1. **Каковы мотивы разработки многоядерных процессоров? Что обозначает термин power wall?  Как новые многопроцессорные архитектуры меняют подходы к разработке высокопроизводительных программ?**

Многоядерные процессоры работают быстрее, потому что каждое ядро может обрабатывать свой собственный поток данных. В то время как многоядерные процессоры могут выборочно кэшировать данные и получить не кэшированные данные из других хранилищ, дополнительное ядро может продолжать выполнение команды и получение информации с нормальной скоростью процессора, в то время как другой процессор извлекает необходимую информацию из медленных устройствах хранения. Таким образом, система не будет тормозить во время получения данных. Многоядерный процессор особенно ценен для многозадачности, где больше чем одна программа обслуживает свой набор данных для их обработки. Отдельные потоки данных могут обрабатываться разными ядрами, увеличивая общую скорость обработки. Для одной программы, чтобы воспользоваться преимуществами многоядерных технологий, она должна иметь технологию одновременной многопоточности (SMT), что позволяет ей направлять параллельные наборы инструкций для нескольких ядер для их использования.

В многоядерных процессорах тактовая частота, как правило, намеренно снижена. Это позволяет уменьшить энергопотребление процессора без потери производительности: энергопотребление растёт как куб от роста частоты процессора. Удвоив количество ядер процессора и снизив вдвое их тактовую частоту, можно получить практически ту же производительность, при этом энергопотребление такого процессора снизится в 4 раза.

*Power wall* относится к метафорической стене, обозначающей ограничение пиковой мощности системы.

Непрерывное масштабирование технологии (т.е. уменьшение размеров транзисторных элементов) позволяет втиснуть больше транзисторов (и, следовательно, большее количество ядер) в заданную область кристалла. Тем не менее, несмотря на пониженное напряжение питания, одновременное переключение этих транзисторных устройств вызывает огромное увеличение плотности мощности, что приводит к провалу стены питания. Является ли приближающаяся стена власти плохой? Да. Почему так? Поскольку увеличение плотности мощности неизменно увеличивает температуру микросхемы, если не используется необходимая стратегия охлаждения. Повышение температуры микросхемы, в свою очередь, замедляет скорость переключения транзисторов и, следовательно, общую скорость компьютера.

Одним из таких методов, который обычно используется в большинстве современных настольных компьютеров, является динамическое масштабирование напряжения и частоты, где рабочее напряжение и частота микросхемы динамически контролируются на основе активности микросхемы. Исследователи также продемонстрировали преимущество технологии почти-пороговых вычислений, которая использует микросхему чуть выше порогового напряжения транзисторных устройств для снижения динамического потребления энергии. Тем не менее, у каждого из этих методов есть свои недостатки, и исследователи эффективно справились с большинством из них, чтобы успешно сдерживать стену электропитания, обеспечивая при этом справедливую производительность.

…

Делают Hardware/ Software адаптируются

1. **Объясните значение терминов parallelism и concurrency. Опишите общий подход к разработке параллельных программ (слайд 19).**

‹Параллельность: состояние системы, в которой несколько задач логически активны одновременно.

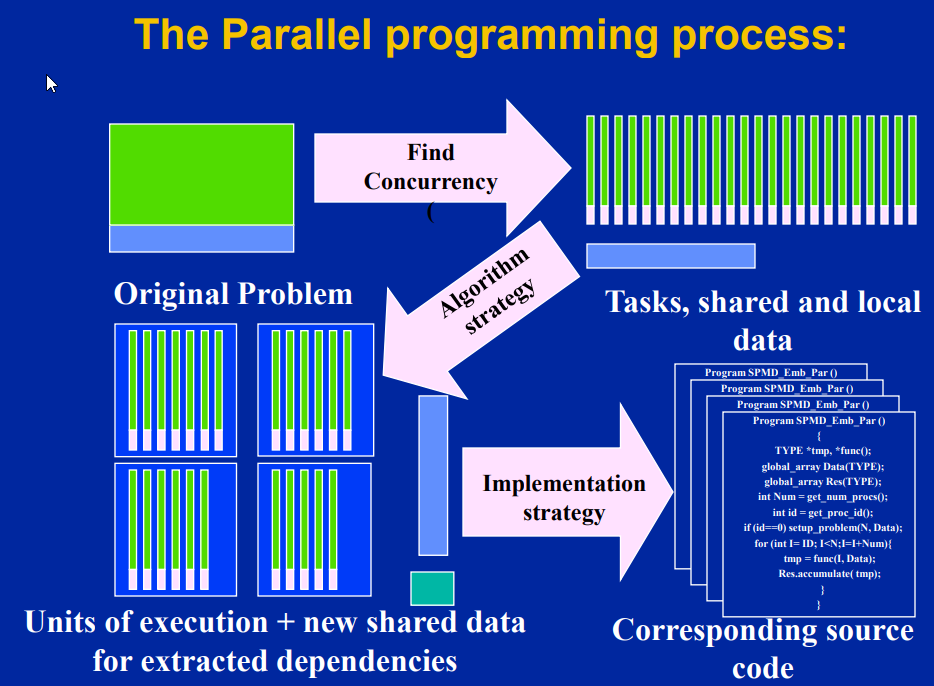
‹Параллелизм: состояние системы, в которой несколько задачи на самом деле активны одновременно.

