ОРДЕНА ЛЕНИНА
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша

К.Н. Ефимкин, О.Ф.Жукова, В.Н. Ильяков, В.А. Крюков,
И.П.Островская, О.А.Савицкая

Средства отладки MPI-программ. Анализатор корректности.

Москва
2006

К.Н. Ефимкин, О.Ф.Жукова, В.Н. Ильяков, В.А. Крюков,

И.П.Островская, О.А.Савицкая

Средства отладки MPI-программ. Анализатор корректности.

Аннотация

Описываются основные возможности анализатора корректности, предназначенного для отладки параллельных программ, написанных на Фортране-77, Фортране-90 или Си/Си++ с использованием библиотеки передачи сообщений MPI.

Efimkin K.N., Zhukova O.F., Ilyakov V.N., Krukov V.A.,
Ostrovskaya I.P., Savitskaya O.A.

MPI program debug tools. Analyzer of the correctness.

Abstract

Main possibilities of the analyzer of MPI program correctness are considered. The analyzer is intended for debugging Fortran-77, Fortran-90 and C/C++ parallel programs with the usage of MPI message passing library.

Содержание

Введение

Необходимость создания инструментальных средств, автоматизирующих отладку параллельных программ, уже давно осознана и обоснована во многих исследованиях и публикациях. Сложность отладки параллельной программы определяется тем фактом, что к многочисленным проблемам отладки последовательной программы добавляются новые проблемы – необходимо анализировать *взаимодействие* нескольких процессов, *число процессов* может достигать тысяч и более, поведение процессов может быть *недетерминированным*, языковые средства задания параллельного выполнения имеют *сложную семантику* и часто непривычны для пользователя. В конечном итоге, главной целью автоматизации отладки параллельной программы является автоматизация обнаружения *новых типов ошибок*, появление которых связано с использованием модели параллельного программирования. Ниже под параллельной программой мы будем понимать программу, написанную на языках Фортран или Си/Си++ с использованием библиотеки передачи сообщений MPI.

Авторами данной работы, совместно с Межведомственным суперкомпьютерным центром РАН, Научно-исследовательским вычислительным центром МГУ им М.В.Ломоносова и интернет-сайтом parallel.ru проводилось анкетирование пользователей, занимающихся разработкой параллельных MPI-программ. Результаты анкетирования приведены в Приложении. Ответы пользователей показывают, что ошибки, связанные с параллельным взаимодействием процессов и использованием MPI-функций, являются главной проблемой при отладке, и средства отладки несомненно должны автоматизировать обнаружение таких ошибок. Следует отметить, что 41 пользователь применял при отладке языковые средства (операторы print, write), и всего 11 – специальные средства автоматизации (трассировщики, функциональные отладчики). Это подчеркивает необходимость создания эффективных, удобных в использовании и доступных для пользователей средств автоматизации отладки.

В настоящее время ведутся интенсивные исследования в области создания инструментальных средств отладки параллельных программ.

Известные подходы к отладке параллельных программ можно разделить на три основных направления: *автоматический контроль корректности* выполнения программы, *сравнительная отладка* (сравнение выполнения программы при ее различных запусках), и *диалоговая отладка* в процессе реального выполнения параллельной программы посредством задания контролируемых точек и наблюдения в них интересующих переменных. Автоматический контроль корректности и сравнительная отладка могут осуществляться либо посредством анализа трасс, собранных при выполнении параллельной программы, либо без использования трасс – динамически в процессе реального выполнения параллельной программы.

При диалоговой отладке пользователю в процессе выполнения программы предоставляется возможность в диалоге указывать, какие именно участки программы необходимо исследовать, и проводить анализ состояния программы в этих точках.

Диалоговая отладка является хорошим средством для функциональной отладки параллельных программ. Типичный и известный представитель этого направления – диалоговый отладчик TotalView [1]. Однако многие типы ошибок, в первую очередь связанных с логикой параллельного поведения процессов, при помощи диалогового отладчика обнаружить трудно или невозможно (например, возможность зависания процессов).

Автоматический контроль корректности выполнения программы позволяет существенно упростить нахождение таких ошибок в программе, которые приводят к аномальным ситуациям (недопустимым в используемой модели программирования).

Сравнительная отладка предполагает наличие двух разных программ (или двух версий одной программы), одна из них считается эталонной, а другая – отлаживаемой, и заключается в сравнении соответствующих переменных в соответствующие моменты выполнения этих программ. Сравнительная отладка позволяет, в частности, обнаруживать ошибки, проявляющиеся в нестабильном поведении программы.

Анализ корректности выполнения программы, а также метод сравнительной отладки были использованы в DVM-системе для отладки DVM-программ [2,3]. Метод сравнительной отладки впервые был предложен для последовательных программ в отладчике Guard [4], а для параллельных программ – в DVM-системе. Анализ корректности MPI-программ реализован в системах Umpire [5], Marmot [6], MPI-Check [7], Intel Message Checker [8].

Метод автоматического контроля корректности выполнения программы и метод сравнительной отладки реализованы в разработанном авторами пакете инструментальных средств отладки MPI-программ, состоящем из:

- анализатора корректности MPI-программ, который накапливает трассы, и, используя их, находит в программе ошибочные ситуации и информирует о них пользователя;
- средств сравнительной отладки, позволяющих путем сравнения двух трасс находить нестабильно проявляющиеся ошибки, а также ряд ошибок, не обнаруженных анализатором корректности;
- анализатора эффективности выполнения параллельной программы, позволяющего посредством анализа трассы выдать пользователю информацию, необходимую для повышения эффективности распараллеливания его прикладной программы.

В данной работе описывается **анализатор корректности**, предназначенный для отладки параллельных программ, написанных на языках Фортран или Си/Си++ с использованием библиотеки передачи сообщений MPI.

Назначение, возможности и состав анализатора корректности

Анализатор корректности MPI-программ предназначен для получения состояния задачи в целом и состояния её отдельных процессов, а также для обнаружения ситуаций, классифицируемых в качестве ошибочных.

В общем случае анализатор непосредственно выявляет не ошибки программиста, а ситуации, являющиеся их проявлением (ошибочные ситуации), которые, для краткости, называются ниже ошибками.

Анализатор корректности находит и информирует пользователя о таких аномальных ситуациях в MPI-программах, как авосты и принудительные завершения, реальные и потенциальные дедлоки, некорректность параметров коллективных операций и операций типа “точка-точка”, некорректное использование буферов приема и передачи сообщений, несогласованное использование операций приема и передачи сообщений и многих других ошибках.

Анализатор корректности MPI-программ состоит из двух компонент:

- динамического анализатора корректности
- анализатора корректности трасс

и позволяет

- накапливать трассы в процессе выполнения программы;
- выполнять динамический анализ корректности и находить ошибки в процессе выполнения программы;
- используя накопленные трассы, выполнять анализ корректности после завершения выполнения программы, находить реальные и потенциальные ошибочные ситуации.

Динамический анализатор корректности подключается к отлаживаемой программе на этапе загрузки и заканчивает свою работу по завершению выполнения программы. Динамический анализатор в процессе выполнения программы накапливает трассы для каждого процесса программы. Кроме того, он отслеживает ряд ошибочных ситуаций времени счета. Таким образом, динамический анализатор состоит из двух независимых компонент:

- трассировщика
- блока динамического контроля.

