Асинхронное программирование численных задач

С.Б. Арыков

В статье обсуждается система параллельного программирования Аспект, позволяющая разрабатывать асинхронные программы для решения численных задач. Предложены формальная модель вычислений, являющаяся развитием асинхронной модели с массовыми операциями и специализированный язык для фрагментированного представления алгоритмов. Кратко рассмотрены особенности реализации системы программирования Аспект, приведены результаты тестовых испытаний на модельных задачах.

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. Поэтому необходимо развитие высокоуровневых систем программирования, которые, по возможности, скрывают от программиста технические детали, позволяя сосредоточиться на алгоритме решаемой задачи.

Исследовательские работы по повышению уровня систем программирования активно ведутся как в России, так и за рубежом. Среди российских систем можно отметить такие проекты как DVM [1] и mpC [2], избавляющие программиста от необходимости программировать коммуникаций, а также OpenTS [3] и НОРМА [4], выполняющие автоматическую генерацию параллельных программ; среди зарубежных работ выделяются Charm++ [5], ALF [6] и RapidMind [7]. Несмотря на большое количество проектов, до сих пор основными средствами разработки остаются OpenMP и MPI, что говорит об актуальности дальнейших исследований в данном направлении.

В работе рассматривается система параллельного программирования Аспект [8-9], в которой за счет специализации предполагается существенно повысить уровень языка программирования. Отличительными чертами системы Аспект являются использование фрагментированного представления алгоритма, асинхронной модели вычислений и ориентации на задачи численного моделирования.

Фрагментированное представление изначально разрабатывалось для параллельных вычислений и позволяет автоматически генерировать высокоэффективные параллельные программы для выбранной предметной области, а также обеспечивать ряд динамических свойств, таких, как настройка на доступные ресурсы и балансировка нагрузки. Специализированность вводит ряд ограничений, существенных для технической реализации системы.

Далее рассматривается формальная модель вычислений, на базе которой построена система Аспект, приводится обзор языка Аспект и реализации транслятора и исполнительной подсистемы, а также приводятся результаты тестовых испытаний на модельных задачах.

2. Модель вычислений

2.1 Представление алгоритма

Процедурное представление алгоритма плохо подходит для высокоуровневой системы параллельного программирования, так как накладывает чрезмерно жесткое прямое управление на алгоритм, скрывает его естественный параллелизм, и, таким образом, существенно усложняет решение задачи по распределению ресурсов.

Фрагментированное представление позволяет преодолеть указанный недостаток. Оно заключается в представлении алгоритма в виде множества *фрагментов данных* и *фрагментов кода*. Фрагмент кода получает на вход набор *входных фрагментов данных*, на основе которых вычисляет набор *выходных фрагментов данных*. Подстановка фрагментов данных в качестве

параметров фрагмента кода называется *применением* фрагмента кода к фрагментам данных (один и тот же фрагмент кода может применяться к различным фрагментам данных). Совокупность фрагмента кода и его входных и выходных фрагментов данных называется *фрагментом вычислений*. На множестве фрагментов вычислений задаётся частичный порядок (*управление*).

Исполнение фрагментированной программы состоит в исполнении фрагментов вычислений в любом порядке, не противоречащем заданному управлению, при этом каждый фрагмент вычислений получает свои ресурсы в момент назначения на исполнение, порождает новый процесс программы и может мигрировать с одного процессора на другой.

Поясним использование фрагментированного подхода на простом примере. Пусть имеются три матрицы *A*, *B* и *C* размером 9x9 каждая, матрица *C* инициализирована нулями. Необходимо разработать фрагментированный алгоритм умножения матриц.

Разобьем каждую матрицу на подматрицы размера 3x3 (рис. 1). Каждая подматрица есть фрагмент данных, который будем обозначать именем матрицы с соответствующим номером. Например, $C_{(1,1)}$ обозначает первый фрагмент матрицы *C*.

