УДК 681.3

Штейнберг Б.Я., Напрасникова М.В., Нис З.Я.

Тестирование преобразований Открытой распараллеливающей системы

В данной работе описывается применение системы автоматического тестирования эквивалентности программ к тестированию оптимизирующих преобразований в Открытой распараллеливающей системе.

Введение

Основой Открытой распараллеливающей системы (ОРС) [1] является библиотека машинно-независимых высокоуровневых преобразований программ. Всякое такое преобразование создает результирующую программу, которая должна быть эквивалентна исходной. Для проверки корректности выполнения преобразований (на этапе разработки библиотеки ОРС) предполагается использовать систему автоматического тестирования [2].

1. Открытая распараллеливающая система

Открытая распараллеливающая система (ОРС) [1] предназначена для автоматического распараллеливания программ с таких языков программирования как Фортран, Паскаль и Си на параллельные суперкомпьютеры, ориентированные на математические вычисления. Система предполагает открытость к возможным изменениям в языке исходных текстов или в суперкомпьютере за счет дописывания (замены) относительно небольших программ. В частности, постоянно могут дописываться дополнительные преобразования программ.

ОРС представляет собой набор модулей следующих типов:

1) Преобразователи исходных текстов программ (Фортран, Паскаль, Си) в специальное внутреннее представление (не зависят от архитектуры суперкомпьютера). Внутреннее представление программ представляет собой специальную структуру данных, в которую преобразуются операторы исходного текста. Основное требование к этой структуре – удобство дальнейших преобразований программ, включая отслеживание эквивалентности. Преобразователи исходных текстов могут генерироваться специальным генератором компиляторов, ориентированным на внутреннее представление системы.
2) Библиотека распараллеливающих (оптимизирующих) преобразований программ во внутреннем представлении. (Библиотека не зависит ни от исходного языка, ни от архитектуры суперкомпьютера). К библиотеке прилагаются модули для построения каких-либо графов информационных связей, необходимых для отслеживания эквивалентности преобразований.

3) Оптимизирующие преобразователи программ. Каждый такой преобразователь представляет собой последовательность вызовов программ из библиотеки преобразований. Такие преобразователи ориентированы на некоторый класс суперкомпьютеров (не зависит от исходного языка).
4) Модули распределения данных в параллельной памяти.
5) Генераторы кодов конкретных суперкомпьютеров (не зависят от исходного языка).

Библиотека преобразований ОРС содержит как сравнительно тривиальные преобразования, так и достаточно сложные, реализация которых требует тщательного тестирования. Такие преобразования могут состоять из последовательности более простых. Приведем примеры некоторых таких преобразований [3, 4].

Пример 1. Замена рекуррентного цикла

```
      DO 99 I = 2, 101
99    X(I) = A(I)*X(I-1)+B(I)
```

на эквивалентный позволяет повысить степень параллелизма

```
      DO 66 I=1, 50
      AA(I+50) = A(I+50)*A(I-1+50)
      BB(I+50) = A(I+50)*B(I-1+50)+B(I+50)
66    X(I) = A(I)*X(I-1)+B(I)
      DO 55 I=51, 100, 2
      X(I) = AA(I)*X(I-2)+BB(I)
55    X(I+1) = AA(I+1)*X(I-1)+BB(I+1)
```

Можно добиться и большего ускорения (при наличии достаточного количества ресурсов).

Пример 2. Выполнение подстановки для индексных переменных выполняется совсем неочевидным и громоздким способом. Цикл

```
      DO 99 J=1,N1
      DO 99 I=1,N2
      DO 99 K=1,N3
      X(I)=A(I,J)-K
99    B(I,J,K)=X(I-1)
```

эквивалентен следующему фрагменту программы

```
      DO 99 J=1,N1
      DO 99 K=1,N3
      X(1)=A(1,J)-K
      B(1,J,K)=X(0)
      DO 99 I=2,N2
      X(I)=A(I,J)-K
99    B(I,J)=A(I-1,J)-K
```

С переименованием переменных происходит то же, что и с перестановкой.

