Функционально-потоковое параллельное программирование при асинхронно поступающих данных

А.И. Легалов, А.В. Редькин, И.В. Матковский

Рассматриваются особенности поведения функционально-потоковых параллельных программ в ситуациях, когда данные могут поступать в неопределённом порядке и в заранее неизвестные моменты времени. Это ведет к специфическим ситуациям для реализации которых необходимо использовать нетрадиционные методы структуризации данных и управления параллельными вычислениями. Рассматривается использование асинхронных списков обеспечивающих написание программ, параллелизм которых динамически изменяется в широких пределах в зависимости от времени выполнения операций преобразования данных и времени передачи данных между этими операциями. Приводятся особенности выполнения вычислений в ходе взаимодействия нескольких функций, использующих асинхронные списки. Предлагается архитектура событийного процессора, поддерживающего выполнение функционально-потоковых параллельных программ.

1. Введение

Использование параллельных вычислений в настоящее время приобретает массовый характер. С одной стороны это связано с применением многоядерных процессоров, что ведет к параллелизму в персональных компьютерах. С другой, постоянно пополняется парк параллельных вычислительных систем (ПВС), применяемых в научных и инженерных расчетах. Массовость предъявляет дополнительные требования к системам программирования, стимулируя развитие подходов к написанию кода, напрямую не зависящих от используемых вычислительных систем.

В настоящее время в параллельном программировании используются различные языки и инструменты, что обусловлено разнообразием присутствующих архитектурных решений. Ряд систем параллельного программирования получили широкое распространение. Например, для многоядерных систем и ПВС с общей памятью используются многопоточные библиотеки и система программирования OpenMP. При разработке программ для систем с распределенной памятью наибольшую популярность получил пакет MPI. Вместе с тем, программы, созданные с применением таких средств, эффективно выполняются только на соответствующих им вычислительных машинах.

Одним из путей обеспечения переносимости является использование архитектурно независимых методов параллельного программирования [1-2]. Программа может писаться так, как будто она использует неограниченные вычислительные ресурсы. После разработки и отладки логики функционирования можно переходить к наложению ограничений, то есть, к решению вопросов архитектурной привязки. В этом случае исходная программа является базовой спецификацией, обеспечивающей более быстрое решение вопросов, связанных с верификацией и отладкой. Разделение на отдельные этапы процессов отладки логики программы и ее архитектурной привязки позволяет не решать эти две проблемы одновременно, что в целом повышает эффективность процесса разработки. Развитие архитектурно-независимых систем программирования обуславливается как совершенствованием соответствующих языков, так и вычислительных систем, обеспечивающих выполнения программы. В частности, языки программирования должны содержать дополнительные методы управления данными, обычно не поддерживаемые в традиционных языках. Вычисления обычно реализуется в соответствующих виртуальных машинах, являющихся промежуточным слоем между языковыми средствами и архитектурами вычислительных систем. Несмотря на то, что на данном этапе развития подобная виртуализация не обеспечивает высокой эффективности, это направление имеет право на существование, так как направлено на поиск новых путей развития информационных технологий.

В работе рассматриваются подходы к управлению данными и организация вычислений в рамках системы программирования, обеспечивающей трансляцию и выполнение функциональ-

но-потоковых параллельных программ, написанных на языке «Пифагор» [1-2] в виртуальной системе, названной событийным процессором [3]. Событийный процессор обеспечивает обход управляющего графа программы, построенного по ее информационному графу. Управляющий граф позволяет управлять вычислениями и проводить эквивалентные преобразования, обеспечивающие изменение стратегии управления [4] без изменения зависимостей между операциями обработки данных. Разработка такого процессора поддерживает архитектурно независимую отладку параллельных программ, а отделение информационного графа от управляющего дает возможность настройки механизма управления вычислениями после выполнения отладки.