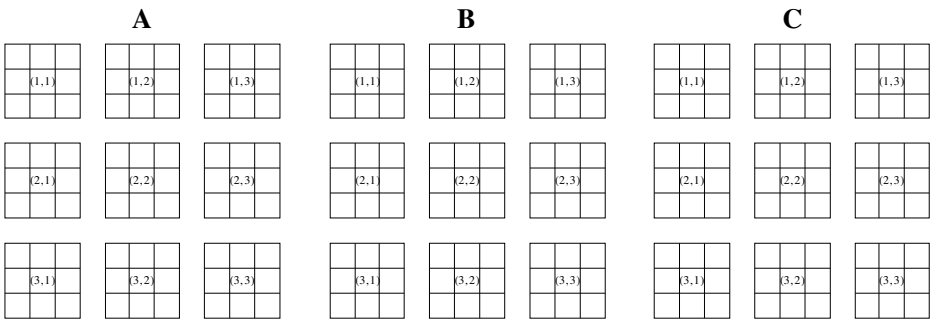

Рис. 1. Фрагментация задачи умножения матриц

Далее, зададим фрагмент кода *F*, который будет получать на вход два фрагмента данных и выполнять умножение первого фрагмента на второй по правилу умножения матриц:

$$c_{ij} = \sum_k a_{ik} b_{kj} \qquad (1)$$

(здесь c_{ij} – элемент в результирующем фрагменте). Тогда произведение матрицы *A* на матрицу *B* можно вычислить по следующей формуле:

$$C_{(i,j)} = \sum_k A_{(i,k)} B_{(k,j)}$$

(здесь $C_{(i,j)}$ – фрагмент матрицы *C*). Сложение результатов можно выполнять как отдельным фрагментом кода, так и встроить его во фрагмент кода *F* (далее предполагается именно этот вариант).

Теперь необходимо применить фрагмент кода к соответствующим фрагментам данных, т. е. задать фрагменты вычислений. Будем обозначать фрагмент вычислений буквой фрагмента кода, за которой в скобках перечислены фрагменты данных, используемые фрагментом кода. Тогда, чтобы полностью вычислить $C_{(1,1)}$, необходимо выполнение трёх фрагментов вычислений: $F(A_{(1,1)}, B_{(1,1)}, C_{(1,1)})$, $F(A_{(1,2)}, B_{(2,1)}, C_{(1,1)})$, $F(A_{(1,3)}, B_{(3,1)}, C_{(1,1)})$. Поскольку каждый из них осуществляет запись во фрагмент данных $C_{(1,1)}$, для исключения ситуации гонок данных необходимо задать порядок исполнения фрагментов вычислений. Например, можно потребовать, чтобы $F(A_{(1,2)}, B_{(2,1)}, C_{(1,1)})$ исполнялся после $F(A_{(1,1)}, B_{(1,1)}, C_{(1,1)})$, а $F(A_{(1,3)}, B_{(3,1)}, C_{(1,1)})$ после $F(A_{(1,2)}, B_{(2,1)}, C_{(1,1)})$. Аналогично формируются фрагменты вычислений для нахождения остальных фрагментов данных матрицы *C*, в результате чего получается фрагментированная программа. Заметим, что фрагменты вычислений, вычисляющие различные фрагменты данных матрицы *C*, могут исполняться параллельно.

Из приведённого примера можно сделать несколько важных выводов:

1. Алгоритм, использованный для реализации фрагмента кода, отличается от формулы (1) дополнительной операцией сложения. Это отличие не случайно: фрагментация алгоритма может потребовать его существенного изменения и, следовательно, не может быть выполнена автоматически.
2. Фрагментировать алгоритм можно различными способами, при этом полученная программа будет иметь разную степень непроцедурности. Например, для матриц размером 8x8 можно использовать фрагменты 2x2, 2x4, 4x4 и др. Важно лишь, чтобы из всех фрагментов данных можно было «собрать» исходные структуры данных.

Алгоритмы различных предметных областей могут фрагментироваться с различным качеством. В работе рассматривается фрагментированное представление алгоритмов предметной области «численное моделирование», для которой оно предоставляет ряд существенных преимуществ:

1. Возможность автоматической генерации параллельной программы. При фрагментированном подходе наиболее сложные зависимости между операциями скрываются внутри фрагментов кода, что позволяет формализовать процесс конструирования параллельной программы на основе схемы управления между фрагментами.
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. Это можно сделать благодаря фрагментированной структуре программы.
3. Переносимость между различными архитектурами. Поскольку фрагментированная программа допускает свободу (в рамках заданного управления) в выборе порядка исполнения фрагментов, имеется существенный резерв для адаптации программы к конкретному вычислителю.
4. Возможность накапливать управляющие схемы решения задач в библиотеке. Во фрагментированном представлении управление полностью отдельно от вычислений, что позволяет накапливать в библиотеке не только вычислительные процедуры (фрагменты кода), но и управляющие схемы.

