УДК 004.4'233

СЕМАНТИЧЕСКИЕ ОШИБКИ В ПАРАЛЛЕЛЬНЫХ ПРОГРАММАХ ДЛЯ СИСТЕМ С РАСПРЕДЕЛЕННОЙ ПАМЯТЬЮ И МЕТОДЫ ИХ ОБНАРУЖЕНИЯ СОВРЕМЕННЫМИ СРЕДСТВАМИ ОТЛАДКИ

К. Е. Афанасьев, А. Ю. Власенко

В условиях постоянно растущего спроса на вычислительные ресурсы со стороны естественных, социальных и других наук все более остро ощущается потребность в качественном и удобном для исследователя программном обеспечении, позволяющем использовать данные ресурсы, предоставляемые вычислительными центрами. На сегодняшний день вычислительные кластеры, объединяющие множество узлов, каждый из которых обладает своей оперативной памятью – самый популярный вид архитектуры высокопроизводительных систем. Стандартом де-факто при программировании вычислительных кластеров является интерфейс MPI (Message Passing Interface), который очень сложен в использовании и требует со стороны программиста управления пересылками данных на низком уровне. Отладка MPI-приложений, где помимо ошибок последовательных программ могут появляться новые ошибки, обусловленные недетерминированным поведением независимых процессов, – очень трудоемкий процесс, для которого необходимо применение специальных инструментальных систем. Поэтому классификация ошибок параллельных программ и анализ методов, а также существующих программных средств отладки таких программ – крайне важная и актуальная в настоящее время задача.

In conditions of constantly increasing demand on computing resources from natural, social and other sciences need of qualitative and easy-to-use software for performing computations on these resources in data centers also increases. Computing clusters, which contain many nodes, each of which has its own operating memory, are the most popular architectural solutions for high performance systems today. MPI (Message Passing Interface) is a defacto standard for programming computing clusters. MPI is very complex and requests managing of data transmitions between nodes in low level from programmer. Debugging of MPI applications is very hard process, because of new errors appearing from undeterministic behavior of independent processes. That is why applying special instrumental systems is essential for parallel debugging. So classification of errors in parallel applications and analysis of methods and existing debugging systems for these applications are extremely actual and important tasks nowdays.

Ключевые слова: высокопроизводительные вычислительные системы, параллельное программирование, MPI, семантические ошибки.

Введение

В настоящее время все большее значение во многих областях науки и производства приобретает компьютерное моделирование процессов, явлений и объектов реального мира. Особую роль вычислительная техника играет в таких задачах, как моделирование природных катаклизмов (цунами, торнадо, ураганы), производство новых лекарств, исследование космических процессов, в задачах генной инженерии и др. При исполнении алгоритмов решения таких задач, реализованных в виде компьютерных программ, на ЭВМ очень часто требуются аппаратные ресурсы, намного превышающие ресурсы настольного персонального компьютера. В связи с этим в последние годы многие научные центры и промышленные предприятия приобретают высокопроизводительные вычислительные комплексы, объединяющие сотни и тысячи автономных вычислительных узлов, каждый из которых снабжен своим процессором (процессорами) и оперативной памятью. Такие комплексы носят название «вычислительные кластеры».

Поскольку узлы не обладают общей памятью, то программируют кластеры следующим образом: большую расчетную задачу разбивают на части, каждая из которых может исполняться независимо от других, затем эти ветви параллельного приложения распределяют по вычислительным узлам. При исполнении параллельной программы на кластере узлу время от времени требуются данные, хранящиеся на других узлах кластера. Поскольку кластерный тип архитектуры не обладает общей памятью, то для обмена данными используются пересылки по коммуникационной сети, связывающей узлы.

Наиболее распространенным прикладным инструментом для создания параллельных программ на сегодняшний день является интерфейс MPI (Message Passing Interface) [3]. Данная технология сложна в использовании, поскольку требует пересылок данных в явном виде с указанием размера и схемы размещения в памяти передаваемых/принимаемых данных. Кроме того, при взаимодействия независимо работающих процессов на разных вычислительных узлах возникает большое множество новых, по сравнению с последовательными программами, ошибок, вызванных недетерминированным поведением параллельной программы, конфликтами при доступе к памяти основной программы и работающей в то же время MPI-функции, неверным использованием внутренних типов данных и объектов MPI и многими другими проблемами. Для обнаружения таких ошибок непригодны методы и средства, используемые применительно к последовательным программам. Поэтому отладка MPI-программ представляет собой чрезвычайно сложный и трудоемкий процесс. К данному моменту уже разработаны несколько программных систем, служащих для поддержки процесса отладки параллельных приложе-

ний. Данные системы используют различные подходы и имеют различные положительные и отрицательные стороны. Некоторые из них свободнораспространяемы, другие же являются коммерческими и обладают довольно высокой стоимостью. Таким образом, для понимания тех проблем, с которыми может столкнуться прикладной программист при разработке параллельного приложения, требуется описание и систематизация ошибок в MPI-программах, а для выбора подходящего инструмента отладки – классификация возможных подходов при построении программных средств и анализ систем, созданных к настоящему моменту.

