## +Понятие и необходимость параллельного программирования. Конвейеризация, векторизация и параллелизм. Примеры параллельных вычислительных систем: суперкомпьютеры, кластеры.

Термин параллельное программирование означает достаточно широкую область, которая связана с организацией расчетов на вычислительных системах, состоящих из нескольких процессорных устройств.

Необходимость параллельного программирования

• Приложения требуют увеличения производительности компьютеров.

• Производительность процессора и памяти ограничена физическими характеристиками применяемых материалов.

• Задачи часто содержат независимые компоненты, которые могут решаться одновременно (т.е. параллельно).

Параллельные вычисления применяются в областях, связанных с проведением больших расчетов:

• системах поддержки проектирования (CAD – Computer Aided Design);

• инженерных приложениях;

• математическом моделировании физических процессов;

• моделирование глобальных процессов в науке о Земле;

• вычислительной химии;

• бизнес-приложениях.

**параллели́зм** — **это** свойство систем, при котором несколько вычислений выполняются одновременно, и при этом, возможно, взаимодействуют друг с другом.

Для повышения скорости выполнения команд был придуман конвейер. Команда в процессе выполнения проходит целую цепь этапов. Каждое звено цепи обслуживается своим аппаратным обеспечением и процессор ждет, пока команда пройдет всю цепь, а затем загружает новую. Смысл конвейера заключается в том, чтобы аппаратное обеспечение не простаивало, а постоянно работало. Это достигается тем, что после выполнения команды и передачи ее далее, звено принимает и выполняет следующую. Таким образом происходит частичное временное перекрытие выполнения нескольких команд, которое зависит от количества звеньев конвейера. Такой способ распараллеливания операций называется **конвейеризацией**.

При обработке массивов данных, состоящих из элементов одинаковой структуры, называемых векторами, производятся одинаковые операции над этими элементами. Для повышения скорости обработки этих массивов, вводятся команды процессора, которые обеспечивают параллельную обработку данных, определяющих структуру вектора. Т.е.одновременно выполняются одни и те же вычисления над различными данными. Такой способ распараллеливания операций называется **векторизацией** и часто используется совместно с конвейерной обработкой.

PVP (Parallel Vector Process) – параллельная архитектура с векторными процессорами

• Основным признаком PVP-систем является наличие специальных векторно-конвейерных процессоров, в которых предусмотрены команды однотипной обработки векторов независимых данных, эффективно выполняющиеся на конвейерных функциональных устройствах.   
• Как правило, несколько таких процессоров (1-16) работают одновременно с общей памятью (аналогично SMP) в рамках многопроцессорных конфигураций. Несколько узлов могут быть объединены с помощью коммутатора (аналогично MPP). Поскольку передача данных в векторном формате осуществляется намного быстрее, чем в скалярном (максимальная скорость может составлять 64 Гбайт/с, что на 2 порядка быстрее, чем в скалярных машинах), то проблема взаимодействия между потоками данных при распараллеливании становится несущественной.

Парадигма программирования на PVP-системах предусматривает векторизацию циклов (для достижения разумной производительности одного процессора) и их распараллеливание (для одновременной загрузки нескольких процессоров одним приложением).

• На практике рекомендуется выполнять следующие процедуры:

1. Производить векторизацию вручную, чтобы перевести задачу в матричную форму. При этом, в соответствии с длиной вектора, размеры матрицы должны быть кратны 128 или 256;

2. Работать с векторами в виртуальном пространстве, разлагая искомую функцию в ряд и оставляя число членов ряда, кратное 128 или 256.

Суперкомпьютер – специализированная вычислительная машина, значительно превосходящая по своим техническим параметрам и скорости вычислений большинство существующих в мире компьютеров.

Как правило, современные суперкомпьютеры представляют собой большое число высокопроизводительных серверных компьютеров, соединенных друг с другом локальной высокоскоростной магистралью для достижения максимальной производительности в рамках подхода распараллеливания вычислительной задачи.

В общеупотребительный лексикон термин «суперкомпьютер» вошёл благодаря распространенности компьютерных систем Сеймура Крэя:

• CDC 6600 (1963)

• CDC 7600 (1969)

• Cray-1 (1975)

• Cray-2 (1985)

• Cray-3 (1988)

• Cray-4 (1994)

Кластер – группа компьютеров, объединённых высокоскоростными каналами связи, представляющая с точки зрения пользователя единый аппаратный ресурс.

Кластер – слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений, и представляющихся пользователю единой системой.

Согласно одному из первых архитекторов кластерной технологии Грегори Пфистеру (Gregory Pfister):

«Кластер — это разновидность параллельной или распределенной системы, которая:

• состоит из нескольких связанных между собой компьютеров;

• используется как единый, унифицированный компьютерный ресурс».

Обычно различают следующие основные виды кластеров:

• отказоустойчивые кластеры (High-availability clusters, HA, кластеры высокой доступности),

• кластеры с балансировкой нагрузки (Load balancing clusters),

• вычислительные кластеры (High performance computing clusters, HPC)

• системы распределенных вычислений.

## +Классификация вычислительных систем: классификация Флинна. Архитектура многопроцессорных вычислительных систем: симметричная мультипроцессорность, одновременная многопотоковость, многоядерность, массивный параллелизм.

Изображение выглядит как стол

Автоматически созданное описание

SISD (Single Instruction Single Data): это обычные последовательные компьютеры. Программа принимает один поток данных и выполняет один поток инструкций по обработке этих данных. Иными словами, инструкции выполняются последовательно, и каждая инструкция оперирует минимальным количеством данных (например, сложение двух чисел).

MISD (Multiple Instruction Single Data): разные потоки инструкций выполняются с одними и теми же данными. Обычно такие системы не приводят к ускорению вычислений, так как разные инструкции оперируют одними и теми же данными, в результате на выходе системы получается один поток данных. • К таким системам относят различные системы дублирования и защиты от сбоев, когда, например, несколько процессоров дублируют вычисления друг друга для надёжности. Иногда к этой категории относят конвейерные архитектуры. Среди процессоров производства Intel, конвейер присутствует начиная с процессора Pentium.

SIMD (Single Instruction Multiple Data): один поток инструкций выполняет вычисления одновременно с разными данными. Например, выполняется сложение одновременно восьми пар чисел. Такие компьютеры называются векторными, так как подобные операции выполняются аналогично операциям с векторами (когда, например, сложение двух векторов означает одновременное сложение всех их компонентов). Зачастую векторные инструкции присутствуют в дополнение к обычным «скалярным» инструкциям, и называются SIMDрасширением (или векторным расширением). • Примеры популярных SIMD-расширений: MMX, 3DNow!, SSE и др.