Параллельное приложение: приложение для которого вычисления логически выполняются одновременно из-за семантики приложения. • Проблема принципиально совпадает.

Параллельное приложение: приложение, для которого вычисления фактически выполняются одновременно Чтобы решить проблему за меньшее время.

• Проблема по сути не требует параллелизм ... вы можете заявить это последовательно

найти совпадение - Единицы исполнения + новые общие данные для извлеченных зависимостей



1. **Дайте общую характеристику технологии OpenMP. Какие средства необходимы для использования OpenMP.**

OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. λ Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения λ Предназначен для программирования многопоточных приложений на многопроцессорных системах с общей памятью и т.д.

1. **Основы программирования мультипроцессорных систем. Понятия: поток выполнения, SMP, NUMA, организация программ для мультипроцессорных систем (с разделяемой памятью).**

**(Поток выполнения** - в вычислительной технике, наименьшая последовательность запрограммированных команд, которые могут независимо управляться планировщиком, являющимся частью операционной системы**)**

Компьютер с общей памятью: любой компьютер, состоящий из нескольких элементы обработки, которые разделяют адресное пространство. Два класса:

**Симметричная многопроцессорность** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Symmetric Multiprocessing*, сокращённо **SMP**) — архитектура [многопроцессорных компьютеров](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80), в которой два или более одинаковых [процессора](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) сравнимой производительности подключаются единообразно к общей памяти (и периферийным устройствам) и выполняют одни и те же функции (почему, собственно, система и называется *симметричной*)

**NUMA** (**Non-Uniform Memory Access** — «*неравномерный доступ к памяти*» или **Non-Uniform Memory Architecture** — «*Архитектура с неравномерной памятью*») — схема реализации [компьютерной памяти](https://ru.wikipedia.org/wiki/%D0%9E%D0%97%D0%A3), используемая в [мультипроцессорных системах](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80), когда время доступа к памяти определяется её расположением по отношению к процессору.

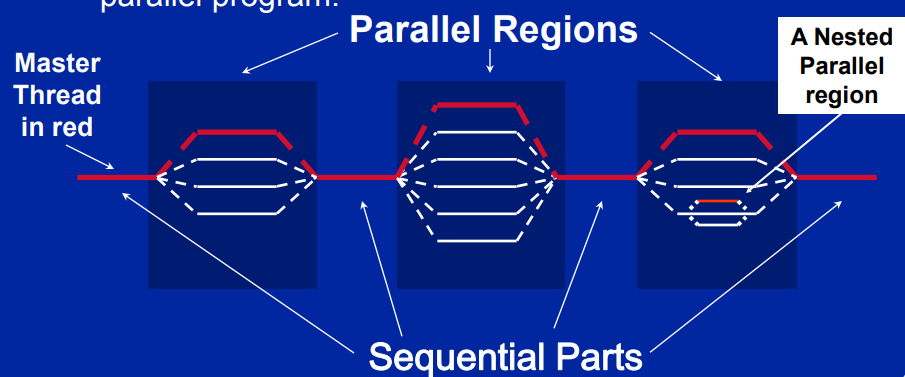
Компьютеры с общей памятью есть везде ... большинство ноутбуков и серверы имеют многоядерные многопроцессорные процессоры address Общее адресное пространство и (как мы увидим) программирование Модели побуждают нас думать о них в системах SMP.

ƒ Реальность сложнее… любой многопроцессорный ЦП с кешем - система NUMA. Начните с обработки системы как SMP и просто примите, что большая часть вашей работы по оптимизации будет касаться случаев где тот случай ломается.

1. **Модель программирования OpenMP: Fork-Join Parallelism.**

Параллелизм Fork-Join: ‹Главный поток порождает команду потоков по мере необходимости. ‹Параллелизм добавляется постепенно до достижения целей встречаются: то есть последовательная программа превращается в параллельная программа.

В [параллельных вычислениях](https://en.wikipedia.org/wiki/Parallel_computing) модель **fork-join** - это способ настройки и выполнения параллельных программ, так что выполнение выполняется параллельно в определенных точках программы, чтобы «объединиться» (объединить) в последующей точке и возобновить последовательное выполнение. Параллельные секции могут проходить [рекурсивно,](https://en.wikipedia.org/wiki/Recursion_(computer_science)) пока не будет достигнута определенная гранулярность задачи. Форк-соединение может рассматриваться как [шаблон](https://en.wikipedia.org/wiki/Design_pattern) параллельного [проектирования](https://en.wikipedia.org/wiki/Design_pattern) .

