УДК 004.4'233

А. Ю. Власенко

Кемеровский государственный университет
ул. Красная, 6, Кемерово, 650043, Россия
E-mail: vlasenko@kemsu.ru

МОДЕЛЬ МАСШТАБИРУЕМОЙ СИСТЕМЫ АВТОМАТИЧЕСКОГО КОНТРОЛЯ КОРРЕКТНОСТИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Отладка параллельных программ, использующих модель распределенной памяти, – сложный и трудоемкий процесс, требующий применения специальных инструментальных средств. В настоящее время существует несколько подходов к построению таких программных систем. В статье приведена классификация ошибок, возникающих в параллельной программе, проведен сравнительный анализ данных подходов и описаны наиболее распространенные на сегодняшний день системы. Описана модель и используемые алгоритмы обнаружения ошибок собственной разработки, относящейся к средствам автоматического контроля корректности.

Ключевые слова: параллельные вычисления, MPI, контроль корректности параллельных программ, логические ошибки в MPI-программах.

Введение. Постановка проблемы

Основным из структурных методов повышения производительности при исполнении больших расчетных задач на ЭВМ является распараллеливание вычислений. При программировании вычислительных кластеров – комплексов самой распространенной в данное время архитектуры, наиболее часто применяются различные реализации стандарта MPI (Message Passing Interface) [1]. Из практики разработки программного обеспечения хорошо известно, что примерно $^2/_3$ времени, затрачиваемого на создание системы, приходится на отладку [2]. Ситуация еще более усложняется тем, что при распараллеливании программы на n независимых процессов, число ошибок в общем случае следует умножать на n. К тому же известно, что стандартные техники тестирования и отладки неэффективны применительно к параллельным программам, потому как из-за взаимодействия параллельно работающих процессов возникает большое множество ошибок, которые не имеют место в последовательных программах. Среди этих ошибок наиболее опасны потенциальные, возникающие из-за большого количества источников недетерминированного поведения таких программ во время работы [3]. Это множество дополняется специфическими ошибками, источником которых является применение интерфейса MPI, обладающего своими типами данных и правилами использования внутренних объектов.

Логические ошибки в параллельных программах

Предлагаемая в данной работе классификация всех возможных ошибок параллельных программ является расширением классификации, представленной в [4].

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;
- *алгоритмические* (логические, семантические).

Алгоритмические 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 неспособна выполнять свои функции;
- *слабые*, которые не являются как таковыми ошибками в смысле данного выше определения, а представляют собой причины неэффективного поведения программы пользователя (например, неполное использование вычислительных ресурсов).

В данной работе наибольшее внимание уделено вопросам обнаружения различных алгоритмических ошибок, поскольку все новые по сравнению с последовательными программами проблемы относятся именно к данной категории. Из множества алгоритмических ошибок 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.

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;
- во время работы MPI-функции данные были изменены основной ветвью процесса;
- программа создает слишком большое число «внутренних объектов» (их лимит ограничен в документе MPI Standard);
- на момент окончания работы в программе существуют не освобожденные запросы;
- для буферизованной посылки недостаточно выделенной памяти.

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

Ошибки синхронизации:

дедлоки
потенциальные;
реальные;
гонки данных.

Ошибки несоответствия.

Дедлоки возникают, когда набор коммуникационных функций, вызванных в различных MPI-процессах, создает условия продолжения работы этих процессов, которые никогда не могут быть удовлетворены некоторыми корректными MPI-реализациями [5]. Коммуникационная операция создает зависимость, когда MPI-стандарт позволяет реализации блокировать процесс до тех пор, пока при работе другого процесса не произойдет некоторое событие. Например, MPI-стандарт позволяет реализовать MPI_Send как синхронную операцию, и процесс, вызвавший эту функцию, может находиться в ожидании до тех пор, пока процесс-получатель не вызовет MPI_Recv. 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, дедлок может произойти или не произойти в зависимости от реализации MPI.

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 *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* могут быть вызваны различными причинами, например, при использовании функции приема 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 Standard является MPI_Barrier. Семантика MPI всегда разрешает гонки данных, поэтому программа не «зависает», но результаты могут оказаться неожиданными для пользователя. Данная группа ошибок относится к множеству проблем, обусловленных недетерминированным поведением параллельной программы.

В MPI Standard определено, что аргументы, переданные в коммуникационную операцию каждым из процессов, должны соответствовать аргументам, переданным другими процессами. В простейшем случае под «соответствующими» аргументами имеют в виду идентичные, но возможны и более сложные случаи. Так, абсолютно корректно послать 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, но никакой точной информации о несоответствии пользователю не выдается. Поэтому поиск *ошибок несоответствия* является важной и сложной задачей.

Методы и программные средства обнаружения алгоритмических ошибок

При построении инструментальных средств, служащих для помощи прикладному программисту в обнаружении семантических ошибок в параллельных приложениях, используются различные концепции.