MIMD (Multiple Instruction Multiple Data): разные потоки инструкций оперируют различными данными. Это системы наиболее общего вида, поэтому их проще всего использовать для решения различных параллельных задач. • MIMD-системы, в свою очередь, принято разделять (классификация Джонсона) на системы с общей памятью (несколько вычислителей имеют общую память) и системы с распределенной памятью (каждый вычислитель имеет свою память; вычислители могут обмениваться данными). Кроме того, существуют системы с неоднородным доступом к памяти (NUMA) — в которых доступ к памяти других вычислителей существует, но он значительно медленнее, чем доступ к «своей» памяти.

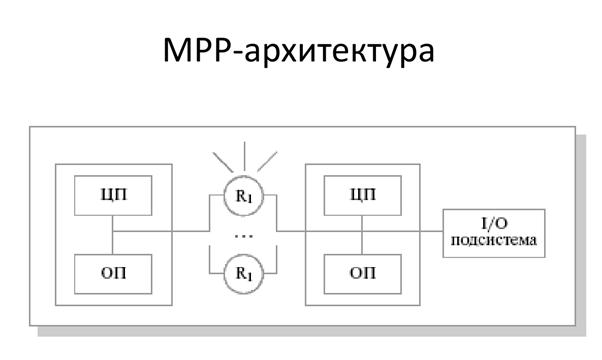
Архитектура многопроцессорных вычислительных систем: симметричная мультипроцессорность, одновременная многопотоковость, многоядерность, массивный параллелизм:

Изображение выглядит как текст

Автоматически созданное описание

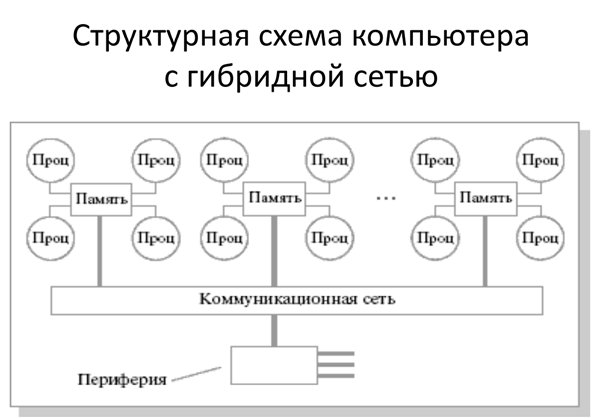
Основные преимущества SMP-систем: • простота и универсальность для программирования. Архитектура SMP не накладывает ограничений на модель программирования, используемую при создании приложения: обычно используется модель параллельных ветвей, когда все процессоры работают независимо друг от друга. Однако можно реализовать и модели, использующие межпроцессорный обмен. Использование общей памяти увеличивает скорость такого обмена, пользователь также имеет доступ сразу ко всему объему памяти. Для SMP-систем существуют довольно эффективные средства автоматического распараллеливания; • простота эксплуатации. Как правило, SMP-системы используют систему кондиционирования, основанную на воздушном охлаждении, что облегчает их техническое обслуживание; • относительно невысокая цена. Недостатки: • системы с общей памятью плохо масштабируются.

MPP архитектура (massive parallel processing) - массивно-параллельная архитектура. • Главная особенность такой архитектуры состоит в том, что память физически разделена. В этом случае система строится из отдельных модулей, содержащих процессор, локальный банк операционной памяти (ОП), два коммуникационных процессора (рутера) или сетевой адаптер, иногда - жесткие диски и/или другие устройства ввода/вывода. Один рутер используется для передачи команд, другой - для передачи данных. По сути, такие модули представляют собой полнофункциональные компьютеры.



MPP-архитектура • Системы с распределенной памятью идеально подходят для параллельного выполнения независимых программ, поскольку при этом каждая программа выполняется на своем узле и никаким образом не влияет на выполнение других программ. • Однако при разработке параллельных программ приходится учитывать более сложную, чем в SMP системах, организацию памяти. Оперативная память в MPP системах имеет 3-х уровневую структуру: • кэш-память процессора; • локальная оперативная память узла; • оперативная память других узлов.

Гибридная архитектура NUMA • Главная особенность гибридной архитектуры NUMA (nonuniform memory access) – неоднородный доступ к памяти . • Гибридная архитектура совмещает достоинства систем с общей памятью и относительную дешевизну систем с раздельной памятью. Суть этой архитектуры – в особой организации памяти, а именно: память физически распределена по различным частям системы, но логически она является общей, так что пользователь видит единое адресное пространство. Система построена из однородных базовых модулей (плат), состоящих из небольшого числа процессоров и блока памяти. Модули объединены с помощью высокоскоростного коммутатора.

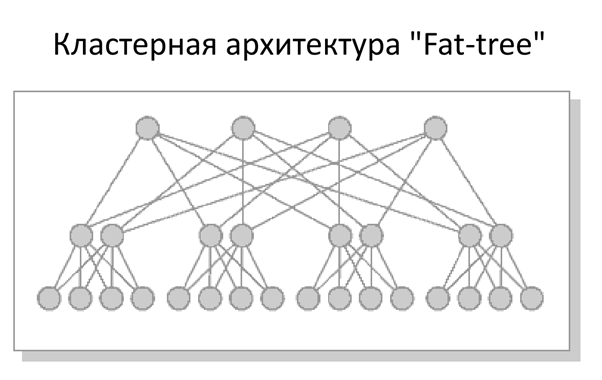


PVP-архитектура • PVP (Parallel Vector Process) – параллельная архитектура с векторными процессорами • Основным признаком PVP-систем является наличие специальных векторно-конвейерных процессоров, в которых предусмотрены команды однотипной обработки векторов независимых данных, эффективно выполняющиеся на конвейерных функциональных устройствах. • Как правило, несколько таких процессоров (1-16) работают одновременно с общей памятью (аналогично SMP) в рамках многопроцессорных конфигураций. Несколько узлов могут быть объединены с помощью коммутатора (аналогично MPP). Поскольку передача данных в векторном формате осуществляется намного быстрее, чем в скалярном (максимальная скорость может составлять 64 Гбайт/с, что на 2 порядка быстрее, чем в скалярных машинах), то проблема взаимодействия между потоками данных при распараллеливании становится несущественной.

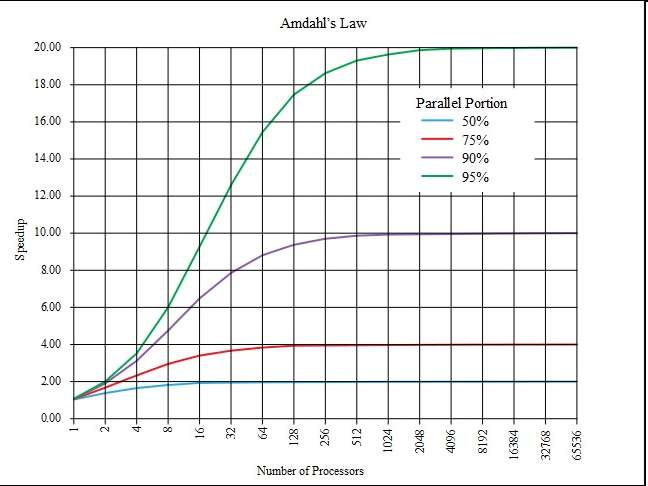
Примеры машин PVP-архитектуры: • CRAY X1, SMP -архитектура. Пиковая производительность системы в стандартной конфигурации может составлять десятки терафлопс.

