МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Г. В. Абрамов, Н. А. Каплиева

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум

Воронеж
Издательский дом ВГУ
2021

Рекомендовано к изданию научно-методическим советом факультета прикладной математики, информатики и механики 15 октября 2021 г., протокол № 2

Рецензент – д-р физ.-мат. наук, проф.

Учебно-метдическое пособие подготовлено на кафедре математического обеспечения ЭВМ факультета прикладной математики, информатики и механики Воронежского государственного университета.

Рекомендовано студентам первого курса магистратуры очной формы обучения факультета прикладной математики, информатики и механики

Для направлений: 01.04.02 – Прикладная математика и информатика,
02.04.02 – Фундаментальная информатика и информационные технологии

1. Общая характеристика параллельных вычислительных систем

Разработка параллельных программ во многих случаях осуществляется для решения вычислительно-трудоемких научно-технических задач. В таких ситуациях одним из наиболее важных критериев эффективности параллельных программ является достижение минимально возможного времени вычислений. Один из основных резервов ускорения вычислений – совершенствование алгоритмов решения задач, но на стадии программной разработки главным является максимальное использование аппаратных возможностей используемых вычислительных систем. Выполняемая при этом оптимизация программ является машинно-зависимой для полного учета характерных особенностей компьютерного оборудования. Тем самым, профессиональная деятельность в области параллельного программирования требует хорошего знания общих принципов и способов построения параллельных вычислительных комплексов, примеров высокопроизводительных систем, в которых наиболее ярко проявляются современные тенденции развития индустрии вычислений.

1.1. Классификация вычислительных систем [1]

Одним из наиболее распространенных способов классификации ЭВМ является *систематика Флинна* (Flynn), в рамках которой основное внимание при анализе архитектуры вычислительных систем уделяется способам взаимодействия последовательностей (*потоков*) выполняемых команд и обрабатываемых данных. В результате такого подхода различают следующие основные типы систем:

- **SISD** (Single Instruction, Single Data) – системы, в которых существует одиночный поток команд и одиночный поток данных; к данному типу систем можно отнести обычные последовательные ЭВМ.
- **SIMD** (Single Instruction, Multiple Data) – системы, с одиночным потоком команд и множественным потоком данных; подобный класс составляют многопроцессорные вычислительные системы, в которых в каждый момент времени может выполняться одна и та же команда для обработки нескольких информационных элементов; подобной архитектурой обладают, например, многопроцессорные системы с единым устройством управления; данный подход широко использовался в предшествующие годы, в последнее время его применение ограничено, в основном, созданием специализированных систем.
- **MISD** (Multiple Instruction, Single Data) – системы, в которых имеется множественный поток команд и одиночный поток данных. К данному типу можно отнести системы с конвейерной обработкой данных.

- **MIMD** (Multiple Instruction, Multiple Data) – системы с множественным потоком команд и множественным потоком данных; к подобному классу систем относится большинство параллельных многопроцессорных вычислительных систем.

Хотя систематика Флинна широко используется при конкретизации типов компьютерных систем, такая классификация приводит к тому, что практически все виды параллельных систем (несмотря на их существенную разновидность) относятся к одной группе MIMD. Поэтому многими исследователями предпринимались неоднократные попытки детализации систематики Флинна. Так, например, для класса MIMD предложена практически общепризнанная структурная схема, в которой дальнейшее разделение типов многопроцессорных систем основывается на используемых способах организации оперативной памяти в этих системах. Данный подход позволяет различать два важных типа многопроцессорных систем *multiprocessors* (*мультипроцессоры* или системы с общей разделяемой памятью) и *multicomputers* (*мультикомпьютеры* или системы с распределенной памятью).

1.2. Мультипроцессоры

Для дальнейшей систематики **мультипроцессоров** учитывается способ построения общей памяти. Возможный подход – использование единой (централизованной) *общей памяти* (*shared memory*), такой подход обеспечивает *однородный доступ к памяти* (*uniform memory access*) – рис. 1а.

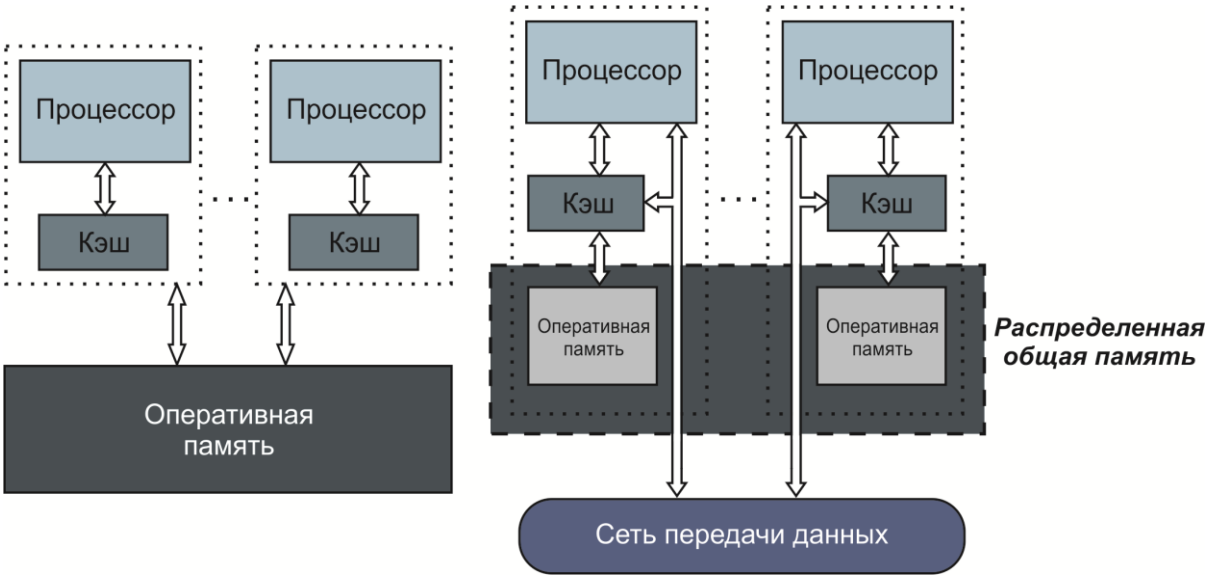


Рис. 1. Архитектура многопроцессорных систем с общей (разделяемой) памятью: системы с однородным (а) и неоднородным доступом к памяти (б)

Одной из основных проблем, которые возникают при организации параллельных вычислений на такого типа системах, является доступ с разных процессоров к общим данным и обеспечение, в этой связи, *однозначности* (*когерентности*) *содержимого разных кэшей* (*cache coherence problem*). Дело в том, что при наличии общих данных копии значений одних и тех же переменных могут оказаться в кэшах разных процессоров. Если в такой ситуации (при наличии копий общих данных) один из процессоров выполнит изменение значения разделяемой переменной, то значения копий в кэшах других процессоров окажутся не соответствующими действительности и их использование приведет к некорректности вычислений. Обеспечение однозначности кэшей обычно реализуется на аппаратном уровне – для этого после изменения значения общей переменной все копии этой переменной в кэшах отмечаются как недействительные и последующий доступ к переменной потребует обязательного обращения к основной памяти. Следует отметить, что необходимость обеспечения когерентности приводит к некоторому снижению скорости вычислений и затрудняет создание систем с достаточно большим количеством процессоров.

Наличие общих данных при выполнении параллельных вычислений приводит к необходимости *синхронизации взаимодействия* одновременно выполняемых потоков команд. Так, например, если изменение общих данных требует для своего выполнения некоторой последовательности действий, то необходимо обеспечить *взаимоисключение* (*mutual exclusion*), чтобы эти изменения в любой момент времени мог выполнять только один командный поток. Задачи взаимоисключения и синхронизации относятся к числу классических проблем, и их рассмотрение при разработке параллельных программ является одним из основных вопросов параллельного программирования.

Общий доступ к данным может быть обеспечен и при физически распределенной памяти (память физически распределена, но логически общедоступна), при этом длительность доступа уже не будет одинаковой для всех элементов памяти. Такой подход называется *неоднородный доступ к памяти* (*non-uniform memory access*).

Использование распределенной общей памяти (*distributed shared memory*), рис. 1б, упрощает проблемы создания мультипроцессоров, однако, возникающие при этом проблемы эффективного использования распределенной памяти (время доступа к локальной и удаленной памяти может различаться на несколько порядков) приводят к существенному повышению сложности параллельного программирования.

Наиболее широко применяемой технологией для организации параллельных вычислений на многопроцессорных системах с общей памятью является технология OpenMP (*Open Multi-Processing*).

1.3. Мультикомпьютеры

Мультикомпьютеры (многопроцессорные системы с распределенной памятью) уже не обеспечивают общий доступ ко всей имеющейся в системах памяти (*non-remote access*) – рис. 2.

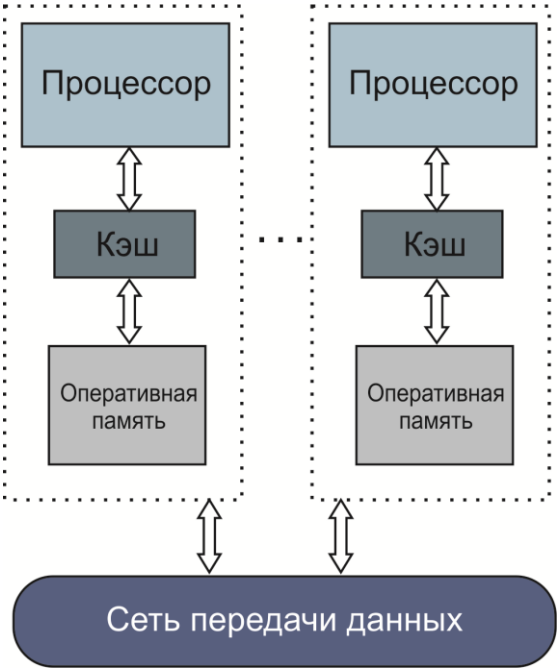


Рис. 2. Архитектура многопроцессорных систем с распределенной памятью

В отличие от систем с распределенной памятью (рис. 1*б*), в мультикомпьютерах каждый процессор системы может использовать только свою локальную память, для доступа к данным, располагаемым на других процессорах, необходимо явно выполнить *операции передачи сообщений* (*message passing operations*). Данный подход используется при построении двух важных типов многопроцессорных вычислительных систем – *массивно-параллельных систем* (*massively parallel processor or MPP*) и *кластеров* (*clusters*).

В настоящее время наиболее распространенной технологией программирования для параллельных компьютеров с распределенной памятью является технология MPI (*Message Passing Interface*).

1.4. Подходы к организации параллелизма

Основой для разработки параллельных программ, реализующих параллельные методы решения задач, являются понятия *процесса* и *потока*. Рассмотрение программы в виде набора процессов/потоков, выполняемых параллельно на разных процессорах или на одном процессоре в режиме

разделения времени, позволяет сконцентрироваться на рассмотрении проблем организации взаимодействия параллельных частей программы, определить моменты и способы обеспечения синхронизации и взаимоисключения процессов/потоков, гарантировать отсутствие тупиков (ситуаций, в которых все или часть параллельных участков программы не могут быть выполнены при любых вариантах продолжения вычислений) в ходе выполнения программы [1].

Процесс – это некоторая последовательность команд, претендующая наравне с другими процессами программы на использование процессора для своего выполнения. Процесс может находиться в одном из трех состояний:

– *активном*, процесс находится в состоянии выполнения;

– *ожидания*, процесс приостановлен и ожидает возможности своего выполнения;

– *блокировки*, временная неготовность процесса к дальнейшему выполнению.

В ходе своего выполнения состояние процесса может многократно изменяться.

Выполняемые программы с точки зрения операционной системы представляют собой процессы, которые параллельно выполняются, взаимодействуют между собой и конкурируют за использование процессоров вычислительной системы. Создание процесса, переключение процессоров между процессами и выполнений других подобных действий занимает достаточно много процессорного времени. Кроме того, процессы выполняются в разных адресных пространствах и, следовательно, организация их взаимодействия требует определенных усилий.

В качестве более простой альтернативы понятию процесса можно рассматривать *поток*. Поток (точно так же, как и процесс) можно представлять как последовательность команд программы, которая может претендовать на использование процессора вычислительной системы для своего выполнения. Но, в отличие от процесса, потоки одной и той же программы работают в общем адресном пространстве (разделяют данные программы). Общность данных, с одной стороны, существенно упрощает организацию взаимодействия потоков, но, с другой стороны, требует соблюдения определенных правил использования разделяемых данных.

Параллелизм за счет нескольких процессов

Первый способ распараллелить приложение – разбить его на несколько однопоточных одновременно исполняемых процессов. Затем эти отдельные процессы могут обмениваться сообщениями, применяя стандартные

каналы межпроцессной коммуникации (сигналы, сокеты, файлы, конвейеры и т. д.) [2]

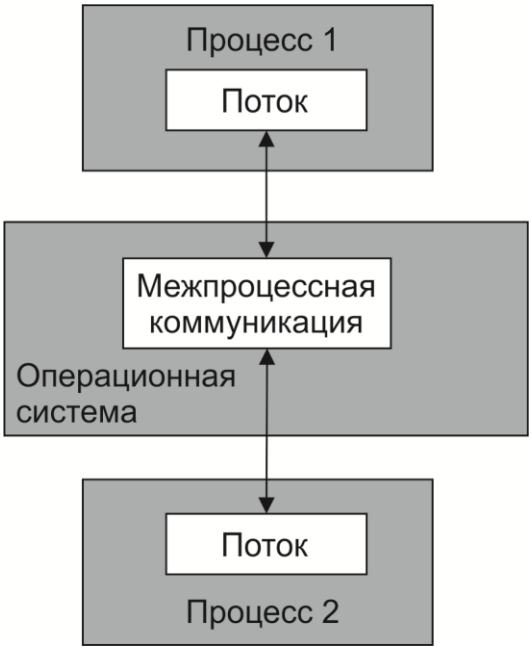


Рис. 3. Коммуникация между двумя параллельно работающими процессорами

Недостаток такой организации связи между процессами в его ***сложности, медленности***, так как операционная система должна обеспечить защиту процессов, чтобы ни один не мог случайно изменить данные, принадлежащие другому. Второй недостаток – ***неустранимые накладные расходы на запуск нескольких процессов***: для запуска процесса требуется время, ОС должна выделить внутренние ресурсы для управления процессом и т. д.

Плюсы: 1) благодаря ***надежной защите процессов, обеспечиваемой операционной системой, и высокоуровневым механизмам коммуникации*** написать *безопасный* параллельный код проще, оперируя процессами, а не потоками; 2) процессы ***можно запускать на разных машинах, объединенных сетью***. Хотя затраты на коммуникацию при этом возрастают, но в хорошо спроектированной системе такой способ повышения степени параллелизма может оказаться очень эффективным, и общая производительность увеличится.

Параллелизм за счет нескольких потоков

Альтернативный подход к организации параллелизма – запуск нескольких потоков в одном процессе. Потоки можно считать облегченными процессами – каждый поток работает независимо от всех остальных, и все потоки могут выполнять разные последовательности команд. Однако все

принадлежащие процессу потоки разделяют общее адресное пространство и имеют прямой доступ к большей части данных – **глобальные переменные остаются глобальными, указатели и ссылки на объекты можно передавать из одного потока в другой**. Для процессов тоже можно организовать доступ к разделяемой памяти, но это и сделать сложнее, и управлять не так просто, потому что адреса одного и того же элемента данных в разных процессах могут оказаться разными.

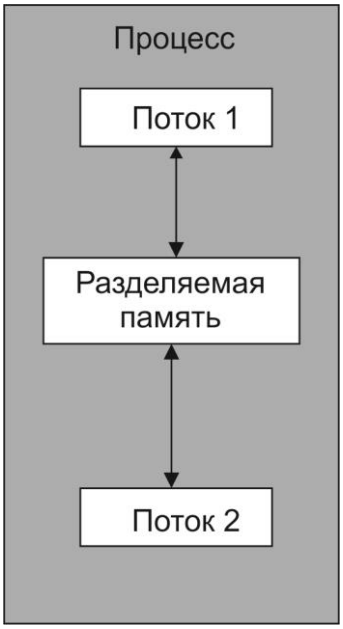


Рис. 4. Коммуникация между двумя параллельно исполняемыми потоками в одном процессе

Благодаря общему адресному пространству и отсутствию защиты данных от доступа со стороны разных потоков накладные расходы, связанные с наличием нескольких потоков, существенно меньше, так как на долю операционной системы выпадает гораздо меньше учетной работы, чем в случае нескольких процессов. Однако за гибкость разделяемой памяти приходится расплачиваться – если к некоторому элементу данных обращаются несколько потоков, то программист должен обеспечить согласованность представления этого элемента во всех потоках, т.е. коммуникацию между потоками необходимо тщательно продумывать.