Трассировщик выполняет сбор трасс во время выполнения параллельной программы для каждого из процессов задачи по мере наступления MPI-событий и сохраняет эти трассы в специальных файлах для последующей обработки. Входы и выходы из MPI-функций рассматриваются как автоматически устанавливаемые контрольные точки. Вызов MPI-функции или выход из нее считаются событиями, все события нумеруются, анализ корректности выполняется по таким событиям. При наступлении MPI-события - вызова MPI-функции или возврата из нее - в трассе запоминаются параметры входа и выхода из функции, которые анализируются как во время выполнения, так и после завершения задачи.

Блок динамического контроля во время выполнения программы отслеживает ошибки времени счета, не связанные с межпроцессными обменами, то есть ошибки, возникающие в рамках одного MPI-процесса. Диагностика об обнаруженных ошибках выдается в терминах исходного текста программы.

Анализатор корректности трасс запускается после завершения выполнения программы и использует для своей работы трассы, накопленные трассировщиком. Он анализирует трассы и находит ошибочные ситуации, связанные с межпроцессными обменами. Результатом его работы является протокол работы анализатора, содержащий описание ошибочных ситуаций и привязку этих ситуаций к исходному тексту программы.

Динамический анализатор

Как отмечалось выше, динамический анализатор состоит из двух независимых компонент: трассировщика и блока динамического контроля.

Трассировщик

Трассировщик предназначен для сбора трассы – информации о выполнении MPI-функций в программе пользователя – с целью последующего её анализа.

Накопление трассы выполняется по событиям. Событиями является, как уже говорилось, входы в MPI-функции, либо выходы из них. Все события трассировки нумеруются.

Трассировка содержит информацию о параметрах обращений к MPI-функциям и выходов из них. Для каждого обращения к MPI-функции возможен вывод соответствующего фрагмента исходного текста программы.

Кроме информации об обращениях к MPI-функциям и параметрах этих обращений, трассировщик собирает информацию о созданных программой коммуникаторах, типах данных, стеках адресов возврата из процедур и т. д.

Трассировщик, кроме трассировки обращений к MPI-функциям, позволяет трассировать передаваемае и принимаемые массивы и любые другие данные пользователя, возможно не имеющие никакого отношения к передаче сообщений.

Трасса накапливается в буферах оперативной памяти трассировщика и записывается в файлы по мере заполнения этих буферов.

При завершении задачи (нормальному или аварийному) трассировщик завершает запись информации на диск. В текущей директории появятся следующие файлы:

- Файлы с трассировкой обращений к MPI-функциям. Число этих файлов равно числу MPI-процессов программы.
- Файл с общими описаниями. Данный файл содержит общие для всех процессов описания коммуникаторов, описания ссылок на исходные тексты

программы пользователя, значения констант и переменных MPI и другую полезную информацию.

Пользователю предоставляется возможность управлять работой трассировщика при помощи параметров: он может увеличивать или сокращать размер трасс, управлять длиной стека адресов возврата из процедур, устанавливать режим сокращенной трассировки и т.д.

Блок динамического контроля.

Блок динамического контроля во время выполнения программы отслеживает ошибки времени счета, возникающие в рамках одного MPI-процесса. Диагностика об обнаруженных ошибках выдается в терминах исходного текста программы.

Кроме того, блок динамического контроля может выполнять контроль за изменениями данных пользователя (данных, не являющихся параметрами MPI-функций).

Блок динамического контроля отслеживает следующие виды внутрипроцессных ошибок:

- Ошибки, возникающие в рамках одного процесса при выполнения MPI-функций (например, затирание буфера при выполнении неблокирующей посылки до выдачи соответствующей MPI_Wait).
- Ошибки, обнаруживаемые самим MPI при выполнении MPI-функций. Эти ошибки перехватываются динамическим анализатором, и о них сообщается пользователю.
- Исключительные ситуации, возникающие во время выполнения, не связанные с выполнением MPI-функций (например, деление на 0, запись в защищенную память и т.д.).

Ещё раз отметим, что пользователь получает не только сообщение об обнаружении ошибки, ему указывается также место в исходном тексте программы, соответствующее обнаруженной ошибке.

Анализатор корректности трасс

Анализатор корректности трасс предназначен для нахождения ошибочных ситуаций, связанных с межпроцессными обменами.

Анализатор корректности трасс запускается пользователем после завершения выполнения задачи и использует для своей работы трассы, накопленные трассировщиком. Он преобразует трассы, после чего анализирует их, выявляя ошибочные ситуации.

Ошибки в программе пользователя обнаруживаются просмотром трассы каждого процесса с контролем параметров каждого события и сопоставлением их с параметрами связанных событий других процессов.

Результаты анализа выводятся в специально предназначенный для этого текстовый файл, называемый **протоколом анализатора**.

Информацию об ошибках при обработке трассы (противоречия в трассе, неудачные открытия файлов, нехватка памяти и т. д.) анализатор выводит в поток сообщений об ошибках, связанный с файлом текущей директории. Файл с сообщениями об ошибках создается в начале работы анализатора и по завершению работы, если он пуст, уничтожается.

Управление функционированием анализатора корректности трасс осуществляется с помощью параметров запуска, задаваемых в специальном файле (файле параметров). С помощью параметров запуска можно:

1. Влиять на схему обнаружения ошибок в программе пользователя, их состав и количество;
2. Управлять реакцией анализатора на нештатные ситуации (некорректности в трассировке, неудачные открытия файлов и другие ошибки обработки);
3. Определять местоположение исходных текстов программы пользователя;
4. Управлять выводом информации в протокол анализатора.

Перечислим основные этапы работы анализатора корректности трасс (в скобках указаны названия этих этапов в протоколе анализатора):

1. Инициализация анализатора, чтение и начальное преобразование трассы (Analyzer initialization and trace reading);
2. Приведение трассы к виду, пригодному для дальнейшей обработки (Trace reconstruction);
3. Предварительная обработка трассы перед поиском ошибок (Analysis preparation);
4. Спаривание операций типа “точка-точка” и стыковка коллективных операций (Event reference creation);
5. Обнаружение всех ошибок, кроме дедлоков и зависаний (Error detection);
6. Обнаружение потенциальных и реальных дедлоков и зависаний (Deadlock detection);
7. Вывод в протокол состояния задачи и процессов и информации о всех ошибках, кроме дедлоков и зависаний (Task state and error printing);
8. Вывод в протокол информации о найденных дедлоках и зависаниях (Deadlock printing);
9. Чтение результатов анализа, записанных ранее в трассу, и подготовка её к возможной дальнейшей обработке, например, визуализации (Error reading);
10. Вывод трассы в протокол в терминах обращений к MPI-функциям и выходов из них (Trace printing).

Типы ошибок, отслеживаемые анализатором корректности

Как уже говорилось, в общем случае анализатор корректности MPI-программ непосредственно выявляет не ошибки программиста, а ситуации, являющиеся их проявлением (ошибочные ситуации), называемые ниже для краткости ошибками.

Все ошибки по характеру возникновения (и по схеме обнаружения) делятся на локальные (внутрипроцессные) и нелокальные (межпроцессные).

В межпроцессных ошибках выделен класс коллективных ошибок, то есть ошибок, относящихся ко всем процессам, выполняющим коллективную MPI-операцию. В межпроцессных ошибках нельзя, как правило, обоснованно выделить процесс, являющийся “виновником” ошибки.

Среди коллективных ошибок особое место – как по причине возникновения и характеру проявления, так и по методу обнаружения – занимают потенциальные и реальные зависания.

