Главный редактор - д.э.н., профессор К.А. Кирсанов
тел. для справок: +7 (925) 853-04-57 (с 11^{00} – до 18^{00})
Опубликовать статью в журнале - http://publ.naukovedenie.ru

Цимина Наталья Александровна
Tsimina Natalia Aleksandrovna
Государственный строительный университет/
Rostov State University of Civil Engineering
Ассистент кафедры Прикладной математики и вычистительной техники РГСУ
Assistant, Department of Applied Mathematics and Computer Science RGSU
E-Mail: nburyk@mail.ru

Чернов Андрей Владимирович
Chernov Andrei Vladimirovich
Ростовский Государственный строительный университет
Rostov State University of Civil Engineering
Заведующий кафедры Прикладной математики и вычистительной техники РГСУ
Head of the Department of Applied Mathematics and Computer Science RGSU
E-Mail: avche@yandex.ru

Кононенко Анна Анатольевна
Kononenko Anna Anatolievna
Ростовский Государственный строительный университет/
Rostov State University of Civil Engineering
Студент магистратуры РГСУ
Graduate student RGSU
E-Mail: nyta.ru88@mail.ru

051318.78 «Математическое моделирование,
численные методы и комплексы программ»

Применение алгоритмов верификации к программам, использующим интерфейс MPI

Application verification algorithm to the program, the interface uses MPI

Аннотация: В этой статье представлен способ для проверки корректности параллельных программ, которые выполняют сложные численные расчеты. В областях, которые требуют обширных вычислений, программа может быть разделена между несколькими процессами, работающими параллельно, чтобы уменьшить общее время выполнения и увеличить общий объем памяти, доступной для программы. Представлены методы, с помощью которых можно установить правильность программы такого типа - то есть, доказать, что программа всегда выдает правильный результат для любых входных данных, и если программа не является правильной выдает соответствующие контрпримеры. Представлены примеры программ, реализующие алгоритмы с матрицами, моделирующие физические явления, или моделирующие эволюцию системы дифференциальных уравнений.

The Abstract: This article shows how to verify the correctness of parallel programs that perform complex numerical calculations. In areas that require extensive calculations, the program can be shared with other processes running in parallel to reduce the total execution time and increase the

total amount of memory available for the program. Presents the methods by which to establish the correctness of a program of this type - that is, to prove that the program always gives the correct result for any input, and if the program is not correct issues relevant counterexamples. Are examples of programs that implement algorithms with matrices that simulate physical phenomena, or simulate the evolution of a system of differential equations.

Ключевые слова: Верификация, частично формализованные методы проверки, формальные модели проверки программное обеспечение, обзор, статический анализ, динамические методы, формальные методы, методы синтеза, проверка на модели, параллельные программы.

Keywords: The verification, partially formalized methods of verification, formal models of verification of the software, review, the static analysis, dynamic methods, formal methods, synthetic methods, model checking, parallel programs.

***

Целью статьи является разработка способа для проверки корректности параллельных программ, которые выполняют сложные численные расчеты, включая расчеты с числами с плавающей точкой.

В областях, которые требуют обширных вычислений, программа может быть разделена между несколькими процессами, работающими параллельно, чтобы уменьшить общее время выполнения и увеличить общий объем памяти, доступной для программы. Процесс распараллеливания в последовательную программу как известно труден, и подвержен большому количеству ошибок. Попытки автоматизировать этот процесс не имели большого успеха, поэтому до сих пор большая часть таких параллельных программ пишутся вручную. Разработчики таких программ тратят огромные усилия на тестирование и отладку, чтобы убедиться в правильности своего кода, поэтому любые методы установления достоверности этих программ или методы нахождения ошибок были бы очень полезны.

Под числовым программированием подразумеваются программы, основная функция которых заключается в проведении численных расчетов. Считается, что на ввод в данной программе подается вектор (с плавающей точкой) и на выходе получается другой аналогичный вектор. Примерами могут служить программы, реализующие алгоритмы с матрицами, моделирующие физические явления, или моделирующие эволюцию системы дифференциальных уравнений. Нас интересуют методы, с помощью которых можно установить правильность программы такого типа - то есть, доказать, что программа всегда выдает правильный результат для любых входных данных, и если программа не является правильной выдает соответствующие контрпримеры. Обычно способ тестирования для достижения этого результата имеет два существенных недостатка. В первую очередь, как правило, невозможно проверить большую часть используемых ресурсов, необходимых для параллельного программирования. Таким образом, тестирование может выявить ошибки, но, как известно, оно не может показать, что программа корректно ведет себя на входных данных, которые не проверяются. Во-вторых, поведение параллельных программ, в том числе сами параллельные численные программы, как правило, зависят от порядка, в котором события происходят в различных процессах. Этот порядок, в свою очередь зависит от нагрузки на процессор, задержки сети связи и других подобных факторов. Параллельные числовые программы таким образом, могут вести себя поразному на разных исполнениях с одним и тем же вектором входных данных, так что получить правильный результат на выполнении теста даже не гарантирует, что программа будет правильно вести себя на другом исполнении с такими же входными данными.

