1. Історія та сучасний стан паралельних систем.

У 1962 році Е.В.Евреіновим спільно з Ю.Г.Косаревим запропонована модель колективних обчислювачів і обгрунтована можливість побудови суперкомп'ютерів на принципах паралельного виконання операцій. У 1977 році була розроблена розподілена обчислювальна система з трьох ЕОМ "Мінськ-32" з оригінальним апаратним і програмним забезпеченням, що підтримує протоколи фізичного, канального і мережевого рівнів, і забезпечує виконання паралельних задач.
З іншого боку, в паралельному програмуванні одночасно використовують кілька обчислювальних елементів для розв'язання однієї задачі. Це уможливлюється розбиттям задачі на підзадачі, кожна з яких може бути вирішена незалежно.
Відповідно, сьогодні паралельні системи використовуються в:
-Прочность
-Аэродинамика
-Тепломассообмен
-Атомная энергетика
-Прогноз погоды
-Взлом зашифрованных данных

Актуальность параллельных вычислений
•Экономит время или деньги
• Дают возможность решения больших задач
•Предоставляют возможность использования удаленных ресурсов

2. Закон Мура.

Г.Мур (засновник Intel) на основі розвитку технології компанії Intel в 1965 році висунув наступне твердження:
Кожні 2 роки кількість транзисторів на кристалі подвоюється.
Другий закон Мура трактує, що вартість фабрик, необхідних для виробництва МікроПроцесорів , подвоюється з кожним їх поколінням (приблизно 4 роки).

3. Класифікація паралельних комп'ютерів.

Паралельні комп'ютери можуть бути грубо класифіковані згідно з рівнем, на якому апаратне забезпечення підтримує паралелізм:
багатоядерність, багатопроцесорність — комп'ютери, що мають багато обчислювальних елементів в межах одної машини, а також кластери(це група слабо зв'язаних комп'ютерів, що тісно співпрацюють),
MPP(Масивно паралельний процесор - це один комп'ютер з багатьма процесорами з'єднаними в мережу.), та grid — системи що використовують багато комп'ютерів для роботи над одним завданням. Також існують спеціальні паралельні комп'ютери.

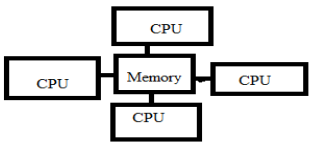
4. Класифікація Фліна.

Эта классификация охватывает только два классификационных признака – тип потока команд и тип потока данных.
Вычислительная система SISD: тільки 1-н потік команд і тільки один потік даних. Всі команди обробляють послідовно одна за одною. Кожна команда ініціює тільки одну скалярну операцію.
MISD. Це визначення обумовлює наявність в архітектурі комп'ютера багатьох процесорів, які здійснюють операції обчислення над одним і тим самим потоком даних. Не створ
Вычислительная система SIMD содержит много процессоров, которые синхронно (как правило) выполняют одну и ту же команду над разными данными.
Системы SIMD делятся на два больших класса:
векторно-конвейерные вычислительные системы;
векторно-параллельные вычислительные системы.
Single instruction Multiple Data – множинний одиночний потік команд – множинний потік даних.
Вычислительная система MIMD содержит много процессоров, которые (как правило, асинхронно) выполняют разные команды над разными данными. Подавляющее большинство современных суперЭВМ имеют архитектуру MIMD

5. Комп'ютери зі спільною та розподіленою пам'яттю.

Shared Memory Systems

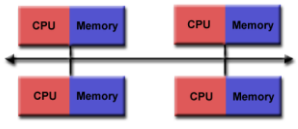
В обчислювальниз системах з загальною пам'ятю значення, записане в пам'ять одним з процесорів напряму доступно для



іншого процесора.

Distributed Memory Systems

В обчислювальних системах з розподіленою пам'ятю кожний процесор має свою локальну пам'ять з локальним адресним простором. Для систем з розподіленою пам'ятю характерно наявність великого числа швидких каналів, які зв'язані окремими частинами цієї памяті з окремими процесорами



6. ГРІД-технології.

С позиций стандартизации (архитектура, протоколы, интерфейсы, сервисы) грид-технологии можно охарактеризовать следующим набором критериев:

- координация использования ресурсов при отсутствии централизованного управления этими ресурсами;
- использование стандартных, открытых, универсальных протоколов и интерфейсов;
- обеспечение высококачественного обслуживания пользователей.

Грид-технологии обеспечивают гибкий, безопасный и координированный общий доступ к ресурсам, под которыми понимаются не только процессорные ресурсы или ресурсы хранения информации, но и сетевые ресурсы, а также системное или прикладное программное обеспечение.

7. Аналіз ефективності паралельних програм.

Ефективність виконання паралельних програм на багатопроцесорних ЕОМ з розподіленою пам'яттю визначається наступними основними факторами:

- ступенем розпаралелювання програми - часткою паралельних обчислень в загальному обсязі обчислень;
- рівномірністю завантаження процесорів під час виконання паралельних обчислень;
- часом, необхідним для виконання міжпроцесорних обмінів;
- ступенем суміщення міжпроцесорних обмінів з обчисленнями;
- ефективністю виконання обчислень на кожному процесорі (а вона може змінюватись значно залежно від ступеня використання кеша).

Эффективность параллельного алгоритма определяется величиной

$$E_N = \frac{S_N}{N},$$

- где S_N - ускорение параллельного алгоритма, N - количество процессоров в системе.
- Из того факта, что $S_N <= N$, следует ограничение сверху на величину эффективности: $E_N <= 1$.

8. Показники якості паралельної програми.

Для оценки эффективности параллельных алгоритмов в основном используют следующие величины:

- *средняя степень параллелизма*;
- *ускорение параллельного алгоритма*;
- *эффективность параллельного алгоритма*.

Средней степенью параллелизма численного алгоритма называется отношение общего числа операций алгоритма к числу его этапов.

При оценке эффективности параллельного алгоритма широко используется модель *абстрактной параллельной ЭВМ с общей памятью* – PRAM (Parallel Random Access Machine). Полагается, что все N процессоров PRAM идентичны. Имеется три типа PRAM, отличающихся моделью того, что происходит при одновременном обращении нескольких процессоров к одной ячейке памяти:
•модель *EREW PRAM* – одновременная запись и чтение из одной ячейки запрещены;
•модель *CREW PRAM* – разрешается одновременное чтение из одной ячейки памяти, но не разрешается одновременная запись;
•модель *CRCW PRAM* – разрешается как одновременное чтение из одной ячейки памяти, так и одновременная запись.
Задача принадлежит *классу задач NC*, если существуют такие константы α, β, и такой алгоритм ее решения, что на PRAM с O(nα) процессорами задача может быть решена за время O(logβn).
Каждая следующая из рассмотренных моделей PRAM может выполнить любой шаг предыдущей модели за конечное число шагов. Доказано также, что каждый шаг наиболее сильной модели CRCW PRAM с N процессорами может быть выполнен наиболее слабой моделью за O log(N) шагов.
Класс *NC* не зависит от модели PRAM

9. Прискорення. Перспективи та обмеження паралелізму.

Ускорение параллельного алгоритма является его наиболее информативной характеристикой, которая показывает во сколько раз применение параллельного алгоритма уменьшает время решения задачи по сравнению с последовательным алгоритмом. Ускорение параллельного алгоритма определяется

$$S_N = \frac{T_1}{T_N},$$

величиной
где T_1- время выполнения алгоритма на одном процессоре, T_N- время выполнения того же алгоритма на процессорах.
Потенційне прискорення алгоритму при збільшенні числа процесорів задається законом Амдала, що вперше був сформульований Жене Амдалем у 1960тих. Він стверджує, що невелика частина програми що не піддається паралелізації обмежить загальне прискорення від розпаралелювання.

10. Поняття послідовної частини, комунікаційних і накладних витрат.

Накладні витрати на паралелізм:$T(m, p) = pT_p - T$. (1.13)
Величина загальних накладних витрат включає сумарні витрати всіх процесорів паралельної системи: на реалізацію обмінів, послідовну частину розпаралеленого алгоритму, непродуктивні витрати на синхронізацію та час простою через незбалансованість завантаження процесорів.
Під час роботи паралельної програми може виникнути конфлікт між збалансованим розподілом об'єктів по процесорах і низькою швидкістю обміну даними між цими процесорами. Деякі процесори можуть простоювати, тоді як інші будуть перевантажені, якщо комунікація між процесорами ведеться на низькій швидкості. З іншого боку, витрати на комунікацію можуть бути великими для збалансованої системи. Саме тому метод балансування повинен бути підібраний таким чином, щоб обчислювальні вузли були завантажені рівномірно, а швидкість обміну даними між процесорами була оптимальною.
Комунікація – з'єднання, передача, повідомлення.

11. Масштабованість паралельної системи

Масштабируемость - способность системы, сети или процесса справляться с увеличением рабочей нагрузки при добавлении ресурсов. Система называется масштабируемой, если она способна увеличивать производительность пропорционально дополнительным ресурсам. Масштабируемость можно оценить через отношение прироста производительности системы к приросту используемых ресурсов. Чем ближе это отношение к единице, тем лучше. Также под масштабируемостью понимается возможность наращивания дополнительных ресурсов без структурных изменений центрального узла системы.

В системе с плохой масштабируемостью добавление ресурсов приводит лишь к незначительному повышению производительности, а с некоторого «порогового» момента добавление ресурсов не даёт никакого полезного эффекта.

Вертикальное масштабирование — увеличение производительности каждого компонента системы с целью повышения общей производительности.

Горизонтальное масштабирование — разбиение системы на более мелкие структурные компоненты и разнесение их по отдельным физическим машинам (или их группам), и (или) увеличение количества серверов, параллельно выполняющих одну и ту же функцию.

13Базові засоби мови програмування для підтримки багатонитковості.

Сучасні мови програмування, які підтримують багатопотоковість, містять стандартні бібліотеки для роботи з потоками. В ці бібліотеки включено набір функцій для створення, запуску, призупинення, знищення, тощо.

Підтримка багатонитковості.Важливою властивістю ОС є можливість розпаралелювання обчислень у межах одного завдання.Багатонитковa ОС розподіляє процесорний час не між завдання-ми, а і між їхніми окремими галузями (нитками).

Багатопроцесорне оброблення.Загальноприйнятим є введення в ОС функцій підтримки багатопроцесорного оброблення даних. Такі функції є в OCSolaris 2.x фірми Sun, Open Server 3.x компанії Santa Crus Operations, OS/2 фірми IBM, Windows NT фірми Microsoft і Netware 4.1 фірми Novell.

Багатопроцесорні ОС можна класифікувати за способом організації обчислювального процесу в системі з багатопроцесорною архітектурою: асиметричні ОС і симетричні ОС.

12. Поняття процесу та потоку.

Потоки и процессы представляют из себя последовательность инструкций, которые должны выполняться в определенном порядке. Инструкции в отдельных потоках или процессах, однако, могут выполняться параллельно. Процессы существуют в операционной системе и соответствуют тому, что пользователи видят, как программы или приложения.

Процесс - совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.

Поток выполнения (от англ. thread — нить) — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

Процессы – выполнение нескольких задач. Потоки – разделение работы процесса.

Каждый поток имеет собственный счетчик команд и стек.

ОС каждому процессу создаёт не менее одного потока. При создании потока или процесса, операционная система генерирует описатель потока (содержит идентификатор потока, данные о правах доступа и приоритете, о состоянии потока и прочее). В исходном состоянии поток находится в приостановленном состоянии. Момент выборки потока на выполнение осуществляется в соответствии с принятым в данной системе правилом предоставления процессорного времени и с учетом всех существующих в данный момент потоков и процессов.

13. Базові засоби мови програмування для підтримки багатонитковості.

Сучасні мови програмування, які підтримують багатопотоковість, містять стандартні бібліотеки для роботи з потоками. В ці бібліотеки включено набір функцій для створення, запуску, призупинення, знищення, тощо.

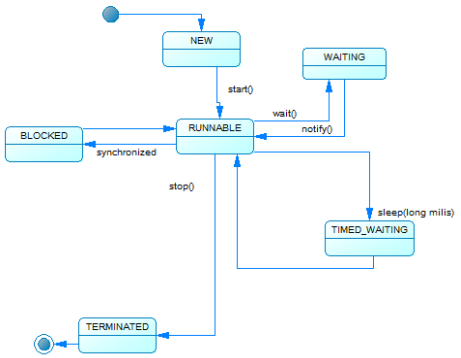
Підтримка багатониткової.Важливою властивістю ОС є можливість розпаралелювання обчислень у межах одного завдання.Багатониткова ОС розподіляє процесорний час не між завданнями, а і між їхніми окремими галузями (нитками).