Ниже приводится список ошибочных ситуаций, которые выявляет анализатор корректности MPI-программ (в скобках – краткие названия ситуаций, используемые в протоколе анализатора).

I. Внутрипроцессные ошибки.

1. Авост в каком-либо процессе задачи (**abend**).
2. Принудительное снятие задачи (предполагается, прежде всего, снятие задачи пользователем нажатием клавиш Ctrl+C; **abort**).
3. Обращение к MPI-функции без возврата из неё (**incomplete call**).

Для операций типа “точка-точка”:

4. Незавершённая передача: невозврат из функции старта передачи или неблокирующая передача без последующего Wait или успешного Test (**unfinished send**).
5. Незавершённый приём: невозврат из функции старта приёма или неблокирующий приём без последующего Wait или успешного Test (**unfinished recv**).
6. Неосвобождённый пассивный запрос (**nonfreed request**).
7. Недопустимое пересечение буфера приёма или передачи запускаемой операции с буфером приёма или передачи какой-либо уже запущенной неблокирующей операции (**overlapping**).
8. Пересечение буферов передачи и приёма в функции MPI_Sendrecv (overlapping).
9. Несовпадение контрольной суммы буфера неблокирующей передачи, подсчитанной при её запуске, с контрольной суммой, подсчитанной при завершении передачи (**send checksum**).
10. Освобождение с помощью функции MPI_Request_free запроса незавершённой неблокирующей операции (предупреждение **nonpersistent request free** – для непостоянного запроса и ошибка **wrong request free** – для persistent)
11. Прерывание неблокирующей операции типа “точка-точка” с помощью функции MPI_Cancel (предупреждение **request cancel**).

Для коллективных операций:

12. Недопустимое пересечение буфера передачи или приёма с буфером передачи или приёма ранее запущенной неблокирующей операции (**overlapping**).

13. Пересечение буфера передачи с буфером приёма (или их частей), взаимное пересечение частей буфера передачи, взаимное пересечение частей буфера приёма (**overlapping**).

II. Межпроцессные ошибки.

Для операций типа “точка-точка”:

1. Передача, которой не удалось поставить в соответствие операцию приёма (**nonpaired send**).
2. Приём, которому не удалось поставить в соответствие операцию передачи (**nonpaired recv**).
3. Несовпадение контрольной суммы буфера передачи с контрольной суммой буфера приёма (**recv checksum**).
4. Несовпадение длин посылаемого и принимаемого сообщений (**incorrect send size** – если длина посылаемого сообщения меньше длины буфера приёма, **wrong send size** – если длина посылаемого сообщения больше длины буфера приёма).
5. Взаимно недопустимые (несогласованные) типы данных в операциях передачи и приёма (**wrong data type**). Для производных типов проверяется согласованность соответствующих элементарных типов.

Для коллективных операций:

6. Не полностью запущенная коллективная операция (не все процессы коммуникатора запустили операцию; **incomplete gop**).
7. Незавершённая коллективная операция (все процессы коммуникатора запустили операцию, но не все по какой-либо причине её завершили; **unfinished gop**).
8. Несовпадение длин посылаемого и принимаемого сообщений (**incorrect recv size, wrong recv size**).
9. Неверная контрольная сумма приёма (**recv checksum**).
10. Взаимно недопустимые типы данных на каком-либо принимающем процессе и на соответствующем ему посылающем (**wrong data type**).
11. Несовпадение кодов редукционной операции хотя бы на двух участвующих в ней процессах (**diff reductions**).
12. Задание пользователем разных root в разных процессах-участниках коллективных операций MPI_Bcast, MPI_Gather, MPI_Gatherv, MPI_Scatter, MPI_Scatterv и MPI_Reduce (**wrong root process**).

III. Зависания и дедлоки.

1. Потенциальное зависание (не образующая цикла цепочка потенциально или реально “зависших” процессов, последний из которых по какой-либо причине завершился, а среди остальных имеется хотя бы один потенциально “зависший”; **potential hang-up**).

2. Реальное зависание (цепочка действительно “зависших” процессов, последний из которых завершился; **real hang-up**).
3. Потенциальный дедлок (цикл потенциально или реально “зависших” процессов, среди которых имеется хотя бы один потенциально “зависший”; **potential deadlock**).
4. Реальный дедлок (цикл действительно “зависших” процессов; **real deadlock**).

Потенциальные и реальные зависания программы пользователя находятся специальным алгоритмом анализа трассы. Основная идея этого алгоритма может быть изложена следующим образом.

Для поиска дедлоков и зависаний трасса программы пользователя разбивается на отдельные для каждого процесса потоки событий. Каждый поток характеризуется состоянием (“открыт” или “закрыт”) и числом прочитанных событий (порядковым номером текущего события). Изначально все потоки открыты, а порядковые номера текущих событий равны нулю.

Пусть {Pi} (i=1,…,n; n > 1) – последовательность различных номеров MPI-процессов (Pk≠Pm при k≠m). Пусть также каждый i-й процесс закрыт при обращении к MPI-функции, условия выполнения которой не обеспечены процессом с номером i+1, i < n. Тогда последовательность {Pi} будем называть зависанием, если процесс Pn по какой-либо причине завершился, и дедлоком, если процесс Pn закрыт при обращении к MPI-функции, условия выполнения которой не обеспечены процессом P0. Другими словами, дедлок – это цикл в графе незавершённых межпроцессных взаимодействий, завершение которых стало невозможным в силу образования этого цикла. Таким образом, все тупиковые ситуации разделены на зависания и дедлоки (хотя в расширенном понимании зависания дедлок является его частным случаем).

Если хотя бы один процесс последовательности {Pi} не закрыт, но мог быть закрыт при некоторых условиях взаимодействия программы со средой её выполнения, то зависание или дедлок называются потенциальными (в отличие от реальных зависаний или дедлоков, в действительности осуществившихся при выполнении задачи).

Чтобы пользователю было легче разобраться в обнаруженных ошибках, анализатор стремится выводить минимальное число диагностик в тех случаях, когда по поводу одной и той же ошибки возможны несколько различных сообщений об ошибке. Например, если из-за неверных типов данных длина передаваемого сообщения оказалась больше длины буфера приёма, то анализатор, обнаружив ситуации wrong send size и wrong data type, выведет только диагностику о неверном типе данных на приёмной стороне.

Число ошибок и предупреждений, обнаруживаемых анализатором, ограничено для каждого процесса значением соответствующего параметра запуска. Этим же параметром ограничено число выявляемых коллективных ошибок и предупреждений. Если число ошибок и предупреждений превышает заданное значение, в протокол анализатора будет выведено соответствующее сообщение.

Протокол работы анализатора

Результаты работы анализатора корректности выводятся в специально предназначенный для этого текстовый файл**,** называемый протоколом анализатора.

Возможности протокола пока ограничены стилем “распечатки”, диалог с пользователем, делающий работу с протоколом более удобной, в настоящий момент не реализован. Содержание протокола в каждом конкретном случае зависит от параметров запуска анализатора.

В протокол анализатора выводятся:

- состояние задачи и отдельных процессов;
- информация об обнаруженных ошибках;
- построенная анализатором трасса процессов задачи в терминах обращений к MPI-функциям и возвратов из них вместе с соответствующими параметрами;
- другая полезная информация, содержащаяся в файлах трассы (таблица коммуникаторов, ссылки на точки исходного текста, сведения об аппаратно-программной среде выполнения задачи и т.д.).

