**Доклад OpenMP**

Когда мы проходили OpenMP мне стало интересно, как некоторые его механизмы работают изнутри. Как написано на их сайте, OpenMP стремится к стандартизации high-level параллелизма, и сегодня он выступает в какой-то степени стандартом (по крайней мере для C++, C и Fortran)

Я рассмотрю несколько происходящих процессов, которые OpenMP делает под капотом, и начну я с параллелизма циклов и алгоритмов, которые лежат в основе делегации задачи разным потокам (т.е. dynamic schedule)

**Параллелизм**

В информатике параллелизм — это одновременное выполнение вычислений, задач и процессов или потоков. Чтобы лучше понять идеи и цели методов планирования циклов, следует вспомнить о том, что есть различие между программным и аппаратным параллелизмом. Оба вида важны и должны работать вместе, чтобы полностью раскрыть потенциал параллелизма. Процессоры сегодня в основном имеют богатую поддержку аппаратного параллелизма, например, несколько ядер, которые могут одновременно выполнять разные задачи, или те методики конвейеризации внутри процессора, о которых мы также говорили.

Также следует оговориться, что рассматриваемые далее модели делегации подзадач используются только в системах с общей памятью.

Начнём тоже немного издалека: основной целью распараллеливания в каких-либо вычислениях обычно является цикл. Под распараллеливанием цикла подразумевается разбиения N итераций цикла (т.е. одной задачи) на несколько задач с меньшим количеством итераций. И анализировать мы будем как раз то, как это разбиение на программном уровне происходит.

**Статическое разбиение**

Тут всё просто: OpenMP видит количество потоков, видит количество итераций, делит их равномерно между потоками и те выполняют. Видно, что разбиение неидеально, но мы всё это уже обсуждали. Серым показано количество неиспользуемого времени.

**Динамическое разбиение**

Более интересны принципы динамического разбиения.

Централизованный/нецентрализованный — либо главный поток раздаёт задачи остальным потокам в runtime, либо все потоки равномерно разбирают задачи (опять же по различным алгоритмам, которые мы рассмотрим далее) до тех пор пока задачи не выполнятся.

Также динамические алгоритмы могут быть адаптивными или неадаптивными. Адаптивные алгоритмы приспосабливаются к имеющимся результатам выполнения программы. Например, какой-то поток получил задание от ОС, не связанное с выполнением текущей программы: тогда он выполнит свою работу куда медленнее. Адаптивный алгоритм, как следует из названия, адаптируется к показателям выполнения задачи в runtime. Неадаптивный, в свою очередь, использует лишь данные, которые были получены до runtime.

**Self-Scheduling**

Идея алгоритма Self-Scheduling заключается в том, что каждая новая итерация выделяется на тот поток, который не занят. Получается, что для каждой итерации мы тратим ресурс диспетчера для того, чтобы загрузить данный поток. И так происходит до тех пор, пока мы не закончим со всеми итерациями. Как вы видите, Chunk Size тут минимально возможный.

Этот вид диспетчеризации является одним из самых старых и примечательно, что по времени накладных расходов на диспетчеризацию они вместе с предыдущим статическим типом находятся на полюсах. У статического подхода время на диспетчеризацию практически отсутствует, в свою очередь, у этого метода оно самое большое, потому как ему нужно выделять каждую отдельную итерацию.

**Fixed Size Chunking**

Следующий вид диспетчеризации — это распределение задач основываясь на формуле из математического анализа (как видите, размер чанка теперь выражается не так просто). Её цель — это как раз найти баланс между статическим распределением и полным динамическим, т.е. как раз поймать наиболее удобное соотношение затрат на диспетчеризацию и объёма выделяемой работы.

**Guided Self-Scheduling**

GSS использует другой подход для уменьшения размера чанка вместо того, чтобы устанавливать фиксированное время. Он использует параметр R, который говорит о количестве оставшихся итераций, что говорит о более-менее динамическом распределении обязанностей, то есть чем ближе к концу выполнения задачи процессор, тем лучше будут выделяться задачи. Проблема этого алгоритма заключается в том, что при начальном выделении мы можем присвоить задачу потоку, который будет выполнять задачи медленнее остальных (НАПРИМЕР). И это может привести к замедлению работы программы.

**Trapezoid Self-Scheduling**

Данный подход пытается использовать концепцию предыдущего метода (GSS), в то же время пытаясь использовать линейную функцию для уменьшения размера чанков ближе к концу выполнения задачи. И если в предыдущем методе мы без лишних ухищрений пытаемся просто линейно подходить к нулю, то здесь мы используем просто другой график.

**Другие алгоритмы**

В источнике, из которого я брал эти алгоритмы, представлены ещё несколько (а именно 8) других методов, которые уже достаточно нетривиальны. С ходу в них разобраться будет довольно сложно, поэтому я про них рассказывать не буду. Сейчас мы перейдём к тому, как вышеописанные способы распараллеливания реализуются в OpenMP.

**Что в OpenMP?**

В OpenMP, как мы знаем, существует три типа диспетчеризации: это static, dynamic и guided. Программист также может указать размер чанка, с которым он хочет работать.

Эти три типа по своей сути опираются на базовые алгоритмы. Для static — это статическая диспетчеризация; для dynamic — это Self-Scheduling, для guided — Guided Self-Scheduling. Однако конкретная реализация зависит от библиотеки, которую вы будете использовать.

**LLVM OpenMP Runtime Library**

В нашем примере мы рассмотрим вот эту библиотеку. Чтобы не было путаницы отдельно отмечу:

1. OpenMP — это интерфейс, который поддерживает кросс-платформенный параллелизм с общей памятью. Благодаря своей популярности он является стандартом, который можно реализовать благодаря его открытости (OpenMP — open-source проект)
2. В свою очередь, LLVM — это проект, который реализует набор модульных технологий и инструментов компилятора. А реализуются эти технологии посредством библиотек. Одной из таких библиотек является libomp

**Структура распараллеливания циклов в OpenMP**

В библиотеке libomp находится большое количество файлов, но интересны нам будут только те, которые представлены на картинке. Перед каждым циклом происходит следующий алгоритм. Описывая его вкратце, можно сказать, что файлы слева подготавливают окружение к работе (т.е. указывают тип диспетчеризации, количество потоков и так далее), а затем запускается выполнение задачи.

Затем, как только программа находит очередную pragma, запускается функция Init() из файла kmp\_dispatch.cpp, которая инициализирует процесс — делается это для каждого потока. После этого каждый поток выполнят функцию Next() до тех пор пока не завершится выполнение задачи.

Реализация необходимых алгоритмов диспетчеризации, которые мы обсуждали выше, лежит в файле kmp\_dispatch.cpp

**Реализация алгоритмов**

Ну и в конце я быстро покажу псевдокод функций Init и Next. Как видите, Init — это банальный switch-case для различных типов диспетчеризации. И в Next реализуются тоже через switch-case конструкцию алгоритмы.

Реализация же самих алгоритмов происходит относительно просто, потому что до этого это всё было хорошо математически описано.

===

Если кому-то интересно посмотреть на более сложные алгоритмы, которые я не осветил, а также посмотреть на их бенчмарки, то я опирался на эту магистерскую диссертацию. Если кому-то надо, могу прислать ссылку.

На этом у меня всё.