****

1. **Паттерны разработки SPMD. Как работает, какие средства предоставляет OpenMP  для реализации данного паттерна.**

В OpenMP предполагается SPMD-модель (Single Program Multiple Data) параллельного программирования, в рамках которой для всех параллельных нитей используется один и тот же код

SPMD обычно относят к программированию через обмен сообщениями для архитектур с распределенной памятью. Распределенная память состоит из набора независимых компьютеров, называемых «узлами». Каждый узел запускает свою собственную программу и взаимодействует с другими узлами посредством отправки и получения сообщений, вызывая для этого процедуры отправки и приема. Барьерная синхронизация может быть реализована при помощи сообщений. Сообщения могут быть отправлены с помощью таких механизмов коммуникации как [TCP/IP](https://ru.wikipedia.org/wiki/TCP/IP) (для [Ethernet](https://ru.wikipedia.org/wiki/Ethernet)) или специализированные высокоскоростные соединения, такие как [Myrient](https://ru.wikipedia.org/w/index.php?title=Myrient&action=edit&redlink=1" \o "Myrient (страница отсутствует)) и Supercomputer Interconnect. Последовательные участки (секции) программ реализуются одинаковыми вычислениями на всех узлах, а не с помощью вычисления результата на одном узле и его отправки на другие.

– Sections/section constructs – Single construct – Task construct

1. **В чем состоит проблема false sharing. Когда с ней можно столкнуться при использовании паттерна SPMD. Поясните на примере реализации алгоритма вычисления числа pi (слайды 54-57).**

Термин **false sharing** означает доступ к разным объектам в программе, разделяющим один и тот же блок кэш-памяти. **False sharing** в многопотоковом приложении, когда в одном блоке оказываются переменные модифицируемые из разных потоков, ведет к снижению производительности и увеличению нагрузки на Cache coherence механизмы.

1. **Что в чем состоит проблема race condition? Формы синхронизации mutual exclusion и barrier.**

**Состояние гонки** (*race condition*) — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода. Своё название ошибка получила от похожей ошибки проектирования электронных схем.

Барьер определяет точку в коде, где все активные потоки будут останавливаться, пока все потоки не достигнут этой точки. При этом вы можете гарантировать, что определенные вычисления завершены. Например, в этом фрагменте кода вычисление y не может продолжаться, пока другой поток не вычислил его значение x.

Иногда необходимо разрешить только одному потоку выполнить фрагмент кода. Такой кусок кода называется *критическим разделом*, и OpenMP имеет несколько механизмов для реализации этого.

Наиболее распространенное использование критических разделов - это обновление переменной. Поскольку обновление включает в себя чтение старого значения и запись нового, это может привести к *условию состязания*: другой поток считывает текущее значение, прежде чем первый сможет его обновить; второй поток обновляет неправильное значение.

Критические разделы - это простой способ превратить существующий код в правильный параллельный код. Однако у этого есть недостатки, и иногда требуется более радикальное переписывание.

1. **Барьерная синхронизация, явные и неявные барьеры, их синтаксис, семантика.**

Помимо директивы барьера, который вставляет явный барьер, OpenMP имеет *неявные барьеры* после конструкции распределения нагрузки. Таким образом, следующий код хорошо определен:

#pragma omp parallel

{

#pragma omp for

for (int mytid = 0; mytid <number\_of\_threads; mytid ++)

x [mytid] = some\_calculation ();

#pragma omp for

for (int mytid = 0; mytid <number\_of\_threads-1; mytid ++)

y [mytid] = x [mytid] + x [mytid + 1];

}

Вы также можете поместить каждый параллельный цикл в собственную параллельную область, но есть некоторые накладные расходы, связанные с созданием и удалением группы потоков между регионами.

В конце параллельной области команда потоков распускается, и продолжается только главный поток. Следовательно, существует *неявный барьер в конце параллельной области*.

Существует некоторое *поведение барьерного поведения* }, связанное с omp для циклов и других *барьеров конструкций с разделением рабочих мест* в}. Например, существует *неявный барьер* в конце цикла. Это барьерное поведение может быть отменено с помощью пункта.

Вы часто будете видеть идиому

#pragma omp parallel

{

#pragma omp for nowait

для (i = 0; i <N; i ++)

a [i] = // некоторое выражение

#pragma omp for

для (i = 0; i <N; i ++)

b [i] = ...... a [i] ......

Здесь предложение nowait подразумевает, что потоки могут запускаться во втором цикле, в то время как другие потоки все еще работают над первым. Поскольку оба цикла используют здесь одно и то же расписание, итерация, использующая [i], может действительно полагаться на то, что это значение было вычислено.