Для контроля функционирования анализатора в протокол могут выводиться времена выполнения основных этапов его работы.

Чтобы иметь информацию о том, к каким MPI-функциям обращалась программа, о числе обращений к ним и временах их выполнения, в протокол анализатора может выводиться информация:

- число обращений к каждой MPI-функции, выполненных всеми процессами (целиком программой);
- число обращений каждого процесса к MPI-функциям и времена их выполнения.

В протокол анализатора может также выводиться следующая информация:

- Описания точек в исходных текстах программы пользователя (каждая точка характеризуется именами файлов и номерами строк исходного текста).
- Описания коммуникаторов, созданных программой или предопределённых.
- Характеристики состояния программы и отдельных процессов и параметры обнаруженных ошибок.
- Параметры обращений к MPI-функциям.

Каждая точка исходного текста – это не только указание имени файла и номера строки, но и информация о последовательности вызовов подпрограмм, которая привела к выполнению оператора в указанной строке (стек вызовов подпрограмм). Более формально, это последовательность (F_1, S_1), ... , (F_n, S_n), где: F_i – имя i-го файла, а S_i – номер строки в файле F_i.

Приведённая последовательность понимается так: был выполнен оператор в строке S_n в файле F_n, и этому выполнению соответствует стек вызова подпрограмм (F_1, S_1), ... , (F_{n-1}, S_{n-1}).

С помощью параметров запуска анализатора можно заказать вывод в протокол не всей, а только фрагментов трассировки процессов, а также вывод трассировки данных пользователя.

Рассмотрим теперь более детально протокол работы анализатора корректности. В качестве иллюстрации при описании таблиц протокола будем использовать таблицы из реального протокола, полученного после выполнения тестовой программы demo.f на двух процессах, в которой обнаружено несоответствие типов посылаемых (complex) и принимаемых (integer) параметров.

В программе demo.f процесс 0 посылает данные типа MPI_COMPLEX (строка 32 файла demo.f):

```
call MPI_Send(A,3,MPI_COMPLEX,1,999,comm,ierr)
```

а процесс 1 принимает данных типа MPI_INTEGER (строка 34 файла demo.f):

```
call MPI_Recv(A,3,MPI_INTEGER,0,999,comm,MPI_Status,ierr)
```

Состояние задачи представляется в протоколе в виде последовательности таблиц. Рассмотрим структуру этих таблиц.

1. **Task state.** Головная таблица протокола анализатора (приведена на Рис.1) содержит:

- число процессов задачи (Nproc);
- числа, авостно, принудительно и успешно завершившихся процессов (abend, abort, normal);
- число ошибок и предупреждений в задаче (Nerr, Nwarn);
- число незавершённых операций send и число незавершённых операций receive (NPsend, NPrecv).

Task state
==========
demo

Nproc	abend	abort	normal	unknown	Nerr	Nwarn	NPsend	NPrecv
2	1	0	1	0	2	1	0	0

Рис.1 Головная таблица протокола анализатора

2. **Current Function**. Таблица с именами различных MPI-функций, к которым были последние обращения (приведена на Рис.2) содержит для каждой функции:

- число процессов, обратившихся к ней (Nproc);
- число различных точек в исходных текстах программы, последнее обращение в которых было к этой функции (Nsrc).

Current functions:

N	function	Nproc	Nsrc
1	Call MPI_Recv	1	1
2	ret MPI_Finalize	1	1

Рис.2 Таблица протокола с последними выполнявшимися MPI-функциями

3. **Current source code points (src).** Таблица со ссылками на исходный текст программы для последних выполнявшихся MPI-функций (приведена на Рис.3) содержит для каждой точки:

- номер строки и имя файла (line, file);
- число процессов с данной текущей точкой выполнения (Nproc);
- имя MPI-функции, к которой произошло обращение в данной точке (function).

Current source code points (src):

N	line	file	Nproc	function
1	34	demo.f	1	call MPI_Recv
2	38	demo.f	1	ret MPI_Finalize

Рис.3 Таблица протокола анализатора со ссылками на исходный текст программы для последних выполнявшихся MPI-функций

4. **All errors/warnings**. Таблица обнаруженных ошибок (приведена на Рис.4) содержит для каждого типа ошибки:

- краткое название ошибки (error name);
- код ошибки и важность ошибки (error code): собственно ошибка (err) или предупреждение (warn);
- число найденных ошибок данного типа (Nerr);
- число процессов, в которых произошла ошибка данного типа (Nproc);
- число различных точек в исходных текстах программы, в которых произошла ошибка данного типа (Nsrc).

All errors/warnings:

N	error code		error name	Nerr	Nproc	Nsrc
1	1	err	abend/abort	1	1	1
2	57	warn	possible hang-up	1	2	2
3	60	err	wrong data type	1	1	1

Рис.4 Таблица обнаруженных ошибок

5. **Source code points of all errors/warnings.** Таблица различных точек в исходных текстах программы, в которых произошли ошибки (любые) (приведена на Рис.5). Для каждой такой точки таблица содержит:

- номер строки и имя файла (line, file);
- число процессов, при выполнении которых в данной точке произошли ошибки (Nproc);
- имя MPI-функции, к которой произошло обращение в данной точке (function).

Source code points of all errors/warnings:

N	line	file	Nproc	function
1	32	demo.f	1	MPI_Send
2	34	demo.f	1	MPI_Recv

Рис.5 Таблица различных точек в исходных текстах программы, в которых произошли ошибки

Кроме того, для каждого типа обнаруженной ошибки выводится отдельная таблица, аналогичная таблице 5. Таблицы для ошибок, обнаруженных в тесте demo.f, приведены на Рис.6.

Source code points of abends/aborts:

N	line	file	Nproc	function
1	34	demo.f	1	call MPI_Recv

Source code points of potential deadlocks and hang-ups:

N	line	file	Nproc	function
1	32	demo.f	1	MPI_Send
2	34	demo.f	1	MPI_Recv

Source code points of data type errors:

N	line	file	Nproc	function
1	34	demo.f	1	MPI_Recv

Рис.6 Таблицы различных точек в исходных текстах программы, в которых произошли ошибки, для каждого типа ошибки

Состояние каждого процесса представлено в протоколе отдельной таблицей (приведена на Рис.7), которая для каждого процесса содержит:

1. номер процесса (Proc);
2. тип завершения процесса (нормальное, авостное или принудительное завершение) (Term);
3. число ошибок (Nerr) и число предупреждений (Nwarn);

4. число операций send (Nsend), число операций receive (Nrecv) и число коллективных операций (Ngop);
5. число незавершённых операций send (Npsend) и число незавершённых операций receive (Nprecv);
6. имя текущей функции MPI;
7. номер строки и имя файла точки завершения процесса;
8. число прохождений через точку завершения процесса (кратность точки завершения);
9. фрагмент исходного текста программы, соответствующий точке завершения;
10. стек обращений к функциям для точки завершения (если длина стека в этой точке больше 1).

State of processes

Proc	Term	Nerr	Nwarn	NPrecv	NPsend	Nrecv	Nsend	Ngop
0	norm		1				1	
	ret MPI_Finalize (1) src=demo.f (38) call MPI_Finalize(ierr)							

Proc	Term	Nerr	Nwarn	NPrecv	NPsend	Nrecv	Nsend	Ngop
1	abend	2	1			1		
	call MPI_Recv (1) src=demo.f (34) call MPI_Recv(A,3,MPI_INTEGER,0,999,comm,MPI_Status,ierr)							

Рис.7 Таблицы состояния процессов программы

Все обнаруженные анализатором ошибки, кроме дедлоков и зависаний, выводятся в протокол отдельно для каждого процесса в порядке их возникновения.

