Высокопроизводительные вычисления

Лекция 2.

Оценка максимально возможного параллелизма

Обеспечение наилучших наилучшего ускорения $S_p = \frac{T_1}{T_p} = p$ эффективности $E_p = \frac{T_1}{p T_p} = 1$ возможно не для всех вычислительно трудоемких задач. Для объяснения ограничений в параллелизации вычислений при наличии сугубо последовательных расчетов сформулирован

Закон Амдаля: Достижению максимального ускорения может препятствовать существование в выполняемых вычислениях последовательных расчетов, которые не могут быть распараллелены. Пусть f - доля последовательных вычислений в применяемом алгоритме обработки данных, тогда ускорение процесса вычислений на p процессорах ограничено и

$$S_p \le \frac{1}{f + \frac{1-f}{p}} \le S^* = \frac{1}{f} \quad .$$

Здесь S^* это ускорение для $p=\infty$.

Таким образом, если $f = 0.1$, т.е. доля последовательных расчетов всего 10%, то эффект от использования параллелизма не может превышать 10. В примере с частными суммами (из предыдущей лекции) $S^* = \frac{n}{\log_2 n}$. Закон Амдаля характеризует одну из самых серьезных проблем параллельного программирования, ведь алгоритмов без последовательных вычислений практически не существует.

Закон Амдаля не учитывает возможную зависимость доли последовательных вычислений f от параметра сложности задачи n , т.е. зависимости $f = f(n)$. Для многих задач, однако, $f(n)$ является убывающей функцией и ее (т.е. долю последовательных вычислений) можно уменьшить увеличив сложность задачи. Это приводит к тому, что ускорение $S_p = S_p(n)$ является возрастающей функцией от параметра сложности задачи.

Закон Густафсона-Барсиса (Gustafson-Barsis's Law)

Оценим максимально достижимое ускорение исходя из имеющейся *доли последовательных расчетов* в выполняемых параллельных вычислениях

$$g = \frac{\tau(n)}{\tau(n) + \frac{\pi(n)}{p}} \quad ,$$

где $\tau(n)$ - время выполнения последовательной части алгоритма, а $\pi(n)$ - время параллельной части. Через вновь введенные параметры можно выразить

$$T_1 = \tau(n)+\pi(n) \ ,$$
$$T_p = \tau(n)+\frac{\pi(n)}{(p)} \ .$$

С учетом выражения для g можно получить выражения

$$\tau(n) \ = \ g\left(\tau(n)+\frac{\pi(n)}{p}\right) \ ,$$
$$\pi(n) \ = \ (1-g)\,p\left(\tau(n)+\frac{\pi(n)}{p}\right) \ ,$$

что, в свою очередь, позволяет получить оценку

$$S_p \ = \ \frac{T_1}{T_p} \ = \ \frac{\tau(n)+\pi(n)}{\tau(n)+\frac{\pi(n)}{p}} \ = \ g+(1-g)p \ = \ p+(1-p)g \ .$$

Выражение

$$S_p \ = \ g+(1-g)p \ = \ p+(1-p)g$$

и есть закон Густафсона-Барсиса, который еще называется *ускорением масштабирования* (scaled speedup), поскольку он позволяет оценить насколько эффективно могут быть организованы вычисления при увеличении сложности задачи.

Масштабируемость параллельных вычислений

Цель применения параллельных вычислений не только в уменьшении времени расчетов, но и в решении более сложных вариантов задач. Способность параллельного алгоритма эффективно использовать процессоры при повышении сложности вычислений характеризует его масштабируемость. Накладные расходы (то время, или те вычисления, которые приходится выполнять для организации параллельных вычислений) это

$$T_0 \ = \ pT_p-T_1 \ .$$

Тогда $T_p \ = \ \frac{T_1+T_0}{p}$ и ускорение можно представить как $S_p = \frac{T_1}{T_p} = \frac{pT_1}{T_1+T_0}$, а эффективность выразить как $E_p = \frac{S_p}{p} = \frac{T_1}{T_1+T_0} = \frac{1}{1+\frac{T_0}{T_1}}$.

Из последнего выражения для эффективности можно сделать вывод, что при

росте числа процессоров она будет убывать из-за роста накладных расходов T_0 при фиксированной сложности задачи T_1. Поэтому повышая сложность T_1 с ростом p, предполагая более медленный рост T_0 можно обеспечить заданную эффективность E_p. Пусть $E_p = const$, тогда выражение для нее можно переформулировать как

$$\frac{T_0}{T_1} = \frac{1-E_p}{E_p} \quad \text{или} \quad T_1 = K T_0 \text{, где } K = \frac{E_p}{1-E_p} .$$

Фиксация эффективности на заданном уровне и увеличени сложности задачи с ростом количества процессоров порождает функцию $n = F(p)$, которую называют *функцией изоэффективности*.

Принципы разработки параллельных алгоритмов

Действия предварающие разработку эффективного алгоритма параллельных вычислений для той или иной задачи могут состоять в следующем:

анализ существующих вычислительных схем для задачи и декомпозиция задачи на параллельные подзадачи;

выделение информационных взаимодействий;

определение необходимой (доступной, подходящей) вычислительной системы.

