

СРЕДСТВА ОТЛАДКИ OPENMP ПРОГРАММ В DVM-СИСТЕМЕ

В.А. Алексахин, В.О. Барина, В.А. Бахтин, В.Д. Емельянов, В.А. Крюков, Ю.Л. Сазанов

1. ВВЕДЕНИЕ

Появление и быстрое распространение многоядерных процессоров привело к всплеску популярности стандарта OpenMP. На первый взгляд, разработка параллельных программ с его использованием выглядит намного проще, чем с использованием стандарта MPI. Однако программам, взаимодействующим через общие переменные, свойственны ошибки, вызывающие недетерминированное поведение программы. Такие ошибки очень тяжело находить с помощью традиционных методов отладки. Тут требуются специальные подходы, автоматизирующие обнаружение ошибок. Кроме того, такие автоматизированные методы обнаружения ошибок крайне необходимы для успешной разработки средств автоматического и автоматизированного распараллеливания последовательных программ, поскольку при отсутствии автора параллельной программы проводить ее отладку традиционными методами очень затруднительно.

Имеющийся у нас опыт распараллеливания на языке Фортран-OpenMP вычислительных программ показывает, что причиной большинства ошибок являлось неверное задание класса переменной (общая или приватная). Проявлением этой ошибки является либо несинхронизированный доступ к общей переменной (data race), либо чтение неинициализированной приватной переменной. Поэтому было решено сосредоточить внимание на обнаружении этих двух ситуаций.

Несинхронизированный доступ к общей переменной происходит в случаях, когда:

1. две или более нитей осуществляют одновременный доступ к одной и той же переменной,
2. по крайней мере, одна из них осуществляет запись в эту переменную,
3. нити не используют какие-либо средства взаимного исключения или координации.

В таких случаях порядок доступа недетерминирован. Следовательно, повторные запуски программы могут давать различные результаты в зависимости от порядка доступа нитей к разделяемым областям памяти.

Чтение неинициализированной приватной переменной возникает тогда, когда читается переменная, которой еще не было присвоено значение. Инициализация приватных переменных в OpenMP-программах может происходить не только явно посредством операторов инициализации, присваивания или ввода, но и неявно – при входе или выходе из параллельного региона или конструкции распределения работ.

Существующие на сегодняшний день инструменты для обнаружения таких ошибок в OpenMP-программах (например, Intel Thread Checker [3] и Sun Studio Thread Analyzer [4]) не всегда доступны пользователю. Они накладывают определенные ограничения на используемые в программе конструкции (например, не должны встречаться вызовы функции опроса количества нитей), не могут быть использованы при отладке программ с реальными производственными данными (из-за требуемых дополнительных ресурсов памяти и времени). Необходимо иметь средства отладки, свободные от этих недостатков.

Подобные средства для функциональной отладки DVM-программ [5, 6] и MPI программ [7] есть в DVM-системе – **сравнительная отладка** и **автоматический контроль корректности**. Сравнительная отладка заключается в сравнении двух запусков программы – эталонного (последовательного) и отлаживаемого (параллельного). Обнаруженные при этом различия используются для локализации ошибок. Автоматический контроль корректности эффективен для поиска неверных спецификаций параллелизма, реальных и потенциальных дедлоков, некорректного доступа к данным, и прочих ошибок. Ниже описываются принципы построения аналогичных средств автоматизированной отладки OpenMP-программ.

2. ИНТЕРФЕЙС ИНСТРУМЕНТАЦИИ ПРОГРАММ НА ЯЗЫКЕ ФОРТРАН-OPENMP

Для применения автоматизированных методов отладки требуется инструментация программы пользователя [8, 9], т.е. дополнение программы вызовами подпрограмм отладчика. Интерфейс инструментации унифицирован, что позволяет линковать одну и ту же инструментированную программу с разными библиотеками. Подпрограммы одной библиотеки могут динамически контролировать выполнение программы, другой – накапливать трассу выполнения, третьей – сравнивать ход выполнения программы с трассой, накопленной при другом выполнении.