Для каждой найденной ошибки выводится (см. Рис.8):

- краткое название ошибки (Error);
- номер процесса, при выполнении которого произошла ошибка (Proc);
- имя MPI-функции, при обращении к которой произошла ошибка;
- номер строки и имя файла точки, в которой произошла ошибка;
- число прохождений через точку, в которой произошла ошибка (кратность точки);
- фрагмент исходного текста программы, соответствующий точке, в которой произошла ошибка;
- информация о коммуникаторе (если ошибка связана с передачей сообщений);
- стек обращений к функциям для точки, в которой произошла ошибка (если длина стека в этой точке больше 1);
- совокупность фрагментов трассировки, каждый из которых непосредственно или косвенно связан с ошибкой; порядковый номер в

трассировке каждого события при этом заканчивается символом “!”, если событие является “аномальным” (событием, с которым связывается ошибка), или символом “i”, если событие является информационным (т. е. выводимым для лучшего понимания ситуации).

```
                    Errors and warnings
                    ===================

_____________________________________________________________________________
| PROC | N    |      Error       | src  | tag/ind   | size       | rcnt=3       |
|______|______|__________________|______|___________|____________|______________|
|    1 | 1    | wrong data type  | 0    | 999       | 12         | INTEGER      |
|      |      |                  |      |           | 24         | scnt=3       |
|      |      |                  |      |           |            | COMPLEX      |
|      |-------------------------------------------------------------------------|
|      |    send type = COMPLEX             receive type = INTEGER               |
|      |    send dtcode = 138               receive dtcode = 0                   |
|      |-------------------------------------------------------------------------|
|      | MPI_Recv                 (      1) src=demo.f                  (34   ) |
|      |       call MPI_Recv(A,3,MPI_INTEGER,0,999,comm,MPI_Status,ierr )       |
|      | C      call MPI_Probe(1,secarr,comm,MPI_Status,ierr)                    |
|      |-------------------------------------------------------------------------|
|      | MPI_Send                 (      1) src=demo.f                  (32   ) |
|      |       call MPI_Send(A,3,MPI_COMPLEX,1,999,comm,ierr)                    |
|      |       else                                                              |
|      |-------------------------------------------------------------------------|
|      | communicator = 1         proc number = 2         tripl number = 1       |
|      | init=0    last=1    step=1                                              |
|______|_________________________________________________________________________|

6!  call MPI_Recv.  src = demo.f(34) time=00.00.00.006858
call MPI_Recv(A,3,MPI_INTEGER,0,999,comm,MPI_Status,ierr )
buf=0xbffffb80 count=3 dtype=INTEGER size=12
source=0 wsource=0 tag=999 comm=1
InitEvent=6 StartEvent=6 FinishEvent=undef MatchedStartEvent=6

                  Trace fragment of process    0
                  ----------------------------

6!  call MPI_Send.  src = demo.f(32) time=00.00.00.006942
call MPI_Send(A,3,MPI_COMPLEX,1,999,comm,ierr)
buf=0xbffff740 count=3 dtype=COMPLEX size=24
dest=1 wdest=1 tag=999 comm=1 sum=0x0 dtcode=138
InitEvent=6 StartEvent=6 FinishEvent=7 MatchedStartEvent=6


_____________________________________________________________________________
| PROC | N    |      Error       | trace event num = 6                        |
|______|______|__________________|____________________________________________|
|    1 | 2    |      abend       |                                            |
|      |-------------------------------------------------------------------------|
|      | call MPI_Recv                (      1) src=demo.f              (34   ) |
|      |       call MPI_Recv(A,3,MPI_INTEGER,0,999,comm,MPI_Status,ierr )       |
|      | C      call MPI_Probe(1,secarr,comm,MPI_Status,ierr)                    |
|      |-------------------------------------------------------------------------|
|      | MPI error: Message truncated                                            |
|______|_________________________________________________________________________|


6!  call MPI_Recv.  src = demo.f(34) time=00.00.00.006858
call MPI_Recv(A,3,MPI_INTEGER,0,999,comm,MPI_Status,ierr )
buf=0xbffffb80 count=3 dtype=INTEGER size=12
source=0 wsource=0 tag=999 comm=1
InitEvent=6 StartEvent=6 FinishEvent=undef MatchedStartEvent=6
```

Рис.8 Таблицы состояния процессов программы и фрагменты трассировки

Кроме того, для каждого типа ошибки может выводиться специфическая для него информация: длина сообщения в байтах, тег, имя типа данных и т. д.

Завершает вывод ошибок информация о дедлоках и зависаниях (Рис. 9): сначала – информация о реальных дедлоках и зависаниях, затем – о потенциальных.

```
                  Potential deadlocks and hang-ups
                  ================================

Warning   1
===========
0:MPI_Send           1:call MPI_Recv         abend !

Proc=0 event=6  call MPI_Send.  src = demo.f(32) time=00.00.00.006942
call MPI_Send(A,3,MPI_COMPLEX,1,999,comm,ierr)
else
buf=0xbffff740 count=3 dtype=COMPLEX size=24
dest=1 wdest=1 tag=999 comm=1 sum=0x0 dtcode=138
InitEvent=6 StartEvent=6 FinishEvent=7 MatchedStartEvent=6
comm = 1   proc number = 2   tripl number = 1
init=0    last=1    step=1

Proc=1 event=6  call MPI_Recv.  src = demo.f(34) time=00.00.00.006858
call MPI_Recv(A,3,MPI_INTEGER,0,999,comm,MPI_Status,ierr )
C      call MPI_Probe(1,secarr,comm,MPI_Status,ierr)
buf=0xbffffb80 count=3 dtype=INTEGER size=12
source=0 wsource=0 tag=999 comm=1
InitEvent=6 StartEvent=6 FinishEvent=undef MatchedStartEvent=6
```

Рис.9 Информация о реальных и потенциальных дедлоках

Примеры использования

В качестве иллюстрации работы блока динамического контроля приведем примеры реакции на ошибки, возникающие на этапе выполнения программы.

1. В программе test1.c возникла ошибка – деление на 0.

Если анализатор корректности не используется, то возникает стандартная диагностика:

```
[1]  Aborting program !
[1] Aborting program!
p1_29283:  p4_error: : 14
```

Блок динамического контроля позволяет пользователю получить не только сообщение о возникновении ошибки, но и дает информацию о месте в исходном тексте программы, соответствующем обнаруженной ошибке. Диагностика блока динамического контроля, в которой указан номер строки – 87 – и приведен текст этой строки, выглядит следующим образом:

```
 [1] *** Dynamic analyzer err: SIGFPE [8]: Integer divide by zero addr=0x804a8fc
```

```
[1] error src = test1.c(87)
 j=j/i;
```

2. В программе test4.c обнаружено несоответствие операторов MPI_Send и MPI_Receive, возникшее из-за неверного задания получателя сообщения.

Получатель (процесс 1) ждет сообщения от отправителя (процесс 0). Процесс 0 “ошибочно” посылает сообщение другому процессу (NDEST=2). Так как указан несуществующий получатель сообщения, программа завершается.