Пример 3. Двойной цикл

```
Do 444 I = 3,100
Do 444 J = 3,100
X(I-1,J+2) = ...
```

```
      ...= X(I,J)+X(I-2,J)
    X(I+1,J-1) =
    444   CONTINUE
```

эквивалентен фрагменту программы

```
      DO 111 J = 3,100
111   XX(1,J) = X(1,J)
      DO 112 J = 3,4
112   XX(2,J) = X(1,J)
      Do 444 I = 3,100
      Do 444 J = 3,100
      XX(I-1,J+2) = ...
      ...       = X(I,J)+XX(I-2,J)
      X(I+1,J-1) =
444   CONTINUE
      DO 555 J = 3,100
555   X(2,J+2) = XX(2,J+2)
      DO 666 J = 3,100
666   X(3,J+2) = XX(3,J+2)
      DO 777 I= 3,100
777   X(I-1,100) = XX(I-1,100)
      DO 888 I = 3,100
888   X(I-1,101) = XX(I-1,101)
      DO 999 I= 3,100
999   X(I-1,102) = XX(I-1,102)
```

Пример 4. Очень сложным выглядит преобразование цикла к виду, удобному для выполнения с минимизацией числа межпроцессорных пересылок:

```
      ……….
      DO 1 In = 0, p-1
1     Y(b10+b1j*Ij,..,bk0+bkj*Ij) = X(a10+ a1j*Ij,..,ak0+ akj*Ij);
      DO 1 I1 = 0, p-1
```

Пусть d - делитель числа p и числа q = p/d. Тогда предыдущий цикл эквивалентен следующему:

```
      DO 1 L1   = 0,p-1
      .....
      DO 1 Ln-1    = 0,p-1
      DO 2 In  =  0,p-1
      E1(In)=(t1*In+L1) mod p
      .....
2     En-1(In)=(tn-1*In+Ln-1) mod p
      DO 1 I' = 0,d-1
      DO 1 I" = 0,q-1
1     Y(b10+b1j*Ej(I"+I'*q)+b1n*(I"+I'*q), ..., bk0+ bkj* Ej(I"+I'*q) +
      bkn*(I"+I'*q) = X(a10+  a1j* Ej(I"+I'*q) + a1n*(I"+I'*q), ...);
```

Приведенные примеры показывают, что визуальная проверка правильности выполнения преобразования не вызывает достаточного доверия. Поэтому автоматизация тестирования таких преобразований вполне целесообразна.

2. Система автоматического тестирования программ.

Для осуществления тестирования была использована платформенно-независимая система PCAT (Parallel Compiler Auto Tester), разработанная одним из авторов статьи в ИСП РАН.

Перед тем, как дать функциональное описание системы PCAT, приведем основные этапы процесса тестирования и опишем, как они реализуются с использованием PCAT.

При тестировании любого программного обеспечения должны быть решены три основных вопроса:

1. построение тестовых наборов;
2. выбор критерия покрытия для измерения полноты тестирования;
3. построения тестового оракула.

Рассмотрим их в другом порядке, сначала выберем критерий покрытия. Так как главной функцией распараллеливающего компилятора является проведение распараллеливающих преобразований, то естественным критерием покрытия при тестировании функциональной корректности такого компилятора будет: *покрытие всех видов производимых распараллеливающих преобразований.*

Основываясь на сформулированном критерии покрытия проще решить задачу построения тестовых наборов. *Будем строить тесты так, чтобы критерий покрытия был конструктивно достижим.* Используя документацию, которая предоставляется вместе с компилятором, информацию, полученную с помощью прямого контакта с разработчиками, и другую литературу можно построить набор программ, отражающих всевозможные конструкции, которые подвергаются компилятором, распараллеливающим преобразованиям.

Остается решить проблему построения оракула. Напомним, что оракулом в тестировании называется алгоритм принятия решения о прохождении тестов через целевую систему [5]. При тестировании распараллеливающего компилятора задача оракула сводится к проверке функциональной эквивалентности исходной и преобразованной программ.

Сформулируем требования, которым должны отвечать тестовые программы:

1. Так как задачи, требующие решения на многопроцессорных ЭВМ, являются, как правило, вычислительными, то кажется нецелесообразным выходить за рамки этого класса программ, которые будут использоваться в качестве тестов.
2. Для проверки функциональной эквивалентности исходной и подвергнутой распараллеливающим преобразованиям программ необходимо сделать так, чтобы их поведение было *наблюдаемым*, т.е. в коде исходной программы должны быть операторы вывода значений основных переменных.

При использовании тестовых программ, отвечающих требованиям (1) и (2), задача проверки функциональной эквивалентности исходной и распараллеленной программ сводится к сравнению их трасс. *Трассой* будем называть поток выходных сообщений, получаемых во время работы программы.

При ближайшем рассмотрении оказывается, что, вообще говоря, недостаточно один раз сравнить трассы для проверки эквивалентности исходной и распараллеленной программ. Так как в зависимости от входных данных могут

быть получены различные трассы исходной программы, одни из которых при сравнении с соответствующими трассами преобразованной программы будут им эквиваленты, а другие нет. Таким образом, для того, чтобы проверить эквивалентность исходной и преобразованной программ, вообще говоря, необходимо получить набор трасс, соответствующих входным данным, 100% покрывающих код исходной и преобразованной программ.