Парадигма программирования на PVP-системах предусматривает векторизацию циклов (для достижения разумной производительности одного процессора) и их распараллеливание (для одновременной загрузки нескольких процессоров одним приложением). • На практике рекомендуется выполнять следующие процедуры: 1. Производить векторизацию вручную, чтобы перевести задачу в матричную форму. При этом, в соответствии с длиной вектора, размеры матрицы должны быть кратны 128 или 256; 2. Работать с векторами в виртуальном пространстве, разлагая искомую функцию в ряд и оставляя число членов ряда, кратное 128 или 256.

Архитектура «Fat-tree» • Наиболее эффективной является архитектура с топологией "толстого дерева" (fat-tree). • Архитектура "fat-tree" (hypertree) была предложена Лейзерсоном (Leiserson) в 1985 году. • Процессоры локализованы в листьях дерева, в то время как внутренние узлы дерева скомпонованы во внутреннюю сеть. • Поддеревья могут общаться между собой, не затрагивая более высоких уровней сети.



## +Оценка максимально достижимого параллелизма (закон Амдала, закон Густавсона-Барсиса). Анализ масштабируемости параллельных вычислений.

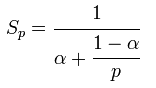


• В случае, когда задача разделяется на несколько частей, суммарное время ее выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента, при условии одинаковой скорости всех вычислителей (Дж.Амдал, 1967 г.).

• Согласно этому закону, ускорение выполнения программы за счет распараллеливания её инструкций на множестве вычислителей ограничено временем, необходимым для выполнения ее последовательных инструкций.

• Предположим, что её алгоритм решения задачи таков, что доля от общего объёма вычислений может быть получена только последовательными расчетами, а, соответственно, доля 1 – может быть распараллелена идеально (то есть время

вычисления будет обратно пропорционально числу задействованных узлов).

• Тогда ускорение, которое может быть получено на вычислительной системе из p процессоров, по сравнению с однопроцессорным решением не будет превышать величины 

• Закон Амдала показывает, что прирост эффективности вычислений зависит от

алгоритма задачи и ограничен сверху для любой задачи с .

• Не для всякой задачи имеет смысл наращивание числа процессоров в вычислительной системе.

• Более того, если учесть время, необходимое для передачи данных между узлами вычислительной системы, то зависимость времени вычислений от числа узлов будет

иметь максимум.

• Это означает, что с определенного момента добавление новых узлов в систему будет увеличивать время расчета задачи.

Изображение выглядит как стол

Автоматически созданное описание

**Закон Густавсона-Барсиса** — оценка максимально достижимого ускорения выполнения параллельной программы, в зависимости от количества одновременно выполняемых потоков вычислений («процессоров») и доли последовательных расчетов. Аналог закона Амдала: Джон Густафсон и Эдвин Барсис представили статью «Переоценка закона Амдала» в 1988 году.

Закон Густавсона-Барсиса выражается формулой: , где

s — доля последовательных расчетов в программе,

n — количество процессоров.

Данную оценку ускорения называют ускорением масштабирования, так как данная характеристика показывает, насколько эффективно могут быть организованы параллельные вычисления при увеличении сложности решаемых задач.

Отличие от закона Амдала:

При оценке ускорения параллельного выполнения закон Амдала предполагает, что объем задачи остается постоянным. Величина ускорения по закону Амдала показывает, во сколько раз меньше времени потребуется параллельной программе для выполнения. Однако величину ускорения можно рассматривать и как увеличение объема выполненной задачи за постоянный промежуток времени. Закон Густафсона появился именно из этого предположения.Изображение выглядит как текст, часы

Автоматически созданное описаниеИзображение выглядит как текст, часы, коллекция картинок

Автоматически созданное описание

Закон Амдала предполагает, что объем задачи остается неизменным. Задача слева обрабатывается одним потоком, задача справа — тремя. Сравнивается время выполнения.

Закон Густавсона предполагает, что время, отведенное на выполнение задачи, остается неизменным. Слева работает один поток, справа — три. Объем выполненной работы увеличен. Сравнивается объем вычислений.

## +Алгоритмы параллельного произведения матриц (различные парадигмы).

• Умножение матрицы A размера mxn и матрицы B размера nxl приводит к получению матрицы С размера mxl , каждый элемент которой определяется в соответствии с выражением:

Изображение выглядит как текст

Автоматически созданное описание

• Каждый элемент результирующей матрицы С есть скалярное произведение соответствующих строки матрицы A и столбца матрицы B:



Изображение выглядит как текст

Автоматически созданное описание

• Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы С.

• При выполнении одной итерации внешнего цикла (цикла по переменной i) вычисляется одна строка результирующей матрицы

Изображение выглядит как текст

Автоматически созданное описание

• Оценка времени выполнения последовательного алгоритма умножения матриц может быть представлена следующим образом:



где - пропускная способность канала доступа к оперативной памяти (константа 64 введена для учета факта, что в случае кэш-промаха из ОП читается кэш-строка размером 64 байт).

• Если помимо пропускной способности учесть латентность памяти, модель приобретет следующий вид:



где α есть латентность оперативной памяти.

• Обе предыдущие модели являются моделями на худший случай и дают сильно завышенную оценку времени выполнения алгоритма, так как доступ к оперативной памяти происходит не при каждом обращении к элементу

• Введем в разработанную модель величину , 0 ≤ ≤ 1, для задания частоты возникновения кэш промахов. Тогда оценка времени выполнения алгоритма матричного умножения принимает вид:



Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

**Базовый параллельный алгоритм умножения матриц**

• Для проведения всех необходимых вычислений каждая подзадача должна производить вычисления над элементами одной строки матрицы А и одного столбца матрицы В.

• Общее количество получаемых при таком подходе подзадач оказывается равным n^2 (по числу элементов матрицы С).

• Для вычисления одной строки матрицы С необходимо, чтобы в каждой подзадаче содержалась строка матрицы А и был обеспечен доступ ко всем столбцам матрицы B.

• Схема организации вычислений при выполнении параллельного алгоритма умножения матриц, основанного на разделении матриц по строкам:

Изображение выглядит как текст, кроссворд

Автоматически созданное описание

• Данный параллельный алгоритм обладает хорошей «локальностью вычислений». Это означает, что данные, которые обрабатывает один из потоков параллельной программы, не изменяются другим потоком.

• Нет взаимодействия между потоками, нет необходимости в синхронизации.