Низкие накладные расходы на запуск потоков внутри процесса и коммуникацию между ними стали причиной популярности этого подхода во всех распространенных языках программирования, несмотря на потенциальные проблемы, связанные с разделением памяти.

2. Распараллеливание средствами языка программирования C++

Стандарт C++11 (C++14, C++17) языка C++ предоставляет достаточно широкий набор функций и классов для разработки многопоточных приложений: запуска потоков, организации их взаимодействия и синхронизации.

2.1. Асинхронные задачи

Рассмотрим в качестве примера следующую задачу: дан одномерный массив, найти сумму его элементов. Если массив достаточно большой, то для повышения производительности можно воспользоваться механизмом распараллеливания по данным. В этом случае каждый поток будет выполнять одну и ту же задачу, но с разными наборами данных (фрагментами массива).

Стандартная библиотека C++ предоставляет шаблон функции `std::async`, объявленные в заголовочном файле `<future>`. Функция `std::async` позволяет запустить *асинхронную задачу*, результат которой может быть получен по мере готовности в будущем. Она возвращает объект-будущее `std::future`, который и будет содержать вычисленное значение. Для получения значения необходимо воспользоваться методом `get()` объекта-будущего. В момент вызова метода `get()`, поток, вызвавший этот метод, будет приостановлен до готовности будущего результата (ожидание завершения работы потоков называется *барьерной синхронизацией*).

В качестве первого аргумента функции `std::async` передается указатель на задачу потока, далее перечисляются параметры самой задачи. Если какой-то параметр задача потока принимает по ссылке, то он передается в обертке `std::ref` (по умолчанию все параметры передаются потоку копированием).

По умолчанию реализацией определяется, запускает ли `std::async` новый поток или задача работает синхронно. Требуемый режим можно задать в дополнительном нулевом параметре (перед указателем на задачу потока). Этот параметр имеет тип `std::launch` и может принимать следующие значения: `std::launch::deferred` – отложить вызов функции до того момента, когда будет вызван метод `wait()` или `get()` объекта-будущего; `std::launch::async` – запускать функцию в отдельном потоке.

В стандартной библиотеке C++ есть две разновидности будущих результатов, реализованных в форме двух шаблонных классов, которые объявлены в заголовке `<future>`: *уникальные будущие результаты* (`std::future<>`) и *разделяемые будущие результаты* (`std::shared_future<>`). На одно событие может ссылаться только один экземпляр `std::future<>`, на несколько экземпляров `std::shared_future<>`. В последнем случае все экземпляры оказываются

готовы одновременно и могут обращаться к ассоциированным с событием данным. Если ассоциированных данных нет, то следует использовать специализации шаблонов `std::future<void>` и `std::shared_future<void>`. Хотя будущие результаты используются как механизм межпоточной коммуникации, сами по себе они не обеспечивают синхронизацию доступа. Если несколько потоков обращаются к единственному объекту-будущему, то они должны защитить доступ с помощью какого-либо механизма синхронизации. Однако каждый из нескольких потоков может работать с собственной копией `std::shared_future<>` безо всякой синхронизации, даже если все они ссылаются на один и тот же асинхронно получаемый результат.

```
#include <iostream>
#include <future>
#include <Windows.h>

// количество элементов в массиве
const size_t COUNT = 110;

// количество используемых потоков
const size_t NTHREAD = 4;

// задача потока - сумма элементов в диапазоне [left..right)
int sum(int* arr, size_t left, size_t right)
{
     int result = 0;
     for (size_t i = left; i < right; i++)
          result += arr[i];
     return result;
}

int sum_parallel(int* arr)
{
     // массив будущих результатов (от каждого потока)
     std::future<int> f[NTHREAD];
     // размер блока массива
     size_t n = COUNT / NTHREAD;
     // так функция main() запускается в отдельном потоке
     // (main thread), то создается NTHREAD - 1 дополнительных
     // потоков
     for (size_t i = 0; i < NTHREAD - 1; i++)
     {
          // запуск асинхронной задачи
          f[i] = std::async(std::launch::async, sum, arr, n*i,
n*(i + 1));
```

```
    }

    // main thread параллельно с дополнительными потоками
    // обрабатывает последний блок массива
    int global_sum = sum(arr, n*(NTHREAD - 1), COUNT);

    // ожидание готовности результатов от дополнительных
    // потоков (барьерная синхронизация)
    for (size_t i = 0; i < NTHREAD - 1; i++)
        global_sum += f[i].get();

    return global_sum;
}

void init_array(int* arr)
{
    for (size_t i = 0; i < COUNT; i++)
        arr[i] = rand() % 100;
}

int main()
{
    int arr[COUNT];

    // инициализация массива
    srand(GetTickCount());
    init_array(arr);

    // вычисление суммы в одном потоке
    int sum_nonparallel = sum(arr, 0, COUNT);
    std::cout << "sum nonparallel = " << sum_nonparallel <<
"\nsum parallel = " << sum_parallel(arr) << std::endl;

    std::cin.get();
    return 0;
}
```

2.2. Потоки. Синхронизация доступа к разделяемым данным

Модели асинхронных задач в каких-то случаях может оказаться достаточно, но если требуется более сложная обработка и более тонкий контроль за выполнением потоков, то стандартная библиотека C++11 предоставляет класс `std::thread`, объявленный в заголовке `<thread>`.

Несмотря на более сложную модель программирования, потоки обеспечивают более эффективные способы синхронизации и координации, что позволяет одному потоку передавать выполнение другому и ждать определенное время или до того момента, пока другой поток не закончит свою работу.

При создании потока с помощью конструктора `std::thread` в качестве первого параметра (так же, как в `std::std::async`) передается указатель на задачу потока, а далее аргументы задачи. Если какой-то параметр задача потока принимает по ссылке, то он передается в обертке `std::ref`.

В отличие от асинхронных задач в потоках не предусмотрен механизм возврата значений. Задачи потока всегда возвращают значение типа void.

В задаче, рассмотренной в п. 2.1, для объединения результатов работы потоков воспользуемся общей переменной `global_sum`. Каждый поток будет накапливать сумму элементов своей части массива в эту переменную. Эта переменная является разделяемым ресурсом, и поэтому необходимо организовать синхронизированный (эксклюзивный) доступ к этому ресурсу. Фрагмент кода, в котором происходит работа с разделяемым ресурсом, называется *критической секцией*.

Для организации критических секций стандартная библиотека C++11 предоставляет примитив синхронизации – мьютекс (mutex = **mut**ual **ex**clusion – взаимное исключение). Для входа в критическую секцию необходимо захватить мьютекс, вызвав метод `lock()`, а для освобождения (выхода из критической секции) – метод `unlock()`.

```
std::mutex mut;

// задача потока
void sum(int* arr, size_t left, size_t right,
         volatile int &global_sum)
{
     for (size_t i = left; i < right; i++)
     {
          // вход в критическую секцию
          mut.lock();
          global_sum += arr[i];
          // выход из критической секции
          mut.unlock();
     }
}
```

Ключевое слово `volatile` информирует компилятор, что значение переменной может меняться извне. Это может произойти под управлением операционной системы, аппаратных средств или другого потока. Поскольку значение может измениться, компилятор каждый раз загружает его из памяти.

Недостатком такой организации критической секции является то, что потокам приходится ожидать входа в критическую секцию для каждого элемента массива, что может значительно снизить общую производительность. В решении ниже потоки обрабатывают свои блоки массива независимо друг от друга (параллельно) и только один раз входят в критическую секцию для обновления значения `global_sum`.

```
// задача потока
void sum(int* arr, size_t left, size_t right, volatile int
&global_sum)
{
     int local_sum = 0;
     for (size_t i = left; i < right; i++)
     {
          local_sum += arr[i];
     }
     // вход в критическую секцию
     mut.lock();
     global_sum += local_sum;
     // выход из критической секции
     mut.unlock();
}
```

При такой организации критической секции имеется потенциальная возможность возникновения *тупика* (взаимной блокировки) – ситуации, когда один или несколько потоков ожидают какого-либо события, которое никогда не произойдет. Например, если при выполнении одного из потоков в критической секции произойдет аварийное завершение работы, то глобальный мьютекс останется в несигнальном (занятом состоянии), и ни какой другой поток не сможет его захватить и закончить свою работу.

Решить эту проблему можно двумя способами. Первый – описать обработчик исключения, в котором освобождать мьютекс даже в случае прерывания выполнения потока.

```
// задача потока
void sum(int* arr, size_t left, size_t right, volatile int
&global_sum)
{
     int local_sum = 0;
     for (size_t i = left; i < right; i++)
     {
          local_sum += arr[i];
     }
     // вход в критическую секцию
```

```
	mut.lock();
	try
	{
		global_sum += local_sum;
	}
	catch (std::string str)
	{
		// выход из критической секции
		mut.unlock();
		throw str;
	}
	// выход из критической секции
	mut.unlock();
}
```

Второй вариант – использовать шаблон класса `std::lock_guard`, предоставляемый стандартной библиотекой C++11 (определен в заголовке `<mutex>`). Данный шаблон реализует идиому RAII – захватывает мьютекс в конструкторе и освобождает в деструкторе, – гарантируя, что захваченный мьютекс обязательно будет освобожден.

```
// задача потока
void sum(int* arr, size_t left, size_t right, volatile int
&global_sum)
{
	int local_sum = 0;
	for (size_t i = left; i < right; i++)
	{
		local_sum += arr[i];
	}
	// вход в критическую секцию
	// (конструирование объекта guard)
	std::lock_guard<std::mutex> guard(mut);
	global_sum += local_sum;
	// выход из критической секции (деструкция объекта guard)
}
```

Ожидание завершения потока (барьерная синхронизация)

Чтобы дождаться завершения работы потоков необходимо вызвать функцию `join()` ассоциированного объекта `std::thread`. Вызов функции `join()` гарантирует, что поток завершится до выхода из функции `sum_parallel()`, то есть раньше, чем будет уничтожена локальная переменная `global_sum`.

При вызове join() очищается вся ассоциированная с потоком память, то есть объект std::thread больше не будет связан с завершившимся потоком. Это означает, что для каждого потока вызвать join() можно только один раз, после первого вызова объект std::thread уже не допускает присоединения. Для проверки возможности присоединения используется булевская функция joinable().

```
#include <iostream>
#include <thread>
#include <Windows.h>
#include <mutex>

// количество элементов в массиве
const size_t COUNT = 110;

// количество используемых потоков
const size_t NTHREAD = 4;

// объект синхронизации - мьютекс
std::mutex mut;

// задача потока - сумма элементов в диапазоне [left..right)
void sum(int* arr, size_t left, size_t right, volatile int
&global_sum)
{
     int local_sum = 0;
     for (size_t i = left; i < right; i++)
     {
          local_sum += arr[i];
     }
     // вход в критическую секцию
     // (конструирование объекта guard)
     std::lock_guard<std::mutex> guard(mut);
     global_sum += local_sum;
     // выход из критической секции (деструкция объекта guard)
}

int sum_parallel(int* arr)
{
     // массив потоков
     std::thread t[NTHREAD];
     // размер блока массива
     size_t n = COUNT / NTHREAD;
```

```
    volatile int global_sum = 0;

    // так функция main() запускается в отдельном потоке
    // (main thread), то создается NTHREAD - 1 дополнительных
    // потоков
    for (size_t i = 0; i < NTHREAD - 1; i++)
    {
        // запуск дополнительного потока
        t[i] = std::thread(sum, arr, n*i, n*(i + 1),
                           std::ref(global_sum));
    }

    // main thread параллельно с дополнительными потоками
    // обрабатывает последний блок массива
    sum(arr, n*(NTHREAD - 1), COUNT, global_sum);

    // ожидание завершения работы потоков
    // (барьерная синхронизация)
    for (size_t i = 0; i < NTHREAD - 1; i++)
        t[i].join();

    return global_sum;
}

void init_array(int* arr)
{
    for (size_t i = 0; i < COUNT; i++)
        arr[i] = rand() % 100;
}

int main()
{
    int arr[COUNT];

    // инициализация массива
    srand(GetTickCount());
    init_array(arr);

    // вычисление суммы в одном потоке
    int sum_nonparallel = 0;
    sum(arr, 0, COUNT, sum_nonparallel);
    std::cout << "sum nonparallel = " << sum_nonparallel <<
"\nsum parallel = " << sum_parallel(arr) << std::endl;

    std::cin.get();
```

```
    return 0;
}
```

2.3. Условные переменные

Если необходим более точный контроль над ожиданием потока, например если необходимо проверить, завершился ли поток, или поток выполнил определенную операцию, или требуется ждать только ограниченное время, то следует использовать условные переменные.

Условная переменная – примитив синхронизации, обеспечивающий блокирование одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания. Условные переменные используются вместе с ассоциированным мьютексом.

Стандартная библиотека C++11 предоставляет две реализации условных переменных `std::condition_variable` и `std::condition_variable_any`. Оба класса объявлены в заголовочном файле `<condition_variable>`. В обоих случаях для обеспечения синхронизации необходимо взаимодействие с мьютексом. Первый класс может работать только с `std::mutex`, второй – с любым классом, который можно заблокировать. Поскольку класс `std::condition_variable_any` более общий, то его использование может обойтись дороже с точки зрения объема потребляемой памяти, производительности и ресурсов операционной системы [2].

Рассмотрим основные методы для работы с условными переменными на примере задачи «Производители – потребители». Пусть есть поток (producer), подготавливающий некоторые данные и размещающий эти данные в потокобезопасном контейнере. И есть поток (consumer), извлекающий данные из контейнера и обрабатывающий их.

Поток-производитель может *подать* сигнал о наступлении какого-то события (данные готовы и размещены в контейнере) с помощью `notify_one()` или `notify_all()`. `notify_one()` разблокирует один из потоков, ожидающих объект `std::condition_variable`. `notify_all()` разблокирует все ожидающие потоки.

Поток-потребитель может *ожидать* наступления некоторого события с помощью метода `wait()`. Ожидающий поток должен сначала захватить мьютекс с помощью `std::unique_lock`, а не `std::lock_guard`. Эта блокировка передается методу `wait()`. Функция `wait()` проверяет условие и возвращает управление, если оно выполнено. Если условие не выполнено, то `wait()` освобождает мьютекс и приостанавливает поток (переводит поток в состояние ожидания). Когда условная переменная получит извещение, отправленное потоком-производителем с помощью `notify_one()`, ожидаю-

щий поток-потребитель пробудится, вновь захватит мьютекс и еще раз проверит условие. Если условие выполнено, то `wait()` вернет управление, причем мьютекс в этот момент будет захвачен. Если же условие не выполнено, то поток снова освобождает мьютекс и возобновляет ожидание.

Ожидающий поток должен освобождать мьютекс, когда находится в состоянии ожидания, и захватывать его по выходе из этого состояния, поэтому для захвата мьютекса используется обертка `std::unique_lock`, а не `std::lock_guard` (`std::lock_guard` не позволяет освобождать мьютекс по требованию). Если бы мьютекс оставался захваченным в то время, когда поток-потребитель спит, то поток-производитель не смог бы его захватить, чтобы разместить данные в контейнер, а ожидаемое условие никогда не выполнилось бы.

Стандартная библиотека C++11 предоставляет две перегрузки метода `wait()`:

- метод, который использует только `std::unique_lock`. Метод блокирует поток и добавляет его в очередь потоков, ожидающих сигнала от условной переменной; поток пробуждается, когда будет получен сигнал от условной переменной или в случае ложного пробуждения.
- метод, который в дополнении к `std::unique_lock`, принимает предикат (условие), используемый в цикле до тех пор, пока он не вернет `false`; эта перегрузка может использоваться, чтобы избежать ложных пробуждений.

```
std::mutex mut;
std::unique_lock<std::mutex> locker(mut);
std::condition_variable cv;
std::queue<int> q; // контейнер
cv.wait(locker);
cv.wait(locker, [q]() {return !q.empty(); });
```

В качестве предиката в последнем варианте используется лямбда-функция, обеспечивающая пробуждение потока-потребителя, когда в контейнере (очереди) появятся данные.

В дополнение к перегруженному методу `wait()`, библиотека C++11 предоставляет два похожих метода с такой же перегрузкой для предиката:

- `wait_for()` блокирует поток и задает интервал времени, после которого поток разблокируется

```
cv.wait_for(locker, std::chrono::seconds(5),
                    [q]() {return !q.empty(); });
```

- `wait_until()` блокирует поток и задает максимальный момент времени, в который поток разблокируется.

Перегрузка этих методов без предиката возвращает `cv_status`, показывающий, произошел ли таймаут, или пробуждение произошло из-за сигнала условной переменной, или это ложное пробуждение.