Багатопроцесорне оброблення.Загальноприйнятим є введення в ОС функцій підтримки багатопроцесорного оброблення даних. Такі функції є в ОСSolaris 2.x фірми Sun, Open Server 3.x компанії Santa Crus Operations, OS/2 фірми IBM, Windows NT фірми Microsoft і Netware 4.1 фірми Novell.

Багатопроцесорні ОС можна класифікувати за способом організації обчислювального процесу в системі з багатопроцесорною архітектурою: асиметричні ОС і симетричні ОС.

14. Створення потоку. Вихід із потоку. Синхронізація при завершенні.

Существуют два способа создания и запуска потока: расширение класса **Thread** или реализация интерфейса **Runnable**. Переопределяется метод run



При выполнении программы объект класса Thread может быть в одном из четырех основных состояний: “новый”, “работоспособный”, “неработоспособный” и “пассивный”. При создании потока он получает состояние “новый” (NEW) и не выполняется. Для перевода потока из состояния “новый” в состояние “работоспособный” (RUNNABLE) следует выполнить метод start(), который вызывает метод run() – основной метод потока.

Получить значение состояния потока можно вызовом метода getState().

При роботі потоку ідентифікатор потоку залишається в несигнальному стані, після завершення роботи потоку, ідентифікатор переходить в сигнальний стан. Це дозволяє дізнатись чи завершився потік, або організувати очікування на завершення роботи кількох потоків.

для завершения потока необходимо освободить все занимаемые потоком ресурсы и выйти из метода run();
•для завершения другого потока его необходимо нотифицировать, вызвав метод interrupt(); при этом внутри прерываемого потока должна быть проверка запроса на прерывание (внутри цикла вызывать метод interrupted() или isInterrupted()), кроме того должно обрабатываться исключение InterruptedException, которое возникает, когда происходит попытка прервать поток, находящийся в состоянии ожидания.

Більше дивись тут

15. Синхронізація потоків.

Потоки иногда должны синхронизироваться. Один поток может ожидать результата другого потока, или одному потоку может понадобиться монопольный доступ к ресурсу, который используется другим потоком.

Все потоки одного процесса используют общие файлы, таймеры, устройства, одну и ту же область оперативной памяти, одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные.

Виртуальное адресное пространство процесса — это совокупность адресов, которыми может манипулировать программный модуль процесса. Операционная система отображает виртуальное адресное пространство процесса на отведенную процессу физическую память.

Один поток может использовать стек другого потока. Между потоками одного процесса нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Чтобы организовать взаимодействие и обмен данными, потокам вовсе не требуется обращаться к ОС, им достаточно использовать общую память — один поток записывает данные, а другой читает их. С другой стороны, потоки разных процессов по-прежнему хорошо защищены друг от друга.

Операционная система обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса.

Если метод отмечен ключевым словом **synchronized,** то его или другие **synchronized-методы** для данного объекта может одновременно выполнить только один поток.

Если переменная объявлена как **volatile**, это означает, что ожидается её изменение несколькими потоками. JRE неявно обеспечивает синхронизацию при доступе к volatile переменным, но с одной очень большой оговоркой: чтение volatile переменных синхронизировано и запись в volatile переменные синхронизирована, а неатомарные операции – нет.

16. **Атомарні операції.**

Атомарна операція в програмуванні означає набір інструкцій з властивістю неперервності цілої операції. Атомарна операція виконується повністю (або відбувається відмова у виконанні), без переривань. Атомарність має особливе значення в багатопроцесорних комп'ютерах і багатозадачних операційних системах, оскільки доступ до ресурсів, що не розподіляються, повинен бути обов'язково атомарним.

Атомарна операція відкрита впливу тільки одної ниті.

Атомарність буває апаратна (коли безперервність забезпечується апаратурою) і програмною, коли використовуються спеціальні засоби міжпрограмної взаємодії (м'ютекс, семафор). За своєю суттю програмні засоби забезпечення атомарності складаються з двох етапів — блокування ресурсу і виконання самої операції. Блокування є атомарною операцією, яка або успішна, або повертає повідомлення про зайнятість.

Набір дій може вважатися атомним, коли виконуються дві умови:

1) Поки повний набір дій не завершується, ніякий інший процес не може знати про зроблені зміни (невидимість); і
2) Якщо будь-яка з дій не виконалася, тоді не виконується повний набір дій, і стан системи відновлюється до того стану, в якій це знаходилося перед тим, як будь-яка з дій почалася.

17. **Монітори.**

Мониторы представляют собой программные модули (объекты), которые реализуют (инкапсулируют) все необходимые действия с разделяемым ресурсом. Общий формат определения монитора может быть представлен следующим образом:

Monitor <Name> {
<объявления переменных>
<операторы инициализации>
<процедуры монитора>**}**

Как можно заметить, описание монитора достаточно близко совпадает с описанием класса в алгоритмическом языке С++. Принципиальное отличие состоит в том, что процедуры монитора, в обязательном порядке, выполняются в режиме взаимоисключения, т. е. при выполнении какой-либо процедуры монитора все остальные попытки вызова других процедур этого же монитора блокируются. Обеспечение такого правила выполнения процедур монитора должно осуществляться средой выполнения, в которой поддерживается концепция мониторов.

Помимо взаимоисключения процедур, другим важным свойством понятия монитора является его полная изолированность от остального кода программы:

Переменные монитора недоступны вне монитора и могут обрабатываться только процедурами монитора.

Вне монитора доступны только процедуры монитора.

Переменные, объявленные вне монитора, недоступны внутри монитора.

Подобная локализация (инкапсуляция) и объединение в рамках монитора всех критических секций потоков приводит к значительному снижению сложности логики параллельного выполнения.

18. **Synchronized** – методи та інструкція synchronized.

Очень часто возникает ситуация, когда несколько потоков, обращающихся к некоторому общему ресурсу, начинают мешать друг другу; более того, они могут повредить этот общий ресурс. Например, когда два потока записывают информацию в файл/объект/поток. Для предотвращения такой ситуации может использоваться ключевое слово synchronized.

Синхронизации не требуют только атомарные процессы по записи/чтению, не превышающие по объему 32 бит.

Если метод отмечен ключевым словом synchronized, то его или другие synchronized-методы для данного объекта может одновременно выполнить только один поток.

Синхронизировать объект можно не только при помощи методов с соответствующим модификатором, но и при помощи синхронизированного блока кода. В этом случае происходит блокировка объекта, указанного в инструкции synchronized, и он становится недоступным для других синхронизированных методов и блоков. Обычные методы на синхронизацию внимания не обращают, поэтому ответственность за грамотную блокировку объектов ложится на программиста.

21. Критичні секції, замки (lock).

Критическая секция — 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:

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

Критическая секция (англ. *critical section*) — 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.

Переменная-замок

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в *критическую секцию* только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из *критической секции* процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе " *Критическая секция* ").

```
shared int lock = 0;
/* shared означает, что */
/* переменная является разделяемой */

while (some condition) {
  while(lock); lock = 1;
    critical section
  lock = 0;
    remainder section
}
```

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию *взаимоисключения*, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P_0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессуP_1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в *критический участок*. Мы получаем два процесса, одновременно выполняющих свои *критические секции*.

s t d : : lock, функция которая умеет захватывать сразу два и более мьютексов без риска получить взаимоблокировку.

20 Методи синхронізації wait, notify, notifyAll.

Методы wait() и notify() используются при освобождении и возврате блокировки в synchronized блоке.

Эти методы используются для управления потоками в ситуации, когда необходимо задать определенную последовательность действий без повторного запуска потоков.

- Метод wait(), вызванный внутри синхронизированного блока или метода, останавливает выполнение текущего потока и освобождает от блокировки захваченный объект. Возвратить блокировку объекта потоку можно вызовом метода notify() для конкретного потока или notifyAll() для всех потоков. Вызов может быть осуществлен только из другого потока, заблокировавшего, в свою очередь, указанный объект.

19. Реалізація моніторів у Java.

Как известно, каждый объект в java имеет свой монитор, и потому, в отличие от того же C++, нет необходимости guard-ить доступ к объектам отдельными mutex-ами. Для достижения эффектов взаимного исключения и синхронизации потоков используют следующие операции:

1) monitorenter: захват монитора. В один момент времени монитором может владеть лишь один поток. Если на момент попытки захвата монитор занят, поток, пытающийся его захватить, будет ждать до тех пор, пока он не освободится. При этом, потоков в очереди может быть несколько.
2) monitorexit: освобождение монитора
3) wait: перемещение текущего потока в так называемый wait set монитора и ожидание того, как произойдёт notify. Выход из метода wait может оказаться и ложным. После того, как поток, владеющий монитором, сделал wait, монитором может завладеть любой другой поток.
4) notify(all): пробуждается один (или все) потоки, которые сейчас находятся в wait set монитора. Чтобы получить управление, пробуждённый поток должен успешно захватить монитор (monitorenter)

От того, есть ли contention на владение монитором, очень сильно зависит то, как производится его захват. Монитор может находиться в следующих состояниях:

- **init**: монитор только что создан, и пока никем не был захвачен
- **biased**: (умная оптимизация, появившаяся далеко не сразу) Монитор «зарезервирован» под первый поток, который его захватил. В дальнейшем для захвата этому потоку не нужны дорогие операции, и захват происходит очень быстро. Когда захват пытается произвести другой поток, либо монитор пере-резервируется для него (**rebias**), либо монитор переходит в состояние thin (**revoke bias**). Также есть дополнительные оптимизации, которые действуют сразу на все экземпляры класса объекта, монитор которого пытаются захватить (**bulk revoke/rebias**)
- **thin**: монитор пытаются захватить несколько потоков, но contention нет (т.е. они захватывают его не одновременно, либо с очень маленьким нахлёстом). В таком случае захват выполняется с помощью сравнительно дешёвых CAS. Если возникает contention, то монитор переходит в состояние inflated
- **fat/inflated**: синхронизация производится на уровне операционной системы. Поток паркуется и спит до тех пор, пока не настанет его очередь захватить монитор. Даже если забыть про стоимость смены контекста, то когда поток получит управление, зависит ещё и от системного шедулера, и потому времени может пройти существенно больше, чем хотелось бы. При исчезновении contention монитор может вернуться в состояние thin

22. Семафори.

Семафо́р (англ. *semaphore*) — объект, ограничивающий количество потоков, которые могут войти в заданный участок кода. Определение введено Эдсгером Дейкстрой. Семафоры используются для синхронизации и защиты передачи данных через разделяемую память, а также для синхронизации работы процессов и потоков.

Типы семафоров

В зависимости от значений, которые может принимать семафор он делится на:

- Двоичный : принимает значения 0 и 1.
- Троичный : принимает значения 0,1 и 2 и тд

Некоторые из проблем, которые могут решать семафоры:

- запрет одновременного выполнения заданных участков кода (критические секции);
- поочерёдный доступ к критическому ресурсу (важному ресурсу, для которого невозможен (или нежелателен) одновременный доступ);
- синхронизация процессов и потоков (например, можно инициировать обработку события отпусканием семафора).

Проблемы семафоров

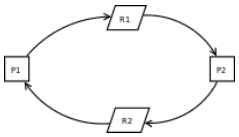
Во-первых, можно написать программу с «утечкой семафора», вызвав `enter()` и забыв вызвать `leave()`. Реже встречаются ошибки, когда дважды вызывается `leave()`.

Во-вторых, семафоры чреваты взаимной блокировкой потоков.

Предложены Дейкстрой еще в середине 1960-х годов.
Под семафором S понимается переменная особого типа, значение которой может опрашиваться и изменяться только при помощи специальных операций P(S) и V(S), реализуемых в соответствии со следующими алгоритмами:
операция P(S)
если S > 0
то S = S – 1
иначе < ожидать S >
операция V(S)
если < один или несколько потоков ожидают S >
то < снять ожидание у одного из ожидающих потоков >
иначе S = S + 1
Принципиальным в понимании семафоров является то, что операции P(S) и V(S) предполагаются неделимыми (атомарными), что гарантирует взаимоисключение при использовании общих семафоров. Семафоры широко используются для синхронизации и взаимоисключения потоков. Так, например, проблема взаимоисключения при помощи семафоров может иметь следующее простое решение.

23. Дедлоки, умови виникнення та алгоритми запобігання.

Взаи́мная блокиро́вка (англ. *deadlock*) — ситуация в многозадачной среде или СУБД, при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими процессами.



Взаимная блокировка двух процессов P1 и P2 нуждающихся в двух ресурсах.

Простейший пример взаимной блокировки

Livelock

Это слово означает такую ситуацию: система не «застревает» (как в обычной взаимной блокировке), а занимается бесполезной работой, её состояние постоянно меняется — но, тем не менее, она «зациклилась», не производит никакой полезной работы.
Жизненный пример такой ситуации: двое встречаются лицом к лицу. Каждый из них пытается посторониться, но они не расходятся, а несколько секунд сдвигаются в одну и ту же сторону.

