УДК 519.685.1

С. Б. Арыков, В. Э. Малышкин

Институт вычислительной математики и математической геофизики СО РАН
пр. Акад. Лаврентьева, 6, Новосибирск, 630090, Россия
E-mail: arykov@mail.ru; malysh@ssd.sscc.ru

АЛГОРИТМЫ КОНСТРУИРОВАНИЯ АСИНХРОННЫХ ПРОГРАММ ЗАДАННОЙ СТЕПЕНИ НЕПРОЦЕДУРНОСТИ МЕТОДОМ ГРУППИРОВКИ

Рассматриваются проблемы разработки параллельных программ, реализующих большие численные модели. Предлагается использовать сборочную технологию программирования и поддержать сборку программ в системе асинхронного программирования с тем, чтобы автоматически обеспечивать реализацию динамических свойств (настройка на доступные ресурсы, балансировка загрузки, динамическое распределение ресурсов и др.) прикладных программ. Предложена специализированная асинхронная модель вычислений и алгоритмы группировки *A*-блоков, позволяющие варьировать накладные расходы на организацию управления в широких пределах за счет изменения степени непроцедурности программ. Рассмотрена система асинхронного параллельного программирования Аспект, реализующая некоторые принципы сборочного программирования на параллельных вычислителях с общей памятью.

Ключевые слова: сборочная технология программирования, асинхронные языки и системы программирования, динамические свойства программ, автоматизация параллельной реализации численных моделей.

Введение

Как известно, численное моделирование природных явлений с высокой точностью требует очень больших объемов вычислений, поэтому для решения большинства таких задач используют суперкомпьютеры.

Разработка хорошей параллельной программы является достаточно сложной задачей. В каждую параллельную программу необходимо закладывать возможность динамической настройки на доступные ресурсы, программировать межпроцессные коммуникации, для чего необходимы глубокие знания архитектуры ЭВМ и теории параллельного программирования. В параллельных программах встречаются специфические ошибки, которые сложно обнаружить и устранить в силу недетерминизма исполнения таких программ. Кроме того, постоянно появляются новые архитектуры процессоров, что требует времени на их изучение и освоение.

Таким образом, для разработки качественной программы моделирования специалисту по численным методам приходится все больше внимания уделять проблемам параллельного программирования. В связи с этим встает вопрос о необходимости разработки языка и системы параллельного программирования высокого уровня, которые позволили бы специалисту сосредоточиться на алгоритме решения задачи, взяв на себя технические вопросы конструирования параллельной программы.

Особенности решения численных задач методом «частицы-в-ячейках»

Рассмотрим особенности параллельной реализации алгоритмов решения больших численных задач на примере использования метода «частицы-в-ячейках» (Particle-In-Cell, далее PIC) в задачах динамики бесстолкновительной плазмы. Обсуждается не PIC, а особенности вычислительной схемы реализации PIC, почему и в описании PIC опущены многие существенные детали метода, которые несущественны для его реализации.

Особенности метода «частицы-в-ячейках». Бесстолкновительная, полностью ионизированная плазма описывается кинетическими уравнениями Власова для функций распределения частиц разных сортов с самосогласованными электромагнитными полями [Григорьев и др., 2004], а также системой уравнений Максвелла для электромагнитных полей. Плотность тока и плотность пространственного заряда определяются как интегралы функций распределения частиц по скоростному пространству.

При моделировании методом PIC исходное физическое пространство (обычно имеющее форму прямоугольного параллелепипеда) представляется регулярной сеткой, разбивающей его на ячейки. В узлах сетки дискретизированы значения электрического и магнитного полей. Плазма представляется множеством модельных частиц, распределенных между ячейками. Каждая частица характеризуется координатами, массой, скоростью и величиной заряда.

Электромагнитные поля, дискретные значения которых заданы в узлах ячейки, действуют на частицы внутри этой ячейки, изменяя их скорости и координаты. В свою очередь, в зависимости от частиц, находящихся в ячейке, рассчитываются средняя плотность тока и заряда, на основе которых пересчитываются электромагнитные поля в узлах данной ячейки. Непосредственно друг с другом частицы не взаимодействуют.

Процесс моделирования состоит из серии шагов по времени. На каждом шаге:

1) на основе значений электромагнитных полей вычисляется сила, действующая на каждую частицу;

2) вычисляются новые скорости и координаты частиц;

3) на основе новых координат и скоростей частиц вычисляются значения средней плотности тока и заряда в узлах сетки;

4) из полученных значений средней плотности тока и заряда пересчитываются значения электромагнитных полей.

Выделим основные особенности, существенные с точки зрения параллельной реализации PIC-метода:

1) очень большой объем данных и вычислений. Для решения реальных задач с необходимой точностью требуются большие трехмерные сетки (например, $1024 \times 1024 \times 1024$) и миллиарды частиц;

2) вычисления производятся на сетке (или нескольких сетках, сдвинутых относительно друг друга), имеющей регулярную структуру;

3) вычисления локальны. Для вычисления значений электрического и магнитного полей в текущей ячейке необходима информация только из соседних ячеек. Для обработки частиц необходима информация о полях только текущей ячейки.

Проблемы параллельной реализации метода «частицы-в-ячейках». Детальное изложение проблем параллельной реализации PIC-метода можно найти в [Вшивков и др., 1997; Kraeva, Malyshkin, 2001], отметим здесь лишь главный момент.

В силу больших объемов данных при параллельной реализации приходится применять декомпозицию пространства моделирования (сетка может просто не помещаться в память одного узла). Тогда в каждый узел попадают только те частицы, которые находятся в соответствующей подобласти пространства моделирования. Однако в процессе моделирования частицы могут перемещаться из одной подобласти в другую, создавая дисбаланс нагрузки. Кроме того, в некоторых задачах большая часть частиц может собраться в одном слое сетки или даже в нескольких ячейках. В этом случае выровнять нагрузку с помощью изменения декомпозиции уже не удастся.