Изображение выглядит как текст

Автоматически созданное описание

• Время работы с оперативной памятью



• Время выполнения параллельного алгоритма составляет



• при введении в модель показателя частоты кэш промахов:



Изображение выглядит как текст

Автоматически созданное описание

**Алгоритм умножения матриц, основанный на ленточном разделении данных**

• Ленточная схема разделения данных применяется и для второй перемножаемой матрицы B.

• Такой подход, в частности, позволяет улучшить локализацию данных в потоках и повысить эффективность использования кэша.

• В качестве базовой подзадачи будем рассматривать процедуру определения одного элемента результирующей матрицы С.

• Но тогда количество базовых подзадач существенно превышает число доступных вычислительных элементов.

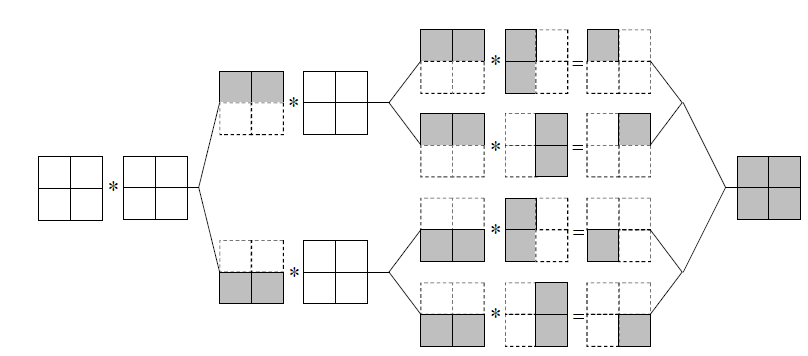
• Определим базовую подзадачу как процедуру вычисления всех элементов прямоугольного блока матрицы С.

Изображение выглядит как объект, часы, антенна

Автоматически созданное описание

• Организация параллельных вычислений при выполнении параллельного алгоритма

умножения матриц, основанного на ленточном разделении данных, и использованием четырех потоков











Изображение выглядит как текст

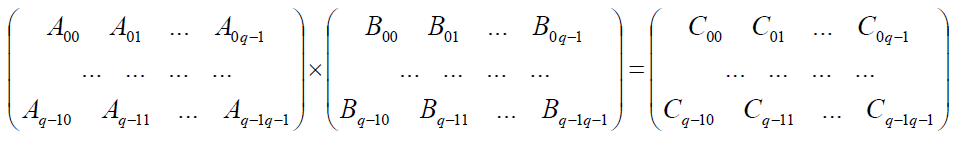
Автоматически созданное описание

**Блочный алгоритм умножения матриц**

• В этом случае не только результирующая матрица, но и матрицы-аргументы матричного умножения разделяются между потоками параллельной программы на прямоугольные блоки.

• Такой подход позволяет добиться большей локализации данных и повысить эффективность использования кэш памяти.

• Исходные матрицы А, В и результирующая матрица С представляются в виде наборов блоков.



• где каждый блок Cij матрицы C определяется в соответствии с выражением

Изображение выглядит как текст, часы

Автоматически созданное описание

• При блочном разбиении данных для определения базовых подзадач естественным представляется взять за основу вычисления, выполняемые над матричными блоками.

• Для выполнения всех необходимых вычислений базовым подзадачам должны

быть доступны соответствующие наборы строк матрицы A и столбцов матрицы B.

Изображение выглядит как кроссворд, текст

Автоматически созданное описание

• Блочное разбиение полос вносит изменение лишь в порядок выполнения вычислений, общий же объем вычислительных операций при этом не изменяется.



• На каждой итерации алгоритма каждый поток выполняет умножение двух матричных блоков размером (n/q)×(n/q).



• Общее число потоков – q^2.

• Таким образом, время считывания данных из оперативной памяти в кэш для блочного алгоритма умножения матриц составляет:



• Время, необходимое на чтение данных из оперативной памяти в кэш составляет:





Изображение выглядит как текст

Автоматически созданное описание

**Блочный алгоритм, эффективно использующий кэш-память**

• Значительная доля времени умножения матриц тратится на многократное чтение элементов матриц A и B из оперативной памяти в кэш.

• Для организации алгоритма умножения матриц, который более эффективно использует кэш-память, воспользуемся разделением матриц на блоки.

• Разделение на блоки следует проводить таким образом, чтобы размер блока был настолько мал, что три блока, участвующие в умножении на данной итерации, возможно было поместить в кэш.

• Количество разбиений матриц теперь определяется не количеством потоков, а объемом кэш памяти.

• Пусть – объем кэш в байтах.

• Максимальный размер квадратного k×k матричного блока составляет: 

• Минимально необходимое число разбиений GridSize при разделении матриц размером n×n составляет:



• Для снижения сложности программной реализации алгоритма количество блоков по горизонтали и вертикали GridSize должно быть таким, чтобы размер матриц n мог быть поделен на GridSize без остатка.

Изображение выглядит как текст

Автоматически созданное описание

**Параллельный алгоритм**

• Базовая подзадача (поток) отвечает за вычисление блока результирующей матрицы.

• Однако теперь, когда количество блоков определяется не количеством потоков, а объемом кэш памяти, число блоков может существенно превосходить число доступных потоков.

• Проведем агрегацию вычислений – пусть каждый поток обеспечивает получение несколько матричных блоков результирующей матрицы C.

• При умножении матриц с помощью рассмотренного алгоритма выполняется q^3 операций умножения матричных блоков.

• При этом, размер блоков определяется таким образом, чтобы на каждой итерации они могли быть одновременно помещены в кэш память.





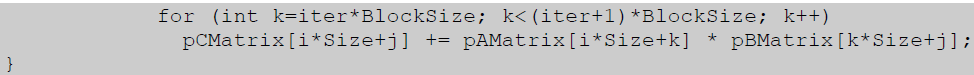
• Общее время выполнения параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш, может быть вычислено по формуле:

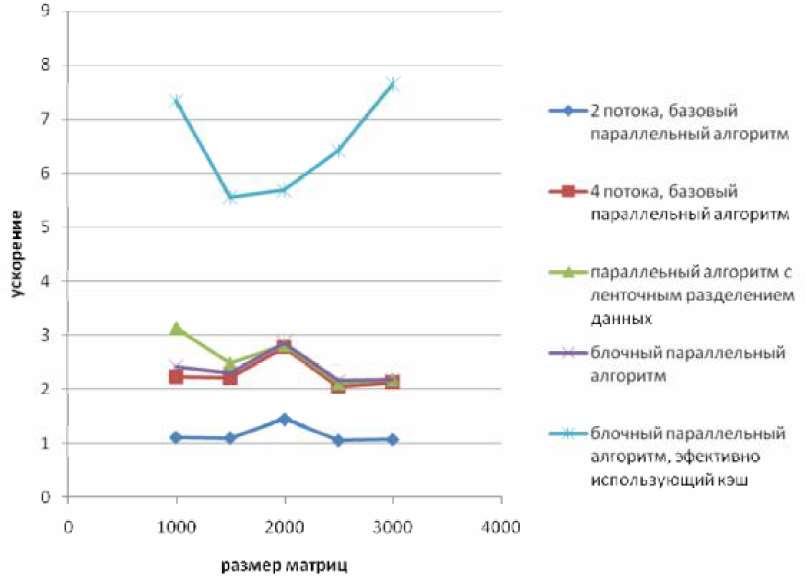


• Если дополнительно учесть частоту кэш промахов, то выражение приобретает следующий вид:



Изображение выглядит как текст

Автоматически созданное описание



## 9.+Синхронизация процессов и потоков: необходимость. Примитивы синхронизации: семафоры, мьютексы, условные переменные, барьеры, мониторы. Классические задачи синхронизации: «производители-потребители», «читатели-писатели».

