VivaMP, система выявления ошибок в коде параллельных программ на языке Си++ использующих OpenMP

А.П. Колосов, Е.А. Рыжков, А.Н. Карпов

В статье приводятся результаты исследований ошибок, которые допускают программисты, использующие Си++ и OpenMP. Для автоматического обнаружения этих ошибок предлагается использование статического анализа. Описывается интегрирующийся в среду Visual Studio анализатор VivaMP, реализующий поставленную задачу.

1. Введение

По оценкам компании Evans Data, проводящей опросы среди разработчиков ПО, общее количество программистов в мире к 2009 году составит 17 миллионов человек[1]. На сегодняшний день 40% из них используют язык Си++, причем около 70% разработчиков занимаются разработкой многопоточных приложений [2]. По данным того же опроса, 13,2% этих разработчиков считают, что главной проблемой таких разработок является нехватка программных средств для создания, тестирования и отладки параллельных приложений. Следовательно, в решении задачи автоматического поиска ошибок в исходном коде непосредственно заинтересованы примерно 630.000 программистов.

Целью работы является создание статического анализатора кода, предназначенного для автоматического обнаружения таких ошибок. В исследовании рассматривался язык Си++, поскольку к коду именно на этом языке чаще всего предъявляются требования высокой производительности. Так как поддержка технологии OpenMP встроена в Microsoft Visual Studio 2005 и 2008, и некоторые специалисты считают, что именно технология OpenMP вскоре приобретет наибольшую популярность [3], рассматривалась именно эта технология.

Анализ обзоров отладчиков для параллельных программ показывает, что ситуация в этой сфере до сих пор далека от идеала. Применительно к отладке программ, написанных на Си++ и использующих OpenMP, как правило, упоминаются TotalView[1] и Intel Thread Checker[2]. Однако, оба эти инструмента предназначены для динамического использовния (это я к тому поправил, что TotalView это отладчик с неоторыми доп. возможностями). До недавнего времени направление статического анализа OpenMP программ практически было не освоено. В качестве примера можно привести, пожалуй, только достаточно качественную диагностику, выполняемую компилятором Sun Studio. Статический анализатор VivaMP заполнил это нишу.

2. Применение статического анализа для отладки параллельных программ

Большинство существующих на данный момент средств отладки параллельных программ являются динамическими анализаторами, предполагающими непосредственное выполнение анализируемой программы. Такой подход имеет свои преимущества, но у него имеются и недостатки.

Динамический анализ предполагает сбор данных только во время выполнения программы, следовательно, нет никакой гарантии, что проверен будет весь код. Более того, данный подход требует от программиста многократного повторения одних и тех же действий, либо применения средств автоматического тестирования для имитации действий пользователя.

Помимо этого при динамическом анализе код отлаживаемого приложения подвергается инструментированию, что снижает быстродействие программы, а изредка может даже приводить к сбоям. Поскольку сбор и анализ информации с целью улучшения быстродействия, как правило, откладывается на конец динамического анализа, в случае критической ошибки в анализируемом

[1] http://www.totalviewtech.com/products/totalview.html
[2] http://www.intel.com/cd/software/products/asmo-na/eng/286406.htm

приложении все результаты анализа оказываются потеряны.

Наконец, динамический анализ далеко не всегда позволяет обнаружить конкретный фрагмент кода, который привод к неожиданному поведению.

Статический анализ позволяет просмотреть весь исходный код приложения, не требует от программиста никаких дополнительных усилий и приводит к обнаружению опасных фрагментов кода. Этот подход, конечно, тоже не является идеальным. Недостаток статического анализа заключается в том, что он не позволяет проверить поведение, зависящее от пользователя. Еще одной проблемой являются ложные срабатывания, уменьшение количества которых требует дополнительных усилий при разработке анализатора. Подробнее вопрос применения статического анализа при разработке параллельных программ рассматривается в статье [4].

В анализаторе VivaMP используется анализ с обходом дерева кода (tree walk analysis). Помимо этого существуют и другие виды статического анализа, предполагающие моделирование выполнения программы, расчет возможных значений переменных и путей выполнения кода. Статический анализ как средство диагностики ошибок в параллельных программах был выбран потому, что данный подход позволяет находить многие ошибки, не диагностируемые динамическими анализаторами. Теперь рассмотрим сами ошибки подробнее.