1. **Critical и atomic, их синтаксис и семантика, особенности применения.**

**Critical** фиксирует, что код выполняется только по одному потоку за раз.

#pragma omp critical

{

// compare a[i] and max again because max

// could have been changed by another thread after

// the comparison outside the critical section

if (a[i] > max)

max = a[i];

}

Директива **ATOMIC** действует аналогично директиве **CRITICAL**, но относится только к идущему непосредственно за ней атомарному оператору, гарантируя корректную работу с общей переменной, стоящей в левой части оператора присваивания.

1. **Конструкции разделения работ OpenMP: Loop, синтаксис семантика.**

Конструкция с общим циклом разделяет цикл итерации среди потоков в команде

#pragma omp parallel { #pragma omp for for (I=0;I

Вызывает разделение работы for в цикле внутри параллельной области между потоками.

конструкции циклического обмена:

Пункт о расписании

Предложение расписания влияет на то, как итерации цикла отображаются на потоки

schedule(static [,chunk]) – Выделите блоки итераций размером «чанк» для каждого потока.

schedule(dynamic[,chunk])

Каждый поток захватывает «чанковые» итерации из очереди до всех итераций были обработаны

schedule(guided[,chunk])

Потоки динамически захватывают блоки итераций. Размер блока начинается с большого размера и уменьшается до размера «чанка» в расчете продолжается.

schedule(runtime)

Расписание и размер чанка взяты из OMP\_SCHEDULE переменная окружения (или библиотека времени выполнения).

|  |  |
| --- | --- |
| STATIC | Предопределено и предсказуемо программa |
| DYNAMIC | Непредсказуемо, весьма переменная работа за  итерация |
| GUIDED | Частный случай динамического сократить планирование накладные расходы |
| AUTO | Когда среда выполнения может «Учиться» из предыдущего казни того же петля |

1. **Конструкции single и master, где применяются, в чем разница, связь с барьерами.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Может быть одним или несколькими из следующих пунктов:   |  |  | | --- | --- | | copyprivate ( *список* ) | Предоставляет механизм использования закрытой переменной в *списке* для передачи значения из среды данных одной неявной задачи в среды данных других неявных задач, принадлежащих параллельной области. | | firstprivate ( *список* ) | Предоставляет расширенный набор функций, предоставляемых частныйпункт. Каждый закрытый объект данных инициализируется значением исходного объекта. | | nowait ( *целочисленное выражение* ) | Указывает, что реализация может опустить барьер в конце области общего доступа. | | личное ( *список* ) | Объявляет переменные частный к каждой теме в команде. | |
| Основная конструкция обозначает структурированную блок, который выполняется только основным потоком.  z Другие темы просто пропустить (нет подразумевается синхронизация).  Одиночная конструкция обозначает блок кода, который  выполняется только одним потоком (не обязательно  мастер нить).  z Барьер подразумевается в конце одного блока (может  убрать барьер с помощью пункта «сейчас»).  В OpenMP предусмотрена как явная, так и неявная синхронизация нитей. Неявная синхронизация происходит в конце параллельной области, если она не отменяется специальной директивой **NOWAIT**. Явная синхронизация выполняется директивой **BARRIER**. Эта директива определяет точку в параллельной области, которую все нити должны преодолеть одновременно. |

**Описание**

Только одному потоку будет разрешено выполнять структурированный блок. Остальные потоки в команде ждут неявного барьера в концеОдинпостроить, если не указано nowait . Если задано nowait , то остальные потоки в команде немедленно выполняют код после структурированного блока.

В следующем примере показано, как использовать эту прагму, чтобы убедиться, что функция printf выполняется только один раз. Все потоки в команде, которые не выполняют функцию, немедленно переходят к следующим вычислениям.

| пример | |
| --- | --- |
| #include <omp.h>  #pragma omp parallel {  #pragma omp single nowait {printf («Начальный расчет \ n»); }    // Делаем некоторые вычисления  } | |
| [master](https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-directives?view=vs-2019#master) | Уточняется, что только основной поток должен выполнять раздел программы. | |

1. **Конструкции разделения работ OpenMP: Sections, синтаксис семантика.**

Конструкция sections используется для объявления разделов программного кода, выполнение которых необходимо распределить между несколькими потоками.

Конструкция разделения разделов дает различные структурированные блоки для каждого потока

#pragma omp parallel { #pragma omp sections { #pragma omp section X\_calculation(); #pragma omp section y\_calculation(); #pragma omp section z\_calculation(); } }

Директива **parallel sections** обеспечивает краткое определение параллельной области, которая содержит одну единственную директиву **sections**. Семантика этой директивы идентична явно специфицированной директиве **parallel** непостредственно следующей директивой **sections**.

1. **Синхронизация с использованием lock-функций. Какие функции используются, в каких случаях применяются, пример (слайд 99)**

Простые процедуры блокировки:

‹Простая блокировка доступна, если она не установлена.