• Под семафором S понимается переменная особого типа, значение которой может опрашиваться и изменяться только при помощи специальных операций P(S) и V(S).

• Принципиальным в понимании семафоров является то, что операции P(S) и V(S) предполагаются неделимыми (атомарными), что гарантирует взаимоисключение при использование общих семафоров (для обеспечения неделимости операции обслуживания семафоров обычно реализуются средствами операционной системы).

• Различают два основных типа семафоров. Двоичные семафоры принимают только значения 0 и 1, область значений общих семафоров – неотрицательные целые значения.

• В момент создания семафоры инициализируются некоторым целым значением.

Операции над семафором

• 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

• **P-операция** над семафором представляет собой попытку декремента значения семафора на 1. Если перед выполнением P-операции значение семафора было больше 0, то P-операция выполняется без задержек. Если перед выполнением P-операции значение семафора было равным 0, то процесс, выполняющий P-операцию, переводится в состояние ожидания до тех пор, пока значение семафора не станет большим 0.

• **V-операция** над семафором представляет инкремент значения семафора на 1. Если при этом имеются процессы, задержанные на выполнении P-операции на данном семафоре, один из этих процессов выходит из состояния ожидания и может выполнить свою P-операцию.

Взаимное исключение на семафоре

• Для реализации взаимного исключения, например, предотвращения возможности одновременного изменения двумя или более процессами общих данных, создается двоичный семафор S.

• Начальное значение этого семафора = 1.

• Критические секции кода обрамляются «скобками» P(S) (в начале секции) и V(S) (в конце секции).

• Процесс, входящий в критическую секцию, выполняет операцию P(S) и переводит семафор в 0.

• Если в критической секции уже находится другой процесс, то значение семафора уже 0, тогда второй процесс, желающий войти в критическую секцию, блокируется в своей P-операции до тех пор, пока процесс, находящийся сейчас в критической секции, не выйдет из нее, выполнив на выходе операцию V(S).

Изображение выглядит как текст

Автоматически созданное описание

• На практике в разных средах выполнения наряду с поддержкой "стандартных" семафоров реализуются некоторые их разновидности – **мьютексы (mutex)**, **замки (lock)** и др.

• Вариации касаются допустимого набора значений для переменных семафоров, рекурсивности вызова, введения дополнительного набора операций и т.д.

• **Мониторы** представляют собой программные модули (объекты), которые реализуют 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 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.

Изображение выглядит как текст, птица, растение

Автоматически созданное описание

Свойства монитора:

• Переменные монитора недоступны вне монитора и могут обрабатываться только процедурами монитора;

• Вне монитора доступны только процедуры монитора;

• Переменные, объявленные вне монитора, недоступны внутри монитора.

**Мьютекс** — примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода. Классический мьютекс отличается от двоичного семафора наличием эксклюзивного владельца, который и должен его освобождать (т.е. переводить в незаблокированное состояние). Встречаются также блокировки чтения-записи, именуемые разделяемыми мьютексами и предоставляющие помимо эксклюзивной блокировки общую, позволяющую совместно владеть мьютексом, если нет эксклюзивного владельца.

Условно классический мьютекс можно представить в виде переменной, которая может находиться в двух состояниях: в заблокированном и в незаблокированном. При входе в свою критическую секцию поток вызывает функцию перевода мьютекса в заблокированное состояние, при этом поток блокируется до освобождения мьютекса, если другой поток уже владеет им. При выходе из критической секции поток вызывает функцию перевода мьютекса в незаблокированное состояние. В случае наличия нескольких заблокированных по мьютексу потоков во время разблокировки планировщик выбирает поток для возобновления выполнения.

Задачей мьютекса является защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, защищенным мьютексом. Если другому потоку будет нужен доступ к данным, защищенным мьютексом, то этот поток блокируется до тех пор, пока мьютекс не будет освобожден. Мьютекс защищает данные от повреждения в результате асинхронных изменений (состояние гонки), однако при неправильном использовании могут порождаться другие проблемы, например, взаимная блокировка или двойной захват.

**Условная переменная** — примитив синхронизации, обеспечивающий блокирование одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания. Условные переменные используются вместе с ассоциированным мьютексом и являются элементом некоторых видов мониторов.

Концептуально, условная переменная — это очередь потоков, ассоциированных с разделяемым объектом данных, которые ожидают выполнения некоторого условия, накладываемого на состояние данных. Таким образом, каждая условная переменная  связана с утверждением . Когда поток находится в состоянии ожидания на условной переменной, он не считается владеющим данными и другой поток может изменить разделяемый объект и просигнализировать ожидающим потокам в случае выполнения утверждения .

Пример

#include *<cstdlib>*

#include *<iostream>*

#include *<thread>*

#include *<mutex>*

#include *<condition\_variable>*

#include *<chrono>*

#define STORAGE\_MIN 10

#define STORAGE\_MAX 20

int storage = STORAGE\_MIN;

std::mutex globalMutex;

std::condition\_variable condition;

*/\* Функция потока потребителя \*/*

void consumer()

{

std::cout << "[CONSUMER] thread started" << std::endl;

int toConsume = 0;

**while**(true)

{

std::unique\_lock<std::mutex> lock(globalMutex);

*/\* Если значение общей переменной меньше максимального,*

*\* то поток входит в состояние ожидания сигнала о достижении*

*\* максимума \*/*

**if** (storage < STORAGE\_MAX)

{

condition.wait(lock , []{**return** storage >= STORAGE\_MAX;} ); *// Атомарно \_отпускает мьютекс\_ и сразу же блокирует поток*

toConsume = storage-STORAGE\_MIN;

std::cout << "[CONSUMER] storage is maximum, consuming "

<< toConsume << std::endl;

}

*/\* "Потребление" допустимого объема из значения общей*

*\* переменной \*/*

storage -= toConsume;

std::cout << "[CONSUMER] storage = " << storage << std::endl;

