ move. Напри-мер,

```
void fun3();

thread   t1( fun3 );
thread   t2;
t2 = move( t1 );
```

1.2. Использование WinAPI для создания и управления потоками MS Windows

В Windows каждый поток связан с уникальным целочисленным идентификатором, но управляется с помощью дескриптора имеющего тип HANDLE. Поток создается функцией WinAPI с именем CreateThread. CreateThread получает шесть аргументов: атрибуты защиты, размер стека для созданного потока, указатель на функцию потока, параметр, передаваемый функции потока, флаги создания (обычно 0), адрес переменной, в которую будет помещено значение идентификатора потока. Функция CreateThread возвращает дескриптор (хэндл) вновь созданного потока или NULL, если поток создать не удалось. Хэндл нужно освободить, когда он уже не нужен, с помощью функции CloseHandle.

Функция потока должна иметь следующую сигнатуру:

```
DWORD WINAPI  fun(LPVOID param);
```

Здесь DWORD синоним unsigned int, а LPVOID - синоним void *.

Чтобы заставить порождающий поток ждать завершения порожденного или порожденных потоков (точнее перехода соответствующих объектов в сигнальное состояние) можно использовать функции WaitForSingleObject и WaitForMultipleObjects.

```
WaitForSingleObject( HANDLE h, DWORD milliseconds );
```

В качестве второго параметра нужно задать либо достаточно большое значение, либо константу INFINITE.

```
WaitForMultipleObjects( DWORD count, HANDLE *hPtr, BOOL wait, DWORD milliseconds );
```

Ожидает перехода в сигнальное состояние count объектов ядра, дескрипторы которых помещены в массив hPtr. Если значение параметра wait TRUE, то ожидается переход всех объектов, если FALSE – какого-нибудь из этих объектов.

Например,

```
#include <windows.h>
#include <iostream>

using namespace std;

int x = 1;

DWORD WINAPI fun(LPVOID) {
  x = 100;
  return 0;
}
int main() {
  HANDLE hThread;
  DWORD id;
  hThread = CreateThread(NULL, 0, fun, 0, 0, &id);
  WaitForSingleObject(hThread, INFINITE);
  /*      или
          WaitForMultipleObjects(1,      &hThread,      TRUE,
INFINITE);
  */
  CloseHandle(hThread);

  cout<<"x="<<x<<endl; // x=100

return 0;}
```

1.3. Задания

1. Создайте класс Matrix, реализующий понятие "матрица действительных чисел". Класс должен поддерживать инициализацию, заполнение, присваива-ние, копирование, вывод матрицы на консоль, а также методы, обеспечива-ющие последовательные версии сложения, вычитания и умножения матриц.
2. Создайте программу для тестирования класса Matrix. В процессе тестиро-вания нужно генерировать матрицы разного размера с помощью генератора псевдослучайных чисел. Для сгенерированных матриц разного размера про-тестируйте работу операций сложения и умножения, замеряя при этом время выполнения этих операций.
3. Создайте для класса Matrix параллельные версии операций сложения и умножения, используя потоки C++11.
4. Протестируйте параллельные версии операций сложения и умножения для матриц того же размера, для которых тестировались последовательные вер-сии этих операций. Измерьте время выполнения операций, сравните с после-довательными версиями и сделайте выводы.
5. Создайте и протестируйте параллельные версии операций сложения и умножения для матриц для потоков MS Windows на основе WinAPI.

1.4. Контрольные вопросы.

1. Как в C++11 реализованы потоки команд?
2. Как запустить функцию на выполнение в новом потоке команд в C++11?
3. Как организовать ожидание завершения вызванного потока команд в вызвавшем потоке в C++11?
4. Как различать потоки команд в C++11?
5. Как создать поток команд с помощью Win API?
6. Как организовать ожидание завершения вызванного потока команд в вызвавшем потоке с помощью Win API?
7. Как различать потоки команд созданные с помощью WinAPI?

2. ЛАБОРАТОРНАЯ РАБОТА №2: ТЕХНОЛОГИЯ OPENMP

Цель работы: Ознакомление с технологией OpenMP для C++.