Следуя, описанной схеме процесса тестирования система PCAT используется для проверки эквивалентности исходной и преобразованной программ. На вход PCAT подаются тексты исходной и преобразованной программ и описание входных данных, включающее тип и границы области определения. Система PCAT способна автоматически генерировать входные данные указанных типов псевдослучайно и для тестирования границ областей определения.

Исходная и преобразованная программы инструментируются PCAT для автоматического слежения за покрытием, компилируются любым внешним компилятором и запускаются с автоматически сгенерированными входными данными. Затем их трасы перехватываются и сравниваются на эквивалентность. Этот процесс будет продолжаться до тех пор пока не будет достигнут указанный пользователем процент покрытия или не будет сгенерированно столько входных данных, сколько указанно пользователем.

С PCAT можно работать через GUI интерфейс и из командной строки (что позволяет автоматизировать запуск проверки эквивалентности сразу нескольких пар программ – как раз наш случай).

Для инструментирования кода исходной программы PCAT требуется сделать синтаксический разбор ее текста. Для сохранения языковой независимости система PCAT может быть интегрирована с любым внешним синтаксическим анализатором, так как во время инструментирования код рассматривается в универсальном формате XML, удовлетворяющем описанной ниже спецификации.

3. Использование системы PCAT для тестирования OPC

В качестве интерфейса между PCAT и OPC разработан конвертор из внутреннего представления OPC в требуемый формат XML, и конвертер

из XML в код на Си.

Рассмотрим упомянутое внутреннее представление в XML на примерах.

Корень документа – тэг Program.

Внутри корневого тэга могут находиться объявления идентификаторов – переменных, массивов, функций. После описаний функций следуют их тела.

Объявления.

Объявление переменных.

Тип переменной может принимать значения: *bool, int, double, char, void*. С каждым типом могут задаваться дополнительные атрибуты: *pointer="true"* и *pointer2pointer ="true"*, означающие, что тип является указателем на базовый тип или двойным указателем на базовый соответственно.

Id – порядковый номер в описаниях переменных, в большинстве случаев можно игнорировать.

Пример. <declvar id="**0" type="int" name="a" />** задает **int a.**

Объявление массивов.

Тип массивов ограничен значениями только примитивных типов (см. выше). Dimension определяет информацию о размерности с индексом id (нумерация слева направо).

```
<declarr id="0" type="int" name="b">
  <dismension id="0" size="100" />
  <dimension id="1" size="12" />
</declarr>
```

Это объявление задает массив int b[100][12].

Объявление функций.

Объявление функции – это её прототип. Тэг для описания этого прототипа *declfunc:*

```
<declfunc type="char" pointer="true" pointer2pointer="true" name="f">
        <param id="0" type="int" />
        <param id="1" type="char" pointer="true" />
        <param id="2" type="bool" alias="true" />
</declfunc>
```

Описания

Тела функций задаются в между тегами <function>, с единственным атрибутом – именем функции: <function name="f">… </function>

Тэг returntype задает возвращаемый тип, например:

```
<returntype name="int" />
```

Тэг <params> задает параметры функции. Разумеется, он может быть пустым. Параметры задаются в таком виде:

```
<params>
        <param id="0" type="double" name="d" />
        <param id="1" type="int" name="i" />
</params>
```

Меньший id соответствует предшествованию в списке параметров функции.

Внутри тела функции допустимы объявления локальных переменных.

Синтаксис при этом полностью совпадает с приведенным выше.

Выражения

Использования констант задаются таким образом:

```
<const type="double" value="1.23" />
```

Использования неиндексированных переменных:

```
<var name="b" />
```

Использование индексированных переменных:

```
<array name="b">
        <indices>
                <index id="0">
                        <const type="int" value="1" />
                </index>
        </indices>
</array>
```

Последний пример иллюстрирует обращение к b[1]

Операции задаются следующим образом:

```
<expression>
<operand id="0">
    <array name="b">
```

```
                <indices>
                  <index id="0">
                        <const type="int" value="1" />
                  </index>
                </indices>
              </array>
              </operand>
            <operator name="plus" />
                <operand id="1">
                <const type="double" value="1.23" />
              </operand>
            </expression>
```

Этот пример реализует b[1] + 1.23

Аналогично *plus* определены коды операций: *minus, mult, div*, а также унарные *deref* и *address* – разыменование указателя и получение адреса соответственно.