}

}

*/\* Функция потока производителя \*/*

void producer()

{

std::cout << "[PRODUCER] thread started" << std::endl;

**while** (true)

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(200));

std::unique\_lock<std::mutex> lock(globalMutex);

++storage;

std::cout << "[PRODUCER] storage = " << storage << std::endl;

*/\* Если значение общей переменной достигло или превысило*

*\* максимум, поток потребитель уведомляется об этом \*/*

**if** (storage >= STORAGE\_MAX)

{

std::cout << "[PRODUCER] storage maximum" << std::endl;

condition.notify\_one();

}

}

}

int main(int argc, char \*argv[])

{

std::**thread** thProducer(producer);

std::**thread** thConsumer(consumer);

thProducer.join();

thConsumer.join();

**return** 0;

}

**Барьерная синхронизация** — метод синхронизации в распределенных вычислениях, при котором выполнение параллельного алгоритма или его части можно разделить на несколько этапов, разделенных барьерами. В частности, с помощью барьера можно организовать точку сбора частичных результатов вычислений, в которой подводится итог этапа вычислений. Использование барьеров как примитивов синхронизации особенно полезно при циклической организации этапов.

**Барьер** для группы потоков (или процессов) в исходном коде означает, что каждый поток (процесс) должен остановиться в этой точке и подождать достижения барьера всеми потоками (процессами) группы. Когда все потоки (процессы) достигли барьера, их выполнение продолжается.

Некоторые среды параллельных вычислений, например, OpenMP и Cilk, имеют неявные барьеры в семантике своих параллельных циклов и блоков кода.

**Задача «Производители–Потребители»**

• В наиболее простом случае предполагается, что существует два потока, один из которых (производитель) генерирует сообщения (изделия), а второй поток (потребитель) их принимает для последующей обработки.

• Потоки взаимодействуют через некоторую область памяти (хранилище), в которой производитель размещает свои генерируемые сообщения и из которой эти сообщения извлекаются потребителем

Изображение выглядит как текст

Автоматически созданное описание

• Хранилище сообщений представляет собой общий разделяемый ресурс, и использование этого ресурса должно быть построено по правилам взаимоисключения.

• Кроме того, следует учитывать, что потребление ресурса иногда может оказаться невозможным (отсутствие сообщений в хранилище), а при добавлении сообщений в хранилище могут происходить задержки (в случае полного заполнения хранилища).

**Решение (один из вариантов)**

• Для организации работы используем три семафора:

• **Access** – двоичный семафор для организации взаимоисключения при доступе к хранилищу;

• **Full** – общий семафор, блокирующий поток-производитель при попытке записи сообщения в полностью заполненное хранилище (в переменной семафора будет хранится количество имеющихся свободных мест для сообщений в хранилище);

• **Empty** – общий семафор, блокирующий поток-потребитель при попытке чтения сообщения из пустого хранилище (в переменной семафора будет хранится количество имеющихся сообщений в хранилище).

Изображение выглядит как текст

Автоматически созданное описание

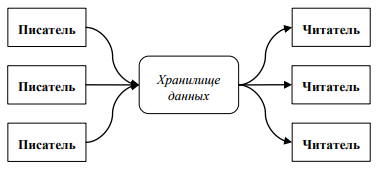
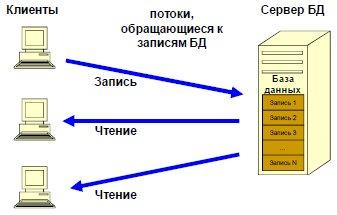
**Задача «Читатели–Писатели»**

• Пусть имеется некоторая область памяти – хранилище данных, с которым одновременно работают несколько потоков.

• По характеру использования хранилища потоки могут быть разделены на два типа.

• Одна группа – это **потоки-читатели**, которые осуществляют только чтение (без удаления) хранилища.

• Другая – оставшаяся часть потоков – это **потоки-писатели**, которые выполняют запись новых значений имеющихся в хранилище данных

• При рассмотрении этой задачи предполагается, что потоки-читатели не изменяют каких-либо параметров хранилища и могут, тем самым, работать одновременно, не мешая друг другу.

• Для потоков-писателей ситуация обратная – предполагается, что запись не может выполняться несколькими потоками одновременно и, понятно, во время записи не допускаются какие-либо операции чтения.

• Постановка задачи может различаться в правилах разрешения ситуации обращения потока-писателя к хранилищу.

• Если при попытке записи имеются активные потоки-читатели, то поток-писатель должен быть заблокирован до завершения работы потоков-читателей. Вопрос состоит в том, что делать с новыми поступающими запросами на чтение.

• Возможный вариант – отдать предпочтение потокам-читателям (т. е. новые потоки-читатели могут начинать свою работу, не обращая внимания на блокированный процесс-писатель), однако в этом случае блокировка потока-писателя может продолжаться бесконечно долго.

• Альтернативный подход – блокировка новых потоков-читателей, появившихся после блокировки потока-писателя.

Изображение выглядит как текст

Автоматически созданное описание

**Решение**

• **ReadCount** – переменная-счетчик количества активных потоков-читателей;

• **ReadSem** – двоичный семафор для взаимоисключения доступа к переменной ReadCount;

• **Access** – двоичный семафор для организации взаимоисключения при доступе к хранилищу.

Изображение выглядит как текст

Автоматически созданное описание

## 12.+Технология OpenMP. Основные особенности. Преимущества от использования. Принципы организации параллелизма. Директивы OpenMP (parallel, for, sections). Основные функции OpenMP.

OpenMP (Open Multi-Processing) – API, предназначенное для программирования многопоточных приложений для систем с общей памятью (для языков C, C++, Fortran)

основной поток создает потоки по мере необходимости один код как для последовательной, так и для параллельной версий

OpenMP – набор параметров директивы #pragma и вспомогательных

функций. Все директивы начинаются с #pragma omp. Каждая директива может иметь несколько дополнительных атрибутов. Все функции OpenMP

начинаются с omp\_. Для использования функций

OpenMP необходимо подключить библиотеку <omp.h>

Цели OpenMP

● Быть стандартом для различных архитектур и платформ с распределенной памятью ● Дать простой, но ограниченный набор директив для параллелизации программы.

● Обеспечивать совместимость и возможность инкрементальной параллелизации программы.

● Дать возможность как для мелкозернистого распараллеливания, так и для крупнозернистого.