2.1. Директивы OpenMP

Стандарт OpenMP ориентирован на языки программирования Си/C++ и Fortran. В настоящем пособии рассматривается только то, что относится к языкам Си/C++. В MS Visual Studio чтобы использовать OpenMP необходимо в свойствах проекта в разделе Configuration Properties выбрать C/C++, за-тем Language и задать в Open MP Support Yes (/openpm).

В Си/C++ директивы OpenMP имеют вид прагм, обрабатываемых препроцессором. Их общий синтаксис таков:

```
#pragma omp директива [предложение [предложение] …]
```

Обычно директивы применяются к части программы, последовательному набору операторов, т.е. к некоторому структурному блоку. Структурный блок задается как обычный блок, т.е. с помощью фигурных скобок. Напри-мер,

```
#pragma omp parallel
{
  // параллельный участок программы
}
```

определяет распараллеленный блок кода, который будет выполняться в N потоках (нитях).

Количество нитей может задаваться в программе или определяется значением переменной окружения процесса OMP_NUM_THREADS. У каждой нити будет свой номер в диапазоне от 0 до OMP_NUM_THREADS-1.

Предложения shared, private, default определяют режим использования указанных в них переменных в структурном блоке. shared задает переменные, общие для всех потоков, private – локальные переменные для каждого потока, default объявляет все переменные, заданные в блоке либо как shared, либо как private, либо как none.

Предложение firstprivate объявляет локальные переменные, инициализируемые в последовательном блоке до входа в параллельный блок программы, lastprivate – переменные, значения которых доступны в последовательной части программы после выхода из параллельного блока.

if(выражение) – если значение выражения рано нуль, то в следующем блоке распараллеливание не проводится. Например,

```
…
int n = 1000;
#pragma omp parallel if( n > 1000 )
{
  // распараллеленная секция
```

```
}
```

Распараллеленная секция будет на самом деле выполняться в одном потоке (последовательно) до тех пор, пока значение переменной n меньше 1001.

reduction позволяет выполнить указанную операцию (или вызвать указанную функцию) над одноименными локальными значениями, полученными в разных потоках, с тем, чтобы полученное значение можно было использовать далее в последовательной части программы. Общий синтаксис:

reduction(знак операции или имя функции : список имен переменных)

Разрешены следующие операции: +, -, *, &, |, ^, &&, ||. Например, если количество используемых для распараллеливания потоков numThreads

```
int sum = 0;

#pragma omp parallel reduction(+: sum)
{
  for(int i=0; i<size; i += numThreads) {
            int index = i + omp_get_thread_num();
            sum += a[index] + b[index];
  }
  #pragma barrier
}

cout << "sum=" << sum << endl;
```

Здесь функция omp_get_thread_num() возвращает номер текущего потока.

#pragma barrier устанавливает барьер, заставляющий основной поток ожидать завершения всех параллельных потоков.

for позволяет организовать цикл, итерации которого будут выполняться параллельно, разделяя доступные потоки. Например,

```
cons tint size = 160;
int a[size], b[size], c[size] = {};
…
#pragma omp parallel for
for(int i=0; i<size; ++i) c[i] = a[i] + b[i];
```

По умолчанию в конце цикла ставится барьер. Если он не нужен, то его нужно явно убрать с помощью предложения nowait в директиве for.

2.2. Встроенные функции OpenMP

Прототипы функций OpenMP находятся в файле заголовков omp.h.

Функция int omp_get_num_threads() возвращает количество потоков выполняющихся в данной параллельной секции. В последовательной части программы возвращает 1.

void omp_set_num_threads(int num) задает количество потоков для параллельной секции кода, в которой не используется предложение num_threads.

int omp_get_max_threads() возвращает наибольшее количество потоков, которые могут быть созданы в параллельной секции программы.

int omp_get_thread_num() возвращает номер текущего потока. Главный поток имеет номер 0.

int omp_get_num_procs() возвращает количество процессоров, доступных в момент вызова функции.

int omp_in_parallel()возвращает ноль вне параллельной секции кода.

void omp_set_dynamic(int num) разрешает или запрещает динамическую настройку числа используемых в параллельном блоке потоков. Если num == 0 – запрещает.