– omp\_init\_lock(), omp\_set\_lock(), omp\_unset\_lock(), omp\_test\_lock(), omp\_destroy\_lock()

Вложенные Замки ‹Вложенная блокировка доступна, если она не установлена ​​или установлена, но принадлежит потоку, выполняющему вложенную функцию блокировки

– omp\_init\_nest\_lock(), omp\_set\_nest\_lock(), omp\_unset\_nest\_lock(), omp\_test\_nest\_lock(), omp\_destroy\_nest\_lock()

Один замок на элемент истории

Обеспечить взаимное исключение на обновить массив

Освободите хранилище, когда закончите.

1. **Какие функции OpenMP следует использовать для реализации сценария, когда в программе требуется фиксированное число потоков в параллельной области.**

Runtime environment routines

Изменить / проверить количество потоков

– omp\_set\_num\_threads(), omp\_get\_num\_threads(), omp\_get\_thread\_num(), omp\_get\_max\_threads()

Мы находимся в активной параллельной области?

– omp\_in\_parallel()

Вы хотите, чтобы система динамически изменяла количество потоки из одной параллельной конструкции в другую?

– omp\_set\_dynamic, omp\_get\_dynamic();

Сколько процессоров в системе?

omp\_num\_procs()

1. **Переменные окружения OpenMP, примеры, для чего используются.**

Эти переменные среды считываются при запуске программы, а изменения в их значениях игнорируют OMP\_NUM\_THREADS

Устанавливает количество потоков в параллельном блоке. По умолчанию, количество

потоков равно количеству виртуальных процессоров.

OMP\_SCHEDULE

Устанавливает тип распределения работ в параллельных циклах с типом runtime.

OMP\_DYNAMIC

Разрешает или запрещает динамическое изменение количества потоков, которые

реально используются для вычислений (в зависимости от загрузки системы). Значение

по умолчанию зависит от реализации.

OMP\_NESTED

Разрешает или запрещает вложенный пар ся во время выполнения

|  |  |
| --- | --- |
| [OMP\_SCHEDULE](https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-environment-variables?view=vs-2019#omp-schedule) | Изменяет поведение положения [о расписании,](https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-clauses?view=vs-2019#schedule) for когда parallel for schedule(runtime) она указана в директиве или директиве. |
| [OMP\_NUM\_THREADS](https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-environment-variables?view=vs-2019#omp-num-threads) | Устанавливает максимальное количество потоков в параллельной области, если не перекрывается [omp\_set\_num\_threads](https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-functions?view=vs-2019" \l "omp-set-num-threads) или [num\_threads.](https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-clauses?view=vs-2019" \l "num-threads) |
| [OMP\_DYNAMIC](https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-environment-variables?view=vs-2019#omp-dynamic) | Определяет, может ли время выполнения OpenMP регулировать количество потоков в параллельной области. |
| [OMP\_NESTED](https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-environment-variables?view=vs-2019#omp-nested) | Определяет, включен ли вложенный параллелизм, если только вложенный omp\_set\_nestedпараллелизм не включен или отключен с помощью . |

Установите количество потоков по умолчанию для использования

–OMP\_NUM\_THREADS int\_literal

OpenMP добавил переменную среды для управления размером стек дочерних потоков

–OMP\_STACKSIZE

Также добавили переменную окружения в подсказку во время выполнения, как лечить пустые темы

OMP\_WAIT\_POLICY – ACTIVE keep threads alive at barriers/locks – PASSIVE try to release processor at barriers/locks

Привязка процесса включена, если есть переменная среда выполнения не будет перемещаться потоками между процессорами

–OMP\_PROC\_BIND true | false

1. **Data environment: режим разделения по умолчанию для классов переменных в OpenMP (языки С/С++).**

Shared Memory programming model:

Большинство переменных являются общими по умолчанию

Global variables are SHARED among threads

C: переменные области видимости файла, статические

- Оба: динамически распределяемая память (ALLOCATE, malloc, new)

Но не все делятся ...

- Переменные стека в подпрограммах (Fortran) или функциях (C), вызываемых из параллельных регионов являются ЧАСТНЫМИ

- Автоматические переменные в блоке операторов являются ЧАСТНЫМИ.

1. **Data environment: предложения для изменения режима разделения переменных SHARED, PRIVATE, FIRSTPRIVATE, LASTPRIVATE, DEFAULT (SHARED | NONE).**

Можно выборочно изменить атрибуты хранения для конструирует используя следующие пункты

– SHARED – PRIVATE – FIRSTPRIVATE

Конечное значение частного внутри параллельного цикла может быть передается в общую переменную вне цикла с – LASTPRIVATE

Оператор firstprivate сообщает OpenMP о необходимости установить значение индивидуальной переменной, равное значению глобальной переменной. Обычно временные индивидуальные переменные имеют какое-то начальное значение, что позволяет снизить издержки при копировании глобальной переменной.

1. **Конструкции разделения работ OpenMP: Tasks. Принцип работы. Когда завершаются запущенные задачи, как определить по коду программы?**

Задача (task) помещается внутрь параллельной области, и задает блок операторов, который может выполняться в отдельном потоке. Задача не создает новый поток. Задача (блок операторов) помещается в пул, из которого ее может взять один из свободных потоков для выполнения.