В качестве *примера* рассмотрим реализацию покотобезопасного контейнера (очереди) и модель задачи «производители – потребители». Поток-производитель генерирует информацию, размещает ее в контейнере и подает сигнал о том, что в контейнере появился элемент для обработки. Поток-потребитель ожидает сигнал о появлении информации в контейнере, если сигнал получен, то информация извлекается из контейнера и обрабатывается.

Для контроля выполняемых операций в задачи потоков `task_producer` и `task_consumer` передается ID потока, и потоки печатают размещаемую и извлекаемую информации.

Потоки-производители должны создать `volume_work_producer` элементов, потоки-потребители должны обработать `volume_work_consumer` элементов. Для контроля этих значений в циклах задач потоков используется Interlocked-функция, которая манипулируют переменными на уровне атомарного доступа. Interlocked-функции гарантируют монопольное изменение значений переменных независимо от того, как именно компилятор генерирует код, и сколько процессоров установлено в компьютере. Interlocked-функции выполняются чрезвычайно быстро.

```
long _InterlockedExchangeAdd(
   long volatile * Addend,
   long Value
);
```

Функция выполняет атомарное увеличение переменной `Addend` на значение `Value` и возвращает исходное значение `Addend`.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <queue>
#include <Windows.h>
#include <condition_variable>

class ThreadSafeQueue
{
private:
     // мьютекс для обеспечения эксклюзивного доступа
     // к элементам контейнера
     std::mutex mutex;
     std::queue<int> queue;
```

```
public:
     void push(int elem, int ID)
     {
          std::lock_guard<std::mutex> locker(mutex);
          queue.push(elem);
          std::cout << 'P' << ID << " -> " << elem << '\n';
     }

     bool try_pop(int& elem, int ID)
     {
          bool result = false;
          std::lock_guard<std::mutex> locker(mutex);
          if (!queue.empty())
          {
               result = true;
               elem = queue.front();
               queue.pop();
               std::cout << 'C' << ID << " <- " << elem << '\n';
          }
          else
               std::cout << 'C' << ID << " sleep\n";
          return result;
     }

     bool empty()
     {
          return queue.empty();
     }
};

ThreadSafeQueue TSQ;
// мьютекс для метода wait()
std::mutex mut;
std::condition_variable cv;

volatile long volume_work_producer = 10;
volatile long volume_work_consumer = 10;

void task_producer(int ID)
{
     // пока есть работа у потока-производителя
     while (_InterlockedExchangeAdd(&volume_work_producer,-1)>0)
     {
          // формируется элемент для последующей обработки
          // потоком-потребителем
```

```
            int elem = rand() % 100;
        std::this_thread::sleep_for(std::chrono::milliseconds(2));
            TSQ.push(elem + ID, ID);
            // поток-производитель передает сигнал,
            // что в контейнере появился элемент для обработки
            cv.notify_all();
        }
}

void task_consumer(int ID)
{
        // пока есть работа у потока-потребителя
        while (_InterlockedExchangeAdd(&volume_work_consumer,-1)>0)
        {
            int elem;
            std::unique_lock<std::mutex> locker(mut);
            cv.wait_for(locker, std::chrono::seconds(5),
                                []() {return !TSQ.empty(); });
            if (TSQ.try_pop(elem, ID))
            {
                // если элемент извлечен из контейнера,
                //то он обрабатывается потоком-потребителем
        std::this_thread::sleep_for(std::chrono::milliseconds(5));
            }
            else
            {
                // работа не выполнена, объем работы потребителя
                // возвращается в исходное состояние
                _InterlockedExchangeAdd(&volume_work_consumer, 1);
            }
        }
}

int main()
{
        srand(GetTickCount());
        // создание потоков:
        // 2 потока-производителя (producer);
        // 3 потока-потребителя (consumer);
        std::thread worker[5];
        for (int i = 0; i < 5; i++)
            if (i < 2)
                worker[i] = std::thread(task_producer, i);
            else
                worker[i] = std::thread(task_consumer, i);
```

```
    for (int i = 0; i < 5; i++)
         worker[i].join();

    std::cin.get();
    return 0;
}
```

3. Общие сведения о технологии OpenMP

OpenMP (*Open Multi-Processing*) – открытый стандарт для распараллеливания программ на языках C, C++ и Фортран. Описывает совокупность директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

При использовании OpenMP разработчик может не создавать новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы. При этом нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором. Директивы синхронизации и распределения работы могут не входить непосредственно в текст параллельной области. Наиболее популярная задача OpenMP – написание программ, ориентированных на циклы.

3.1. Модель параллельной программы

OpenMP предполагается SPMD-модель (Single Program Multiple Data) параллельного программирования, в рамках которой для всех параллельных нитей используется чаще всего один и тот же код. В программе может быть любое количество параллельных и последовательных областей. Само наличие данных, общих для нескольких нитей, приводит к конфликтам при одновременном несогласованном доступе. Поэтому необходимо синхронизировать доступ к общим данным. OpenMP не выполняет синхронизацию доступа различных нитей к одним и тем же файлам.

В основе технологии распараллеливания лежат два принципа

– Использование потоков (общее адресное пространство).
– «Пульсирующий» (fork-join) параллелизм.

Принцип разветвление-объединение (fork-join) (рис. 1) заключается в том, что при выполнении обычного кода (вне параллельных областей) программа исполняется одним потоком (master thread), при появлении директивы parallel происходит создание «команды» потоков для параллельного выполнения вычислений. После выхода из области действия директивы parallel происходит синхронизация, все потоки, кроме master, уничтожаются (или приостанавливаются) и продолжается последовательное выполнение кода (до очередного появления директивы parallel).

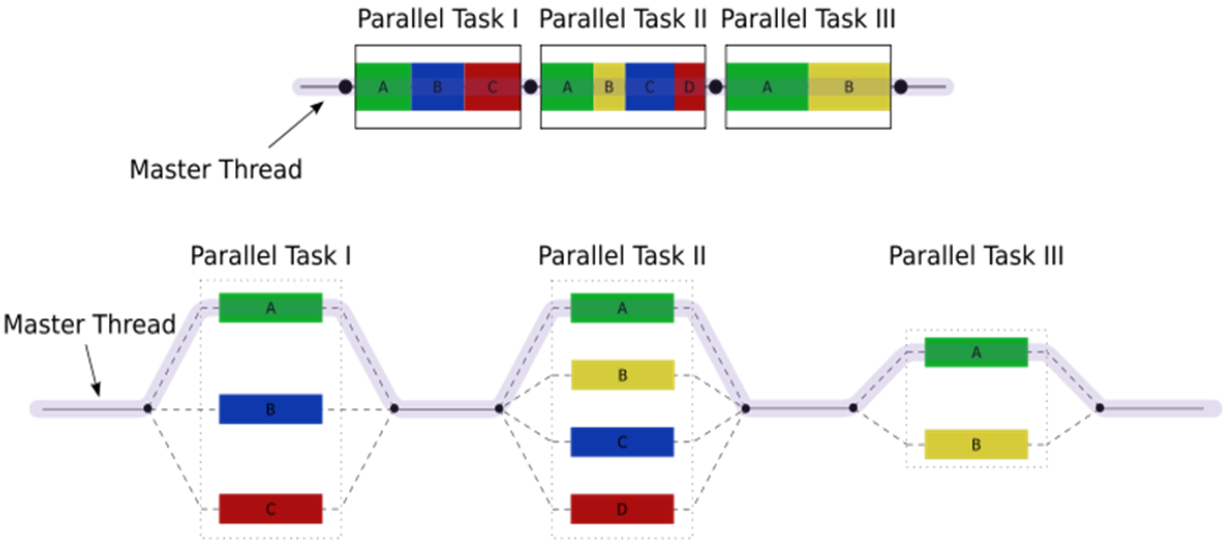


Рис. 5. Последовательное и параллельное (с использованием OpenMP) выполнение задач

Директивы и комментарии обрабатываются специальным препроцессором до начала компиляции программы. Директивы дают указания на возможные способы распараллеливания программы, при этом исходный текст программы остается неизменным. Препроцессор заменяет директивы параллелизма на дополнительный программный код (как правило, в виде обращений к процедурам параллельной библиотеки). При отсутствии директив препроцессора компилятор построит исходный последовательный программный код.

Положительные стороны такого подхода:

– снижается потребность переработки существующего программного кода,

– обеспечивается единство программного кода для последовательных и параллельных вычислений, что снижает сложность развития и сопровождения программ,

– возможно осуществлять поэтапную разработку параллельных программ.

Для использования механизмов OpenMP нужно скомпилировать программу компилятором, поддерживающим OpenMP, с указанием соответствующего ключа (например, в Visual C++ – /openmp). В Microsoft Visual Studio для использования OpenMP в настройках проекта необходимо включить поддержку: Включение OpenMP:

(Проект -> свойства -> C/C++ -> Язык -> Поддержка Open MP)

При использовании OpenMP необходимо подключить файл библиотеки называется omp.h:

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

Директивы OpenMP для C/C++ в общем случае выглядят так:

```
#pragma omp <директива> [опция[[,] опция]...]
```

Директива определяет необходимые действия, а опции директивы управляют ее работой. Если поддержка в проекте не включена, то эти директивы игнорируются. Директивы OpenMP определяют параллельные области, распределение работы и синхронизация. Каждая директива может иметь несколько дополнительных атрибутов.

Все функции, используемые в OpenMP, начинаются с префикса omp_ и записываются строчными буквами.

Замер времени

Для определения времени выполнения программы необходимо использовать функцию для работы с системным таймером `omp_get_wtime()`,которая возвращает в вызвавшей нити астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом.

```
double omp_get_wtime(void);
```

Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменён за время существования процесса. Таймеры разных нитей могут быть не синхронизированы и выдавать различные значения.

Модель данных

Модель данных в OpenMP предполагает наличие как общей для всех нитей области памяти, так и локальной области памяти для каждой нити. В параллельных областях переменные программы разделяются на два основных класса:

`shared` – все нити видят одну и ту же переменную;

`private` – каждая нить видит свой экземпляр данной переменной.

Общая переменная всегда существует лишь в одном экземпляре для всей области действия и доступна всем нитям под одним и тем же именем. Объявление локальной переменной вызывает порождение своего экземпляра данной переменной (того же типа и размера) для каждой нити. Изменение нитью значения своей локальной переменной никак не влияет на изменение значения этой же локальной переменной в других нитях. Если несколько переменных одновременно записывают значение общей переменной без выполнения синхронизации или если как минимум одна нить читает значение общей переменной и как минимум одна нить записывает значение этой переменной без выполнения синхронизации, то возникает ситуация так

называемой «гонки данных» (data race), при которой результат выполнения программы непредсказуем.

По умолчанию, все переменные, порождённые вне параллельной области, при входе в эту область остаются общими. Исключение составляют переменные, являющиеся счетчиками итераций в цикле. Переменные, порождённые внутри параллельной области, по умолчанию являются локальными.

Параллельные и последовательные области

В момент запуска программы порождается единственная **нить-мастер** или **«основная» нить**. Основная нить и только она исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные нити.

Параллельная область задаётся при помощи директивы

```
#pragma omp parallel [опция[[,] опция]...]
```

Все порождённые нити исполняют один и тот же код, соответствующий параллельной области. Предполагается, что в SMP-системе нити будут распределены по различным процессорам.

Пример.

```
int main(){
printf("Последовательная область 1\n");
#pragma omp parallel
   {
      cout<<"\t Параллельная область\n";
   }
      cout<<"Последовательная область 2\n";
}
```

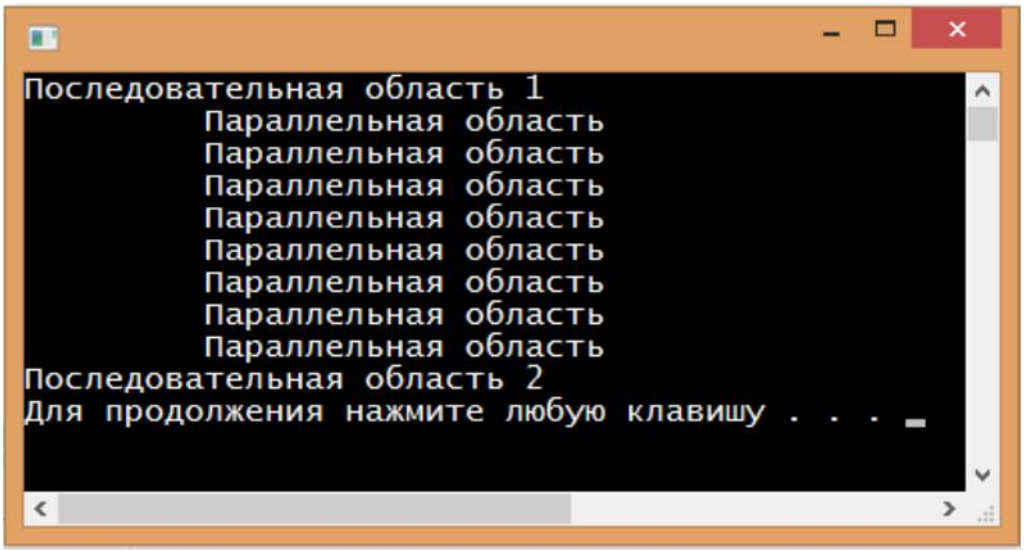


Рис. 6. Пример выполнения программы

Количество нитей в параллельной области определяется в следующим порядком:

1. Значением опции `if`
2. Значением опции `num_threads` оператора `parallel`
3. Последнее значение, установленное с помощью функции `omp_set_num_threads()`
4. Значением переменной окружения `OMP_NUM_THREADS`
5. По умолчанию – обычно это число `CPU` в узле.

Параллельные области могут быть вложенными; по умолчанию вложенная параллельная область выполняется одной нитью. Это управляется установкой переменной среды `OMP_NESTED`. Изменить значение переменной `MP_NESTED` можно с помощью вызова функции `omp_set_nested()`:

```
void omp_set_nested(int nested)
```

Функция `omp_set_nested()` разрешает или запрещает вложенный параллелизм. В качестве значения параметра задаётся 0 или 1. Если вложенный параллелизм разрешён, то каждая нить, в которой встретится описание параллельной области, породит для её выполнения новую группу нитей. Сама породившая нить станет в новой группе нитью-мастером. Если система не поддерживает вложенный параллелизм, данная функция не будет иметь эффекта.

Узнать значение переменной OMP_NESTED можно при помощи функции `omp_get_nested()`:

```
int omp_get_nested(void);
```

Функция *omp_in_parallel()* возвращает 1, если она была вызвана из активной параллельной области программы:

```
int omp_in_parallel(void);
```

Пример. Применение функции `omp_in_parallel()`:

```
void mode(void){
   if(omp_in_parallel()) cout<<"Параллельная область\n";
   else cout<<"Последовательная область\n";  }
int main(int argc, char *argv[]){
mode();
#pragma omp parallel
   {             mode();
#pragma omp master
      {           mode();        }      }
}
```

Функция `mode` демонстрирует изменение функциональности в зависимости от того, вызвана она из последовательной или из параллельной области.

• `if (условие)` – выполнение параллельной области по условию. Вхождение в параллельную область осуществляется только при выполнении некоторого условия (если значение выражения ≠ 0,то осуществляется распараллеливание). Если условие не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме.

Пример.