int omp_get_dynamic() – если возвращает не ноль, динамическая настройка числа потоков разрешена.

void omp_set_nested(int nested) разрешает или запрещает использование вложенных параллельных блоков. Если nested 0, то вложенные параллельные блоки выполняются последовательно в текущем потоке.

int omp_get_nested() возвращает ноль, если вложенные параллельные блоки запрещены (режим по умолчанию).

double omp_get_wtime() возвращает количество секунд,

прошедших с некоторого момента, который выбирается произвольно, но не меняется во время выполнения программы.

double omp_get_wtick() возвращает количество секунд между сигналами от таймера.

2.3. Задания

1. Реализуйте в виде собственных функций следующие алгоритмы численного интегрирования: прямоугольников, трапеций, Симпсона.
2. Создайте программу для тестирования созданных алгоритмов, вычисляющую значение числа п. В процессе тестирования нужно получать результат с разной точностью, замеряя при этом время вычисления.
3. Создайте параллельные версии алгоритмов, использующие возможности OpenMP.
4. Протестируйте параллельные версии алгоритмов. Измерьте время выполнения операций, сравните с последовательными версиями и сделайте выводы.

2.4. Контрольные вопросы

1. Какими преимуществами по сравнению с другими технологиями распараллеливания обладает технология OpenMP?
2. Каковы ограничения технологии OpenMP?
3. Какая директива распараллеливает тело цикла?
4. Как узнать количество процессоров?
5. Как узнать, сколько потоков выполняются в параллельной секции кода программы?
6. Как узнать номер текущего потока?
7. Как вычислить время, затраченное на выполнение некоторого участка программы?

3. ЛАБОРАТОРНАЯ РАБОТА №3: ТЕХНОЛОГИЯ OPENMP

Цель работы: Ознакомление с технологией MPI.

3.1. Основы MPI

Интерфейс передачи сообщений (Message Passing Interface – MPI) предназначен, прежде всего, для систем с разделяемой памятью и представляет собой программный интерфейс для систем параллельных вычислений. Поддерживаются языки программирования Си/C++ и Fortran. Мы в дальнейшем будем рас-

сматривать только реализацию для Си/С++.

Классическим примером использования MPI является вычислительный кластер, узлы которого состоят из автономных вычислительных устройств с собственным процессором (или процессорами), памятью и другими ресурсами. Программа размещается на каждом узле кластера и, по команде с одного из этих узлов, запускается на выполнение. Во время работы процессы, размещенные на разных узлах, взаимодействуют друг с другом обмениваясь сообщениями. Процессы, выполняющие некоторую общую задачу можно объединить в группу. Каждый процесс в группе имеет свой ранг (номер).

Ранг это целое число, которое может принимать значение от нуля до N-1, если N – число процессов в группе. Среду, обеспечивающую поддержку передачи и посылки сообщений, обеспечивает некоторая логическая конструкция MPI, называемая коммуникатором. Есть два вида коммуникаторов, обеспечивающие взаимодействие процессов внутри группы или взаимодействие между двумя группами.

В стандарте MPI предопределены некоторые типы, например, MPI_Group, MPI_Comm. Вообще, почти все имена типов и функций MPI начинаются с префикса MPI_.

Функция int MPI_Group_size(MPI_Group group, int *size) возвращает количество процессов в группе.

int MPI_Group_rank(MPI_Group group, int *rank) возвращает ранг того процесса в группе group, в котором вызвана эта функция.

Изначально определена группа MPI_COMM_GROUP, в которой можно определять другие группы. С этой группой связан коммуникатор MPI_COMM_WORLD. Группу можно создавать так, чтобы она не была связана с коммуникатором, тогда затем нужно будет явно обеспечить эту связь, либо же можно при создании коммуникатора создать новую группу потоков.

Связь уже существующей группы с коммуникатором осуществляет функция

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group).