Алгоритмы и методы предотвращения взаимной блокировки

1)Алгоритм Банкира— это алгоритм распределения ресурсов (англ.) и обхода взаимоблокировок, Він досліджує можливий розвиток подій шляхом відтворення розподілу заздалегідь визначеної кількості ресурсів, і тоді робить перевірку на безпечність стану з метою дослідження на можливі умови взаємних блокувань для всіх інших очікуючих активностей, перед прийняттям рішення чи можна дозволити подальший розподіл.

Алгоритм

Алгоритм банкіра виконується операційною системою щоразу, коли процес запитує ресурси.[2] Алгоритм запобігає взаємним блокуванням шляхом відмови або відкладання запитів, якщо він визначає, що виконання цих запитів переведе систему до небезпечного стану (у якому можливе взаємне блокування). Коли в системі з'являється новий процес, він має заявити максимально потрібну кількість ресурсів кожного типу, але не більше, ніж загальна кількість ресурсів у системі. Також, коли процес отримує в своє розпорядження ресурси, він має повернути їх системі за скінченний проміжок часу.

2)Предотвращение рекурсивных блокировок. Это предотвращает поток от входа в одну и ту же блокировку несколько раз.

Предотвращение взаимной блокировки

Классический способ борьбы с проблемой — разработка иерархии блокировок, установление правила, что некоторые блокировки никогда не могут захватываться в состоянии, в котором уже захвачены какие-то другие блокировки. Говоря точно, речь о разработке отношения сравнения между блокировками, и о запрете захвата «большей» блокировки в состоянии, когда уже захвачена «меньшая».

24. Особливості паралельного середовища OpenMP.

25. Базові поняття OpenMP: паралельні блоки, патерн SPMD, паралельні цикли, редукція.

В большинстве MPI-программ используется шаблон "Одна программа, разные данные" (Single Program Multiple Data, или SPMD) [mattson05]. 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".
Значительная часть параллельного программирования состоит именно в том, чтобы поручить всем потокам выполнение одних и тех же инструкций. Но чтобы использовать OpenMP в полной мере, потребуется что-то большее. Необходимо разделить между потоками работу по выполнению набора инструкций. Такой тип поведения называется "совместное выполнение работы". Самая типичная конструкция для совместной работы - это конструкция цикла (в C это цикл for).

```
#pragma omp for
```

Это работает только для простых циклов стандартного вида:

```
for(i=lower_limit; i<upper_limit; inc_exp)
```

Конструкция for распределяет итерации цикла между потоками группы, созданными ранее с помощью конструкции parallel. Начальное и конечное значения счетчика цикла, а также выражение для шага счетчика (inc_exp) должны быть полностью определены во время компиляции, а все константы, которые используются в этих выражениях, должны быть одинаковы для всех потоков группы. Если вдуматься, это не лишено смысла. Система должна вычислить, сколько итераций цикла должно быть выполнено, чтобы разделить их на наборы, которые будут обрабатывать группы потоков. Это можно сделать только согласованно и точно, если все потоки используют одни и те же наборы счетчиков. 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.
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.

26. Базові поняття OpenMP: секції, бар'єри, приватні та спільні змінні, статичне та динамічне розбиття циклів.

Программа состоит из последовательных и параллельных секций.
•В начальный момент времени создается главная нить, выполняющая последовательные секции программы.
•При входе в параллельную секцию выполняется операция *fork*, порождающая семейство нитей. Каждая нить имеет свой уникальный числовой идентификатор (главной нити соответствует 0). При распараллеливании циклов все параллельные нити исполняют один код. В общем случае нити могут исполнять различные фрагменты кода.
•При выходе из параллельной секции выполняется операция *join*. Завершается выполнение всех нитей, кроме главной.

Директивы OpenMP

#pragma omp задает границы параллельной секции программы. С данной директивой могут использоваться следующие операторы:

• private;
• shared;
• default;
• firstprivate;
• reduction;
• if;
• copyin;
• num_threads.

Директивы pragma для синхронизации

При одновременном выполнении нескольких потоков часто возникает необходимость их синхронизации. OpenMP поддерживает несколько типов синхронизации, помогающих во многих ситуациях.
Один из типов — неявная барьерная синхронизация, которая выполняется в конце каждого параллельного региона для всех сопоставленных с ним потоков. Механизм барьерной синхронизации таков, что, пока все потоки не достигнут конца параллельного региона, ни один поток не сможет перейти его границу.
Неявная барьерная синхронизация выполняется также в конце каждого блока #pragma omp for, #pragma omp single и #pragma omp sections. Чтобы отключить неявную барьерную синхронизацию в каком-либо из этих трех блоков разделения работы, укажите раздел nowait:

```
#pragma omp parallel
{
   #pragma omp for nowait
   for(int i = 1; i < size; ++i)
      x[i] = (y[i-1] + y[i+1])/2;
}
```

Как видите, этот раздел директивы распараллеливания говорит о том, что синхронизировать потоки в конце цикла for не надо, хотя в конце параллельного региона они все же будут синхронизированы. Второй тип — явная барьерная синхронизация. В некоторых ситуациях ее целесообразно выполнять наряду с неявной. Для этого включите в код директиву #pragma omp barrier.
В качестве барьеров можно использовать критические секции. В Win32 API для входа в критическую секцию и выхода из нее служат функции EnterCriticalSection и LeaveCriticalSection. В OpenMP для этого применяется директива #pragma omp critical [имя]. Она имеет такую же семантику, что и критическая секция Win32, и опирается на EnterCriticalSection. Вы можете использовать именованную критическую секцию, и тогда доступ к блоку кода является взаимоисключающим только для других критических секций с тем же именем (это справедливо для всего процесса). Если имя не указано, директива ставится в соответствие некоему имени, выбираемому системой. Доступ ко всем неименованным критическим секциям является взаимоисключающим.

#pragma omp for
Задает границы цикла, исполняемого в параллельном режиме.

```
#pragma omp parallel {
for(int i = 1; i < size; ++i)
x[i] = (y[i-1] + y[i+1])/2;
} // каждый поток выполнит полный цикл for, // проделав много лишней работы

//сокращенный способ записи
#pragma omp parallel for
for(int i = 1; i < size; ++i)
x[i] = (y[i-1] + y[i+1])/2;
```

Общие и частные данные
•Разрабатывая параллельные программы, вы должны понимать, какие данные являются общими (shared), а какие частными (private), — от этого зависит не только производительность, но и корректная работа программы. В OpenMP это различие очевидно, к тому же вы можете настроить его вручную.
•Общие переменные доступны всем потокам из группы, поэтому изменения таких переменных в одном потоке видимы другим потокам в параллельном регионе. Что касается частных переменных, то каждый поток из группы располагает их отдельными экземплярами, поэтому изменения таких переменных в одном потоке никак не сказываются на их экземплярах, принадлежащих другим потокам.
•По умолчанию все переменные в параллельном регионе — общие, но из этого правила есть три исключения.
•Во-первых, частными являются индексы параллельных циклов for.
•Во-вторых, частными являются локальные переменные блоков параллельных регионов.
•В-третьих, частными будут любые переменные, указанные в разделах private, firstprivate, lastprivate и reduction.

Раздел **private** говорит о том, что для каждого потока должна быть создана частная копия каждой переменной из списка. Частные копии будут инициализироваться значением по умолчанию (с применением конструктора по умолчанию, если это уместно). Например, переменные типа int имеют по умолчанию значение 0.
•У раздела **firstprivate** такая же семантика, но перед выполнением параллельного региона он указывает копировать значение общей переменной в каждый поток, используя конструктор копирования, если это уместно.
•Семантика раздела **lastprivate** тоже совпадает с семантикой раздела private, но при выполнении последней итерации цикла или раздела конструкции распараллеливания значения переменных, указанных в разделе lastprivate, присваиваются переменным основного потока. Если это уместно, для копирования объектов применяется оператор присваивания копий (copy assignment operator).

раздел reduction, но он принимает переменную и оператор. Поддерживаемые этим разделом операторы перечислены в таблице , а у переменной должен быть скалярный тип (например, float, int или long, но не std::vector, int [] и т. д.).
•Переменная раздела reduction инициализируется в каждом потоке значением, указанным в таблице. В конце блока кода оператор раздела reduction применяется к каждой частной копии переменной, а также к исходному значению переменной.

nowait - Отменяет барьерную синхронизацию при завершении выполнения параллельной секции.
schedule - По умолчанию в OpenMP для планирования параллельного выполнения циклов for применяется алгоритм, называемый статическим планированием (static scheduling). Это означает, что все потоки из группы выполняют одинаковое число итераций цикла. Если n — число итераций цикла, а T — число потоков в группе, каждый поток выполнит n/T итераций (если n не делится на T без остатка, ничего страшного). Однако OpenMP поддерживает и другие механизмы планирования, оптимальные в разных ситуациях: динамическое планирование (dynamic scheduling), планирование в период выполнения (runtime scheduling) и управляемое планирование (guided scheduling).
ordered - обеспечивает сохранение того порядка выполнения итераций цикла, который соответствует последовательному выполнению программы.

Параллельная обработка в конструкциях, отличных от циклов
•OpenMP поддерживает параллелизм и на уровне функций. Этот механизм называется секциями OpenMP (OpenMP sections).

Синтаксис:

```
#pragma omp parallel sections
{
#pragma omp section
//Вызов первой функции
#pragma omp section
//Вызов второй функции
}
```

Барьерная синхронизация. В некоторых ситуациях ее целесообразно выполнять наряду с неявной. Используется **#pragma omp barrier.**

Для синхронизации кода можно использовать и подпрограммы исполняющей среды, и директивы синхронизации. Преимущество директив в том, что они прекрасно структурированы. Это делает их более понятными и облегчает поиск мест входа в синхронизированные регионы и выхода из них.

•Критическая секция - директива **#pragma omp critical [имя].** Если используется именованная критическая секцию, тогда доступ к блоку кода является взаимоисключающим только для других критических секций с тем же именем (это справедливо для всего процесса). Если имя не указано, директива ставится в соответствие некоему имени, выбираемому системой. Доступ ко всем неименованным критическим секциям является взаимоисключающим.
В параллельных регионах часто встречаются блоки кода, доступ к которым желательно предоставлять только одному потоку, — например, блоки кода, отвечающие за запись данных в файл. Во многих таких ситуациях не имеет значения, какой поток выполнит код, важно лишь, чтобы этот поток был единственным. Для этого в OpenMP служит директива **#pragma omp single.**
•Иногда возможностей директивы single недостаточно. В ряде случаев требуется, чтобы блок кода был выполнен основным потоком, — например, если этот поток отвечает за обработку GUI и вам нужно, чтобы какую-то задачу выполнил именно он. Тогда применяется директива **#pragma omp master.** В отличие от директивы single при входе в блок master и выходе из него нет никакого неявного барьера.
•Чтобы завершить все незавершенные операции над памятью перед началом следующей операции, используйте директиву **#pragma omp flush**
Поддержка динамического создания потоков определяется значением булевого свойства, которое по умолчанию равно false. Если при входе потока в параллельный регион это свойство имеет значение false, исполняющая среда OpenMP создает группу, число потоков в которой равно значению, возвращаемому функцией omp_get_max_threads. По умолчанию omp_get_max_threads возвращает число потоков, поддерживаемых аппаратно, или значение переменной OMP_NUM_THREADS. Если поддержка динамического создания потоков включена, исполняющая среда OpenMP создаст группу, которая может содержать переменное число потоков, не превышающее значение, которое возвращается функцией omp_get_max_threads.
https://msdn.microsoft.com/ru-ru/library/dd335940.aspx

27 Базові поняття MPI: процес, ранг, комунікатор, особливості написання MPI програми.

Процесс - совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.
Коммуникатор представляет собой структуру, содержащую либо все процессы, исполняющиеся в рамках данного приложения, либо их подмножество. Процессы, принадлежащие одному и тому же коммуникатору, наделяются общим контекстом обмена. Операции обмена возможны только между процессами, связанными с общим контекстом, то есть, принадлежащие одному и тому же коммуникатору (рис. 13). Каждому коммуникатору присваивается идентификатор. В MPI есть несколько стандартных коммуникаторов:

- MPI_COMM_WORLD– включаетвсепроцессыпараллельнойпрограммы;
- MPI_COMM_SELF–включаеттолькоданныйпроцесс;
- MPI_COMM_NULL– пустой коммуникатор, не содержит ни одного

процесса.
В MPI имеются процедуры, позволяющие создавать новые коммуникаторы, содержащие подмножества процессов.
Ранг процесса представляет собой уникальный числовой идентификатор, назначаемый процессу в том или ином коммуникаторе. Ранги в разных коммуникаторах назначаются независимо и имеют целое значение от 0 до число процессов – 1.

28 Базові поняття MPI: блокуючий та неблокуючий обмін повідомленнями.