При разработке параллельного кода необходимо минимизировать коммуникационные взаимодействия.

При проектировании удобно бывает использовать различные графы. В предыдущей лекции для представления алгоритмов использовались графы, которые можно назвать «операции-операнды». Иногда удобно использовать граф «подзадачи — информационные зависимости» или граф «потоки — общие данные». В последнем случае информационные зависимости реализуются через общие данные, распределение потоков можно проектировать непосредственно на вычислительные узлы компьютера. Число подзадач может превосходить число процессоров.

Далее некоторые понятия используемые в параллельных вычислениях.

Процесс — программа на вычислительном узле мультикомпьютера.

Поток (thread) — логически выделенная с точки зрения операционной системы последовательность команд, которая может выполняться на одном вычислительном элементе и содержит операции доступа к общим данным.

Общие данные — общий для потоков ресурс (чтение — запись данных).

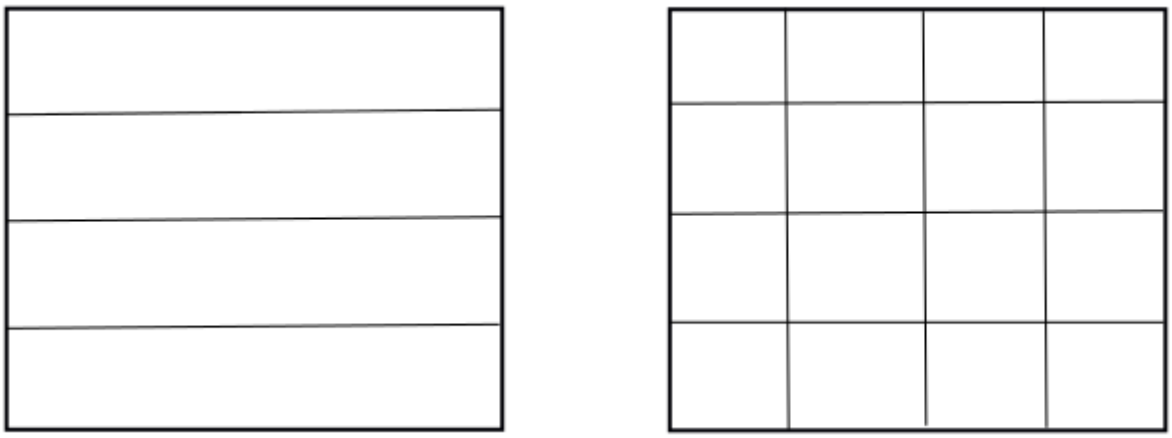
Общие данные должны использоваться согласно правилам *взаимного исключения*, т.е. в каждый момент времени данные доступны только одному потоку. Эти правила могут приводить к задержкам и блокировкам потоков, зависаниям.

Разработка параллельного алгоритма

Рассмотрим общие вопросы разработки параллельного алгоритма на примере задачи поиска максимального значения среди элементов матрицы $\boldsymbol{A}$. Такая задача встает при численном решении систем линейных уравнений для определения ведущего элемента в методе Гаусса. Пусть матрица имеет размер $N \times N$ элементов и формально задачу можно написать как

$$y = \max_{1 \le i, j \le N} a_{ij} .$$

На первом этапе разработки алгоритма нужно определить каким образом вычисления могут быть разделены на *независимые части*. Имеется два основных способа сделать это: 1) параллелизм по данным или *декомпозиция данных* и 2) *функциональная параллелизация*. Для задачи поиска максимального значения лучше подходит способ 1). Он заключается в разделении матрицы $\boldsymbol{A}$ на блоки с работой с каждым блоком в отдельном потоке/процессе, на отдельном вычислительном элементе. Ниже на рисунке представлены варианты разбиения матрицы, а именно ленточная декомпозиция и блочная декомпозиция.



При декомпозиции разработчик должен задать себе следующие вопросы:

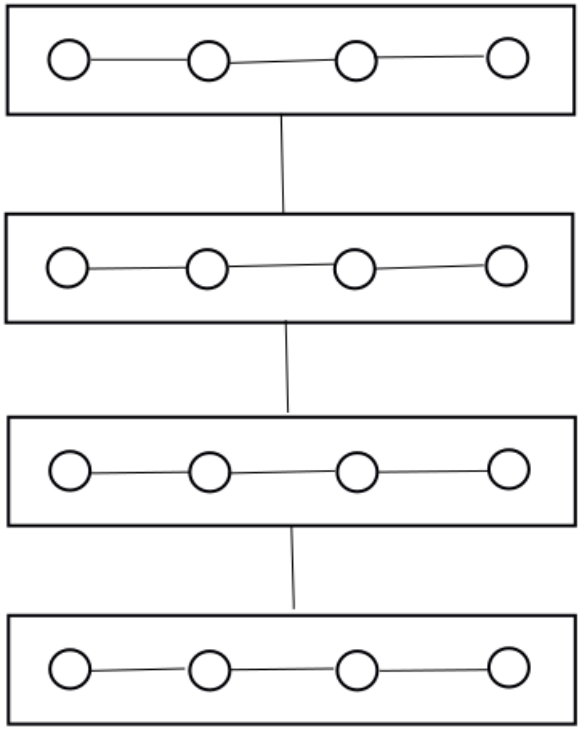
1. Не увеличивает ли выполненная декомпозиция объем вычислений и необходимый объем памяти?
2. Возможна ли при выбранном способе декомпозиции равномерная загрузка всех имеющихся вычислительных элементов компьютера?
3. Достаточно ли выделенных частей процесса вычислений для эффективной загрузки имеющихся узлов?

