Группировка данных в системе асинхронного параллельного программирования Аспект

С.Б. Арыков

Предлагается фрагментированный способ хранения данных, суть которого заключается в разбиении данных на группы небольшого размера, умещающиеся в кэш-памяти процессора, за счет чего уменьшается время исполнения как последовательных, так и параллельных программ. Обсуждаются методы реализации этого способа в системе асинхронного параллельного программирования Аспект «прозрачным» для пользователя образом, когда группировка данных выполняется системой автоматически. Приводятся результаты тестовых испытаний нескольких алгоритмов с использованием группировки данных.

1. Введение

Численное моделирование – одна из областей, где задача разработки параллельных программ была актуальна всегда. И хотя именно в этой области накоплен наибольший опыт параллельных вычислений, создание параллельных программ по-прежнему требует высокой квалификации и специфических знаний архитектуры вычислителя. В ИВМ и МГ СО РАН ведется разработка системы асинхронного параллельного программирования Аспект, ориентированной на решение задач численного моделирования, и позволяющей частично преодолеть обозначенные проблемы.

Основная идея системы Аспект заключается в следующем: алгоритмы представляются с высокой степенью непроцедурности [1], на основе которой для них автоматически генерируется представление меньшей степени непроцедурности, такой, что в системе программирования становится возможной генерация высокоэффективной параллельной программы для конкретного вычислителя. Ключевыми особенностями системы программирования Аспект являются автоматическая группировка данных и вычислений.

Группировка вычислений позволяет автоматически изменять степень непроцедурности программы в широких пределах. Высокая степень непроцедурности программы позволяет максимально загрузить работой доступные вычислительные устройства, однако она также влечет за собой высокие накладные расходы на управление. С помощью группировки вычислений программист имеет возможность опытным путем подобрать такой размер группы, что, во-первых, накладные расходы на управление оказываются приемлемыми, а во-вторых, степень непроцедурности все еще достаточна для загрузки работой доступных процессоров (ядер). Некоторые проблемы реализации группировки вычислений в системе программирования Аспект обсуждаются в [2].

Равномерная загрузка вычислительных устройств необходима для достижения максимальной производительности, однако каждая ветвь программы, исполняющаяся на собственном вычислительном устройстве, также должна работать максимально эффективно. Одним из ключевых параметров, влияющих на эффективное исполнение программы, является ее способность задействовать кэш-память процессора. Группировка данных, применяемая в системе Аспект, позволяет хранить данные по группам, делает вычисления более локальными, что повышает эффективность использования кэш-памяти, а также предоставляет ряд дополнительных преимуществ.

Большое количество близких исследований выполняется в рамках работ над высокопроизводительными библиотеками линейной алгебры [3-6], для которых разрабатываются алгоритмы, «дружественные» к кэш-памяти. Отличительной чертой данной работы является ориентированность не на конкретный алгоритм, а на предоставление инструмента разработки параллельных программ, в которых «дружественность» к кэш-памяти частично обеспечивается системой.

2. Проблема представления массивов в памяти и ее решение с помощью группировки

Пусть задан массовый *A*-блок[1], который осуществляет обработку матрицы *C* (Рис. 1, а), заданной с помощью двухмерного массива. На языке C++ элементы матрицы *C* будут размещены в памяти так, как показано на Рис. 1, б. Пусть для этого блока также используется группировка вычислений.

Если размер *A*-группы[2] совпадает с размером матрицы *C*, то элементы матрицы будут извлекаться из памяти последовательно, что обеспечит высокий процент попаданий в кэш. Если же размер *A*-группы равен, допустим, половине размера матрицы *C* по каждой координате (то есть *A*-группа обрабатывает подматрицу размера 2x2), то при переходе к следующей строке внутри *A*-группы вероятность попадания в кэш существенно снижается (эффект возникает только на матрицах большого размера, когда они целиком не помещаются в кэш-память), поскольку эта строка находится на некотором расстоянии от предыдущей (для первой *A*-группы, состоящей из элементов a_{11}, a_{12}, a_{21}, a_{22}, последний элемент первой строки a_{12} и первый элемент второй строки a_{21} располагаются в памяти не последовательно).

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

а)

a_{11}	a_{12}	a_{13}	a_{14}	a_{21}	a_{22}	a_{23}	a_{24}	a_{31}	a_{32}	a_{33}	a_{34}	a_{41}	a_{42}	a_{43}	a_{44}

б)

a_{11}	a_{12}	a_{21}	a_{22}	a_{13}	a_{14}	a_{23}	a_{24}	a_{31}	a_{32}	a_{41}	a_{42}	a_{33}	a_{34}	a_{43}	a_{44}

в)

Рис. 1. Представление матрицы в памяти

Для преодоления указанной проблемы система программирования Аспект позволяет размещать данные в сгруппированной форме. В этом случае матрица из рассмотренного примера будет представляться в памяти не последовательностью строк, а последовательностью групп (порядок следования элементов внутри группы зависит от порядка их обработки операциями в *A*-группе). Один из возможных вариантов такого представления матрицы *C* приведен на Рис. 1, в. Данные каждой *A*-группы располагаются последовательно и количество промахов при обращении в кэш-память существенно сокращается.