Можно отметить, что директива omp task имеет необязательную опцию if, задающую условие. Если условие истинно — будет создана задача и добавлена в пул, как описано выше, если же условие ложно — задача создана не будет, а ассоциированный с ней блок операторов будет немедленно выполнен в текущем потоке. В примерах статьи опция if не используется.

Задачи являются самостоятельными единицами работы.

• Задачи состоят из:

- код для выполнения

- среда данных

- переменные внутреннего контроля (ICV)

• Потоки выполняют работу каждой задачи.

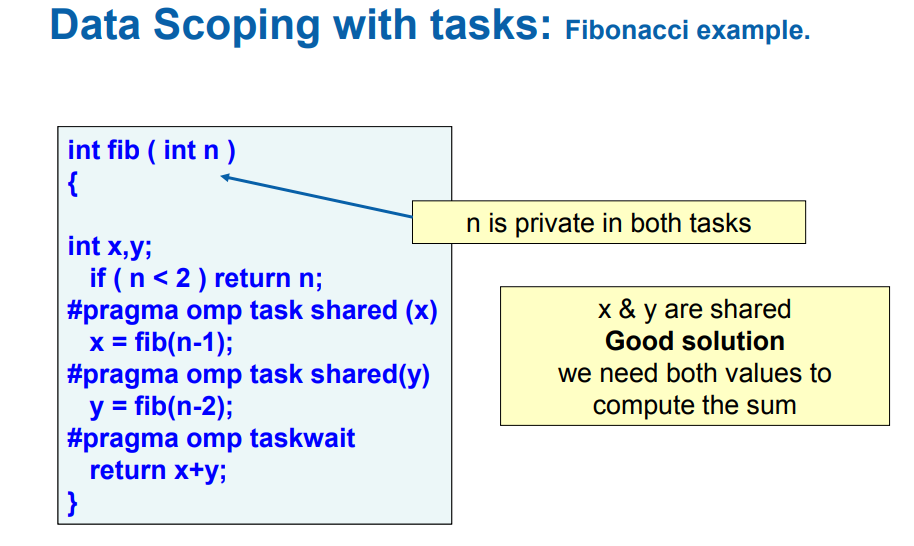
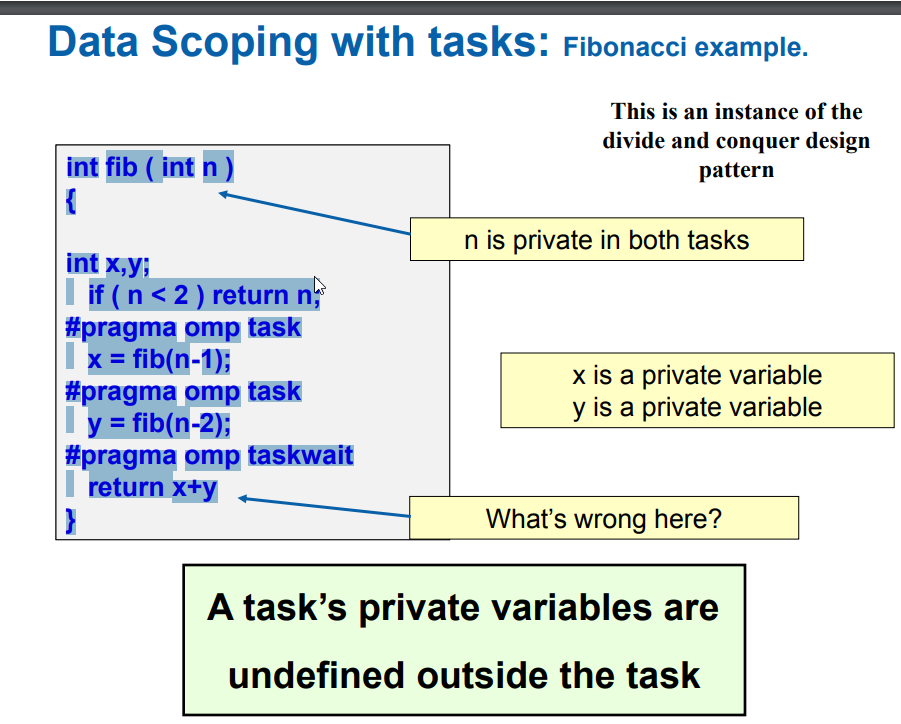
• Система времени выполнения решает, когда задачи выполнены

- Задачи могут быть отложены

- Задачи могут быть выполнены немедленно

1. **Data environment: какой режим разделения в задачах по умолчанию? Как это следует учитывать в программах, пример с вычислением чисел Фибоначчи.**