Семантические ошибки в MPI-программах

Ошибка, которую можно определить как появление одного или более некорректных результатов на каком-либо из этапов преобразований программы из исходного кода в исполняемые файлы и последующей работы, может появиться из-за неверно составленного кода (программная ошибка) или аппаратных сбоев. Программные ошибки, в свою очередь, делятся на:

– *синтаксические* – ошибки, обнаруживаемые компилятором на стадии перевода программы в машинный код;

– и *семантические* (также называемые логическими или алгоритмическими).

Семантические ошибки можно разделить на:

– *сильные* – в этом случае программа проходит логическую последовательность состояний, которая приводит к результату, отличному от ожидаемого, или к тому, что программа не способна выполнять свои функции;

– и *слабые,* которые не являются как таковыми ошибками в смысле данного выше определения, а представляют собой причины неэффективного поведения программы пользователя (не полное использование вычислительных ресурсов, большой процент расхода времени на коммуникационные операции в общем времени работы и т. д.).

В данной работе наибольшее внимание уделено вопросам обнаружения различных логических ошибок, в связи с тем, что все новые, по сравнению с последовательными программами, проблемы относятся именно к данной категории. Из множества логических подробно разобран класс сильных ошибок, потому как поиск слабых – задача оптимизации, а не отладки параллельного приложения.

Среди сильных алгоритмических ошибок выделяют:

– *локальные* – для их обнаружения каждому процессу не требуется информация от других процессов;

– *глобальные*, которые включают 2 и более процессов. Для их нахождения нужно анализировать данные из нескольких ветвей параллельного приложения.

Следует заметить, что под локальными ошибками подразумеваются не те, которые встречаются в последовательных программах, а проблемы, возникающие вследствие некорректного использования коммуникационного интерфейса MPI в пределах одного процесса. К проблемам такого рода относятся, в частности, следующие:

– несколько MPI-функций используют одну и ту же область памяти;

– область памяти, переданная в качестве аргумента MPI-операции, не доступна для записи или чтения;

– попытка записи или чтения данных в буфер, переданный MPI-функции;

– неверная последовательность вызовов;

– программа создает слишком большое число «внутренних объектов» или типов данных (их лимит ограничен в MPI Standard) и т. д.

Глобальные ошибки, возникающие при обмене данных между ветвями параллельного приложения, можно разбить по следующим категориям:

– Ошибки синхронизации:
- дедлоки
  - потенциальные;
  - реальные;
- гонки данных
  - интерфейсные;
  - межпроцессные.

– Ошибки несоответствия:
- несоответствия в типах аргументов;
- несоответствия в длине сообщений.

Дедлоки возникают, когда набор коммуникационных функций, вызванных в различных MPI-процессах, создаёт условия продолжения работы этих процессов, которые никогда не могут быть удовлетворены некоторыми корректными MPI-реализациями [6]. В итоге каждый из таких процессов находится в ожидании сигнала о том, что в другом процессе завершено некоторое действие. Коммуникационная операция создаёт зависимость, когда MPI-стандарт позволяет реализации блокировать процесс до тех пор, пока при работе другого процесса не произойдёт некоторое событие. Например, MPI-стандарт позволяет реализовать MPI_Send (функцию отправки сообщения) как синхронную операцию, и процесс, вызвавший эту функцию, может находиться в ожидании до тех пор, пока процесс-получатель не вызовет MPI_Recv (функцию приема сообщения). С другой стороны, функция MPI_Send может быть реализована как буферизованная посылка и завершаться, не дожидаясь вызова MPI_Recv на принимающей стороне. Поэтому в случае, когда 2 процесса обоюдно вызывают MPI_Send, а затем MPI_Recv, то, в зависимости от реализации MPI, дедлок может произойти или нет.

Пример демонстрирует, что одна и та же параллельная программа в некоторых случаях может приводить к возникновению дедлока, а в некоторых – нет. На основании этого дедлоки разделяют на стандартные (реальные), которые случаются всегда, и потенциальные, которые не проявляются на данной архитектуре или реализации MPI, но могут возникнуть при переносе приложения на другую платформу.