На следующем этапе разработки нужно выделить *информационные зависимости*. Например в нашей задаче поиска максимального значения отдельные процессы могут обмениваться максимальными значениями

определенными в границах каждого блока, а могут передать свои блочные максимальные значения только узлу, который будет собирать результаты параллельной части алгоритма. Можно выделить следующие информационные зависимости:

- локальные и глобальные, т.е. либо между небольшим числом подзадач, либо сразу между всеми;
- структурные и произвольные (для структурных можно выделить регулярную топологию постоянных связей: звезда, кольцо, гиперкуб, ...);
- статические и динамические (участники взаимодействи фиксируются на этапе разработки кода или определяются в ходе выполнения алгоритма).

Ниже на рисунке приведен пример графа информационных зависимостей при ленточной декомпозиции матрицы.



Вопросы разработчика себе:

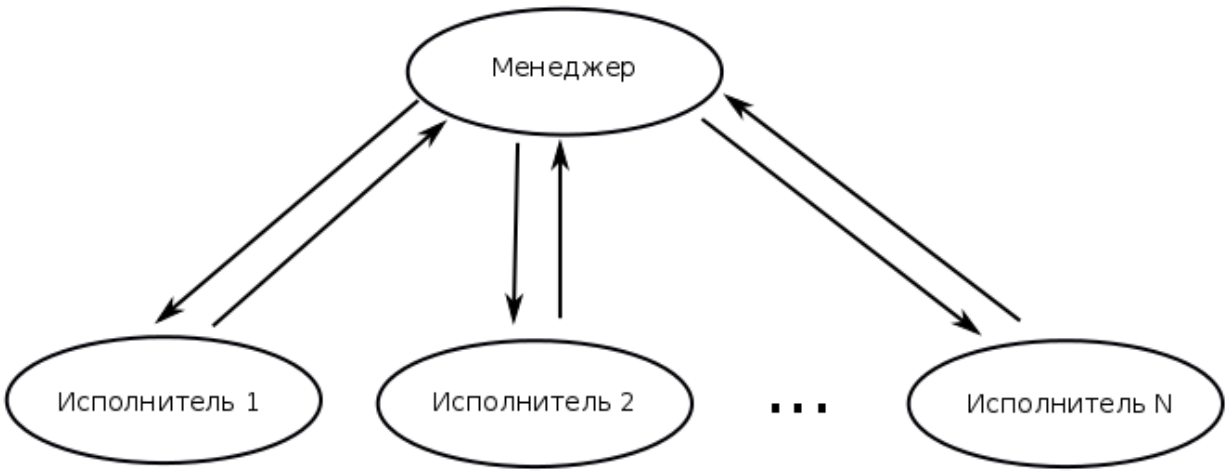
1. Соответствует ли вычислительная сложность подзадач интенсивности их информационных обменов?
2. Является ли интенсивность информационных взаимодействий одинаковой для разных подзадач?
3. Не препятствует ли выявленная информационная зависимость параллельному решению задачи?

Следующий этап разработки параллельного кода связан с *масштабируемостью набора подзадач*. Это необходимо для того, чтобы приводить в соответствие число подзадач и число имеющихся в распоряжении вычислительных узлов. Нужно либо осуществить агрегацию подзадач в случае их большего числа чем число узлов, либо выполнить более детальную декомпозицию. Для того, чтобы программа могла выполняться на различных архитектурах для различного уровня сложности нужно запрограммировать правила агрегации и декомпозиции. На этом этапу уместно спросить у себя следующее:

1. Насколько ухудшится локальность вычислений после масштабирования?
2. Имеют ли подзадачи после масштабирования одинаковую вычислительную и коммуникационную сложность?
3. Соответствует ли число подзадач количеству вычислительных узлов?
4. Зависят ли правила масштабирования от количества вычислительных узлов?

На завершающем этапе разработки необходимо заняться *распределением задач между вычислительными элементами*. Управление нагрузкой каждого процессора возможно только для вычислительных систем с распределенной памятью (как у кластера). В системах с общей памятью распределение нагрузки обычно выполняется операционной системой. Основной путь это равномерное распределение нагрузки между вычислительными элементами и минимизация информационного обмена между ними.

Широко распространенная схема динамического распределения нагрузки называется «менеджер-исполнитель»



Уместные вопросы:

1. Не приводит ли распределение нескольких задач на один процессор к росту дополнительных вычислительных затрат?
2. Существует ли необходимость динамической балансировки?

3. Не является ли элемент-менеджер «узким» местом?