# 1. Введение

Сегодня мы разберёмся, что на самом деле происходит «под капотом», когда мы используем OpenMP.

OpenMP — это высокоуровневая модель параллельного программирования для систем с общей памятью. Она основана на директивах компилятора, которые выглядят просто — например, #pragma omp parallel for — и позволяют быстро распараллелить цикл или блок кода.

Она используется в языках C, C++ и Fortran и стала стандартом де-факто для научных и инженерных задач.

Однако за этими директивами скрывается сложная работа: компилятор трансформирует их в вызовы специальной рантайм-библиотеки, создаются потоки, синхронизируется доступ к данным, планируются итерации — и всё это с учётом производительности и корректности.

Краткий план сегодняшнего доклада:

1. Познакомимся изнутри с тем, как компилятор обрабатывает директивы OpenMP,

2. Познакомимся с моделями исполнения fork-join и thread pool,

3. Поговорим о синхронизации и защите данных

4. Затронем то, как OpenMP планирует выполнение итераций и циклов

5. Рассмотрим проблемы производительности

6. В конце сравним OpenMP с Posix threads

# 2. Как компилятор обрабатывает директивы OpenMP

## Этап 1: Преобразование директив в вызовы рантайм-функций

Когда компилятор встречает директиву OpenMP, например:

#pragma omp parallel

{

work();

}

он не распараллеливает этот код сам. Вместо этого он генерирует вызовы специальных функций из рантайм-библиотеки OpenMP, таких как \_\_kmpc\_fork\_call.

Тело параллельной области выносится в отдельную функцию ("outlined function"), и потоки из пула вызывают её при выполнении.

## Этап 2: Подготовка переменных

Компилятор анализирует переменные и разделяет их на shared и private.

Для private переменных создаются отдельные копии для каждого потока.

Пример:

int x = 10;

#pragma omp parallel private(x)

{

x = omp\_get\_thread\_num();

}

Здесь переменная x становится локальной для каждого потока, избегая гонки данных.

## Этап 3: Вставка барьеров и синхронизации

В конце параллельной области компилятор вставляет вызов \_\_kmpc\_barrier() — это join часть fork-join модели.

В циклах с parallel for добавляются вызовы \_\_kmpc\_for\_static\_init\_X и \_\_kmpc\_for\_static\_fini.

## Подытожим

Компилятор при обработке директив OpenMP:

1. Обособляет параллельную область как отдельную функцию.

2. Генерирует вызовы в рантайм (например, \_\_kmpc\_fork\_call).

3. Обрабатывает переменные (shared, private, reduction).

4. Добавляет синхронизацию (barrier, critical, for-init).

# 3. Модель исполнения: fork-join и thread pool

## Fork-join модель

OpenMP использует классическую модель fork-join: основной поток (master) при входе в параллельную область создает команду потоков (thread team), которые одновременно выполняют заданную функцию.

После завершения параллельного участка потоки синхронизируются (join) и управление возвращается основному потоку.

Этот механизм реализован через вызовы вроде \_\_kmpc\_fork\_call и внутренние механизмы запуска потоков.

## Thread pool (пул потоков)

Чтобы избежать высокой стоимости создания и уничтожения потоков на каждом заходе в параллельную область, OpenMP реализует пул потоков.

Потоки создаются один раз при инициализации рантайма и переиспользуются для последующих параллельных регионов.

Это резко снижает накладные расходы и ускоряет переход от последовательного к параллельному выполнению.

## Синхронизация

Когда работа потоков в параллельной области завершена, OpenMP использует синхронизацию через барьеры (\_\_kmpc\_barrier), чтобы гарантировать корректный порядок выполнения.

Также могут применяться дополнительные механизмы: флаги завершения, атомарные инструкции и состояния выполнения потоков.

Модель гарантирует детерминированность при условии корректного управления разделяемыми ресурсами.

# 4. Синхронизация и защита данных

Один из ключевых аспектов OpenMP — это синхронизация доступа к разделяемым данным.  
Несмотря на то, что OpenMP упрощает параллельное программирование, программист по-прежнему должен явно указывать, где требуется синхронизация.  
Рассмотрим основные примитивы синхронизации, которые предоставляет OpenMP, и как они реализуются под капотом.

## critical

Директива #pragma omp critical позволяет ограничить выполнение участка кода так, чтобы только один поток мог находиться в этой секции одновременно.  
На уровне реализации компилятор добавляет вызовы \_\_kmpc\_critical\_start и \_\_kmpc\_critical\_end.  
Внутри эти вызовы используют мьютексы или spinlock'и, в зависимости от реализации рантайма.  
Важно понимать, что критическая секция — это глобальный барьер на потоки, и её чрезмерное использование может снизить масштабируемость.

## atomic

Директива #pragma omp atomic используется для атомарного обновления переменных.  
Компилятор может сгенерировать специальные инструкции для процессора или архитектурно-зависимые атомарные инструкции, если они доступны.  
В отличие от critical, atomic не блокирует другие потоки, если операции независимы, и потому гораздо эффективнее при простых операциях, например: x += 1.