Диалоговая отладка. Параллельные отладчики обеспечивают обычные функциональные возможности отладчиков, типа выполнения в пошаговом режиме, установки контрольных точек, оценки переменных, и т. д. Но дополнительно позволяют следующее (ниже описаны возможности наиболее распространенного средства TotalView):

- процессы и потоки могут быть запущены, остановлены, перезапущены, просмотрены и удалены;
- значения переменных в программе можно изменять во время сеанса отладки;
- распределенная архитектура параллельного отладчика позволяет отлаживать удаленные программы в локальной сети;
- несколько процессов на разных процессорах могут быть сгруппированы, и когда один процесс достигает точки останова, все процессы группы будут остановлены.

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. Суть этого метода заключается в следующем [2]. 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 – изменение переменной своего значения. Затем производится спецификация свойств, которыми должна обладать система. Например, желательно показать, что некоторая параллельная программа никогда не попадает в тупик. Проверяемые свойства или требования выражаются на формальном математическом языке. 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.

Главным плюсом такого подхода является то, что решается одна из основных проблем диалоговых отладчиков – большая сложность обнаружения потенциальных ошибок, возникающих из-за недетерминизма. 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, число состояний растет в общем случае экспоненциально.

К инструментальным средствам, использующим данный подход, относится система MPI-Spin [3] – расширение верификатора Spin. Абстрактная модель параллельной программы со-

ставляется на языке PROMELA. Программа на PROMELA переводится в программу на C при помощи Spin (Simple Promela INterpreter), а затем компилируется и запускается. При этом перебираются вершины дерева состояний. Наряду с полным перебором состояний Spin имеет также режимы выполнения случайно выбранного маршрута и маршрута, заданного пользователем. При использовании MPI-Spin язык PROMELA дополняется конструкциями, упрощающими моделирование MPI-программ.

Таким образом, MPI-Spin имеет следующие недостатки: для каждой проверяемой программы необходимо составить, по сути, одну дополнительную – модель на языке PROMELA; невозможно обнаружить неверное управление внутренними ресурсами MPI, ошибки несоответствия, многие локальные ошибки; и, наконец, проблема комбинаторного взрыва.

Сравнительная отладка. Основная идея данного подхода заключается в том, чтобы сравнить поведение отлаживаемой версии программы с поведением эталонной и выдать пользователю информацию о расхождениях. Часто за эталонную версию принимают последовательный вариант параллельного приложения. Для этого выбираются определенные точки программы и в них сравниваются значения переменных. Выбор точек может осуществляться как пользователем (отладчик Guard [6]), так и самой программной системой поддержки параллельной отладки (система DVM [7]). 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 память. В связи с этим разработаны различные методы определения тех моментов в программе, где целесообразно устанавливать контрольные точки записи состояния в файл трассы. Естественно, что из-за применения этих методов многие расхождения могут остаться незамеченными для инструментального средства.

Сравнительная отладка применима в том случае, если исследователь обладает уже отлаженным последовательным вариантом своей программы. Кроме того, обозначенный факт неконтролируемого разрастания файла полной трассы, необходимого для тотального анализа программы, является весьма серьезной проблемой при использовании данного подхода.