Поэтому для эффективной реализации метода частиц необходимо, чтобы программа обладала рядом динамических свойств:

1) позволяла настраиваться на доступные ресурсы (количество узлов в мультикомпьютере, количество процессоров / ядер на каждом узле, объем оперативной памяти);

2) умела осуществлять динамическую балансировку загрузки;

3) поддерживала возможность дублирования одного слоя сетки на нескольких узлах для выравнивания нагрузки в случае, если все частицы соберутся в одном или нескольких слоях (о технологии виртуальных слоев см.: [Краева, Малышкин, 1999; Kraeva, Malyshkin, 2001]).

Ручное программирование этих свойств является достаточно сложной задачей, требующей высокой квалификации и большого опыта, особенно с учетом трудностей, возникающих при отладке параллельных программ.

Сборочная технология программирования. Одним из подходов, позволяющих переложить задачу обеспечения динамических свойств на систему программирования, является сборочная технология [Kraeva, Malyshkin, 2001].

При сборочном подходе решение задачи представляется в виде множества атомарных фрагментов. Каждый фрагмент содержит в себе некоторые данные и вычисления, производимые над этими данными. Из атомарных фрагментов по некоторым правилам конструируются минимальные фрагменты, из которых и собирается конечная фрагментированная программа. Фрагментированная структура программы сохраняется в ходе вычислений, каждый фрагмент реализуется отдельным процессом, а это позволяет автоматически организовывать эффективное параллельное исполнение фрагментированных программ. Необходимость двухуровневой сборки следует из технологических соображений: как правило, атомарный фрагмент содержит в себе небольшой объем вычислений и конструирование фрагментированной программы непосредственно из атомарных фрагментов приводит к большим накладным расходам на управление.

Для метода «частицы-в-ячейках» естественным атомарным фрагментом является ячейка пространства моделирования. Данными здесь являются значения электромагнитных полей в узлах и частицы внутри ячейки, а вычисления определяются алгоритмом моделирования. Из ячеек конструируются минимальные фрагменты (например, строка, столбец или слой), из которых собирается все пространство моделирования.

Сборочный подход обладает рядом преимуществ. С точки зрения отдельного вычислительного узла, каждый готовый фрагмент можно запускать на отдельном процессоре или ядре процессора, эффективно используя вычислительные мощности архитектуры с общей памятью. Кроме того, имеется возможность существенно ускорить вычисления, если удастся вместить весь фрагмент в кэш-память.

С точки зрения распределенных вычислений важным преимуществом является возможность выполнять пересылку данных на фоне счета: если один фрагмент не может быть запущен на исполнение в связи с ожиданием данных из другого узла, на исполнение запускается другой фрагмент. Также на уровне фрагментов естественным образом реализуется балансировка загрузки.

Отметим, что все перечисленные преимущества для фрагментированных программ могут быть обеспечены системой программирования автоматически. Для этого необходимо, чтобы в каждый момент времени было достаточно много (в несколько раз больше, чем число исполнительных устройств) готовых к исполнению фрагментов.

В качестве модели вычислений для фрагментированных программ наиболее подходящей является асинхронная модель [Котов, Нариньяни, 1966; Котов, 1972; Котов, 1980], поскольку она:

1) представляет программу в виде множества *A*-блоков, что близко к представлению в виде множества фрагментов;

2) позволяет исполнять *A*-блок сразу после того, как готовы его входные данные, т. е. поддерживает очередь готовых фрагментов заполненной;

3) при добавлении новых ограничений в *A*-программы (явных или неявных) наблюдается полезная «монотонность» в уменьшении недетерминизма исполнения, что согласуется с укрупнением размеров фрагментов в сборочной технологии.

Асинхронная модель вычислений

Базовая асинхронная модель. Асинхронная модель вычислений впервые была предложена достаточно давно – в середине 60-х гг. В наиболее общем виде ее можно описать следующим образом.

Асинхронная программа (или *A*-программа) – это конечное множество *A*-блоков, определенных над *информационной IM* и *управляющей CM* памятями. Память состоит из переменных с неразрушающим чтением и записью, стирающей предыдущее содержимое переменной. Информационная память используется для хранения данных решаемой задачи, а управляющая – для организации управления в программе. Каждый *A*-блок *A* представляется тройкой (*T*, *O*, *C*), где *T – спусковая функция* (или *триггер-функция*), представляющая собой некоторый предикат над *CM*; *O – операция* над *IM*, вычисляющая по входным параметрам in(*A*) значения выходных параметров out(*A*); *C – управляющий оператор*, изменяющий значение управляющей памяти и организующий управление в программе.

Вычисления по асинхронной программе организуются следующим образом.

1. Для всех *A*-блоков вычисляются их триггер-функции. *A*-блок с истинным значением триггер-функции объявляется готовым к исполнению. Формируется множество готовых к исполнению *A*-блоков.

2. Выполняется некоторое подмножество готовых к исполнению *A*-блоков. Выполнение *A*-блока состоит в вычислении его операции и управляющего оператора. После завершения выполнения *A*-блоков переходим на шаг 1.

3. Выполнение *A*-программы завершается, когда выполнение всех стартовавших *A*-блоков завершилось и значения триггер-функций всех *A*-блоков ложны.

Помимо преимуществ с точки зрения реализации сборочной технологии программирования, асинхронная модель обладает также следующими важными достоинствами.

1. Позволяет сохранять весь присущий задаче естественный параллелизм (представлять программу в максимально непроцедурной форме).

2. Благодаря механизму спусковых функций, который лишь разрешает, а не предписывает исполнение *A*-блоков, предоставляет возможность организации гибкого взаимодействия между параллельной программой и вычислительной системой.

3. Полностью отделяет вычисления от управления.

4. Достаточно легко моделирует другие модели вычислений.

5. Для нее разработаны формальные алгоритмы синтеза параллельных программ на основы вычислительных моделей.

Асинхронная модель с массовыми A-блоками. В работе [Вальковский, Малышкин, 1988] асинхронная модель получила дальнейшее развитие. Для представления массовых вычислений в нее было введено понятие массового *A*-блока. Не вдаваясь в детали, новое определение *A*-блока можно сформулировать следующим образом.

