**Федеральное государственное автономное образовательное учреждение высшего образования**
«Нижегородский государственный университет им. Н.И. Лобачевского»
Программа повышение конкурентоспособности ННГУ им. Н.И. Лобачевского
Стратегическая инициатива 7 «Достижение лидирующих позиций в области суперкомпьютерных технологий и высокопроизводительных вычислений»

# Нейросетевые математические модели функций нервной системы (Блок лекций по параллельным вычислениям)

## Лекция №1. Общие сведения из области параллельных вычислений и программирования

На протяжении долгого времени существовало два традиционных метода научных исследований - теория и эксперимент. Например, теоретическая физика занимается построением моделей, объясняющих физические явления, а экспериментальная физика - их изучением, чтобы подтвердить или опровергнуть гипотезы. Однако с развитием компьютерных технологий появился третий тип исследований - численное моделирование, которое позволяет имитировать сложнейшие явления, такие как эволюция звезд или изменение климата, с помощью компьютеров. Этот научный метод начал свое развитие в 60-х, когда в распоряжении инженеров и ученых появились быстродействующие вычислительные машины. С тех пор достижения в области компьютерных технологий активно используются при решении научных задач.

Так произошло и с параллельным программированием, которое возникло в сфере операционных систем. Причиной стало изобретение аппаратных модулей, названных каналами, или контроллерами устройств. Они работают независимо от управляющего процессора и позволяют выполнять операции ввода-вывода параллельно с инструкциями центрального процессора. Канал взаимодействует с процессором с помощью прерывания - аппаратного сигнала, который говорит: "Останови свою работу и начни выполнять другую последовательность инструкций". Вскоре после изобретения каналов началась разработка многопроцессорных машин, хотя в течение двух десятилетий они были слишком дороги для широкого использования. Однако сейчас практически все машины являются многопроцессорными, а самые большие имеют сотни процессоров и часто называются суперкомпьютерами. Такие машины позволяют разным прикладным программам выполняться одновременно на разных процессорах. Они также ускоряют выполнение приложения, если оно написано (или переписано) для многопроцессорной машины. Но как синхронизиро-

вать работу параллельных процессов? Как использовать многопроцессорные системы для ускорения выполнения программ?

Цель технологий параллелизма — обеспечить условия, позволяющие компьютерным программам делать больший объем работы за тот же интервал времени. Поэтому проектирование программ должно ориентироваться не на выполнение одной задачи в некоторый промежуток времени, а на одновременное выполнение нескольких задач, на которые предварительно должна быть разбита программа. Возможны ситуации, когда целью является не выполнение большего объема работы в течение того же интервала времени, а упрощение решения с точки зрения программирования. Иногда имеет смысл думать о решении проблемы как о множестве параллельно выполняемых задач. Например (если взять для сравнения вполне житейскую ситуацию), проблему снижения веса лучше всего представить в виде двух параллельно выполняемых задач: диета и физическая нагрузка. Иначе говоря, для решения этой проблемы предполагается применение строгой диеты и физических упражнений в один и тот же интервал времени (необязательно точно в одни и те же моменты времени). Обычно не слишком полезно (или эффективно) выполнять одну подзадачу в один период времени, а другую — совершенно в другой. Именно параллельность обоих процессов дает естественную форму искомого решения проблемы. Иногда к параллельности прибегают, чтобы увеличить быстродействие программы или приблизить момент ее завершения. В других случаях параллельность используется для увеличения продуктивности программы (объема выполняемой ею работы) за тот же период времени при вторичности скорости ее работы.

При написании параллельной программы необходимо решать, сколько процессов и какого типа нужно использовать, и как они должны взаимодействовать. Эти решения зависят как от конкретного приложения, так и от аппаратного обеспечения, на котором будет выполняться программа. В любом случае ключом к созданию корректной программы является правильная синхронизация взаимодействия процессов.

В базовой последовательной модели программирования инструкции компьютерной программы выполняются поочередно. Разработчик программы разбивает основную задачу на коллекцию подзадач. Все задачи выполняются по порядку, и каждая из них должна ожидать своей очереди. Другими словами, каждая задача, прежде чем приступить к своей работе, должна ожидать до тех пор, пока не получит результатов выполнения предыдущей. В последовательной модели зачастую устанавливается последовательная зависимость задач. В мире параллельного программирования все обстоит по-другому. Здесь сразу несколько инструкций могут выполняться в один и тот же момент времени. Одна инструкция разбивается на несколько мелких частей, которые будут выполняться одновремен-

но. Программа разбивается на множество параллельных задач. Программа может состоять из сотен или даже тысяч выполняющихся одновременно подпрограмм. Несколько задач могут одновременно начать выполнение на любом процессоре без какой бы то ни было гарантии того, что задачи закреплены за определенными процессорами, или такая-то задача завершится первой, или все они завершатся в таком-то порядке. Помимо параллельного выполнения задач, здесь возможно параллельное выполнение частей (подзадач) одной задачи. В некоторых конфигурациях не исключена возможность выполнения подзадач на различных процессорах или даже различных компьютерах.

В настоящее время парадигма параллельного программирования применяется во многих областях, среди которых системы реального времени, управляющие работой электростанций, различной аппаратурой, файловые серверы, системы баз данных, обработка изображений, а также научные вычисления, которые будут основной темой дальнейшего рассмотрения. Поэтому, прежде чем перейти к конкретным примерам решений задач с использованием параллельных алгоритмов, рассмотрим общие понятия.

## Основные подходы: параллельное и распределенное программирование

Параллельное и распределенное программирование — это два базовых подхода к достижению параллельного выполнения составляющих программного обеспечения (ПО). Они представляют собой две различные парадигмы программирования, которые иногда пересекаются. ***Методы параллельного программирования*** позволяют распределить работу программы между двумя (или больше) процессорами в рамках одного физического или одного виртуального компьютера. ***Методы распределенного программирования*** позволяют распределить работу программы между двумя (или больше) процессами, причем процессы могут существовать на одном и том же компьютере или на разных. Другими словами, части распределенной программы зачастую выполняются на разных компьютерах, связываемых по сети, или, по крайней мере, в различных процессах. Программа, содержащая параллелизм, выполняется на одном и том же физическом или виртуальном компьютере. Такую программу можно разбить на ***процессы*** (process) или ***потоки*** (thread).

Многопоточность ограничивается параллелизмом. Формально параллельные программы иногда бывают распределенными, например, при PVM-программировании ( **P**arallel **V**irtual **M**achine — параллельная виртуальная машина). Распределенное программирование иногда используется для реализации параллелизма, как в случае с MPI-программированием (Message Passing Interface — интерфейс для передачи сообщений). Однако не все распределенные программы включают параллелизм. Части распределенной

программы могут выполняться по различным запросам и в различные периоды времени. Например, программу календаря можно разделить на две составляющие. Одна часть должна обеспечивать пользователя информацией, присущей календарю, и способом записи данных о важных для него встречах, а другая часть должна предоставлять пользователю набор сигналов для разных типов встреч. Пользователь составляет расписание встреч, используя одну часть ПО, в то время как другая его часть выполняется независимо от первой. Набор сигналов и компонентов расписания вместе образуют единое приложение, которое разделено на две части, выполняемые по отдельности. При чистом параллелизме одновременно выполняемые части являются компонентами одной и той же программы. Части распределенных приложений обычно реализуются как отдельные программы. Типичная архитектура построения параллельной и распределенной программ показана на рис. 1.

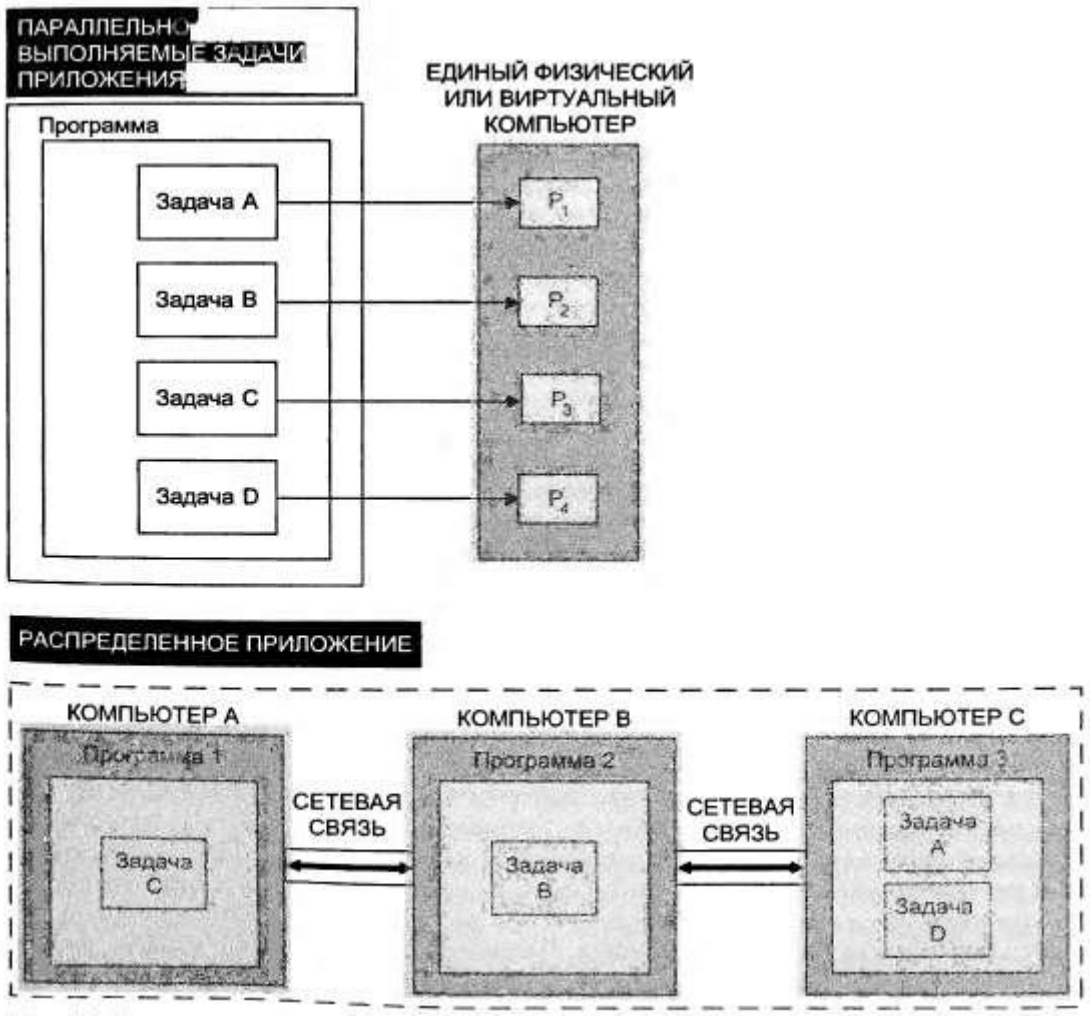


Рисунок 1. Типичная архитектура параллельной и распределенной программ

Параллельное приложение, показанное на рисунке, состоит из одной программы, разделенной на четыре задачи. Каждая задача выполняется на отдельном процессоре, следо-

вательно, все они могут выполняться одновременно. Эти задачи можно реализовать как распределенное приложение, состоящее из трех отдельных программ, каждая из которых выполняется на отдельном компьютере. При этом программа 3 состоит из двух отдельных частей (задачи A и задачи D), выполняющихся на одном компьютере. Несмотря на это, задачи A и D являются распределенными, поскольку они реализованы как два отдельных процесса. Задачи параллельной программы более тесно связаны, чем задачи распределенного приложения. В общем случае процессоры, связанные с распределенными программами, находятся на различных компьютерах, в то время как процессоры, связанные с программами, реализующими параллелизм, находятся на одном и том же компьютере. Конечно же, существуют гибридные приложения, которые являются и параллельными, и распределенными одновременно. Именно такие гибридные объединения становятся нормой.

## Базовые уровни программного параллелизма

Итак, параллелизм можно обеспечить на уровне:

- инструкций;
- подпрограмм (функций или процедур);
- объектов;
- приложений.

*Параллелизм на уровне **инструкций*** возникает, если несколько частей одной инструкции могут выполняться одновременно. Например, в выражении (A+B)•(C-D) компонент (A+B) можно вычислить одновременно с компонентом (C-D). Этот вид параллелизма зачастую поддерживается директивами компилятора и не попадает под управление программиста.

*Параллелизм на уровне **подпрограмм.*** Структуру программы можно представить в виде ряда функций, т.е. сумма работ, из которых состоит программное решение, разбивается на некоторое количество функций. Если эти функции распределить по потокам, то каждую функцию в этом случае можно выполнить на отдельном процессоре, и, если в вашем распоряжении будет достаточно процессоров, то все функции смогут выполняться одновременно.

*Параллелизм на уровне **объектов***. Структуру программного решения можно распределить между объектами. Каждый объект можно назначить отдельному потоку или процессу. Объекты, реализованные в различных потоках или процессах, могут выполнять свои методы параллельно, таким образом, все объекты можно назначить различным ком-

пьютерам одной сети или различным компьютерам различных сетей. Для этого используется стандарт CORBA (Common Object Request Broker Architecture — технологический стандарт написания распределенных объектных приложений).

*Параллелизм на уровне* ***приложений***. Несколько приложений могут сообща решать некоторую проблему. Несмотря на то, что какое-то приложение первоначально предназначалось для выполнения отдельной задачи, принципы многократного использования кода позволяют приложениям сотрудничать. В таких случаях два отдельных приложения эффективно работают вместе подобно единому распределенному приложению. Например, буфер обмена (Clipboard) не предназначался для работы ни с каким конкретным приложением, но его успешно использует множество приложений рабочего стола.

## Классификация параллельных вычислительных систем

### Классификация Флина (по типу потока команд и потока данных)

Упрощенная классификации схем функционирования параллельных компьютеров была предложена М. Флинном (M.J. Flynn). Согласно этой классификации различались две схемы: SIMD (Single-Instruction, Multiple-Data — архитектура с одним потоком команд и многими потоками данных) и MIMD (Multiple-Instruction, Multiple-Data — архитектура со множеством потоков команд и множеством потоков данных). Несколько позже эти схемы были расширены до SPMD (Single-Program, Multiple-Data — одна программа, несколько потоков данных) и MPMD (Multiple-Programs, Multiple-Data — множество программ, множество потоков данных) соответственно. Схема SPMD (SIMD) позволяет нескольким процессорам выполнять одну и ту же инструкцию или программу при условии, что каждый процессор получает доступ к различным данным. Схема MPMD (MIMD) позволяет работать нескольким процессорам, причем все они выполняют различные программы или инструкции и пользуются собственными данными. Таким образом, в одной схеме все процессоры выполняют одну и ту же программу или инструкцию, а в другой все процессоры выполняют различные программы или инструкции. Конечно же, возможны гибриды этих моделей, в которых процессоры могут быть разделены на группы, из которых одни образуют SPMD-модель, а другие — MPMD-модель. При использовании схемы SPMD все процессоры просто выполняют одни и те же операции, но с различными данными. Например, мы можем разбить одну задачу на группы и назначить для каждой группы отдельный процессор. В этом случае каждый процессор при решении задачи будет применять одинаковые правила, обрабатывая при этом различные части этой задачи. Когда все процессоры справятся со своими участками работы, мы получим решение всей задачи. Если же при-

меняется схема MPMD, все процессоры выполняют различные виды работы, и, хотя при этом все они вместе пытаются решить одну проблему, каждому из них выделяется свой аспект этой проблемы. Например, разделим задачу по обеспечению безопасности Web-сервера по схеме MPMD. В этом случае каждому процессору ставится своя подзадача. Предположим, один процессор будет отслеживать работу портов, другой — курировать процесс регистрации пользователей, а третий — анализировать содержимое пакетов и т.д. Таким образом, каждый процессор работает с нужными ему данными. И хотя различные процессоры выполняют разные виды работы, используя различные данные, все они вместе работают в одном направлении — обеспечивают безопасность Web-сервера. Принципы параллельного программирования, рассматриваемые в этой книге, нетрудно описать, используя модели PRAM, SPMD (SIMD) и MPMD (MIMD). И в самом деле, эти схемы и модели успешно используются для реализации практических мелко- и среднемасштабных приложений и вполне могут вас устраивать до тех пор, пока вы не подготовитесь к параллельному программированию более высокой степени организации.