Автоматический анализ корректности. Данный метод подразумевает собой выявление поведения параллельной программы, приводящего или способного привести к логическим ошибкам. Чаще всего анализируются аргументы и последовательность MPI-вызовов, сравниваются с информацией о вызовах в других процессах, а затем на основании некоторых алгоритмов делается вывод о существовании ошибок. Для сбора информации об MPI-вызовах применяется профилировочный интерфейс MPI. Этот инструмент основан на том, что в MPI-стандарте определена возможность для каждой функции быть вызванной двумя способами: при помощи приставок MPI_ и PMPI_. Таким образом, становится возможным разработать собственную библиотеку, каждая функция которой выглядит так:

```
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 и компоновать с программой пользователя. Тогда каждый MPI-вызов будет перехватываться, далее будет производиться анализ аргументов функции на наличие ошибки, затем функция PMPI_Function будет выполнять работу, которую ожидает пользователь. После работы PMPI_Function можно проанализировать возвращенный ею результат.

Анализ по трассе. 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 одну трассу, которая на следующем шаге анализируется отдельной утилитой.

Данный подход использовался при разработке программной системы Intel Message Checker [8]. 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. 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 Intel Message Checker являлась непозволительная продолжительность работы компонента проведения анализа на больших трассах – данная подсистема не была распараллелена.

Анализ во время исполнения. Рассмотрению программных средств, относящихся к этому направлению, и используемых ими алгоритмов будет уделено большее внимание, поскольку предлагаемая в статье программная система также принадлежит к этому классу. 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 [5] существует выделенный сервер отладки (Debug Server), который собирает данные об MPI-вызовах от процессов на рабочих узлах кластера (рис. 1).

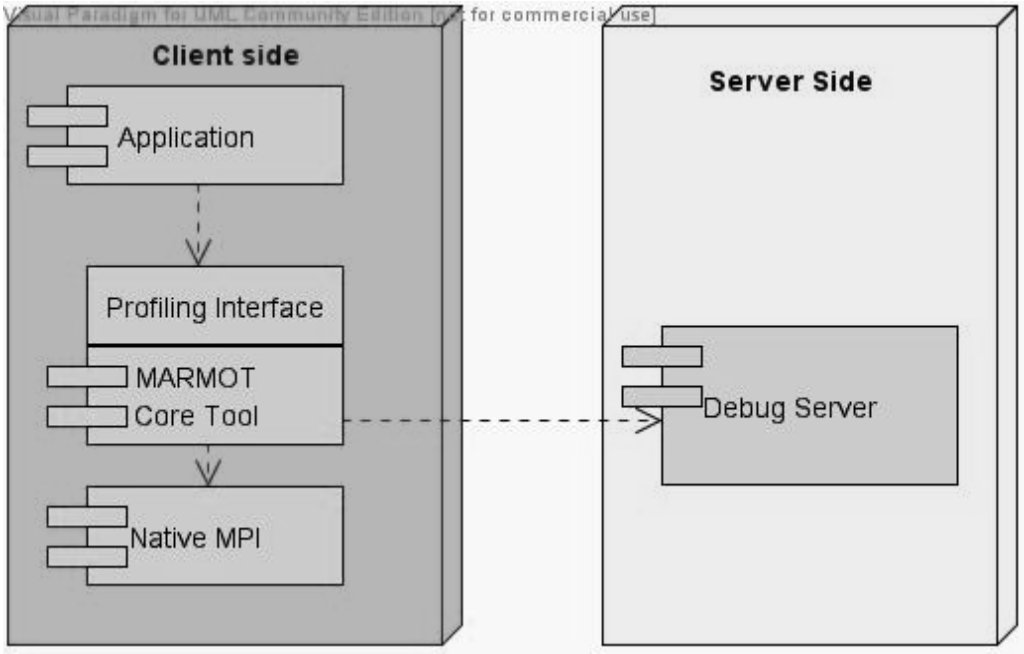

Рис. 1. Архитектура системы MARMOT

По ходу работы процессы вызывают MPI-функции, которые при помощи профилировочного интерфейса перехватываются библиотекой MARMOT. Информация об аргументах функций передается в Debug Server. Данный процесс проводит динамический анализ приложения и по окончанию работы системы выдает пользователю найденные ошибки.

Intel Trace Analyzer and Collector также проводит анализ на существование семантических ошибок в MPI-программах во время исполнения, но имеет кардинальное отличие от MARMOT в архитектуре. Здесь в каждом MPI-процессе на узле кластера работает дополнительный поток, который посредством профилировочного интерфейса MPI считывает параметры вызовов коммуникационных функций. Наличие локальных ошибок этот поток прове-

ряет сразу же, а для обнаружения глобальных осуществляется дополнительная посылка с параметрами вызова одному или нескольким процессам в зависимости от типа проверяемой MPI-операции.

На все ситуации некорректного поведения ITAC реагирует как на предупреждения или ошибки. При этом пользователь может при помощи переменных окружения указывать максимальное число ошибок и предупреждений, после достижения которого параллельная программа будет принудительно завершена.

Рассмотрим алгоритмы, применяемые MARMOT и ITAC для поиска логических ошибок.

Перекрывание областей памяти. Перед началом новой операции ITAC проверяет, не перекрывает ли буфер памяти, используемой данной операцией, память, используемую какой-либо из предыдущих.

Управление внутренними ресурсами. В ITAC каждый раз, когда число активных запросов или типов данных превышает задаваемые пользователем константы, выдаются предупреждения. Также сообщения выдаются, если по окончании работы параллельной программы хотя бы 1 запрос обмена не был освобожден и если запущена операция с активным в текущий момент запросом.