Пусть $F = \{A_1, A_2, \ldots, A_m\}$ – конечное множество *A*-блоков; *N* – множество натуральных чисел, $G \subseteq F \times N$. Элемент $(A, i) \in G$ называется *i*-м *экземпляром* A, $A \in F$ (каждый элемент множества *F* может иметь несколько экземпляров; количество экземпляров у разных элементов может отличаться). *A*-блок *A* называется *простым*, если $N_A = \{k \mid (A, k) \in G\}$ – одноэлементное множество, и *массовым*, если N_A содержит более одного элемента. Множество N_A называется *областью применимости A*-блока *A*. Каждому экземпляру *A*-блока *A* в зависимости от номера *i* ставится в соответствие единственное конечное множество входных in(*A*(*i*)) и единственное конечное множество выходных out(*A*(*i*)) переменных.

Исполнение массовых *A*-блоков уточняется. Значение триггер-функции проверяется для всех невыполняющихся экземпляров. Операция *A*-блока может начать исполняться на всех входных наборах переменных, для которых значение триггер-функции истинно. После исполнения операции выполняется соответствующий управляющий оператор.

При таком определении *A*-блока появляется возможность применять *A*-блоки с массивами.

Проблемы организации управления в асинхронной модели вычислений. Рассмотрим проблемы асинхронной модели вычислений с точки зрения ее реализации на мультипроцессоре или многоядерном процессоре. Такой подход представляется оправданным, поскольку большинство современных распределенных вычислительных систем в качестве узла имеют машину с SMP-архитектурой, а учитывая направления развития микропроцессоров (увеличение количества ядер), эта тенденция будет только возрастать.

Пусть необходимо умножить матрицу *A* на матрицу *B*, а результат поместить в матрицу *D*. Формально решение задачи можно записать в виде следующих формул:

$$c(i, j, k) = a(i, k) * b(k, j), \quad (1)$$

$$d(i, j) = c(i, j, 1) + c(i, j, 2) + \ldots + c(i, j, n). \quad (2)$$

Пример матриц для $i = 1 \ldots 4$, $j = 1 \ldots 4$, $k = 1 \ldots 4$ приведен на рис. 1. Для удобства матрица *C* показана в разрезе по измерению *i*.

Алгоритм умножения матриц допускает очень высокую степень асинхронности исполнения: каждое умножение элемента матрицы *A* на элемент матрицы *B* может быть выполнено

независимо. Сложения результатов умножения каждой строки на каждый столбец также могут быть выполнены независимо.

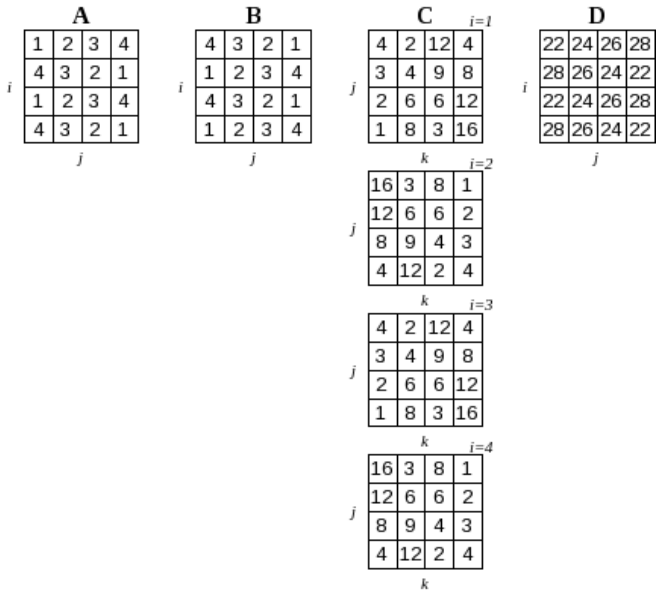

Рис. 1. Результат работы программы умножения матриц по формулам (1) и (2)

С целью сохранения естественного параллелизма алгоритма в асинхронной модели представим его в виде двух массовых *A*-блоков: *A*1 вычисляет матрицу *C* (каждый экземпляр *A*1 вычисляет один элемент матрицы *C* по формуле (1)), а *A*2 – матрицу *D* (каждый экземпляр *A*2 вычисляет один элемент матрицы *D* по формуле (2)), причем каждый из экземпляров второго *A*-блока может быть выполнен только после соответствующих экземпляров (вычисляющих строку матрицы *C*) первого *A*-блока. Из-за высокой степени асинхронности исполнение данной программы потребует больших накладных расходов на организацию управления. Рассмотрим их более внимательно.

Расходы на выполнение A-блоков. Для вычисления всех необходимых произведений при умножении двух матриц размером 4×4 каждая необходимо выполнить 64 экземпляра *A*-блока. Для запуска *A*-блока необходимо проверить истинность его триггер-функции, а после исполнения *A*-блока – выполнить управляющий оператор. Таким образом, кроме 64 операций умножения, дополнительно будет выполнено 64 проверки триггер-функций и 64 управляющих оператора (причем каждый управляющий оператор может содержать несколько операций), что потребует больше времени, чем операции умножения.

Расходы на управляющую память. Если каждый управляющий оператор использует хотя бы по одной переменной (например, для управления разрешением запуска *A*-блока), то объем памяти, затрачиваемой на организацию вычислений будет сопоставим с объемом памяти, необходимым для хранения данных. В частности, в приведенном примере для выделения по одной переменной для управления каждым *A*-блоком необходимо 80 переменных (64 операции умножения плюс 16 редукционных операций суммирования), в то время как под все данные (включая промежуточные матрицы) необходимо 112 переменных, т. е. расходы на управляющую память составляют 71 % от полезного объема данных.

Таким образом, хотя теоретически высокая степень асинхронности позволяет максимально подстроить программу под имеющиеся ресурсы, фактически это ведет к очень большим расходам на организацию управления, сводящим на нет преимущества от распараллеливания.

Один из подходов к решению данной проблемы заключается в разработке вычислителей, поддерживающих асинхронную модель на аппаратном уровне (например, такая попытка была предпринята в проекте МАРС [Котов, Марчук, 1983]), однако они не получили распространения. Другой подход состоит в том, чтобы подобрать размер (или «вычислительный вес») фрагмента вычислений (гранулы параллелизма) таким образом, чтобы сделать накладные расходы в процентном отношении приемлемыми. В частности, такой подход применяет-

ся в Т-системе [Moskovsky, Roganov, Abramov, 2007]. В простейшем случае (шаблон типа «Map») Т-система умеет агрегировать несколько операций в одну, в остальном же задача подбора оптимального размера фрагмента вычислений (Т-функции в терминах Т-системы) возложена на программиста.

