Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Кемеровский Государственный Университет»

На правах рукописи

Власенко Андрей Юрьевич

АВТОМАТИЗИРОВАННЫЙ КОНТРОЛЬ КОРРЕКТНОСТИ MPI-ПРОГРАММ НА ОСНОВЕ ШАБЛОНОВ ОШИБОЧНОГО ПОВЕДЕНИЯ

Специальность 05.13.11 - математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
доктор физико-математических
наук, профессор Афанасьев К.Е.

Кемерово – 2014

Оглавление

Оглавление 2

Введение 5

ГЛАВА 1. СЕМАНТИЧЕСКИЕ ОШИБКИ В ПАРАЛЛЕЛЬНЫХ ПРОГРАММАХ И ПОДХОДЫ К ИХ ОБНАРУЖЕНИЮ..... 17

1.1 Семантические ошибки в параллельных программах 17

1.2 Опрос пользователей 28

1.3 Методы и программные средства обнаружения семантических ошибок 31

1.3.1 Диалоговая отладка 31

1.3.2 Верификация модели программы 34

1.3.3 Сравнительная отладка 37

1.3.4 Автоматизированный анализ корректности 38

1.3.5 Анализ по трассе 40

1.3.6 Анализ во время исполнения 43

ГЛАВА 2. ШАБЛОНЫ ОШИБОЧНОГО ПОВЕДЕНИЯ 52

2.1 Система описания шаблонов ошибочного поведения. 52

2.2 Примеры шаблонов 58

2.2.1 Шаблоны для локальных ошибок 61

2.2.2 Шаблоны для глобальных ошибок 66

ГЛАВА 3. МОДЕЛЬ И АЛГОРИТМЫ ОБНАРУЖЕНИЯ СЕМАНТИЧЕСКИХ ОШИБОК СИСТЕМОЙ АВТОМАТИЗИРОВАННОГО КОНТРОЛЯ КОРРЕКТНОСТИ 74

3.1 Требования к системе автоматизированного контроля корректности. Анализ требований 74

3.2 Моделирование поведения системы автоматизированного контроля корректности 77

3.3 Архитектура и основные структуры данных 81

3.3.1 Конфигурационный файл 81

3.3.2 Утилита запуска 85

3.3.3 Препроцессор 88

3.3.4 MPI-процессы 91

3.3.5 Сервер-анализатор 93

3.4 Процесс выявления соответствия поведения MPI-программы шаблонам ошибочного поведения 105

ГЛАВА 4. ТЕСТИРОВАНИЕ СИСТЕМЫ АВТОМАТИЗИРОВАННОГО КОНТРОЛЯ КОРРЕКТНОСТИ. ПРИМЕРЫ АНАЛИЗА ПРОГРАММ..... 115

4.1 Тестовая база 115

4.2 Используемые тесты 116

4.3 Анализ проведенных тестов 146

4.4 Отладка прикладных MPI-программ 148

Заключение 153

Список литературы 156

Приложение 1. Определение системы описания шаблонов в расширенной форме Бэкуса-Наура 168

Приложение 2. MPI-программа сортировки массива методом чет-нечетной перестановки 169

Приложение 3. MPI-программа поблочного умножения матрицы на вектор 175

Приложение 4. Текст шаблонов для распространенных ошибок MPI-программ 180

Приложение 5. Свидетельство о регистрации разработанного программного средства 191

Приложение 6. Акт о внедрении программного средства 192

Введение

В последние несколько десятков лет при изучении объектов и процессов различной природы все чаще исследователи прибегают к мощному аппарату математического моделирования. Использование численного алгоритма, реализованного на ЭВМ, является на порядки более дешевым методом познания, чем натурный эксперимент. А в некоторых случаях натурный эксперимент невозможен в принципе (например, в задачах геофизики, астрономии и пр.) В модели исследователь стремится отразить как можно большее число факторов, влияющих на протекание изучаемого процесса. Пропорционально числу учитываемых факторов растет количество уравнений и неравенств, что резко увеличивает объем вычислений и, следовательно, время работы программы на ЭВМ. Кроме того, существуют процессы, при моделировании которых в принципе нельзя обойтись без огромного количества вычислений. Это, например такие глобальные проблемы, как: моделирование климата, генная инженерия, проектирование интегральных схем, анализ загрязнения окружающей среды, создание лекарственных препаратов и др.