При запуске программы без анализатора корректности, пользователь увидит следующую стандартную диагностику

```
0 - MPI_SEND : Invalid rank 2
[0]  Aborting program !
p0_4795:  p4_error: : 8262
```

Информация, предоставляемая в той же ситуации блоком динамического контроля, упрощает поиск ошибки, так как указывается номер строки – 83, приведен текст строки, где обнаружена ошибка, и приведены пояснения об отсутствии получателя:

```
[0] *** Dynamic analyzer err: wrong call MPI_Send (incorrect dest 2)
[0] error src = test4.c(83)
        MPI_Send(buf, COUNT, MPI_INT, NDEST, s_tag, comm);

Dynamic analyzer: <test4> task abnormal termination !
Incorrect receiver in Send function:
(0) Send 10 MPI_INT (100, 101, ...) to 2; (1) Recv 10 MPI_INT from 0
```

Более того, если после завершения выполнения программы запустить анализатор корректности трасс, можно получить ещё более подробную информацию об ошибке. Приведем для примера некоторые из таблиц протокола, полученного анализатором корректности трасс для рассматриваемой ситуации.

```
                    Errors and warnings
                    ===================

_____________________________________________________________________________
| PROC | N    |      Error      | trace event num = 12                       |
|______|______|_________________|____________________________________________|
|    0 | 1    |      abend      |                                            |
|      |-------------------------------------------------------------------|
|      | call MPI_Send               (      1) src=test4.c             (83   ) |
|      |         MPI_Send(buf, COUNT, MPI_INT, NDEST, s_tag, comm);          |
|      |  printf("Send is canceled\n");fflush(stdout);                       |
|      |-------------------------------------------------------------------|
|      | incorrect dest 2                                                   |
|______|____________________________________________________________________|

Trace fragment of process    0
------------------------------

12!  call MPI_Send.  src = test4.c(83) time=00.00.02.006
          MPI_Send(buf, COUNT, MPI_INT, NDEST, s_tag, comm);
  buf=0x81793e8 count=10 dtype=INT size=0
  dest=2 wdest=-6789 tag=123 comm=1 sum=0x0 dtcode=0
  InitEvent=12 StartEvent=12 FinishEvent=undef MatchedStartEvent=undef
```

PROC	N	Error	src	tag	size	count=10
1	1	unfinished recv	0	123	40	INT
	MPI_Recv (1) src=test4.c (87)					
	MPI_Recv(buf, COUNT, MPI_INT, 0, r_tag, comm, &r_status);					
	printf("%d: Recv from 0 (%d, %d, ...)\n", rank, buf[0],buf[1]);					
	communicator = 1 proc number = 2 tripl number = 1					
	init=0 last=1 step=1					

```
Trace fragment of process    1
------------------------------

12!  call MPI_Recv.  src = test4.c(87) time=00.00.02.006
   MPI_Recv(buf, COUNT, MPI_INT, 0, r_tag, comm, &r_status);
  buf=0x8178df8 count=10 dtype=INT size=40
  source=0 wsource=0 tag=123 comm=1
  InitEvent=12 StartEvent=12 FinishEvent=undef MatchedStartEvent=undef
```

Приведем также примеры использования анализатора корректности трасс в ситуациях, когда обнаружение ошибок обычными средствами может потребовать значительных усилий.

1. В программе test5.c – скрытый реальный дедлок, так как выданы встречные операции MPI_Send.

Два процесса выдают друг другу сначала встречные MPI_Send, а затем MPI_Recv. Посылаются сообщения типа MPI_INT длиной COUNT*STEP_SIZE. После завершения передачи длина передаваемого массива увеличивается на STEP_SIZE, и операции повторяются.

Программа выполняется до “зависа”, который возникает при переполнении временного системного MPI-буфера (обычно при попытке обменяться сообщениями длиной больше, чем 32K).

Ниже приведены некоторые таблицы из протокола анализатора, из которых видно, в какой программе, в какой строке, при выполнении какой MPI-функции обнаружены ошибочные ситуации.

Общее состояние задачи – задача снята принудительно (abort), анализатор обнаружил 7 ошибок и сделал 22 предупреждения:

```
Task state
 ==========
 test5
```

Nproc	abend	abort	normal	unknown	Nerr	Nwarn	NPsend	NPrecv
2	0	2	0	0	7	22	4	0

Общий список ошибок и предупреждений:

```
All errors/warnings:
```

N	error code	error name	Nerr	Nproc	Nsrc
1	56 warn	possible deadlock	22	2	2
2	1 err	abend/abort	2	2	2
3	44 err	unfinished send	2	2	2
4	46 err	nonpaired send	2	2	2
5	58 err	real deadlock	1	2	2

Описание ошибок в процессе 0:

```
                    Errors and warnings
                    ===================
```

PROC	N	Error	dest	tag	size	count=32768
0	1	unfinished send	1	123	131072	INT
	MPI_Send (23) src=test5.c (84)					
	MPI_Send(buf, STEP_SIZE*cnt, MPI_INT, 1, s_tag, comm);					
	MPI_Recv(buf, STEP_SIZE*cnt, MPI_INT, 1, r_tag, comm, &r_status);					
	communicator = 1 proc number = 2 tripl number = 1					
	init=0 last=1 step=1					

PROC	N	Error	dest	tag	size	count=32768
0	2	nonpaired send	1	123	131072	INT
	MPI_Send (23) src=test5.c (84)					
	MPI_Send(buf, STEP_SIZE*cnt, MPI_INT, 1, s_tag, comm);					
	MPI_Recv(buf, STEP_SIZE*cnt, MPI_INT, 1, r_tag, comm, &r_status);					
	communicator = 1 proc number = 2 tripl number = 1					
	init=0 last=1 step=1					

PROC	N	Error	trace event num = 98
0	3	abort	
	call MPI_Send (23) src=test5.c (84)		
	MPI_Send(buf, STEP_SIZE*cnt, MPI_INT, 1, s_tag, comm);		
	MPI_Recv(buf, STEP_SIZE*cnt, MPI_INT, 1, r_tag, comm, &r_status);		
	SIGINT [2]: Interrupt		

Описание ошибок в процессе 1:

PROC	N	Error	dest	tag	size	count=32768
1	1	unfinished send	0	123	131072	INT
	MPI_Send (23) src=test5.c (87)					
	MPI_Send(buf, STEP_SIZE*cnt, MPI_INT, 0, s_tag, comm);					
	MPI_Recv(buf, STEP_SIZE*cnt, MPI_INT, 0, r_tag, comm, &r_status);					
	communicator = 1 proc number = 2 tripl number = 1					
	init=0 last=1 step=1					

```
_____________________________________________________________________________
| PROC | N    |      Error      | dest | tag       | size      | count=32768   |
|______|______|_________________|______|___________|___________|_______________|
|    1 | 2    | nonpaired send  | 0    | 123       | 131072    | INT           |
|      |-------------------------------------------------------------------------|
|      | MPI_Send                  (     23) src=test5.c                 (87   ) |
|      |  MPI_Send(buf, STEP_SIZE*cnt, MPI_INT, 0, s_tag, comm);                 |
|      |  MPI_Recv(buf, STEP_SIZE*cnt, MPI_INT, 0, r_tag, comm, &r_status);      |
|      |-------------------------------------------------------------------------|
|      | communicator = 1         proc number = 2         tripl number = 1       |
|      | init=0    last=1    step=1                                              |
|______|_________________________________________________________________________|
_____________________________________________________________________________
| PROC | N    |      Error      | trace event num = 98                         |
|______|______|_________________|______________________________________________|
|    1 | 3    |      abort      |                                              |
|      |-------------------------------------------------------------------------|
|      | call MPI_Send               (     23) src=test5.c             (87   ) |
|      |  MPI_Send(buf, STEP_SIZE*cnt, MPI_INT, 0, s_tag, comm);                 |
|      |  MPI_Recv(buf, STEP_SIZE*cnt, MPI_INT, 0, r_tag, comm, &r_status);      |
|      |-------------------------------------------------------------------------|
|      | SIGINT [2]: Interrupt                                                   |
|______|_________________________________________________________________________|
```