Сообщение содержит пересылаемые данные и служебную информацию. Для того, чтобы передать сообщение, необходимо указать:

- ранг процесса-отправителя сообщения;
- адрес, по которому размещаются пересылаемые данные процесса-отправителя;
- тип пересылаемых данных;
- количество данных;
- ранг процесса, который должен получить сообщение;
- адрес, по которому должны быть размещены данные процессом-получателем.
- тег сообщения;
- идентификатор коммуникатора, описывающего область взаимодействия, внутри которой происходит обмен.

Тег это задаваемое пользователем целое число от 0 до 32767, которое играет роль идентификатора сообщения и позволяет различать сообщения, приходящие от одного процесса. Теги могут использоваться и для соблюдения определенного порядка приема
сообщений.
Прием сообщения начинается с подготовки буфера достаточного размера. В этот буфер записываются принимаемые данные. Операция отправки или приема сообщения считается завершенной, если программа может вновь использовать буферы сообщений.
Двухточечные обмены используются для организации локальных и неструктурированных коммуникаций.
При выполнении глобальных операций используются коллективные обмены.
Асинхронные коммуникации реализуются с помощью запросов о получении сообщений. Имеется несколько разновидностей двухточечного обмена.

- Блокирующие прием/передача приостанавливают выполнение процесса на время приема сообщения.
- Неблокирующие прием/передача выполнение процесса продолжается в фоновом режиме, а программа в нужный момент может запросить подтверждение завершения приема сообщения.

29 Базові поняття MPI: редукція, групові операції.

В рассматриваемой программе после входной стадии каждый процессор выполняет по существу те же самые команды до заключительной стадии суммирования. Поэтому, если функция f(x) дополнительно не усложнена (т. е. не требует значительной работы для оценки интеграла по некоторым частям отрезка [a;b], то эта часть программы распределяет среди процессоров одинаковую нагрузку. В заключительной стадии суммирования процесс 0 еще раз получает непропорциональное количество работы.
Возможны также варианты организации передач, как и в случае рассылки входных данных. Поэтому следует использовать другие механизмы, более оптимизированные для этой цели.
"Общая сумма", которую нужно вычислить представляет собой пример общего класса коллективных операций коммуникации, называемых операциями редукции. В глобальной операции редукции, все процессы (в коммуникаторе) передают данные, которые объединяются с использованием бинарных операций. Типичные бинарные операции - суммирование, максимум и минимум, логические и т.д. MPI содержит специальную функцию для выполнения операции редукции:
Операция приведения, результат которой передается одному процессу:
int MPI_Reduce(void *buf, void *result, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
Входные параметры:

- buf - адрес буфера передачи;
- count - количество элементов в буфере передачи;
- datatype - тип данных в буфере передачи;
- op - операция приведения;
- root - ранг главного процесса;

• comm - коммуникатор.
MPI_Reduce применяет операцию приведения к операндам из buf, а результат каждой операции помещается в буфер результата result. MPI_Reduce должна вызываться всеми процессами в коммуникаторе comm, а аргументы count, datatype и op в этих вызовах должны совпадать.

Существует также возможность определения собственных операций.
Таким образом, завершение программы вычисления интеграла будет следующим:
/* Суммирование результатов от каждого процесса */
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
MPI_SUM, 0, MPI_COMM_WORLD);
/* Вывод результата */
Следует отметить, что каждый процессор вызывает MPI_Reduce() с одинаковыми аргументами.
Например, если total имеет значение только для процесса 0, каждый процесс тем не менее использует этот аргумент.
Глобальные операции редукции

- Сумма
- Произведение
- Минимум
- Максимум
- Битовые операции

«Собираемая» переменная может быть простой или массивом.

30. Закон Амдела.

$$P = \frac{1}{fs + \frac{1 - fs}{N}}.$$

В многопоточной программе количество потоков, выполняющих полезную работу, может изменяться в процессе исполнения. Даже если каждый поток на всем протяжении своего существования делает что-то полезное, первоначально в приложении имеется всего один поток, который должен запустить все остальные. Но такой сценарий крайне маловероятен. Потоки часто не работают, а ожидают друг друга или завершения операций ввода/вывода.

Всякий раз, как один поток чего-то ждет (неважно, чего именно), а никакого другого потока, готового занять вместо него процессор, нет, процессор, который мог бы выполнять полезную работу, простаивает.

Упрощенно можно представлять, что программа состоит из «последовательных» участков, в которых полезные действия выполняет только один поток, и «параллельных», где задействованы все имеющиеся процессоры. Если программа исполняется на машине с большим числом процессоров, то теоретически «параллельные» участки могли бы завершаться быстрее, а «последовательные» такими бы и остались. Приняв такие упрощающие предположения, можно оценить потенциальное повышение производительности при увеличении количества процессоров: если доля «последовательных» участков равна /x, то коэффициент повышения производительности *P* при числе процессоров *N* составляет

!!!Тут должна бытъ формула которая сверху

Это *закон Амдала*, на который часто ссылаются при обсуждении производительности параллельных программ. Если код полностью распараллелен, то есть доля последовательных участков нулевая, то коэффициент ускорения равен *N*. Если же, к примеру, последовательные участки составляют треть программы, то даже при бесконечном количестве процессоров не удастся добиться ускорения более чем в три раза.

Конечно, эта картина чересчур упрощенная, потому что редко встречаются бесконечно делимые задачи, без чего это соотношение неверно, и не менее редко вся работа сводится только к процессорным вычислениям, как то предполагается в законе Амдала. Во время исполнения потоки могут ожидать разных событий.

Но из закона Амдала все же следует, что если целью распараллеливания является повышение производительности, то следует проектировать всё приложение так, чтобы процессорам всегда было чем заняться. За счет уменьшения длины «последовательных» участков или времени ожидания можно повысить выигрыш от добавления новых процессоров. Альтернативный подход - подать на вход системы больше данных и тем самым загрузить параллельные участки работой; при этом можно будет уменьшить долю последовательных участков и повысить коэффициент *P*.

По существу, масштабируемость - это возможность либо уменьшить время, затрачиваемое на какое-то действие, либо увеличить объем данных, обрабатываемых в единицу времени, при увеличении количества процессоров. Иногда оба свойства эквивалентны (можно обработать больше данных, если каждый элемент обрабатывается быстрее), но это необязательно. Прежде чем выбирать способ распределения работы между потоками, важно определить, какие аспекты масштабируемости представляют для вас наибольший интерес.

В начале этого раздела я уже говорил, что у потоков не всегда есть чем заняться. Иногда они вынуждены ждать другие потоки, завершения ввода/вывода или еще чего-то. Если на время этого ожидания загрузить систему какой-нибудь полезной работой, такое простаивание можно «скрыть».

31. Ефект Гайдна.

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью. Однако размер матрицы, описывающей систему линейных уравнений, является существенно большим, чем число потоков в программе (т.е., *N<< n*), и базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько строк матрицы. При этом применение последовательной схемы разделения данных для параллельного решения систем линейных уравнений приведет к неравномерной вычислительной нагрузке между потоками: по мере исключения (на прямом ходе) или определения (на обратном ходе) неизвестных в методе Гаусса для большей части потоков все необходимые вычисления будут завершены и они окажутся простаивающими. Так проявляется ***эффект Гайдна***.

32. Паралельний метод прогонки на 2 процессори. Ефективність.

33. Паралельний метод прогонки на n процесорів. Ефективність.

(32-33 разом)

Постановка задачи

$$\begin{aligned} a_{11}x_1 &+ a_{12}x_2 &+ \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 &+ a_{21}x_2 &+ \ldots + a_{2n}x_n &= b_2 \\ &\ldots \\ a_{n1}x_1 &+ a_{n2}x_2 &+ \ldots + a_{nn}x_n &= b_n \end{aligned}$$

$$Ax=b,$$

Метод прогонки

$$A=\begin{bmatrix} c_1 & b_1 & 0 & \ldots & 0 \\ a_2 & c_2 & b_2 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & a_{n-1} & c_{n-1} & b_{n-1} \\ 0 & \ldots & 0 & a_n & c_n \end{bmatrix} \qquad Ax=f,$$

Предположим $x_i=\alpha_{i+1}x_{i+1}+\beta_{i+1}$

$$(\alpha_i a_i+c_i)x_i+b_i x_{i+1}=f_i-a_i\beta_i.$$

Прямой ход:

$$\alpha_2=-b_1/c_1,\ \alpha_{i+1}=\frac{-b_i}{a_i\alpha_i+c_i},\ i=2,\ldots,n-1,$$

$$\beta_2=f_1/c_1,\ \beta_{i+1}=\frac{f_i-a_i\beta_i}{a_i\alpha_i+c_i},\ i=2,\ldots,n-1.$$

Обратный ход:

$$x_n=\frac{f_n-a_n\beta_n}{a_n\alpha_n+c_n},\ x_i=\alpha_{i+1}x_{i+1}+\beta_{i+1},\ i=n-1,\ldots,1.$$

Левая прогонка

Прямой ход:

$$\xi_n=-a_n/c_n,\ \xi_i=\frac{-a_i}{c_i+b_i\xi_{i+1}},\ i=n-1,\ldots,2;$$

$$\eta_n=f_n/c_n,\ \eta_i=\frac{f_i-b_i\eta_{i+1}}{c_i+b_i\xi_{i+1}},\ i=n-1,\ldots,2.$$

Обратный ход:

$$x_1=\frac{f_1-b_1\eta_2}{b_1\xi_2+c_1},\ x_{i+1}=\xi_{i+1}x_i+\eta_{i+1},\ i=1,\ldots,n-1.$$

При параллельном решении системы в первом потоке по формулам вычисляются прогоночные коэффициенты α_i, β_i, при $1\le i\le p$, а во втором потоке по формулам находятся ξ_i η_i, при $p\le i\le n$. При $i=p$ проводится сопряжение решений в форме : находим значение x_p из системы

$$\begin{cases} x_p=\alpha_{p+1}x_{p+1}+\beta_{p+1} \\ x_{p+1}=\xi_{p+1}x_p+\eta_{p+1} \end{cases}.$$

Парралельный вариант метода прогонки

Рассмотрим теперь схему распараллеливания метода прогонки при использовании p потоков. Пусть нужно решить трехдиагональную систему линейных уравнений

$$a_i x_{i-1} + c_i x_i + b_i x_{i+1} = f_i,\ i=1,\ldots,n,\ x_0=x_{n+1}=0,$$

с использованием p параллельных потоков.

Применим блочный подход к разделению данных: пусть каждый поток обрабатывает $m = \lfloor n/p \rfloor$ строк матрицы A, т.е. k-й поток обрабатывает строки с номерами $1+(k-1)m \le i \le km$. Для простоты изложения мы предполагаем, что число уравнений в системе кратно числу потоков, в общем случае изменится только число уравнений в последнем потоке. Ниже представлено разделение данных для трех потоков в случае системы из 12 уравнений.

$$\left(\begin{array}{cccccccccccc|c}
c_1 & b_1 & & & & & & & & & & & f_1 \\
a_2 & c_2 & b_2 & & & & & & & & & & f_2 \\
 & a_3 & c_3 & b_3 & & & & & & & & & f_3 \\
 & & a_4 & c_4 & b_4 & & & & & & & & f_4 \\
\hline
 & & & a_5 & c_5 & b_5 & & & & & & & f_5 \\
 & & & & a_6 & c_6 & b_6 & & & & & & f_6 \\
 & & & & & a_7 & c_7 & b_7 & & & & & f_7 \\
 & & & & & & a_8 & c_8 & b_8 & & & & f_8 \\
\hline
 & & & & & & & a_9 & c_9 & b_9 & & & f_9 \\
 & & & & & & & & a_{10} & c_{10} & b_{10} & & f_{10} \\
 & & & & & & & & & a_{11} & c_{11} & b_{11} & f_{11} \\
 & & & & & & & & & & a_{12} & c_{12} & f_{12}
\end{array}\right)$$

прямой ход

$$\left(\begin{array}{cccccccccccc|c}
c_1 & b_1 & & & & & & & & & & & f_1 \\
 & \bar{c}_2 & b_2 & & & & & & & & & & \bar{f}_2 \\
 & & \bar{c}_3 & b_3 & & & & & & & & & \bar{f}_3 \\
 & & & \bar{c}_4 & b_4 & & & & & & & & \bar{f}_4 \\
\hline
 & & & a_5 & c_5 & b_5 & & & & & & & f_5 \\
 & & & d_6 & & \bar{c}_6 & b_6 & & & & & & \bar{f}_6 \\
 & & & d_7 & & & \bar{c}_7 & b_7 & & & & & \bar{f}_7 \\
 & & & d_8 & & & & \bar{c}_8 & b_8 & & & & \bar{f}_8 \\
\hline
 & & & & & & & a_9 & c_9 & b_9 & & & f_9 \\
 & & & & & & & d_{10} & & \bar{c}_{10} & b_{10} & & \bar{f}_{10} \\
 & & & & & & & d_{11} & & & \bar{c}_{11} & b_{11} & \bar{f}_{11} \\
 & & & & & & & d_{12} & & & & \bar{c}_{12} & \bar{f}_{12}
\end{array}\right)$$

Количество операций ~6n, точнее 5n/N+6(n-n/N).