Потенциальные условия **гонки данных** (ошибки соревнования) могут быть вызваны различными причинами, например, при использовании функции приёма с макросом MPI_ANY_SOURCE в качестве номера отправителя или макросом MPI_ANY_TAG в качестве тэга, при использовании случайных чисел и т.д. Некоторые пользователи также полагаются на то, что коллективные функции являются синхронизирующими, однако единственной такой операцией по MPI-стандарту является MPI_Barrier. Так, в программе, содержащей следующий фрагмент (листинг 1), в зависимости от реализации MPI_Bcast, нулевой процесс первым может принять сообщение от первого или второго процесса.

Листинг 1.

```
if (rank == 0) {
   MPI_Recv(..., MPI_ANY_SOURCE,
MPI_ANY_TAG,...);
   MPI_Bcast(....);
   MPI_Recv(..., MPI_ANY_SOURCE,
MPI_ANY_TAG,...);
}
else if (rank == 1) {
   MPI_Send(...);
   MPI_Bcast(...);
}
else if (rank == 2) {
   MPI_Bcast(...);
   MPI_Send(...);
}
```

Выделяют *межпроцессные* и *интерфейсные* ошибки соревнования. Межпроцессная гонка данных происходит, когда несколько процессов пытаются послать одинаковое по логике, но разное по содержанию сообщение. Интерфейсные гонки данных возникают при параллельной работе процесса и вызванной им MPI-функции. Примерами могут служить попытки записи в буфер до окончания операции MPI_Isend или попытки чтения до окончания MPI_Irecv. Интерфейсные гонки данных можно отнести к локальным ошибкам в силу того, что они образуются в пределах одного процесса, но в данной классификации они находятся в числе глобальных, так как являются результатом одновременной работы различных последовательностей команд – основным потоком MPI-процесса и потоком, порожденным для выполнения коммуникации.

Семантика MPI всегда разрешает гонки данных, поэтому программа не «зависает», но результаты могут оказаться неожиданными для пользователя. Данная группа ошибок относится к классу обусловленных недетерминированным поведением параллельной программы.

В MPI Standard определено, что аргументы, переданные каждым из процессов, участвующих в коммуникационной операции, должны согласовываться с аргументами, переданными другими процессами. В простейшем случае под «соответствующими» аргументами имеют в виду идентичные, но возможны и более сложные случаи, если в вызове коммуникационной функции фигурирует составной тип данных. Так, абсолютно корректно послать, например, два элемента составного типа (MPI_INT, MPI_DOUBLE) и получить один (MPI_INT, MPI_DOUBLE, MPI_INT, MPI_DOUBLE). MPI-реализации обычно прерывают приложение, когда есть несоответствие типов данных, например при посылке MPI_INT и получении MPI_DOUBLE, но никакой точной информации о несоответствии пользователю не выдаётся. Поэтому реализация поиска **ошибок несоответствия** при построении программной системы, служащей для поддержки процесса отладки параллельных приложений, является важной и сложной задачей.

Методы и программные средства обнаружения алгоритмических ошибок

При построении инструментальных средств, служащих для помощи прикладному программисту обнаруживать семантические ошибки в параллельных приложениях, используются различные концепции. К ним относятся:

– диалоговая отладка (отладчики TotalView, Distributed Debugging Tool, mpishim);

– верификация модели программы (MPI-Spin);

– сравнительная отладка (Distributed Virtual Machine);

– автоматический анализ корректности:

- анализ во время работы программы (MARMOT);

- анализ по трассе (Intel Message Checker).

Диалоговая отладка

Параллельные отладчики обеспечивают обычные интерактивные функциональные возможности отладчиков, типа выполнения в пошаговом режиме, установки контрольных точек, оценки переменных и т. д., но дополнительно позволяют пользователю контролировать и воздействовать на группы процессов в отдельном сеансе отладки.

Самым распространенным и технически развитым средством этой категории является коммерческий отладчик TotalView. Поскольку системы данной группы обладают очень схожим функционалом, то в настоящей статье будет рассмотрено только это программное средство.

TotalView (TV) – высокоуровневый отладчик оконного типа, специально спроектированный для многопроцессорных и многоядерных вычислительных систем. TV воспринимает множество реализаций MPI. TV имеет следующие возможности, которые отличают его от последовательных отладчиков:

– процессы и потоки внутри каждого процесса могут быть запущены, остановлены, перезапущены, просмотрены и удалены;

– значения переменных в программе можно изменять во время сеанса отладки;

– распределенная архитектура TV позволяет отлаживать удаленные программы в локальной сети;

– несколько процессов на разных процессорах могут быть сгруппированы, и когда один процесс

достигает точки останова, все сгруппированные процессы будут остановлены.