Диагностика о дедлоке – встречные операции MPI_SEND в процессе 0 и процессе 1:

```
Real deadlocks and hang-ups
===========================

Error   1
=========
  0:MPI_Send           1:MPI_Send           deadlock !

Proc=0 event=98  call MPI_Send.  src = test5.c(84) time=00.00.02.474
    MPI_Send(buf, STEP_SIZE*cnt, MPI_INT, 1, s_tag, comm);
    MPI_Recv(buf, STEP_SIZE*cnt, MPI_INT, 1, r_tag, comm, &r_status);
   buf=0x81934b8 count=32768 dtype=INT size=131072
   dest=1 wdest=1 tag=123 comm=1 sum=0x5fb200 dtcode=196608
   InitEvent=98 StartEvent=98 FinishEvent=undef MatchedStartEvent=undef
   comm = 1   proc number = 2   tripl number = 1
   init=0    last=1    step=1

Proc=1 event=98  call MPI_Send.  src = test5.c(87) time=00.00.02.477
    MPI_Send(buf, STEP_SIZE*cnt, MPI_INT, 0, s_tag, comm);
    MPI_Recv(buf, STEP_SIZE*cnt, MPI_INT, 0, r_tag, comm, &r_status);
   buf=0x8192e88 count=32768 dtype=INT size=131072
   dest=0 wdest=0 tag=123 comm=1 sum=0x5fe400 dtcode=196608
   InitEvent=98 StartEvent=98 FinishEvent=undef MatchedStartEvent=undef
   comm = 1   proc number = 2   tripl number = 1
   init=0    last=1    step=1
```

2. В программе test12.c скрытая ошибка: запись в буфер посылки до завершения операции MPI_Isend

В тексте программы встречается следующая последовательность операторов:

```
    if (rank==0) {
  ……..
  MPI_Isend(buf, COUNT, MPI_INT, 1, s_tag, comm, &s_req);
        buf[0]=COUNT;buf[COUNT-1]=COUNT+1;
  MPI_Wait(&s_req,&s_status);
    } else if(rank==1) {
        MPI_Recv(buf, COUNT, MPI_INT, 0, r_tag, comm, &r_status);
```

Таким образом, процесс 0 посылает целочисленный массив из COUNT элементов процессу 1. До завершения операции MPI_Isend, то есть до выполнения MPI_Wait, в буфер посылки осуществляется запись. Программа заканчивается без диагностики ошибок, однако работает неверно, так как буфер посылки “затирается”.

Анализатор корректности трасс, работающий с собранными трассами, выполняет сравнение контрольных сумм сообщения до и после выполнения операции MPI_Isend и сообщает о скрытой ошибке в таблицах своего протокола.

Общее состояние задачи – задача завершена нормально, но анализатор обнаружил 1 ошибку:

Task state
==========
test12

Nproc	abend	abort	normal	unknown	Nerr	Nwarn	NPsend	NPrecv
2	0	0	2	0	1	0	0	0

Общий список ошибок и предупреждений – неверная контрольная сумма, вычисленная для буфера операции отправки:

All errors/warnings:

N	error code	error name	Nerr	Nproc	Nsrc
1	49 err	send checksum	1	1	1

Описание ошибок в процессе 0:

Errors and warnings
===================

PROC	N	Error	dest	tag	size	count=10
0	1	send checksum	1	123	40	INT
	addr = 81793e8 start chksum = 2d finish chksum = 39					
	MPI_Wait (1) src=test12.c (81)					
	MPI_Wait(&s_req,&s_status);					
	} else if(rank==1) {					
	MPI_Isend (1) src=test12.c (79)					
	MPI_Isend(buf, COUNT, MPI_INT, 1, s_tag, comm, &s_req);					
	buf[0]=COUNT;buf[COUNT-1]=COUNT+1;					
	communicator = 1 proc number = 2 tripl number = 1					
	init=0 last=1 step=1					

Фрагмент трассы процесса 0 – признак i – “информационный”:

```
Trace fragment of process    0
------------------------------

12i  call MPI_Isend.  src = test12.c(79) time=00.00.01.003
   MPI_Isend(buf, COUNT, MPI_INT, 1, s_tag, comm, &s_req);
  buf=0x81793e8 count=10 dtype=INT size=40
  dest=1 wdest=1 tag=123 comm= 1 sum=0x2d req=1
  InitEvent=12 StartEvent=12 FinishEvent=15 MatchedStartEvent=12
```

Фрагмент трассы процесса 0 – признак ! – “ошибочный”:

```
Trace fragment of process    0
------------------------------

14!  call MPI_Wait.  src = test12.c(81) time=00.00.01.003
   MPI_Wait(&s_req,&s_status);
  req=1 sum=0x39
  InitEvent=12 StartEvent=12 FinishEvent=15 MatchedStartEvent=12
```

3. В программе test7.c – различный порядок выдачи коллективных и редукционных операций.

В тексте программы встречается следующая последовательность операторов:

```
if (rank==0) {
        MPI_Bcast(s_buf, COUNT, MPI_INT, ROOT, comm);
  MPI_Allreduce(s_buf,r_buf,COUNT,MPI_INT,MPI_SUM,comm);
    } else if (rank==1) {
  MPI_Allreduce(s_buf,r_buf,COUNT,MPI_INT,MPI_SUM,comm);
  MPI_Bcast(s_buf, COUNT, MPI_INT, ROOT, comm);
    }
```

В операции MPI_Bcast в качестве рассылаемого массива задается динамический массив типа integer длиной COUNT.

Выполнение программы приводит к следующей ситуации: процесс 0 выполняет последовательность операций MPI_Bcast(); MPI_Allreduce(); процесс 1 “ошибочно” выполняет операции в другом порядке: MPI_Allreduce(); MPI_Bcast().

Если длина рассылаемого массива (COUNT) меньше 32К и в качестве корня (ROOT) указан процесс 0, то программа успешно завершается. В остальных случаях (если в качестве корня указан процесс 1 или длина рассылаемого массива больше 32К) процесс 0 зависает на MPI_Bcast, а процесс 1 - на MPI_Allreduce.

Даже в случае успешного завершения программы, протокол анализатора корректности трасс позволяет увидеть возможный потенциальный дедлок. Фрагменты протокола приведены ниже.