Исключение элементов над диагональ

$$\left(\begin{array}{cccccccccccc|c}
c_1 & & & g_1 & & & & & & & & & \bar{f}_1 \\
 & \bar{c}_2 & & g_2 & & & & & & & & & \bar{f}_2 \\
 & & \bar{c}_3 & g_3 & & & & & & & & & \bar{f}_3 \\
 & & & \bar{c}_4 & b_4 & & & & & & & & \bar{f}_4 \\
\hline
 & & & \bar{a}_5 & c_5 & & & g_5 & & & & & \bar{f}_5 \\
 & & & \bar{d}_6 & & \bar{c}_6 & & g_6 & & & & & \bar{f}_6 \\
 & & & \bar{d}_7 & & & \bar{c}_7 & g_7 & & & & & \bar{f}_7 \\
 & & & d_8 & & & & \bar{c}_8 & b_8 & & & & \bar{f}_8 \\
\hline
 & & & & & & & \bar{a}_9 & c_9 & & & & \bar{f}_9 \\
 & & & & & & & \bar{d}_{10} & & \bar{c}_{10} & & & \bar{f}_{10} \\
 & & & & & & & \bar{d}_{11} & & & \bar{c}_{11} & & \bar{f}_{11} \\
 & & & & & & & d_{12} & & & & \bar{c}_{12} & \bar{f}_{12}
\end{array}\right)$$

Количество операций ~6n, точнее 4n/N+6(n-n/N).

Система для последовательного решения

$$\left(\begin{array}{cccc|c}
\bar{c}_4 & b_4 & & & \bar{f}_4 \\
\bar{a}_5 & c_5 & g_5 & & \bar{f}_5 \\
 & d_8 & \bar{c}_8 & b_8 & \bar{f}_8 \\
 & & \bar{a}_9 & c_9 & \bar{f}_9
\end{array}\right)$$

В результате ее решения находим значения x на границе полос, количество уравнений *2N-2*, количество операций – *8(2N-2) = 16N-16.*
После – находим решения для внутренних элементов полос
Количество операций ~5n, точнее 3*2(n/N-1)+5(n-2n/N-2N+2)
Общее количество операций ~17n

2-ая стратегия исключения

$$\left(\begin{array}{cccccccccccc|c}
c_1 & & g_1 & & & & & & & & & & \bar{f}_1 \\
 & \bar{c}_2 & g_2 & & & & & & & & & & \bar{f}_2 \\
 & & \bar{c}_3 & b_3 & & & & & & & & & \bar{f}_3 \\
 & & & \bar{c}_4 & & & g_4 & & & & & & \bar{f}_4 \\
\hline
 & & & \bar{a}_5 & c_5 & & g_5 & & & & & & \bar{f}_5 \\
 & & & \bar{d}_6 & & \bar{c}_6 & g_6 & & & & & & \bar{f}_6 \\
 & & & d_7 & & & \bar{c}_7 & b_7 & & & & & \bar{f}_7 \\
 & & & d_8 & & & & \bar{c}_8 & & & g_8 & & \bar{f}_8 \\
\hline
 & & & & & & \bar{a}_9 & & c_9 & & g_9 & & \bar{f}_9 \\
 & & & & & & \bar{d}_{10} & & & \bar{c}_{10} & g_{10} & & \bar{f}_{10} \\
 & & & & & & d_{11} & & & & \bar{c}_{11} & b_{11} & \bar{f}_{11} \\
 & & & & & & d_{12} & & & & & \bar{c}_{12} & \bar{f}_{12}
\end{array}\right)$$

$$\left(\begin{array}{ccc|c}
\bar{c}_4 & g_4 & & \bar{f}_4 \\
d_8 & \bar{c}_8 & g_8 & \bar{f}_8 \\
 & d_{12} & \bar{c}_{12} & \bar{f}_{12}
\end{array}\right)$$

Количество операций ~6n, точнее 3n/N+5(n-n/N).
После – находим решения для внутренних элементов полос
Количество операций ~5n
Общее количество операций ~17n

Эффективность

$$S_N = \frac{T_1}{T_N} \approx \frac{8n\tau}{\frac{17n\tau}{N}} \approx \frac{8N}{17}$$

$$E_N = \frac{S_N}{N} \approx \frac{8N}{17N} \approx \frac{8}{17}$$

Якщо матриця системи є розрідженою, тобто містить велику кількість нульових елементів, то в такому випадку застосовують ще одну модифікацію методу Гаусса — **метод прогонки**.
Нехай дано **систему лінійних рівнянь** з тридіагональною матрицею виду:

$$\begin{cases}
b_1x_1 + c_1x_2 = d_1 \\
a_2x_1 + b_2x_2 + c_2x_3 = d_2 \\
a_3x_2 + b_3x_3 + c_3x_4 = d_3 \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1} \\
a_nx_{n-1} + b_nx_n = d_n
\end{cases} \quad (1)$$

Запишемо систему (1) у матрично-векторній формі $Ax = d$, де

$$A = \begin{pmatrix}
b_1 & c_1 & 0 & 0 & \ldots & 0 & 0 & 0 \\
a_2 & b_2 & c_2 & 0 & \ldots & 0 & 0 & 0 \\
0 & a_3 & b_3 & c_3 & \ldots & 0 & 0 & 0 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & 0 & 0 & \ldots & a_{n-1} & b_{n-1} & c_{n-1} \\
0 & 0 & 0 & 0 & \ldots & 0 & a_n & b_n
\end{pmatrix}; d = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \ldots \\ d_{n-1} \\ d_n \end{pmatrix} \quad (2)$$

При цьому, як правило, всі елементи $b_1, b_2, \ldots, b_n$ відмінні від нуля.
Хід роботи за **методом прогонки** складається з двох етапів — **прямої прогонки** і **оберненої прогонки**. Ідея **прямої прогонки** полягає в тому, що кожне невідоме x_i виражається з допомогою прогоночних коефіцієнтів α_i, β_i:

$$x_i = \alpha_i x_{i+1} + \beta_i, i = 1,2,\ldots,n-1 \quad (3)$$

З першого рівняння системи (1) знайдемо:

$$x_1 = -\frac{c_1}{b_1}x_2 + \frac{d_1}{b_1}$$

З іншої строни, використовуючи формулу (3) бачимо, що $x_1 = \alpha_1 x_2 + \beta_1$. Прирівнюючи коефіцієнти в обох виразах для x_1 отримаємо:

$$\alpha_1 = -\frac{c_1}{b_1}, \beta_1 = \frac{d_1}{b_1} \quad (4)$$

Підставимо в друге рівняння системи замість x_1 значення $\alpha_1 x_2 + \beta_1$, отримаємо:

$$a_2(\alpha_1 x_2 + \beta_1) + b_2 x_2 + c_2 x_3 = d_2$$

і звідси знаходимо x_2:

$$x_2 = \frac{-c_2 x_3 + d_2 - a_2 \beta_1}{a_2 \alpha_1 + b_2}$$

або $x_2 = \alpha_2 x_3 + \beta_2$, де

$$\alpha_2 = -\frac{c_2}{a_2 \alpha_1 + b_2};\ \beta_2 = \frac{d_2 - a_2 \beta_1}{a_2 \alpha_1 + b_2}$$

Аналогічно можна знайти коефіцієнти для будь-якого i:

$$\alpha_i = -\frac{c_i}{a_i \alpha_{i-1} + b_i};\ \beta_i = \frac{d_i - a_i \beta_{i-1}}{a_i \alpha_{i-1} + b_i};\ i = 2,3,\dots,n-1 \quad (5)$$

Обернена прогонка полягає у послідовному обчисленні невідомих x_i. Для цього спочатку, з останнього рівняння системи, знаходимо x_n:

$$a_n x_{n-1} - b_n x_n = d_n$$

Підставимо в останній вираз замість x_{n-1} значення $\alpha_{n-1} x_n + \beta_{n-1}$ отримаємо:

$$x_n = \frac{d_n - a_n \beta_{n-1}}{a_n \alpha_{n-1} + b_n}$$

Дальше, використовуючи формулу (1) і вирази для прогоночних коефіцієнтів (4), (5), послідовно знаходимо всі невідомі $x_{n-1}, x_{n-2}, \dots, x_1$

Розв'язок системи лінійних рівнянь методом прогонки — приклад:

Використовуючи **метод прогонки**, **розв'язати систему лінійних алгебраїчних рівнянь** наступного виду:

$$\begin{cases} x_1 + 2x_2 = -5 \\ x_1 + 10x_2 - 5x_3 = -9 \\ x_2 - 5x_3 + 2x_4 = -20 \\ x_3 + 4x_4 = -27 \end{cases}$$

Для цього, на першому кроці, скориставшись **прямим ходом методу прогонки** (розрахункові формули (4), (5)), обчислюємо прогоночні коефіцієнти α_i та β_i:

$$\alpha_1 = -\frac{c_1}{b_1} = -\frac{2}{1} = -2;$$

$$\beta_1 = \frac{d_1}{b_1} = \frac{-5}{1} = -5;$$

$$\alpha_2 = -\frac{c_2}{a_2 \cdot \alpha_1 + b_2} = -\frac{-5}{1 \cdot (-2) + 10} = \frac{5}{8} = 0.625;$$

$$\beta_2 = \frac{d_2 - a_2 \cdot \beta_1}{a_2 \cdot \alpha_1 + b_2} = \frac{-9 - 1 \cdot (-5)}{1 \cdot (-2) + 10} = -\frac{4}{8} = -0.5;$$

$$\alpha_3 = -\frac{c_3}{a_3 \cdot \alpha_2 + b_3} = -\frac{2}{1 \cdot 0.625 - 5} = -\frac{2}{-4.375} = 0.457;$$

$$\beta_3 = \frac{d_3 - a_3 \beta_2}{a_3 \cdot \alpha_2 + b_3} = \frac{-20 - 1 \cdot (-0.5)}{1 \cdot 0.625 - 5} = \frac{-19.5}{-4.375} = 4.457;$$

$$\beta_4 = \frac{d_4 - a_4 \cdot \beta_3}{a_4 \cdot \alpha_3 + b_4} = \frac{-27 - 1 \cdot 4.457}{1 \cdot 0.457 + 4} = \frac{-31.457}{4.457} = -7.059;$$

Далі, використовуючи знайдені прогоночні коефіцієнти, **знаходимо значення невідомих** x_i (**обернена прогонка**):

31. **Паралельний метод Гаусса. Ефективність та алгоритмічна складність.**
32. **Паралельний алгоритм обчислюваня зворотнього ходу метода Гаусса.**

(34-35 разом)

Параллельный алгоритм

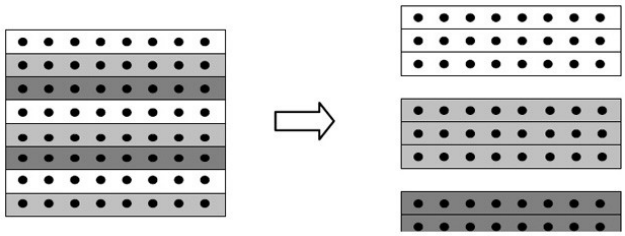
При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В качестве *базовой подзадачи* можно принять тогда все вычисления, связанные с обработкой одной строки матрицы *A* и соответствующего элемента вектора *b*. Рассмотрим общую схему параллельных вычислений и возникающие при этом информационные зависимости между базовыми подзадачами.

Для выполнения прямого хода метода Гаусса необходимо осуществить $(n-1)$ итерацию по исключению неизвестных для преобразования матрицы коэффициентов *A* к верхнему треугольному виду. Выполнение итерации i, $1 \le i \le n$, прямого хода метода Гаусса включает ряд последовательных действий. Прежде всего, в самом начале итерации необходимо выбрать ведущую строку, которая при использовании метода главных элементов определяется поиском строки с наибольшим по абсолютной величине значением среди элементов столбца i, соответствующего исключаемой переменной x_i. Зная ведущую строку, подзадачи выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной x_i.

При выполнении обратного хода метода Гаусса подзадачи выполняют необходимые вычисления для нахождения значения неизвестных. Как только какая-либо подзадача i, $1 \le i \le n$, определяет значение своей переменной x_i, это значение должно быть использовано всеми подзадачам с номерами k, $k < i$: подзадачи подставляют полученное значение новой неизвестной и выполняют корректировку значений для элементов вектора *b*.