Над группами можно выполнять операции объединения, пересечения, разности.

int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newGroup)

int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newGroup)

int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newGroup)

Функция int MPI_Group_free(MPI_Group *group) освобождает объект group и назначает ему значение MPI_GROUP_NULL (признак ошибки).

int MPI_Comm_size(MPI_Comm comm, int *size) возвращает количество процессов в коммуникаторе comm.

int MPI_Comm_rank(MPI_Comm comm, int *rank) возвращает ранг запрашивающего процесса в коммуникаторе comm.

int MPI_Comm_create(MPI_Comm comm, MPI_Group group1, MPI_Comm *newComm) создает новый коммуникатор, а функция

int MPI_Comm_free(MPI_Comm *comm) уничтожает указанный коммуникатор.

MPI требует начальной инициализации системы до ее использования и финализации при завершении работы. Для этого используются функции
int MPI_Init(int *argc, char ***argv) и
int MPI_Finalize().

Поэтому общей схемой для MPI-программы является следующая

```
#include "mpi.h"

int main(int argc, char *argv[]) {
  MPI_Init(&argc,&argv);
  ...
  MPI_Finalize();
  return 0;
}
```

3.2. Посылка и прием сообщений

Сообщения могут пересылаться между двумя процессами, так называемый протокол "точка-точка" (point-to-point) либо между многими процессами внутри группы.

Операции точка-точка могут блокирующими или неблокирующими, буферизованными, синхронными или ориентированными на состояние.

1) Блокированная передача данных

Посылка сообщения

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dstRank, int tag, MPI_Comm comm), здесь buf – буфер, содержащий передаваемые данные, количество передаваемых данных в буфере, datatype – тип данных в буфере, dstRank – ранг процесса, которому передаются данные), tag – тэг сообщения, comm – коммуникатор.

Некоторые встроенные типы данных MPI указаны в таблице.

MPI-тип	Си-тип
MPI_CHAR	unsigned char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Прием сообщения

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int srcRank, int tag, MPI_Comm comm, MPI_Status *status), здесь MPI_Status – это структура с тремя полями MPI_SOURCE, MPI_TAG, MPI_ERROR. count задает максимальное количество принимаемых элементов типа datatype, чтобы узнать, сколько

элементов передано на самом деле нужно вызвать функцию
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count).

2) Неблокированная передача данных

Посылка сообщения

int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dstRank, int tag, MPI_Comm comm, MPI_Request *request), здесь request используется для того, чтобы запрашивать состояние связи или ждать ее завершения.

Прием сообщения
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int srcRank, int tag, MPI_Comm comm, MPI_Request *request)
Для завершения неблокированной посылки или передачи данных используется функция
int MPI_Wait(MPI_Request *request, MPI_Status *status), а для проверки того, завершена передача или нет – функция

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status). Если flag не равен нулю, значит операция, заданная параметром request завершилась.
Пример. Программа запущена на нескольких узлах и узел с номером (рангом) ноль посылает сообщение с целым значением 120, а узел с номером 1 принимает его. Коммуникатор задан переменной comm, тэг – в переменной tag.

```
…
int rank;
int data; // данные для пересылки или принимаемые
MPI_Comm_rank(comm, &rank);
if( !rank ) // если rank == 0 послать сообщение
  MPI_Send(&data,1,MPI_INT,1,tag,comm);
else if( rank == 1 ) { // принять сообщение
  MPI_Status status;
  MPI_Recv(&data,1,MPI_INT,0, tag, comm, &status);
}
```

3.3. Создание и запуск на выполнение MPI-приложений

Существует множество реализаций стандарта MPI.

Наиболее известными являются MPICH2, OpenMPI и Intel MPI. Первые две являются открытыми разработками и могут быть получены, соответственно, на сайтах http://www.mpich.org/, http://www.open-mpi.org/.

Нужно установит какую-нибудь версию MPI. Для компиляции нужно использовать компилятор mpicc, например, если программа записана в файл mpi_prog.c и нужно получит исполняемый файл с именем mpi_prog_exe

```
mpicc –o mpi_prod_exe mpi_prog.c
```

Если программу нужно запустить параллельно на 8 компьютерах вычислительного кластера (или на восьми ядрах единственного процессора) нужно выполнить команду

```
mpirun –n 8 mpi_prog_exe
```

В распределенной вычислительной системе обычно используется запуск с ключом –mashinesfile, через который передается текстовый файл, содержащий сетевые адреса компьютеров кластера. Например, если есть файл comps, содержащий

```
povtias1.dstu.edu.ru
physics.dstu.edu.ru
povtias2.dstu.edu.ru
```

команда запуска будет выглядеть так

```
mpirun –mashinesfile comps –n 9 mpi_prog_exe
```

На каждом из трех компьютеров кластера будет запущено по три процесса.