TV имеет окна и области для отображения отлаживаемых процессов и потоков, групп процессов MPI-программы, исходного кода, активных точек, списков адресов и значений переменных.

Несмотря на то, что по ходу отладки программа может быть исправлена «на лету» и, таким образом, есть возможность проверить различные сценарии исполнения, одним из главных недостатков программных средств этой группы является невозможность обнаружения ошибок, связанных с недетерминированным поведением параллельного приложения.

Верификация модели программы

Верификация модели программы (проверка на модели) – это один из подходов к решению проблемы автоматизации отладки и проверки правильности программ [4]. Появление этой техники произошло совсем недавно (в 80-х годах XX в. в работах Кларка, Эмерсона, Квайла и Сифакиса), но, несмотря на это, она уже нашла широкое применение при разработке параллельных приложений.

Суть метода заключается в следующем. Для заданной анализируемой программы строится ее абстрактная формальная модель. Чаще всего она представляется в виде системы переходов между состояниями. В качестве состояния выступает кортеж значений переменных, фигурирующих в программе, а в качестве перехода между состояниями, соответственно, изменение переменной своего значения. Затем производится спецификация свойств, которыми должна обладать система. Например, желательно показать, что некоторая параллельная программа никогда не попадает в тупик. Чтобы модель была пригодна для верификации, в ней должны проявляться те свойства, анализ которых необходим для установления ее корректности. С другой стороны, она должна быть свободна от частных особенностей, не влияющих на проверяемые свойства, но усложняющих верификацию. Проверяемые свойства или требования выражаются на формальном математическом языке (например, представляются в виде логических формул). После этого верификация программы сводится к проверке выполнимости формализованного требования (спецификации) на абстрактной модели. При этом ведется перебор возможных состояний по разным маршрутам в графе. Если результаты проверки отрицательные, то пользователю предоставляют трассу, содержащую ошибку. Она строится в качестве контрпримера для проверяемого свойства и помогает проектировщику проследить, где возникает ошибка.

Главным плюсом такого подхода является то, что решается одна из основных проблем предыдущего класса инструментальных средств отладки – большая сложность обнаружения потенциальных ошибок, возникающих из-за недетерминированного поведения параллельной программы. Каждый раз при запуске приложения под управлением диалогового отладчика оно проходит лишь один из возможных маршрутов в графе состояния и проблемы, которые могут появиться на других маршрутах, остаются «в тени». Основная же трудность, которую приходится преодолевать в ходе проверки на модели, обусловлена эффектом «комбинаторного взрыва» в пространстве состояний, суть которого заключается в том, что с ростом числа процессов параллельной задачи, участвующих в проверке, число состояний растет, в общем случае, экспоненциально.

К инструментальным средствам, использующим данный подход, относится система MPI-Spin [7] – расширение известного верификатора Spin. Языком, на котором составляется абстрактная модель параллельной программы, является PROMELA (PRocess MEta LAnguage). Программа на PROMELA переводится в программу на C при помощи Spin (Simple Promela INterpreter), а затем компилируется и запускается обычным способом. Вершины дерева состояний перебираются по алгоритму обхода в глубину. Наряду с полным перебором состояний в графе, Spin имеет также режимы выполнения случайно выбранного маршрута и выполнения маршрута, заданного пользователем (при достижении системой каждой вершины, на которой может происходить разветвление, пользователю выдается сообщение и предлагается сделать выбор из нескольких альтернатив). При использовании MPI-Spin язык Promela дополняется конструкциями, упрощающими моделирование MPI-программ.

Таким образом, применение MPI-Spin для отладки MPI-программ имеет следующие недостатки: для каждой проверяемой программы необходимо составить, по сути, одну дополнительную – модель на языке PROMELA; невозможно обнаружить ошибки, связанные с неверным управлением внутренними ресурсами MPI, ошибки несоответствия, многие локальные ошибки (например перекрывание буферов памяти в различных коммуникационных операциях); и, наконец, вышеупомянутая проблема комбинаторного взрыва.