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью. Однако размер матрицы, описывающей систему линейных уравнений, является существенно большим, чем число потоков в программе (т.е., $p \ll n$), и базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько строк матрицы. При этом применение последовательной схемы разделения данных для параллельного решения систем линейных уравнений приведет к неравномерной вычислительной нагрузке между потоками: по мере исключения (на прямом ходе) или определения (на обратном ходе) неизвестных в методе Гаусса для большей части потоков все необходимые вычисления будут завершены и они окажутся простаивающими. Возможное решение проблемы балансировки вычислений может состоять в использовании ленточной циклической схемы для распределения данных между укрупненными подзадачами. В этом случае матрица *A* делится на наборы (полосы) строк вида (см. рис. 7.1).

$$A = (A_1, A_2, \dots, A_p)^T \; A_i = (a_{i_1}, a_{i_2}, \dots, a_{i_k}) \; i_j = i + jp, 1 \le j \le$$



Метод исключения Гаусса

Прямой ход состоит в последовательном исключении неизвестных в уравнениях решаемой системы линейных уравнений.

На итерации i, $1 \le i < n$, метода производится исключение неизвестной i для всех уравнений с номерами k, больших i (т.е. $i < k \le n$). Для этого из этих уравнений осуществляется вычитание строки i, умноженной на константу a_{ki}/a_{ii} с тем, чтобы результирующий коэффициент при неизвестной x_i в строках оказался нулевым – все необходимые вычисления могут быть определены при помощи соотношений:

$$\begin{aligned} a'_{kj} &= a_{kj} - \mu_{ki} a_{ij}, \\ b'_k &= b_k - \mu_{ki} b_i, \end{aligned} \quad i \le j \le n, i < k \le n, 1 \le i < n \;,$$

где $\mu_{ki} = a_{ki} / a_{ii}$ – *множители Гаусса*.

В итоге приходим к системе $Ux=c$ с верхней треугольной матрицей

$$U = \begin{pmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,n} \\ 0 & u_{2,2} & \dots & u_{2,n} \\ & & \dots & \\ 0 & 0 & \dots & u_{n,n} \end{pmatrix},$$

При выполнении прямого хода метода Гаусса строка, которая используется для исключения неизвестных, носит наименование *ведущей*, а диагональный элемент ведущей строки называется *ведущим элементом*.

Обратный ход алгоритма состоит в следующем. После приведения матрицы коэффициентов к верхнему треугольному виду становится возможным определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_n, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-1} и т.д. В общем виде выполняемые вычисления при обратном ходе метода Гаусса могут быть представлены при помощи соотношений:

$$x_i = \left(c_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) \Big/ u_{ii} \;, \; i=n,\dots,1.$$

Оценим трудоемкость метода Гаусса. При выполнении прямого хода число операций составит

$$\sum_{i=1}^{n-1} 2(n-i)^2 = \frac{n(n-1)(2n-1)}{3} = \frac{2}{3} n^3 + O(n^2) \,.$$

Для выполнения обратного хода потребуется

$$\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2} = O(n^2) \,.$$

Таким образом, общее время выполнения метода Гаусса при больших n можно оценить как

$$T_1 = \frac{2}{3} n^3 \tau \,,$$

Эффективность

При разработке параллельного алгоритма все вычислительные операции, выполняемые алгоритмом Гаусса, были распределены между потоками параллельной программы. Следовательно, время, необходимое для выполнения вычислений на этапе прямого хода, можно оценить как $T_n=T_1/N$.

$$S_N = \frac{T_1}{T_N} \approx \frac{2n^3\tau/3}{\frac{2n^3\tau/3}{N}} \approx N$$

$$E_N = \frac{S_N}{N} \approx \frac{N}{N} = 1$$

33. Паралельний алгоритм LU розкладу.

$$l_{i1} = a_{i1} \ , \ i = 1 \ to \ n$$

$$u_{1j} = \frac{a_{1j}}{l_{11}} \ , \ j = 2 \ to \ n$$

$$For \ j = 2 \ to \ n-1 \ do \ steps \ 1, 2$$

$$Step \ 1 : l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \ , \ i = j \ to \ n$$

$$Step \ 2 : u_{jk} = \frac{a_{jk} - \sum_{i=1}^{j-1} l_{ji} u_{ik}}{l_{jj}} \ , \ k = j+1 \ to \ n$$

$$l_{nn} = a_{nn} - \sum_{k=1}^{n-1} l_{nk} u_{kn}$$

LU-разложение

- LU-разложение - представление матрицы A в виде

A=LU,

где *L* - нижняя треугольная матрица с диагональными элементами, равными единице, а *U* - верхняя треугольная матрица с ненулевыми диагональными элементами. *LU*-разложение также называют *LU*-факторизацией. Известно, что *LU*-разложение существует и единственно, если главные миноры матрицы *A* отличны от нуля.

Алгоритм *LU*-разложения тесно связан с методом исключения Гаусса. В самом деле, пусть мы решаем систему уравнений вида *Ax=b*. Непосредственно проверяется, что преобразования *k*-го шага метода Гаусса равносильны домножению системы *Ax=b* слева на матрицу

$$M^{(k)} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -\mu_{k+1,k} & 1 & & \\ & & -\mu_{k+2,k} & & 1 & \\ & & & & & \ddots \\ & & -\mu_{n,k} & & & 1 \end{bmatrix},$$

- где μ_{ik} - множители Гаусса. Прямой ход метода Гаусса преобразует исходную систему уравнений к виду *Ux=c*, с верхней треугольной матрицей *U*. Зная матрицы *M*(*i*), можно записать матрицу *U* и вектор *c* как

$$U = M^{(n-1)} M^{(n-1)} \ldots M^{(1)} A,$$
$$c = M^{(n-1)} M^{(n-1)} \ldots M^{(1)} b \ .$$

Обозначим $L^{-1} = M^{(n-1)} M^{(n-1)} \ldots M^{(1)} A$, тогда

$$L = \begin{bmatrix} 1 & & & & \\ \mu_{21} & 1 & & & \\ \vdots & \vdots & \ddots & & \\ \mu_{n-1,1} & \mu_{n-1,2} & \cdots & 1 & \\ \mu_{n,1} & \mu_{n,2} & \cdots & \mu_{n,n-1} & 1 \end{bmatrix}$$

- Таким образом, матрицу L можно получить как нижнюю треугольную матрицу коэффициентов Гаусса, а матрицу U - как верхнюю треугольную матрицу, получаемую в результате работы метода Гаусса. При этом очевидно, что трудоемкость получения LU-факторизации будет такой же –

$$2/3n^3 + O(n^2).$$

Если LU - разложение получено, то решение системы *Ax=b* сводится к последовательному решению двух систем уравнений с треугольными матрицами (*обратный ход)*

Ly=b, *Ux=y*. (1.21)

Обратный ход требует $O(n^2)$ операций. Как следует из приведенных оценок, вычислительная сложность метода исключения Гаусса и метода *LU*-разложения одинакова. Однако если необходимо решить несколько систем с одинаковыми матрицами коэффициентов, но различными векторами свободных членов (правая часть СЛАУ), то метод *LU*-разложения окажется предпочтительным, так как в этом случае нет необходимости производить разложение матрицы коэффициентов многократно. Достаточно лишь сохранить полученные треугольные матрицы в памяти и, подставляя различные вектора свободных членов, получать решения методами прямой и обратной подстановки. Это позволит значительно сократить объем вычислений по сравнению с методом Гаусса.

34. Ітераційні методи розв'язання СЛАР. Метод Якобі.

Постановка задачи

При большом числе неизвестных метод Гаусса становится весьма сложным в плане вычислительных и временных затрат. Поэтому иногда удобнее использовать приближенные (итерационные) численные методы, метод Якоби относится к таким.

В работе требуется решить систему линейных алгебраических уравнений вида:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}$$

$$Ax = b$$

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2(n-1)} & a_{2n} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{(n-1)1} & a_{(n-1)2} & \ldots & a_{(n-1)(n-1)} & a_{(n-1)n} \\ a_{n1} & a_{n2} & \ldots & a_{n(n-1)} & a_{nn} \end{pmatrix} \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_{n-1} \\ b_n \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_{n-1} \\ x_n \end{pmatrix}$$

При предположении, что диагональные коэффициенты ненулевые.

$$a_{ii} \neq 0 \qquad i = (1,2,..,n)$$

Метод решения

Решив 1-ое уравнение системы относительно x1 получим:

$$x_1 = \frac{b_1 - (a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n)}{a_{11}}$$

2-ое - относительно x2 , n-ое - относительно xn

В итоге эквивалентная система, в которой диагональные элементы строки выражены через оставшиеся.

$$\begin{cases} x_1 = \frac{b_1 - (a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n)}{a_{11}} \\ x_2 = \frac{b_2 - (a_{21}x_1 + a_{23}x_3 + \ldots + a_{2n}x_n)}{a_{22}} \\ \ldots \\ x_n = \frac{b_n - (a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{n(n-1)}x_{n-1})}{a_{nn}} \end{cases}$$

или

$$x = \alpha x + \beta$$

$$\alpha = \begin{pmatrix} 0 & -\frac{a_{12}}{a_{11}} & -\frac{a_{13}}{a_{11}} & \ldots & -\frac{a_{1n}}{a_{11}} \\ -\frac{a_{21}}{a_{22}} & 0 & -\frac{a_{23}}{a_{22}} & \ldots & -\frac{a_{2n}}{a_{22}} \\ & & \ldots & & \\ -\frac{a_{n1}}{a_{nn}} & -\frac{a_{n2}}{a_{nn}} & -\frac{a_{n3}}{a_{nn}} & \ldots & 0 \end{pmatrix} \qquad \beta = \begin{pmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ \ldots \\ \frac{b_n}{a_{nn}} \end{pmatrix}$$

Далее вводится некоторое начальное приближение - вектор x(0)=[b1/a11, … ,bi/aii, … ,bn/ann], затем используя x(1) находится x(2).

Данный процесс называется итерационным, условием окончания является достижение заданной точности (система сходится и есть решение) или прерывание процесса. Процесс прерывается когда число итераций превышает заданное допустимое количество, при этом система не сходится либо заданное количество итераций не хватило для достижения требуемой точности.

Итерационный процесс. Верхний индекс в скобках - номер итерации.

$$\begin{cases} x_1^{(k+1)} = \frac{b_1 - \left(a_{12}x_2^{(k)} + a_{13}x_3^{(k)} + \ldots + a_{1n}x_n^{(k)}\right)}{a_{11}} \\ x_2^{(k+1)} = \frac{b_2 - \left(a_{21}x_1^{(k)} + a_{23}x_3^{(k)} + \ldots + a_{2n}x_n^{(k)}\right)}{a_{22}} \\ \ldots \\ x_n^{(k+1)} = \frac{b_n - \left(a_{n1}x_1^{(k)} + a_{n2}x_2^{(k)} + \ldots + a_{n(n-1)}x_{n-1}^{(k)}\right)}{a_{nn}} \end{cases}$$

$$x^{(k+1)} = \alpha x^{(k)} + \beta$$

$x^{(1)} = \alpha x^{(0)} + \beta$ - первое приближение

$x^{(2)} = \alpha x^{(1)} + \beta$ - второе приближение

$x^{(k+1)} = \alpha x^{(k)} + \beta$ - (k+1) - ое приближение

Если последовательность приближений (x(0),x(1),…,x(k+1),…) имеет предел

$$x = \lim_{k \to \infty} x^{(k)}$$

то этот предел является решением. k=1,2,3,…N-1. , N-1 - заданное количество итераций

Достаточный признак сходимости метода Якоби:
Если в системе выполняется диагональное преобладание, то метод Якоби сходится.

$$|a_{ii}| > \sum_{j=1,(j \neq i)}^{n} |a_{ij}| \qquad i = 1,2,..,n$$

Критерий окончания итераций при достижении требуемой точности имеет вид:

$$\left|\left|x^{(k+1)} - x^{k}\right|\right| < \varepsilon$$

где ε - заданная точность вычисления

Параллельная схема

При распараллеливании алгоритма предполагается, что размерность системы больше числа процессоров. Каждый процессор считает "свои" элементы вектора X=(x1,x2,x3,…,xn).

Перед началом выполнения метода на каждый процессор рассылаются необходимые данные:

1) размерность системы N
2) начальное приближение X0
3) строки матрицы A
4) элементы вектора свободных членов b.

После получения необходимой информации каждый процессор будет вычислять соответствующие компоненты вектора X и передавать их главному процессору. Главный процессор при получении очередного приближения решения Xk должен сравнить его с предыдущим приближением Xk-1. Если норма разности этих векторов:

$$||Xk - Xk\text{-}1|| = \max\{|Xk,i - Xk,i\text{-}1|, i = 1,1,\ldots n\}$$

окажется меньше или равной заданной точности (ε), то вычисления закончатся. В противном случае вектор Xk поэлементно будет разослан всем процессам и будет вычисляться очередное приближение решения.