Большинство OpenMP-директив можно представить в следующем виде:

```
C$OMP <директива> [<клауза> [[,]<клауза>]...]
```

```
<структурный блок>
```

```
C$OMP END <директива> [<клауза> [[,]<клауза>]...]
```

где <директива> – это одна из следующих: PARALLEL, DO, SECTIONS, SINGLE, CRITICAL, MASTER, ORDERED, а <клауза> – содержит некоторую дополнительную информацию о директиве.

Для каждой OpenMP-директивы был разработан собственный набор функций для инструментации. Схематично инструментация программы выполняется следующим образом:

```
call DBG_Before<директива> (<параметры функции>)
C$OMP <директива> [<клауза> [[,<клауза>]...]
call DBG_<директива>Event (<параметры функции>)
<структурный блок>
C$OMP END <директива> [<клауза> [[,<клауза>]...]
call DBG_After<директива> (<параметры функции>)
```

В программу добавляются вызовы специальных функций до открывающей директивы, перед первым оператором структурного блока, а также после закрывающей директивы. Статическая информация о директиве (тип директивы, имя файла, строка начала/окончания директивы, информация о клаузах) передается при помощи *контекстной строки* [8], которая представляет собой последовательность пар «атрибут=значение» разделенных звездочками «*».

```
<length>*type=<type>*name1=<name>*file=<file>*line1=<number>*...**
```

Помимо директив, регистрируются все используемые в программе переменные (их тип, размерность для массивов, наличие спецификации SAVE или оператора DATA) и COMMON-блоки. Инструментируются все чтения/записи в переменные, входы/выходы из процедур. Реализована возможность управлять степенью подробности инструментации с помощью опций, задаваемых при инструментации, или директив, указанных в программе.

3. ДИНАМИЧЕСКИЙ КОНТРОЛЬ КОРРЕКТНОСТИ OPENMP-ПРОГРАММ

Автоматический контроль корректности может осуществляться либо динамически во время выполнения программы (так он реализован для DVM-программ), либо после завершения программы посредством анализа собранных трасс (так он производится для MPI-программ). Контроль корректности OpenMP-программ решено выполнять динамически. Ниже описываются принципы обнаружения несинхронизированного доступа к общей переменной и чтения инициализированной приватной переменной.

3.1. НЕСИНХРОНИЗИРОВАННЫЙ ДОСТУП

Несинхронизированный доступ возникает тогда, когда нити не обеспечили взаимное исключение при модификации одной и той же переменной (посредством конструкций `single`, `master`, `critical`, `atomic`, `ordered`, или с помощью механизма замков), либо не обеспечили координацию между модификацией переменной в одной нити и ее чтением в других нитях (посредством конструкций `barrier`, `flush`). Поскольку наиболее сложным представляется контроль взаимного исключения с помощью механизма замков, то основное внимание было уделено ему.

Для обнаружения несинхронизированных модификаций переменных используется алгоритм *Lockset* [10, 11]. Алгоритм *Lockset* анализирует обращения к общим переменным и проверяет, что каждая переменная защищается некоторым замком, пока нить осуществляет доступ к этой переменной. Т.к. при динамическом контроле нет возможности точно определить, какие замки предназначены для защиты каких переменных, то предлагается действовать следующим образом.

Для каждой общей переменной v определяется множество $C(v)$ замков, кандидатов для защиты v . В это множество попадают те замки, которые были захвачены каждой нитью в момент доступа к переменной v . Когда новая переменная v инициализируется, считается, что ее множество замков $C(v)$ включает в себя все возможные замки. В момент обращения к переменной v ее множество $C(v)$ обновляется пересечением $C(v)$ со множеством замков, захваченных текущей нитью. Этот процесс, называемый *уменьшением множества замков*, гарантирует, что любой замок, который, возможно, защищает v , содержится во множестве $C(v)$. Если некоторый замок l защищает v , он будет оставаться в $C(v)$. Если множество $C(v)$ становится пустым, это означает, что нет замка, который бы защищал v .