● Поддержка Fortran (77, 90 и 95), C, и C++

Модель с разделяемой памятью

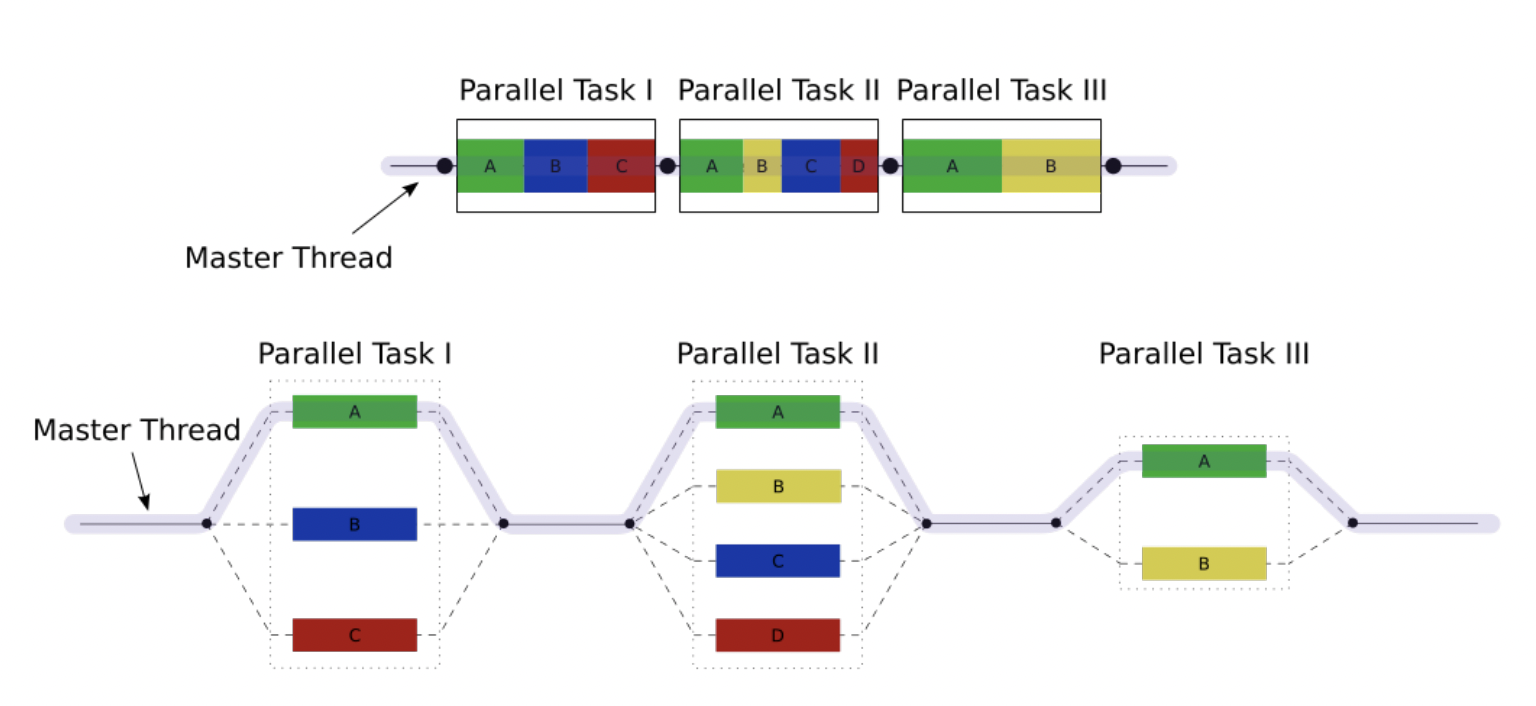
● Все потоки имеют доступ к глобальной разделяемой памяти

● Данные могут быть разделяемые и приватные

● Разделяемые данные доступны всем потокам

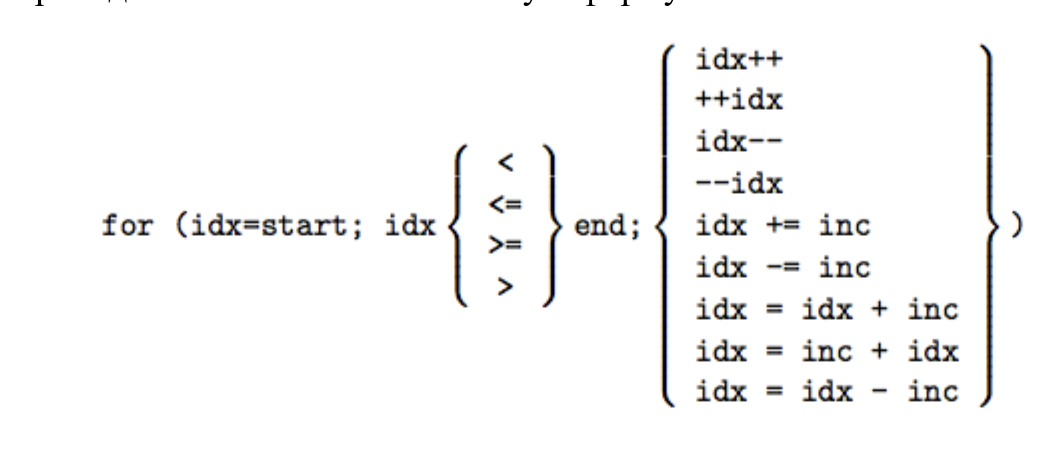
● Приватные — только одному

● Синхронизация требуется для доступа к общим данным



Директива parallel определяет параллельную область. Когда основной поток доходит до параллельной области создается группа потоков. Код параллельной области дублируется между параллельно исполняемыми потоками. Если к параллельной области относится несколько операторов, то они заключаются в фигурные скобки.

Директива for позволяет распределить между параллельно выполняемыми потоками итерации цикла, который должен следовать непосредственно после директивы. Потоки не создаются. Директива for накладывает ограничения на структуру соответствующего цикла, который должен иметь каноническую форму:



В случае, если в параллельной области находится только один цикл,

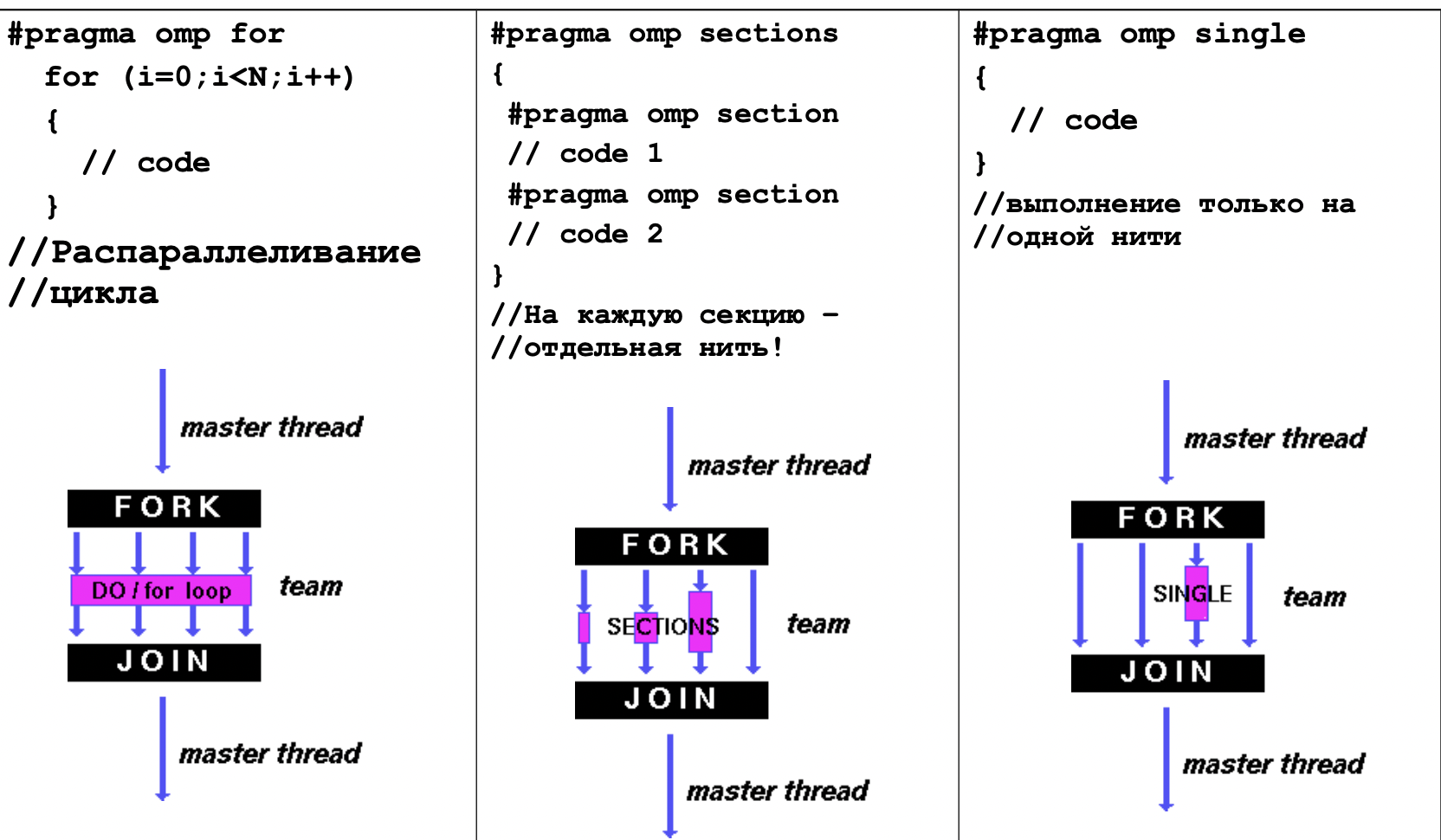
итерации которого необходимо раздать параллельно выполняемым потокам, директивы parallel и for можно объединить #pragma omp parallel for [clauses]

Директива sections позволяет реализовать параллелизм по задачам.

Потоки не создаются.

В случае, если в параллельной области находится только один цикл,

итерации которого необходимо раздать параллельно выполняемым потокам, директивы parallel и for можно объединить #pragma omp parallel sections [clauses]



Функции

Функции OpenMP носят скорее вспомогательный характер, так как реализация параллельности осуществляется за счет использования директив. Однако в ряде случаев они весьма полезны и даже необходимы. – Функции исполняющей среды – Функции блокировки/синхронизации – Функции работы с таймерами. Все эти функции имеют имена, начинающиеся с omp\_, и определены в заголовочном файле omp.h.

OpenMP поддерживает директивы private, parallel, for, section, sections, single, master, critical, flush, ordered и atomic и ряд других, которые определяют механизмы разделения работы или конструкции синхронизации.

Получение текущего момента времени

выполнения программы обеспечивается при помощи функции omp\_get\_wtime() , результат вызова которой есть количество секунд, прошедших от некоторого определенного момента времени в прошлом.

Функции omp\_get\_thread\_num()и omp\_get\_num\_threads() позволяют узнать индекс текущего потока и общее количество запущенных потоков. Поток с индексом 0 – главный поток. Именно он остается после завершения параллельной области.

случае, если нужно задать необходимое количество потоков для всей

программы, а не для конкретной параллельной области, можно использовать функцию omp\_set\_num\_threads( int num\_threads), где num\_threads – требуемое количество потоков в параллельной области.

В случае, если требуемое количество потоков не задано в программе,

значение берется из переменной окружения OMP\_NUM\_THREADS.

При необходимости определить оптимальное количество потоков можно

использовать функцию omp\_get\_num\_procs(), которая возвращает число процессоров, доступных приложению.

Функция omp\_get\_max\_threads() возвращает максимально допустимое количество потоков.

## 13.+Технология OpenMP. Синхронизация потоков в OpenMP. Директивы OpenMP (critical, atomic, barrier). Вложенный параллелизм.

В OpenMP предусмотрены следующие конструкции синхронизации: critical – критическая секция atomic – атомарность операции barrier – точка синхронизации master – блок, который будет выполнен только основным потоком. Все остальные потоки пропустят этот блок. В конце блока неявной синхронизации нет. ordered – выполнять блок в заданной последовательности flush – немедленный сброс значений разделяемых переменных в память.

Наиболее простой способ синхронизации потоков – использование

атомарных операций.

Целый набор директив в OpenMP предназначен для синхронизации работы

нитей.

Директива #pragma omp atomic определяет переменную, доступ к которой (чтение/запись) должен быть выполнен как неделимая операция . Список допустимых операций приведен ниже:

+, -, \*, /, &, ^, |, >>, << В случае, если нужно организовать более сложное взаимоисключение

потоков или операция недопустимая, то можно использовать директиву

#pragma omp critical [name],

которая определяет фрагмент кода, который должен выполняться только одним потоком в каждый текущий момент времени (критическая секция).

Для взаимоисключения потоков также можно использовать замки, которые

работают как мьютексы.

Барьер

Самый распространенный способ синхронизации в OpenMP – барьер. Он

оформляется с помощью директивы barrier.

Си:

#pragma omp barrier

Нити, выполняющие текущую параллельную область, дойдя до этой дирек-

тивы, останавливаются и ждут, пока все нити не дойдут до этой точки про-

граммы, после чего разблокируются и продолжают работать дальше. Кроме

того, для разблокировки необходимо, чтобы все синхронизируемые нити за-

вершили все порождённые ими задачи (task).

Пример: демонстрирует применение директивы barrier. Директива bar-

rier используется для упорядочивания вывода от работающих нитей. Выда-

чи с разных нитей "Сообщение 1" и "Сообщение 2" могут перемежаться в

произвольном порядке, а выдача "Сообщение 3" со всех нитей придёт строго

после двух предыдущих выдач.

Вложенный параллелизм позволяет создавать новые параллельные регионы внутри исходных. Ниже приведен пример использования вложенных параллельных регионов. Также вложенный параллелизм может потребоваться при распараллеливании рекурсивных алгоритмов.