В совокупности с группировкой вычислений группировка данных, помимо оптимизации использования кэш-памяти, предоставляет ряд существенных преимуществ:

1. Возможность автоматической генерации параллельной программы. Если алгоритм решения задачи представлен как множество фрагментов, на котором задан частичный порядок, отсутствует необходимость выделять параллельные ветви. Фрагменты можно исполнять в любой последовательности, не противоречащей заданному порядку, с чем вполне справится исполнительная подсистема времени выполнения.

---

[1] Массовый *A*-блок в простейшем случае – это некоторая операция, примененная ко всем элементам массива (в данном случае – ко всем элементам матрицы *C*). Подробней см. в [2].

[2] *A*-группа – это несколько операций массового *A*-блока, объединенных в один фрагмент. Подробней см. в [2].

2. Возможность автоматически обеспечить параллельной программе ряд динамических свойств, в том числе настройку на доступные ресурсы вычислителя (количество процессоров/ядер, объем оперативной и кэш-памяти) и динамическую балансировку загрузки.
3. Переносимость между различными архитектурами. Для реализации группировки данных нет необходимости прибегать к низкоуровневой оптимизации на ассемблере, а изменение размеров группы позволяет подстраивать программу под различный объем кэш-памяти.

3. Реализация группировки данных в системе программирования Аспект

Для управления группировкой данных программисту предоставляется возможность задать размеры группы для каждого массива в специальном конфигурационном файле, который подается на вход транслятору вместе с текстом Аспект-программы [7]. На основании этой информации транслятор автоматически заменяет все обращения к массивам на обращения к группам данных.

В общем случае, каждое обращение к массиву A вида

```
A[i1][i2]...[iN]
```

с заданным размером группы A_FRG_I1xA_FRG_I2x...xA_FRG_IN должно быть заменено на обращение вида

```
A[A_base + j1*A_FRG_J2*A_FRG_J3*...*A_FRG_JN + j2*A_FRG_J3*...*A_FRG_JN +
                                    ... + jN],
```

где A_base – адрес первого элемента группы в массиве A; j1...jN – индексы элемента внутри группы; A_FRG_J2...A_FRG_JN – размеры соответствующих (индексам j1..jN) размерностей массива *A*.

Соответствия между переменными i1..iN и j1...jN, а также A_FRG_I2..A_FRG_IN и A_FRG_J2...A_FRG_JN могут меняться в зависимости от требуемого порядка элементов внутри группы. Например, если i1=j1, i2=j2, ..., iN=jN и A_FRG_I1=A_FRG_J1, A_FRG_I2=A_FRG_J2,...,A_FRG_IN=A_FRG_JN, то порядок соответствует порядку элементов в многомерных массивах языка C++.

Во время трансляции Аспект-программы транслятор проверяет все блоки кода на предмет наличия в них обращений к массивам, и в случае если для данного массива задана группировка данных, выполняет необходимую замену.

Поясним вышесказанное на примере. Пусть в коде *A*-блока транслятор встретил строку

```
C[i][j] += A[i][k]*B[k][j],
```

и пусть для массива C программистом задана группировка данных. Тогда транслятор преобразует ее в строку следующего вида:

```
C[C_base+i*C_FRG_J+j] += A[A_base+i*A_FRG_K+k]*B[B_base+k*B_FRG_J+j].
```

Такая реализация накладывает ограничения на фрагментацию алгоритма: в системе Аспект работают лишь такие фрагментации, в которых каждый фрагмент получает на вход данные от фиксированного числа других фрагментов. Допускается наличие нескольких типов фрагментов, у каждого из которых может быть собственное количество зависимостей (от других фрагментов), однако количество различных типов фрагментов также фиксировано (задано программистом в Аспект-программе).

Процесс программирования с использованием предложенного подхода выглядит следующим образом. Пользователю необходимо разработать фрагментированную версию

алгоритма решения задачи и записать её на языке программирования Аспект [7]. Вычисления внутри операций языка Аспект задаются на C++, поэтому перенос существующих программ (написанных на C++) в систему Аспект достаточно прост, тем не менее он не может быть выполнен механически, поскольку язык Аспект не является императивным языком (управление задается не последовательностью операций, а частичным порядком на множестве операций).