MARMOT ведет свой собственный учет внутренних ресурсов MPI и, таким образом, дублирует управление, выполняемое MPI-реализацией. В дополнение к вышеприведенным MARMOT также обнаруживает такие ошибки, как 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. Поэтому объекты запросов всегда следует освобождать, также как коммуникаторы и типы данных. MARMOT выдает предупреждение, когда существуют активные и неосвобожденные запросы на момент вызова MPI_Finalize.

Гонки данных. Одной из причин возникновения гонок данных является некорректное использование буферизованных посылок – при вызове подряд несколько раз функции буферизованной отправки сообщения предыдущая посылка не успевает покинуть буфер, как ее уже затирает последующая. В связи с этим каждый раз при использовании буферизованной посылки ITAC выдает сообщение, содержащее всю информацию о свободном месте, активных и завершенных сообщениях и соответствующих участках памяти.

MARMOT обнаруживает гонки данных, вызванные использованием макросов MPI_ANY_SOURCE и MPI_ANY_TAG. Для этого данное средство регистрирует их присутствие и выдает предупреждение вне зависимости от того, было ли их употребление оправданным и могло ли привести к появлению некорректного результата или нет.

Дедлоки. В настоящее время обнаружение дедлоков в MARMOT и ITAC базируется на механизме тайм-аутов, и, таким образом, отыскиваются все реальные дедлоки. Для обнаружения потенциальных блокировок, вызванных ситуациями, когда несколько процессов взаимно посылают друг другу сообщения при помощи операции MPI_Send и затем принимают друг от друга, функция MPI_Send в ITAC заменяется на MPI_Ssend.

Ошибки несоответствия. Перед каждой коллективной операцией при помощи MPI_Bcast в системе ITAC всем процессам рассылается тип операции и root-процесс операции, после этого все процессы сравнивают эти параметры со своими значениями. MARMOT проводит сравнение на Debug Server. В случае несоответствия параметров длины или типов аргументов выводится ошибка. Обе системы не могут обнаруживать несоответствия при использовании производных типов данных.

При работе с MARMOT Debug Server может стать узким местом, очень негативно влияющим на производительность параллельной программы во время исполнения. Архитектура ITAC хорошо подходит для обнаружения локальных ошибок. В то же время нельзя не отметить того, что глобальные проверки проводятся довольно грубым образом – например, общие дедлоки ищутся по тайм-ауту, а для обнаружения дедлоков в операциях точка-точка идет принудительная подмена функции, которую вызвал пользователь, на синхронизирующую. Для более тонкого анализа, как то: построения графа вызовов и выявления в нем циклов для обнаружения потенциальных дедлоков или анализа последовательности вызванных в

нескольких процессах функций, требуются большие накладные расходы – множество дополнительных коммуникаций. Методы, применяемые MARMOT для поиска самых опасных и трудно обнаруживаемых вручную ошибок – дедлоков и гонок данных, а также ошибок несоответствия, нельзя считать удовлетворительными.

Построение модели эффективной программной системы автоматического контроля корректности

Архитектура и принципы функционирования системы. Предлагаемый подход для построения системы автоматического контроля корректности сочетает в себе плюсы систем MARMOT и ITAC. На рис. 2 представлена архитектура системы.

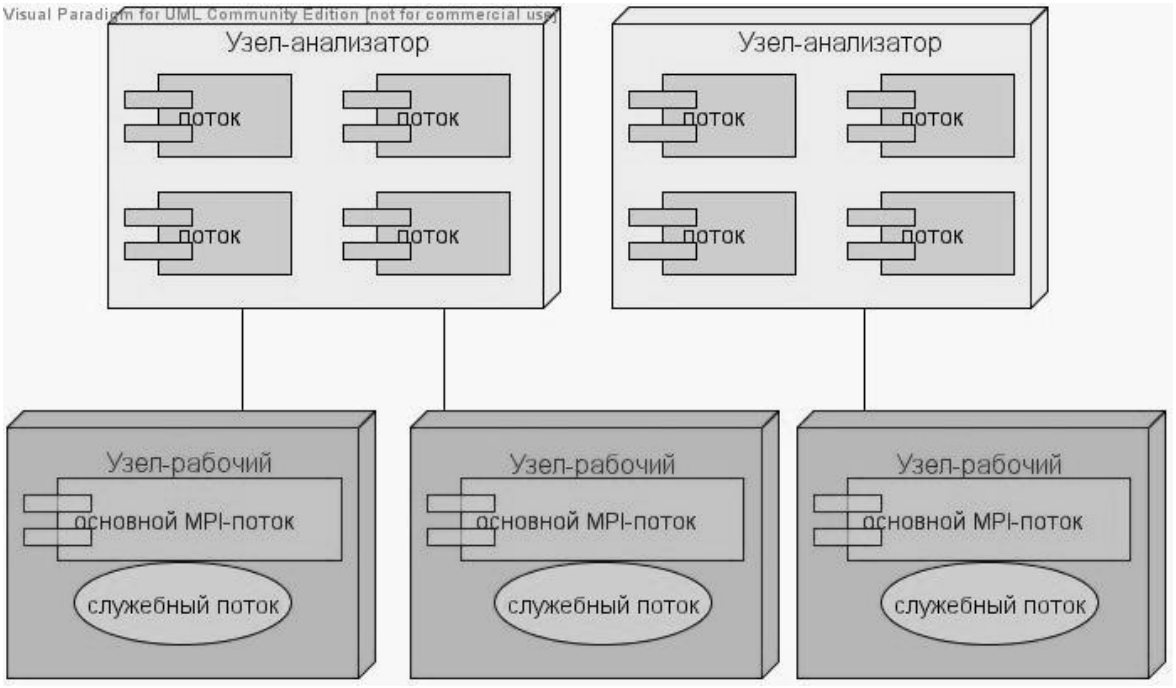

Рис. 2. Компонентная модель системы

Пояснения к обозначениям на рис. 2:

- *узел-рабочий* – вычислительный узел кластера, основной задачей которого является выполнение MPI-процесса программы пользователя;
- *узел-анализатор* – узел кластера, который посредством механизма сокетов принимает посылки от служебных потоков, работающих на узлах-рабочих, содержащие параметры вызовов MPI-функций и проводит анализ на возникновение семантических ошибок;
- *основной MPI-поток* на узле-рабочем – поток, создаваемый используемой MPI-реализацией;
- *служебный поток* на узле-рабочем – поток, создаваемый системой автоматического контроля корректности для анализа вызываемых MPI-функций.

