Конкурсная работа Acceler8 2011

Поиск подматрицы с максимальной суммой элементов на многоядерных ЭВМ с общей памятью

А. С. БОГАТЫЙ
Московский государственный университет имени М. В. Ломоносова
механико-математический факультет, 3 курс
e-mail: bogatyia@gmail.com

А. Р. ЛАПИН
Московский государственный университет имени М. В. Ломоносова
механико-математический факультет, 3 курс
e-mail: lapinra@gmail.com

октябрь - ноябрь 2011

1 Введение

Данная статья, в сущности, является описанием программы, написанной нашей командой для конкурса по параллельным алгоритмам **Acceler8 2011** компании **Intel**, проводимого с 15 октября по 15 ноября 2011 года.

Задача, поставленная перед участниками, заключалась в нахождении подматрицы с максимальной суммой элементов среди всех подматриц данной матрицы. В одном тесте может быть несколько матриц, каждая из которых имеет до 20000 строк и столбцов и генерируется по алгоритму, заданному в условии. Для каждой матрицы необходимо было вывести координаты угловых элементов, сумму элементов и площадь найденной подматрицы. При наличии нескольких подматриц с наибольшей суммой элементов ответом на задачу могла являться любая из них.

Критерием оценки решения являлась скорость работы программы на многоядерных машинах с различными конфигурациями при условии корректной работы программы (нахождения решений для всех матриц из данного теста за конечное время).

При создании нашей программы мы рассмотрели несколько последовательных алгоритмов: от самого простого решения перебором до более сложного алгоритма Джея Кадане (Jay Kadane). Мы предъявим способ распараллеливания алгоритма Кадане, рассмотрим особенности его реализации и предоставим сравнительный анализ полученных результатов.

2 Описание решения поставленной задачи

Здесь и далее мы будем рассматривать случай, когда текущая матрица хранится в двумерном массиве, который полностью помещается в оперативную память.

2.1 Наивный алгоритм

Самым простым алгоритмом решения поставленной задачи является перебор всех возможных подматриц и подсчета суммы элементов для каждой из них.

Пусть матрица имеет n строк и m столбцов, тогда подсчет суммы ее элементов выполняется за $n \times m$ операций. Всего количество подматриц в матрице порядка $(n \cdot m)^2$ (тоже самое, что количество способов выбрать левый верхний и правый нижний углы подматрицы). В сумме мы получаем, что данный алгоритм работает за $O(n^3 \cdot m^3)$.

2.2 Наивный алгоритм с предподсчетом

Попробуем улучшить предыдущий алгоритм, используя то, что матрица фиксирована и ее элементы не меняются.

Пусть нам дана матрица $A = (a_{ij})$ размера $n \times m$. Определим вспомогательную матрицу $B = b_{ij}$ следующим образом: элемент b_{ij} есть сумма элементов подматрицы A, левый верхний элемент которой совпадает с левым верхним углом матрицы A, а правым нижним углом является элемент с индексами i и j.

Утверждение 1: Пусть матрица B построена, тогда сумма элементов s подматрицы с угловыми элементами i_1, j_1 и i_2, j_2 (левым верхним и правым нижним, соответственно) вычисляется по формуле:

$$s = b_{i_2,j_2} - b_{i_1-1,j_2} - b_{i_2,j_1-1} + b_{i_1-1,j_1-1}$$

Это утверждение легко доказывается, если нарисовать наглядную картинку. По сути, берется сумма элементов большой матрицы, из нее вычитаются лишние куски и прибавляется та часть, которая была вычтена два раза.

Утверждение 2: Матрицу B можно построить за $n \times m$ операций.

Это утверждение явно следует из формулы для элементов матрицы B:

$$b_{i,j} = a_{i,j} + b_{i-1,j} + b_{i,j-1} - b_{i-1,j-1},$$

которая выводится из тех же соображений, что и формула из предыдущего утверждения.

Теперь мы можем сначала посчитать матрицу B, и, после этого, перебрать все подматрицы. Таким образом, мы получили алгоритм, работающий за $O(n^2 \cdot m^2)$ времени (т.к. сумма элементов любой подматрицы считается за $O(1)$).

2.3 Последовательный алгоритм Кадане

Следующая оптимизация алгоритма вытекает из перестановки слагаемых в формуле из **утверждения 1**:

$$s = (b_{i_2,j_2} - b_{i_1-1,j_2}) - (b_{i_2,j_1-1} - b_{i_1-1,j_1-1})$$

Для двух фиксированных строк i_1 и i_2 мы можем перебирать правый столбец подматрицы, а левый столбец выбирать с минимальной разностью $(b_{i_2,j_1-1} - b_{i_1-1,j_1-1})$. Текущий минимум можно обновлять при переходе от стобца j_2 к столбцу j_2+1. Таким образом, получается алгоритм, работающий за $O(n^2 \cdot m)$ времени.

2.4 Параллельный алгоритм Кадане

За основу параллельного алгоритма мы взяли последовательный алгоритм Кадане, который заключался в переборе верхней строки, нижней строки и правого столбца подматрицы, и распределили перебираемые верхние строки по потокам.