Сравнительная отладка

Основная идея данного подхода заключается в том, чтобы сравнить поведение данной (отлаживаемой) версии программы с поведением эталонной и выдать пользователю информацию о расхождениях. Часто за эталонную версию принимают последовательный вариант данного параллельного приложения. Для этого выбираются определенные точки программы и в них сравниваются значения переменных. Выбор точек может осуществляться как пользователем, так и самой программной системой поддержки параллельной отладки (как в системе DVM [2]). 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. При отладке больших параллельных вычислительных задач размеры полных трасс при этом могут достигать очень больших размеров и не помещаться в имеющуюся оперативную память. В связи с этим разработаны различные методы опре-

деления тех моментов в программе, где целесообразно устанавливать контрольные точки записи состояния в файл трассы [1]. К ним относятся сбор трассы с граничных итераций вложенных друг в друга циклов, сбор трассы только на угловых значениях переменных циклов (например, если в программе есть двойной цикл *(i, j), i=a..b, j=c..d*, то угловыми будут пары *(a, c), (a, d), (b, c) и (b, d)* соответственно). Такой подход базируется на тех соображениях, что при граничных значениях переменных цикла высока вероятность появления таких ошибок, как неинициализированные переменные, выход за границы массива и пр. Естественно, что из-за применения таких стратегий многие расхождения могут остаться незамеченными для инструментального средства.

Интегрированный в систему DVM сравнительный отладчик OPENMP работает по следующей схеме (для анализа MPI-программ может быть применен аналогичный алгоритм). Сначала производится первый прогон параллельной программы со сбором трассы с запущенных потоков, затем автономной утилитой полученная трасса приводится к последовательному виду, т. е. записи трассы располагаются в таком порядке, в котором они были бы выданы при последовательном выполнении. Затем запускается инструментированная последовательная программа, и полученная на предыдущем шаге трасса сравнивается динамически с реальным вычислением. В случае OPENMP-программ для получения последовательной программы из параллельной достаточно всего лишь не принимать во внимание OPENMP-директивы при компиляции.

Метод сравнительной отладки при разработке параллельных приложений применим в том случае, если исследователь обладает уже отлаженным и корректно работающим последовательным вариантом своей программы, что, естественно, бывает далеко не всегда. Кроме того, обозначенный выше факт неконтролируемого разрастания файла полной трассы, необходимого для тотального анализа отлаживаемой программы, является весьма серьезной проблемой при использовании данного метода.

Автоматический анализ корректности

Автоматический анализ корректности MPI-программ подразумевает под собой выявление поведения параллельной программы, способного привести к семантическим ошибкам. Чаще всего анализируются аргументы и последовательность вызовов MPI-функций, сравниваются с информацией о вызовах в других процессах (в случае проверки на глобальную ошибку) или с данными о ранее вызванных MPI-функциях в том же процессе (при проверке на локальную ошибку), а затем на основании некоторых алгоритмов делается вывод о существовании реальных или потенциальных ошибок. Для сбора информации об MPI-вызовах применяется профилировочный интерфейс MPI. Этот инструментарий основан на том, что в любой реализации MPI каждая функция стандарта должна иметь возможность быть вызванной двумя способами: при помощи приставок MPI_ и PMPI_. Таким образом, становится возможным разработать собственную библиотеку MPI-функций, каждая из которых выглядит следующим образом:

Листинг 2.

```
MPI_Function( arg_1, arg_2, …, arg_n) {
[Some_work]
Result = PMPI_Function(arg_1, arg_2, …, arg_n)
[Some_work]
return Result;
}
```

Такую библиотеку (называемую профилировочной) можно скомпилировать в статическую и компоновать с программой пользователя. При этом важно, чтобы во время трансляции пользовательской программы при перечислении статических библиотек в строке компиляции данная библиотека стояла раньше MPI-библиотек реализации. Тогда каждый MPI-вызов будет перехватываться профилировочной библиотекой, далее будет производиться анализ аргументов функции на возникновение ошибки и, возможно, их запись в некоторую структуру для последующего сопоставления с параметрами вызываемых в будущем MPI-функций, затем функция PMPI_Function будет производить работу, которую ожидает от MPI_Function пользователь, потому как осуществляется реализацией MPI. Наконец, после работы PMPI_Function, можно проанализировать возвращенный ею результат и передать его в качестве выходного значения во внешнюю функцию.

В качестве простого примера можно привести алгоритм проверки на возникновение локальной ошибки управления внутренними ресурсами MPI. Пусть в некотором процессе параллельной программы встретился вызов неблокирующей передачи сообщения MPI_Isend. Процесс, достигнув данной функции по ходу работы, записывает факт использования объекта-запроса в некоторую структуру данных, хранящуюся в памяти, либо в файл трассы на жестком диске. Предположим, что далее этот процесс не вызывал соответствующей функции MPI_Request_free. При вызове завершающей функции MPI-программы (MPI_Finalize) данной ветвью параллельного приложения в первом случае просматривается данная структура и фиксируется факт наличия записи о неосвобожденном запросе, во втором случае в файле трассы фиксируется отсутствие записи о соответствующем вызове функции освобождения объекта-запроса. Информация о найденной ошибке выдается пользователю.