**Классификация по типу строения оперативной памяти**

В вычислительных системах с *общей (разделяемой) памятью* (Common Memory Systems или Shared Memory Systems) (рис.2) значение, записанное в память одним из процессоров, напрямую доступно для другого процессора. Общая память обычно имеет высокую пропускную способность памяти (bandwidth) и низкую латентность памяти (latency) при передаче информации между процессорами, но при условии, что не происходит одновременного обращения нескольких процессоров к одному и тому же элементу памяти. К общей памяти доступ разными процессорами системы осуществляется, как правило, за одинаковое время. Поэтому такая память называется еще UMA-памятью (Unified Memory Access) — памятью с одинаковым временем доступа. Система с такой памятью носит название вычислительной системы с одинаковым временем доступа к памяти. Системы с общей памятью называются также сильно связанными вычислительными системами.

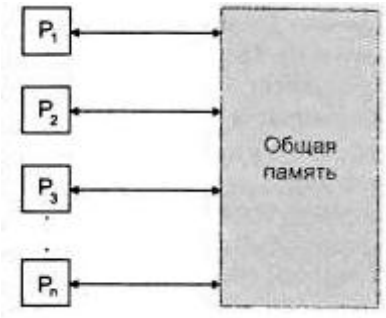

Рисунок 2

В вычислительных системах с *распределенной памятью* (Distributed Memory Systems) каждый процессор имеет свою локальную память с локальным адресным пространством. Для систем с распределенной памятью характерно наличие большого числа быстрых каналов, которые связывают отдельные части этой памяти с отдельными процессорами. Обмен информацией между частями распределенной памяти осуществляется обычно относи-

тельно медленно. Системы с распределенной памятью называются также слабосвязанными вычислительными системами.

Вычислительные системы с *гибридной памятью* - (Non-Uniform Memory Access Systems) имеют память, которая физически распределена по различным частям системы, но логически разделяема (образует единое адресное пространство). Такая память называется еще логически общей (разделяемой) памятью (logically shared memory). В отличие от UMA-систем, в NUMA-системах время доступа к различным частям оперативной памяти различно.

## Простейшие модели параллельного программирования

Возможно, самой простой и распространенной моделью *распределенной* обработки данных является модель типа «клиент/сервер». В этой модели программа разбивается на две части: одна часть называется ***сервером,*** а другая — ***клиентом.*** Сервер имеет прямой доступ к некоторым аппаратным и программным ресурсам, которые желает использовать клиент. В большинстве случаев сервер и клиент располагаются на разных компьютерах. Обычно между клиентом и сервером существует отношение типа «множество-к-одному», т.е., как правило, один сервер отвечает на запросы многих клиентов. Сервер часто обеспечивает опосредованный доступ к огромной базе данных, дорогостоящему оборудованию или некоторой коллекции приложений. Клиент может запросить интересующие его данные, сделать запрос на выполнение вычислительной процедуры или обработку другого типа. В качестве примера приложения типа «клиент/сервер» приведем механизм поиска (search engine). Механизмы (или машины) поиска используются для поиска заданной информации в Internet или корпоративной Intranet. Клиент служит для получения ключевого слова или фразы, которая интересует пользователя. Часть ПО клиента затем передает сформированный запрос той части ПО сервера, которая обладает средствами поиска информации по заданному пользователем ключевому слову или фразе. Сервер либо имеет прямой доступ к информации, либо связан с другими серверами, которые имеют его. В идеальном случае сервер находит запрошенное пользователем ключевое слово или фразу и возвращает найденную информацию клиенту. Несмотря на то, что клиент и сервер представляют собой отдельные программы, выполняющиеся на разных компьютерах, вместе они составляют единое приложение. Разделение ПО на части клиента и сервера и есть основной метод распределенного программирования. Модель типа «клиент/сервер» также имеет другие формы, которые зависят от конкретной среды. Например, термин «изготовитель-потребитель» (producer-consumer) можно считать близким родственником термина

«клиент/сервер». Обычно клиент-серверными приложениями называют большие программы, а термин «изготовитель-потребитель» относят к программам меньшего объема. Если программы имеют уровень операционной системы или ниже, к ним применяют термин «изготовитель-потребитель», если выше — то термин «клиент/сервер» (конечно же, исключения есть из всякого правила).

*Параллельное* программирование представляет дополнительные источники сложности - необходимо явно управлять работой тысяч процессоров, координировать миллионы межпроцессорных взаимодействий. Для того решить задачу на параллельном компьютере, необходимо распределить вычисления между процессорами системы, так чтобы каждый процессор был занят решением части задачи. Кроме того, желательно, чтобы как можно меньший объем данных пересылался между процессорами, поскольку коммуникации значительно больше медленные операции, чем вычисления. Часто, возникают конфликты между степенью распараллеливания и объемом коммуникаций, то есть чем между большим числом процессоров распределена задача, тем больший объем данных необходимо пересылать между ними. Среда параллельного программирования должна обеспечивать адекватное управление распределением и коммуникациями данных.

Из-за сложности параллельных компьютеров и их существенного отличия от традиционных однопроцессорных компьютеров нельзя просто воспользоваться традиционными языками программирования и ожидать получения хорошей производительности. Рассмотрим основные модели *параллельного программирования:*

***Процесс/канал*** (*Process/Channel*). В этой модели программы состоят из одного или более процессов, распределенных по процессорам. Процессы выполняются одновременно, их число может измениться в течение времени выполнения программы. Процессы обмениваются данными через каналы, которые представляют собой однонаправленные коммуникационные линии, соединяющие только два процесса. Каналы можно создавать и удалять.

***Обмен сообщениями*** (*Message Passing*). В этой модели программы, возможно различные, написанные на традиционном последовательном языке, исполняются процессорами компьютера. Каждая программа имеют доступ к памяти исполняющего ее процессора. Программы обмениваются между собой данными, используя подпрограммы приема/передачи данных некоторой коммуникационной системы. Программы, использующие обмен сообщениями, могут выполняться только на MIMD компьютерах.

***Параллелизм данных*** (*Data Parallel*). В этой модели единственная программа задает распределение данных между всеми процессорами компьютера и операции над ними. Распределяемыми данными обычно являются массивы. Как правило, языки программиро-

вания, поддерживающие данную модель, допускают операции над массивами, позволяют использовать в выражениях целые массивы, вырезки из массивов. Распараллеливание операций над массивами, циклов обработки массивов позволяет увеличить производительность программы. Компилятор отвечает за генерацию кода, осуществляющего распределение элементов массивов и вычислений между процессорами. Каждый процессор отвечает за то подмножество элементов массива, которое расположено в его локальной памяти. Программы с параллелизмом данных могут быть оттранслированы и исполнены как на MIMD, так и на SIMD компьютерах.

***Общей памяти*** (*Shared Memory*)**.** В этой модели все процессы совместно используют общее адресное пространство. Процессы асинхронно обращаются к общей памяти как с запросами на чтение, так и с запросами на запись, что создает проблемы при выборе момента, когда можно будет поместить данные в память, когда можно будет удалить их. Для управления доступом к общей памяти используются стандартные механизмы синхронизации - семафоры и блокировки процессов.

## Способы взаимодействия процессов.

В соответствии с возможностями параллельного компьютера процессы взаимодействуют между собой обычно одним из следующих способов:

***Обмен сообщениями (****message passing****)***. Посылающий процесс формирует сообщение с заголовком, в котором указывает, какой процессор должен принять сообщение, и передает сообщение в сеть, соединяющую процессоры. Если, как только сообщение было передано в сеть, посылающий процесс продолжает работу, то такой вид отправки сообщения, называется неблокирующим (*non-blocking send*). Если же посылающий процесс ждет, пока принимающий процесс не примет сообщение, то такой вид отправки сообщения, называется блокирующим (*blocking send*). Принимающий процесс должен знать, что ему необходимы данные, и должен указать, что готов получить сообщение, выполнив соответствующую команду приема сообщения. Если ожидаемое сообщение еще не поступило, то принимающий процесс приостанавливается до тех пор, пока сообщение не поступит.

***Обмен через общую память*** *(Transfers through shared memory)***.** В архитектурах с общедоступной памятью процессы связываются между собой через общую память - посылающий процесс помещает данные в известные ячейки памяти, из которых принимающий процесс может считывать их. При таком обмене сложность представляет процесс обнаружения того, когда безопасно помещать данные, а когда удалять их. Чаще всего для этого используются стандартные методы операционной системы, такие как *семафоры* или *блокировки* процессов. Однако это дорого и сильно усложняет программирование. Некоторые

архитектуры предоставляют биты *занято/свободно*, связанные с каждым словом общей памяти, что обеспечивает легким и высокоэффективный способ синхронизации отправителей и приемников.

***Прямой доступ к удаленной памяти*** *(Direct remote-memory access).* В первых архитектурах с распределенной памятью работа процессоров прерывалась каждый раз, когда поступал какой-нибудь запрос от сети, соединяющей процессоры. В результате процессор плохо использовался. Затем в таких архитектурах в каждом процессорном элемент стали использовать пары процессоров - один процессор (вычислительный), исполняет программу, а другой (процессор обработки сообщений) обслуживает сообщения, поступающие из сети или в сеть. Такая организация обмена сообщениями позволяет рассматривать обмен сообщениями как прямой доступ к удаленной памяти, к памяти других процессоров. Эта гибридная форма связи, применяется в архитектурах с распределенной памятью, обладает многими свойствами архитектурах с общей памятью.

Рассмотренные механизмы связи необязательно используются только непосредственно на соответствующих архитектурах. Так легко промоделировать обмен сообщениями, используя общую память, с другой стороны можно смоделировать общую память, используя обмен сообщениями. Последний подход известен как *виртуальная общая память*.

## Средства синхронизации и межпроцессорного взаимодействия

В параллельных программах используется два основных типа синхронизации: взаимное исключение и условная синхронизация. В частности синхронизация может быть осуществлена с помощью семафоров, событий, мьютексов, критических секций, барьеров и т.д.

*Семафор* — это переменная специального вида, которая может быть доступна только для выполнения узкого диапазона операций. Семафор используется для синхронизации доступа процессов и потоков к разделяемой модифицируемой памяти или для управления доступом к устройствам или другим ресурсам. Семафор можно рассматривать как ключ к ресурсам. Этим ключом может владеть в любой момент времени только один процесс или поток. Какал бы задача ни владела этим ключом, он надежно запирает (блокирует) нужные ей ресурсы для ее монопольного использования. Блокирование ресурсов заставляет другие задачи, которые желают воспользоваться этими ресурсами, ожидать до тех пор, пока они не будут разблокированы и снова станут доступными. После разблокирования ресурсов следующая задача, ожидающая семафор, получает его и доступ к ресурсам. Какая задача будет следующей, определяется стратегией планирования, действующей для данного потока или процесса.

*Мью́текс* (англ. mutex, от mutual exclusion — «взаимное исключение») — это один из вариантов семафорных механизмов для организации взаимного исключения. Они реализованы во многих ОС, их основное назначение — организация взаимного исключения для потоков из одного и того же или из разных процессов. Это простейшие двоичные семафоры, которые могут находиться в одном из двух состояний — отмеченном или неотмеченном (открыт и закрыт соответственно). Когда какой-либо поток, принадлежащий любому процессу, становится владельцем объекта mutex, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным. Задача мьютекса — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом. Если другому потоку будет нужен доступ к переменной, защищённой мьютексом, то этот поток засыпает до тех пор, пока мьютекс не будет освобождён. Цель использования мьютексов — защита данных от повреждения в результате асинхронных изменений (состояние гонки), однако могут порождаться другие проблемы — такие, как взаимная блокировка (клинч). Мьютекс отличается от семафора общего вида тем, что только владеющий им поток может его освободить, т.е. перевести в отмеченное состояние.

*Критическая секция* (англ. critical section) — участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком исполнения. При нахождении в критической секции двух (или более) процессов возникает состояние «гонки» («состязания»). Для избежания данной ситуации необходимо выполнение четырех условий:

- Два процесса не должны одновременно находиться в критических областях.
- В программе не должно быть предположений о скорости или количестве процессоров.
- Процесс, находящийся вне критической области, не может блокировать другие процессы.
- Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

Критическая секция— объект синхронизации потоков, позволяющий предотвратить одновременное выполнение некоторого набора операций (обычно связанных с доступом к данным) несколькими потоками. Критическая секция выполняет те же задачи, что и мьютекс. Между мьютексом и критической секцией есть терминологические различия, так процедура, аналогичная захвату мьютекса, называется входом в критическую секцию

(англ. enter), снятию блокировки мьютекса — выходом из критической секции (англ. leave). Процедура входа и выхода из критических секций обычно занимает меньшее время, нежели аналогичные операции мьютекса, что связано с отсутствием необходимости обращаться к ядру ОС. В операционных системах семейства Microsoft Windows разница между мьютексом и критической секцией в том, что мьютекс является объектом ядра и может быть использован несколькими процессами одновременно, критическая секция же принадлежит процессу и служит для синхронизации только его потоков.Критические секции Windows имеют оптимизацию, заключающуюся в использовании атомарно изменяемой переменной наряду с объектом «событие синхронизации» ядра. Захват критической секции означает атомарное увеличение переменной на 1. Переход к ожиданию на событии ядра осуществляется только в случае, если значение переменной до захвата было уже больше 1, то есть происходит реальное «соревнование» двух или более потоков за ресурс. Таким образом, при отсутствии соревнования захват/освобождение критической секции обходятся без обращений к ядру. Кроме того, захват уже занятой критической секции до обращения к ядру какое-то малое время ждёт в цикле (кол-во итераций цикла (англ. spin count) задаётся функциями InitializeCriticalSectionAndSpinCount() или SetCriticalSectionSpinCount()) опроса переменной числа текущих пользователей, и, если эта переменная становится равной 0, то захват происходит без обращений к ядру. Сходный объект в ядре Windows называется FAST_MUTEX (ExAcquire/ReleaseFastMutex). Он отличается от критической секции отсутствием поддержки рекурсивного повторного захвата тем же потоком. Аналогичный объект в Linux называется фьютекс.

*Барьер* - это точка синхронизации, которую должны достигнуть все процессы до выполнения дальнейших действий, т.е. в конце итерации более быстрый процесс должен ожидать более медленные для выполнения следующей итерации. Процессы создаются один раз в начале вычислений, а потом синхронизируются в конце каждой итерации.

```
process Worker[i = 1 to n] {
while (true) {
код решения задачи i;
ожидание завершения всех n задач;
    }
}
```

Точка задержки в конце каждой итерации представляет собой барьер, которого для продолжения работы должны достигнуть все процессы, поэтому этот механизм называется барьерной синхронизацией. Барьеры могут понадобиться в конце циклов, как в данном примере, или на промежуточных стадиях, как будет показано ниже. Для реализации барь-

ерной синхронизации можно использовать счетчик, который будет увеличивать значение на 1 при завершении каждого из n процессов; когда значение счетчика достигнет n, все процессы приступают к следующей итерации. Существуют реализации барьерной синхронизации, основанные на использовании флагов и управляющего процесса, а также симметричные барьеры (отсутствует управляющий процесс).

## Обзор программных средств высокопроизводительных вычислений

Область параллельного программирования находится в постоянном и активном развитии, поэтому в Таблице 1 представлен неполный список параллельных языков программирования и систем разработки параллельных программ, применяемых в различных областях.

*Таблица 1*

| Для систем с разделяемой памятью | Для систем с распределенной памятью | Параллельные объектно-ориентированные | Параллельные декларативные |
|---|---|---|---|
| OpenMP | PVM | HPC++ | Parlog |
| Linda | MPI | MPL | Multilisp |
| Orca | HPF | CA | Sisal |
| Java | Cilk | Distributed Java | Concurrent Prolog |
| Pthreads | C* | Charm++ | GHC |
| Opus | ZPL | Concurrent Aggregates | Strand |
| SDL | Occam | Argus | Tempo |
| Ease | Concurrent C | Presto | |
| SHMEM | Ada | Nexus | |
| | FORTRAN M | uC++ | |
| | CSP | sC++ | |
| | NESL | pC++ | |
| | MpC | | |

Рассмотрим подробнее OpenMP и MPI.