Главный редактор - д.э.н., профессор К.А. Кирсанов
тел. для справок: +7 (925) 853-04-57 (с 11^{00} – до 18^{00})
Опубликовать статью в журнале - http://publ.naukovedenie.ru

Предлагаемый метод, который сочетает в себе метод *model checking* [1] с символическим исполнением, решает эти два ограничения: он может быть использован, чтобы показать, что параллельное численное программирование дает правильный результат на любом входном векторе, независимо от конкретного способа чередования событий из параллельных процессов. При применении метода *model checking* в этой ситуации, возникают два вопроса. Во-первых, этот метод требует от программиста ограничиться конечным числом состояний модели проверяемой программы. Но численные программы обычно представляются огромными массивами данных с плавающей точкой, а сама природа наших задач диктует, что мы не можем просто использовать далекие абстрактные данные. В связи с этим не ясно, каким образом создать соответствующие модели программ с конечным числом состояний без существенного обострения проблемы взрыва состояний. Второй вопрос касается характерной особенности, мы хотим проверить утверждение, что результат, получаемый в рамках правильной программы должен быть сформулирован в некотором виде, потенциально пригодном для инструментов *model checking*. Мы имеем дело с первым вопросом по символическому моделированию вычислений в программе. В нашей модели входом будем считать вектор x_i символических констант, и получим на выходе некоторый вектор символических выражений x_j. В программе числовые операции заменены на соответствующие символьные операции в модели. Кроме того, каждое символическое выражение представляет собой одно целое, которое препятствует увеличению размера вектора состояния и позволяет легко выразить модель на языке стандартных инструментов *model checking*, таких, как *SPIN*[2]. Второй вопрос требует, чтобы пользователь представил последовательную версию программы, которая будет проверена. *Model checking* будет использован, чтобы показать, что параллельные и последовательные программы функционально эквивалентны, то есть, что они получат один и тот же результат на любых входных данных. Конечно, это означает, что наш метод только сводит задачу проверки параллельной программы к проблеме последовательной проверки. Тем не менее, большинство проблем в этой области имеют гораздо более простое решение, чем последовательное решение параллельной программы, а это уже обычная практика для разработчиков программного обеспечения работать с последовательной версией программы или построить образец, для тестирования и других целей. Кроме того, наш метод дает информацию, которая поможет проверить правильность последовательной программы.

Другим вопросом, который возникает при таком подходе, является то, что численные программы содержат много условий, которые связаны с входными данными. Такие программы можно рассматривать как множество случаев, каждый случай, состоит из вектора на входе и соответствующего символического вектора на выходе. Мы используем модель проверки, изучающую все возможные пути последовательной программы, и для каждого такого пути мы записываем пару состояний, булевозначных символических выражений на входе, которые прошли этот путь и были выполнены. Модель параллельной программы спроектирована так, чтобы принимать в качестве входных данных не только символический входной вектор, но и также пару условий состояний. Модель проверки используется для изучения всех возможных путей параллельной программы, которые соответствуют им. Для каждого компьютера, результат запуска параллельной программы всегда совпадает с результатом получающимся при запуске последовательной программы.

Мы считаем, что параллельная численная программа $Q_{\text{пар}}$ состоит из фиксированного числа параллельных процессов. Пусть n число параллельных процессов. Полагается так же, что эти процессы не имеют общей памяти и общаться только посредством передачи сообщений, с помощью таких функций, которые предусмотрены интерфейсом передачи сообщений (*MPI*) (*Message Passing Interface*)[5]. Будем считать, дана последовательная программа $Q_{посл}$,

Главный редактор - д.э.н., профессор К.А. Кирсанов
тел. для справок: +7 (925) 853-04-57 (с 11^{00} – до 18^{00})
Опубликовать статью в журнале - http://publ.naukovedenie.ru

которая служит спецификацией для $Q_{\text{пар}}$. Мы также считаем, что обе программы $Q_{\text{пар}}$ и $Q_{\text{посл}}$ на каждом входе имеют свойство, которое может быть проверено с помощью более традиционных методов проверки моделей. В некоторых случаях можно также ввести ограничение на число итераций некоторых циклов в программе, чтобы убедиться, что модель, которую мы строим не будет чрезмерно большой (или даже бесконечной) по числу состояний.