```
int main()
{
   int n;
     cout << "one thread" << endl;
     cout << "input number of threads: ";
     cin >> n;
     omp_set_num_threads(n);
#pragma omp parallel if( n>1 )
    {
          int k = omp_get_thread_num();
          cout << "in thread #" << k << endl;
    }
     cout << "one thread" << endl;
   return 0;
}
```

• `num_threads (целочисленное выражение)` – явное задание количества нитей, которые будут выполнять параллельную область.

• `private (list)` – задаёт список переменных, для которых создается локальная копия в каждом треде. Переменные должны быть объявлены до вхождения в параллельную область. Начальное значение локальных копий переменных из списка не определено и задается в параллельной области.

• `firstprivate (list)` – задаёт список переменных, для которых создается локальная копия в каждом треде. Переменные должны быть объявлены до вхождения в параллельную область. Начальное значение локальных копий переменных из списка определяется их значением в корневом треде.

• `shared (list)` – задаёт список переменных, которые являются общими для всех тредов. Переменные должны быть объявлены до вхождения в параллельную область. Все треды могут не только считывать, но и изменять их значения и корректность использования обеспечивает программист.

• `default (shared|none)`. Если указана опция `default (shared)`, то всем переменным в параллельной области, которым явно не назначена

локализация, будет назначена `shared` *(эта опция используется по умолчанию),* если `default (none)`, то всем переменным в параллельной области локализация должна быть назначена явно.

• `reduction (operator: list)` – задаёт **оператор** (возможны операторы `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`) и список переменных (ранее объявленных) для каждой переменной создаются локальные копии в каждом треде, локальные копии инициализируются:

– для `+ - | ^ ||` – **0** или аналоги**,**

– для `* & &&` – **1** или аналоги;

над локальными копиями переменных **после** выполнения всех операторов параллельной области выполняется заданный оператор.

3.2. Параллельные циклы

Наиболее распространенным способом распараллеливания является то или иное распределение итераций циклов. Если между итерациями некоторого цикла нет информационных зависимостей, то их можно каким-либо способом раздать разным процессорам для одновременного исполнения. Различные способы распределения итераций позволяют добиваться максимально равномерной загрузки нитей, между которыми распределяются итерации цикла.

Ограничения на параллельные циклы:

• при входе в цикл должно быть точно определено число итераций.

• результат программы должен не зависеть от того, какой именно тред выполнит конкретную итерацию цикла.

• нельзя использовать побочный выход (`break`, `goto`) из параллельного цикла.

• размер блока итераций, указанный в опции `schedule`, не должен изменяться в рамках цикла.

• Если в параллельной области встретился оператор цикла, то, согласно общему правилу, он будет выполнен всеми нитями текущей группы, то есть каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву `for`:

```
#pragma omp for [опция [[,] опция]...]
```

Эта директива относится к идущему следом за данной директивой блоку, включающему операторы `for`.

Формат параллельных циклов можно представить следующим образом:

```
for([целочисленный тип] i = инвариант цикла;
 i {<,>,=,<=,>=} инвариант цикла;
```

```
 i {+,-}= инвариант цикла)
```

Если директива параллельного выполнения стоит перед вложенными циклами, то директива действует только на самый внешний цикл.

Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции `lastprivate`.

Опции директивы for :

- `private (список);`
- `firstprivate (список);`
- `lastprivate (список)` – переменным, перечисленным в списке, присваивается результат с последнего витка цикла;
- `reduction (оператор:список);`
- `schedule(type[, chunk])` – опция задаёт, каким образом итерации цикла распределяются между нитями;
- `collapse(n)` — опция указывает, что n последовательных тесновложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между нитями; если опция collapse не задана, то директива относится только к одному непосредственно следующему за ней циклу;
- `ordered` – опция, говорящая о том, что в цикле могут встречаться директивы ordered; в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле;
- `nowait` – в конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция nowait позволяет нитям, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными.

Синтаксис директивы for:

Используется внутри параллельной области, заданной директивой parallel, для указания на распараллеливание конкретного цикла. Блок (составной оператор {}) не является обязательным для единственного оператора. Варианты использования директивы:

1. С использованием блока

```
#pragma omp parallel
{
```

```
     #pragma omp for[опции ...]
     {    ...for ...
     }
}
```

2. Без использования блока

```
#pragma omp parallel
{
      #pragma omp for[опции ...]
      {   ...for ...
      }
}
```

3. Сокращенный оператор (параллельная область совпадает с циклом).

```
#pragma omp parallel for[опции ...]
      {   ...for ...
      }
```

Пример

```
#define N 15
int main(int argc, char *argv[]){
int A[N], B[N], C[N], i, n;
for (i=0; i<N; i++){ A[i]=i; B[i]=2*i; C[i]=0; }
#pragma omp parallel shared(A, B, C) private(i, n)
   { n=omp_get_thread_num();
#pragma omp for
      for (i=0; i<N; i++)
       {          C[i]=A[i]+B[i];
       printf("Нить %d сложила элементы с номером %d\n", n, i);
      }
   }
   cout<<"массив A: "; for ( i = 0; i < N; i++)  { cout << A[i]
<<", " ; } cout<<"\n";
   cout<<"массив B: "; for ( i = 0; i < N; i++)  { cout << B[i]
<<", " ; } cout<<"\n";
   cout<<"массив C: "; for ( i = 0; i < N; i++)  { cout << C[i]
<<", " ; } cout<<"\n";
}
```

Для управление нагрузкой может быть использована опция `schedule`:

`schedule (type [,chunk])`

В зависимости от параметров (`type, chunk`) выполнение итераций цикла распределяется между нитями. По умолчанию `chunk =1`. Если опция

schedule не указана, то по умолчанию распределение зависит от реализации (CPU, ОС). Возможные значения type:

- static – блочно-циклическое распределение итераций цикла; размер блока – chunk. Первый блок из chunk итераций выполняет нулевая нить, второй блок – следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение chunk не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера, и полученные порции итераций распределяются между нитями.
- dynamic – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает chunk итераций (по умолчанию chunk=1), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из chunk итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.
- guided – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины chunk (по умолчанию chunk=1) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. Количество итераций в последней порции может оказаться меньше значения chunk.
- auto – способ распределения итераций выбирается компилятором и/или системой выполнения. Параметр chunk при этом не задаётся.
- runtime – способ распределения итераций выбирается во время работы программы по значению переменной среды OMP_SCHEDULE. Параметр chunk при этом не задаётся. Изменить значение переменной OMP_SCHEDULE из программы можно с помощью вызова функции omp_set_schedule().

```
void omp_set_schedule(omp_sched_t type, int chunk);
```

Допустимые значения констант описаны в файле omp.h (omp_lib.h). Как минимум, они должны включать для языка Си следующие варианты:

```
static - 1, dynamic - 2, guided - 3, auto - 4
```

При помощи вызова функции omp_get_schedule() пользователь может узнать текущее значение переменной OMP_SCHEDULE.

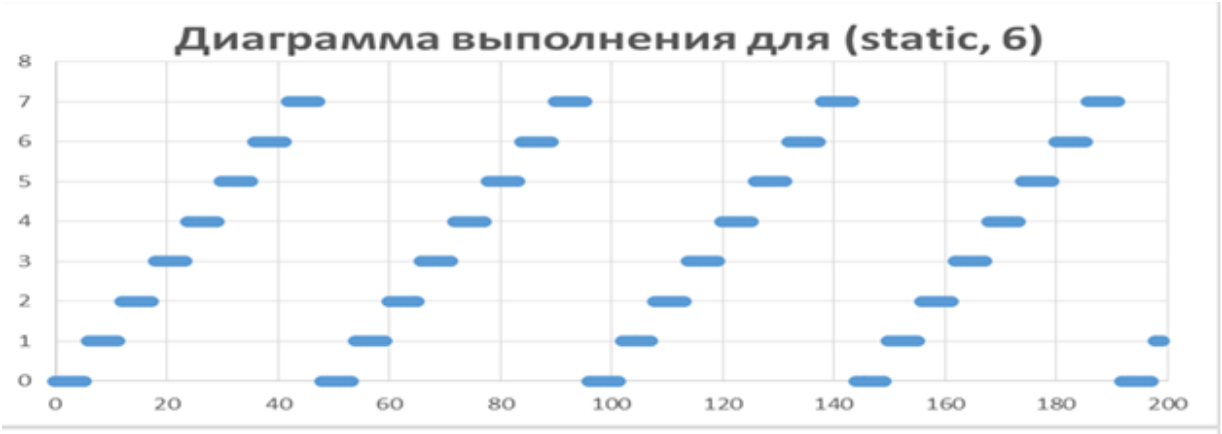
```
void omp_get_schedule(omp_sched_t* type, int* chunk);
```

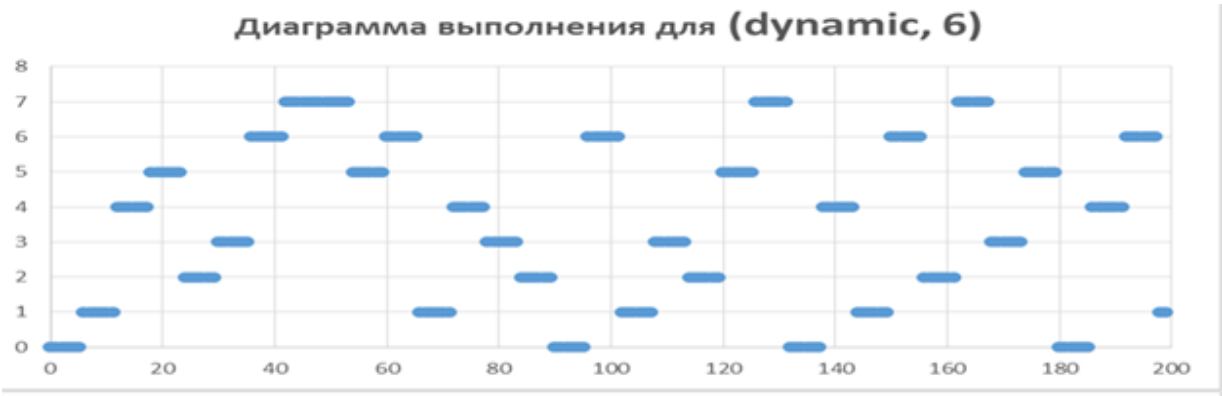
Пример

```
int main(int argc, char *argv[])
{       int i;
#pragma omp parallel private(i)
   {
```

```
#pragma omp for schedule (static, 6)
//#pragma omp for schedule (dynamic, 6)
//#pragma omp for schedule (guided, 6)
      for (i=0; i<200; i++)
      {          printf("Нить %d выполнила итерацию %d\n",
                 omp_get_thread_num(), i);
         Sleep(1);
      }
}
}
```

На рис. 7 видна регулярность распределения порций по 6 итераций при указании (static, 6), более динамичная картина распределения таких же порций при указании (dynamic, 6) и распределение уменьшающимися порциями при указании (guided, 6). В последнем случае размер порций уменьшался с 24 в самом начале цикла до 6 в конце





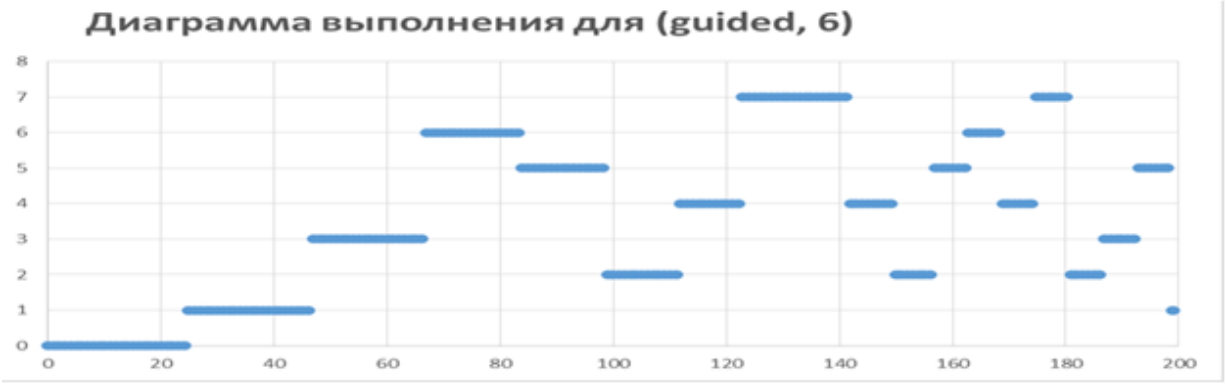


Рис. 7. Распределение итераций между нитями при использовании различных параметров shedule

- `collapse(n)` – `n` последовательных тесновложенных циклов ассоциируется с данной директивой. Для циклов образуется общее пространство итераций, которое делится между тредами. Если опция не задана, то директива относится только к одному – непосредственно следующему за ней циклу.
- `ordered` – опция для указания о том, что в цикле могут встречаться директивы `ordered`.

В этом случае определяется блок внутри тела цикла, который должен выполняться в порядке, установленном в последовательном цикле

- `nowait` – опция для отмены установленной по умолчанию в конце параллельного цикла неявной барьерной синхронизация параллельно работающих нитей. Дальнейшее выполнение происходит только тогда, когда все нити достигнут данной точки (*барьера*). Если подобная задержка не нужна, используют опцию nowait**.** Это позволяет тредам, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными тредами.

Параллельные секции

Директива `sections` используется для задания конечного (неитеративного) параллелизма:

```
#pragma omp sections [опция [[,] опция]...]
```

Эта директива определяет набор независимых секций кода, каждая из которых выполняется своей нитью. Директива `section` задаёт участок кода внутри секции `sections` для выполнения одной нитью:

```
#pragma omp section
```

Перед первым участком кода в блоке `sections` директива `section` не обязательна. Какие именно нити будут задействованы для выполнения какой секции, не специфицируется. Если количество нитей больше количе-

ства секций, то часть нитей для выполнения данного блока секций не будет задействована. Если количество нитей меньше количества секций, то некоторым (или всем) нитям достанется более одной секции.

```
int main(){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
int n;
#pragma omp parallel private(n)
   {       n=omp_get_thread_num();
#pragma omp sections
      {
#pragma omp section
         {          cout<< "Первая секция, процесс "<< n
<<"\n"; }
#pragma omp section
         { cout<< "Вторая секция, процесс "<< n  <<"\n"; }
#pragma omp section
         {     cout<< "Третья секция, процесс "<< n  <<"\n"; }
      }
     cout<< "Параллельная область, процесс "<< n  <<"\n";
   }
}
```

Низкоуровневое распараллеливание

OpenMP предлагает несколько вариантов распределения работы между запущенными нитями. Конструкции распределения работ в OpenMP не порождают новых нитей. Все нити в параллельной области нумеруются последовательными целыми числами от 0 до N–1, где N – количество нитей, выполняющих данную область.

Можно программировать на самом низком уровне, распределяя работу с помощью функций `omp_get_thread_num()` и `omp_get_num_threads()`, возвращающих номер нити и общее количество порождённых нитей в текущей параллельной области, соответственно.

Вызов функции `omp_get_thread_num()` позволяет нити получить свой уникальный номер в текущей параллельной области:

```
int omp_get_thread_num(void);
```

Вызов функции `omp_get_num_threads()` позволяет нити получить количество нитей в текущей параллельной области:

```
int omp_get_num_threads(void);
```

```
int main(){
int count, num;
#pragma omp parallel
   {
      count=omp_get_num_threads();
      num=omp_get_thread_num();
      if (num == 0) printf("Всего нитей: %d\n", count);
      else printf("Нить номер %d\n", num);
   }
}
```

Использование функций `omp_get_thread_num()` и `omp_get_num_threads()` позволяет назначать каждой нити свой кусок кода для выполнения, и таким образом распределять работу между нитями в стиле технологии MPI. Однако использование этого стиля программирования в OpenMP далеко не всегда оправдано – программист в этом случае должен явно организовывать синхронизацию доступа к общим данным. Другие способы распределения работ в OpenMP обеспечивают значительную часть этой работы автоматически.

Синхронизация

Самый распространенный способ синхронизации в OpenMP – барьер. Он оформляется с помощью директивы `barrier`:

```
#pragma omp barrier
```

Нити, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все нити не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше.

Директивы `ordered` определяют блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле:

```
#pragma omp ordered
```

Блок операторов относится к самому внутреннему из объемлющих циклов, а в параллельном цикле должна быть задана опция `ordered`. Нить, выполняющая первую итерацию цикла, выполняет операции данного блока. Нить, выполняющая любую следующую итерацию, должна сначала дождаться выполнения всех операций блока всеми нитями, выполняющими предыдущие итерации. Может использоваться, например, для упорядочения вывода от параллельных нитей.

```
using namespace std;
int main(){
int i, n;
```

```
 #pragma omp parallel private (i, n)
   {       n=omp_get_thread_num();
#pragma omp for ordered
      for (i=0; i<5; i++)
      {
         printf("Нить %d, итерация %d\n", n, i);
#pragma omp ordered
         {
            printf("ordered: Нить %d, итерация %d\n", n, i);
         }
      }
   }
system("Pause");
}

Нить 0, итерация 0
ordered: Нить 0, итерация 0
Нить 3, итерация 3
Нить 4, итерация 4
Нить 2, итерация 2
Нить 1, итерация 1
ordered: Нить 1, итерация 1
ordered: Нить 2, итерация 2
ordered: Нить 3, итерация 3
ordered: Нить 4, итерация 4
Для продолжения нажмите любую клавишу .
```

С помощью директив `critical` оформляется критическая секция программы:

```
#pragma omp critical [(<имя_критической_секции>)]
```

В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение данной критической секции. Как только работавшая нить выйдет из критической секции, одна из заблокированных на входе нитей войдет в неё. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание.

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и тоже имя, рассматриваются единой секцией, даже если находятся в разных па-

раллельных областях. Побочные входы и выходы из критической секции запрещены.