Фрагментированное представление алгоритма близко к блочному представлению, которое используется в высокопроизводительных библиотеках линейной алгебры (ATLAS [10], Plasma[11] и др.). Основная цель блочного представления – оптимизация выбранного численного алгоритма, достигаемая за счёт «дружественности» к кэш-памяти. Управление между блоками явно не выделяется, а сам подход не ориентирован на создание больших прикладных программ.

Размер фрагмента данных определяет общее количество фрагментов вычислений в программе и является важнейшим параметром фрагментированной программы, а потому заслуживает отдельного исследования. Понятно, что фрагментация должна быть достаточно мелкой, чтобы загрузить доступные ресурсы вычислителя, и в то же время достаточно крупной, чтобы расходы на управление оказались приемлемыми. Частично этот вопрос обсуждается в [12].

2.2 Асинхронная модель с массовыми операциями

Фрагментированное представление алгоритма может использоваться в различных моделях вычислений, однако наиболее подходящей является асинхронная модель с массовыми вычислениями [13], поскольку она представляет программу в виде множества *A*-блоков, что близко к представлению в виде множества фрагментов вычислений, а также полностью отделяет вычисления от управления.

Асинхронная программа (или *А-программа*) есть набор $P=(M,A,A_0)$, где *M – память*; *A* – конечное множество *А-блоков*, $A=\left\{A_k \middle| k\in\overline{1,n}\right\}$; A_0 – конечное множество готовых экземпляров.

Память *M* состоит из ячеек с неразрушающим чтением и записью, стирающей предыдущее содержимое ячеек. Выделяется информационная *IM* (используется для хранения данных решаемой задачи) и управляющая *CM* (используется для организации управления в программе) памяти: $M=IM\cup CM$. Информационная память состоит из множества $X=\{x,y,\ldots,z\}$ *простых* переменных и множества $Y=\{\bar{x},\bar{y},\ldots,\bar{z}\}$, разбитого на конечное число счетных, непересекающихся, линейно упорядоченных подмножеств, которые называются *массивами*. Элементы массива $\bar{x}=\{x_1,x_2,\ldots x_n\}$ называются *компонентами* $\bar{x}$; компонент $\bar{x}_i$ обозначается $\bar{x}[i]$. $X\cap Y=\varnothing$.

Каждый *А*-блок A_k образован четверкой (M_k,T_k,O_k,C_k), где M_k – подмножество памяти *M*, используемое *А*-блоком A_k (его входные, выходные и управляющие переменные); T_k – *спусковая функция* (или *триггер-функция*), представляющая собой предикат от переменных $x_1,x_2,\ldots x_m$, $x_i\in M_k$; O_k – *операция*, которая по значениям входных переменных $x_1,x_2,\ldots,x_m$ вычисляет значения выходных переменных $y_1,y_2,\ldots,y_n$, $x_i\in M_k$, $y_j\in M_k$; C_k – *управляющий оператор*, который по значениям входных переменных $z_1,z_2,\ldots,z_l$ вычисляет значения выходных переменных $z_1,z_2,\ldots,z_l$, $z_l\in M_k$ (каждая переменная z_i является входной и выходной одновременно).

Пусть *N* – множество натуральных чисел, $G\subseteq A x\mathrm{N}$. Элемент $(A_k,i)\in G$ обозначается A_k^i и называется *i*-м *экземпляром* *А*-блока A_k. Пусть $N_{A_k}=\left\{i\in\mathrm{N}\middle|(A_k,i)\in G\right\}$. *А*-блок A_k называется *простым*, если N_{A_k} – одноэлементное множество, и *массовым*, если N_{A_k} содержит более одного элемента. Множество N_{A_k} называется *областью применимости* *А*-блока A_k.

Каждому экземпляру A_k^i в управляющей памяти ставится в соответствие переменная логического типа, которую будем называть *признаком завершения* экземпляра A_k^i и обозначать $P(A_k^i)$. Истинность переменной $P(A_k^i)$ означает, что экземпляр A_k^i завершил исполнение. Если переменная ложна, то экземпляр может быть неготовым к исполнению, готовым к исполнению либо исполняться. В начальном состоянии памяти $P(A_k^i)$=ЛОЖЬ для всех A_k^i.

