ДИНАМИЧЕСКИЙ КОНТРОЛЬ КОРРЕКТНОСТИ OPENMP-ПРОГРАММ

А.А. Смирнов

Введение. В современном мире все чаще и чаще возникает необходимость создавать параллельные программы, чтобы воспользоваться в полной мере теми возможностями, которые предоставляют современные вычислительные машины. Практически все новые процессоры содержат более одного ядра, а следовательно для их эффективного использования требуется создавать программы, с привлечением технологий параллельного программирования на моделях общей памяти. К таким технологиям как раз и относится OpenMP[1].

В технологии OpenMP замечательно то, что, во-первых, она позволяет превращать последовательные программы в параллельные, во многих случаях с минимальными изменениями исходного кода программы. Во-вторых, OpenMP поддерживается многими компиляторами, позволяя разработчику пользоваться привычным инструментарием и легко переносить программу на другие платформы. В-третьих, директивы в OpenMP понятны и удобны в использовании, благодаря чему программист может сосредоточиться на разработке алгоритма, а не отвлекаться на решение проблем коммуникаций между нитями.

Однако, простота и понятность OpenMP несут в себе много подводных камней, которые могут привести к неожиданным результатам работы программы. Почти в каждой директиве есть свои правила умолчания, предопределяющие поведения программы, в случае если программист что-либо не указал. Это может привести к неприятным ошибкам, которые трудно обнаружить, поскольку такие ошибки могут проявляться не на каждом запуске.

Отладка любых параллельных программ - дело сложное и трудоемкое. Традиционными средствами отладки последовательных программ, такими как точки останова или отладочная печать, трудно пользоваться даже для небольших параллельных программ. Поэтому для нахождения ошибок, предпочтительнее использовать автоматические методы, одним из которых является динамический контроль корректности программы. Суть данного метода заключается в том, что во время выполнения отлаживаемой программы отладчик следит за тем, чтобы все исполняемые операторы и используемые данные были корректными, т.е. чтобы не возникало ситуаций, потенциально приводящих к ошибке, иначе он сигнализирует о наличии ошибки в программе.

Главная проблема динамического контроля корректности программы состоит в том, что в процессе отладки требуется на порядок больше памяти и в сотни, а то и в тысячи раз больше времени, нежели для выполнения этой же программы без отладки, при условии, что во время отладки производился контроль всех операторов и всех данных. Это приводит к тому, что метод динамического контроля неприменим к реальным вычислительным задачам над большими данными.

В данной статье описывается разработанный метод динамического контроля корректности OpenMP-программ. А также способы сокращения требуемых ресурсов в процессе использования отладчика.

Область применения динамического контроля корректности OpenMP-программ. Основная проблема отладчика, реализующего динамический контроль корректности OpenMP-программ, — большие требования к ресурсам в процессе отладки. Поэтому необходимо определить те виды часто допускаемых ошибок, которые отладчик должен обнаруживать.

В 2004-2005 годах в University of Kassel(Германия) проводилось исследование[2], целью которого было выявление наиболее часто совершаемых ошибок, обусловленных некорректным использованием функций и директив OpenMP, и приводящих к неверному выполнению программы.

По результатам исследования наиболее часто допускаемыми ошибками были: незащищенный доступ к общей памяти и использование приватных переменных, которые не были отмечены как private. Эти две ошибки по своей сути создают одну и ту же ситуацию, называемую data race, следствием которой является недетерминированный результат программы. Объединим эти две ошибки в одну группу и назовем ее *ошибками общей памяти*. Ошибки данного вида возникают, когда несколько нитей работают с общей памятью без какой-либо синхронизации. В этом случае возможны следующие ситуации:

- все нити только читают переменную, тогда ошибки нет, т.к. значение переменной в любой момент времени остается неизменным.
- все нити только пишут в переменную. Поскольку они это делают одновременно, то нельзя определить, какое значение получит переменная после выполнения всех операций записи. И значение данной переменной будет при каждом запуске программы различным.
- часть нитей читают переменную, а часть пишут в нее. Здесь помимо эффекта предыдущего случая, наблюдается аналогичная неопределенность. Когда какая-либо нить пытается прочитать значение переменной, то неизвестно, какое именно значение будет получено.