Анализ по трассе

На основании двух подходов к обнаружению ошибок в MPI-программах, изложенных в приведенном примере, автоматический анализ корректности разделяют на анализ по трассе (посмертный анализ) и динамический анализ. При применении посмертного анализа каждый процесс параллельного приложения записывает данные об MPI-вызовах в локальный файл трассы, после работы файлы с каждого процесса собираются в единую трассу, которая на следующем шаге анализируется отдельной утилитой. Конкретное содержание файлов трассы зави-

сит от формата. Обычно записываются имя MPI-функции, время и аргументы вызова.

Данный подход использовался при разработке программной системы Intel Message Checker [5]. Данное средство состояло из трех компонент:

– Trace Collector – подсистема для сбора информации об MPI-вызовах в файле трассы во время выполнения параллельной программы.

– Analyzer Engine – утилита, которая проводит анализ файла трассы. Диагностическую информацию Analyzer Engine записывает назад в файл трассы для третьего компонента Visualizer.

– Visualizer – загружает файл трассы и позволяет пользователю просматривать ошибки с автоматическим перемещением к строкам кода, где они возникли.

Так же, как и большинство инструментальных средств автоматического контроля корректности, компонент Trace Collector использует профилировочный интерфейс MPI для сбора информации об MPI-вызовах. Для каждого вызова он записывает все входные и выходные параметры и некоторую дополнительную информацию, например контрольную сумму буфера сообщения. Вся трассировочная информация собирается в структурированный файл трассы после нормального или аварийного завершения программы.

Analyzer Engine находит реальные и потенциальные ошибки по файлу трассы. Результатом работы данного компонента является файл с информацией об ошибках и предупреждениях с соответствующими ссылками на события, записанные в трассу.

Программная система в состоянии обнаружить следующие ошибки:

– несоответствия функций посылки и приема из-за некорректного указания посылающего или принимающего процесса в вызове коммуникационной функции;

– потенциальные и реальные дедлоки;

– ошибки, вызванные записыванием в буфер посылки перед завершением функции MPI_Isend;

– разный порядок вызовов коллективных и редукционных операций;

– ошибки, вызванные перекрыванием буфера посылки или приема с другой коммуникационной операцией (обнаружение данной ошибки не реализовано для производных типов данных);

– ошибки несоответствия размеров сообщений в операциях посылки и приема;

– несоответствия контрольных сумм посланных и принятых сообщений;

– аварийное завершение работы во время вызова MPI-функции;

– несоответствия типов передаваемых и принимаемых данных.

Компонент Visualizer представляет собой графическое приложение, имеющее несколько окон: информационное окно, содержащее перечень найденных в программе ошибок; область исходного кода, где по клику на строке в информационном окне подсвечивается строка исходного кода, где произошла ошибка; окно для отображения трассы в графическом виде.

В настоящее время корпорация Intel отказалась от дальнейшей разработки данного программного средства, и сейчас многие идеи, заложенные в нем, реализованы в системе Intel Trace Analyzer and Collector, которое будет рассмотрено ниже. Один из недостатков инструментальных средств посмертного анализа описан выше при обсуждении систем сравнительной отладки – неконтролируемое разрастание файла трассы при работе с параллельной программой, ветви которой интенсивно обмениваются сообщениями. Другой слабой стороной является непозволительная продолжительность работы компонента Analyzer Engine на больших трассах – данная утилита не была распараллелена.

Анализ во время исполнения

Отличие данного подхода от предыдущего заключается в том, что анализ вызовов MPI-функций производится не по собранной трассе после завершения параллельной программы, а динамически по ходу работы. Здесь опять существует несколько вариантов относительно того, какой процесс (или какие процессы) проводит данный анализ.

В случае системы MARMOT [6] – разработке Штуттгартского центра высокопроизводительных вычислений, существует выделенный сервер отладки, который собирает данные об MPI-вызовах от процессов на рабочих узлах кластера. Архитектуру MARMOT можно представить следующим образом:

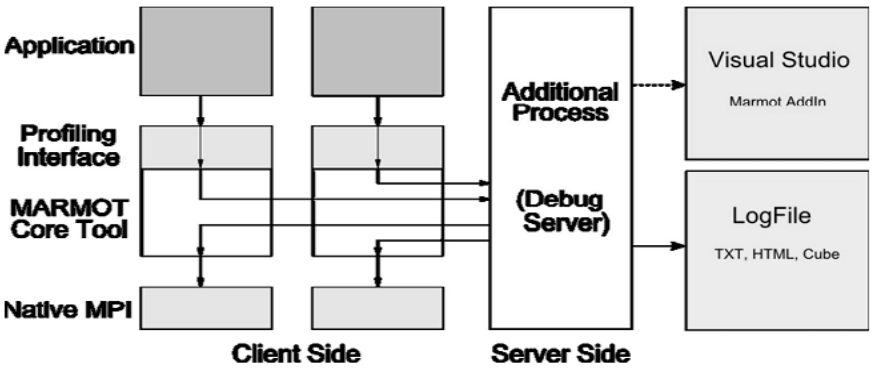

Рис. 1. ***Архитектура системы MARMOT***

По ходу работы процессы вызывают MPI-функции, которые при помощи профилировочного интерфейса перехватываются библиотекой MARMOT. Информация об аргументах функций передается процессу, называемому Debug Server, на выделенном хосте. Данный процесс проводит динамический анализ приложения и по окончании работы системы выдает пользователю найденные ошибки. Опционально Debug Server может записывать трассу в доступном для анализа пользователем виде: TXT-, HTML-файлов или Cube-формате для последующей визуализации при помощи программного средства Cube. MARMOT для Windows реализован в виде плагина для среды Visual Studio.

Остановимся подробнее на алгоритмах обнаружения семантических ошибок, применяемых в MARMOT.

Дедлоки. В настоящее время обнаружение дедлоков в MARMOT базируется на механизме таймаутов и таким образом отыскиваются все реальные дедлоки. Сервер отладки MARMOT фиксирует время, проведенное процессом в MPI-вызове. Если это время превысило определенный пользователем лимит на всех процессах, то процесс отладки выдает предупреждение о дедлоке. Затем пользователь может просмотреть несколько последних вызовов на каждом узле.

Гонки данных. Ранее в статье было указано, что одной из причин появления гонок данных является использование макросов MPI_ANY_SOURCE и MPI_ANY_TAG в коммуникациях точка-точка. MARMOT регистрирует их присутствие и выдает предупреждение вне зависимости от того, было ли их использование оправданным и могло ли привести к появлению некорректного результата или нет.

Несоответствия. При различии длин или типов аргументов в MPI-функциях система выдает предупреждение пользователю. Но при использовании производных типов данных вполне корректно, например, послать два элемента типа (MPI_INT, MPI_DOUBLE) и принять один (MPI_INT, MPI_DOUBLE, MPI_INT, MPI_DOUBLE). Такие ситуации MARMOT не в состоянии отслеживать и делать правильные выводы.

Управление внутренними ресурсами. MARMOT может сохранять историю корректного создания, использования и удаления всех MPI-ресурсов, таких как коммуникаторы, группы, типы данных и т.д. Для этого данное программное средство ведет свой собственный учет внутренних ресурсов MPI и, таким образом, дублирует управление, выполняемое MPI-реализацией. MARMOT также проверяет корректность использования запросов и других аргументов (тэгов, рангов и т. д.), например повторное использование активных запросов.

Исчерпание памяти и других ресурсов. Неблокирующие функции, такие как MPI_Isend и др., могут завершиться без вызова соответствующих функций ожидания или тестов на окончание. В то же время разработчику необходимо учитывать, что число доступных дескрипторов ограничено (и зависит от реализации). Поэтому объекты запросов всегда следует освобождать, также как и коммуникаторы, типы данных и т.д. MARMOT выдает предупреждение, когда существуют активные и неосвобожденные запросы на момент вызова MPI_Finalize.

Перекрывание областей памяти. Еще одна проблема – повторное использование памяти, которая находится в использовании на текущий момент, например чтение/запись из/в буфер/-а во время незаконченной операции посылки/приема. Поскольку из множества всех операторов в программе пользователя MARMOT обладает информацией только о вызванных MPI-функциях, то он не в состоянии обнаружить эти ошибки, тем не менее, когда некоторая область памяти, уже переданная в качестве аргумента одной MPI-функции, передается другой, то выдается ошибка.

Как видно, система MARMOT применяет довольно грубые и неточные алгоритмы для поиска самых опасных и трудно-обнаруживаемых вручную ошибок – дедлоков и гонок данных. Также нельзя считать удовлетворительными методы обнаружения ошибок несоответствия. Кроме того, при использовании данного средства Debug Server может стать узким местом, очень негативно влияющим на производительность параллельной программы во время исполнения.