Вычисления по асинхронной программе организуются следующим образом:

1. В пустое множество готовых экземпляров A' включаются все экземпляры из множества A_0.
2. Из множества готовых экземпляров A' выбирается некоторое подмножество A'', экземпляры которого запускаются на исполнение. Исполнение экземпляра $A_k^i\in A''$ состоит в вычислении его операции O_k^i, а затем управляющего оператора C_k^i. Управляющий оператор присваивает признаку завершения $P(A_k^i)$ значение ИСТИНА и добавляет в множество A' экземпляры, готовые к исполнению.
3. Выполнение *А*-программы завершается, когда множество A' становится пустым и не существует исполняющихся экземпляров.

Если в этом определении рассматривать каждую переменную (простую либо компонент массива) как фрагмент данных, каждый *А*-блок – как фрагмент кода, а каждый экземпляр *А*-блока – как фрагмент вычислений, то представление алгоритма в асинхронной модели будет соответствовать фрагментированному представлению. При этом, однако, управление необходимо программировать вручную в управляющих операторах, что является рутинным процессом, ведущим к большому числу ошибок. Поэтому эту работу необходимо автоматизировать.

2.3 Алгоритм генерации управляющих операторов

Пусть задан частичный строгий порядок на множестве экземпляров фрагментов вычислений M. Сконструировать управляющий оператор для экземпляра фрагмента вычислений $s_i \in M$ можно по следующему алгоритму:

1. Генерируем команду, которая установит признак завершения для s_i в значение ИСТИНА.
2. Формируем множество T экземпляров фрагментов вычислений, зависящих от s_i: $T = \{t_j | s_i < t_j, t_j \in M\}$. В результате исполнения s_i каждый фрагмент вычислений t_j потенциально может стать готовым к исполнению.
3. Для каждого фрагмента вычислений t_j:
   - формируем множество U экземпляров фрагментов вычислений, которые должны исполниться до запуска t_j: $U = \{u_k | u_k < t_j, u_k \in M\}$.
   - генерируем команду создания переменной с именем $flag_j$ и присваиваем ей значение ИСТИНА.
   - для каждого элемента u_k генерируем команду, объединяющую значение признака завершения u_k со значением переменной $flag_j$ по принципу И.
   - генерируем набор команд, который:
     1) проверяет значение переменной $flag_j$;
     2) если оно истинно, проверяет значение триггер-функции t_j;
     3) если триггер-функция истина, добавляет фрагмент t_j в очередь готовых фрагментов.

3. Язык программирования Аспект

3.1 Ключевые особенности

Язык программирования Аспект [15] разработан на основе языка ОПАЛ (Описание Параллельных Алгоритмов) [16]. ОПАЛ был создан для описания задач на вычислительных моделях с массивами и содержит наиболее характерные способы задания массовых вычислений. Он позволяет формализовать некоторую предметную область, описав множество допустимых на ней алгоритмов, а затем формулировать задачи для этой предметной области. При этом алгоритм решения каждой задачи может быть синтезирован автоматически [13].

В отличие от ОПАЛ, язык Аспект позволяет задать представление конкретного алгоритма с необходимой степенью непроцедурности и частичным распределением ресурсов. Ключевыми особенностями языка являются:

1. Статическая типизация.
2. Явное описание зависимостей между операциями. В императивных языках на множестве операций задается линейный порядок, что ограничивает возможности параллельного исполнения программ. В отличие от них, Аспект позволяет на множестве операций задать частичный порядок.
3. Частичное распределение ресурсов. Как и в императивных языках, в Аспект допускается повторное присваивание значения переменной. Это означает, что в одной переменной программы могут находиться (в разное время) различные переменные алгоритма.
4. Ориентация на регулярные структуры данных. Язык программирования Аспект ориентирован на решение задач численного моделирования, поэтому основой организации вычислений являются массовые операции, позволяющие эффективно обрабатывать регулярные структуры данных, а основной представления данных – массивы.

5. Отсутствие средств описания вычислений. Аспект позволяет описать данные, операции, и взаимосвязи между ними. Для описания вычислений внутри операций предлагается использовать существующие языки программирования (в настоящее время поддерживается только C++).

Если в императивных языках все команды по умолчанию исполняются последовательно, а участки программы, пригодные для параллельного исполнения, необходимо явно указать, то в языке Аспект используется прямо противоположный подход: по умолчанию считается, что все фрагменты вычислений могут исполняться параллельно, а если необходим порядок (например, из-за зависимости по данным), то его необходимо явно указать.