3. Диагностируемые ошибки

Список возможных ошибок, не обнаруживаемых компилятором Visual Studio, был составлен в результате исследования работ, посвященных параллельному программированию с использованием OpenMP, а также на основе личного опыта авторов. Все ошибки можно разделить на четыре основные категории:

- Недостаточное знание синтаксиса OpenMP.
- Недостаточное понимание принципов работы OpenMP.
- Некорректная работа с памятью (незащищенный доступ к общей памяти, отсутствие синхронизации, неправильный режим доступа к переменным, и т. п.).
- Ошибки производительности

Первые три вида ошибок являются логическими ошибками, которые приводят к изменению логики работы программы, к неожиданным результатам и (в некоторых случаях) к аварийному завершению программы. Последняя категория объединяет ошибки, приводящие к снижению быстродействия.

Приведем примеры ошибок каждого вида и их краткое описание.

3.1. Отсутствие ключевого слова parallel

Рассмотрим простейшую ошибку, которая может возникнуть при неправильном написании директив OpenMP. Поскольку эти директивы имеют достаточно сложный формат, такую ошибку по тем или иным причинам может допустить любой программист. Пример корректного и некорректного кода приведен на Рис. 1.

```
// Некорректно:
  #pragma omp for
  ... // цикл for
// Корректно:
  #pragma omp parallel for
  ... // цикл for
// Корректно:
  #pragma omp parallel
  {
```

```
    #pragma omp for
    ... // цикл for
  }
```

Рис. 1. Пример ошибки, вызванной отсутствием ключевого слова parallel

Приведенный выше фрагмент некорректного кода будет успешно скомпилирован компилятором, даже без предупреждений. Некорректная директива будет проигнорирована, и цикл, следующий за ней, выполнится только одним потоком. Распараллеливания не произойдет, и обнаружить это на практике будет почти невозможно. Однако, статический анализатор легко укажет на этот потенциально некорректный участок кода.

3.2. Неправильное применение блокировок

Если программист использует для синхронизации и/или защиты объекта от одновременной записи функции вида omp_set_lock, каждый поток должен содержать вызовы соответствующих функций omp_unset_lock, причем с теми же переменными. Попытка освобождения блокировки, захваченной другим потоком, или отсутствие вызова разблокирующей функции может привести к ошибкам во время выполнения программы и бесконечному ожиданию. Рассмотрим пример кода (см. Рис. 2):

```
// Некорректно:
  omp_lock_t myLock;
  omp_init_lock(&myLock);
  #pragma omp parallel sections
  {

      #pragma omp section
      {
            ...
            omp_set_lock(&myLock);
            ...
      }
      #pragma omp section
      {
            ...
            omp_unset_lock(&myLock);
            ...
      }
  }
// Некорректно:
  omp_lock_t myLock;
  omp_init_lock(&myLock);
  #pragma omp parallel sections
  {

      #pragma omp section
      {
            ...
            omp_set_lock(&myLock);
            ...
      }
      #pragma omp section
      {
            ...
            omp_set_lock(&myLock);
```

```
            omp_unset_lock(&myLock);
            ...
        }
    }

// Корректно:
    omp_lock_t myLock;
    omp_init_lock(&myLock);
    #pragma omp parallel sections
    {

        #pragma omp section
        {
            ...
            omp_set_lock(&myLock);
            ...
            omp_unset_lock(&myLock);
            ...
        }
        #pragma omp section
        {
            ...
            omp_set_lock(&myLock);
            ...
            omp_unset_lock(&myLock);
            ...
        }
    }
```

Рис. 2. Пример некорректного использования блокировок

Первый пример некорректного кода приведет к ошибке во время выполнения программы (поток не может освободить переменную, занятую другим потоком). Второй пример иногда будет работать корректно, а иногда будет приводить к зависанию. Зависеть это будет от того, какой поток завершается последним. Если последним будет завершаться поток, в котором выполняется блокировка переменной без ее освобождения, программа будет выдавать ожидаемый результат. Во всех остальных случаях будет возникать бесконечное ожидание освобождения переменной, захваченной потоком, работающим с переменной некорректно.

Теперь рассмотрим другую ошибку более подробно и приведем соответствующее правило для анализатора.

3.3. Незащищенный доступ к общей памяти

Эта ошибка может встретиться в любой параллельной программе, написанной на любом языке. Также она называется состоянием гонок (race condition) и суть ее заключается в том, что значение общей переменной, изменяемой одновременно из нескольких потоков, в результате может оказаться непредсказуемым. Рассмотрим простой пример для Си++ и OpenMP (см. Рис. 3):

```
// Некорректно:
    int a = 0;
    #pragma omp parallel
    {
        a++;
    }

// Корректно:
```

```
int a = 0;
#pragma omp parallel
{
    #pragma omp atomic
    a++;
}
```

Рис. 3. Пример состояния гонок

Эту ошибку также можно обнаружить средствами статического анализатора. Рассмотрим правило, по которому статический анализатор VivaMP сможет обнаружить эту ошибку:

«Опасным следует считать инициализацию или модификацию объекта (переменной) в параллельном блоке, если объект относительно этого блока является глобальным (общим для потоков).