3.4. Задания

1. Реализуйте в виде собственных функций следующие алгоритмы численного интегрирования: прямоугольников, трапеций, Симпсона.
2. Создайте программу для тестирования созданных алгоритмов, вычисляющую значение числа п. В процессе тестирования нужно получать результат с разной точностью, замеряя при этом время вычисления.
3. Создайте параллель- ные версии алгоритмов, ис-

пользующие возможности MPI.

4. Протестируйте параллельные версии алгоритмов. Измерьте время выполнения операций, сравните с последовательными версиями и сделайте выводы.

3.5. Контрольные вопросы

1. Какими преимуществами по сравнению с другими технологиями распараллеливания обладает технология MPI?
2. Каковы ограничения технологии MPI?
3. Как инициализировать MPI?
4. Как послать сообщение в стандарте MPI?
5. Как принять сообщение в стандарте MPI?
6. Какие реализации стандарта MPI Вы знаете?
7. Как задать выполнение приложения на нескольких узлах?

4. ЛАБОРАТОРНАЯ РАБОТА №4: ТЕХНОЛОГИЯ OPENCL

Цель работы: Ознакомление с технологией OpenCL.

4.1. Основы OpenCL

OpenCL - это открытый стандарт, разработанный для поддержки параллельных вычислений в гетерогенных системах, включающих в себя центральные процессоры, графические процессоры и, быть может, другие ускорители. OpenCL состоит из API для координации вычислений в гетерогенных системах и кроссплатформенного языка программирования. В OpenCL-программе выделяется управляющее устройство (host), использующее контекст (context) с помощью OpenCL API. Контекст включает в себя одно или несколько устройств (compute device), каждое из которых содержит в себе какое-то количество вычислительных элементов (compute unit), каждый вычислительный элемент содержит несколько процессоров (processing element).

На устройствах (device) параллельно выполняются Си-образные функции, называемые ядрами (kernels). Ядро получает необходимые данные через свои аргументы и не возвращает никакого значения. Собственно OpenCL-программа состоит из набора ядер, запускаемых на устройстве и функций, выполняющихся

на host. Команды, выполняемые устройством, в том числе и ядра, организованы в виде очереди команд (command queue). Для синхронизации команд в очереди или синхронизации работы двух разных очередей команд используются события. Одно и то же ядро выполняется параллельно в нескольких потоках. Каждый поток имеет собственный номер (индекс). В зависимости от алгоритма распараллеливания, индексирование потоков можно задавать в виде числа (нумерация элементов в последовательности), двух чисел, или трех чисел. Потоки объединяются в группы. У каждой группы потоков есть свой номер (groupID), каждый поток в группе получает свой локальный номер (localID). Синхронизировать работу потоков в группе можно с помощью барьеров. Между группами синхронизация не проводится. Один хост и контекст, который может содержать несколько устройств, в терминологии OpenCL образуют платформу. OpenCL может работать с несколькими платформами. Но одна платформа представляет некоторую специфичную среду, обычно предоставленную одним разработчиком конкретной реализации OpenCL API, например, AMD, NVIDIA или Intel. Узнать количество доступных программе платформ или получить информацию о них можно с помощью функции clGetPlatformIDs. Узнать количество доступных конкретной платформе устройств или получить информацию о них можно с помощью функции clGetDeviceIDs. Создать контекст и связать его с указанными устройствами можно с помощью функции clCreateContext. Очередь команд создает функция clCreateCommandQueue.

Память системы состоит из памяти хоста и памяти устройства (device), организованной следующим образом.