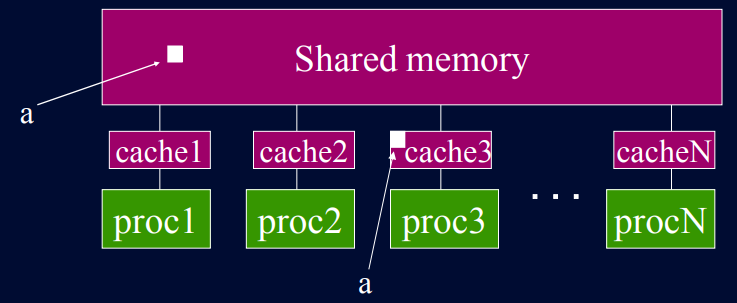
Это пример разделяй и властвуй дизайн шаблон



1. **Модель памяти в OpenMP. Понятия memory coherence, memory consistency, program order, compiler order, commit order.**

OpenMP поддерживает модель разделяемой памяти.

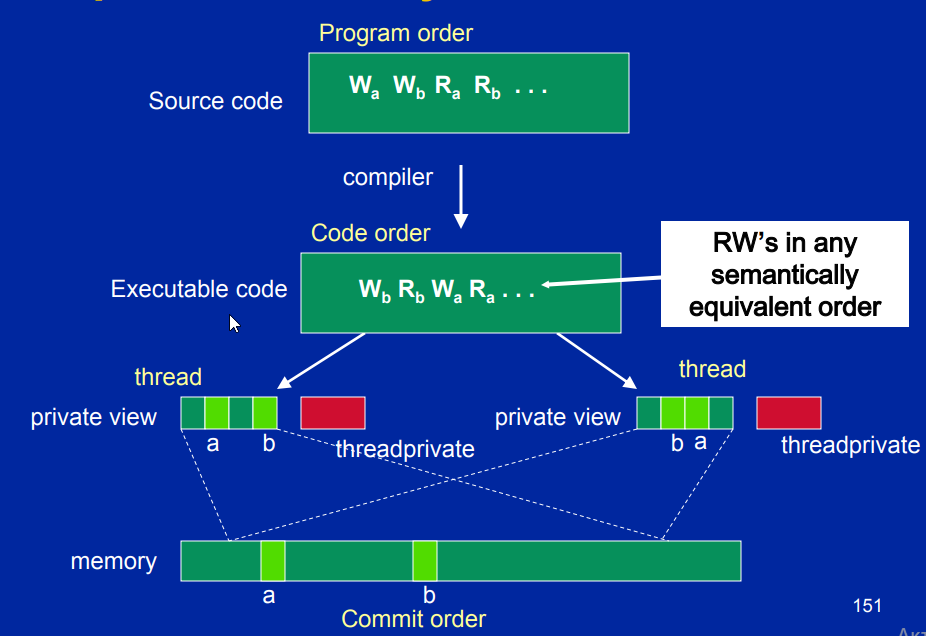
Все потоки разделяют адресное пространство, но это может усложниться:



Модель памяти определяется в терминах:

‹ Coherence: поведение системы памяти, когда один адрес доступен для нескольких потоков.

‹ **consistency**: порядок чтения, записи или синхронизации (RWS) с различными адресами и несколькими потоками.



Re-ordering:

‹Компилятор переупорядочивает программный заказ на код заказа ‹Машина переупорядочивает порядок кода в порядке фиксации памяти

В определенный момент времени «частный вид», видимый поток может отличаться от представления в общем Память.

Модели согласованности определяют ограничения на порядки Чтение (R), Запись (W) и Синхронизация (S)

‹… Т.е. как значения, « видимые »потоком, изменяются по мере того, как вы изменить порядок действий (→) другие операции.

‹Возможности включают в себя:

– R→R, W→W, R→W, R→S, S→S, W→S

1. **Модель памяти в OpenMP. Понятия sequential consistency, relaxed consistancy.**

В мультипроцессоре, операции (R, W, S) последовательно

соответствует, если:

- Они остаются в программном порядке для каждого процессор.

- Они видятся в том же общем порядке каждый из других процессоров.

‹Заказ программы = заказ кода = заказ фиксации

Расслабленная последовательность:

‹Удалить некоторые ограничения для заказа

Оперативная память (R, W, S).

OpenMP определяет согласованность как вариант слабая согласованность:

‹Невозможно изменить порядок операций S с R или W на одном и том же нить

- Гарантии слабой согласованности

S → W, S → R, R → S, W → S, S → S

z Операция синхронизации, относящаяся к этому обсуждение вровень

1. **Какая модель согласования обращений к памяти используется в OpenMP, как работает flush? Пример поставщик-потребитель (слайд 163).**

В OpenMP отсутствуют конструкции синхронизации, которые работать между парами потоков.