К глобальным объектам относительно параллельного блока относятся:

1. Статические переменные.
2. Статические члены класса (В версии VivaMP 1.0 не реализовано).
3. Переменные, объявленные вне параллельного блока.

Объект может быть как переменной простого типа, так и экземпляром класса. К операциям изменения объекта относится:

1. Передача объекта в функцию по не константной ссылке.
2. Передача объекта в функцию по не константному указателю (В версии VivaMP 1.0 не реализовано).
3. Изменение объекта в ходе арифметических операций или операции присваивания.
4. Вызов у объекта не константного метода».

На первый взгляд правило кажется не очень сложным. Однако, чтобы избежать ложных срабатываний, приходится вводить множество исключений. Проверяемый код следует считать безопасным если:

1. Код не находится в параллельном блоке (нет директивы "parallel").
2. Модификация объекта защищена директивой "atomic".
3. Код находится в критической секции, заданной директивой "critical".
4. Код находится в критической секции, образованной функциями вида omp_set_lock.
5. Код находится в блоке директивы "master".
6. Код находится в блоке директивы "single".
7. К объекту (переменной) применена директива "threadprivate", либо выражение "private", "firstprivate", "lastprivate" или "reduction". Это исключение не касается статических (static) переменных и статических полей классов, которые всегда являются общими.
8. Инициализация или модификация объекта осуществляется внутри оператора for (внутри самого оператора, а не внутри тела цикла). Такие объекты согласно спецификации OpenMP автоматически считаются локальными (private).
9. Модификация объекта осуществляется внутри секции, заданной директивой section.

Пользуясь этим правилом и перечисленными исключениями, анализатор сможет обнаружить ошибку по дереву кода.

В заключение этого раздела отметим, что более полный список обнаруженных в результате исследований ошибок и их более подробные описания можно найти в статье [5]. Помимо этого пример программы, демонстрирующей ошибочные и исправленные версии проблемного кода можно найти на сайте проекта VivaMP[1].

Теперь перейдем к описанию самого анализатора.

4. Анализатор VivaMP

Анализатор VivaMP разработан на основе библиотеки анализа кода VivaCore и вместе с

[1] http://www.viva64.com/ru/vivamp-tool

анализатором Viva64 составляет единую линейку продуктов в области тестирования ресурсоемких приложений. Viva64 предназначается для поиска ошибок, связанных с переносом 32-битного ПО на 64-битные платформы. VivaMP предназначен для проверки параллельных приложений, построенных на базе технологии OpenMP. Как и Viva64, VivaMP интегрируется в среду разработки Visual Studio 2005/2008, добавляя новые команды в интерфейс. Настройка анализатора делается через стандартный для среды механизм, диагностические сообщения выводятся так же, как сообщения стандартного компилятора – в окна Error List и Output Window. Помимо этого подробное описание ошибок приведено в справочной системе анализатора, интегрирующейся в справку Visual Studio. Контекстная справка реализована через стандартный механизм среды. Интерфейс анализатора VivaMP, интегрированного в среду Visual Studio приведен на Рис. 4.