OpenMP — это API-интерфейс, который является отраслевым стандартом для создания параллельных приложений для компьютеров с совместным использованием памяти. Главная задача OpenMP - облегчить написание программ, ориентированных на циклы. Такие программы часто создаются для высокопроизводительных вычислений. Кроме того, в

OpenMP были включены компоненты для поддержки таких параллельных алгоритмов как SPMD, "главный и рабочий процесс", конвейерный и т.д. OpenMP стал очень успешным языком параллельного программирования. Он имеется на каждом компьютере с совместным использованием памяти, выходящем на рынок. Кроме того, недавно в Intel был создан вариант OpenMP для поддержки кластеров. OpenMP поддерживает такой стиль программирования, при котором параллелизм добавляется постепенно, пока имеющаяся последовательная программа не превратится в параллельную. Впрочем, это преимущество является и самым слабым местом OpenMP. Если параллелизм добавляется постепенно, программист может не выполнить широкомасштабную перестройку программы, которая часто необходима для достижения максимальной производительности. OpenMP основывается на модели программирования "разветвление-объединение" (fork-join). Работа программы OpenMP начинается с одного потока. Когда программисту требуется добавить в программу параллелизм, выполняется разветвление на несколько потоков, чтобы создать группу потоков. Эти потоки выполняются параллельно в рамках фрагмента кода, который называется параллельным участком. В конце параллельного участка все потоки заканчивают свою работу и снова объединяются вместе. После этого исходный (или "главный") поток продолжает выполняться до тех пор, пока не начнется следующий параллельный участок (или не наступит конец программы).

MPI (Message Passing Interface) — стандарт систем передачи сообщений между параллельно исполняемыми процессами. В большинстве MPI-программ используется шаблон "Одна программа, разные данные" (Single Program Multiple Data, или SPMD). В его основе лежит простой принцип: каждый элемент обработки (processing element, PE) выполняет одну и ту же программу. Каждому элементу обработки присваивается уникальный целочисленный идентификатор, который определяет его ранг в наборе элементов обработки. Программа использует этот ранг, чтобы распределить работу и определить, какой элемент PE какую работу выполняет. Иными словами, программа всего одна, но благодаря выбору, сделанному в соответствии с идентификатором, данные для каждого элемента обработки могут быть разными. Это и есть шаблон "Одна программа, разные данные". Ключевое понятие MPI - коммуникатор. При создании набора процессов они образуют группу, которая может совместно использовать среду для связи. Группа процессов в сочетании со средой связи образует уникальный коммуникатор. Преимущества такой концепции становятся очевидными, если рассмотреть использование библиотек в программе. Если программист будет невнимателен, сообщения, созданные разработчиком библиотеки, могут конфликтовать с сообщениями, которые используются в программе, вызывающей эту библиотеку. Но при наличии коммуникаторов разработчик библиотеки может создать собственную

среду связи и при этом гарантировать, что пока речь идет о сообщениях, передаваемых в системе, все, что происходит в библиотеке, не выйдет за ее пределы. При запуске MPI-программы создается коммуникатор по умолчанию, MPI_COMM_WORLD, который передается каждой MPI-подпрограмме в качестве первого аргумента. Остальные аргументы определяют источник сообщения и буферы для хранения сообщений. MPI-подпрограммы возвращают целочисленное значение как параметр ошибки, что позволяет узнать о любых проблемах, имевших место при выполнении подпрограммы. Большая часть программы представляет собой обычный последовательный код, язык которого программист выбирает сам. Как упоминалось выше, хотя все процессы выполняют одинаковый код, поведение программы различается в зависимости от ранга процесса. В тех точках, где требуется связь или иное взаимодействие между процессами, вставляются MPI-подпрограммы. В первую версию MPI входило более 120 подпрограмм, а в последней версии (MPI 2.0) их количество еще возросло. Впрочем, в большинстве программ используется очень небольшой набор MPI-функций. Ниже представлена одна подпрограмма для выполнения редукции и возврата конечного результата одному из процессов в группе.

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_OP op, int root, MPI_COMM comm.)
```

Эта функция принимает "count" значений типа "datatype" в буфере "sendbuf", собирает результаты от каждого процесса с помощью операции "op", а затем помещает результат в буфер "recvbuf" процессов ранга "root". MPI_Datatype и MPI_OP принимают интуитивно понятные значения, такие как "MPI_DOUBLE" или "MPI_SUM". Другие часто используемые в MPI подпрограммы служат для широковещательной передачи сообщения (MPI_Bcast), определения барьерных точек синхронизации (MPI_Barrier), отправки (MPI_Send) или получения (MPI_Recv) сообщения.

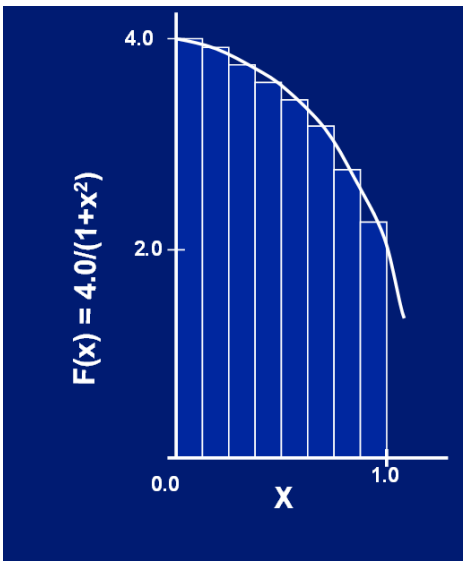


Рисунок 3.

*Пример: интегрирование по формуле прямоугольников*

Из курса математики известно, что интеграл можно представить геометрически в виде площади под кривой. Отсюда следует алгоритм приближенного вычисления значения интеграла. Участок интегрирования разбивается на большое число отрезков, как показано на рис.3. Каждый из отрезков становится основанием прямоугольника, высота которого равна значению подынтегральной функции в центре данного отрезка. Приближенное значение интеграла равно сумме площадей всех прямоугольников. Простая программа на языке С, реализующая этот алгоритм, выглядит следующим образом:

```
static long num_steps = 100000;
double step;
void main ()
{
int i;
double x, pi, sum = 0.0;
step = 1.0/(double) num_steps;
for (i=0;i<= num_steps; i++){
x = (i+0.5)*step;
sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
}
```

*Программа, распараллеленная с помощью OpenMP:*

Языковые конструкции в OpenMP определены как директивы компилятора, которые сообщают компилятору, что он должен делать, чтобы реализовать требуемый параллелизм. В C и C++ такие директивы называются "прагмы". Прагма OpenMP всегда имеет один и тот же вид: #pragma omp construct_name one_or_more_clauses

Имя_конструкции - это параллельное действие, которое требуется программисту, а операторы позволяют изменить это действие или управлять средой данных, которую наблюдают потоки. Для создания потока в OpenMP используется конструкция "parallel". Если такая конструкция используется без уточняющих операторов, то число потоков, которые создает программа, определяется средой выполнения (обычно это число равно числу процессоров или ядер). Каждый поток будет выполнять блок инструкций, который следует за прагмой parallel. Это может быть почти любой набор разрешенных инструкций в C; единственным ограничением является запрет на переходы внутрь этого блока инструкций или из него. Если вдуматься, это не лишено смысла. Если потоки выполняют весь набор инструкций, а результат работы программы должен быть осмысленным, то потоки не должны беспорядочно перемещаться внутрь конструкции в рамках параллельного участка или из нее. В OpenMP это - общее ограничение. Такой блок инструкций без переходов называется "структурированным блоком".

Самая типичная конструкция для совместной работы - это конструкция цикла (в C это цикл for). Конструкция for распределяет итерации цикла между потоками группы, созданными ранее с помощью конструкции parallel. Необходимо отметить, что сама по себе конструкция for потоки не создает. Это можно сделать только с помощью конструкции parallel. Для простоты можно поместить конструкции parallel и for в одну и ту же прагму (#pragma omp parallel for). При этом будет создана группа потоков для выполнения итераций цикла, который следует непосредственно за прагмой. Итерации цикла должны быть независимыми, чтобы результат выполнения цикла оставался неизменным независимо от того, в каком порядке и какими потоками выполняются эти итерации. Если один поток записывает переменную, которую затем считывает другой поток, то наблюдается кольцевая зависимость, и результат работы программы будет неверным. Программист должен тщательно проанализировать тело цикла, чтобы убедиться в отсутствии кольцевых зависимостей. В большинстве случаев такая зависимость возникает, если в переменную записываются промежуточные результаты, которые используются в данной итерации цикла. В этом случае приобретенной зависимости можно избежать, объявив, что каждый поток должен иметь собственное значение для этой переменной. Это можно сделать с помощью оператора private. Например, если в цикле используется переменная "tmp", в которой хранится временное значение, к конструкции OpenMP можно добавить следующий оператор,

после чего переменную можно будет использовать в теле цикла, не создавая приобретенных зависимостей. Кроме того, часто встречается ситуация, когда переменная внутри цикла используется для сложения значений, полученных в каждой итерации. Например, это происходит в цикле, который суммирует результаты вычислений, чтобы получить одно итоговое значение. Такая ситуация часто возникает в параллельном программировании. Она называется "редукция". В OpenMP имеется оператор reduction: reduction(+:sum). Как и оператор private, он добавляется в конструкцию OpenMP, чтобы сообщить компилятору, что следует ожидать редукции. После этого создается временная закрытая переменная, которая используется для получения промежуточного результата операции суммирования для каждого потока. В конце выполнения конструкции значения этой переменной для каждого потока суммируются, чтобы получить конечный результат. Операция, которая используется при редукции, также указывается в операторе. В данном случае это операция "+". OpenMP определяет значение для закрытой переменной, которая используется в редукции, на основе соответствующей математической операции. Например, для "+" это значение равно нулю. Чтобы не усложнять задачу, будем использовать фиксированное число шагов. Кроме того, будет использоваться только число потоков по умолчанию. В последовательной программе имеется единственный цикл, который требуется распараллелить. Итерации цикла полностью независимы, если не считать значения зависимой переменной "x" и накопительной переменной "sum". Необходимо отметить, что "x" используется как временное хранилище для вычислений в итерации цикла. Соответственно, эту переменную необходимо сделать локальной для каждого потока с помощью оператора private. С технической точки зрения, управляющий счетчик цикла создает приобретенную зависимость. Впрочем, в OpenMP управляющий счетчик цикла автоматически становится локальным (закрытой переменной) для каждого потока.

```
#include "omp.h"  // включен стандартный файл для OpenMP
static long num_steps = 100000;
double step;
void main ()
{
int i;
double x, pi, sum = 0.0;
step = 1.0/(double) num_steps;
#pragma omp parallel for private(x)  reduction(+:sum)
for (i=0;i<= num_steps; i++){
x = (i+0.5)*step;
sum = sum + 4.0/(1.0+x*x);
```

```
}
pi = step * sum;
}
```

*Программа, распараллеленная с помощью MPI:*

MPI-программа является непосредственной модификацией исходного последовательного кода. Чтобы избежать излишнего усложнения, число шагов по-прежнему будет задаваться в программе, а не вводиться и передаваться другим процессам. В начале программы подключается файл MPI, чтобы определить типы данных, константы и подпрограммы MPI. Затем следует стандартная тройка подпрограмм, чтобы инициализировать среду MPI и сделать основные параметры (число процессов и ранг) доступными для программы.

```
#include "mpi.h"
static long num_steps = 100000;
void main (int argc, char *argv[])
{
int i, my_id, numprocs;
double x, pi, step, sum = 0.0 ;
step = 1.0/(double) num_steps ;
// вызов трех подпрограмм для настройки применения MPI
MPI_Init(&argc, &argv) ;
MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
my_steps = num_steps/numprocs ;
for (i=my_id; i<num_steps; i+numprocs)
{
x = (i+0.5)*step;
sum += 4.0/(1.0+x*x);
}
sum *= step ;
MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD) ;
/*В конце каждой MPI-программы должна присутствовать подпрограмма,
закрывающая среду. Эта функция возвращает целочисленное значение,
представляющее собой код ошибки*/
MPI_Finalize(ierr);
}
```

После этого используется стандартный прием для разделения итераций цикла между набором процессов. Необходимо отметить, что начальное и конечное значения счетчика

цикла были изменены таким образом, чтобы счетчик цикла изменялся от идентификатора каждого процесса до числа итераций с шагом, равным числу процессов в группе. Это работает, поскольку ранг в MPI используется как идентификатор и может принимать значения от нуля до числа процессов минус один. В сущности, в результате этого простого изменения итерации цикла распределяются путем циклического перебора, как если бы разным процессам сдавали колоду карт. После завершения работы каждого процесса он вносит свой вклад в сложение, помещая готовую промежуточную сумму в переменную "sum", а затем выполняется редукция с помощью следующего вызова:

```
MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD;
```

Промежуточная сумма, "sum", используется как буфер отправки, а переменная "pi" - как буфер приема. Это значение будет доставлено процессу ранга "0", который указан в качестве шестого аргумента подпрограммы MPI_Reduce. Буфер отправки содержит одно значение типа MPI_DOUBLE с операцией накопления суммирования (MPI_SUM). И, наконец, процессы, участвующие в этой редукции, используют коммуникатор MPI_COMM_WORLD.

## Обзор технических средств высокопроизводительных вычислений

Несмотря на то, что современные компьютеры имеют многоядерные процессоры и могут использоваться для параллельных вычислений, реальные задачи требуют значительно большей производительности. Для решения сложных задач науки, промышленности, экономики применяются суперкомпьютеры. В зависимости от прикладной задачи суперкомпьютер может быть построен по-разному. Самыми дорогими, но и самыми эффективными являются машины на основе закрытой коммерческой архитектуры. Они требуют специального программного обеспечения и обычно не используются для промышленных задач. Примерами таких систем являются BlueGene и RoadRunner компании IBM, а также старшие модели компьютеров компании CRAY. Наиболее распространенными и универсальными суперкомпьютерами являются системы на основе вычислительных кластеров, собранных из стандартных компонентов. Благодаря относительной дешевизне комплектующих и совместимости с огромным числом готовых коммерческих программных продуктов для инженерных расчетов такие системы стали стандартом “де-факто” в области высокопроизводительных вычислений. Кластерные вычисления представляют собой особую технологию высокопроизводительных вычислений, зародившуюся вместе с развитием коммуникационных средств и ставшую прекрасной альтернативой использованию суперкомпьютеров. Кластер предполагает более высокую надежность и эффективность, нежели локальная вычислительная сеть, и существенно более низкую стоимость в сравнении

с другими типами параллельных вычислительных систем за счет использования типовых аппаратных и программных решений. В разряд кластерных вычислений фактически входят любые параллельные вычисления, где все компьютеры системы используются как один унифицированный ресурс.

С развитием пропускной способности каналов связи у компьютеров появилась возможность выполнять задачу совместно. На этой основе появилась концепция виртуального суперкомпьютера или метакомпьютера, где масштабная задача выполняется совместно в единой сети обычных компьютеров. Для пользователя такой системы создается полная иллюзия, что он работает с одним устройством. Компьютеры могут быть удалены друг от друга и могут использовать разные типы коммуникаций, однако для конечного программного продукта и пользователя они играют роль единой вычислительной машины.

Развитием и обобщением идей метакомпьютинга являются Грид-технологии. В качестве процессорных мощностей рассматриваются не только суперкомпьютеры, а вообще любые компьютеры. Разделяемые ресурсы: коммуникации, данные, программное обеспечение,процессорное время. В грид-системах пользователю не нужно знать о физическом расположении ресурсов, отведенных его задаче, и для него создается иллюзия работы в едином информационном пространстве, обладающем огромными вычислительными мощностями и объемом памяти. Основные вычислительные мощности сосредоточены не только в суперкомпьютерных центрах, но и в простаивающем в какой-либо лаборатории или организации компьютерном парке. Согласно авторам данной концепции (Я. Фостер, К. Кессельман), Грид-вычисления (англ.grid – решетка, сеть) – это согласованная, открытая и стандартизованная среда, которая обеспечивает гибкое, безопасное, скоординированное разделение (общий доступ) ресурсов в рамках виртуальной организации. Для грид характерно отсутствие центра управления вычислительными ресурсами, использование открытых стандартов и нетривиальный уровень обслуживания.

Альтернативой многоядерным процессорам являются графические процессоры, которые с некоторых пор стали использовать для вычислений. GPGPU (англ. General-purpose graphics processing units — «GPU общего назначения») — техника использования графического процессора видеокарты, который обычно имеет дело с вычислениями только для компьютерной графики, чтобы выполнять расчёты в приложениях для общих вычислений, которые обычно проводит центральный процессор. Это стало возможным благодаря добавлению программируемых шейдерных блоков и более высокой арифметической точности растровых конвейеров, что позволяет разработчикам ПО использовать потоковые процессоры для неграфических данных. Технология CUDA (англ. Compute Unified Device Architecture) - программно-аппаратная архитектура, позволяющая производить вычисле-

ния с использованием графических процессоров NVIDIA, поддерживающих технологию GPGPU. Архитектура CUDA впервые появились на рынке с выходом чипа NVIDIA восьмого поколения - G80 и присутствует во всех последующих сериях графических чипов, которые используются в семействах ускорителей GeForce, ION, Quadro и Tesla. CUDA SDK позволяет программистам реализовывать на специальном упрощённом диалекте языка программирования Си алгоритмы, выполнимые на графических процессорах NVIDIA и включать специальные функции в текст программы на Си. CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического ускорителя и управлять его памятью, организовывать на нём сложные параллельные вычисления.

## Лекция №2. Параллельное программирование в научных вычислениях: сеточные вычисления

История научных вычислений тесно связана с общей историей вычислений. Первые вычислительные машины разрабатывались для решения научных проблем, а Фортран — первый язык высокого уровня — был создан специально для программирования численных методов. Научные вычисления также стали синонимом высокопроизводительных вычислений, заставляя увеличивать предельные возможности самых быстрых машин. В данном разделе представлены популярные и широко известные примеры таких вычислений. При написании программ использовался псевдоязык программирования, основанный на стандартных понятиях из языков C, C++, Java.

Дифференциальные уравнения в частных производных (partial differential equations — PDE) применяются для моделирования разнообразных физических систем: прогноза погоды, обтекания крыла потоком воздуха, турбулентности в жидкостях и т.д. Некоторые простые PDE можно решить прямыми методами, но обычно нужно найти приближенное решение уравнения на конечном множестве точек, применяя итерационные численные методы. В этом разделе показано, как решить одно конкретное PDE — двухмерное уравнение Лапласа — с помощью сеточных вычислений по так называемому методу конечных разностей. Но при решении других PDE и в других приложениях сеточных вычислений, например, при обработке изображений, используется такая же техника программирования.

### Уравнение Лапласа

Уравнение Лапласа является примером так называемого эллиптического дифференциального уравнения в частных производных. В двухмерном варианте это уравнение имеет следующий вид.

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = 0$$

Функция Ф представляет собой некоторый неизвестный потенциал, например, теплоту или напряжение. По данной области пространства и известным значениям в точках на границах этой области нужно аппроксимировать стационарное решение во внутренних точках области. Это можно сделать, покрыв область равномерной сеткой точек (рис.4). Каждая внутренняя точка инициализируется некоторым значением. Затем с помощью повторяемых итераций вычисляются стационарные значения внутренних точек. На каждой итерации новое значение точки является комбинацией старых и/или новых значений соседних точек. Вычисления прекращаются либо после определенного количества итераций,

либо тогда, когда разность между каждым новым и соответствующим предыдущим значением становится меньше заданной величины.


Рисунок 4. Аппроксимация уравнения Лапласа с помощью сетки

Для решения уравнения Лапласа существует несколько итерационных методов: Якоби, Гаусса—Зейделя, последовательная сверхрелаксация (successive over-relaxation—SOR) и многосеточный. Ниже будет показано, как запрограммировать метод итераций Якоби (с помощью разделяемых переменных и передачи сообщений), поскольку он наиболее прост и легко распараллеливается.

## Метод последовательных итераций Якоби

В методе итераций Якоби новое значение в каждой точке сетки равно среднему из предыдущих значений четырех ее соседних точек (слева, справа, сверху и снизу). Этот процесс повторяется, пока вычисление не завершится. Ниже строится простая последовательная программа и приводится ряд оптимизаций кода, повышающих производительность программы.

Предположим, что сетка представляет собой квадрат размером nxn и окружена квадратом граничных точек. Одна матрица нужна для представления области и ее границы, другая — для хранения новых значений.

real grid[0:n+1,0:n+1], new[0:n+1,0:n+1];

Границы обеих матриц инициализируются соответствующими граничными условиями, а внутренние точки — некоторым начальным значением, например 0.

Ниже представлена уже оптимизированная последовательная программа для метода итераций Якоби. Для оптимизации деление заменено умножением; для завершения вычислений использовано конечное число итераций, а максимальная разность вычисляется только один раз; цикл развернут дважды и его тело переписано, поскольку дополнительный индекс и операторы изменения ролей матриц не нужны.

*Оптимизированная последовательная программа для метода итераций Якоби:*

```
real grid[0:n+1,0:n+1], new[0:n+1,0:n+1];
real maxdiff = 0.0;
{ инициализация сеток, включая границы;}
```

```
   for [iters = 1 to MAXITERS by 2]  {
   # вычислить новые значения во всех внутренних точках
       for [i = 1 to n, j = 1 to n]
          new[i,j]  =  (grid[i-l,j]  +  grid[i+l,j]  +grid[i,j-l]  +
grid[i,j+1]) * 0.25;
   # снова вычислить новые значения во внутренних точках
      for [i = 1 to n, j = 1 to n]
         grid[i,j]  =  (new[i-l,j]  +  new[i+l,j]  +new[i,j-l]  +
new[i,j+l]) * 0.25;
   }
   # вычислить максимальную разность
      for [i = 1 to n, j = 1 to n]
   maxdiff = max(maxdiff, abs(grid[i,j]-new[i,j]));
```

Здесь maxiters — константа или аргумент в командной строке. Если программист при запуске нового кода обнаружит, что конечное значение maxdiff слишком велико, следует перезапустить программу с большим значением maxiters. Чтобы подобрать нужное значение maxiters, достаточно сделать несколько пробных запусков программы. Затем подобранное таким образом значение maxiters можно использовать в рабочих запусках программы.

Представленную программу можно оптимизировать еще двумя способами. Поскольку новые значения точек вычисляются дважды в теле внешнего цикла for, из первого цикла вычислений можно убрать умножение на о. 25, а во втором вставить умножение на 0.0625. Такая замена избавит от половины всех умножений. Программу можно улучшить, встраивая вызовы функций в последнем цикле, вычисляющем максимальную разность. Вызовы функций max и abs можно заменить телами этих функций:

temp = grid[I,j]-new[I,j]; if (temp < 0) temp = -temp; if (temp > maxdiff) maxdiff = temp;

Это позволяет избежать накладных расходов на два вызова функций. Здесь использовано также то, что maxdiff и является аргументом max, и сохраняет результат. Встраивание функции мало повышает производительность, поскольку максимальная разность 26женные26ется в программе только один раз. Однако, если функция вызывается во внутреннем 26желе, выполняемом многократно, встраивание может оказаться очень эффективным.

### Метод итераций Якоби с разделяемыми переменными

Рассмотрим, как распараллелить метод итераций Якоби. Предположим, что есть PR процессоров, и размерность сетки n намного больше pr. Чтобы параллельная программа была эффективной, желательно распределить вычисления между процессорами поровну. Для данной задачи сделать это несложно, поскольку для обновления каждой точки сетки нужно одно и то же количество работы. Следовательно, можно использовать pr процессоров так, чтобы каждый из них отвечал за одно и то же число точек сетки. Можно разделить сетку на pr прямоугольных блоков или прямоугольных полос. Используем полосы, поскольку для них немного проще написать программу. Вероятно также, что использование полос более эффективно, поскольку при длинных полосах локальность данных выше, чем при коротких блоках, что оптимизирует использование 27жен данных. Предположив для простоты, что n кратно pr, а массивы хранятся в памяти по строкам, назначим 27жждому процессу горизонтальную полосу размера n/PR x n. Каждый процесс обновляет свою полосу точек. Однако, поскольку процессы имеют общие точки, располо27женные на границах полос, после каждой фазы обновлений нужна барьерная синхронизация для того, чтобы все процессы завершили фазу обновлений перед тем, как любой процесс начнет выполнять следующую.

Ниже приведена параллельная программа для метода итераций Якоби с разделяемыми переменными. Использован стиль “одна программа — много данных” (single program, multiple data — SPMD): каждый процесс выполняет один и тот же код, но обрабатывает различные части данных. Здесь каждый рабочий процесс сначала инициализирует свои полосы, включая границы, в двух матрицах. Тело каждого рабочего процесса основано на последовательной программе. Синхронизировать процессы можно используя барьеры.

*Метод итераций Якоби с разделяемыми переменными:*

```
real grid[0:n+1,0:n+1], new[0;n+1,0:п+1] ;
int HEIGHT = n/PR; # предполагается, что n кратно PR
real maxdiff[1:PR] = ([PR]  0.0);
process worker[w = 1 to PR] {
     int firstRow = (w-1)*HEIGHT + 1;
  int lastRow = firstRow + HEIGHT - 1;
     real mydiff = 0.0;
     инициализация своей полосы, включая границы, в grid и new;
     синхронизация процессов
     for [iters =1 to MAXITERS by 2] {
# вычислить новые значения в своей полосе
```

```
    for [i = firstRow to lastRow, j = 1 to n]
        new[i,j] = (grid[i-1,j]+grid[i+1,j]+grid[i,j-
1]+grid[i,j+1])*0.25;
                    синхронизация процессов;
    #снова вычислить значения в своей полосе
for [i = firstRow to lastRow, j = 1 to n]
     grid[i,j] = (new[i-1,j]+new[i+1,j]+new[i,j-1] + new[i,j+1])* 0.25;
                    синхронизация процессов;
                }
           # вычислить максимальную разность в своей полосе
          for [i = firstRow to lastRow, j = 1 to n]
              mydiff = max(mydiff, abs(grid[i,j]-new[i,j]));
          maxdiff[w] = mydiff;
          синхронизация процессов;
         #максимальная разность — это максимум в maxdiff[*]
    }
```

Эта программа также иллюстрирует эффективный способ вычисления максимальной разности во всех парах окончательных значений в точках сетки. Каждый рабочий процесс вычисляет максимальную разность для точек в своей полосе, используя локальную переменную mydiff, а затем сохраняет полученное значение в массиве максимальных разностей. После второго барьера каждый рабочий процесс может параллельно вычислить максимальное значение в maxdiff [*] (все эти вычисления можно также проводить только в одном рабочем процессе). Локальные переменные в каждом процессе позволяют избежать использования критических секций для защиты доступа к единственной глобальной переменной, а также конфликтов в кэше, которые могли бы возникнуть из-за ложного разделения массива maxdiff. Программа могла бы работать немного эффективнее, если встроить вызовы процедуры barrier. Однако встраивание кода вручную сделает программу тяжелой для чтения. Поэтому лучше всего использовать компилятор, поддерживающий оптимизацию встраивания.

### Метод итераций Якоби с передачей сообщений

Рассмотрим реализацию метода итераций Якоби на машине с распределенной памятью, используя передачу сообщений. Один из способов написать параллельную программу с передачей сообщений — модифицировать программу с разделяемыми переменными. Сначала разделяемые переменные распределяются между процессами, затем в тех местах, где процессам нужно обменяться данными, добавляются операторы отправки и получе-

ния. Вновь используем PR рабочих процессов, обновляющих каждый раз полосу точек сетки. Распределим массивы grid и new так, чтобы полосы были локальными для соответствующих рабочих процессов. Также нужно продублировать строки на краях полос, поскольку рабочие не могут считывать данные, размещенные в других процессах. Таким образом, каждому процессу нужно хранить точки не только своей полосы, но и границ соседних полос. Каждый процесс выполняет последовательность фаз обновления; после каждого обновления он обменивается краями своей полосы с соседями. Такая схема обмена соответствует алгоритму пульсации. Обмены заменяют точки барьерной синхронизации. Нужно также распределить вычисление максимальной разности после того, как каждый процесс завершит обновление в своей полосе сетки. Как и ранее, каждый процесс просто вычисляет максимальную разность в своей полосе. Затем выбирается один процесс, который собирает все полученные значения. Все это можно запрограммировать, используя либо сообщения, передаваемые от процесса к процессу, либо коллективное взаимодействие, как в MPI.

Далее представлена программа для метода итераций Якоби с передачей сообщений. В каждой из двух фаз обновления используется алгоритм пульсации, поэтому соседние процессы дважды обмениваются краями во время одной итерации главного вычислительного цикла. Первый обмен программируется следующим образом.

if (w > 1) send up[w-l](new[l,*]);

if (w < PR) send down[w+1](new[HEIGHT,*]);

if (w < PR) receive up[w](new[HEIGHT+l,*]);

if (w > 1) receive down[w](new[0,*]);

Все процессы, кроме первого, отправляют верхний ряд своей полосы соседу сверху. Все процессы, кроме последнего, отправляют нижний ряд своей полосы соседу снизу. Затем каждый получает от своих соседей края; они становятся границами его полосы. Второй обмен идентичен первому, только вместо new используется grid.

*Метод итераций Якоби с передачей сообщений*

```
chan up[1:PR](real edge[0:n+1]);
chan down[1:PR](real edge[0:n+l]);
chan diff(real);
process worker[w = 1 to PR] {
      int HEIGHT = n/PR;      # n кратно PR
      real grid [0 : HEIGHT+1,0 : n+1 ] , new[ 0:HEIGHT+1,0 : n+1];
      real mydiff = 0.0, otherdiff = 0.0;
      инициализация grid и new с границами;
```

```
          for [iters = 1 to MAXITERS by 2]  {
         #вычислить новые значения в своей полосе
             for [i = 1 to HEIGHT, j = 1 to n]
      new[i,j]=(grid[i-1,j]+grid[i+1,j]+grid[i,j-1]+grid[i,j+1]) *0.25;
    обмен краями в матрице new (см. текст);
    # снова вычислить новые значения в своей полосе
    for [i = 1 to HEIGHT, j = 1 to n]
    grid[i,j] = (new[i-1,j] + new[i+1,j] +new[i,j-1] + new[i,j+1]) *
0.25;
    обмен краями в матрице grid (см. текст);
    }
    #вычислить максимальную разность в своей полосе
    for [i = 1 to HEIGHT, j = 1 to n]
        mydiff = max(mydiff, abs(grid[i,j]-new[i,j]));
     if (w > 1)
        send diff(mydiff) ;
    else    # рабочий процесс 1 собирает разности
       for [i = 1 to w-1] {
    receive diff(otherdiff);
    mydiff = max(mydiff, otherdiff);
    }
    # максимальная разность - это значение mydiff в процессе 1
    }
```

После подходящего числа итераций каждый рабочий процесс вычисляет максимальную разность в своей полосе, затем первый из них собирает полученные значения. Как отмечено в конце листинга, глобальная максимальная разность равна окончательному значению mydiff в первом процессе.

Программу можно оптимизировать. Во-первых, для данной программы и многих других сеточных вычислений нужно проводить обмен краями после каждой фазы обновлений. Здесь, например, можно обменивать края после каждой второй фазы обновлений. Это вызовет “скачки” значений в точках на краях, но, поскольку алгоритм сходится, он все равно будет давать правильный результат. Во-вторых, можно перепрограммировать оставшийся обмен так, чтобы между отправкой и получением сообщений выполнялись локальные вычисления. Например, можно реализовать взаимодействие, при котором каждый рабочий: 1) отправляет свои края соседям; 2) обновляет внутренние точки своей полосы; 3) получает края от соседей; 4) обновляет края своей полосы. Такой подход значительно повысит вероятность того, что края от соседей придут раньше, чем они нужны, и,

следовательно, задержки операций получения не будет. Таким образом, получим оптимизированную программу.

*Улучшенный метод итераций Якоби с передачей сообщений*