Пусть нам доступно p потоков (процессоров/ядер), строки матрицы нумеруются от 1 до n, столбцы от 1 до m. Тогда в первом потоке будем обрабатывать случаи, когда верхней строкой подматрицы является строка с номером вида $1 + kp$, где $k \geqslant 0$, $1 + kp \leqslant n$, во втором потоке - строка $2 + kp$, в r-ом потоке, соответственно, строка $r + kp$. Таким образом мы получаем равномерную загруженность всех доступных ядер.

2.5 Особенности практической реализации

Наше решение написано на языке `C` с использованием библиотеки `pthreads`. Приведем список примененных нами оптимизаций:

1) Если писать параллельный алгоритм, отдавая каждому потоку по начальной строке и дальше идти по этой строке, то алгоритм выходит неэффективным с точки зрения L1 и L2 кэша, так как при каждой операции элемент нижней строки подматрицы не содержится ни в L1, ни в L2 кэше. Блочная версия алгоритма позволяет устранить этот недочет. Мы брали квадратные блоки со стороной кратной 4, чтобы эффективно работала оптимизация 2).

```
// Блок имеет размер BLOCK_ROW x BLOCK_COL
#define BLOCK_ROW 64
#define BLOCK_COL 64
```

2) Разгрузив оперативную память уже можно делать другие оптимизации, например, векторизацию. Но и тут есть подводные камни, алгоритм Кадана - это динамика и поэтому нельзя вычислять следующие элементы строк, не вычислив все предыдущие. Но можно заметить, что вычисления в строчках друг от друга никак не зависят, поэтому в один вектор можно уместить операции вычисления следующих элементов в 4-х строчках сразу. Для этого после копирования части матрицы в кэш, этот блок нужно транспонировать и грамотно организовать циклы.

3) Мы делаем привязку `i`-го потока к `i`-ому ядру, чтобы соседние потоки имели общий кэш и процессоры обращались за памятью к своему `Numa` узлу:

```
// Привязываем поток к процессору
CPU_ZERO(&CPU_ID);
CPU_SET(pargs->CPU_ID, &CPU_ID);  // pargs->CPU_ID - реальный номер потока
sched_setaffinity(0, sizeof(cpu_set_t), &CPU_ID);
```

4) Данный алгоритм генерации матрицы является псевдо-случайным, зависящим от одного последнего значения. Следовательно, можно найти прецикл и цикл этой последовательности. Если L – длина цикла, тогда это можно сделать за $O(L)$ шагов, например при помощи ρ - эвристики Полларда. Найдя цикл,

оставшиеся элементы вычисляются без применения арифметический операций, и, таким образом, программа значительно ускоряется.

5) Вычислив предцикл и цикл матрицу можно генерировать в нескольких потоках, что дает значительное ускорение на больших матрицах.

6) Мы обрабатываем параллельно несколько кейсов, где количество потоков на конкретный кейс зависит от размеров матрицы. Например, на матрицы, которые помещяются в L3 кэш мы отводим не более 10 потоков.

7) Верно следующее утверждение:

Утверждение 3: Всегда существует подматрица с максимальной суммой чисел, у которой верхняя строка находится не ниже строки с номером M, где M – число различных строк в матрице.

Число M не больше, чем число, по модулю которого генерируется матрица. При этом M обычно много меньше этого числа и его можно найти за время O(n), рассматривая количество различных чисел в первом столбце. Таким образом верхнюю строку можно перебирать до M.

2.6 Сравнение времени работы программы при разном количестве используемых процессоров

Замеры проводились на тесте из 1 тест кейса – `19170 19471 4244 632 5183 15601`. Это - первый тест из `huge.txt`, выданного организаторами. Мы взяли только 1 кейс, чтобы проверить качество распараллеливания самого алгоритма, а не системы параллельной обработки нескольких кейсов сразу.

Количество процессоров	Время (sec)	Ускорение
1	1767.19	1.00
2	885.30	1.99
4	448.59	3.93
10	184.74	9.56
20	94.47	18.70
40	51.13	34.56

3 Заключение

В результате работы нами был получен эффективный алгоритм решения задачи. Причём на случайных матрицах(с большим размером цикла) наш параллельный алгоритм оказался весьма выгодным – ускорение почти в 35 раз на 40-ка ядрах. Но и эту константу можно увеличить, если более аккуратно работать с `Numa` системой, однако это потребует дополнительного объема оперативной памяти, которой и так может нехватать на слабой машине. В целом мы довольны проделанной работой, т.к. в процессе конкурса узнали много нового - векторизация вычислений, компиляторы `intel`, привязка потоков и впервые попробовали силы на действительно "многоядерной"машине, а не на 4-х ядерных поделках.

Список литературы

[1] *Богачев К. Ю.* Основы параллельного программирования. М.: Бином, 2003.

[2] *Богачев К. Ю., Миргасимов А. Р.* Об оптимизации вычислительных приложений для многопроцессорных систем с общей неоднородной памятью // Вычислительные методы и программирование. 2010. Т. 11