2. Управление вычислениями с использованием асинхронных списков

Для реализации управления на основе асинхронно поступающих данных в функциональную модель потоковых вычислений добавлен асинхронный список [5-6]. Его особенностью является упорядочение элементов в соответствии с последовательностью их порождения. По сравнению с обычным списком данных, выдающим сигнал готовности после поступления всех его элементов, асинхронный список сигнализирует только о появлении первого значения. Это накладывает определенную специфику на его интерпретацию. Можно считать, что об асинхронном списке достоверно известно следующее: он не готов, или содержит как минимум один готовый элемент (а об остальных данных ничего неизвестно), или является пустым. Неготовность асинхронного списка определяется отсутствием сигналов, информирующих о наличии сформированных внутри него данных. В этом случае, в соответствии с принципами потокового управления по готовности данных, такой список не может быть обработан. Выдача сигнала подтверждает готовность асинхронного списка к вычислениям, информируя при этом об одной из двух ситуаций: список содержит хотя бы один элемент или он пуст. Тогда определение внутреннего состояния возможно с использованием функции, определяющей длину асинхронного списка. Длина пустого списка равна нулю.

Длина непустого асинхронного списка всегда равна единице. Это объясняется спецификой его интерпретации. Как только внутри списка появляется хотя бы один вычисленный элемент, происходит выдача сигнала, подтверждающего готовность списка к последующей обработке. Даже при одновременном порождении внутри асинхронного списка нескольких элементов будет выдаваться только один сигнал. В этой ситуации первый элемент списка определяется внутренним арбитром, стратегия работы которого не имеет существенного значения. В данной ситуации о наличии одного элемента достоверно известно. Для того чтобы определить, есть ли в асинхронном списке другие данные, необходимы дополнительные операции.

Наряду с определением длины, над асинхронным списком допустимы операции чтения первого элемента и формирования нового асинхронного списка, не содержащего первый элемент. Это, в соответствии с концепциями событийного управления, позволяет описывать асинхронный параллелизм как процесс обработки последовательно наступающих событий.

Реализация асинхронного списка в языке программирования «Пифагор» позволяет писать параллельные программы в несколько ином стиле, при котором учитывается асинхронное поступление данных. В частности, сложение элементов одномерного массива может быть реализовано следующим образом:

```
// Функция, возвращающая сумму элементов асинхронного списка
A_VecSum<< funcdef A
      // Формат аргумента: A=asynch(x1, x2, … , xn)
      // где x1, x2, … , xn - числа
{
  x1<< A:1;      // Поступивший элемент данных
  tail_1<< A:-1;  // Хвост асинхронного списка
  // Проверка на пустоту остатка списка
  [((tail_1:|, 0):[=, !=]):?]^  (
     x1, // В списке, только один элемент, определяющий сумму
     {  // Выделение второго аргумента с последующим суммированием
        block {
           x2<< tail_1:1;      // второй аргумент
           s<< (x1,x2):+;      // сумма двух поступивших элементов
```

```
            tail_2<< tail_1:-1;  // "хвост" от "хвоста"
            // Рекурсивная обработка оставшихся элементов
            [((tail_2:|, 0):[=, !=]):?]^
            (s, { asynch( tail_2:[], s):A_VecSum } ):. >>break
          }
        }
  ):. >>return;
}
```

Использование асинхронных списков позволяет разрабатывать программу, которая, в зависимости от временных соотношений между ее операциями, может интерпретироваться как набор альтернативных алгоритмов, описывающих одну и ту же задачу. Например, если интервал времени Δt_{dat} между порождением данных внутри асинхронного списка больше времени выполнения операции сложения Δt_{add} (от момента чтения данных из списка до засылки их во вновь формируемый асинхронный список), то суммирование будет выполняться последовательно. Данная ситуация проиллюстрирована на рис. 1.

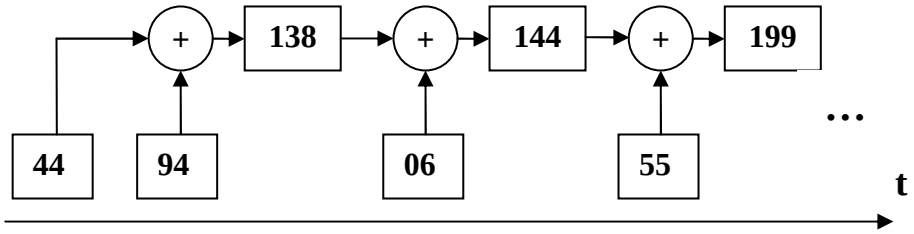

Рис. 1. Последовательное суммирование при $\Delta t_{dat} \geq \Delta\, t_{add}$

В том случае, если интервал времени поступления данных станет меньше времени их сложения, начнет появляться параллелизм в выполнении этих операций. Количество параллельных потоков будет кратно отношению времени Δt_{add} к Δt_{dat}. Например, при $\Delta t_{add}/\Delta t_{dat} = 2$ будет динамически сформировано два потока операций сложения (рис. 2).