Детальное описание языка Аспект выходит за рамки статьи, поэтому ниже поясняются только его основные конструкции, необходимые для понимания процесса разработки программ.

3.2 Структура программы и основные конструкции

3.2.1 Структура программы

Упрощённая структура Аспект-программы показана на рис. 2. Каждая Аспект-программа имеет уникальное имя и несколько разделов с объявлениями.

```
program <Имя программы>
preface {
   <Раздел объявлений внешнего языка>
};
data fragments
   <Раздел объявления фрагментов данных>
code fragments
   <Раздел объявления фрагментов кода>
task data
   <Раздел объявления данных задачи>
task computations
   <Раздел объявления вычислений задачи>
task control
   <Раздел описания управления>
end
```

Рис. 2. Структура Аспект-программы

Разделы «code fragments», «task data» и «task computations» являются обязательными. Остальные разделы могут быть опущены.

Раздел «preface» позволяет задать необходимые определения внешнего языка. Для C++ это объявление типов данных, объявление констант, подключение заголовочных файлов и др.

3.2.2 Разделы объявления фрагментов данных и данных задачи

Раздел объявления фрагментов данных состоит из набора строк, каждая из которых имеет следующий синтаксис:

<Тип внешнего языка> <Имя фрагмента данных>;

или

<Тип внешнего языка> <Имя фрагмента данных>[<Индекс1>][<Индекс2>]...[<ИндексN>];

где <Тип внешнего языка> – тип данных внешнего языка (встроенный или объявленный в секции «preface»), <Имя фрагмента данных> – произвольный идентификатор, <Индекс> – число либо константа, объявленная в разделе «preface». Определение первого вида позволяет задать простой фрагмент данных, а определение второго вида – фрагмент-массив.

Раздел объявления данных задачи состоит из набора строк, каждая из которых имеет следующий синтаксис:

<Имя фрагмента данных> <Имя переменной>;

или

<Имя фрагмента данных> <Имя переменной>[Индекс1][Индекс2]...[ИндексN];

Простые числовые типы данных внешнего языка (для C++ это int, float, double) являются фрагментами данных по умолчанию. Таким образом, все переменные (данные задачи) в языке Аспект состоят из фрагментов.

3.2.3 Разделы объявления фрагментов кода и вычислений задачи

Раздел объявления фрагментов кода позволяет задать один или более фрагментов кода, для чего используется следующий синтаксис:

<Имя фрагмента кода> (<Имя фрагмента данных> <Имя переменной>, ...) {
  <Вычисления на внешнем языке>
};

После имени фрагмента кода в скобках задаются имена переменных (параметров). Для каждой переменной указывается её тип. <Вычисления на внешнем языке> – это любой допустимый набор команд внешнего языка, который осуществляет вычисления на основе переданных параметров без побочных эффектов.

Раздел объявления вычислений задачи позволяет задать применение фрагментов кода к фрагментам данных и имеет следующий синтаксис:

<Имя фрагмента вычислений>: <Имя фрагмента кода> (<переменная1>, …, <переменнаяN>);

или

<Имя фрагмента вычислений>[<Индекс1>]...[<ИндексN>]: <Имя фрагмента кода> (<переменная1>, …, <переменнаяN>) where <Индекс1>: <нач. зн.> … <кон. зн.>, ... , <ИндексN>: <нач. зн.> … <кон. зн.>

где <Индекс> – это произвольный идентификатор, <Переменная> – это одна из переменных, объявленных в разделе «task data».

В первом случае задаётся простой фрагмент вычислений (однократное применение фрагмента кода к данным), а во втором случае – массовый фрагмент вычислений.

Каждый экземпляр массового фрагмента вычислений однозначно идентифицируется его индексами. Значения, которые может принимать каждый индекс, задаются после ключевого слова «where». Запись <Индекс>: <нач. зн.> … <кон. зн.> означает, что <Индекс> может принимать все целые значения начиная с <нач. зн.> и заканчивая <кон. зн.>.

3.2.4 Раздел описания управления

Управление в языке Аспект задается определением частичного порядка на множестве фрагментов вычислений. Выделяется два типа управления: между различными фрагментами вычислений и между экземплярами одного массового фрагмента вычислений. Если между фрагментами вычислений имеется информационная зависимость, управление является потоковым, иначе – прямым.