```
chan up[l:PR](real edge[0;n+1]);
chan down[l:PR](real edge[0:n+1]);
chan diff(real);
process worker[w = 1 to PR] {
    int HEIGHT = n/PR;    # n кратно PR
    real grid[0 :HEIGHT+1, 0 :n+1] , new[0:HEIGHT+1, 0:n+1] ;
    real mydiff = 0.0, otherdiff = 0.0;
    инициализация grid и new, включая границы;
    for [iters = 1 to MAXITERS by 2]    {
        #вычислить новые значения в своей полосе
        for [i = 1 to HEIGHT, j = 1 to n]
new[i,j] =(grid[i-1,j]+grid[i+1,j] +grid[i,j-1] + grid[i,j+1]) *
0.25;
        #отправить края в new соседям
        if (w > 1) send up[w-1] (new[1,*] ) ;
        if (w < PR) send down[w+1](new[HEIGHT,*]);
        #вычислить новые значения во внутренних точках своей полосы
        for [i = 2 to HEIGHT-1, j = 1 to n]
grid[i,j] =(new[i-1,j] + new[i+1,j] + new[i,j-1] + new[i,j+1]) *
0.25;
        #получить края в new от соседей
        if (w < PR) receive up[w](new[HEIGHT+1,*]);
        if (w > 1) receive down[w](new[0,*]);
        #вычислить новые значения на краях своей полосы
        for [j = 1 to n]
grid[1,j] = (new[0,j] + new[2,j] + new[l,j-1] + new[l,j+1]) * 0.25;
        for [j = 1 to n]
grid[HEIGHT,j] = (new[HEIGHT-1,j]+new[HEIGHT+1,j]+new[HEIGHT,j-1] +
new[HEIGHT,j+1]) * 0.25;
}
вычислить максимальную разность как в предыдущей программе
}
```

**Лекция №3. Параллельное программирование в научных вычислениях: точечные вычисления**

Сеточные вычисления обычно используются для моделирования непрерывных систем, которые описываются дифференциальными уравнениями в частных производных. Для моделирования дискретных систем, состоящих из отдельных частиц (точек), воздействующих друг на друга, применяются точечные методы. Примерами являются заряженные частицы, взаимодействующие с помощью электрических и магнитных сил, молекулы (их взаимодействие обусловлено химическим строением) и астрономические тела, воздействующие друг на друга благодаря гравитации. Здесь рассматривается типичное приложение, называемое гравитационной задачей n тел. После постановки задачи представлены последовательная и параллельная программы, основанные на алгоритме сложности O(n).

## Гравитационная задача n тел

Предположим, что дано большое число астрономических тел галактики (звезд, пылевых облаков и черных дыр), и нужно промоделировать ее эволюцию. Каждое тело имеет массу, начальное положение и скорость. Гравитация вызывает перемещение и ускорение тел. Движение системы n тел имитируется пошагово с помощью дискретных отрезков времени. На каждом временном шаге вычисляются силы, действующие на каждое тело, и обновляются скорости и положения тел. Таким образом, имитация гравитационной задачи n тел имеет следующую структуру:

```
инициализировать тела
for [time = start to finish by DT] {
вычислить силы;
  переместить тела;
}
```

Значение DT (delta time) является временным шагом.

Согласно физике Ньютона величина силы гравитации между двумя телами i и j вычисляется по формуле

$$F = \frac{G\, m_i\, m_j}{r^2}$$

где $m_i, m_j$— массы тел, r — расстояние между ними. Гравитация является чрезвычайно слабым взаимодействием, поэтому значение гравитационной постоянной G— очень малое число $6{,}67 \cdot 10^{-11}$. Предположим для простоты, что все тела расположены на одной плоскости, т.е. задача является двухмерной. Пусть $(p_i x, p_i y)$ и $(p_j x, p_j y)$ — положения двух тел. Евклидово расстояние r между ними определяется по формуле

$$\sqrt{(p_i x - p_j x)^2 + (p_i y - p_j y)^2}$$

Если два тела подходят слишком близко друг к другу (близки к столкновению), r очень мало. Это приводит к значительной неустойчивости при вычислении сил, однако здесь данная проблема не рассматривается.

Направление силы, действующей на тело i со стороны тела j, задается единичным вектором, указывающим от i в сторону j, а силы воздействия тела i на тело j — противоположным вектором. Таким образом, величины сил, действующих между любыми двумя телами, равны, а направления противоположны. Общая сила, действующая на тело, есть сумма сил воздействия всех остальных тел. (Поскольку суммы векторов коммутативны, векторы сил можно складывать в любом порядке.)

Гравитационные силы, действующие на тело, вызывают его ускорение и перемещение. Отношение между силой, массой и ускорением описывается знаменитым уравнением $a = {}^{F}/_{m}$, т.е. ускорение тела i равно общей силе, действующей на тело, деленной на его массу. Если за малый интервал времени $dt$ ускорение $a_i$ тела i остается практически постоянным, то изменение скорости приблизительно равно $dv_i = a_i dt$. Изменение положения тела есть интеграл его скорости и ускорения на интервале времени $dt$, который приблизительно равен

$$dp_i = v_i dt + \frac{a_i}{2} dt^2 = (v_i + \frac{dv_i}{2}) dt$$

Эта формула устроена по так называемой скачкообразной (leapfrog — чехарда) схеме, в которой одна половина изменения в положении тела обусловлена предыдущей скоростью, а другая — новой.

Здесь представлена последовательная программа для решения задачи n тел. В программе использован простой алгоритм сложности O(n), о чем свидетельствует цикл for процедуры calculateForces. Для каждого тела вычисляются силы, действующие на него со стороны других тел. При этом учитывается свойство симметрии сил, действующих между телами: каждая пара тел рассматривается только один раз (второй индекс j в главном цикле вычислений изменяется от i+1 до n).

*Последовательная программа для решения задачи n тел выглядит следующим образом*

```
type point = rec(double x,y);
point p[1:n], v[1:n], f[1:n];       # положение,скорость, сила и
double m[1:n];                  # масса тел
double G = 6.67e-11;
```

```
инициализировать положения, скорости, силы и массы;
#вычислить общую силу для всех пар тел
 procedure calculateForces() {
    double distance, magnitude;
    point direction;
    for [i = 1 to n-1, j = i+1 to n] {
      distance = sqrt( (p[i].x - p[j].x)**2 +(P[i].y - p[j].y)**2);
       magnitude = (G*m[i]*m[j]) / distance**2;
       direction = point(p[j].x-p[i].x, p[j].y-p[i].y);
       f[i].x = f[i].x + magnitude*direction.x/distance;
       f[j].x = f[j].x - magnitude*direction.x/distance;
       f[i].y = f[i].y +, magnitude*direction.y/distance;
       f[j].y = f[j].y - magnitude*direction.y/distance;
   }
}
# вычислить новые скорости и положения тел
procedure moveBodies() {
    point deltav;       # dv = f/m * DT
    point deltap;       # dp = (v + dv/2) * DT
    for [I = 1 to n]{
        deltav = point(f[i].x/m[i] *)DT, f[i].y/m[i] * DT);
        deltap = point( (v[i].x + deltav.x/2) * DT, (v[i].y +
deltav.y/2) * DT );
        v[i].x = v[i].x + deltav.x;
        v[i].y = v[i].y + deltav.y;
        p[i].x = p[i].x + deltap.x;
        p[i].y = p[i].y + deltap.y;
        f[i].y = f[i].y = 0.0;     #обнуление вектора силы
     }
}
# имитация с временным шагом DT
for [time = start to finish by DT] {
     calculateForces();
     moveBodies();
}
```

Программу можно сделать более эффективной. Например, сократить мощность, заменив операции деления и возведения в степень умножением. Кроме того, некоторые выра-

жения можно вычислять не многократно, а только один раз. Встраивание двух процедур немного поможет, но усложнит понимание программы.

### Программа с разделяемыми переменными

Рассмотрим, как распараллелить программу для задачи n тел. Как всегда, вначале нужно решить, как распределить работу. Предположим, что есть PR процессоров, и, следовательно, будут использованы PR рабочих процессов. В реальных задачах n намного больше PR. Предположим, что n кратно PR.

Большая часть расчетов в последовательной программе проводится в цикле for процедуры calculateForces. В методе итераций Якоби работа просто распределялась по полосам, и на каждый рабочий процесс приходилось по n/ PR строк сетки. В результате вычислительная нагрузка была сбалансированной, поскольку по методу итераций Якоби в каждой точке сетки выполняется один и тот же объем работы.

Для задачи n тел соответствующим разделением было бы назначение каждому рабочему процессу непрерывного блока тел: процесс 1 обрабатывает первые n/PR тел, 2— следующие n/PR тел и т.д. Однако это привело бы к слишком несбалансированной нагрузке. Процессу 1 придется вычислить силы взаимодействия тела 1 со всеми остальными, затем — тела 2 со всеми, кроме тела 1, и т.д. С другой стороны, последнему процессу нужно вычислить только те силы, которые не были вычислены предыдущими процессами, т.е. лишь между последними n/ PR телами.

Пусть для конкретного примера n равно 8, а PR — 2, т.е. используются два рабочих процесса. Назовем их черным (black — B) и белым (white — W). На рис.5 показаны три способа назначения тел процессам. При блочном распределении первые четыре тела назначаются процессу B, последние четыре — W. Таким образом, число пар сил, вычисляемых B, равно 22 (7 + 6 + 5 + 4), а вычисляемых W — 6 (3 + 2 +1).

| | 1 2 3 4 5 6 7 8 | |
|---|---|---|
| *Схема* | | *Рабочая нагрузка* |
| блоки | B B B B W W W W | B = 22, W = 6 |
| полосы | B W B W B W B W | B = 16, W = 12 |
| обратные полосы | B W W B B W W B | B = 14, W = 14 |

Рисунок 5. Схемы распределения нагрузки между процессорами

Вычислительная нагрузка будет более сбалансированной, если назначать тела иначе: 1 — процессу B, 2 — W, 3 — B и т.д. Эта схема называется распределением по полосам по аналогии с полосами зебры. (Она называется еще циклическим распределением, поскольку схема повторяется.) Распределение по полосам приводит к следующей нагрузке про-

цессов: 16 пар сил для B и 12 — для W. Нагрузку можно еще больше сбалансировать, если распределить тела по схеме, похожей на ту, которая обычно используется при выборе команд в спортивных играх: назначим тело 1 процессу B, тела 2 и 3 — W, тела 4 и 5 — B и т.д. Эта схема называется распределением по обратным полосам, поскольку полосы каждый раз меняют порядок: черный, белый; белый, черный; черный, белый и т.д. Таким образом в данном примере нагрузка полностью сбалансирована — по 14 пар сил на каждый процесс.

Любая из приведенных выше стратегий распределения легко обобщается на любое число PR рабочих процессов. Блочная стратегия тривиальна. Для полос используется PR различных цветов. Первым цветом окрашивается тело 1, вторым — тело 2 и т.д. Цикл повторяется, пока все тела не будут окрашены. При использовании обратных полос порядок цветов меняется, когда начинается новый цикл, т.е. после окраски PR тел.

Для этой и любой другой задачи с похожей схемой индексов в главном вычислительном цикле распределение данных по схеме обратных полос дает наиболее сбалансированную вычислительную нагрузку. Однако схема полос программируется намного легче, чем схема обратных полос, и для больших значений n дает практически сбалансированную нагрузку. Поэтому в дальнейшем используем схему полос.

Сначала рассмотрим процедуру calculateForces. В цикле for рассматривается каждая пара тел i и j. Пока один процесс вычисляет силы между телами i и j, другой вычисляет силы между телами i' и j'. Некоторые из этих номеров могут быть равными: например, j в одном процессе может совпадать с i' в другом. Следовательно, процессы могут мешать друг другу при обновлении вектора сил f. Таким образом, четыре оператора присваивания, обновляющих f, образуют код критической секции.

Один из способов защитить критическую секцию — использовать одну переменную блокировки. Однако это неэффективно, поскольку критическая секция выполняется много раз каждым процессом. В другом предельном случае можно использовать массив переменных блокировки, по одной на тело. Это почти устраняет конфликты при блокировке, но за счет использования гораздо большего объема памяти. Промежуточный способ — использовать одну переменную блокировки на каждый блок из B тел, но и тут возникают конфликты блокировки. Для хорошей производительности лучше всего вообще устранить накладные расходы на блокировку, избавившись от критических секций (если возможно). Это можно сделать двумя способами. При первом каждый рабочий процесс берет на себя полную ответственность за назначенные ему тела, т.е. процесс, отвечающий за тело i, вычисляет все силы, действующие на тело i, но не обновляет вектор сил для любого другого тела j. Взамен, процесс, отвечающий за тело j, вычисляет все силы, действующие на него,

в том числе и со стороны тела i. Такой способ можно запрограммировать, используя индексную переменную j в for цикле calculateForces со значениями в диапазоне от 1 до n, а не от i+1 до n, и исключая присваивания f [j]. Однако при этом не используется симметричность сил между двумя телами, так что все силы вычисляются дважды. Второй способ устранения критических секций — вектор сил заменить матрицей сил, каждая строка которой соответствует одному процессу. Вычисляя силу между телами i и j, процесс W обновляет два элемента своей строки в матрице сил. А когда нужно будет вычислить новые положение и скорость тела, он сначала сложит все силы, действующие на данное тело и вычисленные ранее всеми остальными процессами. Оба способа устранения критических секций используют дублирование. В первом методе дублируются вычисления, во втором — данные. Это еще один призер пространственно- временного противоречия. Поскольку целью параллельной программы обычно является уменьшение времени выполнения, то лучший выбор — использовать как можно больше пространства (разумеется, в пределах доступной памяти).

Осталось рассмотреть еще один важный момент — нужны ли барьеры, и если да, то где именно. Значения новых скоростей и положений, вычисляемых в moveBodies, зависят от сил, вычисленных в calculateForces. Поэтому, пока не будут вычислены все силы, нельзя перемещать тела. Аналогично силы зависят от положений и скоростей, поэтому нельзя пересчитывать силы, пока не перемещены тела. Таким образом, после каждого вызова процедур calculateForces и moveBodies нужна барьерная синхронизация.

Итак, здесь описаны три способа назначения тел процессам. Использование схемы полос приводит к вполне сбалансированной и легко программируемой вычислительной нагрузке. Рассмотрена проблема критической секции, показано, как применять блокировку и избегать ее. Наиболее эффективен следующий подход: исключить критические секции, чтобы каждый процесс обновлял свой собственный вектор сил. Наконец, независимо от распределения рабочей нагрузки и управления критическими секциями нужны барьеры после вычисления сил и перемещения тел.

Все представленные решения включены в код. В основном структура программы совпадает со структурой последовательной программы. Для реализации барьерной синхронизации добавлена третья процедура barrier. Единый главный цикл имитации заменен имитацией на PR рабочих процессорах, и каждый из них выполняет цикл имитации. Процедура barrier вызывается после как вычисления сил, так и перемещения тел. Во всех вызовах процедур содержится аргумент w, который указывает на вызывающий процесс.

*Общий вид программы с разделяемыми переменными*

```
type point=rec(double x,y); double G=6.67e-11;
point p[1:n], v[1:n], f[1:PR,1:n];  # положения,
double m[l:n];                  # скорости, силы и массы тел
инициализировать положения, скорости, силы и массы;
# вычислить силы, действующие на тела, назначенные процессу w
procedure calculateForces(int w) {
    double distance, magnitude; point direction;
  for [i = w to n by PR, j = i+1 to n] {
      distance = sqrt( (p[i].x - p[j].x)**2 +(P[i].y - p[j].y)**2);
       magnitude = (G*m[i]*m[j]) / distance**2;
       direction = point(p[j].x-p[i].x, p[j].y-p[i].y);
       f[w, i].x = f[w, i].x + magnitude*direction.x/distance;
       f[w, j].x = f[w, j].x - magnitude*direction.x/distance;
       f[w, i].y = f[w, i].y +, magnitude*direction.y/distance;
       f[w, j].y = f[w, j].y - magnitude*direction.y/distance;
    }
 }
#переместить тела, назначенные процессу w
 procedure moveBodies(int w) {
     point deltav;      # dv = f/m * DT
     point deltap;      # dp = (v + dv/2) * DT
     point force = (0.0, 0.0);
     for [i = w to n by PR] {
         #сложить силы, действующие на тело i, и обнулить f[*, i]
         for [k = 1 to PR] {
             force.x +=f[k,i].x;   f[k,i].x = 0.0;
             force.y +=f[k,i].y;   f[k,i].y = 0.0;
         }
         deltav = point(force.x/m[i] * DT,  force.y/m[i] * DT);
      deltap= point((v[i].x+deltav.x/2)*DT,(v[i].y+deltav.y/2)*DT);
         v[i].x = v[i].x + deltav.x;
         v[i].y = v[i].y + deltav.y;
         p[i].x = p[i].x + deltap.x;
         p[i].y = p[i].y + deltap.y;
         force.x = force.y = 0.0;
      }
}
process Worker[w = 1 to PR] {
    # имитировать с временным шагом DT
```