```
int main(int argc, char *argv[]){
int n;
#pragma omp parallel
   {
#pragma omp critical
      {         n=omp_get_thread_num();
              cout<<"Нить "<<  n <<"\n";
      }
   }
}
```

Директива atomic

Частым случаем использования критических секций на практике является обновление общих переменных. Например, если переменная `sum` является общей и оператор вида `sum=sum+expr` находится в параллельной области программы, то при одновременном выполнении данного оператора несколькими нитями можно получить некорректный результат. Чтобы избежать такой ситуации можно воспользоваться механизмом критических секций или специально предусмотренной для таких случаев директивой `atomic`:

```
#pragma omp atomic
```

Данная директива относится к идущему непосредственно за ней оператору присваивания (на используемые в котором конструкции накладываются достаточно понятные ограничения), гарантируя корректную работу с общей переменной, стоящей в его левой части. На время выполнения оператора блокируется доступ к данной переменной всем запущенным в данный момент нитям, кроме нити, выполняющей операцию. Атомарной является только работа с переменной в левой части оператора присваивания, при этом вычисления в правой части не обязаны быть атомарными.

Директива flush

```
#pragma omp flush [(список)]
```

Выполнение данной директивы предполагает, что значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущей нити, будут занесены в основную память; все изменения переменных, сделанные нитью во время работы, станут видимы остальным нитям; если какая-то информация хранится в буферах

вывода, то буферы будут сброшены и т.п. При этом операция производится только с данными вызвавшей нити, данные, изменявшиеся другими нитями, не затрагиваются. Поскольку выполнение данной директивы в полном объёме может повлечь значительных накладных расходов, а в данный момент нужна гарантия согласованного представления не всех, а лишь отдельных переменных, то эти переменные можно явно перечислить в директиве списком. До полного завершения операции никакие действия с перечисленными в ней переменными не могут начаться.

4. Основы технологии MPI [1]

В вычислительных системах с распределенной памятью (рис. 2) процессоры работают независимо друг от друга. Для организации параллельных вычислений в таких условиях необходимо иметь возможность *распределять* вычислительную нагрузку и *организовывать* информационное взаимодействие (*передачу данных)* между процессорами.

В настоящее время MPI (*message passing interface*) является наиболее распространенной технологией программирования для параллельных компьютеров с распределенной памятью. *Основной способ взаимодействия параллельных процессоров в таких системах – передача сообщений друг другу. Стандарт MPI фиксирует интерфейс, который должен соблюдаться как системой программирования на каждой вычислительной платформе, так и при разработке пользовательских программ.*

Для распределения вычислений между процессорами необходимо проанализировать алгоритм решения задачи, выделить информационно независимые фрагменты вычислений, провести их программную реализацию и затем разместить полученные части программы на разных процессорах. В рамках MPI принят более простой подход – *для решения поставленной задачи разрабатывается одна программа и эта единственная программа запускается одновременно на выполнение на всех имеющихся процессорах.* При этом для того, чтобы избежать идентичности вычислений на разных процессорах, можно, во-первых, подставлять разные данные для программы на разных процессорах, а во-вторых, в MPI имеются средства для идентификации процессора, на котором выполняется программа (и тем самым предоставляется возможность организовать различия в вычислениях в зависимости от используемого программой процессора).

Для организации информационного взаимодействия между процессорами в самом минимальном варианте достаточно организовать операции приема и передачи данных, для этого должна существовать техническая возможность коммуникации между процессорами – *каналы* или *линии связи.* В MPI существует множество операций передачи данных. Они обеспечивают разные способы пересылки данных, реализуют практически все необходимые для организации параллельных вычислений коммуникационные операции. Такие возможности являются наиболее сильной стороной MPI.

Таким образом, MPI – это, во-первых, стандарт, которому должны удовлетворять средства организации передачи сообщений, во-вторых, – это программные средства, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта MPI. По стандарту, эти программные средства должны быть организованы в виде библиотек программных модулей (*библиотеки MPI)*.

Достоинства технологии:

• MPI позволяет в значительной степени снизить остроту проблемы переносимости параллельных программ между разными компьютерными системами – параллельная программа, разработанная на алгоритмическом языке C или Fortran с использованием библиотеки MPI, как правило, будет работать на разных вычислительных платформах.

• MPI содействует повышению эффективности параллельных вычислений, поскольку в настоящее время практически для каждого типа вычислительных систем существуют реализации библиотек MPI, в максимальной степени учитывающие возможности используемого компьютерного оборудования.

• MPI уменьшает, в определенном плане, сложность разработки параллельных программ, т. к., с одной стороны, большая часть основных операций передачи данных предусматривается стандартом MPI, а с другой стороны, уже имеется большое количество библиотек параллельных методов, созданных с использованием MPI.

4.1. Основные понятия и определения

Понятие параллельной программы

Под *параллельной программой* в рамках MPI понимается множество одновременно выполняемых *процессов.* Процессы могут выполняться как на разных процессорах, так и на одном процессоре в режиме разделения времени. В предельном случае для выполнения параллельной программы может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы.

Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода *(модель SPMP – single program multiple processes).* Данный программный код, представленный в виде исполняемой программы, должен быть доступен в момент запуска параллельной программы на всех используемых процессорах. Исходный программный код для исполняемой программы разрабатывается на алгоритмических языках C или Fortran с использованием той или иной реализации библиотеки MPI.

Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI-программ и в ходе вычислений меняться не может. Все процессы программы последовательно перенумерованы от 0 до $np - 1$, где np есть общее количество процессов. Номер процесса именуется *рангом* процесса.

Операции передачи данных

Основу MPI составляют операции передачи сообщений. Среди предусмотренных в составе MPI функций различаются *парные (point-to-point*) операции между двумя процессами и *коллективные* (*collective)* коммуникационные действия для одновременного взаимодействия нескольких процессов.

Для выполнения парных операций могут использоваться разные *режимы передачи:* синхронный, блокирующий и др.

Понятие коммуникаторов

Процессы параллельной программы объединяются в группы. Под *коммуникатором* в MPI понимается специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров *(контекст),* используемых при выполнении операций передачи данных.

Как правило, парные операции передачи данных выполняются для процессов, принадлежащих одному и тому же коммуникатору. Коллективные операции применяются одновременно для всех процессов коммуникатора. Важно, что указание используемого коммуникатора является обязательным для операций передачи данных в MPI.

В ходе вычислений могут создаваться новые и удаляться существующие группы процессов и коммуникаторы. Один и тот же процесс может принадлежать разным группам и коммуникаторам. Все имеющиеся в параллельной программе процессы входят в состав создаваемого по умолчанию коммуникатора с идентификатором MPI_COMM_WORLD.

При необходимости передачи данных между процессами из разных групп необходимо создавать глобальный коммуникатор *(inter communicator*).

Типы данных

При выполнении операций передачи сообщений для указания передаваемых или получаемых данных в функциях MPI необходимо указывать *тип* пересылаемых данных. MPI содержит большой набор *базовых типов* данных, во многом совпадающих с типами данных в алгоритмических языках C и Fortran. Кроме того, в MPI имеются возможности для создания новых *производных типов* данных для более точного и краткого описания содержимого пересылаемых сообщений.

4.2. Разработка параллельных программ

Приведем минимально необходимый набор функций MPI, достаточный для разработки простых параллельных программ.

1. Инициализация и завершение MPI программ. *Первой вызываемой функцией* MPI должна быть функция

```
int MPI_Init (int *argc, char ***argv)
```

для инициализации среды выполнения MPI-программы. Параметрами функции являются количество аргументов в командной строке и текст самой командной строки.

Последней вызываемой функцией MPI обязательно должна являться функция

```
int MPI_Finalize (void)
```

Структура параллельной программы, разработанная с использованием MPI, должна иметь следующий вид:

```
#include "mpi.h"
int main(int argc, char *argv[])
{
     <программный код без использования MPI функций>
     MPI_Init(&argc, &argv);
     <программный код с использованием MPI функций>
     MPX_Finalize();
     <программный код без использования MPI функций >
     return 0;
}
```

Файл `mpi.h` содержит определения именованных констант, прототипов функций и типов данных библиотеки MPI.

Функции `MPI_Init` и `MPI_Finalize` являются обязательными и должны быть выполнены (и только один раз) каждым процессом параллельной программы.

Перед вызовом `MPI_Init` может быть использована функция `MPI_Initialized` для определения того, был ли ранее выполнен вызов `MPI_Init`.

2. Определение количества и ранга процессов. Определение *количества процессов* в выполняемой параллельной программе осуществляется при помощи функции

```
int MPI_Comm_size (MPI_Comm comm, int *size)
```

Для определения *ранга процесса* используется функция:

```
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

Как правило, вызов функций `MPI_Comm_size` и `MPI_Comm_rank` выполняется сразу после `MPI_Init`:

```
#include "mpi.h"
int main(int argc, char *argv[])
{
     int ProcNum, ProcRank;
     <программный код без использования MPI функций>
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
     MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
     <программный код с использованием MPI функций>
     MPI_Finalize();
     <программный код без использования MPI функций>
     return 0;
}
```

Коммуникатор `MPI_COMM_WORLD` создается по умолчанию и представляет все процессы выполняемой параллельной программы,

Ранг, получаемый при помощи функции `MPI_Comm_rank`, является рангом процесса, выполнившего вызов этой функции, т. е. переменная `ProcRank` будет принимать различные значения в разных процессах.

3. Передача сообщений. Для *передачи сообщения* процесс-отправитель должен выполнить функцию:

```
int MPI_Send (void *buf, int count, MPI_Datatype type,
              int dest, int tag, MPI_Comm comm),
```

где

`buf` – адрес буфера памяти, в котором располагаются данные отправляемого сообщения,

`count` – количество элементов данных в сообщении,

`type` – тип элементов данных пересылаемого сообщения,

`dest` – ранг процесса, которому отправляется сообщение,

`tag` – значение-тег, используемое для идентификации сообщений,

`comm` – коммуникатор, в рамках которого выполняется передача данных.

Для указания типа пересылаемых данных в *MPI* имеется ряд базовых типов (табл. 1).

Таблица 1

Базовые (предопределенные) типы данных MPI для алгоритмического языка C

MPI_Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Отправляемое сообщение определяется через указание блока памяти *(буфера),* в котором это сообщение располагается. Используемая для указания буфера триада `(buf, count, type)` входит в состав параметров практически всех функций передачи данных.

Процессы, между которыми выполняется передача данных, в обязательном порядке должны принадлежать коммуникатору, указываемому в функции `MPI_Send`.

Параметр `tag` используется только при необходимости различения передаваемых сообщений, в противном случае в качестве значения параметра может быть использовано произвольное целое число.

Сразу же после завершения функции `MPI_Send` процесс-отправитель может начать повторно использовать буфер памяти, в котором располагалось отправляемое сообщение. В момент завершения функции `MPI_Send` состояние самого пересылаемого сообщения может быть совершенно различным – сообщение может располагаться в процессе-отправителе, может находиться в процессе передачи, может храниться в процессе-получателе или же может быть принято процессом-получателем при помощи функции `MPI_Recv`. Тем самым, завершение функции `MPI_Send` означает лишь, что операция передачи начала выполняться и пересылка сообщения будет рано или поздно будет выполнена.

4. Прием сообщений. Для *приема сообщения* процесс-получатель должен выполнить функцию:

```
int MPI_Recv (void *buf, int count, MPI_Datatype type,
              int source, int taq, MPI_Comm comm,
              MPI_Status *status),
```

где

`buf, count, type` – буфер памяти для приема сообщения, назначение каждого отдельного параметра соответствует описанию в MPI_Send,

`source` – ранг процесса, от которого должен быть выполнен прием сообщения,

`tag` – тег сообщения, которое должно быть принято для процесса,

`comm` – коммуникатор, в рамках которого выполняется передача данных,

`status` – указатель на структуру данных с информацией о результате выполнения операции приема данных.

Буфер памяти должен быть достаточным для приема сообщения, а тип элементов передаваемого и принимаемого сообщения должны совпадать; при нехватке памяти часть сообщения будет потеряна и в коде завершения функции будет зафиксирована ошибка переполнения.

При необходимости приема сообщения от любого процесса-отправителя для параметра `source` может быть указано значение `MPI_ANY_SOURCE`.

При необходимости приема сообщения с любым тегом для параметра *`tag`* может быть указано значение `MPI_ANY_TAG`.

Параметр `status` позволяет определить ряд характеристик принятого сообщения:

`status.MPI_SOURCE` – ранг процесса-отправителя принятого сообщения,

`status.MPI_TAG` – тег принятого сообщения.

Функция

```
MPI_Get_count (MPI_Status *status, MPI_Datatype type,
               int *count)
```

возвращает в переменной `count` количество элементов типа *type* в принятом сообщении.

Вызов функции `MPI_Recv` не должен согласовываться со временем вызова соответствующей функции передачи сообщения `MPI_Send` – прием сообщения может быть инициирован до момента, в момент или после момента начала отправки сообщения.

По завершении функции `MPI_Recv` в заданном буфере памяти будет располагаться принятое сообщение. Принципиальный момент здесь состоит в том, что функция `MPI_Recv` является *блокирующей* для процесса-получателя, т. е. его выполнение приостанавливается до завершения работы функции. Таким образом, если по каким-то причинам ожидаемое для прие-

ма сообщение будет отсутствовать, выполнение параллельной программы будет блокировано.

Все функции MPI возвращают в качестве своего значения *код завершения*. При успешном выполнении функции возвращаемый код равен `MPI_SUCCESS`. Другие значения кода завершения свидетельствуют об обнаружении тех или иных ошибочных ситуаций в ходе выполнения функций. Для выяснения типа обнаруженной ошибки используются предопределенные именованные константы, среди которых:

`MPI_ERR_BUFFER` – неправильный указатель на буфер,

`MPI_ERR_COMM` – неправильный коммуникатор,

`MPI_ERR_RANK` – неправильный ранг процесса,

и др. (полный список констант для проверки кода завершения содержится в файле mpi.h)

Определение времени выполнение MPI-программы

Используемые обычно средства для измерения времени работы программ зависят, как правило, от аппаратной платформы, операционной системы, алгоритмического языка и т. п. Стандарт MPI включает определение специальных функций для измерения времени, использование которых позволяет устранить зависимость от среды выполнения параллельных программ.

Получение времени текущего момента выполнения программы обеспечивается при помощи функции

```
double MPI_Wtime(void)
```

результат вызова которой есть количество секунд, прошедшее от некоторого определенного момента времени в прошлом. Этот момент времени в прошлом, от которого происходит отсчет секунд, может зависеть от среды реализации библиотеки MPI. Для ухода от такой зависимости функцию *`MPI_Wtime`* следует использовать только для определения длительности выполнения тех или иных фрагментов кода параллельных программ. Возможная схема применения функции *`MPI_Wtime`*:

```
double tl, t2, dt;
tl = MPI_Wtime();
. . .
t2 = MPI_Wtime();
dt = t2 – tl;
```

Точность измерения времени также может зависеть от среды выполнения параллельной программы. Для определения текущего значения точности может быть использована функция

```
double MPI_Wtick(void)
```

позволяющая определить время в секундах между двумя последовательными показателями времени аппаратного таймера используемой компьютерной системы.

4.3. Коллективные операции передачи данных

Функции *MPI_Send* и MPI_Recv, обеспечивают возможность выполнения *парных операций* передачи данных между двумя процессами параллельной программы. Для выполнения коммуникационных *коллективных операций,* в которых принимают участие все процессы коммуникатора, в MPI предусмотрен специальный набор функций.

В качестве примера рассмотрим пример суммирования элементов вектора arr.

1. Передача данных от одного процесса всем процессам программы. Первая проблема при выполнении рассмотренного параллельного алгоритма суммирования состоит в необходимости передачи значений вектора *x* всем процессам параллельной программы. Для решения этой проблемы можно воспользоваться рассмотренными ранее функциями парных операций передачи данных:

```
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
for (int i = 1; i < ProcNum; i++)
     MPI_Send(&arr, size, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
```

Однако такое решение будет крайне неэффективным, поскольку повторение операций передачи приводит к суммированию затрат (латентностей) на подготовку передаваемых сообщений. Кроме того, данная операция может быть выполнена за $log_2 p$ итераций передачи данных.

Достижение эффективного выполнения операции передачи данных от одного процесса всем процессам программы (*широковещательная рассылка* данных) может быть обеспечено при помощи функции MPI:

```
int MPI_Bcast (void *buf,int count,MPI_Datatype type,
               int root, MPI_Comm comm),
```

где

buf, count, type – буфер памяти с отправляемым сообщением (для процесса с рангом root), и для приема сообщений для всех остальных процессов,

root – ранг процесса, выполняющего рассылку данных,

comm – коммуникатор, в рамках которого выполняется передача данных.

Функция MPI_Bcast осуществляет рассылку данных из буфера buf, содержащего count элементов типа type с процесса, имеющего номер root, всем процессам, входящим в коммуникатор comm (см. рис. 8).