Отметим, что программа $Q_{\text{пар}}$ функционально эквивалентна программе $Q_{\text{посл}}$,то есть на выходе каждой функции должна получиться детерминированная функция ее соответствующего входа. Таким образом, если одна из этих программ может получить различные выходные результаты на двух исполнениях с одними и теми же входными данными, то мы считаем, что были нарушены свойства функциональной эквивалентности, вне зависимости от поведения на других наборах входных данных. Наш метод на самом деле выявляет такие нарушения, а также наиболее типичные нарушения, при которых обе программы ведут себя, как функции, но не согласуются на некотором наборе вводимых данных. Начнем с объяснения метода при условии, что ни одна программа не содержит ветви выражений с переменными, которые моделируются символически.

Рассмотрим пример на рис.1, произведения двух матриц $A_{N\times L}$ и $B_{L\times M}$, результирующая матрица $C_{N\times M}$, последовательного кода программы на языке программирования *C*.

```
double A[N][L], B[L][M], C[N][M];
                    :
                    :
int i,j,k;
for (i=0; i<N; i++)
  for (j=0; j<M; j++) {
    C[i][j] = 0.0;
    for (k=0; k<L; k++)
      C[i][j] += A[i][k]*B[k][j];
  }
```

***Рис. 1**. Код последовательной программы произведения матриц*

Существует много способов распараллелить данную программу, например, как показано на рис.2 с использованием функций *MPI* для межпроцессорной связи. Каждый процесс должен рассматриваться в качестве самостоятельного исполнимого процесса, со своей локальной памятью. Каждый процесс может также получить свой ранг (уникальное целое число между 0 и *n* - 1) из *MPI*-инфраструктуры. Для этого кода, который использует подход родитель-наследник для достижения автоматического баланса, мы предполагаем, что $N \geq n - 1 \geq 1$, и что все три матрицы хранятся в локальной памяти процесса ранга 0 (родитель).

Для вычисления результата, родитель будет распределять работу между процессами положительных рангов (наследников). Мы предполагаем, что для каждого подчиненного процесса уже есть копия *B* в его локальной памяти. Родитель начинает отправив первую команду первому наследнику, вторую команду - второму наследнику, и так далее, до тех пор, пока первые *n*-1 строк будут сданы.

Главный редактор - д.э.н., профессор К.А. Кирсанов
тел. для справок: +7 (925) 853-04-57 (с 11^{00} – до 18^{00})
Опубликовать статью в журнале - http://publ.naukovedenie.ru

```
int rank,nprocs,i,j,numsent,sender,row,anstype;
double buffer[L], ans[M];
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) { /* I am the master */
  numsent=0;
  for (i=0; i<nprocs-1; i++) {
    for (j=0; j<L; j++)
      buffer[j] = A[i][j];
    MPI_Send(buffer, L, MPI_DOUBLE, i+1, i+1, MPI_COMM_WORLD);
    numsent++;
  }
  for (i=0; i<N; i++) {
    MPI_Recv(ans, M, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
       &status);
    sender = status.MPI_SOURCE;
    anstype = status.MPI_TAG-1;
    for (j=0; j<M; j++)
      C[anstype][j] = ans[j];
    if (numsent<N) {
      for (j=0; j<L; j++)
        buffer[j] = A[numsent][j];
      MPI_Send(buffer, L, MPI_DOUBLE, sender, numsent+1, MPI_COMM_WORLD);
      numsent++;
    }
    else MPI_Send(buffer, 1, MPI_DOUBLE, sender, 0, MPI_COMM_WORLD);
  }
} else { /* I am a slave */
  while (1) {
    MPI_Recv(buffer, L, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    if (status.MPI_TAG==0) break;
    row = status.MPI_TAG-1;
    for (i=0; i<M; i++) {
      ans[i] = 0.0;
      for (j=0; j<L; j++)
        ans[i] += buffer[j]*B[j][i];
    }
    MPI_Send(ans, M, MPI_DOUBLE, 0, row+1, MPI_COMM_WORLD);
  }
}
```

Рис. 2. *Код параллельной программы произведения матриц*

Наследник, после получения вектора-строки длиной *L*, от родителя, умножает его на *B*, и посылает результирующий вектор-строку длиной *M* к родителю. Родитель ждет на входе сообщение от любого процесса. После одного или более пришедших сообщений, родитель выбирает одно для приема, копию вектора, полученного в соответствующих строках матрицы *C*, и посылает следующую строку к наследнику, который только что вернул результат и возвращается к модели получателя. Так продолжается, пока все строки не будут вычислены. После этого, каждый раз, когда наследник посылает результат, он посылает обратно сообщения о прекращении работы. После того, как все результаты пришли к родителю, и последнее сообщение о прекращении работы было отослано, матрица *C* должна содержать произведение матриц *A* и *B*, и все процессы должны завершиться нормально.