## barrier

OpenMP по умолчанию добавляет барьер в конце параллельной области, но директиву #pragma omp barrier можно вызывать и вручную.  
Она реализуется через счётчики и флаги готовности, обычно с использованием атомарных операций или spin-wait.  
Барьер заставляет все потоки дождаться друг друга, прежде чем продолжить выполнение, и необходим для согласованного состояния данных.

## reduction

Директива reduction объединяет значения из всех потоков в одно с помощью указанной операции (например, +, \*, max).  
Под капотом компилятор создает приватные копии переменной в каждом потоке, выполняет локальное накопление, а затем на барьере объединяет их в одно значение.  
Это может реализовываться либо вручную (copy-in/copy-out), либо с использованием встроенных шаблонов рантайма.

# 5. Планирование итераций циклов

При использовании директивы #pragma omp for компилятор и рантайм должны решить, как распределить итерации цикла между потоками.  
Для этого используется несколько стратегий, каждая из которых влияет на производительность и балансировку нагрузки.

## static

Итерации распределяются между потоками заранее, до начала выполнения.  
Каждый поток получает определённое количество итераций, как правило, равномерно.  
Это эффективно, если нагрузка на каждую итерацию одинаковая, но может привести к дисбалансу, если время выполнения итераций варьируется.  
Компилятор генерирует вызов \_\_kmpc\_for\_static\_init с параметрами, определяющими диапазон итераций для каждого потока.

## dynamic

Итерации назначаются потокам во время выполнения.  
Когда поток завершает свою текущую порцию, он получает новую.  
Это хорошо работает при неоднородной нагрузке на итерациях, но имеет больший overhead из-за синхронизации между потоками.  
Реализуется через очередь заданий или атомарный счётчик итераций, общий для всех потоков.

## guided

Гибридный подход: потоки получают большие порции итераций в начале, и размер порций постепенно уменьшается.  
Это снижает накладные расходы в начале и обеспечивает хорошую балансировку ближе к концу.  
Подходит для задач со слабо предсказуемым временем исполнения итераций.  
Реализуется с помощью адаптивного счётчика и динамически изменяющегося размера блока.

Выбор стратегии планирования должен зависеть от характера задачи.  
Static — самый быстрый, но может не сбалансировать нагрузку.  
Dynamic и guided обеспечивают лучшую балансировку, но имеют дополнительные накладные расходы.

# 6. Проблемы производительности: гонки данных и false sharing

Гонки данных (data races) и ложное разделение кэша (false sharing).

## Гонки данных (Data Races)

Гонка данных возникает, когда несколько потоков одновременно читают и/или записывают одну и ту же переменную без должной синхронизации.  
Результат зависит от порядка исполнения и может быть непредсказуемым.  
Например:  
#pragma omp parallel  
{  
 shared\_var++;  
}  
Здесь все потоки обращаются к shared\_var одновременно. Это приводит к некорректным результатам.  
Для устранения гонок используются critical-секции, atomic-операции или приватные переменные.

## False Sharing (ложное разделение кэша)

False sharing — это ситуация, когда потоки работают с разными переменными, но эти переменные лежат в одной и той же кэш-линии.  
Из-за механизма когерентности кэша между ядрами возникает избыточная синхронизация.  
Это особенно критично в многопроцессорных системах: каждый поток невольно заставляет другие ядра инвалидировать и перезагружать кэш.  
Симптом — резкое снижение производительности при, казалось бы, независимой работе потоков.

## Решения

1. Выравнивание данных по границам кэш-линий с помощью аннотаций или специальных структур (например, с паддингом).  
2. Использование приватных переменных, чтобы потоки не делили одни и те же кэш-блоки.  
3. Перераспределение данных и изменение алгоритма так, чтобы потоки обращались к своим сегментам памяти.  
4. Профилирование и использование инструментов, таких как Intel VTune или perf, для выявления узких мест.

# 7. Сравнение с POSIX Threads

OpenMP и POSIX Threads (pthread) — это два разных подхода к параллельному программированию, каждый из которых имеет свои сильные и слабые стороны.

OpenMP предоставляет высокоуровневый интерфейс: программист описывает только, что должно быть выполнено параллельно, а детали управления потоками, синхронизации и распределения работы берет на себя компилятор и рантайм.

Пример:

#pragma omp parallel for

for (int i = 0; i < n; i++) { ... }

В pthread подходе программист сам создаёт и управляет потоками с помощью функций, таких как pthread\_create, pthread\_join, pthread\_mutex\_lock и т.д.

Это позволяет реализовать более точный контроль, но требует больше кода, знаний и повышает риск ошибок.

Таким образом:

- OpenMP проще в использовании, хорошо подходит для быстрого распараллеливания циклов и вычислений.

- POSIX Threads — это низкоуровневый инструмент, подходящий для системного программирования, написания сложных многопоточных систем и тюнинга производительности на уровне ядра.

Выбор зависит от:

- уровня контроля, необходимого программисту;

- требований к переносимости и совместимости;

- важности предсказуемости исполнения и отладки.