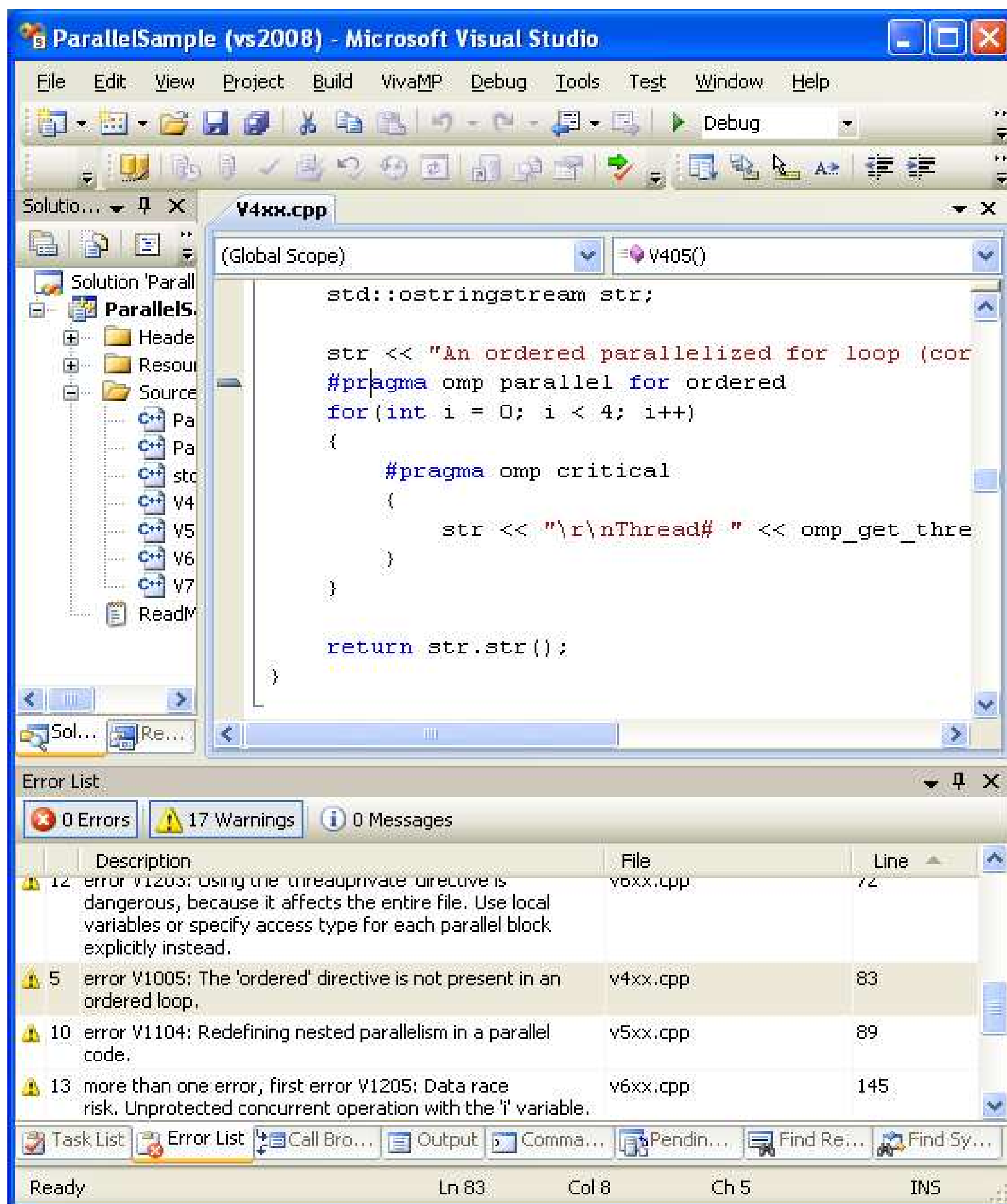

Рис. 4. Интерфейс VivaMP.

На данный момент выпущена первая версия анализатора, информацию о которой можно получить по адресу http://www.viva64.com/vivamp-tool. Первая версия VivaMP диагностирует около двух десятков ошибок, однако, количество собранного материала и результаты экспериментов позволяют существенно увеличить это число (как минимум в два раза) в последующих версиях. Кроме того, работа по поиску новых ошибок продолжается и сейчас. Если вам, уважаемые коллеги, известны какие-либо паттерны таких ошибок, мы будем благодарны, если вы сообщите нам о них, используя контактную информацию с упомянутого выше сайта проекта VivaMP.

Кроме того, нам интересно протестировать наш анализатор на реальных серьезных проектах, применяющих технологию OpenMP. Если вы разрабатываете такой проект и нуждаетесь в анализаторе кода, пожалуйста, свяжитесь с нами.

Литература

1. Timothy Prickett Morgan Evans Data Cases Programming Language Popularity: [http://www.itjungle.com/tug/tug121406-story03.html], 14.12.2006.
2. Janel Garvin Evans Data: Market Situation and Predictions: [http://www.softwareproductconference.com/prague/_doc/presentations/Janel_Garvin_Market_Situation.pdf], 8.04.2008
3. Michael Suess Why OpenMP is the way to go for parallel programming: [http://www.thinkingparallel.com/2006/08/12/why-openmp-is-the-way-to-go-for-parallel-programming/], 12.08.2006.
4. Е.А. Рыжков, О.С. Середин. Применение технологии статического анализа кода при разработке параллельных программ. Известия ТулГУ. Технические науки. Вып.3. – Тула: Изд-во ТулГУ, 2008. – 267 с. Стр. 191 – 196.
5. Алексей Колосов, Евгений Рыжков, Андрей Карпов. 32 подводных камня OpenMP при программировании на C++. RSDN Magazine #2-2008. Стр. 3 – 17.