```
        for [time = start to finish by DT] {
            calculateForces(w);
            синхронизация процессов;
            moveBodies(w);
            синхронизация процессов;
    }
```

Тела процедур, вычисляющих силы и перемещающих тела, изменены так, как описано выше. Наконец, главные циклы в этих процедурах изменены так, что приращение по i равно PR, а не 1 (этого достаточно, чтобы присваивать тела рабочим по схеме полос). Этот код можно сделать более эффективным, оптимизировав его, как и последовательную программу.

## Программы с передачей сообщений

Рассмотрим, как решить задачу n тел, используя передачу сообщений. Нам нужно построить метод распределения вычислений между рабочими процессами, чтобы рабочая нагрузка была сбалансированной. Необходимо также минимизировать накладные расходы на взаимодействие или хотя бы снизить их по сравнению с объемом полезной работы. Эти проблемы непросты, поскольку в задаче n тел вычисляются силы для всех пар тел, и, следовательно, каждому процессу приходится взаимодействовать со всеми остальными.

### *Программа типа "управляющий-рабочие"*

Применим парадигму "управляющий-рабочие". Управляющий обслуживает портфель задач; рабочие многократно получают задачу, выполняют ее и возвращают результат управляющему. В задаче n тел есть две фазы. В задачах первой фазы вычисляются силы между всеми парами тел, в задачах второй — перемещаются тела. Вычисление сил является основной частью работы, которая может оказаться несбалансированной. Поэтому есть смысл при вычислении сил использовать динамические задачи, а и при перемещении тел — статические (по одной на рабочего).

Предположим, что у нас есть PR процессоров и используются PR рабочих процессов. (Управляющий процесс работает на том же процессоре, что и один из рабочих.) Предположим для простоты, что n кратно PR. При использовании парадигмы "управляющий-рабочие" для сбалансированности нагрузки нужно, чтобы задач было хотя бы вдвое больше, чем рабочих процессов. Однако очень большое число задач или рабочих процессов нежелательно, поскольку рабочие процессы потратят слишком много времени на взаимодействие с менеджером. Один из способов распределить вычисления сил, действующих на n тел, состоит в следующем.

- Множество тел разделяется на PR блоков размером n/PR. Первый блок содержит первые n/PR тел, второй — следующие n/PR тел и т.д.
- Образуются пары (i,j) для всех возможных комбинаций номеров блоков. Таких пар будет PR* (PR+1) / 2 (сумма целых чисел от 1 до PR).
- Пусть каждая пара представляет задачу, например, для пары (i, j) — вычислить силы, действующие между телами в блоках i и j.

В качестве конкретного примера предположим, что PR равно 4. Тогда десять задач представлены парами:

(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4).

Рабочему нужны данные о положениях, скоростях и массах тел в одном или двух блоках. Управляющий мог бы передавать эти данные вместе с задачей, но можно значительно сократить длину сообщений, если у каждого рабочего процесса будет своя собственная копия данных о всех телах. Аналогично можно избавиться и от необходимости отправлять результаты управляющему, если каждый рабочий процесс отслеживает силы, которые он вычислил для каждого тела. После вычисления всех сил, т.е. когда портфель задач пуст, рабочим процессам нужно обменяться силами и затем переместить тела. Простейшее статическое назначение работы в фазе перемещения тел состоит в том, чтобы каждый рабочий перемещал тела в одном блоке: рабочий процесс 1 перемещает тела в блоке 1, рабочий 2 — в блоке 2 и т. д. Следовательно, каждому рабочему w нужны векторы сил для тел блока w.

*Программа типа "управляющий-рабочие" для задачи n тел:*

```
chan getTask(int worker), task[1:PR](int block1, block2);
chan bodies[1:PR](int worker; point post[*], vel[*]);
chan forces[1:PR](point forcet[*]);
process Manager {
    декларации и инициализация локальных переменных;
    for [time = start to finish by DT] {
        инициализация портфеля задач;
        for [i = 1 to numTasks+PR ] {
            receive getTask(worker);
        выбрать следующую задачу;
        если портфель пуст, сигнализировать (0, 0);
            send task[worker] (block1, block2);
         }
       }
```

```
    }
    process Worker[w = 1 to PR] {
        point p[l:n], v[l:n], f[l:n];  # положения, скорости
        double m[l:n];       # силы и массы тел
        декларации остальных локальных переменных, например tf[ 1: n ],
tp[l:n], tv[l:n];
        инициализация всех локальных переменных;
        for [time = start to finish by DT] {
            while (true) {
                send getTask(w); receive task[w](block1, block2);
                if (block1 == 0) break;    # портфель пуст
                вычислить силы между телами block1 и block2;
              }
              for [i = 1 to PR st i != w]   # обмен силами
                  send forces[i](f[*]);
              for [i = 1 to PR st i != w]
    receive forces[w](tf[*]);
    добавить значения tf к значениям в f;
               }
               обновить p и v в своем блоке тел;
               for [i = 1 to PR st i !=w]    # обмен телами
     send bodies[i](w, p[*], v[*]);
               for [i = 1 to PR st i !=w]
     receive bodies[w](worker, tp[*], tv[*]);
    переместить тела процесса worker из tp и tv в p и v;
               }
               реинициализировать f нулями;
          }
    }
```

Собирать и распределять силы, вычисленные каждым рабочим процессом, мог бы управляющий, но эффективность повысится, если каждый рабочий будет отправлять силы для блока тел непосредственно тому рабочему, который будет перемещать эти тела. Если же доступны глобальные примитивы взаимодействия (как в библиотеке MPI), возможен другой вариант: рабочие используют их для рассылки и удаления (добавления) значений в векторах сил. После того как все рабочие переместят тела в своих блоках, им нужно обменяться новыми положениями и скоростями тел. Для этого можно использовать сообщения “от точки к точке” или глобальное взаимодействие.

В данном коде внешний цикл в каждом процессе выполняется один раз на каждом временном шаге имитации. Внутренний цикл в управляющем процессе выполняется numTasks+PR раз, где numTasks — число задач в портфеле. На последних PR итерациях портфель пуст, и управляющий отправляет пару (0,0) как сигнал о том, что портфель пуст. Получая этот сигнал, каждый из PR процессов выходит из цикла вычисления сил.

### *Программа пульсации*

Аналогом алгоритма с разделяемыми переменными является алгоритм пульсации, использующий передачу сообщений; вычислительные фазы в нем чередуются с барьерами. Программа с разделяемыми переменными имеет соответствующую структуру, поэтому, чтобы использовать передачу сообщений, программу можно изменить: 1) каждому рабочему процессу назначается подмножество тел; 2) каждый рабочий сначала вычисляет силы в своем подмножестве, а затем обменивается силами с другими рабочими; 3) каждый рабочий перемещает свои тела; 4) рабочие обмениваются новыми положениями и скоростями тел. При имитации эти действия повторяются на каждом шаге по времени.

Назначать тела рабочим процессам можно по любой схеме распределения (см. рис.5): по блокам, полосам или обратным полосам. Распределение по полосам или обратным полосам дает гораздо более сбалансированную вычислительную нагрузку, чем по блокам фиксированного размера. Однако каждому рабочему процессу придется иметь свою собственную копию векторов положений, скоростей, сил и масс. Кроме того, после каждой фазы необходим обмен целыми векторами со всеми остальными рабочими. Все это приводит к большому числу длинных сообщений.

Если рабочим назначать блоки тел, то будет использоваться приблизительно вдвое меньше сообщений, причем они будут короче. (Рабочим процессам также нужно меньше локальной памяти.) Чтобы понять, почему так происходит, рассмотрим пример из предыдущего раздела, в котором четыре процесса и десять задач. При назначении по блокам процесс 1 обрабатывает первые четыре задачи: (1, 1), (1, 2), (1, 3) и (1, 4). Он вычисляет все силы, связанные с телами блока 1. Для этого ему необходимы положения и скорости тел из блоков 2, 3 и 4, и в эти же блоки ему нужно возвратить силы. Однако процесс 1 никогда не передает другим процессам положения и скорости своих тел, и ему не нужна информация о силах от любого другого процесса. С другой стороны, процесс 4 отвечает только за задачу (4, 4). Ему не нужны положения или скорости остальных тел, и он не должен отправлять силы остальным рабочим.

Если все блоки одного размера, то рабочая нагрузка оказывается несбалансированной; затраты времени при этом могут даже превысить выигрыш от отправки меньшего числа сообщений. Но нет и причин, по которым все блоки должны быть одного размера, как в

программе с разделяемыми переменными. Более того, тела можно переместить за линейное время, поэтому использование блоков разных размеров приведет к относительно небольшому дисбалансу нагрузки в фазе перемещения тел. Ниже представлена схема кода для программы пульсации, в которой рабочим процессам назначаются блоки разных размеров. Как показано, части кода, связанные с передачей сообщений, не симметричны. Разные рабочие отправляют неодинаковое число сообщений в различные моменты времени. Однако этот недостаток превращен в преимущество с помощью совмещения взаимодействий и вычислений во времени. Каждый рабочий процесс отправляет сообщения и затем до получения и обработки сообщений выполняет локальные вычисления.

*Программа пульсации для задачи n тел*

```
chan bodies[1:PR](int worker; point post*], vel[*]);
 chan forces[1:PR](point force[*]);
process Worker[w = 1 to PR] {
     int blocksize = размер своего блока тел;
     int tempSize = максимальное число других тел в сообщениях;
     point p[l:blockSize], v[1:blocksize], f[1:blocksize];
     point tp[1:tempSize], tv[l:tempSize], tf[1:tempSize];
     double m[l:n];
     декларации остальных локальных переменных;
     инициализация всех локальных переменных;
     for [time = start to finish by DT] {
         # отправить свои тела рабочим с меньшими номерами
         for [i = 1 to w—1]
             send bodies[i](w, p[*], v[*]);
             вычислить силы f для своего блока тел;
             #получить тела от рабочих с большими номерами и отправить им силы
             for [i = w+1 to PR] { #получить тела от других
    receive bodies[w](other, tp[*], tv[*]);
    вычислить силы между своим и другим блоками;
    send forces[other] (tf[*]) ;
    }
    # получить силы от рабочих с меньшими номерами
    for [i = 1 to w—1 ] {
        receive forces[w](tf[*]);
       добавить силы из tf к силам в f;
     }
```

```
    обновить p и v для своих тел;
    реинициалировать f нулями;
}
}
```

### *Программа с конвейером*

Рассмотрим решение задачи n тел с помощью конвейерного алгоритма. Напомним, что в конвейере информация двигается в одном направлении от процесса к процессу. Конвейер бывает открытым, круговым и замкнутым. Здесь нужна информация о телах, циркулирующая между рабочими процессами, поэтому следует использовать или круговой, или замкнутый конвейер. Управляющий процесс не нужен (за исключением, возможно, лишь инициализации рабочих процессов и сбора конечных результатов), поэтому достаточно кругового конвейера.

Рассмотрим идею использования кругового конвейера. Предположим, что каждому рабочему процессу назначен блок тел и каждый рабочий вычисляет силы, действующие только на тела в своем блоке. (Таким образом, сейчас выполняются лишние вычисления; позже мы используем симметрию сил между телами.) Каждому рабочему для вычисления сил, создаваемых “чужими” телами, нужна информация о них. Таким образом, достаточно, чтобы блоки тел циркулировали между рабочими процессами. Это можно сделать следующим образом.

*Отправить p и v своего блока тел следующему рабочему процессу; вычислить силы, действующие между телами своего блока;*

*for [i = 1 to PR-1] {*

*получить p и v блока тел от предыдущего рабочего процесса;*

*отправить этот блок тел следующему рабочему процессу;*

*вычислить силы, с которыми тела нового блока действуют на тела своего блока;*

*}*

*получить обратно свой блок тел; переместить тела ;*

*реинициализировать силы, действующие на свои тела, нулями;*

Каждый рабочий процесс выполняет данный код на каждом временном шаге имитации. (Последняя отправка и получение не обязательны в приведенном коде, однако они понадобятся позже.)

При описанном подходе выполняется вдвое больше вычислений сил, чем необходимо. Следовательно, нам желательно рассматривать каждую пару тел только один раз и пропускать силы, уже вычисленные для какой-либо группы тел. Чтобы сбалансировать вы-

числительную нагрузку, нужно или назначать разные количества тел каждому процессу (как в программе пульсации), или назначать тела по полосам или обратным полосам, как в программе с разделяемыми переменными. Применим одну из схем назначения тел по полосам, которая даст наиболее сбалансированную нагрузку.

Здесь приведен код кругового конвейера, использующий схему присваивания тел по полосам. Каждое сообщение содержит положения, скорости и силы, уже вычисленные для подмножества тел. В коде не показаны все подробности учета, необходимые, чтобы точно отслеживать, какие тела принадлежат каждому подмножеству; однако это можно определить по идентификатору владельца подмножества. Каждое подмножество тел циркулирует по конвейеру, пока не вернется к своему владельцу, который затем переместит эти тела.

*Программа с конвейером для задачи n тел*