Для описания управления используется следующий синтаксис:

<Имя фрагмента вычислений1> < <Имя фрагмента вычислений2>;

где <Имя фрагмента вычислений1> – имя фрагмента вычислений, который должен выполниться первым, <Имя фрагмента вычислений2> – имя зависимого фрагмента вычислений. Если фрагмент вычислений является массовым, после его имени указываются индексы, идентифицирующие номер экземпляра.

Допускается задание сложного управления, например

((S1 & S2) | S3) < S4

означает, что фрагмент вычислений S4 может выполниться, либо если выполнился фрагмент вычислений S3, либо если выполнились оба фрагмента вычислений S1 и S2.

Все фрагменты вычислений, для которых порядок не задан, могут выполняться одновременно.

3.3 Пример Аспект-программы

В качестве примера на рис. 3 приведён текст Аспект-программы умножения матриц (фрагментация алгоритма рассмотрена в разделе 2.1). Инициализация данных и вывод результата опущены.

```
program MultMatrix
preface {
   const int M = 3;
   const int N = 3;
};
data fragments
   double Frg[M][M];
code fragments
   Mult(Frg X, Frg Y, Frg Z) {
      for(int i=0; i<M; ++i)
         for(int j=0; j<M; ++j)
            for(int k=0; k<M; ++k)
               Z[i][j] += X[i][k]*Y[k][j];
   };
task data
   Frg A[N][N];
   Frg B[N][N];
   Frg C[N][N];
task computations
   S[i][j][k]: Mult(A[i][k], B[k][j], C[i][j])
      where i: 0..N-1, j: 0..N-1, k: 0..N-1;
task control
   S[i][j][k] < S[i][j][k+1];
end
```

Рис. 3. Аспект-программа умножения матриц

В программе заданы матрицы A, B и C, состоящие из NxN элементов. Каждый элемент есть матрица размера MxM (фрагмент данных Frg). Кроме того, определён фрагмент кода Mult, который получает две матрицы (фрагменты данных X и Y) на вход и вычисляет их произведение (фрагмент данных Z) на выходе. Вычисления задаются на языке C++.

В секции «task computation» задан массовый фрагмент вычислений S с областью применимости (0..N-1)x(0..N-1)x(0..N-1), при этом имя S[i][j][k] обозначает соответствующий экземпляр S. Каждый экземпляр осуществляет вычисление фрагмента кода Mult с фрагментами данных, указанными для экземпляра в качестве фактических параметров.

В секции «task control» на множестве фрагментов вычислений задан такой частичный порядок, что каждый *k*-й фрагмент вычислений должен выполниться перед соответствующим (*k*+1)-м фрагментом. Управление отражает тот факт, что для вычисления C[i][j] необходимо просуммировать все матрицы, полученные в результате вычисления произведения фрагментов

A[i][k] на B[k][j] (это делается внутри фрагмента Mult добавлением результата умножения к старому значению C[i][j]), и для исключения «гонки» данных (data race) суммирование необходимо выполнять последовательно.

Все фрагменты вычислений S[i][j][k] с одинаковым индексом k могут исполняться одновременно.

4. Реализация системы программирования Аспект

4.1 Общая архитектура

Система асинхронного параллельного программирования Аспект [8-9] разрабатывается в Институте вычислительной математики и математической геофизики СО РАН. Она предназначена для решения задач численного моделирования на мультипроцессорных/многоядерных вычислителях.

Рис. 4. Разработка программы в системе Аспект

В основе системы Аспект лежит асинхронная модель вычислений, описанная в разделе 2.2. Система состоит из двух основных компонентов: транслятора и исполнительной подсистемы (рис. 4). На вход транслятору подается текст программы на языке Аспект, на выходе генерируется асинхронная программа на C++, которую необходимо скомпилировать вместе с исполнительной подсистемой компилятором C++ (например, g++ или icpc) в исполняемый файл.

4.2 Транслятор

Транслятор имеет стандартную архитектуру и состоит из лексического анализа, синтаксического анализа, семантического анализа, генератора внутреннего представления и генератора кода. На выходе он создаёт один *.cpp файл с асинхронной программой.

В начало файла записывается подключение необходимых библиотек, а также копируется раздел «preface» Аспект-программы. Затем генерируются класс с *A*-программой, после чего генерируются функции aplRun (запрашивает очередной *A*-блок из очереди готовых *A*-блоков и осуществляет его исполнение) и main (осуществляет инициализацию и запуск исполнительной подсистемы).