В данной работе сделана попытка автоматизировать процесс подбора оптимального размера фрагмента вычислений. Программист лишь задает требуемый размер, а сам фрагмент конструируется автоматически. Подход не является универсальным, однако хорошо подходит для задач с регулярными структурами данных и вычислений. К таким задачам, в частности, относится метод частиц, описанный выше.

Асинхронная модель с группировкой вычислений

Общий подход к организации группировки. Пусть задана некоторая *A*-программа. Идея автоматизированного подбора оптимального размера фрагмента вычислений заимствована из сборочной технологии. Исходные экземпляры *A*-блоков будем считать атомарными фрагментами вычислений (по аналогии с атомарными фрагментами в сборочной технологии), из которых с помощью объединения нескольких экземпляров в один, более крупный, будем конструировать *группы* (минимальные фрагменты вычислений). Экземпляры, включенные в одну группу, будем называть *элементами* этой группы.

Оставляя пока в стороне вопросы объединения в группу триггер-функций экземпляров и операций экземпляров (поскольку они представляют собой «полезные» вычисления и сэкономить на них невозможно), отметим, что механическое объединение управляющих операторов экземпляров в управляющий оператор группы (когда управляющий оператор группы представляет собой последовательность управляющих операторов экземпляров), очевидно, не даст существенного эффекта, так как объем «управляющих» вычислений не сократится.

Чтобы при объединении экземпляров иметь возможность сокращать «управляющие» вычисления, необходимо точно знать, какие вычисления производятся в управляющих операторах. Ни базовая, ни асинхронная модель с массовыми *A*-блоками не делают никаких предположений относительно этих вычислений, поэтому для эффективной реализации группировки необходима конкретизация асинхронной модели.

Конкретизация асинхронной модели. Прежде всего, необходимо выбрать способ управления в программе. Наиболее очевидным является подход, при котором каждой входной переменной каждого *A*-блока сопоставляется признак готовности этой переменной (например, он используется в [Вальковский, Малышкин, 1988; Лельчук, 1984]). Если признаки готовности всех входных переменных *A*-блока истинны, значит, он готов к исполнению. Однако такой подход имеет существенные недостатки. Во-первых, для хранения признаков готовности необходимо дополнительно выделять большой объем памяти, что является неприемлемым для крупных вычислительных задач. Во-вторых, при записи значения в переменную необходимо изменять соответствующим образом и признак готовности переменной, т. е. вместо одной записи в память получается две. В-третьих, для проверки условия готовности требуется много времени, если на вход *A*-блока поступает большой объем данных.

Альтернативный вариант заключается в том, что признак готовности вводится не для каждой входной переменной *A*-блока, а для всего *A*-блока в целом. Аналогичный подход используется при аппаратной реализации некоторых вычислителей с управлением потоком данных, однако там его реализация (с программной точки зрения) тривиальна, поскольку отсутствуют структурные переменные (например, массивы), массовые операции и группировка операций (о группировке см. в следующем разделе). Этот подход выбран в качестве базового в данной работе.

Вначале конкретизируем управление для базовой асинхронной модели. Каждому *A*-блоку *A* поставим в соответствие некоторое число $N(A)$, равное количеству входных переменных *A*-блока *A*. Это число будем называть *числом зависимостей A*-блока *A*. В управляющей памяти для каждого *A*-блока *A* выделим по одной переменной, которую будем называть *счетчиком зависимостей A* и обозначать $\mathrm{dep}(A)$. Будем считать, что в начальном состоянии памяти $\mathrm{dep}(A) = N(A)$.

Исполнение *A*-программы уточняется следующим образом.

1. Проверка триггер-функций происходит только для тех *A*-блоков, чьи счетчики зависимости равны нулю. Поскольку данное условие гарантирует, что все входные переменные *A*-блока вычислены, то в случае ложности триггер-функции из дальнейших проверок данный *A*-блок исключается.

2. Управляющий оператор *A*-блока осуществляет уменьшение счетчиков зависимости у всех *A*-блоков, входные переменные которых вычислены операцией *A*-блока.

Заметим, что у *A*-блока могут быть входные переменные, которые не вычисляются другими *A*-блоками (начальные данные). Такие переменные не учитываются при вычислении числа зависимостей *A*-блока (в противном случае его исполнение никогда не начнется).

Использование счетчиков зависимостей для определения готовности *A*-блоков позволяет сократить объем управляющей памяти, необходимый для каждого *A*-блока, до одной переменной.

Распространим теперь введенные определения на асинхронную модель с массовыми *A*-блоками. Каждому *i*-му экземпляру *A*-блока $A(i)$ поставим в соответствие некоторое число $N(A(i))$, которое будем называть *числом зависимостей экземпляра*. В управляющей памяти для каждого экземпляра *A*-блока *A* выделим по одной переменной, которую будем называть *счетчиком зависимостей i-го экземпляра A*-блока *A* и обозначать $\mathrm{dep}(A(i))$.

Поскольку в модели с массовыми *A*-блоками имеются структурные переменные (например, записи или массивы), вычисление $N(A(i))$ имеет ряд особенностей. Структурные переменные могут поступать на вход *A*-блока как целиком (например, весь массив), так и по частям (некоторые поля записи). Если структурная переменная целиком будет вносить в значение $N(A(i))$ единицу (как в случае базовой асинхронной модели), тогда возникнет проблема с тем, сколько будет вносить часть (один или несколько компонентов) структурной переменной.

Для ее решения введем вспомогательное определение. Каждой переменной *x* поставим в соответствие некоторое число $\mathrm{M}(x)$, равное единице, если *x* – простая переменная, либо сумме $\mathrm{M}(x(i))$, если *x* – структурная переменная (здесь $x(i)$ – компоненты *x*). Это число будем называть *числом компонентов переменной*. Например, если переменная *x* есть массив из 10 записей, каждая из которых содержит по 2 целых числа, то число компонентов *x* равно 20.