```
chan bodies[1:PR](int owner;,point p[*], v[*], f[*]);
process Worker[w = 1 to PR] {
     int owner, setSize = n/PR, next = w%PR + 1;
     point p[l:setSize], v[1:setSize] , f[1:setSize];
     point tp[l:setSize], tv[l:setSize], tf[1:setSize];
     double m[l:n];
     декларации других локальных переменных;
     инициализация своего блока тел и других переменных;
     for[time = start to finish by DT] {
         send bodies[next](w, p[*], v[*], f[*]);
         вычислить силы, действующие между телами своего блока;
          for [i = 1 to PR-1] {
              receive bodies[w] (owner, tp[*], tv[*], tf[*] ) ;
              вычислить силы между своими и новыми телами;
              send bodies[w](owner, tp[*], tv[*], tf[*]);
          }
          # получить свои тела (owner будет равен w)
          receive bodies[w](owner, tp[*], tv[*], tf[*]);
          добавить силы из tf к силам в f;
          обновить p и v для своего подмножества тел;
          реинициализировать силы, действующие на свои тела, нуля-
ми;
       }
}
```

### *Сравнение программ*

Приведенные три программы с передачей сообщений отличаются друг от друга по нескольким параметрам: легкость программирования, сбалансированность нагрузки, число сообщений и объем локальных данных. Легкость программирования субъективна; она зависит от того, насколько программист знаком с каждым из стилей. Тем не менее, если сравнить длину и сложность программ, простейшей окажется программа с конвейером. Она короче других, отчасти потому, что в ней используется наиболее регулярная схема взаимодействия. Программа типа “управляющий-рабочие” также весьма проста, хотя ей нужен дополнительный процесс. Программа пульсации сложнее других (хотя и не намного), поскольку в ней используются блоки тел разных размеров и асимметричная схема взаимодействия.

Рабочая нагрузка у всех трех программ будет достаточно сбалансированной. Программа типа “управляющий-рабочие” динамически распределяет работу, поэтому нагрузка здесь почти сбалансирована, даже если процессоры имеют разные скорости или некоторые из них одновременно выполняют другие программы. При выполнении на специализированной однородной архитектуре у конвейерной программы с назначением по полосам рабочая нагрузка будет сбалансирована достаточно, а с назначением по обратным полосам — почти идеально. Программа пульсации использует блоки разных размеров, поэтому нагрузка окажется не идеально, но все- таки почти сбалансированной; кроме того, асимметричная схема взаимодействия и перекрытие взаимодействия и вычислений во времени отчасти скроет несбалансированность вычисления сил.

Во всех программах на каждом шаге по времени отправляются (и получаются) O(PR2) сообщений. Однако действительные количества сообщений отличаются, как и их размеры (табл. 11.1). Алгоритм пульсации дает наименьшее число сообщений. Конвейер — на PR сообщений больше, но они самые короткие. Больше всего сообщений в программе типа “управляющий-рабочие”, однако здесь обмен значениями между рабочими процессами можно реализовать с помощью операций группового взаимодействия.

*Таблица 2. Сравнение программ с передачей сообщений для задачи n тел*

| Программа/Фаза | Количество | Число тел на сообщение |
|---|---|---|
| Управляющий-рабочие | | |
| Получение задачи | 2 | 2 |
| Обмен силами | PR *(PR-1) | n |
| Обмен телами | PR *(PR-1) | 2*n |
| Пульсация | | |
| Обмен телами | PR *(PR-1) / 2 | 2*n |

| | | |
|---|---|---|
| Обмен силами | PR *(PR-1) / 2 | n |
| Конвейер | | |
| Циркуляция тел | PR * PR | 3*n |

Наконец, программы отличаются объемом локальной памяти каждого рабочего процесса. В программе типа “управляющий-рабочие” каждый рабочий процесс имеет копию данных о всех телах; ему также нужна временная память для сообщений, число которых может достигать 2n. В программе пульсации каждому процессу нужна память для своего подмножества тел и временная память для наибольшего из блоков, которые он может получить от других. В конвейерной программе каждому рабочему процессу нужна память для своего подмножества тел и рабочая память для еще одного подмножества, размер которого — n/PR. Следовательно, конвейерной программе нужно меньше всего памяти на один рабочий процесс.

Подведем итог; для задачи n тел наиболее привлекательной выглядит программа с конвейером. Она сравнительно легко пишется, дает почти сбалансированную нагрузку, требует наименьшего объема локальной памяти и почти минимального числа сообщений. Кроме того, конвейерная программа будет эффективнее работать на некоторых типах коммуникационных сетей, поскольку каждый рабочий процесс взаимодействует только с двумя соседями. Однако различия между программами относительно невелики, поэтому приемлема любая из них. Оставим читателю интересную задачу реализации всех трех программ и экспериментального сравнения их производительности.

## Лекция №4. Параллельное программирование в научных вычислениях: матричные вычисления

Сеточные и точечные вычисления являются фундаментальными в научных вычислениях. Третий основной вид вычислений — матричные. В данном разделе представлено использование матриц для решения систем линейных уравнений. Задачи такого типа составляют основу многих научных и инженерных приложений, а также задач экономического моделирования и многих других. Вначале рассмотрен метод исключений Гаусса. Затем описан более общий метод, который называется LU-разложением, и для него построена последовательная программа. Наконец разработаны параллельные программы для LU-разложения с разделяемыми переменными и с передачей сообщений. В упражнениях представлены другие матричные вычисления, в том числе обращение матриц.

## Метод исключений Гаусса

Рассмотрим систему из трех уравнений с тремя неизвестными

a + b + c = 6

2a - b + c = 3

-a + b - c = -2.

Решение системы линейных уравнений эквивалентно решению матричного уравнения Ax = b, где A — квадратная матрица коэффициентов, b — вектор-столбец правых частей уравнений, а x— вектор-столбец неизвестных. Строка с номером i матрицы A содержит коэффициенты для неизвестных i-го уравнения, а i-й элемент в столбце b— значение правой части i-го уравнения.

Метод исключений Гаусса реализуется серией преобразований матрицы A и вектора b. Матрица A приводится к верхней треугольной матрице, у которой все элементы, расположенные ниже главной диагонали, равны нулю. В нашем примере начальное значение матрицы A таково:

$$\begin{matrix} 1 & 1 & 1 \\ 2 & -1 & 1 \\ -1 & 1 & -1 \end{matrix}$$

Соответствующие значения в b— (6, 3, -2). Прямой ход начинается с левого столбца и преобразует A и b следующим образом. Первый шаг: вычисляем множитель A[2,1]/A[1,1], умножаем на него первую строку матрицы A и первый элемент b; полученные строку и элемент вычитаем из второй строки A и второго элемента b. Второй шаг: вычисляем множитель A [ 3,1 ] / A [ 1,1 ], умножаем на него первую строку матрицы A и первый элемент b, и вычитаем их из третьей строки A и третьего элемента b. После этих двух шагов в первом столбце A будут нули во второй и третьей строках. Последний шаг прямого хода в нашем несложном примере — вычисляем множитель A[3,2]/A[2,2], умножаем на него вторую строку A и второй элемент b и вычитаем их из третьей строки A и третьего элемента b. В итоге матрица A примет вид

$$\begin{matrix} 1 & 1 & 1 \\ 0 & -3 & -1 \\ 0 & 0 & -2/3 \end{matrix}$$

Соответствующие значения для b —(6, -9, -2).

Метод исключений Гаусса можно использовать при решении многих систем n уравнений с n неизвестными.” Однако на каждом шаге прямого хода вычисляются множители вида A[k,i]/A[i,i]. Элемент A[i, i] называется ведущим элементом столбца i. Если он равен нулю, то получится “деление на ноль”. Кроме того, если ведущий элемент слишком мал,

то множитель будет слишком большим. Это может сделать алгоритм численно неустойчивым.

Обе проблемы можно решить, используя метод главных элементов. В каждом столбце i выбирается главный элемент A[k, i], имеющий наибольшее абсолютное значение. Перед следующим шагом исключений выполняется перестановка строк k и i. Ее лучше реализовать с помощью перестановки указателей на строки, а не путем реальных обменов значений их элементов. При этом уравнения должны быть независимыми, т.е. никакое уравнение системы не может быть получено из других. Точнее, A должна быть несингулярной (неособенной) матрицей.

## LU-разложение

Метод исключений Гаусса преобразует уравнение Ax = b в эквивалентное ему уравнение Ux = y, где U— верхняя треугольная матрица. При этом вычисляется последовательность множителей. Вместо того, чтобы отбрасывать их, предположим, что они сохраняются в третьей матрице L. Пусть матрица L — нижняя треугольная матрица, в которой все элементы, расположенные выше главной диагонали, равны нулю. Каждый элемент L [ j , i ] на диагонали и под ней имеет значение вида A [ j , i ] /pivot, где pivot — значение ведущего элемента для столбца i. После заполнения матрицы L произведение матриц L и U будет в точности равно исходной матрице A (если не учитывать возможных ошибок округления). В частности, для нашей системы уравнений получим:

$$\begin{matrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -2/3 & 1 \end{matrix} \times \begin{matrix} 1 & 1 & 1 \\ 0 & -3 & -1 \\ 0 & 0 & -2/3 \end{matrix} = \begin{matrix} 1 & 1 & 1 \\ 2 & -1 & 1 \\ -1 & 1 & -1 \end{matrix}$$

L x U = A

Описанное преобразование матрицы A в треугольные матрицы L и U называется LU-разложением A. Основная причина использования LU-разложения состоит в том, что после того, как вычислены матрицы L и U, уравнение Ax = b легко решить для любых правых частей b. Любую систему уравнений можно записать в виде LUx =b. Данное уравнение представляет две треугольные системы: Ly = b и Ux = y. Для решения первой системы относительно y можно применить прямой ход процедуры Гаусса, а затем для решения второй системы относительно x — обратный ход.

Еще одно свойство LU-разложения состоит в том, что для обеих матриц не нужна дополнительная память. Преобразуя A в L и U, мы можем записывать элементы L и U вместо предыдущих элементов A. Это очевидно для U, поскольку преобразованные значения A — это значения и. Множители в L вычисляются точно так же, как при исключении эле-

ментов A, поэтому их можно записывать там, где находились бы нули в A. Наконец, все диагональные элементы L равны единице, так что хранить их не нужно.

В листинге представлена последовательная программа для LU-разложения матрицы A. Чтобы сделать код более понятным, результаты сохраняются в отдельной матрице LU. При этом сохраняется матрица A, которая может понадобиться в дальнейшем. Во избежание деления на нуль или на малые числа использован метод главных элементов. Индексы ведущих строк хранятся в векторе ps. Вначале ps [ i ] равно i; элементы ps переставляются каждый раз, когда в качестве ведущего выбран недиагональный элемент. Таким образом, ps всегда содержит перестановку номеров от 1 до n. Во время процедуры исключений строки A и LU доступны с помощью ps [ i ], а не i.

*Последовательная программа LU-разложения*

```
double A[l:n,l:n], LU[1:n,1:n];    # предполагается, что A инициализирована
int ps[l:n];       # индексы ведущих строк
double pivot; int pivotRow;  # ведущее значение и строка
double mult; int t;     #временные переменные
#инициализация ps и LU
for [i = 1 to n] {
ps[i] = i ;
for [j = 1 to n]
    LU[i,j] = A[i,j];
}
# исключения Гаусса с помощью ведущих элементов
for [k = 1 to п-1] {     # итерации вниз по главной диагонали
    pivot = abs(LU[ps[к],к]); pivotRow = к;
    for [i = к+1 to n] {     # выбор ведущего элемента в столбце k
        if (abs(LU[ps[i],к]) > pivot) {
          pivot = abs(LU[ps[i],k]); pivotRow = i;
        }
    }
    if (pivotRow != к) {     # перестановка строк с помощью перестановки указателей
      t = ps[k]; ps[k] = ps[pivotRow]; ps[pivotRow] = t;
    }
    pivot = LU[ps[k],k];     # настоящее значение ведущего элемента
    for [i = k+1 to n] {     # для всех строк в подматрице
       mult = LU[ps[i],k]/pivot;   # вычислить множитель
```

```
        LU[ps[i],k] = mult;          # и сохранить его
        for [j = k+1 to n]     # исключение по столбцам
            LU[ps[i],j] = LU[ps[i],j] - mult*LU[ps[k],j] ;
      }
}
```

В следующем листинге представлена последовательная программа для решения уравнения Ax = b по данному LU-разложению матрицы A. В первом цикле уравнение Ly = b решается относительно y; во втором — уравнение Ux = y относительно x. Программа сохраняет промежуточные результаты y в векторе x, поскольку это упрощает фазу обратного хода.

*Решение уравнения Ax=b по данному LU-разложению A*

```
double LU[1:n,1:n]; int ps[1:n];
double sum, x[1:n], b[1:n];
# прямой ход для решения Ly = b с записью y в x
for [i = 1 to n] {
    sum = 0.0;
    for [j = 1 to i—1]
        sum = sum + LU[ps[i],j] * x[j];
     x[i] = b[ps[i]] - sum;
}
# обратный ход для решения Ux = y относительно x
for [i = n to 1 by -1] {
    sum = 0.0;
    for [j = i+1 to n] sum = sum + LU[ps[i],j] * x[j];
  x[i] = (x[i] - sum) / LU[ps[i],i];
}
```

### Программа с разделяемыми переменными

Рассмотрим, как распараллелить эти программы, используя PR процессоров и, соответственно, PR рабочих процессов. Вначале рассмотрим LU-разложение. В нем есть две фазы: инициализация ps и LU, а затем прямой ход исключений Гаусса. В фазе инициализации тела циклов независимы, поэтому их можно разделить между рабочими, используя любую схему распределения, которая каждому рабочему назначает поровну элементов данных.

Внешний цикл (по k) в фазе исключений должен выполняться последовательно каждым рабочим процессом, поскольку LU-разложение происходит итеративно вниз по глав-

ной диагонали и разлагает подматрицу LU[k:n,k:n], Тело внешнего цикла имеет две фазы — выбор ведущего элемента и ведущей строки, затем сокращение строк под ведущей. Ведущий элемент можно выбрать следующими тремя способами.

1. Каждый процесс просматривает все элементы в LU [k:n, k] и выбирает наибольший. Если каждый процесс сохраняет свою собственную копию индексов ведущих элементов ps, то после завершения этой фазы барьер не нужен.
2. Один процесс просматривает все элементы в LU [k:n, k], выбирает наибольший и меняет местами ведущую строку и строку k. Здесь нужна точка барьерной синхронизации.
3. Каждый рабочий процесс проверяет свое подмножество элементов из LU[k:n,k], выбирает наибольший элемент из подмножества и затем согласовывает с другими выбор ведущего элемента. Здесь также нужна точка барьерной синхронизации и в зависимости от того, как она запрограммирована, собственные копии ps.

Для малых значений n более быстрым будет первый подход, поскольку в нем нет барьеров. Для больших значений n более быстрым, вероятно, окажется третий подход. Точка пересечения (графиков сложности) зависит от того, как накладные расходы, связанные с барьером, соотносятся со временем выбора наибольшего элемента.

После выбора ведущего элемента все строки под ведущей строкой можно исключить параллельно. Для каждой строки сначала вычисляется и сохраняется множитель mult, затем выполняются итерации по столбцам, находящимся справа от столбца с ведущим элементом. По мере выполнения LU-разложения подматрица, в которой проводятся исключения, уменьшается, как и объем работы в фазах исключений. Таким образом, LU-матрицу нужно назначить рабочим процессам по полосам или обратным полосам, чтобы у каждого процесса постоянно была какая-то работа, кроме последних нескольких итераций в главном цикле. Вновь используем схему распределения по полосам, поскольку она проще программируется, чем схема с обратными полосами, и приводит к достаточно сбалансированной нагрузке.

Ниже представлен эскиз параллельной программы LU-разложения с разделяемыми переменными. По сравнению с последовательной программой она имеет следующие основные отличия: 1) в фазах инициализации и исключений используются полосы строк; 2) после инициализации, каждой фазы выбора ведущего элемента (если необходимо) и каждого шага исключений установлены барьеры. Кроме того, каждому рабочему, возможно, нужна своя собственная локальная копия ведущих индексов, поскольку это упрощает перестановку строк и не требует синхронизации.

*Структура программы LU-разложения с разделяемыми переменными*

```
   double A[l:n,l:n], LU[1:n,1:n];     # предполагается, что A инициа-
лизирована
    int ps[l:n];      # индексы ведущих строк
   process Worker(w = 1 to PR) {
       double pivot, mult;
       декларации других локальных переменных, например копии ps ;
       for [i = w to n by PR]
          инициализация ps и своих полос LU;
          барьерная синхронизация;
       # исключения Гаусса с ведущими элементами
       for [k = 1 to п-1] {      # итерации вниз по главной диагонали
           поиск наибольшего ведущего элемента — см. текст;
           если необходимо, перестановка ведущей строки со строкой k,
           затем синхронизация;
           pivot = LU[ps[k] ,k];      # вычисление настоящего значения
ведущего элемента
           for [i = к+1 to n st (i%PR- == 0)] { # на своей полосе
              mult = LU[ps[i],к]/pivot;    #вычислить множитель
              LU[ps[i],k] = mult;      # и сохранить его
              for [j = k+1 to n]      # исключения по столбцам
   LU[ps[i],j] =LU[ps[i],j] - mult*LU[ps[k],j];
           }
          барьерная синхронизация;
        }
   }
```

Рассмотрим, как распараллелить прямой и обратный проходы. Один из способов достигнуть параллельности — использовать так называемую синхронизацию фронта волны (wave front synchronization). В фазе прямого хода итерации назначаются рабочим по полосам. Поскольку вычисления x [ i ] зависят от предыдущих значений x [ 1: i—1 ], с каждым элементом x можно связать флаг (или семафор). Закончив вычисление x [ i ], процесс устанавливает флаг для этого элемента. Когда процессу нужно прочитать значение x [ i ], он сначала ждет, пока для этого элемента не будет установлен флаг. Например, код, выполняемый рабочим процессом w в прямом ходе, мог бы быть таким.