Исходная Аспект-программа преобразуется в одноимённый класс, наследуемый от виртуального класса AProgram, который предоставляет интерфейс для добавления *A*-блоков в очередь исполнения, а также осуществляет учёт необходимых параметров (количество исполняемых *A*-блоков, количество готовых *A*-блоков и др.).

Для каждого фрагмента данных создаётся отдельный тип, с использованием которого объявляются данные задачи (переменные класса). Для каждого фрагмента кода генерируется метод класса, а также две переменных: признак завершения и переменная синхронизации. Для триггер-функций и управляющих операторов генерируются соответствующие методы (тела управляющих операторов генерируется согласно алгоритму, описанному в разделе 2.3).

Также генерируется специальный метод aplMain, который выполняет инициализацию всех переменных программы и добавляет в очередь на исполнение *A*-блоки, не зависящие от других *A*-блоков.

4.3 Исполнительная подсистема

Исполнительная подсистема состоит из слоя абстрагирования от операционной системы, набора функциональных модулей (менеджер потоков, менеджер памяти, планировщик) и интерфейса системных вызовов.

Слой абстрагирования от ОС включается в себя все подпрограммы, в которых используются API операционной системы (определение доступных ресурсов, работу с потоками, функции для работы со временем и др.). Это позволяет переносить исполнительную подсистему из одной ОС в другую заменой лишь одного, выделенного слоя.

Менеджер потоков управляет распределением работы между процессорами (ядрами). При старте программы по умолчанию создается по одному потоку на каждый процессор (ядро). Каждый поток обращается за очередной порцией работы к планировщику.

Менеджер памяти управляет распределением памяти в системе, осуществляет её выделение/освобождение для очередей и *A*-программ.

Планировщик управляет набором очередей, каждая из которых имеет определённый приоритет и реализована в виде монитора. Алгоритм планирования имеет сложность O(1) и заключается в следующем: при каждом обращении очереди просматриваются в порядке убывания приоритетов; если в текущей очереди имеется готовый *A*-блок, он запускается на исполнение. Таким образом, *A*-блоки из очередей с более высоким приоритетом всегда будут запущены на исполнение раньше, чем *A*-блоки из очередей с более низким приоритетом. Программа завершается, когда все очереди становятся пустыми.

5. Результаты испытаний

Тестовые испытания проводились в Сибирском суперкомпьютерном центре Института вычислительной математики и математической геофизики СО РАН на системе HP ProLiant DL580 G5 со следующей конфигурацией:

- Аппаратное обеспечение: 4 процессора Intel Xeon X7350 (16 ядер), 256 Гбайт RAM, 9 Тбайт HDD.
- Программное обеспечение: Cent OS 5.3 64 bit, Intel C++ compiler professional for Linux 11.1.

Для тестирования были выбраны три задачи: умножение матриц, LU-разложение и явная разностная схема. Все задачи были реализованы на языке Аспект и на C++/OpenMP и компилировались с максимальной оптимизацией (ключ -O3) для архитектуры x86_64. Каждая задача с выбранными параметрами запускалась на исполнение 100 раз и в таблицу заносилось минимальное время исполнения в секундах.

Таблица 1. Результаты измерений производительности задачи умножения матриц

Реализация	Количество ядер				
	1	2	4	8	16
C++/OpenMP	460,21	245,66	156,53	105,14	61,22
Аспект	148,98	74,73	37,52	18,98	9,68

Все использованные матрицы – квадратные, размера 5040x5040. Элементы матриц – вещественные числа с двойной точностью (тип double), инициализировались случайными числами. Размер фрагментов данных в Аспект-реализации – 90x90.

Таблица 2. Результаты измерений производительности задачи LU-разложения

Реализация	Количество ядер				
	1	2	4	8	16
C++/OpenMP	267,14	139,10	90,53	67,27	67,29

Аспект	31,01	15,6	7,93	4,13	2,35

Фрагментированное представление алгоритма умножения матриц было реализовано по описанию раздела 2.1. Аналогичным образом фрагментировался алгоритм явной разностной схемы. Для алгоритма LU-разложения было разработано специальное фрагментированное представление [9].

Таблица 3. Результаты измерений производительности явной разностной схемы

Реализация	Количество ядер				
	1	2	4	8	16
C++/OpenMP	43,87	22,69	12,51	11,39	11,35
Аспект	40,68	21,76	12,24	11,27	11,18

Существенное превосходство Аспект-реализации над C++/OpenMP-реализацией в первых двух тестах объясняется фрагментированным представлением алгоритмов, за счёт которого достигается более эффективное использование кэш-памяти. По этой же причине фрагментированный код лучше масштабируется.