Есть глобальная память, доступ которой имеют все вычислительные элементы (compute unit), локальная память (local memory), доступная всем потокам одного вычислительного элемента и закрытая память (private memory), доступная только одному потоку. Константная память – это часть глобальной памяти, содержащая неизменяемые значения

4.2. Создание и вызов ядра в технологии OpenCL

OpenCL вводит некоторые ограничения на синтаксис языка C99, такие, как запрет на рекурсивные вызовы функций, запрет на использование указателей на функции, отсутсвие моделей памяти (т.е. нельзя использовать спецификаторы auto, register, static, extern), но добавляет некоторые расширения. Так, чтобы объявить функцию ядра (ядро), необходимо задать модифика-

тор __kernel. Так как ядро выполняется на устройстве, для переменных, используемых ядром нужно специфицировать вид памяти: __global, __constant, __local или __private. Аргументы ядра не могут быть __private.

OpenCL поддерживает следующие скалярные типы данных:

Си-тип	OpenCL-тип	описание
bool		значения true, false или 1, 0
char	cl_char	8-битовое знаковое целое
unsigned char, uchar	cl_uchar	8-битовое беззнаковое целое
short	cl_short	16-битовое знаковое целое
unsigned short, ushort	cl_ushort	16-битовое беззнаковое целое
int	cl_int	32-битовое знаковое целое
unsigned int, uint	cl_uint	32-битовое беззнаковое целое
long	cl_long	64-битовое знаковое целое
unsigned long, ulong	cl_ulong	64-битовое беззнаковое целое
float	cl_float	32-битовое значение с плавающей точкой
double	cl_double	64-битовое значение с плавающей точкой
half	cl_half	16-битовое значение с плавающей точкой
size_t		беззнаковый целый тип, возвращаемый операцией sizeof
ptrdiff_t		знаковый целый тип, результат вычитания двух указателей

intptr_t		знаковый целый тип, в который всегда может быть преобразован нетипизированный указатель
uinptr_t		беззнаковый целый тип, в который всегда может быть преобразован нетипизированный указатель
void	void	тип с пустым множеством допустимых значений

Можно использовать и соответствующие Си-типы, и OpenCL-типы, а также указатели на эти типы. Кроме того, определены соответствующие векторные типы, например, cl_char2. Размерности векторов могут быть равными 2, 3, 4, 8 или 16.

Например, ядро, использующее count потоков для нахождения компонентов вектора – суммы двух других векторов, может выглядеть следующим образом

```
__kernel void add(__global const float* a, __global const float*
b, __global float* c, int count) {
    int index = get_global_id(0);
    if (index >-1 && index< count) {
        c[index] = a[index] + b[index];
    }
}
```

Здесь используется последовательная нумерация потоков. Функция get_global_id возвращает индекс текущего потока. Если используется двумерная нумерация, то get_global_id(0) возвращает индекс по первому, а get_global_id(1) – по второму измерению.

В OpenCL существует тип cl_kernel, представляющий объект "ядро". Идентификатор самой программы представлен типом cl_program_id, программа типом cl_program, платформы типом cl_platform_id, контекст типом cl_context, устройство типом cl_device.

4.3. Создание OpenCL-приложений

Для создания OpenCL-программы необходимо выполнить следующие шаги: выбрать доступную платформу, выбрать подходящее устройство, создать контекст для этого устройства, создать

в этом контексте очередь команд, создать буферы под используемые данные, откомпилировать программу и ядро, передать ядру аргументы и выполнить ядро.