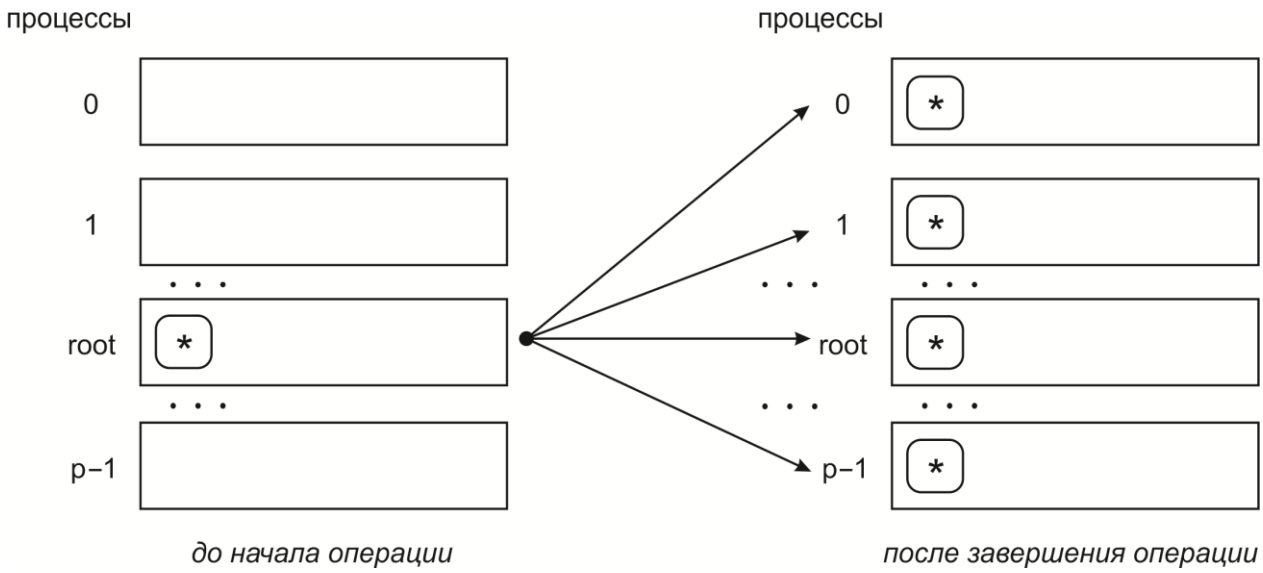


Рис. 8. Общая схема операции передачи данных от одного процесса всем процессам

Функция `MPI_Bcast` определяет коллективную операцию и, тем самым, при выполнении необходимых рассылок данных вызов функции `MPI_Bcast` должен быть осуществлен всеми процессами указываемого коммуникатора (см. далее пример программы).

Указываемый в функции `MPI_Bcast` буфер памяти имеет различное назначение в разных процессах. Для процесса с рангом `root`, с которого осуществляется рассылка данных, в этом буфере должно находиться рассылаемое сообщение. Для всех остальных процессов указываемый буфер предназначен для приема передаваемых данных.

```
#include "mpi.h"
int main(int argc, char* argv[])
{
     const int size = 110;
     int arr[size], TotalSum, ProcSum = 0;
     int ProcRank, ProcNum;
     MPI_Status Status;
     // инициализация
     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
     MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
     // подготовка данных
     if (ProcRank == 0) DataInitialization(x,N);
     // рассылка данных на все процессы
     MPI_Bcast(arr, size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
     // вычисление частичной суммы на каждом из процессов
```

```
    // на каждом процессе суммируются элементы вектора
    // arr от i1 до i2
    int k = size / ProcNum;
    int i1 = k * ProcRank;
    int i2 = k * (ProcRank + 1);
    if (ProcRank == ProcNum-1) i2 = N;
    for (int i = i1; i < i2; i++)
        ProcSum = ProcSum + arr[i];
    // сборка частичных сумм на процессе с рангом 0
    if (ProcRank == 0)
    {
        TotalSum = ProcSum;
        for (int i=1; i < ProcNum; i++ )
        {
            MPI_Recv(&ProcSum, 1, MPI_DOUBLE,
            MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &Status);
            TotalSum = TotalSum + ProcSum;
        }
    }
    else // все процессы отсылают свои частичные суммы
        MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD);
    // вывод результата
    if (ProcRank == 0)
        cout << TotalSum << endl;
    MPI_Finalize();
}
```

2. Передача данных от всех процессов одному процессу. Операции редукции. В рассмотренной программе суммирования числовых значений имеющаяся процедура сбора и последующего суммирования данных является примером часто выполняемой коллективной *операции передачи данных от всех процессов одному процессу.* В этой операции над собираемыми значениями осуществляется та или иная обработка данных (для подчеркивания последнего момента данная операция еще именуется *операцией редукции данных).* Как и ранее, реализация операции редукции при помощи обычных парных операций передачи данных является неэффективной и достаточно трудоемкой. Для наилучшего выполнения действий, связанных с редукцией данных, в MPI предусмотрена функция

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

где

sendbuf – буфер памяти с отправляемый сообщением,

`recvbuf` – буфер памяти для результирующего сообщения (только для процесса с рангом root),

`count` – количество элементов в сообщениях,

`type` – тип элементов сообщений,

`op` – операция, которая должна быть выполнена над данными,

`root` – ранг процесса, на котором должен быть получен результат,

`comm` – коммуникатор, в рамках которого выполняется операция.

В качестве операций редукции данных могут быть использованы предопределенные в MPI операции (табл. 2).

Помимо данного стандартного набора операций могут быть определены и новые дополнительные операции непосредственно самим пользователем библиотеки MPI.

Общая схема выполнения операции сбора и обработки данных на одном процессоре показана на рис. 9. Элементы получаемого сообщения на процессе *`root`* представляют собой результаты обработки соответствующих элементов передаваемых процессами сообщений, т. е.

$$y_j = \overset{n-1}{\underset{i=0}{\otimes}} x_{ij}, \quad 0 \le j < n,$$

где $\otimes$ есть операция, задаваемая при вызове функции *`MPI_Reduce`* (для пояснения на рис. 4. показан пример выполнения операции редукции данных).

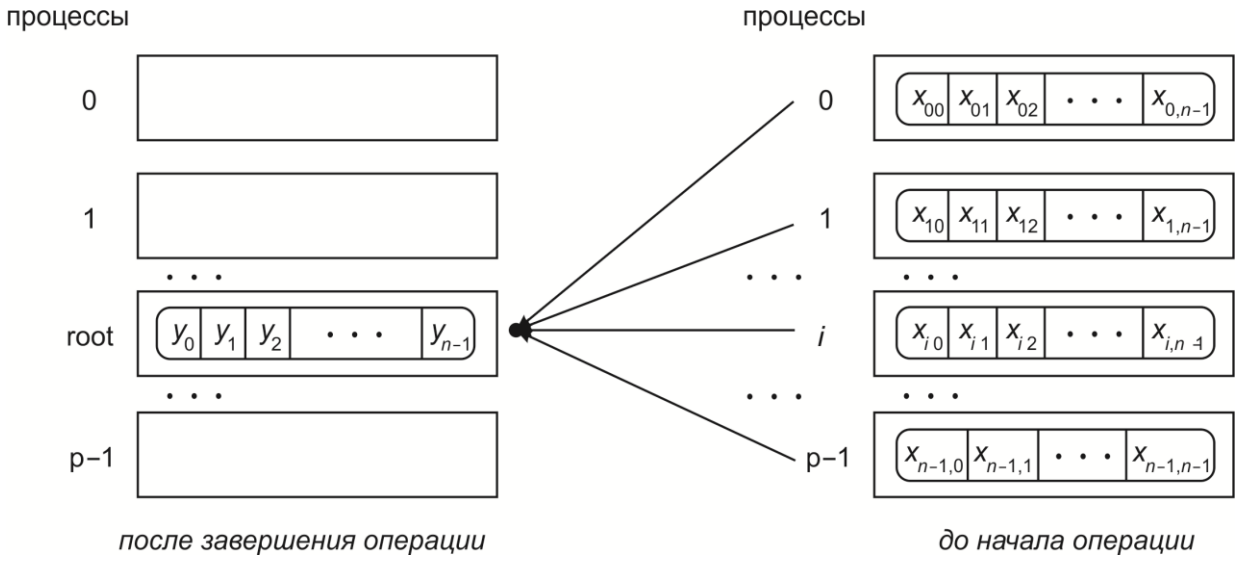


Рис. 9. Общая схема операции сбора и обработки на одном процессе данных от всех процессов

Таблица 2

Базовые (предопределенные) типы операций MPI для функций редукции данных

Операция	Описание
MPI_MAX	Определение максимального значения
MPI_MIN	Определение минимального значения
MPI_SUM	Определение суммы значений
MPI_PROD	Определение произведения значений
MPI_LAND	Выполнение логической операции «И» над значениями сообщений
MPI_BAND	Выполнение битовой операции «И» над значениями сообщений
MPI_LOR	Выполнение логической операции «ИЛИ» над значениями сообщений
MPI_BOR	Выполнение битовой операции «ИЛИ» над значениями сообщений
MPI_LXOR	Выполнение логической операции исключающего «ИЛИ» над значениями сообщений
MPI_BXOR	Выполнение битовой операции исключающего «ИЛИ» над значениями сообщений
MPI_MAXLOC	Определение максимальных значений и их индексов
MPI_MINLOC	Определение минимальных значений и их индексов

Функция MPI_Reduce определяет коллективную операцию и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора, все вызовы функции должны содержать одинаковые значения параметров count, type, op, root, comm.

Передача сообщений должна быть выполнена всеми процессами, результат операции будет получен только процессом с рангом root.

Выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений. Так, например, если сообщения содержат по два элемента данных и выполняется операция суммирования MPI_SUM, то результат также будет состоять из двух значений, первое из которых будет содержать сумму первых элементов всех отправленных сообщений, а второе значение будет равно сумме вторых элементов сообщений соответственно.

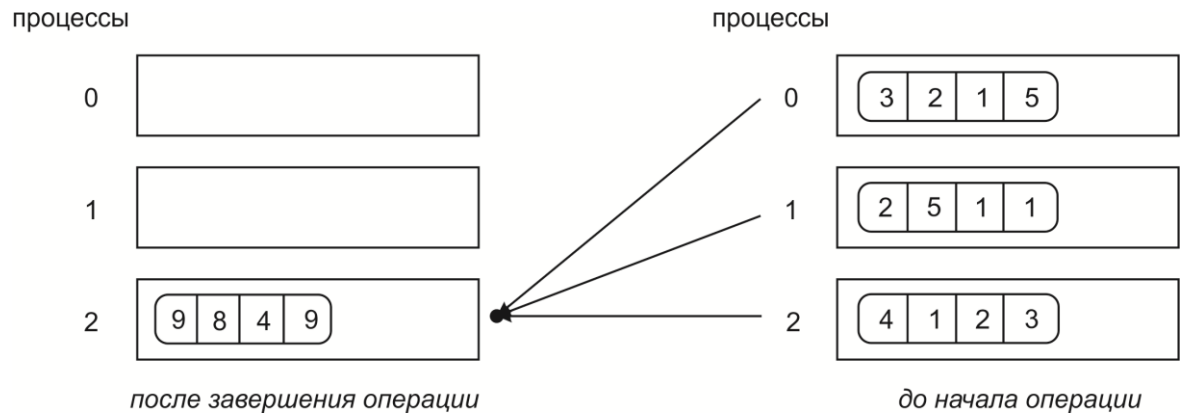


Рис. 10. Пример выполнения операции редукции при суммировании пересылаемых данных для трех процессов (в каждом сообщении 4 элемента, сообщения собираются на процессе с рангом 2)

Весь программный код, выделенный рамкой, теперь может быть заменен на вызов одной лишь функции `MPI_Reduce`:

```
// сборка частичных сумм на процессе с рангом 0
MPI_Reduce (&ProcSum, &TotalSum, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD);
```

3. Синхронизация вычислений. В ряде ситуаций независимо выполняемые в процессах вычисления необходимо синхронизировать. Так, например, для измерения времени начала работы параллельной программы необходимо, чтобы для всех процессов одновременно были завершены все подготовительные действия, перед окончанием работы программы все процессы должны завершить свои вычисления и т. п.

Синхронизация процессов, т. е. одновременное достижение процессами тех или иных точек процесса вычислений, обеспечивается при помощи функции MPI:

```
int MPI_Barrier(MPI_Comm comm);
```

Функция `MPI_Barrier` определяет коллективную операцию и, тем самым, при использовании должна вызываться всеми процессами используемого коммуникатора. При вызове функции `MPI_Barrier` выполнение процесса блокируется, продолжение вычислений процесса произойдет только после вызова функции `MPI_Barrier` всеми процессами коммуникатора.

4.4. Операции передачи данных между двумя процессами

Режимы передачи данных

Рассмотренная ранее функция `MPI_Send` обеспечивает так называемый *стандартный (Standard*) режим отправки сообщений, при котором:

– на время выполнения функции процесс-отправитель сообщения блокируется,

– после завершения функции буфер может быть использован повторно,

– состояние отправленного сообщения может быть различным – сообщение может располагаться в процессе-отправителе, может находиться в процессе передачи, может храниться в процессе-получателе или же может быть принято процессом-получателем при помощи функции `MPI_Recv`.

Кроме стандартного режима в MPI предусматриваются следующие дополнительные *режимы передачи* сообщений:

Синхронный (Synchronous) *режим* состоит в том, что завершение функции отправки сообщения происходит только при получении от процесса-получателя подтверждения о начале приема отправленного сообщения, отправленное сообщение или полностью принято процессом- получателем, или находится в состоянии приема.

Буферизованный (Buffered) режим предполагает использование дополнительных системных буферов для копирования в них отправляемых сообщений; как результат, функция отправки сообщения завершается сразу же после копирования сообщения в системный буфер.

Режим передачи по готовности (Ready) может быть использован только, если операция приема сообщения уже инициирована. Буфер сообщения после завершения функции отправки сообщения может быть повторно использован.

Для именования функций отправки сообщения для разных режимов выполнения в MPI используется название функции `MPI_Send`, к которому как префикс добавляется начальный символ названия соответствующего режима работы, т. е.

`MPI_Ssend` – функция отправки сообщения в синхронном режиме,

`MPI_Bsend` – функция отправки сообщения в буферизованном режиме,

`MPI_Rsend` – функция отправки сообщения в режиме по готовности.

Список параметров всех перечисленных функций совпадает с составом параметров функции *`MPI_Send`*.

Для использования буферизованного режима передачи должны быть создан и передан MPI буфер памяти для буферизации сообщений. Используемая для этого функция имеет вид

```
int MPI_Buffer_attach (void *buf, int size)
```

`buf` – буфер памяти для буферизации сообщений,

`size` – размер буфера.

После завершения работы с буфером он должен быть отключен от MPI при помощи функции

```
int MPI_Buffer_detach (void *buf, int *size)
```

По практическому использованию режимов можно привести следующие рекомендации:

1. Режим передачи по готовности формально является наиболее быстрым, но используется достаточно редко, т.к. обычно сложно гарантировать готовность операции приема.

2. Стандартный и буферизованный режимы также выполняются достаточно быстро, но часто приводят к большим расходам ресурсов (памяти) – в целом могут быть рекомендованы для передачи коротких сообщений.

3. Синхронный режим является наиболее медленным, т. к. требует подтверждения приема. В то же время, этот режим наиболее надежен – можно рекомендовать его для передачи длинных сообщений.

Для функции приема MPI_Recv не существует различных режимов работы.

5. Общие сведения о технологии CUDA

Изначально класс устройств графических разрабатывался для обработки графики. Техника совершенствовалась, GPU наращивали производительность, и в какой-то момент оказалось, что GPU можно успешно использовать не только для задач компьютерной графики, но и как математический сопроцессор для CPU, получая при этом существенный прирост в производительности.

Этот класс технологий получил название GPGPU (General-purpose computing for graphics processing units) – использование графического процессора для различных вычислительных задач. Популярный представитель этого класса – CUDA (Compute Unified Device Architecture), который был впервые представлен компанией NVIDIA в 2007 году. CUDA может использоваться на GPU производства NVIDIA, таких как GeForce, Quadro, Tesla, Fermi, Kepler, Maxwell, Pascal и Volta. Некоторые из этих устройств не имеют видеовыхода, и эти видеокарты предназначены исключительно для высокопроизводительных вычислений.

CUDA работает только с устройствами производства NVIDIA, но существуют и другие аналогичные технологии, например, OpenCL, AMD FireStream, CUDAfy и PyCUDA.