На практике есть три метода программирования, которые нарушают эту дисциплину:

1. *инициализация* – общие переменные инициализируются часто и без захвата замков;
2. *чтение общих данных*: некоторые общие переменные модифицируются только при инициализации и впоследствии только читаются. Доступ к ним может осуществляться без каких-либо замков;
3. *замки чтения-записи* разрешают чтение общих переменных многим нитям одновременно, а запись – только в монопольном режиме.

Инициализация и чтение разделяемых данных. Для нити, инициализирующей переменную, нет необходимости записывать ее от остальных нитей, если они не модифицируют эту переменную. Для предотвращения ложных предупреждений об ошибках при таких незащищенных записях считается, что разделяемая переменная будет проинициализирована до момента первого доступа другой нитью. Пока доступ к переменной осуществляется только единственной нитью, чтение и запись в нее не влияют на множество $C(v)$.

Поскольку одновременные чтения общей переменной многими нитями не являются несинхронизированными, нет необходимости защищать переменную. Для поддержки беззамкового доступа к общей переменной несинхронизированный доступ диагностируется только тогда, когда после инициализации она модифицируется более чем одной нитью.

Замки чтения-записи. Многие программы используют не только замки монопольного доступа, но и замки, которые разрешают чтение общих переменных многим нитям одновременно, а запись – только в монопольном режиме. При анализе этого вида замков используется следующий алгоритм *уменьшения множества замков*: требуется, чтобы для каждой переменной v был некоторый замок m , защищающий ее по записи при каждой модификации v и защищающий ее по записи или по чтению при каждом чтении v .

3.2. ЧТЕНИЕ НЕИНИЦИАЛИЗИРОВАННОЙ ПЕРЕМЕННОЙ

Помимо явных способов инициализации переменных (операторы присваивания, операторы ввода и блоки DATA) в OpenMP-программе существуют правила сохранения значений переменных при входе в параллельные области (и конструкции распределения работ) и выходе из них с помощью различных клауз, например:

- Приватные переменные не сохраняют своих значений, если не указана клауза `firstprivate`
- Переменные, породившие приватные копии, считаются неопределенными после выхода, если не указана клауза `lastprivate`

Для обнаружения чтения неинициализированных переменных с каждой переменной связывается специальный флаг инициализации, который устанавливается в точках модификации переменной, а проверяется при ее чтении. При этом используется граф порождения переменных, в котором отражаются связи между переменными каждой нити и соответствующими копиями приватных переменных, создаваемыми при порождении дочерних нитей.

4. СРАВНИТЕЛЬНАЯ ОТЛАДКА OPENMP ПРОГРАММ

Подсистема сравнительной отладки OpenMP-программ продолжает линию сравнительных отладчиков для DVM-программ и MPI-программ, реализованных в DVM-системе [5,6,7]. В настоящей статье представлены некоторые новые возможности, введенные для сравнительной отладки OpenMP-программ.

Сравнительная отладка заключается в сравнении соответствующих переменных в соответствующие моменты выполнения двух версий одной программы (или двух разных программ), одна из которых считается эталонной, другая – отлаживаемой. Для сравнения двух выполнений программы можно сначала накопить трассы, фиксирующие выполненные операторы и значения переменных, а затем сравнивать эти трассы, либо сначала собрать трассу при одном выполнении программы а затем динамически сравнивать с ней другое выполнение.

Метод сравнительной отладки впервые был предложен для последовательных программ в отладчике Guard [14, 15], а для параллельных программ – в DVM-системе. Главное отличие между ними заключается в том, что в отладчике Guard сам пользователь задает точки в программах, в которых надо сравнивать указанные данные, а в DVM-системе и точки и сравниваемые данные определяются автоматически компилятором. Для OpenMP-программ эталонным может считаться выполнение последовательной программы (с игнорированием при компиляции OpenMP-директив), а отлаживаемым – выполнение программы, скомпилированной в режиме OpenMP и выполняющейся на одной или нескольких нитях.

Однако высокие требования к ресурсам (оперативная и внешняя память, время) зачастую не позволяют производить отладку программ на реальных данных с накоплением полных трасс выполнения. Подготовка специальных модельных (отладочных) данных может быть затруднительна. К тому же на модельных данных ошибка может и не проявляться. Кроме того, использование сравнительной отладки затрудняется недетерминированным поведением программ.