Анализ эффективности

Время, затраченное на последовательное выполнение алгоритма, выражается следующей формулой:
T = k * (2*n^2 - n), где *k* – число выполненных итераций, а *n* – число уравнений.

Время, затраченное на параллельное выполнение алгоритма на p процессорах без учета затрат на передачу данных, выражается формулой:
T(p) = k/p * (2*n^2 - n)

Тогда ускорение равно:
S = T(1) / T(p) = p

Эффективность:
E = T1 / p*Tp = 1

Разработанный способ параллельных вычислений позволяет достичь идеальных показателей ускорения и эффективности.

Определим время, затрачиваемое на передачу данных. Для этого используем модель Хокни.

Первоначальная рассылка данных требует следующее время:
Tcomm1 = (p-1) * (4*alfa + (n^2 / p + n) / betta), где *alfa* - латентность, *betta* - пропускная способность сети

Передача данных, выполняемая в итерационном процессе, затрачивает следующее время:
Tcomm2 = k * (p-1) * (3*alfa + (n / p + n) / betta), где *k* - количество выполненных итераций

В итоге общее время передачи данных выражается формулой:
Tcomm = (p-1) * (4*alfa + (n^2 / p + n) / betta) + k * (p-1) * (3*alfa + (n / p + n) / betta)

Это время зависит от числа итераций. Как правило, их количество меньше числа уравнений *n*. Значит время на передачу данных можно оценить величиной:
Tcomm = O(n^2)

В свою очередь и ко времени выполнения алгоритма применима та же оценка:
T = O(n^2)

Если число итераций будет сравнимо с *n*, то для времени выполнения алгоритма будет справедлива уже другая оценка:
T = O(n^3)

35. Ітераційні методи розв'язання СЛАР. Метод Зейделя.

АЛГОРИТМ ПАРАЛЛЕЛЬНОГО РЕШЕНИЯ СЛАУ МЕТОДОМ ГАУССА - ЗЕЙДЕЛЯ*

This work concerns the parallel Gauss - Zeidel method. The peculiarity of this method is consecutive calculation; therefore, the effective organization of parallel computing for this method could be synthesized to other calculus of approximations. The parallel method is based on the principle of preliminary calculations, allowing minimizing the influence of consecutive calculation of the roots.

Локальные сети на базе персональных ЭВМ без дополнительных материальных вложений нетрудно превратить в кластеры для параллельных вычислений [1]. Время решения системы линейных алгебраических уравнений (СЛАУ) и ряда других задач на таких кластерах может быть уменьшено во много раз. При этом кластеры, в частности вузовские, могут быть объединены в сеть для распределенных вычислений, что позволит многократно увеличить степень распараллеливания вычислений.

* Авторы статьи - сотрудники кафедры информатики.

$S_{\text{Л},i-1} = \sum_{j=1}^{i-2} a_{i-1,j} x_j^{k+1}$ и $S_{\text{П},i-1} = \sum_{j=i-1}^{m} a_{i-1,j} x_j^{k} + b_{i-1}$, тогда $x_{i-1}^{k+1} = S_{\text{Л},i-1} + S_{\text{П},i-1}$. Полученное значение этой компоненты вектора приближения через коммуникационную систему рассылается всем процессорам выше и ниже процессора строки i-1.

Теперь рассмотрим последовательность операций в процессоре строки i, где уже сформировано значение суммы $S_{\text{Л},i} = \sum_{j=1}^{i-2} a_{i,j} x_j^{k+1}$. Такие же суммы уже вычислены и всеми низлежащими процессорами для своих коэффициентов. Далее в строке i необходимо выполнить две следующие операции: $S_{\text{Л},i} = S_{\text{Л},i} + a_{i,i-1} x_{i-1}^{k+1}$ и $x_i^{k+1} = S_{\text{Л},i} + S_{\text{П},i}$. Эти операции на уровне строки i требуют времени $t = t_*+t_+$, где t_* и t_+ - время операции умножения и сложения соответственно. Все операции на уровнях $i + 1$, $i + 2$ и так далее до уровня m будут повторяться, поэтому полное время T_n параллельного решения СЛАУ методом Гаусса - Зейделя без учета обменов для m процессоров будет $T_n = mt$.

Как отмечалось, полученное в строке i значение x_i^{k+1} посылается также и в процессоры, лежащие выше строки i. Эти процессоры уже освободились от вычисления компонент x_l^{k+1}, где $l < i$, и используются для вычисления в каждой l-й строке сумм $\sum_{j=1}^{i} a_{l,j} x_j^{k+1}$. Поэтому после вычисления последней компоненты вектора приближения x_m^{k+1} в матрице будут получены все суммы строк для следующей $(k+2)$-й итерации.

Приведенная ситуация для $n = m$ использовалась для пояснения принципа распараллеливания. На практике обычно $n < m$ или даже $n<<m$. Рассмотрим процесс параллельных вычислений для двух смежных процессоров p - 1 и p (рис. 1 *б)*, каждый из которых вычисляет m/n компонент вектора приближения.

При входе в зону процессора p - 1 (точка z_{p-1}) для каждой его строки, как и прежде (см. рис. 1 *а)*, на основе вычисленных ранее компонент вектора приближения x_l^{k+1} сформированы суммы для левой и правой частей строк, за исключением треугольника a. Таким образом, для получения m/n компонент в зоне процессора p - 1 необходимо произвести вычисления в треугольнике a. Эти вычисленные компоненты через коммуникационную систему будут переданы во все процессоры, находящиеся выше и ниже процессора p - 1, в том числе и в процессор p, который вычислит очередные m/n компонент вектора приближения. Назовем шагом вычислений все операции, отнесенные к одному процессору. Следовательно, сумма затрат времени на всех n шагах и составит время решения СЛАУ T_n=$nt_{\text{ш}}$.

Пусть время одного шага $t_{\text{ш}}$ состоит из времени перехода от точки z_{p-1} до точки z_p. Чтобы процессор p мог начать вычисления в точке z_p, должны быть выполнены следующие действия.

- Ожидание при вычислении первой части компонент треугольника a:

$$t_{\text{ож}} = \left(\frac{m}{ne}\right)^2 \frac{1}{2} t$$

где $t = t_*+t_+$, $e = 1, \ldots, m/n$ - доля компонент вектора приближения, вычисляемых в процессоре p - 1 до первой посылки в другие процессоры. Если $e = 1$, передача производится после вычисления всех компонент процессором p - 1; в случае $e = m/n$ - после вычисления каждой компоненты.

- Далее выполняется e тактов. Каждый такт состоит из двух фаз: передача e-й части компонент вектора приближения во все n процессоров (время t_{e1}); вычисление очередной e-й части компонент вектора приближения в треугольнике a и строк в прямоугольнике b (t_{e2}). Тогда

$$t_e = t_{e1} + t_{e2} = \left(t_{\text{л}} + \frac{vm}{ne} t_{\text{сл}}\right) + \left(\frac{m}{ne}\right)^2 t$$

где t_* - время одного такта обработки, $t_{\text{л}}$ - латентность (задержка) коммуникатора, $t_{\text{сл}}$ - время передачи одного слова (8 байтов) коммуникационной системой, v - коэффициент, зависящий от реализации операции *bcast* в коммуникаторе. Если устройство коммуникатора позволяет производить передачу всем процессорам одновременно, тогда v=1; в случае передачи по двоичному дереву v= $\log_2 n$; если передача производится поочередно каждому процессору, то v=n [5]. Время te_2 определяется временем вычисления части прямоугольника b, поскольку оно превосходит время вычисления соответствующей части треугольника c.

Так как треугольники c и a равны, то для процессора p вычисления выполняются аналогично, поэтому время расчета одной итерации при n>1 равно

$$T_n = n(t_{\text{ож}} + e \cdot t_e). \quad (1)$$

Если средства кластера позволяют совмещать вычисления и обмен, то время T_n будет существенно меньше, чем определено выражением (1).

Величина T_n является наиболее объективной характеристикой оценки параллельного алгоритма, но часто используют параметр ускорения [1]:

$$R = T_1 / T_n = m^2 t / T_n, \quad (2)$$

здесь T_1, T_n- время расчета одной и той же итерации для однопроцессорной и n-процессорной систем.

Формула (2) часто дает завышенные результаты для кластеров на ПЭВМ, поскольку в этом случае время T_1 из-за недостатка оперативной памяти может быть чрезмерно большим. Для кластеров типа СКИФ с большим объемом оперативной памяти это явление выражено намного слабее и показатель R более объективный.

На рис. 2 представлены экспериментальные графики измерения времени: на кластере ПЭВМ (МП Celeron с частотой 1.7 GHz и коммуникатор Fast Ethernet со скоростью обмена 10 MB/s [5]) и кластере СКИФ-1000 (МП Opteron с частотой 2.4 GHz и коммуникатором Infiniband со скоростью обмена 830 MB/s [6]). Программы были написаны в стандарте MPI [1], матрица коэффициентов имела размер 4000 x 4000. Очевидным в обоих случаях является факт уменьшения времени счета при использовании параллельного алгоритма.

здесь T_1, T_n- время расчета одной и той же итерации для однопроцессорной и n-процессорной систем.

Формула (2) часто дает завышенные результаты для кластеров на ПЭВМ, поскольку в этом случае время T_1 из-за недостатка оперативной памяти может быть чрезмерно большим. Для кластеров типа СКИФ с большим объемом оперативной памяти это явление выражено намного слабее и показатель R более объективный.

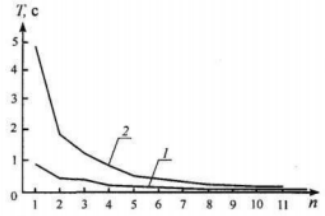


Рис. 2. Зависимость времени счета от числа процессоров: *1* – СКИФ, *2* – ПЭВМ

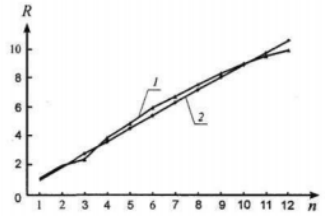


Рис. 3. Кластер СКИФ: *1* – расчет, *2* – эксперимент ($v = n$, $e = 2$)

На рис. 3 представлены характеристики ускорения для кластера СКИФ, полученные

расчетом по формуле (2) и по результатам эксперимента, подтверждающие возможность использования данной формулы (2) для оценочных расчетов.

Результаты экспериментов и расчета показывают, что ускорение для кластеров ПЭВМ и СКИФ К-1000 имеет примерно линейный характер с хорошим коэффициентом утилизации $R/n = 0{,}8 - 0{,}9$, который будет сохраняться при одновременном увеличении размера матрицы m и числа процессоров n.

Таким образом, разработанный алгоритм имеет высокую эффективность и может использоваться в практике. Эффективность достигнута распределением процессоров по строкам, что позволяет за счет резкого сокращения обменов при суммировании в одной строке выполнять не только упреждающие вычисления, но и одновременно готовить строки для новой итерации.

Методом Зейделя решить с точностью *0,001* систему линейных уравнений, приведя ее к виду, удобному для итерации.

$3.6x_1 + 1.8x_2 - 4.7x_3 = 3.8$

$2.7x_1 - 3.6x_2 + 1.9x_3 = 0.4$

$1.5x_1 + 4.5x_2 + 3.3x_3 = -1.6$

Решение. Умножаем матрицы A^TA.

$$A^TA = \begin{pmatrix} 22,5 & 3,51 & -6,84 \\ 3,51 & 36,45 & -0,45 \\ -6,84 & -0,45 & 36,59 \end{pmatrix}$$

Умножаем матрицы A^Tb.

$$A^Tb = \begin{pmatrix} 12,36 \\ -1,8 \\ -22,38 \end{pmatrix}$$

Приведем к виду:

$x_1=0.55+0.16x_2-0.3x_3$

$x_2=-0.0494+0.0963x_1-0.0123x_3$

$x_3=-0.61-0.19x_1-0.0123x_2$

Покажем вычисления на примере нескольких итераций.

N=1

$x_1=0.55 - 0 \bullet 0.16 - 0 \bullet (-0.3)=0.55$

$x_2=-0.0494 - 0.55 \bullet 0.0963 - 0 \bullet (-0.0123)=-0.1$

$x_3=-0.61 - 0.55 \bullet (-0.19) - (-0.1) \bullet (-0.0123)=-0.51$

N=2

$x_1=0.55 - (-0.1) \bullet 0.16 - (-0.51) \bullet (-0.3)=0.41$

$x_2=-0.0494 - 0.41 \bullet 0.0963 - (-0.51) \bullet (-0.0123)=-0.0952$

$x_3=-0.61 - 0.41 \bullet (-0.19) - (-0.0952) \bullet (-0.0123)=-0.54$

N=3

$x_1=0.55 - (-0.0952) \bullet 0.16 - (-0.54) \bullet (-0.3)=0.4$

$x_2=-0.0494 - 0.4 \bullet 0.0963 - (-0.54) \bullet (-0.0123)=-0.0946$

$x_3=-0.61 - 0.4 \bullet (-0.19) - (-0.0946) \bullet (-0.0123)=-0.54$

Остальные расчеты сведем в таблицу.