На основе Аспект-программы система автоматически сгенерирует параллельную программу, причём эта программа будет обладать свойствами, перечисленными в разделе 2 (т.е. система берёт на себя все технические вопросы организации параллельной программы). Группировка данных будет выполнена системой также автоматически, от пользователя требуется лишь указать размер группы для каждого массива.

4. Результаты экспериментов

4.1 Подход к тестированию

Рассмотрим результаты тестирования нескольких задач из линейной алгебры. В качестве тестовой платформы использовался компьютер со следующей конфигурацией:

- процессор: Athlon 64 X2 3600+ (2*256 L2);
- память: 1024 DDR2;
- операционная система: Windows Vista Home Premium (32 bit);
- компилятор: Visual C++ 9.0 (с опциями /O2 и /arch:SSE2).

Все задачи были реализованы на C++. Элементы матриц – вещественные числа с двойной точностью (тип double). Для каждой задачи рассматривались следующие способы реализации:

1. Хранение массивов по столбцам. Этот способ размещения данных используется в языке программирования Fortran, и включен в обзор, так как Fortran до сих пор пользуется большой популярностью для решения задач вычислительной математики.
2. Хранение массивов по строкам. Этот способ размещения данных является «родным» для C++. Здесь использовался тот же алгоритм, что и в п. 1.
3. AMD Core Math Library 4.2.0 (ACML). Специализированная библиотека, оптимизированная для процессоров фирмы AMD.
4. Группировка данных (массивы по столбцам). Внутри группы элементы массива размещаются по столбцам. В скобках также указан размер группы.
5. Группировка данных (массивы по строкам). Внутри группы элементы массива размещаются по строкам. В скобках также указан размер группы.

Для задачи умножения матриц дополнительно рассмотрены варианты «Группировка данных (ACML)» и «Ручное решение (простая оптимизация)». Первый отличается тем, что умножение подматриц размером 64x64 выполняется с помощью процедуры библиотеки ACML. Второй вариант представляет собой хорошо известную оптимизацию умножения матриц, заключающуюся в перестановке циклов (т.е. в изменении порядка перебора индексов i, j, k).

Описания алгоритмов протестированных задач широко доступны и с целью экономии места не приводятся. Их можно найти, например, в [8].

Во всех таблицах время счёта приведено в секундах.

4.2 Умножение матриц

При решении с использованием ACML использовалась процедура «dgemm». Результаты тестовых испытаний представлены в Таблице 1.

Во-первых, стоит обратить внимание на разницу между ручными решениями с разным способом хранения массивов. В оптимизированном варианте матрица *A* хранится по строкам, а матрица *B* - по столбцам. Цифры еще раз подтверждают, что современное оборудование таково,

что разработка программ без учета архитектурных особенностей приводит к неприемлемым потерям в производительности.

Таблица 1. Решение задачи умножения матриц

Способ/Размер матрицы	512x512	1024x1024	2048x2048
Ручное решение (хранение массивов по столбцам)	3,73	47,94	511,10
Ручное решение (хранение массивов по строкам)	3,26	45,64	499,22
Ручное решение (простая оптимизация)	0,45	3,70	29,61
AMD Core Math Library 4.2.0 (ACML)	0,09	0,73	5,67
Группировка данных (массивы по строкам, 8x8)	0,22	1,78	14,52
Группировка данных (простая оптимизация, 64x64)	0,17	1,39	11,00
Группировка данных (ACML, 64x64)	0,16	1,00	7,89

Во-вторых, метод группировки данных (простая оптимизация) выигрывает у лучшей ручной реализации в 2,5 раза, что является неплохим результатом. В то же время он проигрывает в те же 2,5 раза библиотеке ACML. Это неудивительно, ведь ACML максимально оптимизирована на уровне ассемблера под конкретную архитектуру.

В-третьих, если для вычислений внутри группы использовать процедуру библиотеки ACML, разница с лучшим результатом составляет всего 40%.

Таблица 2. Влияние размера группы данных на скорость вычислений

Способ (1024x1024)/Размер блока	4	8	16	32	64	128	256	1024
Группировка данных (массивы по строкам)	3,26	**1,78**	2,09	2,34	2,69	6,87	22,19	45,34
Группировка данных (простая оптимизация)	4,50	2,20	1,76	1,51	**1,39**	2,08	3,66	3,70
Группировка данных (ACML)	59,91	10,91	2,80	1,61	1,00	0,92	0,84	**0,73**

Таблица 2 отражает изменение времени счета при различных размерах группы данных. Лучший результат группировки данных (простая оптимизация) достигнут при размере фрагмента 64x64. Это можно объяснить тем, что фрагмент полностью помещается в один сегмент кэш-памяти первого уровня (Athlon X2 3600+ имеет 2-ассоциативную кэш-память первого уровня объемом 64 Кбайт).