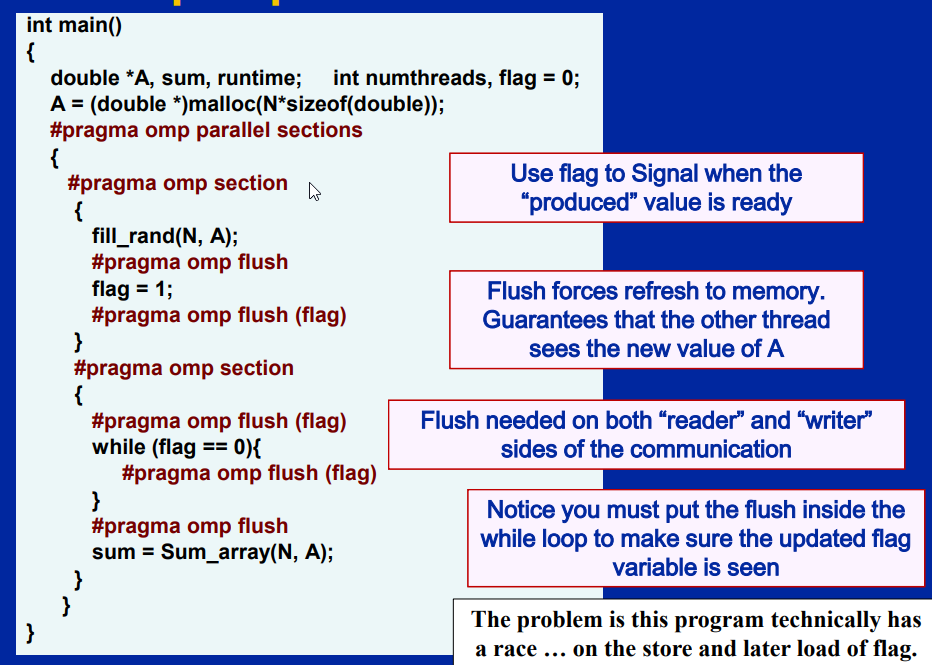
Когда это необходимо, вы должны построить его сами.

Парная синхронизация

‹Используйте переменную общего флага ‹Reader вращается в ожидании нового значения флага

‹Используйте сбросы для принудительного обновления в памяти и из памяти

Эта конструкция осуществляет немедленный сброс значений разделяемых переменных в память. Таким образом гарантируется, что во всех потоках значение переменной будет одинаковое. Неявно flush присутствует в следующих директивах: barrier, начале и конце критических секций, параллельных циклов, параллельных областей, single секций.. С ее помощью можно посылать сигналы потоком используя переменную как семафор. Когда поток видит, что значение разделяемой переменной изменилось, то это говорит, что произошло событие и следовательно можно продолжить выполнение программы далее. (Пример не работает. Не происходит блокирования)



1. **Data environment: THREADPRIVATE, COPYIN.**

Делает глобальные данные приватными для потока

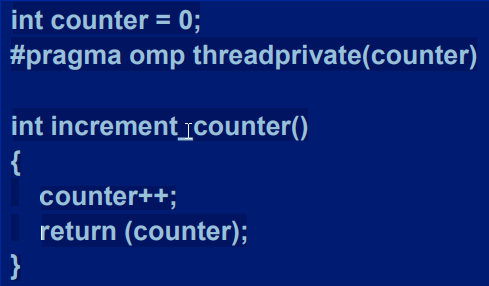
‹Фортран: ОБЩИЕ блоки ‹C: Область видимости файла и статические переменные, статические члены класса

Отличается от того, чтобы сделать их ЧАСТНЫМИ ‹С ЧАСТНЫМИ глобальными переменными маскируются.

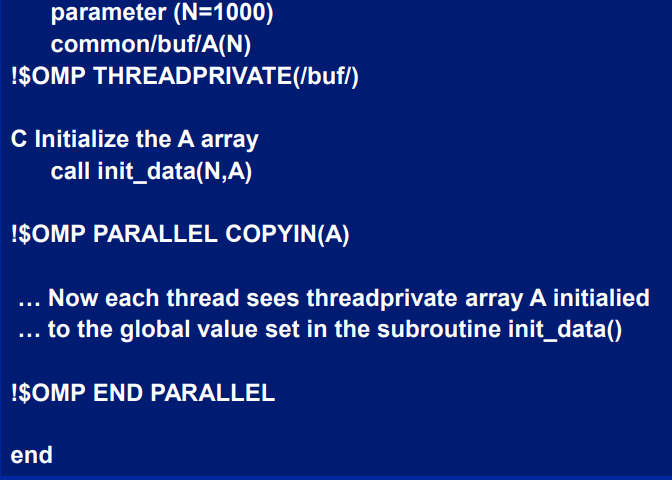
‹THREADPRIVATE сохраняет глобальный охват в каждом нить

Приватные переменные можно инициализировать с помощью COPYIN или во время определения (используя возможности инициализации, определяемые языком).

Используйте threadprivate, чтобы создать счетчик для каждого нить.



Вы инициализируете данные приватного потока, используя пункт.



Используется с одним регионом для трансляции ценностей рядовых от одного члена команды до остальной части команды.