Например, если в переменной count содержится длина вектора c – результата сложения двух других векторов a и b

```
…
cl_int errno; // номер ошибки
cl_platform_id platform;
errno = clGetPlatformIDs(1,&platform,NULL); // выбрать первую платформу
cl_device_id device;
// Выбрать первое устройство
errno = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, NULL);
cl_context_properties ccp[] =
{ CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0 };
// Создать контекст
cl_context context = clCreateContext(cps,1,&device,NULL,NULL,&errno);
// Создать очередь команд
cl_command_queue queue = clCreateCommandQueue(context,device,0,&errno);
// Создать буферы данных и переслать данные на устройство
cl_mem bufferA = clCreateBuffer( context, CL_MEM_READ_ONLY, cout*sizeof(float), NULL, &errno); // память для вектора a на устройстве
errno = clEnqueueWriteBuffer(queue,bufferA,CL_TRUE,0,cout*sizeof(float),(void *)a,0,NULL,NULL); // скопировать массив a на устройство
cl_mem bufferB = clCreateBuffer( context, CL_MEM_READ_ONLY, cout*sizeof(float), NULL, &errno); // память для вектора b на устройстве
errno = clEnqueueWriteBuffer(queue,bufferB,CL_TRUE,0,cout*sizeof(float),(void *)b,0,NULL,NULL); // скопировать массив b на устройство
cl_mem bufferC = clCreateBuffer( context, CL_MEM_READ_ONLY, cout*sizeof(float), NULL, &errno); // память для вектора c на устройстве
// Ядро в виде строки
char *source =
```

```
"__kernel \n"
"void add(__global const float* a,__global const float* b,__global float* c"
", int count) {\n"
"    int index = get_global_id(0);\n"
"    if (index >-1 && index< count) {\n"
"        c[index] = a[index] + b[index];\n"
"    }\n"
"}\n";
// Компиляция программы и ядра
cl_program program = clCreateProgramWithSource( context, 1, (const char **)&source, NULL, &errno);
errno = clBuildProgram(program,0,NULL,NULL,NULL,NULL);
cl_kernel kernel = clCreateKernel(program,"add",&errno);
// Задать аргументы ядру
clSetKernelArg(kernel,0,sizeof(cl_mem),&bufferA);
clSetKernelArg(kernel,0,sizeof(cl_mem),&bufferB);
clSetKernelArg(kernel,0,sizeof(cl_mem),&bufferC);
clSetKernelArg(kernel,0,sizeof(cl_int),&count);
// Задать размеры локальной и глобальной рабочих групп
size_t localws[1] = { 16 }; // считаем, что размер массива кратен 16
size_t globalws[1] = {count};
// Запуск ядра
errno = clEnqueueNDRangeKernel( queue, kernel, 1, NULL, globalws, localws, 0, NULL, NULL);
// Пересылка результата на хост
errno = clEnqueueReadBuffer( queue, bufferC, CL_TRUE, 0, count*sizeof(float), (void *)c, 0, NULL, NULL);
```

4.4. Задания

1. Реализуйте в виде собственных функций следующие алгоритмы численно-го интегрирования: прямоугольников, трапеций, Симпсона.
2. Создайте программу для тестирования созданных алгоритмов, вычисля-ющую значение числа п. В процессе тестирования нужно получать результат с разной точностью, замеряя при этом время вычисления.
3. Создайте параллельные версии алгоритмов, использующие возможности OpenCL.
4. Протестируйте параллельные версии алгоритмов. Измерьте время выпол-нения операций, сравните с последовательными версиями и сделайте вы- воды.

4.5. Контрольные вопросы

1. Какими преимуществами по сравнению с другими технологиями распараллеливания обладает технология OpenCL?
2. Каковы ограничения технологии OpenCL?
3. Как виды памяти поддерживаются в OpenCL?
4. Какие расширения синтаксиса языка Си введены в стандарте OpenCL?
5. Что такое ядро в стандарте OpenCL?
6. Какие реализации стандарта OpenCL Вы знаете?
7. Какие другие технологии вычислений с использованием графических карт Вы знаете?

5. ЛАБОРАТОРНАЯ РАБОТА №5: SIMD-РАСШИРЕНИЯ АРХИТЕКТУРЫ INTEL 8086

Цель работы: Ознакомление с набором SIMD-инструкций процессоров семейства Intel 80x86.

СПИСОК ЛИТЕРАТУРЫ

1. Э. Таненбаум Компьютерные сети. – СПб: Питер, 2012.
2. Федотов И.Е. Модели параллельного программирования. – СОЛОН-Пресс, 2012. – URL: https://e.lanbook.com/book/13807
3. Болодурина И.П., Волкова Т.В. Проектирование компонентов распределенных информационных систем. – Оренбург-ский государс-твенный университет, ЭБС АСВ, 2012. – URL: http://www.iprbookshop.ru/30122.html
4. Э. Таненбаум, М. ван Стеен Распределенные системы. Принципы и парадигмы. – СПб.: Питер, 2003.