Следует упомянуть еще об одном виде ошибок — *ошибки инициализации*. Суть этой ошибки

заключается в том, что в программе может возникнуть ситуация, когда происходит чтение переменной, которой еще не присвоили начальное значение. Технология OpenMP только увеличивает число возможных причин этой ошибки. Дело в том, что приватные переменные могут быть определены по-разному. Обычная приватная переменная (private) при входе в область, где она таким образом определена, не имеет начального значения. Однако, если сделать ее firstprivate, то кроме того, что она становится приватной, так ей еще будет присвоено значение, которое переменная имела до данной области. В OpenMP существуют разные параметры директив, которые аналогичным образом определяют передачу значений от исходных приватным переменным и от приватных исходным. Поэтому, если программист неправильно задал тип приватной переменной, то, скорее всего, проявится именно эта ошибка.

Таким образом, предлагается использовать динамический контроль корректности OpenMP программ для обнаружения наиболее часто встречаемых и трудно обнаруживаемых ошибок, а именно, ошибок общей памяти и инициализации.

Принцип работы отладчика OpenMP программ. Отладчик работает по следующему принципу. В исходный код проверяемой программы встраиваются вызовы интерфейсных функций отладчика, так называемый процесс инструментации. Полученный проинструментированный код компилируется с подключением библиотеки отладчика. В результате получается программа, которая при запуске выдает сведения о найденных ошибках.

Более подробно, в процессе инструментации в код вставляются вызовы функций отладчика в тех местах, которые отладчику необходимо отслеживать в процессе выполнения программы, а именно: встреченные директивы OpenMP, чтение и запись переменных и массивов, вызовы функций и т.д. Таким образом отладчик может следить за тем, какой оператор программы сейчас выполняется и соответствующим образом на это реагировать. После компиляции получается программа с интегрированным в нее отладчиком, которая при запуске, можно сказать, сама в себе находит ошибки.

Описание отладчика. Алгоритм работы отладчика состоит в том, чтобы во время выполнения программы поддерживать в актуальном состоянии и анализировать модель, описывающую взаимосвязь между данными и нитями при каждом обращении к отладчику.

Для обнаружения ошибок общей памяти и инициализации, необходимо отслеживать все обращения к памяти, к тому же нужно еще хранить информацию об этих обращениях, чтобы можно было обработать полученные данные и выявить ошибки. Поэтому для каждой переменной или элемента массива нужно иметь информацию (VarInfo), которая бы говорила, какая нить читала или писала в эту переменную.

Обнаружение ошибок может производиться двумя способами. Первый — каждая нить при обращении к переменной изменяет соответствующую структуру VarInfo, общую для всех нитей. При этом, чтобы не возникало ошибок общей памяти внутри самого отладчика, используются критические секции. Как показал эксперимент такой метод хоть и требует относительно не большой объем памяти, но замедляет выполнение программы в десятки тысяч раз. Так что на практике такой способ совершенно не приемлем.

Во втором подходе, который был реализован в отладчике, во время параллельной работы нитей происходит сбор информации об обращениях к переменным, после чего в точках синхронизации (явный и неявный barrier) вся накопленная информация анализируется и делаются выводы о наличии ошибок. Этот метод значительно быстрее первого, однако, он требует гораздо больше памяти. Особенно это заметно при работе с большими массивами, ведь размер структуры VarInfo зачастую в несколько раз больше самого массива, а таких структур нужно как минимум по одной для каждой нити. По этой причине, динамический контроль корректности в чистом виде не применим для реальных задач над большими данными. Для этого необходимо применять методы сокращения затрачиваемых на отладку ресурсов, о которых написано дальше.

Поскольку во время выполнения существуют несколько параллельно работающих нитей, которые объединены в группы, соответствующие параллельным областям, которые могут быть вложенными, то для анализа текущего состояния программы удобно построить дерево, отображающее взаимосвязь нитей в данный момент. Это позволит сохранять состояния нитей перед входом во вложенную параллельную область и обеспечить защиту общей части дерева от параллельного доступа несколькими нитями.

Алгоритмы динамического контроля корректности были реализованы в отладчике для Fortran программ. Чтобы представить себе степень замедления программы в процессе отладки достаточно взглянуть на таблицу 1, в которой, для примера, приведены средние времена выполнения тестов класса S из набора NPB 2.3 (тесты выполнялись на 2 нитях).

Таблица 1. Времена выполнения тестов из набора NPB 2.3 класса S

Название теста	Без отладки (сек)	С отладкой (сек)
BT	~0,08	~36,53
CG	~0,06	~57,72
EP	~0,93	~50,72
FT	~0,14	~27,45
LU	~0,08	~14,25
MG	~0,01	~7,06
SP	~0,06	~28,93

Отсюда видно, что во время отладки программа замедляется в несколько сотен раз. При этом при увеличении размерности программы наблюдается также и рост замедления. Например, тест SP класса W, на 2 нитях выполнился без отладки за 4,72 секунды, а с отладкой — за 5006,33 сек. При этом замедление превысило тысячу раз.