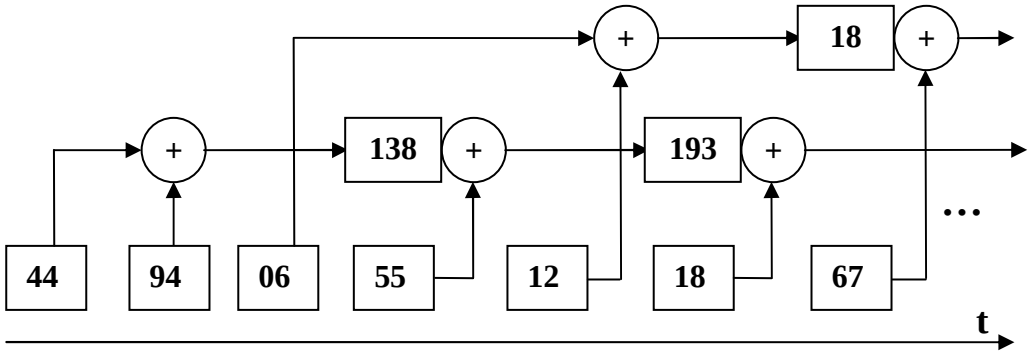

Рис. 2. Формирование двух параллельных потоков при $\Delta t_{add} = 2\Delta t_{dat}$

Если Δt_{add} значительно превышает Δt_{dat}, вычисления будут протекать по каскадной схеме, обеспечивающей максимальный параллелизм. Возможны также ситуации, когда через асинхронные списки происходит взаимодействие нескольких функций. В этом случае динамически осуществляется их конвейеризация, что не наблюдается при использовании традиционных функций, синхронизируемых по входным данным. Подобные методы программирования повышают гибкость разрабатываемых программ и их адаптацию под различные вычислительные ресурсы. Однако, для обеспечения подобной гибкости требуются нетрадиционные методы управления вычислениями.

3. Внутреннее представление потоковой программы

Программа, выполняемая событийным процессором, состоит из двух графов: информационного и управляющего, которые формируются в ходе трансляции исходной программы. Процесс трансляции протекает в два этапа. На первом осуществляется трансляция исходного текста

в информационный граф, заданный во внутреннем представлении. Затем по полученному информационному графу формируется управляющий граф. Процесс трансляции можно рассмотреть на примере вычисления абсолютной величину числа:

```
Abs<< funcdef X {[((X,0):[<,>=]):?]^({X:-}, X):. >>return}
```

Программа на данном языке является информационным графом, определяющим взаимосвязь между используемыми функциями обработки данных и передаваемыми по связям величинам, получаемыми в ходе вычислений. Информационный граф, задающий эквивалентное представление рассматриваемой программы приведен на рис. 3. Стрелки указывают пути передачи данных от одной операции к другой. Контур, заданный пунктирной линией, определяет задержанный список (в программе представлен выражением `{X:-}`). Его использование позволяет без необходимости не выполнять вычисления в альтернативных ветвях.