Для сокращения ресурсов, требуемых для сравнительной отладки (объем трассы и время ее накопления), в отладчиках DVM-программ был реализован ряд специальных возможностей управления составом (а тем самым и объемом и временем сбора) трассы. Первая возможность основана на использовании только *границных итераций*, т.е. сборе и анализе данных только на итерациях (витках цикла) близких к границам расчетной области (а в параллельном цикле также и к границам локальных частей расчетной области на каждом процессоре). Можно задавать границы двух видов: «границы» – итерация считается граничной, если она находится на расстоянии не более заданного (ширина границы) от ребра прямоугольного параллелепипеда множества итераций многомерного гнезда циклов; «уголки» – итерация считается граничной, если она находится на расстоянии не более заданного от вершины этого параллелепипеда. Размеры границ можно менять с помощью параметров.

При этом исходили из того, что на граничных витках цикла выше вероятность проявления многих ошибок: ошибки общего вида (неинициализированные переменные, выход за границу массива и т.п.) – при крайних значениях переменных циклов, т.е. у границы области, а ошибки работы с распределенными данными – на границах локальных частей массивов.

Однако многие реальные программы характеризуются неоднородностью вычислений по расчетной области. Поэтому ограничение «границами», а тем более «уголками» может выводить из-под контроля значительные части кода. Для сравнительного отладчика OpenMP-программ решено пересмотреть вопрос отбора трассируемой информации. В описываемом сравнительном отладчике OpenMP-программ предложены следующие средства ограничения объема трассы и сокращения времени ее накопления.

Во-первых, унификация интерфейса трассировки с помощью *контекстных строк* позволяет управлять трассировкой, используя номера соответствующих контекстных строк. Т.к. контекстные строки создаются как для операторов, так и для описаний переменных, управление может выполняться и на уровне операторов, и на уровне переменных. Управление может быть статическим, т.е. пользователь задает параметры при запуске программы. Например, если первоначально обнаруживается расхождение в *окончательных результатах* работы программы, то пользователь может заказать трассировку только интересующих его переменных и найти первое расхождение *в них*. Проанализировав оператор, вызвавший первое расхождение, пользователь может изменить список контролируемых переменных и так за несколько шагов добраться до первопричины.

Управление сбором трассы может быть также автоматическим (динамическим), т.е. трассировщик по некоторым критериям может сам выбирать, что трассировать. Первый критерий, как развитие возможностей сравнительного отладчика DVM-программ, состоит в выборе трассируемых итераций – трассируется первая итерация в каждом блоке (chunk) итераций, выделенных некоторой нитью (именно на них можно ожидать проявления ошибок типа чтения неинициализированной переменной, необъявленной зависимости между витками цикла и т.п.), и итерация с последним значением переменной цикла (здесь возможна ошибка, связанная с отсутствием передачи последнего значения приватной переменной). Другой критерий – сокращение объема трассы за счет отключения контроля при повторных выполнениях одних и тех же операторов.

Поскольку номера «интересных» итераций и операторов при последовательном выполнении предсказать невозможно, то сначала выполняется отладочный прогон программы на нескольких нитях с накоплением трассы. Затем автономной программой полученная трасса приводится к *последовательному* виду, т.е. записи трассы располагаются в таком порядке, в которой они были бы выданы при последовательном выполнении. Заметим, что операторы в параллельном регионе OpenMP, но вне блока разделения работы (DO или SECTIONS) выполняются всеми нитями, т.е. могут дублироваться. Программа преобразования трассы сообщает о несоответствиях в этих записях, в частности, о зависимости работы от номера нити. Такая зависимость, вообще говоря, делает невозможным сравнение с последовательной программой.