Такая архитектура позволяет отлавливать локальные ошибки на служебном потоке без дополнительных коммуникационных операций, а для анализа на возникновение глобальных ошибок – передавать параметры MPI-функций узлам-анализаторам (число которых определяет пользователь), хранящим данные о вызовах с нескольких узлов. При этом для того, чтобы избежать излишней нагрузки на узел-анализатор в случае обработки программы с интенсивными коммуникациями, можно распределить работу между несколькими такими узлами.

На каждом узле-анализаторе, в свою очередь, может работать несколько потоков, участвующих в анализе.

Для распределения нагрузки на несколько узлов-анализаторов при каждом вызове MPI-функции узел-рабочий передает ее параметры соответствующему узлу-анализатору. Выбор соответствующего анализатора происходит следующим образом. Предположим, что вызвана операция типа точка-точка между процессами с номерами a и b в рамках глобального коммуникатора MPI_COMM_WORLD, общее число процессов составляет N и число узлов-анализаторов равно K. В цикле перебором по всем парам (i, j), где $i < j$, $i = 0..N - 2$, $j = 1..N - 1$ отыскивается пара (a, b) (предварительно (a, b) упорядочивается по возрастанию). На каждой итерации данного цикла паре ставится в соответствие число из диапазона $0..K - 1$. Причем:

$(0, 1) - 0$;
$(0, 2) - 1$;
…
$(0, K) - K - 1$;
$(0, K + 1) - 0$;
…

Узлу-анализатору, имеющему номер, равный поставленному в соответствие паре (a, b) числу, и передаются параметры MPI-функции. Для оценки производительности представленного алгоритма поиска соответствующего посылке точка-точка узла-анализатора проводился вычислительный эксперимент на компьютере с процессором AMD Athlon XP 2500+ (1,83 Ггц) с 1 Гб оперативной памяти. Для $N = 5000$, $a = 5000$ и $b = 4999$ время счета составило менее 1 с. Этот результат можно считать удовлетворительным, поскольку MPI-приложения нечасто используют более нескольких тысяч вычислительных узлов и далеко не для каждой коммуникации требуется полный перебор упорядоченных пар, как в проведенном эксперименте.

При вызове коллективных и некоторых других MPI-функций данные о параметрах также передаются узлу-анализатору, номер которого в этом случае вычисляется как остаток от деления номера процесса, вызвавшего данную операцию, на K.

В основном служебные потоки предназначены для обнаружения локальных ошибок, но в некоторых ситуациях более рациональным представляется выполнить посылку небольшого объема служебному потоку на другом вычислительном узле для проверки на наличие глобальной ошибки. Для посылок такого рода служебные потоки связываются так называемыми «теневыми» коммуникаторами. Каждый раз, когда в основной программе пользователя создается новый коммуникатор, система отладки дублирует его соответствующим теневым. Целесообразность введения этих объектов пояснена ниже.

Используемые структуры данных. Для анализа вызываемых в параллельной программе MPI-функций на вычислительных узлах и узлах-анализаторах заполняются несколько структур данных.

Журналы ошибок, в которые производится запись о найденных по ходу работы MPI-программы ошибках. Журналы ошибок ведутся как на узлах-анализаторах, так и на вычислительных узлах.

Списки отправленных сообщений, хранящиеся на вычислительных узлах и содержащие имя операции, номер процесса-получателя, коммуникатор, тэг сообщения и поле для фиксации факта доставки.

Списки активных «запросов обмена», также хранящиеся на вычислительных узлах и служащие для обнаружения ошибок повторного использования данных объектов.

Списки используемых областей памяти. В них служебными потоками вычислительных узлов записываются начальный и конечный адреса буферов памяти, задействованных в работающих неблокирующих коммуникациях. При вызове функции ожидания для данной пересылки или функции теста на завершение с положительным результатом из данного списка удаляется соответствующая запись.

Флаг занятости буфера представляет собой битовое поле, устанавливаемое в 1 при отправке буферизованной посылки и сбрасываемое в 0 при получении сигнала от принимающего процесса о том, что сообщение доставлено.

Списки структур MPI-функций типа точка-точка, хранящиеся на узлах-анализаторах, в которые ведется запись параметров и некоторой служебной информации обо всех вызываемых MPI-процессами операций типа точка-точка. Ниже приведено определение элемента данного списка.