Заключение

В настоящее время при численном решении большой и ресурсоемкой задачи исследователь вынужден распараллеливать задачу для возможности ее исполнения на вычислительных кластерах. Во время работы такая параллельная программа представляет собой набор процессов, запущенных на разных узлах кластера и обменивающихся данными друг с другом посредством сетевого сегмента. Вследствие этого взаимодействия и использования программных интерфейсов, созданных для поддержки разработки параллельных приложений, возникает множество новых ошибок, крайне сложных для обнаружения без специализированных инструментальных средств. Применительно к программам, созданным при помощи интерфейса MPI, из множества данных ошибок можно выделить такие подклассы, как ошибки неверного использования внутренних объектов MPI-реализации; ошибки синхронизации при организации взаимодействия процессов (гонки данных, дедлоки); несоответствия в операциях посылки/приема.

При разработке систем отладки MPI-программ используются различные подходы. В общем случае каждый класс средств отладки наиболее подходит для обнаружения того или иного типа ошибок. Так, при помощи диалоговых отладчиков удобно обнаруживать ошибки несоответствия, но они практически бесполезны при поиске ошибок, обусловленных недетерминированным поведением параллельной программы. Сильной стороной метода верификации модели программы является, напротив, простота выявления ошибок синхронизации, но, поскольку

система работает не с самой программой, а с моделью, абстрагированной от конкретных типов данных, то невозможным становится обнаружение неверного использования внутренних объектов и типов данных MPI. Автоматический анализ корректности по собранной трассе приводит к большим накладным расходам при работе параллельной программы, поскольку в этом случае ведется запись на жесткий диск файла трассы вызванных MPI-функций, а, как известно, операции по обращению к диску крайне медленны по сравнению с работой процессора с оперативной памятью и тем более процессорным кешем. Для сравнительной отладки требуется корректно работающий эталонный вариант программы и также нельзя не отметить неконтролируемого разрастания файла полной трассы, необходимого для тотального анализа отлаживаемой программы.

Автоматический анализ MPI-программ во время исполнения лишен вышеперечисленных недостатков. Благодаря использованию эвристических алгоритмов позволяет обнаруживать все типы ошибок и требует незначительных накладных расходов – коммуникации между сервером отладки (специально выделенном для этих целей узле кластера) и вычислительными узлами содержат только метаданные пересылок. Имеющееся на сегодняшний день средство этой категории – система MARMOT – обладает серьезным архитектурным недостатком – сервер отладки может стать узким местом при анализе параллельного приложения и, к тому же, алгоритмы, используемые MARMOT, нельзя назвать совершенными. В связи с этим в будущем следует ожидать развития программных систем автоматического анализа параллельных программ во время исполнения и появления новых архитектурных решений при построении средств этой категории.

Литература

1. Средства отладки OPENMP-программ в DVM-системе / В. А. Алексахин, В. О. Баринова, В. А. Бахтин и др. // Тр. Всеросс. науч. конф. «Научный сервис в сети Интернет: технология распределённых вычислений» 22-27 сентября 2008 г., г. Новороссийск – М.: Изд-во МГУ, 2008.

2. Средства отладки MPI-программ в DVM-системе / В. Ф. Алексахин, К. Н. Ефимкин, В. Н. Ильяков и др. // Тр. Всеросс. науч. конф. «Научный сервис в сети Интернет: технология распределённых вычислений» 19-24 сентября 2005 г., г. Новороссийск – М.: Изд-во МГУ, 2005. – С. 113 – 115.

3. Афанасьев К. Е. Многопроцессорные вычислительные системы и параллельное программирование / К. Е. Афанасьев, С. В. Стуколов. - Кемерово: Кузбассвузиздат, 2003. – 233 с.

4. Эдмунд, М. Верификация моделей программ: Model Checking / М. Эдмунд, Кларк, О. Грамберг, Д. Пелед. - М.: издательство Московского центра непрерывного математического образования, 2002. – 416 с.

5. Desouza J. Automated, scalable debugging of MPI programs with Intel Message Checker / J. Desouza, B. Kuhn, B. Supinski // Proceedings of the second international workshop on Software engineering for high performance computing system applications. – St. Louis, Missouri, 2005. - P. 78 – 82.

6. Krammer B. MPI Application Development Using the Analysis Tool MARMOT / B. Krammer, M. Mueller, M. Resch // Lecture Notes in Computer Science. Vol. 3038. – Springer Berlin, 2004. - P. 464 – 471.

7. Siegel S. Verifying Parallel Programs with MPI-Spin / S. Siegel // Proceedings of the 14th European PVM/MPI Users' Group Meeting. – Paris, September/October 2007. - P. 13 – 14.

Научный руководитель – К. Е. Афанасьев – доктор физико-математических наук, профессор, заведующий кафедрой ЮНЕСКО по НИТ КемГУ.

Рецензент – В. П. Потапов – директор института угля и углехимии СО РАН.