4.3 LU-разложение

При решении с использованием ACML использовалась процедура «dgetrf». Результаты тестовых испытаний представлены в Таблице 3.

Таблица 3. Решение задачи LU-разложения

Способ/Размер матрицы	512x512	1024x1024	2048x2048
Ручное решение (хранение массивов по столбцам)	2,55	24,98	244,06
Ручное решение (хранение массивов по строкам)	0,27	2,14	16,30
AMD Core Math Library 4.2.0 (ACML)	0,03	0,26	2,11
Группировка данных (массивы по столбцам, 8x8)	0,08	0,67	5,52
Группировка данных (массивы по строкам, 8x8)	0,08	0,66	5,23

В отличие от задачи умножения матриц, в LU-разложении время счета ручного решения (массивы по столбцам) и ручного решения (массивы по строкам) отличается на порядок, однако применение группировки позволяет свести эту разницу практически к нулю, т. е. даже не зная об особенностях представления массивов в памяти, с использованием группировки человек не

может «промахнуться» слишком сильно, что является несомненным плюсом подхода.

В этой задаче, к сожалению, воспользоваться процедурой ACML для вычислений внутри группы возможно только для фрагментов, стоящих на главной диагонали, поэтому фактически является бесполезным, так как основную вычислительную нагрузку создают именно остальные фрагменты. Невозможность применения ACML связана с тем, что группировка данных может требовать изменения исходного алгоритма и в данной задаче расчеты фрагментов слева и снизу от текущего диагонального фрагмента отличаются обращением за данными к другим фрагментам.

4.4 QR-разложение

Вручную задача QR-разложения решена методом вращений [8]. При решении с использованием ACML использовалась процедура «dgeqrf». Результаты тестовых испытаний представлены в Таблице 4.

Таблица 4. Решение задачи QR-разложения

Способ/Размер матрицы	512x512	1024x1024	2048x2048
Ручное решение (хранение массивов по столбцам)	3,16	31,65	291,08
Ручное решение (хранение массивов по строкам)	0,31	2,45	18,69
AMD Core Math Library 4.2.0 (ACML)	0,08	0,59	4,39
Группировка данных (массивы по столбцам, 64x64)	0,17	1,47	11,75
Группировка данных (массивы по строкам, 64x64)	0,17	1,44	11,37

Полученные результаты качественно соответствуют результатам решения задачи LU-разложения. Здесь также возможно использование подпрограмм ACML для вычислений внутри группы только для фрагментов, расположенных на главной диагонали.

5. Заключение

Результаты тестовых исследований показывают, что реализация модельных задач с использованием группировки данных как минимум в 2.5 раза быстрее задач, реализованных на C++ вручную (без изменения алгоритма и применения ассемблера). На примере с умножением матриц видно, что если для расчетов внутри группы использовать оптимизированную процедуру ACML, разница с лучшим результатом ACML составляет порядка 40%, что представляется умеренной платой за набор дополнительных преимуществ (см. раздел 2).

Следующий этап исследования – применение предложенного подхода для решения прикладной задачи. В качестве последней предполагается выбрать одну из реализаций метода «частицы-в-ячейках» [9-10].

Литература

1. Вальковский В.А., Малышкин В.Э. Синтез параллельных программ и систем на вычислительных моделях. – Новосибирск: Наука, 1988. – 129 с.
2. Арыков С.Б., Малышкин В.Э. Система асинхронного параллельного программирования "Аспект" // Вычислительные методы и программирование. – 2008. – Т.9. – № 1. – С. 205-209.
3. Automatically Tuned Linear Algebra Software (ATLAS) [http://math-atlas.sourceforge.net/], 10.02.2009.
4. AMD Core Math Library (ACML) [http://developer.amd.com/cpu/Libraries/acml/Pages/default.aspx], 10.02.2009.
5. Intel ® Math Kernel Library (MKL)

[http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm], 10.02.2009.

6. Buttari, A., Langou, J., Kurzak, J., Dongarra, J. "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures," ICL Technical Report, UT-CS-07-600 (also LAPACK Working Note 191), September 7, 2007, 2007.
7. Арыков С.Б. Язык программирования АСПЕКТ // Известия Томского политехнического университета. – 2008. – Т. 313. – № 5. – С. 89–92.
8. Беклемишев Д.В. Дополнительные главы линейной алгебры. – М.: Наука. Главная редакция физико-математической литературы, 1983. – 336 с.
9. Григорьев Ю. Н., Вшивков В. А., Федорук М. П. Численное моделирование методами частиц-в-ячейках. – Новосибирск: Издательство СО РАН, 2004. – 360 с.
10. Вшивков В. А., Краева М. А., Малышкин В. Э. Параллельная реализация метода частиц // Программирование. – 1997. – № 2. – С. 39-51.