```
class Nodepointtopoint
{
  public:
  int number;       // номер узла в списке
  double time_begin; // момент времени, в который была вызвана функция
  double time_end; // момент времени, в который функция была завершена
  int pair;         // номер парного узла
  int process;      // номер процесса, пославшего сообщение
  char operation[3]; // сокращенное имя операции
  int count;        // число элементов в посылки
  int datatype;     // тип данных
  int receiver;     // номер процесса-получателя
  int sender;       // номер процесса-отправителя
  int tag;          // тэг сообщения
  bool fin;         // флаг, указывающий, закончилась ли операция
  Nodepointtopoint *next; // указатель на следующий узел в списке
  Nodepointtopoint *prev; // указатель на предыдущий узел в списке
};
```

Обработка операций точка-точка. Процедуру обработки MPI-операций типа точка-точка можно изобразить следующей диаграммой последовательности в стандарте UML (рис. 3). Действующими объектами, представленными на ней, являются узел-отправитель сообщения (Sender), узел-получатель (Receiver) и узел-анализатор (Node-analyzer), соответствующий данной паре. Дополнительными ветвями процессов Sender и Receiver (линиями, начерченными коротким пунктиром) обозначены служебные потоки.

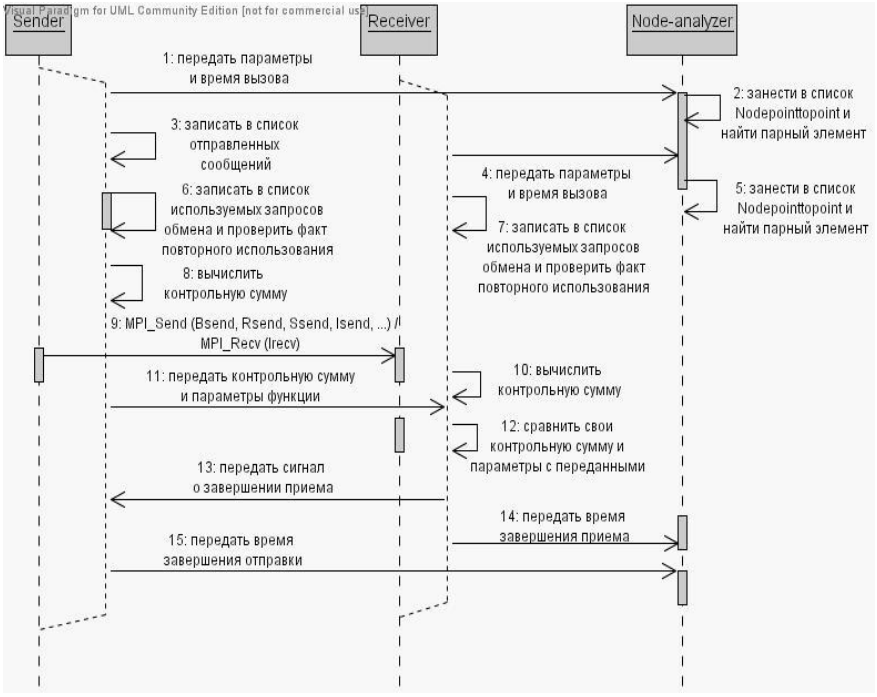

Рис. 3. Диаграмма последовательности

Далее представлен подробный соответствующий приведенной диаграмме алгоритм обработки коммуникационных операций типа точка-точка предлагаемой системой. Необходимо отметить, что обмен дополнительными сообщениями, которые описаны ниже, между служебными потоками производится через «теневой» коммуникатор. Эти дополнительные сообщения проходят прозрачно для пользователя, и необходимость введения нового коммуникатора для них вызвана требованием безопасности. Действительно, если некоторый процесс послал другому сообщение точка-точка, после этого при обработке данного вызова служебный поток посылает процессу-приемнику дополнительные сообщения, а в это время основной поток делает еще одну посылку тому же приемнику в пределах того же коммуникатора, то если приемник вызывает функцию получения сообщения с тэгом MPI_ANY_TAG, вместо основного сообщения может быть принято дополнительное. Следовательно, возникнет ошибка гонки данных.

Алгоритм приведен для произвольной функции типа точка-точка независимо от того, является ли она операцией приема или отправки. В тех шагах, для которых это принципиально, сделаны соответствующие оговорки.

1. Обрабатываемый вызов перехватывается системой при помощи профилировочного интерфейса MPI-реализации.
2. На соответствующий узел-анализатор отправляется сообщение, содержащее время вызова, параметры и имя функции.
3. Приняв сообщение от вычислительного узла, узел-анализатор создает новый элемент в списке Nodepointtopoint и заполняет его поля принятыми параметрами. Поле pair при этом устанавливается в (–1), что означает отсутствие парной коммуникации для данной или то, что пара пока не найдена.
4. Вновь созданному элементу ищется парный. Предположим, что анализируемой функцией является MPI_Send. Тогда парным для данного элемента считается элемент, который удовлетворяет следующим условиям:

- в поле operation записано rcv или irc (т. е. элемент был создан в результате поступления сообщения о вызове функции MPI_Recv или MPI_Irecv);
- имеет значение (–1) в поле pair;
- в поле process имеет значение, равное полю receiver во вновь добавленном элементе (т. е. функция была вызвана процессом-приемником по отношению к анализируемому сообщению);
- поле sender равно номеру процесса, пославшего рассматриваемое сообщение, или константе MPI_ANY_SOURCE, обозначаемой в системе как (–2);
- поле tag идентично такому же полю в рассматриваемом сообщении или имеет значение (–3), свидетельствующее о том, что в функции приема был указан макрос MPI_ANY_TAG.

В поиске парного узла участвуют все потоки процесса на узле-анализаторе, потому как если обрабатывается большая вычислительная задача, то список структур Nodepointtopoint может разрастись до значительных размеров.

5. Если вызвана функция посылки, то служебный поток на узле-отправителе вносит в список отправленных сообщений данные о вызове.
6. В случае неблокирующей функции служебный поток делает проверку на то, задействован ли объект «запрос обмена», фигурирующий в вызове функции, в уже работающих неблокирующих операциях. Для этого просматривается список активных «запросов обмена». Если находится элемент с данными об объекте, идентичном рассматриваемому, то в журнал ошибок вносится запись об ошибке повторного использования «запроса обмена». В противном случае в данный список добавляется новый элемент, соответствующий рассматриваемому объекту.
7. В списке используемых областей памяти ищется пересечение с буферами, над которыми производится операция в настоящий момент. В случае обнаружения пересечения в журнал ошибок записываются данные о функциях и используемых ими общем буфере. Таким образом, происходит обнаружение ошибок перекрывания областей памяти.