6. Заключение

В статье предложен подход к асинхронному программированию численных задач, основанный на фрагментированном представлении алгоритмов. Разработаны специализированная модель вычислений, являющаяся уточнением асинхронной модели с массовыми операциями и алгоритм автоматического конструирования управляющих операторов. На основе предложенной модели разработаны и реализованы язык и система асинхронного параллельного программирования Аспект.

Проведённое тестирование показало достаточно высокую эффективность системы и подтвердило основную научную гипотезу исследования: по высокоуровневому фрагментированному представлению алгоритма для выбранной предметной области возможна автоматическая генерация достаточно эффективных асинхронных параллельных программ.

Дальнейшие планы исследований связаны с расширением модели вычислений с целью поддержки архитектур с распределенной памятью, а также с реализацией методов динамической балансировки загрузки для таких архитектур.

Литература

1. Коновалов Н.А., Крюков В.А., Сазанов Ю.Л. C-DVM – язык разработки мобильных параллельных программ // Программирование. – 1999. – № 1. – С.46-55.
2. Lastovetsky A.L. Parallel Computing on Heterogeneous Networks. – John Wiley & Sons, 2003. – 350 p.
3. Moskovsky A., Roganov V., Abramov S. Parallelism granules aggregation with the T-system // 9th International Conference on Parallel Computing Technologies. LNCS 4671, pp. 293-302.
4. Андрианов А.Н. Система Норма: разработка, реализация и использование для решения задач математической физики на параллельных ЭВМ. Дис. д-ра техн. наук. 05.13.11. – Москва, 2001. – 158 с.
5. Kale L., Krishnan S. CHARM++ : A Portable Concurrent Object Oriented System Based On C++. Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (Sept-Oct 1993). ACM Sigplan Notes, Vol. 28, No. 10, pp. 91-108.

6. Accelerated Library Framework for Cell Broadband Engine: [http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/41838EDB5A15CCCD002573530063D465], 10.01.2010.
7. McCool M., Wadleigh K., Henderson B., Lin H. Performance evaluation of GPUs using the RapidMind development platform // ACM/IEEE Conference on Supercomputing, Tampa, Florida, Article No. 181. ACM, New York (2006).
8. Арыков С.Б., Малышкин В.Э. Система асинхронного параллельного программирования "Аспект" // Вычислительные методы и программирование. – 2008. – Т.9. – № 1. – С. 205-209.
9. Arykov S., Malyshkin V. Asynchronous Language and System of Numerical Algorithms Fragmented Programming // 10th International Conference on Parallel Computing Technologies (Novosibirsk, Russia, August 31-September 4, 2009). LNCS 5698, pp. 1-7.
10. Automatically Tuned Linear Algebra Software (ATLAS): [http://math-atlas.sourceforge.net/], 10.01.2010.
11. Buttari A., Langou J., Kurzak J., Dongarra J. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures // Parallel Computing. – 2009. – Vol. 35, Issue 1. – P. 38-53.
12. Арыков С.Б., Малышкин В.Э. Алгоритмы конструирования асинхронных программ заданной степени непроцедурности методом группировки // Вестн. Новосиб. гос. ун-та. Серия: Информационные технологии. – 2009. – Т. 7, вып. 1. – С. 3-15.
13. Вальковский В.А., Малышкин В.Э. Синтез параллельных программ и систем на вычислительных моделях. – Новосибирск: Наука, 1988. – 129 с.
14. Арыков С.Б. Группировка данных в системе асинхронного параллельного программирования Аспект // Труды международной научной конференции «Параллельные вычислительные технологии (ПаВТ'2009)» (Нижний Новгород, 30 марта – 3 апреля 2009 г.). – Челябинск: ЮУрГУ, 2009. – С. 357-363.
15. Арыков С.Б. Язык программирования Аспект // Известия Томского политехнического университета. – 2008. – Т. 313. – № 5. – С. 89-92.
16. Малышкин В.Э. ОПАЛ – язык описания параллельных алгоритмов / В кн. Теоретические вопросы параллельного программирования и многопроцессорные ЭВМ. – Новосибирск: ВЦ СО АН СССР, 1983. – С. 91-109.