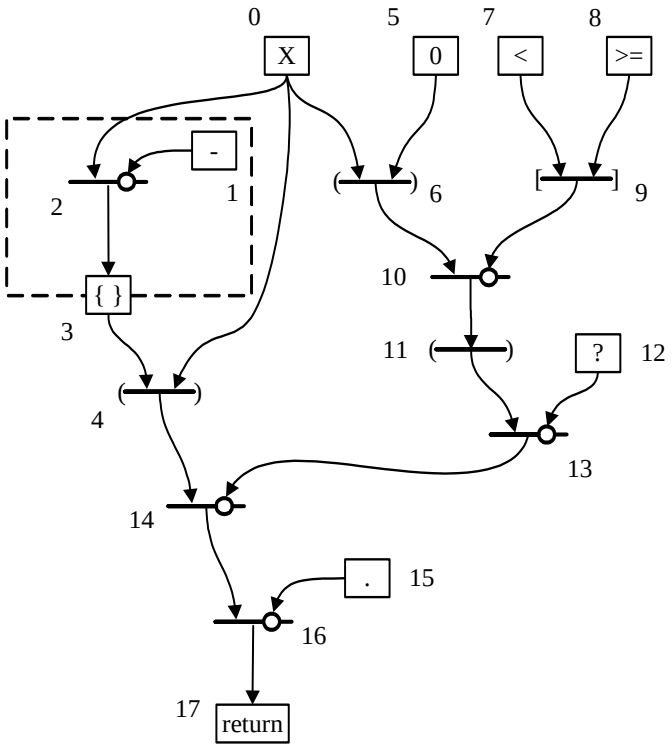

Рис. 3. Информационный граф функции Abs

На втором этапе, при формировании управляющего графа программы, осуществляется обход внутреннего представления информационного графа от точки возврата результата к исходным данным (к аргументу). Каждому управляющему узлу ставится в соответствие узел информационного графа. Управляющие связи строятся вдоль информационных (рис. 4). Если узел информационного графа является константным оператором, то на соответствующей связи управляющего графа ставится фишка, задающая сигнал о готовности данных, определяемых этой константой. Узлы, входящие в задержанный список не получают и не формируют сигналы готовности данных, до тех пор, пока список не будет раскрыт. На рисунке эти сигналы представлены жирными точками. Узлы информационного графа показаны номерами, стоящими в узлах управляющего графа.

4. Событийный процессор

Сформированный граф управления размещается в соответствующих внутренних структурах и используется событийным процессором для организации вычислений. Сигналы, определяющие начальную разметку управляющего графа, задают события, связанные с наличием данных, формируемых в ходе вычислений. Каждое событие содержит номер узла, которому передается управление.

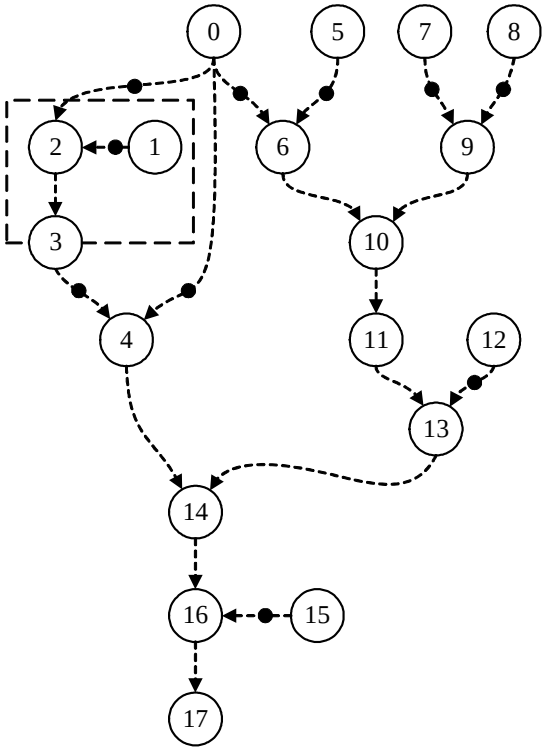

Рис. 4. Управляющий граф, формируемый для функции Abs

Получение события активизирует управляющий узел, переводя его в новое состояние, определяемое как функцией связанного с ним узла информационного графа, так и своим текущим состоянием. Обобщенная структура событийного процессора приведена на рис. 5.