Число зависимости $N(A(i))$ *i*-го экземпляра *A*-блока *A* по определению положим равным сумме чисел компонентов всех входных переменных экземпляра *A*-блока *A* ($N(A(i)) = \sum \mathrm{M}(x)$, где $x \in (A(i))$). Будем считать, что в начальном состоянии памяти $\mathrm{dep}(A(i)) = N(A(i))$. Исполнение *A*-программы уточняется аналогично базовой асинхронной модели.

В качестве примера рассмотрим *A*-программу умножения матриц, состоящую из двух массовых *A*-блоков: *A*1 (выполняет операции умножения) и *A*2 (выполняет операции сложения). Так как элементы матрицы являются числами (простые переменные), то счетчик зависимости каждого экземпляра *A*-блока *A*2 перед началом исполнения программы будет равен четырем ($N(A2(i)) = 1 + 1 + 1 + 1$).

Конструирование управляющего оператора. Для автоматического конструирования управляющего оператора необходимы исходные данные. Оставляя за рамками статьи причины выбора таких данных и способы их описания, далее будем считать, что программистом задано множество *A*-блоков $F = \{A_1, A_2, ..., A_n\}$ с пустым управляющим оператором и предикат $\mathrm{Pr}ev(A_i, A_j)$, принимающий значение ИСТИНА, если *A*-блок A_i должен выполниться раньше *A*-блока A_j (т. е. между ними существует информационная зависимость).

Пусть необходимо сконструировать управляющий оператор для *A*-блока *A*. Как было сказано выше, управляющий оператор каждого *A*-блока должен выполнить уменьшение счетчиков зависимости у всех *A*-блоков, входные переменные которых вычисляет данный *A*-блок.

Конструирование управляющего оператора для базовой асинхронной модели достаточно очевидно и может быть выполнено по следующему алгоритму (для *A*-блока *A*).

1. Формируем множество **M** всех $B_i \in F$, таких, что пересечение $\mathrm{out}(A) \cap \mathrm{in}(B_i)$ не пусто и $\Pr ev(A, B_i)$ принимает значение ИСТИНА.

2. Для каждого B_i из **M** включаем в управляющий оператор A-блока A команду уменьшения $\mathrm{dep}(B_i)$ на величину $|\mathrm{out}(A) \cap \mathrm{in}(B_i)|$.

Для асинхронной модели с массовыми вычислениями, как и в случае с определением числа зависимостей экземпляра, из-за наличия структурных переменных возникает ряд сложностей. Дополнительные трудности создает также то, что простые A-блоки могут вычислять переменные для массовых A-блоков, и наоборот.

Вначале рассмотрим конструирование управляющего оператора для простого A-блока A, вычисляющего одну переменную x для простого A-блока B (зависимость «Простой A-блок – Простой A-блок»). Поскольку x может быть как простой так и структурной переменной, возможно несколько вариантов: A-блок A вычисляет x целиком либо только некоторый компонент x; А-блок B на входе имеет либо x целиком, либо некоторый компонент x. Поэтому в управляющий оператор A-блока A необходимо включить одну команду, уменьшающую значение dep(B) на величину Y, которая рассчитывается по следующим правилам:

- если выходная переменная вычисляется целиком, входная переменная потребляется целиком, то $Y = \mathrm{M}(x)$;
- если выходная переменная вычисляется целиком, входная переменная потребляется частично, то $Y = \mathrm{M}(x(i))$, где $x(i)$ – потребляемый компонент переменной x;
- если выходная переменная вычисляется частично, входная переменная потребляется целиком, то $Y = \mathrm{M}(x(i))$, где $x(i)$ – вычисленный компонент переменной x;
- если выходная переменная вычисляется частично, входная переменная вычисляется частично, то необходимо проверить, совпадает ли вычисленный компонент x(i) с потребляемым компонентом $x(j)$. Если совпадение обнаружено, $Y = \mathrm{M}(x(i)) = \mathrm{M}(x(j))$. В противном случае $Y = 0$.

При конструировании управляющего оператора для простого A-блока A, вычисляющего одну переменную x для экземпляров массового A-блока B (зависимость «Простой A-блок – Массовый A-блок») необходимо рассмотреть два варианта.

1. Входная переменная x A-блока B не является массовой (т. е. одна и та же для всех экземпляров B). В этом случае в управляющий оператор A-блока A необходимо включить i команд, каждая из которых уменьшает соответствующий dep($B(i)$) на величину Y, рассчитываемую аналогично случаю «Простой A-блок – Простой A-блок».

2. Входная переменная x A-блока B является массовой (т. е. каждый экземпляр A-блока B получает на вход свой собственный компонент переменной). В этом случае в управляющий оператор A-блока A необходимо включить одну команду, уменьшающую значение dep($B(i)$) на величину Y, которая рассчитывается по следующим правилам:

- если выходная переменная вычисляется целиком, то в управляющий оператор A-блока A необходимо включить i команд, каждая из которых уменьшает соответствующий dep($B(i)$) на величину $\mathrm{M}(x(i))$;
- если выходная переменная вычисляется частично, то необходимо проверить, попадает ли вычисленный компонент $x(j)$ в область применимости A-блока A. Если попадает, то в управляющий оператор A-блока A необходимо включить одну команду, уменьшающую число зависимости экземпляра, потребляющего вычисленное значение на $\mathrm{M}(x(j))$.

Аналогичным образом рассматриваются зависимости «Массовый A-блок – Простой A-блок» и «Массовый A-блок – Массовый A-блок».

Для A-программы имеет место зависимость «Массовый A-блок – Массовый A-блок». Входная переменная x A-блока $A2$ является массовой, поэтому в управляющий оператор каждого экземпляра $A1$, который вычисляет переменные, потребляемые экземплярами $A2$, будет добавлено по одной команде, уменьшающей значение счетчика соответствующего экземпляра $A2$ на единицу.

При автоматическом конструировании управляющего оператора в него можно встроить проверки готовности всех зависимых A-блоков к исполнению (проверки на равенство нулю их счетчиков зависимостей и истинность их триггер-функций). При таком подходе количест-

во проверок условия готовности каждого *A*-блока не превосходит количества его переменных, причем оно может быть даже меньше, чем число входных переменных *A*-блока в случае, когда один *A*-блок вычисляет для другого *A*-блока сразу несколько входных переменных. Тогда проверка условия готовности будет выполнена один раз – после завершения обновления dep(*A*) зависимого *A*-блока *A*.