8. Если вызванной операцией является функция отправки сообщения, то вычисляется контрольная сумма посылаемого массива.

9. Производится основная операция посылки / приема сообщения. Сразу после завершения ее работы запоминается момент времени окончания.

10. Процесс-приемник вычисляет контрольную сумму по факту получения.

11. Выполняется агрегированная посылка от отправителя получателю, содержащая контрольную сумму и данные о количестве и типе аргументов, а также тэге сообщения в анализируемой операции.

12. Процесс-приемник сравнивает вычисленную контрольную сумму с переданной. В случае неравенства этот факт фиксируется в журнале ошибок. Так в системе происходит обнаружение ошибок изменения передаваемых данных основным потоком MPI-процесса. Для проверки наличия ошибки несоответствия в параметрах операции точка-точка процесс-получатель сравнивает свои параметры вызова функции приема сообщения с принятыми параметрами.

13. Если вызвана функция буферизованной отправки сообщения, то считывается значение флага занятости буфера. Если флаг установлен в 1, то, значит, предыдущая буферизованная посылка еще не принята, и в журнал ошибок заносится запись об этом. Данным методом отыскиваются гонки данных, возникающие при использовании буферизованных посылок.

14. Приняв сообщение, процесс-приемник отсылает сигнал об этом отправителю. Затем процесс-отправитель помечает поле для фиксации факта доставки в списке отправленных сообщений. Если посылка была отправлена при помощи буферизованной функции, то флаг занятости буфера сбрасывается в 0.

15. После окончания работы функции на вычислительном узле на узел-анализатор передается еще одно сообщение, содержащее параметры вызова и время окончания работы. Приняв данное сообщение, узел-анализатор находит соответствующий элемент списка и заполняет поля time_end и fin. Если коммуникационная функция имеет неблокирующий тип, то такое сообщение отсылается при вызове функции ожидания или теста на завершение с положительным результатом. При этом во втором случае в элементе на узле-анализаторе заполняется поле fin, а time_end выставляется в (–1), поскольку определить точное время завершения анализируемой операции невозможно.

16. Выполняется проверка на возникновение дедлока. Как и ранее, предположим, что анализу подвергается коммуникация между процессами *a* и *b*, вызванная узлом *a*. Тогда по истечении заданного интервала времени узел *a* передает сообщение служебному потоку на вычислительном узле *b*, содержащее данные о зависшей операции. Служебный поток, в свою очередь, проверяет, не находится ли основной MPI-поток на *b* в это время в вызове операции точка-точка, парным для которой является некоторый другой процесс *c*. Если это так, то служебному потоку на *c* отправляется сообщение с информацией об обеих коммуникациях. Эта процедура продолжается до тех пор, пока сообщение не будет доставлено процессу, который в данный момент не участвует в операции точка-точка или пока не придет на служебный поток процесса *a*. Тот, обнаружив среди переданных параметров операций свой номер, делает в журнале ошибок запись о создавшемся кольце пересылок. Данный алгоритм в состоянии обнаруживать не только реальные, но и потенциальные дедлоки.

Заключительный анализ MPI-программы. Обнаружение ошибок некоторых типов проводится по окончании работы MPI-процессов – после того, как процессы вызовут функцию MPI_Finalize.

Во-первых, перед завершением служебный поток на каждом узле просматривает свой список отправленных сообщений и делает вывод о существовании непарных коммуникаций в программе пользователя – функций отправки или приема сообщения, для которых не было вызвано парных. Результаты также вносятся в журналы ошибок.

Далее каждый процесс на вычислительных узлах посылает уведомление об этом узлу-анализатору, и, если складывается ситуация, что все процессы за исключением одного послали такое уведомление, то по истечении заданного пользователем временного интервала производится замер времени, которое MPI-процесс провел в вызове последней функции, процессу посылается сигнал завершения, и в журнал ошибок записывается сообщение о зависании процесса во время выполнения данной операции. Эта проверка производится для ситуации, когда ранее примененными аналитическими алгоритмами не удалось обнаружить

причину зависания. В этом случае пользователю предоставляется, по крайней мере, информация об имевшем место факте зависания на некоторой функции.

Также на заключительной стадии происходит поиск неблокирующих операций, для которых не было вызвано соответствующих функций ожидания или успешных тестов на завершение. Просматривается список Nodepointtopoint и обнаруживаются все элементы, которые имеют значение false в поле fin. В журнал ошибок вносятся данные об этих элементах.