Общее состояние задачи – задача завершена нормально, но анализатор выдает 1 предупреждение:

```
                        Task state
                        ==========
                        test7
```

Nproc	abend	abort	normal	unknown	Nerr	Nwarn	NPsend	NPrecv
2	0	0	2	0	0	1	0	0

Общий список ошибок и предупреждений – обнаружен потенциальный дедлок:

```
All errors/warnings:
```

N	error code	error name	Nerr	Nproc	Nsrc
1	56 warn	possible deadlock	1	2	2

Диагностика о потенциальном дедлоке – возможен неверный порядок выполнения операций MPI_Bcast и MPI_ Allreduce:

```
Potential deadlocks and hang-ups
================================

Warning   1
===========
  0:MPI_Bcast          1:MPI_Allreduce      deadlock !

Proc=0 event=10  call MPI_Bcast.  src = test7.c(84) time=00.00.01.002
           MPI_Bcast(s_buf, COUNT, MPI_INT, ROOT, comm);
           printf("%d: End of Bcast\n", rank);
  buf=0x81793e8 count=10 dtype=INT size=40
  root=0 wroot=0 sum=0x415
  comm=1 OpNum=2 StartEvent=10 FinishEvent=11 GlobalColl=81ec384
  comm = 1   proc number = 2   tripl number = 1
  init=0    last=1    step=1

Proc=1 event=10  call MPI_Allreduce.  src = test7.c(89) time=00.00.01.003
   MPI_Allreduce(s_buf,r_buf,COUNT,MPI_INT,MPI_SUM,comm);
   MPI_Bcast(s_buf, COUNT, MPI_INT, ROOT, comm);
  op=SUM sbuf=0x8178df8 rbuf=0x8178e28
  count=10 dtype=INT arg size=40
  comm=1 OpNum=2 StartEvent=10 FinishEvent=11 GlobalColl=81ec398
  comm = 1   proc number = 2   tripl number = 1
  init=0    last=1    step=1
```

Заключение

В данной работе описаны основные возможности анализатора корректности, предназначенного для отладки параллельных программ, написанных на Фортране-77, Фортране-90 или Си/Си++ с использованием библиотеки передачи сообщений MPI.

Анализатор корректности входит в состав разработанного в ИПМ им. М.В.Келдыша пакета инструментальных средств отладки MPI-программ, который включает также средства сравнительной отладки и анализатор эффективности выполнения параллельной программы.

Анализатор корректности реализован для различных аппаратно-программных сред и может применяться с операционными системами Windows и Linux.

Более подробная информация может быть получена на сайтах http://www.keldysh.ru/dvm/, http://www.kiam.ru/dvm/, http://sp.cmc.msu.ru/dvm/.

Дальнейшее развитие созданных средств отладки MPI-программ связано с разработкой диалоговой системы, обеспечивающей графическое представление результатов анализа и синхронный просмотр диагностической информации об ошибках, трасс и исходных текстов программ, а также с разработкой и реализацией параллельных алгоритмов анализа.

Приложение

В приложении приводятся результаты ответов на некоторые вопросы, заданные при анкетировании пользователей, занимающихся разработкой параллельных MPI-программ. Эти ответы дают представление о типах специфических ошибок в MPI-программах и важности их обнаружения с точки зрения пользователей.

Какие типы ошибок должны обнаруживать средства отладки? Оцените сложность от 0 (менее важно) до 10 (наиболее важно)		**Сумма в баллах**
1.	Неверное число и/или тип параметров MPI-функций	**195**
2.	Несогласование send/receive из-за неверного задания отправителя или получателя сообщения	**244**
3.	Несогласование send/receive из-за ошибки в управлении выполнением программы (например, сообщение не послано из-за того, что один из процессов вышел из выполнения цикла раньше времени)	**248**
4.	Несогласование send/receive из-за ошибочной трактовки порядка выполнения функций (например, не принимается во внимание, что функция MPI_Send может приводить к ожиданию, пока получатель не вызовет соответствующую функцию MPI_Recv)	**233**
5.	Неверные типы данных в send/receive	**179**
6.	Ошибочное задание длины сообщения в send/receive	**216**
7.	Неверный адрес буфера или пересылаемых/принимаемых данных	**206**
8.	Неповторяющиеся/недетерминированные ошибки, зависящие от времени или порядка принимаемых сообщений	**241**
9.	Неповторяющиеся/недетерминированные ошибки из-за записи в буфер отправки или приема, когда неблокирующие send/receive не завершились	**226**
10.	Неповторяющиеся/недетерминированные ошибки из-за чтения из буфера приема до того, как MPI_Irecv завершилась	**221**
11.	Различная последовательность вызова коллективных операций и редукционных операций в разных процессах (MPI_Bcast, MPI_Gather, MPI_Allreduce, …)	**202**
12.	Несовпадение параметров для коллективных операций и редукционных операций в разных процессах	**170**
13.	Неповторяющиеся/недетерминированные ошибки из-за неинициализированных переменных	**209**
14.	Неповторяющиеся/недетерминированные ошибки из-за выхода индекса за границу массива	**238**
15.	Ошибки точности вычисления в редукционных функциях (суммирование) из-за другого порядка вычислений (по сравнению с последовательным)	**207**

16.	Ошибки при создании и использовании коммуникаторов, производных типов и топологий	**181**
17.	Исчерпание памяти из-за порождения коммуникаторов, производных типов и т.п., без их освобождения	**184**
18.	Исчерпание памяти при использовании неблокирующих операций посылки без задания операции wait	**208**
19.	Ошибки, возникающие при изменении числа процессоров для выполнения и связанные с неверной параметризацией по числу процессоров, размерам массивов и т.п.	**159**
20.	Ошибки, возникающие при изменении среды компиляции или выполнения (например, другая параллельная платформа, компилятор или библиотека MPI	**222**
21.	Ошибки, возникающие при изменении объема рабочей загрузки	**158**

Какие типы ошибок в MPI-программах наиболее сложны для обнаружения? Оцените сложность от 0 (просто) до 10 (наиболее сложно)		**Сумма в баллах**
1.	Ошибки, приводящие к зависанию программы и завершению по истечению времени	**289**
2.	Ошибки, приводящие к авосту в программе (деление на нуль, переполнение, обращение к чужой памяти)	**182**
3.	Зацикливание программы	**160**
4.	Ошибки, приводящие к неверным результатам (неверные *значения* расчетных переменных)	**271**
5.	Ошибки, приводящие к авостам в MPI-функциях (плохая диагностика)	**211**
6.	Ошибки, приводящие к недостатку ресурсов (например, если не освобождаются порождаемые ресурсы)	**194**
7.	Ошибки, приводящие к нестабильному/недетерминированному поведению программы (например, два запуска программы дают различные результаты вычислений либо различное завершение)	**334**

Литература

1. http://www.etnus.com
2. В.А.Крюков, Р.В.Удовиченко, “Отладка DVM-программ”, Препринт ИПМ им. М.В.Келдыша РАН №56, 1999.
3. В.Ф. Алексахин, К.Н. Ефимкин, В.Н. Ильяков, В.А. Крюков, М.И. Кулешова, Ю.Л. Сазанов. Средства отладки MPI-программ в DVM-системе. В сб. Труды Всероссийской научной конференции “Научный сервис в сети Интернет”, Новороссийск, 19-24 сентября, 2005 г.
4. http://www.csse.monash/edu/au/%7Edavida/papers/pdsc00.pdf
5. http://www.llnl.gov/CASC/people/vetter/pubs/sc00-umpire-vetter.pdf
6. http://www.hlrs.de/people/mueller/projects/marmot/
7. http://andrew.ait.iastate.edu/HPC/MPI-CHECK.htm
8. Самофалов В.В., Желтов С.Н., Гаврина Е.В., Братанов С.В., Kuhn B.H., DeSouza J., Невидин К.В., Стариков В.С. Анализ корректности MPI-программ. В сб. Труды Всероссийской научной конференции “Научный сервис в сети Интернет”, Новороссийск, 19-24 сентября, 2005 г.