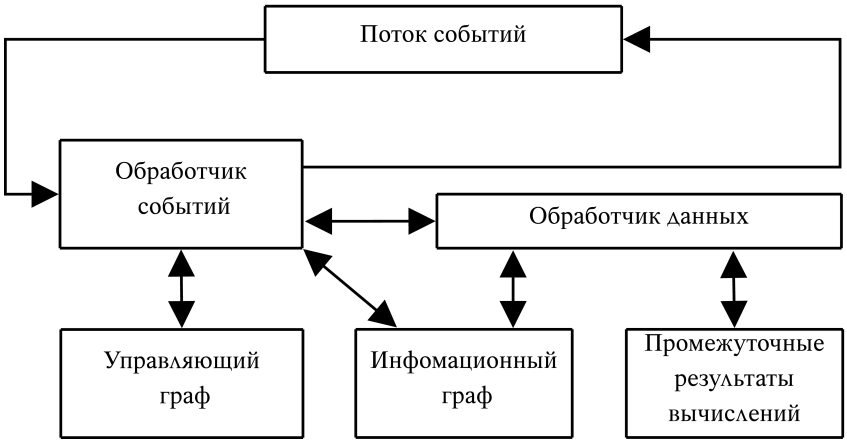

Рис. 5. Структура событийного процессора

Функционирование событийного процессора осуществляется следующим образом. Исходные события, определяемые начальной разметкой управляющего графа, выставляются в поток событий, из которого они передаются обработчику событий в соответствии с дисциплиной обслуживания. В простейшем случае это может быть очередь. Обработчик событий выбирает указанный узел и, на основе анализа его состояния, может обратиться к информационному графу за кодом выполняемой операции. В случае, когда операция обработки данных должна быть выполнена, происходит обращение к обработчику данных, который осуществляет требуемые функциональные преобразования и сохраняет промежуточные результаты. После обработки данных управляющий узел переходит в новое состояние и при необходимости формирует новое событие, поступающее в поток.

Состояния узла управляющего графа является совокупностью состояний входов узла. Кратность дуг, а соответственно и число входов, может меняться динамически в ходе выполне-

ния программы. Это происходит, например, при применении операции группировки в параллельный список. Таким образом, состояние каждой вершины управляющего графа определяется динамической структурой данных, хранящей информацию о готовности данных для каждого входа. В зависимости от состояния вершины управляющего графа и правил передачи управления для соответствующего типа вершины информационного графа происходит передача управления обработчику данных и/или последующее порождение события.

Заключение

Предложенные в работе методы позволяют создавать архитектурно-независимые параллельные приложения. Наличие только информационных зависимостей между функциями облегчает верификацию и отладку логики приложения, позволяя отбросить процесс анализа тупиковых и конфликтных ситуаций из-за отсутствия на данном этапе ресурсных ограничений.

Вместе с тем следует отметить, что, как и в случае распараллеливания последовательных программ, остается нерешенной проблема эффективного использования реальных вычислительных ресурсов. Однако, на наш взгляд, далее к ее ручному решению проще переходить не от последовательной, а от параллельной программы, обладающей максимальным параллелизмом, обуславливаемым отсутствием ресурсных ограничений. Использование событийного процессора обеспечивает при этом более гибкое представление механизмов управления и облегчает преобразование множества асинхронных информационных зависимостей функционально-потоковых программ к заданному числу одновременно выполняемых потоков или процессов императивных программ, выполняемых на реальных параллельных вычислительных системах.

Литература

1. А. И. Легалов, Ф. А. Казаков, Д. А. Кузьмин, Д. В. Привалихин. На пути к переносимым параллельным программам. // Открытые системы. – 2003. – № 5. – С. 36-42.
2. Легалов А.И. Функциональный язык для создания архитектурно-независимых параллельных программ. // Вычислительные технологии, № 1 (10) – 2005. С. 71-89.
3. Редькин А.В., Легалов А.И. Событийное управление выполнением функционально-потоковых параллельных программ. / Научный вестник НГТУ, № 3 (32). – 2008. – С. 111-120.
4. Легалов А.И. Об управлении вычислениями в параллельных системах и языках программирования // Научный вестник НГТУ, № 3 (18). – 2004. С. 63-72.
5. Легалов А.И., Редькин А.В. Расширение асинхронного управления по готовности данных. / Труды III Международной конференции «Параллельные вычисления и задачи управления» PACO’2006. – ISBN 5-201-14990-1. М.: Институт проблем управления им. В.А. Трапезникова РАН, 2006. С 1272-1281.
6. Легалов А.И. Использование асинхронно поступающих данных в потоковой модели вычислений. / Третья сибирская школа-семинар по параллельным вычислениям. / Томск. Изд-во Томского ун-та, 2006. С 113-120.