После того, как завершены все процессы и все проверки, журналы ошибок с узлов-анализаторов и вычислительных узлов собираются на одном узле-анализаторе, и формируется единый список.

Заключение

При разработке параллельных программ процесс отладки занимает наибольшее время. Это обусловлено тем, что пользователь сталкивается не только со всеми известными ошибками последовательных программ, но и с новыми ошибками, возникающими вследствие взаимодействия одновременно работающих процессов и применения специальных программных средств для организации параллельных вычислений, как, например, какой-либо реализации MPI.

Для поиска ошибок существуют различные методы. Наиболее распространены диалоговая отладка, сравнительная отладка, методы верификации модели программ и автоматический анализ корректности. Последний подход отличается тем, что посредством применения эвристических алгоритмов средство отладки может обнаружить ошибки, обусловленные недетерминизмом, для проведения анализа достаточно одного запуска параллельной программы и нет необходимости создавать программный код эталонной версии, также не нужно составлять модель программы на каком-либо псевдоязыке и ожидать, когда система произведет перебор всех вариантов исполнения параллельного приложения. При построении систем автоматического анализа корректности существует несколько подходов. Первый – анализ по собранной трассе. На данный момент не существует широко распространенных и известных программных систем, использующих данный подход ввиду того, что для полного анализа требуется собирать всю информацию о вызванных MPI-функциях в параллельной программе, что ведет к неконтролируемому разрастанию файла трассы. Другим методом является анализ во время работы приложения пользователя. В настоящее время существует несколько программных систем, использующих данный метод, но каждая из них обладает серьезными архитектурными недостатками, которые не позволяют проводить качественный и эффективный анализ MPI-программ.

В данной работе предложена система автоматического контроля корректности, комбинирующая положительные черты известных программных средств и обладающая потенциально неограниченной масштабируемостью, которая достигается за счет возможности использования для анализа на возникновение ошибок произвольного количества узлов кластера. Построенная модель также претендует на высокую эффективность ввиду того, что основные потоки MPI-приложения работают, в полной мере используя те же вычислительные ресурсы, что и без участия системы, а обнаружением локальных ошибок на вычислительных узлах занимаются дополнительные потоки. Некоторые узлы кластера при этом выделяются исключительно для целей обнаружения ошибок, но так как современные вычислительные комплексы обладают в большинстве своем количеством процессоров, равным нескольким сотням и даже тысячам, то такую избыточность не следует считать существенной слабой стороной системы.

Список литературы

1. *Афанасьев К. Е., Стуколов С. В.* Многопроцессорные вычислительные системы и параллельное программирование. Кемерово: Кузбассвузиздат, 2003. 233 с.

2. *Эдмунд М. К., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: Изд-во московского центра непрерывного математического образования, 2002. 416 с.

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

4. *Корнеев В. Д., Малышкин В. Э.* Параллельное программирование мультикомпьютеров. Новосибирск: издательство НГТУ, 2006. 296 с.

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

6. *Abramson D., Watson G., Dung L.* Guard: A Tool for Migrating Scientific Applications to the .NET Framework. // Proceedings of the International Conference on Computational Science (ICCS 2002), Amsterdam, The Netherlands, April 21st 2002. P. 834–843.

7. *Алексахин В. А., Баринова В. О., Бахтин В. А. и др.* Средства отладки OPENMP-программ в DVM-системе // Тр. Всерос. науч. конф. «Научный сервис в сети Интернет: технология распределенных вычислений» 22–27 сентября 2008 г., г. Новороссийск. М.: Изд-во МГУ, 2008. С. 281–285.

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

Материал поступил в редколлегию 29.07.2009

A. Yu. Vlasenko

MODEL OF SCALABLE SYSTEM OF PARALLEL PROGRAMS AUTOMATIC CORRECTNESS CHECKING

Problem of debugging parallel programs in model of distributed memory is very difficult process that needs special software tools. In current time there are several approaches for construction such program systems. Comparative analysis of these approaches, mistakes classification in parallel programs and the most widely-spread systems to this time is given in this paper. Also paper describes model and algorithms of detection mistakes by Kemerovo University's automatic correctness checking system.

Keywords: parallel computing, Message Passing Interface, correctness cheching of parallel programs, logical errors in MPI-programs.