Группировка вычислений. Как было отмечено, общий подход к группировке заключается в объединении нескольких экземпляров *A*-блока в более крупный блок, называемый группой. Если *A*-блок простой (состоит из одного экземпляра), то он является группой по определению (остается без изменений). Далее рассматриваются только массовые *A*-блоки.

С учетом введенных уточнений группировку экземпляров массового *A*-блока можно определить следующим образом. Пусть задан *A*-блок *A*, определяющий экземпляры $A_1, A_2, \ldots, A_n$. Разделим экземпляры *A*-блока *A* на группы по *k* элементов в каждой. Входные переменные группы есть объединение входных переменных элементов, выходные переменные группы есть объединение выходных переменных элементов. Вместо *k* счетчиков зависимостей (для каждого элемента) выделяется один счетчик зависимостей на всю группу целиком. Таким образом, группа может быть исполнена, только если все ее входные переменные вычислены.

Объединение триггер-функций. Если исходный *A*-блок имел пустую триггер-функцию, то новый *A*-блок также будет иметь пустую триггер-функцию. Триггер-функции, не влияющие на количество вычисляемых экземпляров (область применимости массового *A*-блока известна до начала вычислений), должны удовлетворять дополнительному ограничению: для массовых *A*-блоков триггер-функция не может зависеть от массовых переменных. Это позволяет исключить ситуации, когда из-за ложности триггер-функции одного экземпляра отменяется выполнение всей группы целиком [1].

Если триггер-функция влияет на количество вычисляемых экземпляров (область применимости *A*-блока до начала вычислений неизвестна), необходимо дополнительно изменять управляющие операторы экземпляров, однако этот вопрос выходит за рамки статьи. На практике такие триггер-функции используются в задачах численного моделирования для организации главного вычислительного цикла в программе, после завершения которого завершается и программа.

Объединение операций. В общем случае между экземплярами *A*-блока *A* может существовать произвольная зависимость (и она может изменяться при изменении входных данных), поэтому объединить их в группы до исполнения программы таким образом, что внутри каждой группы они исполняются один за другим, невозможно, однако для получения выигрыша от объединения это необходимо. Один из возможных подходов к решению этого противоречия – введение ограничений за счет учета специфики предметной области (в данном случае – численного моделирования). Для численных задач достаточно, чтобы область применимости *A*-блока имела вид $N_A = \left\{ i \,\middle|\, i = k \pm b, k = \overline{n...m} \wedge n, m, b \in N \right\}$ (это условие является *ключевым* для данной работы). Тогда экземпляры *A*-блока можно объединять в группы в порядке возрастания номера экземпляра. Порядок вычисления операций экземпляров внутри группы задается пользователем.

Конструирование управляющего оператора. Поскольку область применимости имеет вид $i = k \pm b$, то и компоненты массовых выходных переменных будут вычисляться последовательно. Поэтому при конструировании управляющего оператора группы (в соответствии с алгоритмом, приведенным ранее) в тех местах, где необходимо проверять, попадает ли вычисленная переменная в область применимости зависимого блока, можно сразу искать пересечение диапазонов и уменьшать значение счетчика зависимого экземпляра на число компонентов в пересечении. Другими словами, за счет линейности области применимости *A*-блока *A* проверку всего диапазона удается заменить проверкой на границах.

[1] Если же возможность запуска каждого экземпляра необходимо подчинить некоторому условию, зависящему от массовых переменных, проверку этого условия программист может поместить непосредственно в код операции экземпляра.

Поясним сказанное на примере. Пусть экземпляры *A*-блока *A*(*i*) вычисляют массив целых чисел *x*(*i*), а экземпляры *A*-блока *B*(*i*) потребляют значения массива *x*(*i*). Пусть экземпляры *A*-блока *B* сгруппированы в группы по 5 и нам нужно сгруппировать экземпляры *A*-блока *A* в группы по 3. Рассмотрим управляющий оператор группы *A*(1) (в эту группу входят 1, 2 и 3-й экземпляры исходного *A*-блока). Вместо того, чтобы проверять, что каждый *x*(1), *x*(2), *x*(3) попадают в область применимости *B*(1), ищется пересечение диапазона переменных, вычисляемых группой *A*(1) (этот диапазон *x*[1...3]) с областью переменных, потребляемых блоком *B*(1) (этот диапазон *x*[1...5]), после чего счетчик зависимостей *B*(1) уменьшается на 3 одной командой. Если бы экземпляры *B* были сгруппированы в группы по 2, то потребляемый *B*(1) диапазон был бы другим (*x*[1...2]) и управляющий оператор *A*(1) уменьшал бы его счетчик зависимостей только на 2 (так как мощность пересечения *x*[1...3] с *x*[1...2] равна 2).

После объединения экземпляров *A*-блока *A* в группы управляющие операторы всех *A*-блоков, которые вычисляют переменные для *A*-блока *A*, также необходимо сконструировать заново (теоретически, этого можно и не делать, однако в этом случае накладные расходы на управление запуском групп *A*-блока *A* не уменьшаться).

Заметим, что термин «группа» введен только для удобства, чтобы отличать сконструированные экземпляры от исходных. С точки зрения модели вычислений, вместо исходного массового *A*-блока *A* с *n* экземплярами мы имеем новый массовый *A*-блок *A*' с *m* экземплярами.

Таким образом, программа в сгруппированном виде имеет меньшую асинхронность, но и меньший объем затрат на реализацию управления, т. е. может исполняться более эффективно. Степень асинхронности получаемой программы зависит от размера группы и может варьироваться в широких пределах.