Затем *запускается* инструментированная последовательная программа и полученная «последовательная» трасса сравнивается динамически с реальным вычислением. Для сокращения времени работы используется динамическое включение и отключение точек трассировки (сравнения) на нескольких уровнях. Самый внешний уровень отключения – это уровень итераций. Далее на каждой итерации из трассы известно *ожидаемое* событие. Отладчик может избирательно включить именно этот оператор, т.е. поставить признак обработки только для него. Реакция на этот признак может быть *внешней*, т.е. в инструментированной программе сгенерирована проверка этого признака и обход вызова подпрограмм отладчика (предпочтительно), или *внутренней*, т.е. на выставленные признаки реагирует сам отладчик, но это делается уже после вызова соответствующих подпрограмм, что дает заметные накладные расходы.

Таким образом, сравнительный отладчик OpenMP-программ состоит из трех частей: подпрограммы сбора трассы, преобразования трассы и динамического сравнения трассы. Первая и последняя реализованы на базе описанной выше инструментации OpenMP-программ в виде модулей, которые компилируются и линкуются с отлаживаемой программой. Один – при компиляции в режиме OpenMP, а другой - при компиляции в обычном последовательном виде. Программа преобразования трасс автономна. Заметим, что все три(!) операции могут выполняться на разных платформах. Трассы формируются в символьном виде для упрощения их переноса с платформы на платформу, причем формат (точность) выдачи чисел с плавающей точкой может задаваться при компиляции программы (и, конечно, должен быть одинаковым для обоих выполнений). При переходе на двоичный формат трасс преобразование формата для другой платформы могла бы выполнять программа преобразования трасс.

Работа выполнялась в рамках научно-технической программы Союзного государства «СКИФ-ГРИД», Программы №14 Президиума РАН, а также была поддержана грантом Президента РФ для ведущих научных школ НШ-2139.2008.9 и грантом РФФИ № 08-07-00086.

ЛИТЕРАТУРА:

1. OpenMP Forum, “OpenMP: A Proposed Industry Standard API for Shared Memory Programming,” October, 1997. <http://www.openmp.org>.
2. OpenMP Application Program Interface. Version 2.5 May 2005
3. James Cownie and Shirley Moore. Portable OpenMP debugging with TotalView, Etnus LLC Framingham MA, USA
4. Sun Studio 12: Thread Analyzer User's Guide

5. В.А. Крюков, Р.В.Удовиченко, "Отладка DVM-программ", Препринт ИПМ им. М.В.Келдыша РАН №56, 1999
6. В.А. Крюков, Р.В.Удовиченко, "Отладка DVM-программ", Программирование 2001,
7. В.Ф. Алексахин, К.Н. Ефимкин, В.Н. Ильяков, В.А. Крюков, М.И. Кулешова, Ю.Л. Сазанов. Средства отладки MPI-программ в DVM-системе. Труды Всероссийской научной конференции «Научный сервис в сети № 3
8. Bernd Mohr , Allen D. Malony, Hans-Christian Hoppe, Frank Schlimbach, Grant Haab, Jay Hoeflinger, and Sanjiv Shah. A Performance Monitoring Interface for OpenMP.
9. Bernd Mohr, Allen D. Malony, Sameer Shende and Felix Wolf.Design and Prototype of a Performance Tool Interface for OpenMP. Dept. of Computer and Information Science +Research Centre J'ulich, ZAM. University of Oregon Julich, Germany
10. Stefan Savage, University of Washington, Michael Burrows, Greg Nelson, Patrik Sobalvarro, Digital Equipment Corporation, Thomas Anderson, University of California at Berkeley. Eraser: A Dynamic Data Race Detector for Multithreaded Programs
11. Kranthi K Gade, Prashant Puniya, Department of Computer Science, New York University. Portable Data Race and Deadlock Detector.
12. Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. Appears in the Proceedings of the 2005 USENIX Annual Technical Conference Anaheim, California, April 10–15, 2005.
13. Petersen P.M. Evaluation of Programs and Parallelizing Compilers Using Dynamic Analysis Techniques. Champaign, IL, USA: University of Illinois at Urbana-Champaign, 1993. 164 p
14. Guard Parallel Relative Debugger. <http://sourceforge.net/projects/guardsoft/>
15. Abramson D.A., Sosic R. Relative Debugging using Multiple Program Versions // Intensional Programming I. Sydney: World Scientific. 1995. <http://www.csse.monash.edu.au/%7Edavida/papers/islip.pdf>