N	x_1	x_2	x_3	e_1	e_2	e_3
0	0	0	0			
1	0.55	-0.1	-0.51	0.55	0.1	0.51
2	0.41	-0.0952	-0.54	-0.14	-0.0071	0.0259
3	0.4	-0.0946	-0.54	-0.00899	-0.000546	0.00167
4	0.4	-0.0946	-0.54	-0.000594	-3.7E-5	0.000111

Ответ: $x_1 = 0.4$; $x_2 = -0.0946$; $x_3 = -0.54$.

36. Паралельні методи розв'язання ДРЧП. Рівняння теплопровідності. Явна схема.

4.2. Решение задачи теплопроводности

Процесс распространения тепла в одномерном однородном стержне $0<x<l$ описывается уравнением

$$\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} + f(x,t),$$

Без ограничения общности можно считать $l=1$, $\alpha^2=1$. Мы будем рассматривать первую краевую задачу в области

$$D=\{0\le x\le 1,\ 0\le t\le T\}.$$

Требуется найти непрерывное в D решение $u=u(x,t)$ задачи

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + f(x,t),\ 0<x<1,\ 0<t\le T,$$

$$u(x,0)=u_0(x),\ 0\le x\le 1,\ u(0,t)=u_1(t),\ u(1,t)=u_2(t),\ 0\le t\le T.$$

С целью построения разностной схемы в области D введем сетку

$$\omega_{h\tau}=\{(x_i,t_j):\ x_i=ih,\ 0\le i\le n,\ h=1/n,\ t_j=j\tau,\ 0\le j\le m,\ \tau=T/m\}.$$

с шагами h по x и τ по t. Используя приближенные формулы вычисления производных

$$\left(\frac{\partial u}{\partial t}\right)_{ij} \approx \frac{u_i^{j+1}-u_i^j}{\tau},\ \left(\frac{\partial^2 u}{\partial x^2}\right)_{ij} \approx \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{h^2},$$

можно записать разностную схему вида

$$\frac{v_i^{j+1}-v_i^j}{\tau} = \sigma\frac{v_{i+1}^{j+1}-2v_i^{j+1}+v_{i-1}^{j+1}}{h^2} + (1-\sigma)\frac{v_{i+1}^j-2v_i^j+v_{i-1}^j}{h^2} + \varphi_i^j, \qquad (4.7)$$

$$v_i^0 = u_0(x_i), v_0^j = u_1(t_j), v_n^j = u_2(t_j), 0\le i\le n, 0\le j\le m.$$

где v_i^j – сеточная функция, являющаяся точным решением разностной схемы и аппроксимирующая точное решение дифференциального уравнения в узлах сетки; σ – параметр, называемый весом, а φ_i^j – некоторая правая часть, например, $\varphi_i^j = f_i^j$. В общем случае схема (4.7) определена на шеститочечном шаблоне, изображенном на рис. 4.5.

Рис. 4.5. Шаблон схемы с весом

Далее рассмотрим важные частные случаи схемы (4.7).

4.2.1. Явная разностная схема

В случае $\sigma=0$ схема (4.7) приобретет вид

$$\frac{v_i^{j+1}-v_i^j}{\tau} = \frac{v_{i+1}^j - 2v_i^j + v_{i-1}^j}{h^2} + \varphi_i^j, \qquad (4.8)$$

где $\varphi_i^j = f_i^j$; граничные условия при этом останутся неизменными. Шаблон схемы 4.6 представлен на рис. 4.6.

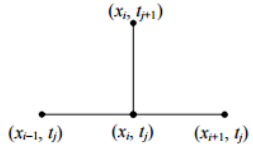


Рис. 4.6. Шаблон явной схемы

Данная схема является явной. Действительно, значения сеточной функции на $(j+1)$-м слое можно найти по явной формуле

$$v_i^{j+1} = \left(1-\frac{2\tau}{h^2}\right)v_i^j + \frac{\tau}{h^2}\left(v_{i+1}^j + v_{i-1}^j\right) + \tau\varphi_i^j, 0\le j\le L-1 \qquad (4.9)$$

используя известные значения на j-м слое.

Известно [4], что явная разностная схема будет вычислительно устойчива лишь при выполнении условия

$$\tau\le h^2/2.$$

т.е. схема является условно устойчивой. При этом схема аппроксимирует исходное уравнение с порядком $O(\tau+h^2)$.

Условие вычислительной устойчивости налагает существенные требования на шаг по времени, поэтому данная вычислительная схема требует существенных вычислительных затрат. Применение явных разностных схем оправдано лишь в случае более слабых ограничений на шаг τ (см. п. 4.1.1).

37. Паралельні методи розв’язання ДРЧП. Рівняння теплопровідності. Неявна схема схема.

4.1.1. **Розв'язання задачі Коші. Явні схеми** Задача Коші для (4.1), (4.5) полягає у знаходженні T x t (,): ∀ > −∞ < < +∞ t x 0, , яке задовольняло б (4.1), а на прямій t = 0 по- чаткову умову (4.5), де Θ(x) – відома функція для ∀ ∈ −∞ +∞ x (,). Для отримання наближеного розв'язку (4.1) початкову умову (4.5) дискретизують. Вибір сітки тут досить простий: xj = jΔx, tn = nΔt (j = 0, ±1,…; n = 0, 1, 2,…), де Δx − крок у напрямку Ox, Δt − крок у напрямку Ot. Вузли (xj , tn) вважатимемо внутрішніми, 90 якщо n ≥ 1. При n = 1 (t = 0) вузли вважатимемо граничними. За такого вибору сітки початкова умова (4.5) набуває вигляду 0 , 0, 1, 2,... T j j j = θ = ± ± , де ( ). j j θ = Θ x (4.6) Дискретизація диференціального рівняння (4.10) залежить від вибору шаблону. При використанні явних шаблонів єдина неві- дома, наприклад n 1 Tj + , фігурує у лівій частині алгебраїчної фор- мули, яку отримали внаслідок дискретизації (рис.)

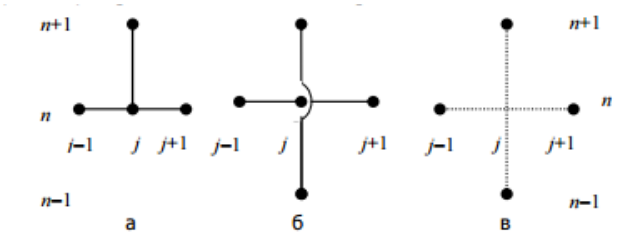


Рис. 4.2. Явні шаблони для рівняння дифузії:
а – FTCS; б – Річардсона; в – Дюфорта–Франкела

Якщо для похідної за часом увести двоточкову різницеву ап- роксимацію зі зсувом уперед, а для просторової похідної – дво- точкову апроксимацію із центральною різницею (шаблон FTCS – уперед за часом, центральна за простором), то диференціальне рівняння (4.1) набуде вигляду

$$\frac{T_j^{n+1}-T_j^n}{\Delta t}-\frac{\alpha(T_{j-1}^n-2T_j^n+T_{j+1}^n)}{\Delta x^2}=0.$$

У (4.7) просторова похідна апроксимується на відомому n-му шарі за часом. Після деяких перетворень скінченнорізницеву апроксимацію (4.7) зведемо до алгоритму, який можна вже ви- користовувати для числових розрахунків:

$$T_j^{n+1}=sT_{j-1}^n+(1-2s)T_j^n+sT_{j+1}^n,\quad s=\alpha\cdot\Delta t/\Delta x^2$$

Головні характеристики кожної скінченнорізницевої схеми – це точність апроксимації та умови її стійкості. Для визначення точності апроксимації скінченнорізницевої схеми за шаблоном FTCS підставимо до скінченнорізницевого рівняння (4.7) точний розв'язок T , для чого розвинемо кожний член рівняння (4.7) у ряд Тейлора в околі вузла (j, n) і зведемо подібні члени:

$$\left[\frac{\partial\bar T}{\partial t}-\alpha\frac{\partial^2\bar T}{\partial x^2}\right]_j^n=\left[\alpha\left(\frac{\Delta x^2}{2}\right)\left(s-\frac{1}{6}\right)\frac{\partial^4\bar T}{\partial x^4}\right]_j^n+O\left(\Delta t^2,\Delta x^4\right).\quad(4.9)$$

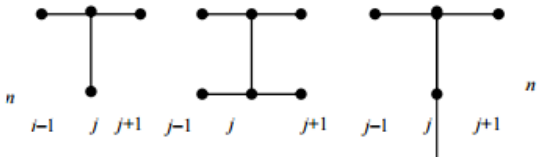
У (4.9) праворуч – головний член помилки скінченнорізнице- вої апроксимації (ГЧП) рівняння (4.1) за шаблоном FTCS. Отже, згідно з (4.9), помилка апроксимації тут ≈ O(Δt, Δ x 2). Звернемо увагу на те, що для цього випадку помилку апроксимації можна знайти й простіше, виходячи з оцінок апроксимації окремих похі- дних під час заміни диференціальних операторів скінченнорізни- цевими (за (2.7)–(2.10)), або використовуючи інші, уже розглянуті в розд. 2 процедури заміни диференціального рівняння сітковим. Водночас використання диференціального наближення скінчен- норізницевої схеми дозволяє передбачити деякі суттєві особливо- сті її поведінки. Наприклад, при виборі s = 1/6 в (4.9) ГЧП оберта- ється на нуль, і точність апроксимації становить уже ≈ O(Δt 2 , Δ x 4). Проведений аналіз стійкості схеми FTCS за Нейма- ном указує на умовну стійкість схеми при s ≤ 0,5 [1]. При побудові алгоритму схеми FTCS для апроксимації похід- ної за часом використано односторонню різницеву формулу, що дає тільки перший порядок точності. Природно, що при викори- станні центральної різницевої формули для апроксимації похід- ної за часом для рівняння

$$\frac{T_j^{n+1}-T_j^{n-1}}{2\Delta t}-\frac{\alpha(T_{j-1}^n-2T_j^n+T_{j+1}^n)}{\Delta x^2}=0$$

загальна точність апроксимації мала б покращитися. Але засно- вана на цьому шаблоні схема Річардсона (див. рис. 3.2) при ана- лізі її стійкості за Нейманом виявляється безумовно нестійкою [1].

4.1.2. Розв'язання задачі Коші. Неявні схеми

При використанні неявних схем для розв'язання задачі Коші (4.1)–(4.5) просторовий член у (4.1) апроксимується, за меншою 93 мірою частково, на невідомому (n + 1)-му шарі за часом (рис. 4.3). На практиці це спричиняє взаємозв'язок рівнянь для кожного з вузлів на цьому шарі і, як наслідок, необхідність роз- в'язання на кожному кроці за часом системи лінійних алгебраїч- них рівнянь.



Якщо для похідної за часом увести двоточкову різницеву ап- роксимацію зі зсувом назад, а для просторової похідної на неві- домому шарі – двоточкову апроксимацію із центральною різни- цею (шаблон FI – чисто неявна схема), диференціальне рівняння (4.1) набуде вигляду

$$\frac{T_j^{n+1}-T_j^n}{\Delta t}-\frac{\alpha\left(T_{j-1}^{n+1}-2T_j^{n+1}+T_{j+1}^{n+1}\right)}{\Delta x^2}=0.$$

Алгоритм FI тут запишемо як

$$-sT_{j-1}^{n+1}+(1+2s)T_j^{n+1}-sT_{j+1}^{n+1}=T_j^n,$$

де s = α · Δt/Δx**2 . Головний член похибки диференціального наближення скінченнорізницевої схеми G n j для FI має вигляд

$$G_j^n=-\frac{\Delta t}{2}\left(1+\frac{1}{6s}\right)\left[\frac{\partial^2\bar T}{\partial t^2}\right]_j^n+O(\Delta t^2,\Delta x^4).$$

Отриманий вираз для похибки апроксимації у (4.16) має той самий порядок, що й для явної схеми FTCS при s ≠ 1/6, незва- жаючи на те, що сталий множник тут дещо більший. Аналізуючи стійкість за Нейманом, легко побачити, що схе- ма FI безумовно стійка. Це виявляє її очевидну перевагу перед умовно стійкими явними схемами [1].

Як було зазначено, для переходу до наступного шару за ча- сом для FI потрібно розв'язати методом прогонки (TDMA) сис- тему лінійних алгебраїчних рівнянь (4.15) із тридіагональною стрічковою матрицею. На практиці це потребує вдвічі більше комп'ютерного часу, ніж для розв'язання тієї самої задачі явним методом FTCS. Але крок за часом для неявної схеми може бути суттєво більш