```
   for [i = w to n by PR] {      # на своей полосе x[*]
       sum =0.0;      # локальное значение
       for [j = 1 to i — 1 ] {
          ожидать, пока не будет установлен флаг x[j];
```

```
            sum = sum + LU[ps[i],j] * x[j];
        }
        x[i] = b[ps[i]] - sum;
        установить флаг x[i];
    }
```

Фронт волны представляет собой установку флагов по мере вычисления новых элементов. (Термин фронт волны обычно используется для матриц; волна, как правило, представляет собой диагональную линию, движущуюся по матрице.)

Волновые фронты эффективны, если накладные расходы при синхронизации невелики по сравнению с объемом вычислений. Здесь же на каждый элемент приходится очень мало вычислений, поэтому синхронизацию можно запрограммировать с помощью простых флагов и активного ожидания. Это должно дать небольшое увеличение производительности данного приложения.

### Программа с передачей сообщений

Рассмотрим, как реализовать LU-разложение с помощью передачи сообщений. Можно использовать любой из подходов (управляющий-рабочие, алгоритмы пульсации и конвейера), однако если в программе с разделяемыми переменными для некоторого приложения применяются барьеры, то естественнее всего построить распределенную программу на основе алгоритма пульсации. Ниже приведен эскиз программы пульсации для LU-разложения. Как обычно, при создании распределенной программы сначала нужно решить, как распределить данные, чтобы вычислительная нагрузка оказалась сбалансированной. Поскольку LU-разложение работает с уменьшающимися подматрицами, объем работы также уменьшается по мере выполнения исключений. Поэтому можно назначить строки по полосам. Если предположить, что есть PR рабочих процессов, то рабочему процессу назначается каждая PR- я строка LU, по n/PR строк на каждый процесс.

Первый шаг в LU-разложении — инициализация локальных строк LU и индексов ведущих элементов ps. Все процессы могут выполнить этот шаг параллельно. После инициализации барьер не нужен, поскольку здесь нет разделяемых переменных. Главный шаг в LU-разложении — многократное повторение выбора ведущего элемента и ведущей строки с последующим исключением всех строк, расположенных ниже ведущей. Каждый рабочий процесс может выбрать наибольший элемент в столбце k своих n/PR строк в матрице LU. Однако для выбора глобального максимума рабочим нужно взаимодействовать. Можно использовать один процесс в качестве управляющего, который собирает максимальные значения от всех процессов, выбирает наибольшее из них и рассылает его копии. Или же, если доступны такие глобальные примитивы взаимодействия, как в библиотеке

MPI, то для вычисления ведущего значения можно использовать примитив редукции. После выбора ведущего значения процесс, которому принадлежит ведущая строка, должен передать ее другим, поскольку она им нужна в фазе исключения. Получив ведущие значение и строку, каждый рабочий процесс может выполнить исключение строк своей области под ведущей строкой.

В листинге содержится эскиз программы с передачей сообщений для LU- разложения. Все шаги в программе такие, как описано выше. Явные барьеры здесь не нужны, поскольку обмен сообщениями, необходимый при выборе ведущего значения и ведущей строки, по сути, является барьером. В фазе исключений используется переменная myRow, чтобы отображать глобальный индекс i-й строки (который находится в диапазоне от 1 до n) в индекс соответствующей строки в локальном массиве строк.

*Эскиз программы с передачей сообщений для LU-разложения*

```
декларации каналов;
process Worker(w = 1 to PR) {
    double LU[1:n/PR,1:n/PR];      # свои строки LU
    int ps[l:n/PR];     #индексы ведущих строк
    double pivot, mult, pivotRow[n];
    int myRow;
    декларации других локальных переменных;
    инициализация ps и своих строк в LU;
    # исключения Гаусса с главными элементами
    for [k = 1 to n-1] {     # итерации вниз по главной диагонали
        поиск наибольшего ведущего элемента в столбце k своих
строк;
        обмен pivot с другими процессами;
        выбор глобального максимального элемента и обновление ps;
        if (владелец ведущей строки)
            рассылка копий pivotRow другим процессам;
        else
            получение pivotRow;
        # исключение своих строк в LU с помощью pivot и pivotRow
        for [i = к+1 to n st (i%PR == 0)]   { # для своей полосы
           myRow = i/PR;     # преобразовать индекс строки
           mult = LU[ps[myRow],k]/pivot;     # вычислить множитель
           LU[ps[myRow],k] = mult;           # и сохранить его
           for [j = k+1 to n]           # исключение по столбцам
```

```
LU[ps[myRow],j] = LU[ps[myRow],j] -mult * pivotRow[j];
        }
      }
}
```

Когда программа в листинге 11.14 завершается, результаты LU-разложения размещаются в локальных массивах рабочих процессов. Чтобы решить систему уравнений, нужно выполнить как прямой, так и обратный ход, т.е. действия, требующие доступа ко всем элементам LU-разложения. Первый подход состоит в использовании процесса, который собирает все строки LU и затем выполняет код из листинга 11.12. При втором используется круговой конвейер, чтобы реализовать синхронизацию фронта волны с помощью передачи сообщений.

В круговом конвейере первый рабочий процесс вычисляет x[1] и передает его второму. Второй процесс вычисляет x[2] и передает x [2] и x[1] третьему. Последний процесс вычисляет x[PR] и передает его и все предыдущие значения первому. Это продолжается до тех пор, пока не будет вычислен x[n]. Можно использовать такой же конвейер для обратного хода, вычисляя окончательные значения x[n], x[n-1] и так вплоть до x[1] и передавая их по конвейеру.

Конвейер для прямого и обратного хода относительно легко программируется, параллелен по существу и не требует сбора всех элементов LU. Однако ему нужно много сообщений, поэтому вполне вероятно, что он может оказаться менее эффективным, чем алгоритм с одним процессом.

**Лекция №5. Примеры научных высокопроизводительных вычислений**

Рассмотренные в предыдущих разделах алгоритмы можно назвать базовыми или классическими, к ним сводятся задачи из различных областей. В данной главе обзорно представлены примеры использования параллельных вычислений в сравнительно новых научных задачах.

## Использование параллельных вычислений в исследованиях климатических условий и окружающей среды

Одна из важных проблем современной науки состоит в оценке будущих изменений климата и их последствий для окружающей среды. Наиболее перспективным средством получения таких оценок являются математические модели климатической системы, которые включают описание широкого круга физических, химических и биологических процессов, происходящих в атмосфере, гидросфере, криосфере и биосфере. В качестве основного математического аппарата в этих моделях используются уравнения гидромеханики и термодинамики сплошных и пористых сред при определенном уровне упрощающих предположений. По пространственному масштабу множество моделей, описывающих процессы в климатической системе, условно можно разбить на три класса: глобальные, региональные и локальные (вихреразрешающие) модели. Глобальные модели достигли в настоящее время высокого уровня в качестве воспроизведения современного климата. Для сравнения климатических моделей между собой и для определения путей их дальнейшего развития организованы международные проекты.

Реализация климатических моделей на многопроцессорных вычислителях с распределенной памятью основана на геометрической декомпозиции расчетной области. Она заключается в том, что каждому вычислительному узлу ставится в соответствие конкретная подобласть. Полнота данных, используемых в шаблонах численных схем, обеспечивается при помощи обменов граничными значениями с узлами, обрабатывающими соседние подобласти (рис. 6).

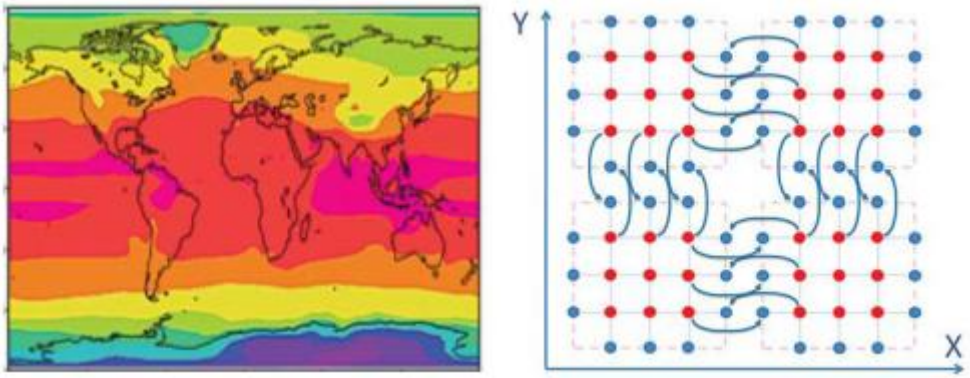


Рисунок 6. Схема обмена данными между соседними подобластями при реализации атмосферных моделей на суперкомпьютерах с распределенной памятью

Использование процедуры декомпозиции позволяет достичь хорошей масштабируемости при различных размерах расчетной области и на большом количестве процессоров. Как правило, возможности обменных сетей традиционных кластерных систем не позволяют эффективно реализовывать параллельные алгоритмы со значительной долей обменов данными. По этой причине перспективность вычислений существенно зависит от возможности выбора численных алгоритмов с меньшей зависимостью по данным и суперкомпьютерных архитектур с большей плотностью вычислительных элементов в узле (многоядерность, сопроцессоры и ускорители, графические процессоры) и быстрой оперативной памятью.

## Использование параллельных вычислений в генетических исследованиях

Лавинообразное получение данных в области биологии и использование современных технологий для их обработки привело к возникновению отдельной научной дисциплины - биоинформатики. Ее целью является как накопление биологических знаний в форме, обеспечивающей их наиболее эффективное использование, так и построение и анализ математических моделей биологических систем и их элементов. Информация о строении материальных элементов, обеспечивающих функционирование организма, хранится в последовательности нуклеотидов ДНК (или РНК), образующей его геном. Установление нуклеотидных последовательностей ДНК геномов организмов (секвенирование) стало к началу 21 века хорошо освоенной технологией. Количество секвенированных геномов быстро увеличивается и определяется, в основном, только объемом средств, которые можно затратить на эти цели. Возникают крупные международные научно-исследовательские проекты, самый известный из которых «Геном человека».

В биоинформатике существует специальный раздел, называемый геномикой, предметом которого является моделирование и исследование способов хранения информации о строении основных материальных элементов биологических систем, закодированной в

последовательностях ДНК и РНК. Накопленная биологическая информация хранится в различных базах данных (UniProtKB, ProSite, OMIA, SIMAP, PDB), также существуют программы и алгоритмы, позволяющие обрабатывать эту информацию (BLAST, FASTA). Однако по-прежнему основные усилия ученых всего мира, работающих в области геномики сосредоточены на том, чтобы выработать эффективные приемы компьютерного анализа генетических "текстов", представляющих собой последовательности нуклеотидов генома клетки. Этот анализ представляет собой совокупность множества биологически важных задач, определенных прежде всего на последовательностях, или строках: воссоздание длинных строк ДНК по перекрывающимся фрагментам строк; определение физических и генетических карт по опытным данным, взятым из протоколов различных экспериментов; хранение, выдача и сравнение строк ДНК; поиск сходства в двух или более строках; поиск родственных строк и подстрок в базах данных; формализация и использование различных понятий родственности строк; поиск новых или плохо определенных образцов (паттернов), часто встречающихся в ДНК; поиск структурных образцов в ДНК и белках; определение вторичной (двумерной) структуры РНК; нахождение сохраненных, но слабовыраженных паттернов во многих последовательностях ДНК и белков и др. Многие из этих задач сводятся к поиску образца в тексте или файле, которая сформулирована следующим образом:

> *Дана последовательность базы данных (или текст)* $T = t_1 t_2 ... t_n$ *длины* $n$ *и конечный набор* $r$ *паттернов* $P = p^1, p^2, ..., p^r$, *где каждый* $p^i$ *представляет собой строку* $p^i = p^i_1 p^i_2 ... p^i_m$ *длины* $m$, *задача состоит в обнаружении всех совпадений паттернов из набора* $P$ *с любыми паттернами в последовательности* $T$.

Однако стоит отметить, что в биологии степень «похожести» будет отличаться от привычного определения. Мы привыкли считать две последовательности тем более похожими, чем больше символов в них совпадает. Однако, в алгоритмах биоинформатики учитываются, например, процессы эволюции. Это нашло выражение в использовании вставок/делеций и замен. Для первых вводят некоторый «штраф», для вторых используют матрицы сравнения, которые описывают вероятность того, что аминокислоты двух данных типов могут взаимозаменяться в процессе эволюции. Всем этим событиям поставлены в соответствие некоторые значения, совокупность которых определяет качество установленного результата.

Итак, для анализа биологических последовательностей используются различные строковые алгоритмы, которые уже можно назвать классическими (Кнута-Морриса-Пратта, Бойера-Мура), а также их модификации (алгоритмы Commentz-Walter, Wu-Manber, Salmela-Tarhio-Kytöjoki). Ознакомиться с ними предлагается самостоятельно. Однако объ-

ем обрабатываемых данных настолько велик (и продолжает расти), что даже самый эффективный алгоритм не смог бы справиться с подобной задачей за разумное время без использования технологий параллельного программирования.

Чтобы добиться максимальной эффективности, используются гибридные технологии распараллеливания, которые объединяют преимущества систем и с разделяемой, и с распределенной памятью. Таким образом, используемая вычислительная система состоит из множества взаимодействующих многоядерных компьютеров. На первом уровне параллелизм осуществляется на многоядерных компьютерах с использованием MPI, где каждый узел ответственен за один процесс MPI. На следующем уровне каждый процесс MPI распараллеливается среди процессоров одного компьютера с использованием директив OpenMP; каждому потоку (нити) OpenMP назначается процессор (ядро). Ниже представлен псевдокод параллельной программы на основе рассмотренной гибридной модели.

```
Main procedure
main()
{
1. Инициализация процедур MPI и OpenMP;
2. If (process==master) then call master(); else call worker();//вызов соответствующей процедуры
3. Выходное сообщение о выполненных операциях;
}
Процедура «мастер»
master()
{
1. Транслировать имя набора образцов и текст «рабочим»; (MPI_Bcast)
2. Транслировать смещение текста, размера блока и числа нитей (потоков) «рабочим»; (MPI_Bcast)
3. Получить результаты (т.е. совпадения) от всех «рабочих»; (MPI_Reduce)
4. Печать общих результатов;
}
Процедура «рабочий»
worker()
{
1. Получить имя набора образцов и текст; (MPI_Bcast)
2. Предварительно обработать набор образцов;
```

3. *Получить смещение текста, размера блока и количество потоков; (MPI_Bcast)*

4. *Открыть набор образцов и текстовых файлов на локальном диске и сохранить локальный подтекст (из текст + смещение в текст + смещение + размер блока) в памяти;*

5. *Вызвать выбранный алгоритм множественного сравнения образцов для указателя на подтекст в памяти;*

*6. Разделить подтекст между доступными потоками (#pragma omp parallel for);*

7. *Определить число совпадений в каждом потоке(reduction(+: matches));*

*8. Передать результаты (т.е. совпадения) «мастеру»;*

}

## Контрольные вопросы

1. В чем заключается различие параллельного и распределенного программирования?

2. Укажите способы классификации и соответствующие им типы параллельных вычислительных систем.

3. Назовите основные способы взаимодействия процессов.

4. Какие программные и технические средства параллельного программирования вам известны? Расскажите подробнее о некоторых из них.

5. Приведите примеры задач, в которых могут использоваться сеточные, точечные, матричные вычисления в совокупности с методами параллельного программирования.

6. Какие трудности возможны при реализации алгоритма программы с разделяемыми переменными и алгоритма программы с передачей сообщений?

7. Как может быть организовано взаимодействие процессов в программе с передачей сообщений (алгоритм пульсаций, конвейерный алгоритм, “управляющий-рабочие”)?

8. Расскажите об одном из известных научных проектов, связанных с суперкомпьютерными вычислениями.

# Литература

1. «Основы многопоточного, параллельного и распределенного программирования», Грегори Р.Эндрюс, Издательский дом «Вильямс», 2003

2. «Параллельное и распределенное программирование с использованием C++», Кэмерон Хьюз, Трэйси Хьюз, Издательский дом «Вильямс», 2004

3. «Параллельное программирование для многопроцессорных вычислительных систем», Сергей Немнюгин, Ольга Стесик, Издательство «БХВ-Петербург», 2002

4. «Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология», Дэн Гасфилд, Издательство «БХВ-Петербург», 2003

5. Parallel Processing of Multiple Pattern Matching Algorithms for Biological Sequences: Methods and Performance Results, Charalampos S. Kouzinopoulos, Panagiotis D. Michailidis, Konstantinos G. Margaritis

6. http://www.intuit.ru

7. https://software.intel.com