Вызов функции:

```
<subroutine name="testFunc" />
<params>
        <param id="0">
                <const type="double" value="1.12" />
        </param>
</params>
```

Операторы

Используются следующие операторы: goto, if, присваивания, for, while, do..while, return, switch/case. Опишем некоторые из них.

Все операторы задаются внутри тегов *statement*. Например:

```
<statement id="2" name="return" label="L1">
```

Атрибут id задает порядковый номер оператора в блоке (операторных скобках)

- *goto*

```
<statement id="1" name="goto">
  <label name="L1" />
</statement>
```

- *if*

```
<statement id="1" name="if">
<cond>
<expression type="binary">
  <operator name="gt" />
<operand id="0">
  <var name="d" />
    </operand>
<operand id="1">
  <const type="int" value="0" />
    </operand>
    </expression>
    </cond>
<then>
```

```
<statement id="0" name="assign">
<assign id="0">
  <var name="d" />
    </assign>
<assign id="1">
  <const type="double" value="12.1" />
    </assign>
    </statement>
    </then>
<else>
<statement id="0" name="assign">
<assign id="0">
  <var name="d" />
    </assign>
<assign id="1">
  <const type="double" value="32.1" />
    </assign>
    </statement>
    </else>
    </statement>
```

Пример для
if (d > 0) d = 12.1; else d = 32.1;

- *return*

```
<statement id="1" name="return">
  <const type="int" value="123" />
</statement>
```

- *присваивания*

```
<statement id="0" name="assign">
<assign id="0">
  <var name="aaa" />
    </assign>
<assign id="2">
  <var name="bbb" />
    </assign>
<assign id="1">
  <var name="ccc" />
    </assign>
    </statement>
```

Этот код реализует aa=bb=cc

- *цикла while*

```
<statement id="0" name="while">
  <cond />
  <body>
  <statement id="0" name="assign">
     <assign id="0">
          <var name="d" />
    </assign>
  <assign Id="1">
     <const type="double" value="12.3" />
```

```
</assign>
</statement>
</body>
</statement>
```

Реализация while(1) d=12.3;

Общая схема работы алгоритма тестирования распараллеливающих преобразований имеет вид:

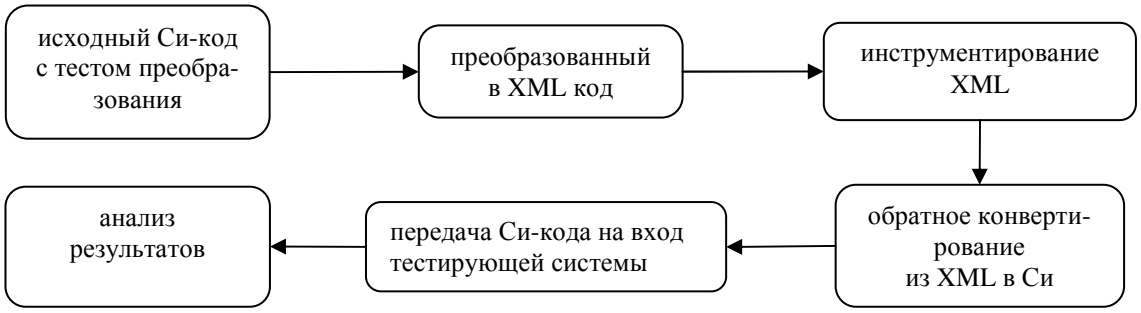

Схема 1. Алгоритм тестирования распараллеливающих преобразований ОРС

Литература

1. Штейнберг Б.Я. Открытая распараллеливающая система // РАСО'2001: Труды международной конференции «Параллельные вычисления и задачи управления». Москва, 2–4 октября 2001 г., ИПУ РАН. С. 214-220.
2. Напрасникова М.В. Система автоматического тестирования программ для дистанционного обучения программированию, V-ая молодежная научно-техническая конференция учащихся, студентов, аспирантов и молодых ученых "Наукоемкие технологии и интеллектуальные системы - 2003", МГТУ, апрель 2003 г.
3. Векторизация программ. // Векторизация программ: теория, методы, реализация. / Сборник переводов статей М.: Мир, 1991.
4. Штейнберг Б.Я. Математические методы распараллеливания рекуррентных циклов для суперкомпьютеров с параллельной памятью. – Ростов н/Д: Изд-во Рост. Ун-та, 2004. – 192 с.
5. Glossary of terms used in software testing Version 6.2. http://www.testingstandards.co.uk/Gloss6_2.htm