Оптимизация отладчика. Реальные производственные программы сами по себе требуют значительных ресурсов вычислительных систем. А при использовании динамического отладчика этих ресурсов необходимо в сотни раз больше. Поэтому, чтобы сделать возможным применение динамического отладчика, нужен метод сокращения затрат.

Обычно программы на Фортране содержат несколько больших массивов и циклы, в которых выполняются типовые действия над элементами этих массивов. Отсюда получается, что большая часть времени потрачена отладчиком на выполнение однотипных операций, выдающих один и тот же результат. Кроме того часто, если возникает ошибка на каком-либо элементе массива, то она будет обнаружена так же, если не на всех элементах, то на большом их подмножестве. Таким образом появляется две возможности сократить используемые отладчиком ресурсы. Первая — это исследовать не все итерации циклов, а только некоторые, что приведет к увеличению скорости работы отладчика. Вторая — обрабатывать не все элементы массивов, а только некоторое их подмножество, что уменьшит затраты по памяти.

Очевидно, что в зависимости от выбора набора контролируемых итераций цикла, изменяется и число ошибок, которые отладчик может обнаружить. Это значит, что если убрать из набора итерации, на которых проявляется ошибка, то отладчик ее не заметит. Тоже самое касается и выбора представительных элементов массива. Например, если массив будет проинициализирован везде, кроме одного элемента, не включенного в число представительных, то скорее всего возникнет ошибка инициализации, которую отладчик пропустит. Так что следует определить представительные множества таким образом, чтобы свести к минимуму вероятность пропуска ошибки.

Основное достоинство представительных итераций заключается в том, что отладчик может обрабатывать очень незначительное число итераций, по сравнению с их общим количеством в выполняемой программе. Это приводит к сильному сокращению времени отладки программы.

Явным минусом такого подхода является возможность пропуска какой-либо ошибки, однако всегда можно подобрать такое множество итераций, на котором бы обнаруживались все существующие ошибки.

Метод представительных элементов массива позволяет ощутимо сократить затраты по памяти в процессе отладки. Поскольку отладчик работает только с определенным подмножеством всех элементов массива, он может не хранить информацию для остальных элементов. Кроме того, отладчик игнорирует обращения к непредставительным элементам массивов, что положительно сказывается на скорости выполнения программы, но необходимая проверка, является ли элемент массива представительным, практически полностью перечеркивает достигнутый прирост скорости.

Отрицательной стороной метода, как и для предыдущего, является возможность пропуска ошибки, в случае, если она возникает на определенных элементах массива, не вошедших в число обрабатываемых. Однако, опять-таки эту проблему можно решить подходящим подбором наборов элементов массива.

Для того, чтобы воспользоваться преимуществами обоих методов предлагается следующий алгоритм отладки программы. Он состоит из нескольких этапов, каждый из которых предполагает компиляцию и запуск отлаживаемой программы и получение промежуточного результата, который может быть использован многократно. Так же промежуточные данные могут быть созданы или дополнены другими путями, без запуска отлаживаемой программы, например, статическими анализаторами. Итак, сам метод состоит из:

1. нахождение базового набора представительных итераций
2. нахождение набора представительных элементов массивов
3. расширение набора представительных итераций
4. отладка программы

1.Нахождение базового набора представительных итераций. Вначале определим для произвольного

цикла набор итераций, на которых вероятнее всего может произойти ошибка. Пусть этот набор включает в себя граничные итерации циклов и итерации, на которых выполняется код, недостижимый на уже включенных в набор итерациях. Конечно, для получения такого набора нужен отдельный запуск отлаживаемой программы в режиме нахождения представительных итераций. Но построенный таким образом набор представительных итераций полностью покрывает достижимый в ходе выполнения программы код, с одной оговоркой, что программа будет запущена в тех же условиях, над теми же данными. Благодаря этому вероятность пропустить ошибку достаточно мала. К тому же, замедление выполнения программы в процессе сбора итераций по отношению к выполнению программы без отладки и инструментации составляет всего несколько раз, что ничтожно мало по сравнению с замедлением в процессе полной отладки.

В результате мы поучаем базовый набор представительных итераций циклов, причем состоять он будет из крайне малого числа итераций.

2.Нахождение набора представительных элементов массивов. Теперь, получив базовый набор представительных итераций для каждого цикла, можно построить набор представительных элементов массивов. Для этого потребуется еще один запуск программы, во время которого отладчик определит, какие элементы массива были использованы во время выполнения программы, при условии, что в циклах обрабатывались только представительные итерации. Полученные элементы и будут искомыми наборами. Этот метод хорош тем, что исключает уменьшение количества находимых ошибок, при использовании в совокупности с представительными итерациями. Т.е., если на представительных итерациях обнаруживается ошибка, то она будет выявлена на элементах массивов из этих множеств. Скорость процесса сбора элементов массивов сопоставима со скоростью поиска представительных итераций.