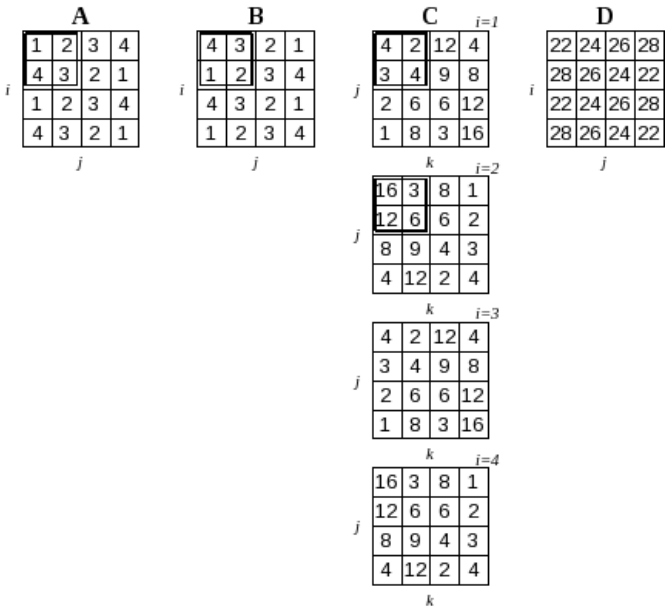

Рис. 2. Умножение матриц с группировкой вычислений

В качестве примера рассмотрим группировку *A*-блока *A*1 программы умножения матриц. Сгруппируем экземпляры *A*-блока *A*1 в группы по два по каждой координате. Тогда вместо 64 экземпляров *A*-блока *A*1 получим 8 групп *A*-блока *A*1', каждая из которых вычисляет параллелепипед размером $2 \times 2 \times 2$. На рис. 2 квадратами выделены данные, которые потребляет / вычисляет первая группы *A*-блока *A*1'.

Используя приведенные ранее алгоритмы, можно автоматически перестроить управление таким образом, что *A*-блок *A*2 будет запускаться только после вычисления соответствующей пары групп *A*-блока *A*1'.

Выигрыш от группировки очевиден. Во-первых, сократилось количество объектов, которыми необходимо управлять во время исполнения программы: вместо 64 экземпляров осталось 8 групп. Во-вторых, сокращение количества управляющих операторов не привело к

увеличению вычислительной сложности оставшихся: если в *A*-программе для запуска одного экземпляра *A*2 необходимо было выполнить 4 операции уменьшения его счетчика зависимостей, то теперь это количество сократилось до 2 операций.

Система программирования Аспект

Система асинхронного параллельного программирования Аспект [Арыков, Малышкин, 2008] разрабатывается в Институте вычислительной математики и математической геофизики СО РАН и является одной из реализаций идей сборочной технологии программирования. Она предназначена для решения задач численного моделирования и состоит из двух основных компонентов: транслятора и исполнительной подсистемы (рис. 3).

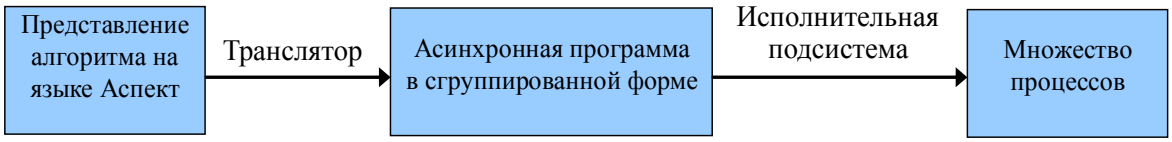

Рис. 3. Компоненты системы Аспект и их взаимодействие

На вход транслятора поступает программа на языке Аспект. Этот язык позволяет представлять алгоритм в виде множества операций и задавать связи между ними. Средствами языка для каждой операции описываются только входные и выходные переменные, а способ вычисления выходных переменных из входных («тело» операции) может быть задан на любом существующем языке (в текущей реализации используется C++). Поэтому Аспект не является языком программирования в обычном смысле, его основная функция – описание способа применения операций к данным, а также задание управления (как потокового, так и прямого). Транслятор осуществляет генерацию асинхронной программы в сгруппированной форме по описанным в разделе 4 алгоритмам.

Исполнительная система осуществляет отображение асинхронной программы в множество процессов. Она поддерживает очередь готовых к исполнению групп, распределяет их между доступными процессами, осуществляет управление памятью и коммуникациями, а также решает ряд сервисных задач.

Схема процесса работы с системой программирования Аспект приведена на рис. 4. Вначале программист разрабатывает программу на языке программирования Аспект. Затем с помощью транслятора эта программа преобразуется в программу на языке C++, которая затем компилируется совместно с исполнительной подсистемой компилятором C++, в результате чего получается готовый исполняемый файл.

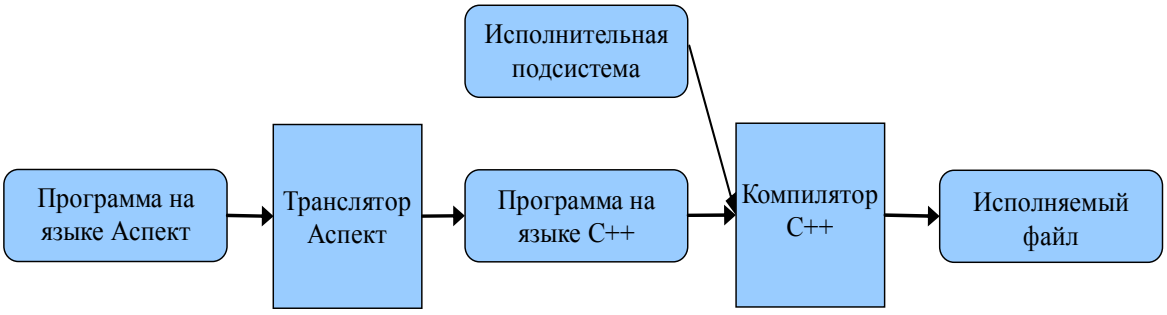

Рис. 4. Процесс получения исполняемого файла в системе Аспект

Рассмотрим результаты тестирования системы Аспект на примере модельной задачи умножения матриц. Для тестирования использовалась следующая конфигурация: Athlon 64 X2 3600+ (2*256 L2), 1024 DDR2 RAM, Windows Vista Ultimate. Результаты тестирования для матриц размером 400 на 400 элементов приведены на рис. 5 (отметка Б/А показывает скорость аналогичной программы, разработанной вручную).