CPU (central processing unit) – центральный процессор, основная функция которого выполнение цепочки инструкций за максимально короткое время. CPU спроектирован таким образом, чтобы выполнять несколько цепочек одновременно или разбивать один поток инструкций на несколько и, после выполнения их по отдельности, сливать их снова в одну, в правильном порядке. Каждая инструкция в потоке зависит от следующих за ней. Именно поэтому в CPU так мало исполнительных блоков, а весь упор делается на скорость выполнения и уменьшение простоев, что достигается при помощи кэш-памяти и конвейера.

GPU (graphics processing unit) – графический процессор, основная функция которого рендеринг 3D-графики и визуальных эффектов. GPU получает на вход полигоны, а после проведения над ними необходимых математических и логических операций выдаёт координаты пикселей. По сути, работа GPU сводится к оперированию над огромным количеством независимых между собой задач. Поэтому он содержит огромное количество исполнительных блоков – в современных GPU их 2048 и более.

Отличие CPU от GPU :

- **Доступ к памяти**. В GPU он ориентирован на то, что если из памяти читается элемент текстуры, то через некоторое время настанет очередь и соседних элементов. С записью ситуация аналогичная.

- **Размер кэш-памяти**. Графическому процессору, в отличие от универсальных процессоров, не нужна кэш-память большого размера. Для текстур требуются лишь 128–256 килобайт.
- **Поддержка многопоточности**. Центральный процессор исполняет 1–2 потока вычислений на одно ядро, а графический процессор может поддерживать несколько тысяч потоков на каждый мультипроцессор, которых в чипе несколько штук. И если переключение с одного потока на другой для CPU стоит сотни тактов, то GPU переключает несколько потоков за один такт.
- **Архитектура**. В CPU большая часть площади чипа занята под буферы команд, аппаратное предсказание ветвления и огромные объемы кэш-памяти, а в GPU большая часть площади занята исполнительными блоками.

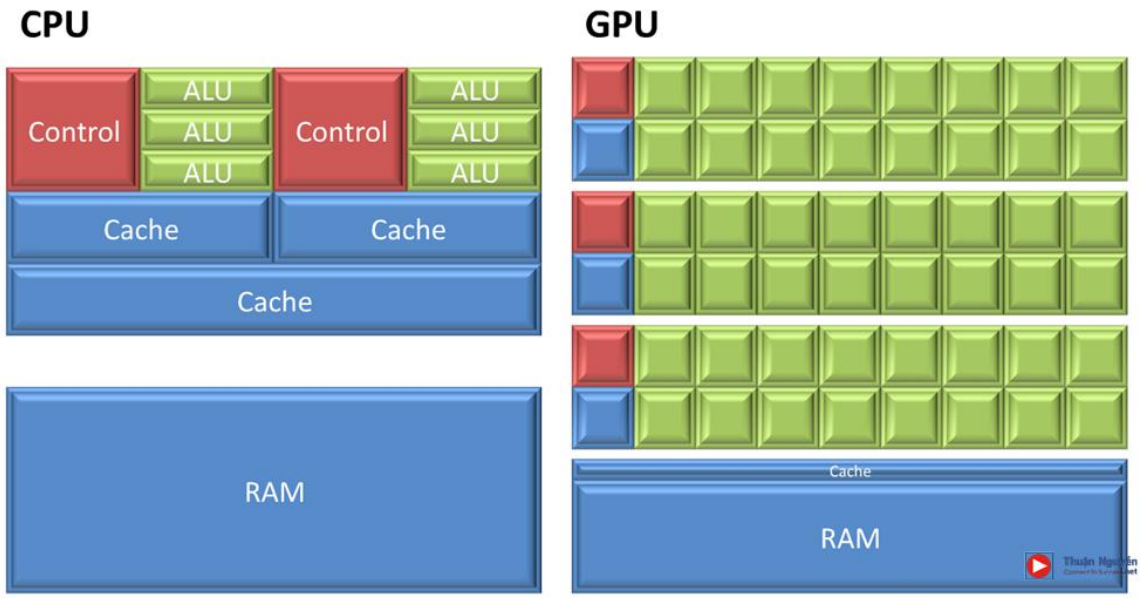


Рис. 11. Структура CPU и GPU

Параллельная обработка данных распределяет элементы данных на параллельно обрабатываемых потоках. GPU особенно хорошо подходит для решения проблем, которые могут быть выражены как вычислений данными параллельно – та же программа выполняется на многих элементов данных параллельно – с высокой интенсивностью – арифметическое отношение арифметических операций к операциям с памятью.

Примеры:

- Сложение векторов: в ядре складывается пара элементов, количество потоков равно длине векторов.
- Вычисление матричного произведения по определению: в ядре вычисляется один элемент результата, количество потоков равно количеству элементов результирующей матрицы.

• Сложение векторов: в ядре складывается пара элементов, количество потоков равно длине векторов.

• Вычисление матричного произведения по определению: в ядре вычисляется один элемент результата, количество потоков равно количеству элементов результирующей матрицы.

• Численное решение ДУ с помощью явной разностной схемы: в ядре пересчитывается одно сеточное значение. е пересчитывается одно сеточное значение.

5.1. Основные понятия технологии CUDA

Хост (Host) – центральный процессор, управляющий выполнением программы.

Устройство (Device) – видеоадаптер, выступающий в роли сопроцессора центрального процессора.

Ядро (Kernel) – параллельная часть алгоритма, выполняется на гриде.

Грид (Grid) – объединение блоков, которые выполняются на одном устройстве. Состоит из блоков.

Блок (Block) – объединение тредов, которое выполняется целиком на одном потоковом мультипроцессоре. Имеет свой уникальный идентификатор внутри грида.

Тред (Thread, поток, нить) – единица выполнения программы. Имеет свой уникальный идентификатор внутри блока.

Варп (Warp) – 32 последовательно идущих треда, выполняется физически одновременно.

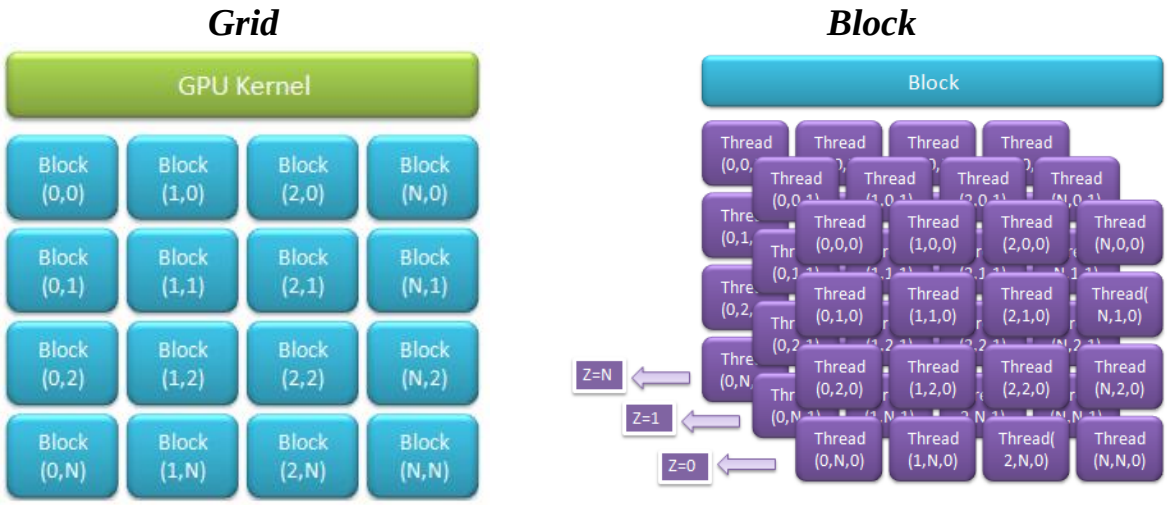


Рис. 12. Организация

Каждый блок потоков может быть распределён на любом количестве процессорных ядер, в любом порядке, параллельно или последовательно,

так что скомпилированная CUDA-программа может выполняться на любом количестве процессорных ядер

5.2. Расширение языка Си

Спецификаторы функций определяют, как и откуда буду вызываться функции:

`__host__` – выполнятся на CPU, вызывается с CPU.

`__global__` – выполняется на GPU, вызывается с CPU.

`__device__` – выполняется на GPU, вызывается с GPU.

В язык добавлены следующие ***специальные переменные***

- `gridDim` – размер grid'а (имеет тип dim3)
- `blockDim` – размер блока (имеет тип dim3)
- `blockIdx` – индекс текущего блока в grid'е (имеет тип uint3)
- `threadIdx` – индекс текущей нити в блоке (имеет тип uint3)
- `warpSize` – размер warp'а (имеет тип int)

В язык добавлены 1/2/3/4-мерные ***вектора из базовых типов*** – `char1`, `char2`, `char3`, `char4`, `uchar1`, `uchar2`, `uchar3`, `uchar4`, `short1`, `short2`, `short3`, `short4`, `ushort1`, `ushort2`, `ushort3`, `ushort4`, `int1`, `int2`, `int3`, `int4`, `uint1`, `uint2`, `uint3`, `uint4`, `long1`, `long2`, `long3`, `long4`, `ulong1`, `ulong2`, `ulong3`, `ulong4`, `float1`, `float2`, `float3`, `float2` и `double2`.

Обращение к компонентам вектора идет по именам – `x`, `y`, `z` и `w`.

5.3. Последовательность выполнения программы с использованием CUDA

1. Получить данные для расчетов.
2. Скопировать эти данные в GPU память.
3. Произвести вычисление в GPU.
4. Скопировать вычисленные данные из GPU памяти в ОЗУ.
5. Вывести результаты.
6. Высвободить используемые ресурсы.

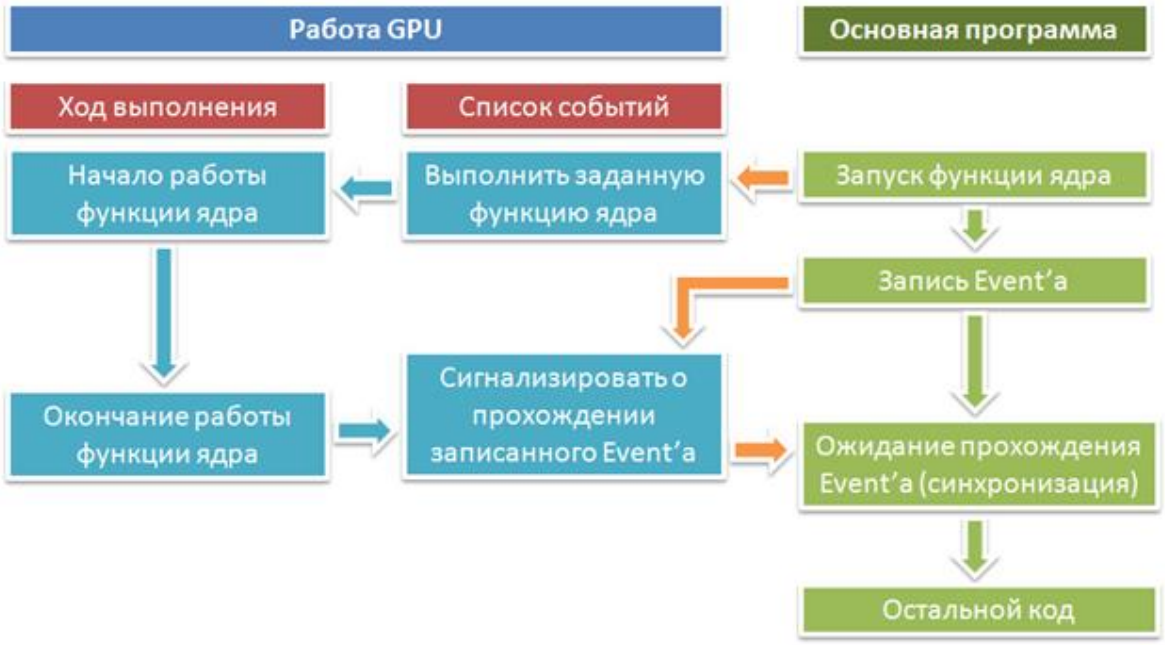


Рис. 13. Взаимодействие устройств при использовании технологии CUDA

Копирование данных в память GPU

1. Выделение памяти – функция `cudaMalloc`:

```
cudaError_t cudaMalloc( void** devPtr, size_t count ),
```

где `devPtr` – указатель, в который записывается адрес выделенной памяти,
`count` – размер выделяемой памяти в байтах.
Возвращает:

- `cudaSuccess` – при удачном выделении памяти
- `cudaErrorMemoryAllocation` – при ошибке выделения памяти

Пример:

```
cudaMalloc((void**)&devVec1, sizeof(float) * SIZE);
```

2. Для копирования – функция `cudaMemcpy`:

```
cudaError_t cudaMemcpy(void *dst, const void *src ,size_t count,
                       enum cudaMemcpyKind),
```

где

`dst` – указатель, содержащий адрес места-назначения копирования,
`src` – указатель, содержащий адрес источника копирования,
`count` – размер копируемого ресурса в байтах,
`cudaMemcpyKind` – перечисление, указывающее направление копирования (может быть `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyHostToHost`, `cudaMemcpyDeviceToDevice`).

Возвращает:

`cudaSuccess` – при удачном копировании,

`cudaErrorInvalidValue` – неверные параметры аргумента (например, размер копирования отрицателен),

`cudaErrorInvalidDevicePointer` – неверный указатель памяти в видеокарте,

`cudaErrorInvalidMemcpyDirection` – неверное направление (например, перепутан источник и место-назначение копирования).

Пример:

```
cudaMemcpy(devVec1, vec1, sizeof(float) *SIZE,
           cudaMemcpyHostToDevice);
```

5.4. Вычисления в GPU

Синтаксис запуска ядра имеет следующий вид:

```
myKernelFunc<<<gridSize, blockSize, sharedMemSize, cudaStream
>>>(float*param1, float*param2),
```

где

`myKernelFunc` – функция ядра (спецификатор `__global__`)

`gridSize` – размерность сетки блоков (`dim3`), выделенную для расчетов,

`blockSize` – размер блока (`dim3`), выделенного для расчетов,

`sharedMemSize` – размер дополнительной памяти, выделяемой при запуске ядра,

`cudaStream` – переменная `cudaStream_t`, задающая поток, в котором будет произведен вызов.

Некоторые переменные при вызове ядра можно опускать (`sharedMemSize` *и* `cudaStream`).

`cudaDeviceSynchronize()` – функция используется для синхронизации потоков.

Пример:

```
addKernel<<<1, size>>>(dev_c, dev_a, dev_b);
cudaDeviceSynchronize();
```

Освобождение используемых ресурсов

Прототип:

```
cudaError_t cudaFree(void *devPtr),
```

где

`*devPtr` – указатель, в который записывается адрес выделенной памяти

Пример:

```
cudaFree(dev_c);
```

Пример 1. Программа для вывода на экран характеристик видеокарты.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <windows.h>
#include <iostream>
#include <stdio.h>
int main()
{    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
int deviceCount;
cudaGetDeviceCount(&deviceCount);
for(int device = 0; device < deviceCount; device++) {
   cudaDeviceProp deviceProp;
   cudaGetDeviceProperties(&deviceProp, device);
printf("Номер устройства: %d\n",device);
printf("Имя устройства: %s\n", deviceProp.name);
printf("Объем глобальной памяти: %d Мбайт\n",
 deviceProp.totalGlobalMem/1024/1024);
printf("Объем shared-памяти в блоке : %d\n",
deviceProp.sharedMemPerBlock);
printf("Объем регистровой памяти: %d\n", de-
viceProp.regsPerBlock);
printf("Размер warp'a: %d\n", deviceProp.warpSize);
printf("Размер шага памяти: %d\n", deviceProp.memPitch);
printf("Макс количество потоков в блоке: %d\n",
deviceProp.maxThreadsPerBlock);
printf("Максимальная размерность потока: x = %d, y = %d, z =
%d\n",
      deviceProp.maxThreadsDim[0],  deviceProp.maxThreadsDim[1],
deviceProp.maxThreadsDim[2]);
   printf("Максимальный размер сетки: x = %d, y = %d, z = %d\n",
      deviceProp.maxGridSize[0],    deviceProp.maxGridSize[1],
deviceProp.maxGridSize[2]);
printf("Тактовая частота: %d кГц\n", deviceProp.clockRate);
printf("Общий объем константной памяти: %d\n",
deviceProp.totalConstMem);
printf("Вычислительная мощность: %d.%d\n", deviceProp.major,
 deviceProp.minor);
printf("Величина текстурного выравнивания : %d\n",
 deviceProp.textureAlignment);
printf("Количество процессоров: %d\n",
```

```
deviceProp.multiProcessorCount);

system("Pause");
    return 0;
}
}
```

Пример 2.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
cudaError_t addWithCuda(int *c, const int *a, const int *b, un-
signed int size);

__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
int main() {
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

cudaError_t  addWithCuda(c, a, b, arraySize);

printf("{1,2,3,4,5} + {10,20,30,40,50} =
{%d,%d,%d,%d,%d}\n",c[0], c[1], c[2], c[3], c[4]);

cudaDeviceReset();
return 0;}
cudaError_t addWithCuda(int *c, const int *a, const int *b, un-
signed int size)
{
 int *dev_a = 0;    int *dev_b = 0;     int *dev_c = 0;
cudaSetDevice(0);
cudaMalloc((void**)&dev_c, size * sizeof(int));
cudaMalloc((void**)&dev_a, size * sizeof(int));
cudaMalloc((void**)&dev_b, size * sizeof(int));
cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDe-
vice);
cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDe-
vice);
```

```
addKernel<<<1, size>>>(dev_c, dev_a, dev_b);
cudaDeviceSynchronize();
cudaMemcpy(c, dev_c,   size *
sizeof(int),cudaMemcpyDeviceToHost);
cudaFree(dev_c);
cudaFree(dev_a);
cudaFree(dev_b);
}
```

Оценка времени, затраченного на вычисления

Для оценки времени вычисления технология представляет следующие средства:

`cudaEventCreate()` – создание временных меток `start`, `stop`;

`cudaEventRecord()` – фиксация времени старта;

`cudaEventRecord()` – фиксация времени завершения.