На первом шаге создается пространство состояний модели $Q_{\text{сек}}$ из $Q_{\text{посл}}$ и записывается на языке *SPIN*-кода[3]. Модель будет использовать символические выражения. Символические выражения могут рассматриваться как древовидная структура, в которой листы узлы литеры или символические константы. Символические константы обозначаются *x0, x1,...*, и соответствуют компонентам входного вектора. Каждый из неконечных узлов в дереве, связан (унарными или бинарными) операторами, например, +, -, □, /, или любыми другими арифметическими операторами, со следующим узлом, которые имеют место в программе. Каждый численный расчет в программе с участием символически моделируемой переменной заменя-

Главный редактор - д.э.н., профессор К.А. Кирсанов
тел. для справок: +7 (925) 853-04-57 (с 11^{00} – до 18^{00})
Опубликовать статью в журнале - http://publ.naukovedenie.ru

ется на операции с символьными выражениями в модели. Символьные операции просто формирует новое дерево с корнем в этом операторе с поддеревом, представляющих один или два операнда.

MPI-SPIN является расширением *SPIN* для моделирования *MPI* программы таким образом, что устраняются ограничения, описанные выше. Синтаксис этих функций почти точно такой же, как и для языка *C*[4]. В частности, эти функции принимают произвольные указатели *C*; они поддерживают различные типы данных; они позволяют сообщения любого типа, которые могут быть отправлены в любое время, и они поддерживают наиболее распространенные сокращения операций. Никаких особых усилий не требуется со стороны пользователя для модели *MPI*-инфраструктуры. Все это значительно сокращает усилия, необходимые для построения моделей *MPI* программ.

Основная идея реализации *MPI-SPIN* - использовать связи в структуре записей для представления каждого коммуникационного запроса, или буфера сообщений в системе состояний. Поле для этой *C* структуры включает в себя указатель на отправителя или получателя буфера по умолчанию, ряды источника и назначения процессов, и так далее. Значения связных записей, созданных в процессе поиска в пространстве состояний хешируются и им присваивается уникальный идентификационный номер, так же, как и в случае с символьными выражениями, и именно эти идентификационные номера, присвоенные переменным, и раскрывают модель. Вместо того, чтобы использовать несколько каналов, единый глобальный массив сохраняет и изменяет для различных *MPI*-функций все записи общения. Дополнительные *MPI* процессы используются для моделирования всех возможных поведений *MPI* инфраструктуры и отвечает за такие действия, как отправка и получение запросов, буферизацию сообщений, и завершения запросов.

MPI-SPIN также вводит тип *MPI_Symbolicfor*, представляющий символические выражения, вместе с числом операций над этим типом. Поэтому *MPI-SPIN* содержит все составляющие, необходимые для применения символического метода сравнения для более сложных *MPI* программ.

Как мы уже отмечали, *MPI* блокировка сообщений является особым случаем неблокирующий связи и, фактически, блокируемые функции такие как *MPI_Sendor MPI_Recv* реализуются в *MPI-SPIN*, просто вызвав соответствующую неблокируемую функцию, и после этого сразу же *MPI_Wait*. Представляется, однако, что для программ, которые используют исключительно блокируемые связи, *MPI-SPIN* подход не столь эффективен, как канал-ориентированного подхода. Например, верификация преобразований Гаусса для $n = 5$, требует 6.7 млн. состояний и 1015 Мб для *MPI-SPIN*, в то время как наш специально разработанная модель, требует 790,6 состояний и 35 Мб.

Сообщения, отправленные по каналам это целочисленные Идентификаторы значений записей. Таким образом, оптимизация поддерживает в полном объеме *MPI-SPIN* подход. Для использования этой оптимизации, пользователю нужно только указать место, где в модели используются исключительно блокируемые связи; никаких изменений в самой модели не требуется. Для матричного умножения, например, *MPI-SPIN* модель потребляет примерно на 20% меньше состояний/памяти, чем на самой большой конфигурации.

Главный редактор - д.э.н., профессор К.А. Кирсанов
тел. для справок: +7 (925) 853-04-57 (с 11^{00} – до 18^{00})
Опубликовать статью в журнале - http://publ.naukovedenie.ru

ЛИТЕРАТУРА

1. Кларк Э.М., Гамбург О., Пелед Д. Верификация моделей программ: Model Checking. // Пер. с англ./ Под ред. Р. Смелянского. - М.: МЦНМО, 2002. – 416 с.

2. G. J. Holzmann. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 2003.

3. http://spinroot.com/.

4. http://www.microsoft.com/whdc/devtools/tools/SDV.mspx.

5. I. Lee, S. Kannan, M. Kim, O. Sokolsky, M. Viswanathan. Runtime Assurance Based On Formal Specifications. Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA’1999, pp. 279-287, 1999.