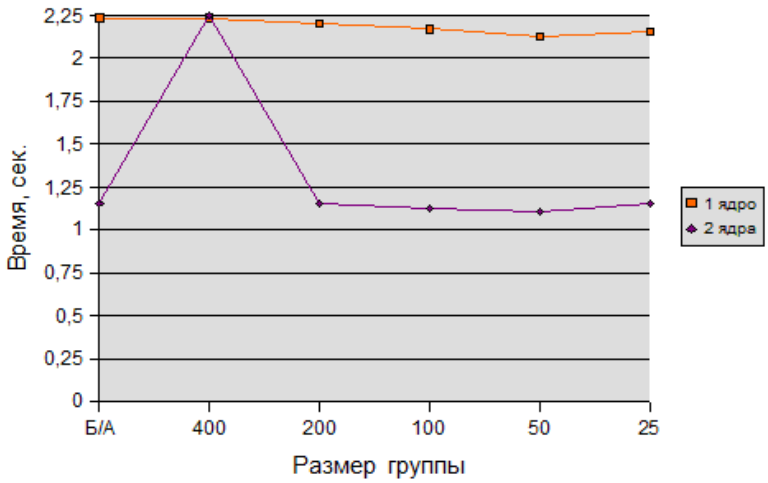

Рис. 5. Результаты умножения матриц с различными размерами групп

Из диаграммы видно, что ускорение достигает значения 1,93 (при размере группы в 50 экземпляров). Это объясняется тем, что, хотя каждое ядро имеет отдельный кэш второго уровня, доступ к оперативной памяти производится через общий контроллер. Таким образом, понятно, что эффективное использование кэш-памяти для многоядерных процессоров имеет первостепенное значение.

Скачок при размере группы в 400 экземпляров происходит потому, что ее размер совпадает с исходным размером матриц, т. е. вся матрица представляет собой одну группу. В этой ситуации для 2-го ядра просто нет работы.

Следует обратить внимание на то, что ось групп не линейна. Например, при размере группы в 100 экземпляров всего получается 80 групп (4*4*4 + 4*4), а при размере в 50 экземпляров – уже 576 групп. Таким образом, при существенном увеличении числа групп скорость счета не возрастает (она даже несколько снижается под влиянием кэш-памяти), что свидетельствует об эффективности работы ядра системы.

Заключение

В статье рассмотрен один из подходов к реализации сборочной технологии программирования. Предложена специализированная асинхронная модель вычислений, а также алгоритмы автоматической генерации программ с заданной степенью непроцедурности, что позволяет программисту менять степень непроцедурности программы, не изменяя ее текста, и в автоматическом / автоматизированном режиме подбирать оптимальную степень непроцедурности программы для ее эффективного исполнения на конкретном вычислителе.

На основе предложенной модели вычислений разработаны и реализованы язык и система асинхронного параллельного программирования Аспект, которую предполагается использовать для решения задач численного моделирования.

Дальнейшие планы по разработке системы Аспект связаны с расширением модели вычислений для ее использования в архитектурах с распределенной памятью, что позволит исполнять программы на кластерах и массивно-параллельных системах, а также с реализацией методов динамической балансировки загрузки для таких архитектур.

Список литературы

Григорьев Ю. Н., Вшивков В. А., Федорук М. П. Численное моделирование методами частиц-в-ячейках. Новосибирск: Изд-во СО РАН, 2004. 360 с.

Вшивков В. А., Краева М. А., Малышкин В. Э. Параллельная реализация метода частиц // Программирование. 1997. № 2. С. 39–51.

Краева М. А., Малышкин В. Э. Алгоритмы динамической балансировки загрузки при реализации метода частиц в ячейках на МИМД-мультикомпьютерах // Программирование. 1999. № 1. С. 47–53.

Kraeva M. A., Malyshkin V. E. Assembly technology for parallel realization of numerical models on MIMD-multicomputers // Future Generation Computer Systems. 2001. Vol. 17. P. 755–765.

Котов В. Е., Нариньяни А. С. Асинхронные вычислительные процессы над памятью // Кибернетика. 1966. № 3. С. 64–71.

Котов В. Е. О практической реализации асинхронных параллельных вычислений // Системное и теоретическое программирование. Новосибирск: ВЦ СО АН СССР, 1972. С. 110–125.

Котов В. Е. О параллельных языках. II // Кибернетика. 1980. № 3. С. 1–10.

Котов В. Е., Марчук А. Г. Некоторые итоги и перспективы развития проекта МАРС // Актуальные проблемы развития архитектуры и программного обеспечения ЭВМ и вычислительных систем. Новосибирск: ВЦ СО АН СССР, 1983. С. 13–23.

Moskovsky A., Roganov V., Abramov S. Parallelism granules aggregation with the T-system // Poc. of the 9th Int. Conf. on Parallel Computing Technologies (PaCT-2007). Lecture Notes in Computer Science. Berlin: Springer, 2007. Vol. 4671. P. 293–302.

Вальковский В. А., Малышкин В. Э. Синтез параллельных программ и систем на вычислительных моделях. Новосибирск: Наука, 1988. 129 с.

Лельчук Т. И. Языковая реализация параллельной асинхронной модели вычислений // Кибернетика. 1984. № 5. С. 32–37.

Арыков С. Б., Малышкин В. Э. Система асинхронного параллельного программирования «Аспект» // Вычислительные методы и программирование. 2008. Т. 9. № 1. С. 205–209.

Материал поступил в редколлегию 16.07.2008

S. B. Arykov, V. E. Malyskin

ALGORITHMS OF ASYNCHRONOUS PROGRAMS CONSTRUCTION WITH PREDEFINED LEVEL OF NON-PROCEDURALITY BASED ON GROUPING METHOD

Problems of asynchronous programs development for parallel implementation of the large scale numerical models are considered. Assembly technology is proposed to be used in order to support program assembling in the asynchronous programming system. This provides automatic implementation of dynamic properties (setting up on the available resources, dynamic load balancing, dynamic resource distribution, etc.) of application program. Special version of asynchronous model of computation is proposed which allows in the wide range to vary the overheads of the program execution with the help of computation fragments grouping. Asynchronous parallel programming system Aspect is considered which implements some principles of the assembly technology on the symmetric multiprocessor or multicores computers.

Keywords: assembly technology of programming, asynchronous languages and programming systems, dynamic program's properties, automation of the parallel realization of the numeric models.