`cudaEventSynchronize()` – синхронизация асинхронных процессов;

`cudaEventElapsedTime()` – вычисление разницы во времени.

Пример.

```
cudaEvent_t start, stop;
float gpuTime;
cudaEventCreate( &start );
cudaEventCreate( &stop );
cudaEventRecord( start, 0 );
 ...
cudaEventRecord( stop, 0 ) ;
 cudaEventSynchronize( stop ) );
cudaEventElapsedTime( &gpuTime, start, stop ) ;
printf("time spent executing by the GPU: %.2f millseconds\n",
     gpuTime );
cudaEventDestroy( start) ;
cudaEventDestroy( stop) ;
```

Организация памяти устройства

В CUDA имеется всего шесть типов памяти, которые различаются объемом, скоростью доступа и областью доступа:

1. Регистровая память (Register).
2. Локальная память (Local memory).
3. Глобальная память (Global memory).
4. Разделяемая память (Shared memory).
5. Константная память (Constant memory).

6. Текстурная память (Texture memory)

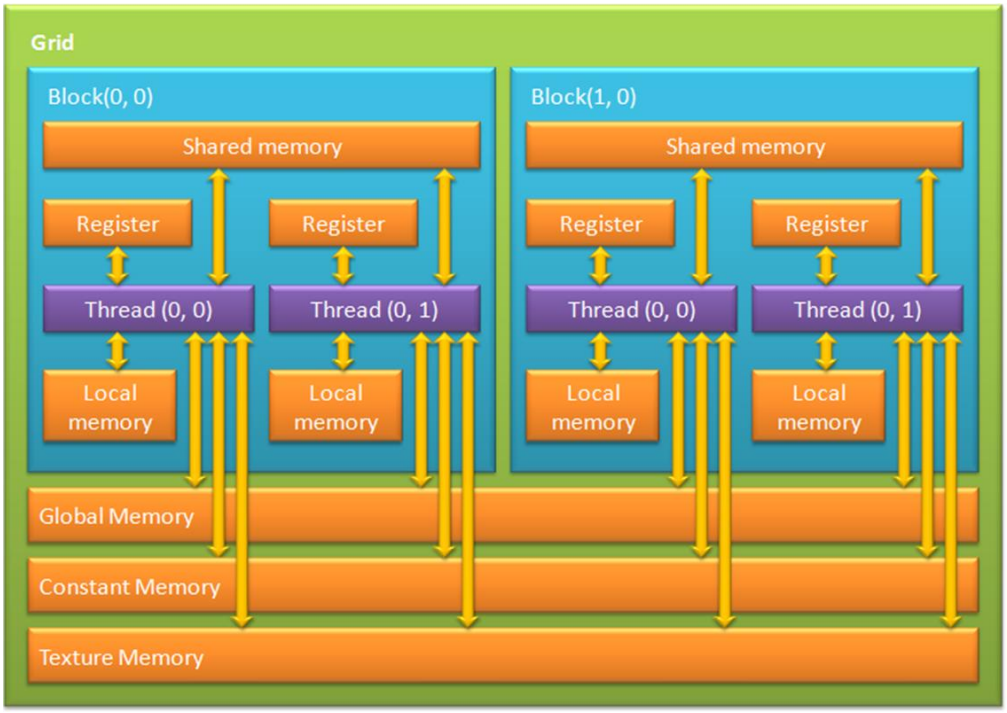


Рис. 14. Типы памяти при использовании технологии CUDA

Тип памяти	Доступ	Уровень выделения (доступа)	Скорость работы
регистры	R/W	per-thread	высокая (on chip)
local	R/W	per-thread	низкая (DRAM)
shared	R/W	per-block	высокая (on-chip)
global	R/W	per-grid	низкая(DRAM)
constant	R/O	per-grid	высокая(on chip L1 cache)
texture	R/O	per-grid	высокая(on chip L1 cache)

Спецификаторы переменных для задания размещения в памяти GPU

`__device__` – переменная находится в глобальной памяти и доступна всем нитям,

`__constant__` – размещена в константной памяти, откуда она может быть только прочитана любой из нитей,

`__shared__` – переменная – в разделяемой памяти, где доступна только всем нитям «своего» блока.

Регистровая память (register)

Является самой быстрой из всех видов. Рассчитать количество регистров, доступных одной нити GPU – для этого необходимо разделить общее число регистров на произведение количества нитей в блоке и количества блоков в гриде. Определить количество регистров доступных GPU можно с помощью уже функции `cudaGetDeviceProperties`.
Расчет количества регистров, доступных одной нити GPU при вызове функций ядра:

```
myKernelFunc<<< gridSize, blockSize, sharedMemSize, cudaStream
>>> (float* param1,float * param2),
```

$$\text{Количество регистров одной нити} = \frac{\text{общее число регистров}}{gridSize * blockSize}.$$

Все регистры GPU 32 разрядные. В CUDA нет явных способов использования регистровой памяти. Это определяет компилятор.

Скалярные переменные без квалификатора помещаются в регистры, в случае недостатка регистров – выгружаются в локальную память потока. Регистровая память:

- Расположение: multiprocessor
- Кэшируемость: no
- Уровень выделения: on chip
- Доступ: GPU – R/W, CPU – no
- Скорость работы: «максимальная»
- Уровень доступа: per-thread, SP
- Время жизни: thread

Локальная память

Локальная память (local memory) может быть использована компилятором при большом количестве локальных переменных в какой-либо функции. По скоростным характеристикам локальная память значительно медленнее, чем регистровая. В документации от nVidia рекомендуется использовать локальную память только в самых необходимых случаях. Локальная память:

- Расположение: DRAM
- Кэшируемость: no
- Уровень выделения: DRAM
- Доступ: GPU – R/W, CPU – no

- Скорость работы: низкая (латентность – 400–600 тактов)
- Уровень доступа: per-thread, SP
- Время жизни: thread

Примечание: нет механизмов, позволяющих явно запретить компилятору использование локальной памяти для конкретных переменных. ***Массивы без квалификатора размещаются в локальной памяти потока.***

Глобальная память

Глобальная память (global memory) – самый медленный тип памяти, из доступных GPU. Глобальные переменные можно выделить с помощью

- Динамически с хоста через `cudaMalloc()`
- Статически – глобальная переменная с атрибутом `__device__`
- Динамически из ядер через `malloc()`

Глобальная память в основном служит для хранения больших объемов данных, поступивших на device с host'a, данное перемещение осуществляется с использованием функций cudaMemcpyXXX.

Global memory – DRAM GPU. Обладает высокой пропускной способностью (более 100 ГБ/с) и возможностью произвольной адресации глобальной памяти.

Глобальная память :

- Расположение: DRAM
- Кэшируемость: no
- Уровень выделения: DRAM
- Доступ: GPU – R/W, CPU – R/W
- Скорость работы: низкая (латентность – 400–600 тактов)
- Уровень доступа: per-grid, CPU
- Время жизни: выделяется CPU

Основное назначение: передача данных между CPU и GPU. В алгоритмах, требующих высокой производительности, ***количество операций с глобальной памятью необходимо свести к минимуму.***

Переменные с атрибутами `__device__` и `__constant__` находятся в глобальной области видимости и хранятся в объектном модуле как отдельные символы. Память под них выделяется статически при старте приложения, как и под обычные глобальные переменные. Работать с ними на хосте можно через функции:

- `cudaMemcpyAsync (void* dst, const void* src, size_t count, enum cudaMemcpyKind kind, cudaStream_t)` – асинхронное копирование, только для блокированной памяти
- `cudaMemcpyToSymbol (const T& symbol, const void* src, size_t count, size_t offset, enum cudaMemcpyKind kind)` – копиро-

вание count байт с адреса src на адрес, определяемый переменной symbol, со смещением offset; переменная глобальной памяти или памяти констант задаётся либо переменной с адресом в GPU, либо текстовой строкой с именем переменной; cudaMemcpyKind определяет направление пересылки (перекрытие областей запрещено):

– `cudaMemcpyHostToDevice`
– `cudaMemcpyDeviceToDevice`

• `cudaGetSymbolAddress (void** devPtr, const T& symbol)` – получить указатель на переменную в глобальной памяти GPU; переменная глобальной памяти или памяти констант задаётся либо переменной с адресом в GPU, либо текстовой строкой с именем переменной

• `cudaGetSymbolSize (size_t* size, const T& symbol)` – получить размер переменной в глобальной памяти GPU; переменная глобальной памяти или памяти констант задаётся либо переменной с адресом в GPU, либо текстовой строкой с именем переменной.

Разделяемая память

Разделяемая память (shared memory) относиться к быстрому типу памяти. Разделяемую память рекомендуется использовать для минимизации обращение к глобальной памяти, а так же для хранения локальных переменных функций.

Адресация разделяемой памяти между нитями потока одинакова в пределах одного блока, что может быть использовано для обмена данными между потоками в пределах одного блока. Для размещения данных в разделяемой памяти используется спецификатор `__shared__`.

Алгоритм работы с разделяемой памятью:

• Обработка одной порции данных должна происходить одним блоком потоков.

• Загрузить данные из глобальной памяти в разделяемую, используя все параллельные потоки.

• После синхронизации произвести требуемые вычисления над данными в разделяемом кэше.

• Скопировать результат обратно в глобальную память.

Разделяемая память:

• Размещена непосредственно в мультипроцессоре(SM)
• Видна всем нитям в пределах блока
• Латентность минимальна(1–16 тактов)
• Выделяется непосредственно на устройстве – не видна хосту
• Объем памяти на SM аппаратно ограничен

• Память делится между всеми одновременно выполняющимися на SM блоками.

Параметры функций, выполняемых на устройстве, передаются через shared-память.

Константная память

Константная память (constant memory) является достаточно быстрой из доступных GPU. Отличительной особенностью константной памяти является возможность записи данных с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти, что и обуславливает её название.

Для размещения данных в константной памяти предусмотрен спецификатор __constant__. Если необходимо использовать массив в константной памяти, то его размер необходимо указать заранее, так как динамическое выделение в отличие от глобальной памяти в константной не поддерживается.

Для записи с хоста в константную память используется функция cudaMemcpyToSymbol, и для копирования с device'а на хост cudaMemcpyFromSymbol, как видно этот подход несколько отличается от подхода при работе с глобальной памятью.

Constant memory – физически не отделена от глобальной памяти.

Расположение: DRAM

Кэшируемость: yes, 8 KB at MP

Уровень выделения: on chip L1 cache

Доступ: GPU – R/O, CPU – R/W

Скорость работы: высокая (cache) / низкая (400–600 тактов)

Уровень доступа: per-grid, CPU

Время жизни: выделяется CPU

Кэш существует в единственном экземпляре для одного мультипроцессора, а значит, общий для всех задач внутри блока.

Текстурная память

Предназначена главным образом для работы с текстурами. Она оптимизирована под выборку 2D данных.

Примеры использования различных типов памяти

Транспонирование матрицы. Параметры:

`inputMatrix` – указатель на исходную матрицу

`outputMatrix` – указатель на матрицу результат

`width` – ширина исходной матрицы (она же высота матрицы-результата)

`height` – высота исходной матрицы (она же ширина матрицы-результата)

Пример1. Реализация на CPU (последовательный вариант)

```
__host__ void transposeMatrixCPU(float* inputMatrix,
                    float* outputMatrix, int width, int height)
{
     for (int y = 0; y < height; y++)
     {
          for (int x = 0; x < width; x++)
          {
               outputMatrix[x * height + y] = inputMatrix[y *
width + x];
          }
     }
}
```

Пример 2. Использование только глобальной памяти.

```
__global__ void transposeMatrixSlow(float* inputMatrix, float*
outputMatrix, int width, int height)
{
  int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
  if ((xIndex < width) && (yIndex < height))
  {
    int inputIdx = xIndex + width * yIndex;
    int outputIdx = yIndex + height * xIndex;
    outputMatrix[outputIdx] = inputMatrix[inputIdx];
  }
}
```

Пример 3. Использование константной памяти.

```
#define N 100
__constant__  float  devInputMatrix[N];
__global__ void transposeMatrixSlow(float* inputMatrix, float*
outputMatrix, int width, int height)
{   int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
  if ((xIndex < width) && (yIndex < height))
  {     int inputIdx = xIndex + width * yIndex;
        int outputIdx = yIndex + height * xIndex;
        outputMatrix[outputIdx] = inputMatrix[inputIdx];  }  }
```

Пример 4. Использование разделяемой памяти.

```
__global__ void transposeMatrixFast(float* inputMatrix, float*
outputMatrix, int width, int height)
{  __shared__ float temp[BLOCK_DIM][BLOCK_DIM];
  int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
  int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
  if ((xIndex < width) && (yIndex < height))
  {
    int idx = yIndex * width + xIndex;
   temp[threadIdx.y][threadIdx.x] = inputMatrix[idx];
  }
  __syncthreads();
  xIndex = blockIdx.y * blockDim.y + threadIdx.x;
  yIndex = blockIdx.x * blockDim.x + threadIdx.y;
  if ((xIndex < height) && (yIndex < width))
  {
     int idx = yIndex * height + xIndex;
outputMatrix[idx] = temp[threadIdx.x][threadIdx.y];
  }
__syncthreads();
}
```

Библиографический список

1. Гергель В. П. Современные языки и технологии параллельного программирования: учебник / В. П. Гергель. – М. : Издательство Московского университета, 2012. – 408 с. (Серия «Суперкомпьютерное образование»)

2. Уильямс Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ / Э. Уильямс. – М. : ДМК Пресс, 2012. – 672 с.

3. Антоноа А. С. Технологии параллельного программирования MPI и OpenMP: учеб. пособие / А. С. Антонов. – М. : Издательство Московского университета, 2012. – 344 с. (Серия «Суперкомпьютерное образование»)

4. Снытников А. В. Математическое моделирование и программная модель CUDA: учеб. пособие / А. В. Снытников, А. С. Колганов, Н. Н. Попова. – М. : МАКС Пресс, 2018. – 176 с. (Серия «Суперкомпьютерное образование»)

5. Лин К. Принципы параллельного программирования: учеб. пособие / К. Лин, Л. Снайдер. – М. : Издательство Московского университета, 2013. – 408 с. (Серия «Суперкомпьютерное образование»)

6. Боресков А. В. Основы работы с технологией CUDA / А. В. Боресков, А. А. Харламов. – М. : ДМК Пресс, 2010. – 232 с.

7. Сандерс Дж. Технология CUDA в примерах: Введение в программирование графических процессоров / Дж. Сандерс, Э. Кэндрот. – М. : ДМК Пресс, 2013. – 232 с.

8. Cheng J. Professional CUDA C Programming / J. Cheng, M. Grossman, T. McKercher. – Indianapolis: John Wiley & Sons, Inc., 2014. – 528 p.

9. Антонов А. С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие / Антонов А. С – М. : Изд-во МГУ, 2009. – 77 с.

10. Левин М. П. Параллельное программирование с использованием OpenMP: учебное пособие / М. П. Левин . – М. : Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2008. – 118 с.

Содержание

Учебное издание

Абрамов Геннадий Владимирович,
Каплиева Наталья Алексеевна

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум

В авторской редакции

Подписано в печать 08.11.2021. Формат 60×84/16.
Уч.-изд. л. _,_. Усл. печ. л. 4,4. Тираж 30 экз. Заказ ___

Издательский дом ВГУ
394018 Воронеж, пл. Ленина, 10

Отпечатано с готового оригинал-макета
в типографии Издательского дома ВГУ
394018 Воронеж, ул. Пушкинская, 3