3.Расширение набора представительных итераций. Множество представительных итераций следует расширить для того, чтобы отладчик мог обнаружить ошибки общей памяти, возникающие на исследуемых элементах массивов. Так как ошибка общей памяти требует для нахождения как минимум обращения двух нитей (2 итераций параллельного цикла), то базового набора представительных итераций не достаточно. Может возникнуть такая ситуация: ошибка существует на одном элементе массива, но представительная итерация, которая работает с этим элементом только одна, следовательно ошибка не будет найдена. Поэтому, чтобы избежать данной ситуации нужно расширить множество обрабатываемых итераций.

На третьем этапе, после нахождения представительных элементов массивов, следует, при очередном запуске программы, запомнить все итерации циклов, на которых были зафиксированы обращения к представительным элементам какого-либо массива. В результате мы получим расширение первоначального множества представительных итераций. И новое множество уже способно обеспечить обнаружение ошибок общей памяти на выбранных элементах массивов, поскольку любое обращение в циклах к исследуемому элементу массива попадет на представительную итерацию.

4.Отладка программы. На четвертом этапе производится сама отладка программы, но уже с представительными элементами массивов и на расширенном наборе представительных итераций. В результате чего, достигается существенное сокращение расходов по памяти и ускорение по отношению к полной отладке, что позволяет применять разработанный динамический отладчик для реальных приложений с большими объемами данных.

Метод отлично подходит для случаев, когда зависимости итераций циклов по данным выражаются непрерывными функциями. Но в других случаях, например, когда индекс массива зависит от времени или же может принимать произвольные значения, отладчик может пропустить ошибку.

Были проведены эксперименты, с тем, насколько сокращается время отладки программы, если отладчик будет обрабатывать только представительные итерации, полученные на первом этапе. В качестве отлаживаемых программ были взяты несколько тестов из набора NPB 2.3 класса A. В таблице 2 приведены полученные результаты.

Таблица 2. Время выполнения тестов класса A без отладки и с отладкой на представительных итерациях.

Название теста	Без отладки (сек)	С отладкой (сек)
BT	54.97	142.71
CG	2,91	17,49
EP	18,55	22,22
FT	5,25	109,18
LU	72,03	141,51

Из таблицы видно, что отладка на представительных итерациях значительно быстрее полной, замедление которой может достигать тысячи раз. В итоге можно говорить о том, что отладка по такому алгоритму больших программ возможна.

Заключение. В настоящее время отладка параллельных программ остается трудным и долгим делом.

Существующие инструменты автоматического контроля корректности (такие как Intel Thread Checker[3,4], Sun Thread Analyzer[3,5]) из-за чрезмерных требований к ресурсам системы оказываются не применимы для отладки реальных приложений. В данной статье приводятся результаты работы, целью которой было создание средства отладки, позволяющего автоматически обнаруживать ошибки в OpenMP программах. При этом отладчик должен быть применим к большим научно-техническим программам, оперирующими значительными объемами данных. В результате работы были разработаны алгоритмы поиска часто допускаемых при использовании технологии OpenMP ошибок. Разработанные алгоритмы были реализованы в отладчике Fortran программ. Были проведены тестирования, показавшие высокие требования к ресурсам в процессе отладки. Также были разработаны методы сокращения накладных расходов отладчика, что привело к значительному ускорению и снижению требуемой памяти при отладке программы. Проведенные эксперименты показали, возможность использования разработанных методов для отладки реальных научно-технических приложений.

ЛИТЕРАТУРА:

1. OpenMP Application Program Interface Version 3.0 May 2008 [PDF](http://www.openmp.org/mp-documents/spec30.pdf)
2. M. Suess, C. Leopold. Common mistakes in OpenMP and how to avoid them. 2006. [PDF] (http://www.michaelsuess.net/publications/suess_leopold_common_mistakes_06.pdf)
3. Christian Terboven. Comparing Intel Thread Checker and Sun Thread Analyzer. 2007. [PDF](http://www.fz-juelich.de/nic-series/volume38/terboven.pdf)
4. Intel Thread Checker 3.1 – Documentation.[PDF] (http://software.intel.com/en-us/articles/intel-thread-checker-documentation/)
5. Sun Studio 12: Thread Analyzer User'sGuide. 2007 [HTML,PDF](http://docs.sun.com/app/docs/doc/820-0619)