Таким образом, потребности прикладных исследователей являются одним из главных факторов, способствующих постоянному росту производительности вычислительных систем. Для увеличения быстродействия ЭВМ применяются различные методы, среди которых одну из ключевых ролей играет распределение вычислительной нагрузки по нескольким устройствам. Этот принцип носит название «параллелизм». Параллелизм применяется на многих этапах обработки информации в компьютере [46].

Среди всех компонентов компьютера наибольшая вычислительная нагрузка при исполнении большой расчетной задачи ложится на центральный процессор. Поэтому существует немало различных инженерных решений относительно

распределения работы как между элементами внутри процессора, так и между несколькими процессорами. В подтверждение первого тезиса можно привести тот факт, что все современные процессоры имеют конвейер, что означает постоянную загрузку нескольких устройств, обрабатывающих инструкции, находящиеся в различных стадиях. Кроме того, современные процессоры являются суперскалярными, то есть они способны исполнять более одной инструкции одновременно. Это возможно за счет введения нескольких конвейеров, работающих независимо друг от друга.

Приблизительно с 1970-го по 1985 год производительность процессоров росла преимущественно за счет совершенствования элементной базы и увеличения тактовой частоты. Затем, вплоть до 2000 года, основную роль стали играть архитектурные усовершенствования — конвейеризация, суперскалярность, спекулятивные вычисления, кэширование, увеличение разрядности [30, 41]. К настоящему времени ресурс повышения тактовой частоты практически исчерпан и в апреле 2005 года ведущие компании этого рынка Intel и AMD одновременно приступили к продаже двухъядерных процессоров даже для персональных компьютеров. На сегодняшний день число ядер процессоров увеличивается, но, несмотря на это, для многих численных задач (примеры таких задач приведены выше) производительности современных ПК явно недостаточно. Пределом увеличения производительности за счет множества процессорных ядер в пределах одного сервера являются суперкомпьютеры с общей памятью. Но когда такая система включает более 32 (максимум 64) ядер, то встает проблема некоторых физических ограничений, не только не способствующих росту, но даже сокращающих производительность [30].

Следующим шагом к повышению производительности вычислительной системы является объединение нескольких компьютеров (серверов) в один комплекс. Самой распространенной архитектурой таких комплексов являются вы-

числительные кластеры, где вычислительные узлы и коммуникационные сети между ними построены из серийно выпускаемых промышленностью компонентов. Абсолютное большинство вычислительных систем списка Top500 самых мощных суперкомпьютеров мира построено именно по кластерному принципу.

Для проведения расчетов на вычислительных кластерах необходимо применение инструментов, которые позволяют распределить выполнение задачи между узлами. Для этой цели существуют:

- **Языки программирования**: языки модели PGAS (Partitioned Global Address Space) [55, 67], SNet [62], MC# [63], система Норма[33], Т-система[11], C-DVM (Distributed Virtual Machine) [36], Fortran-DVM[15].

- **Предметно-ориентированные высокоуровневые библиотеки [16]**: MKL [38], PETSc [43, 53].

- **Библиотеки односторонних обменов**: SHMEM [68], GASNET (Global-Address Space Networking) [57], ARMCI (Aggregate Remote Memory Copy Interface) [79];

- **Системы автоматизированного распараллеливания**: CAPTools (Computer Aided Parallelization Tools) [80], Parawise [69], BERT77 [56].

Однако стандартом де-факто при разработке параллельных программ для вычислительных систем с распределенной памятью является MPI (Message Passing Interface) [78, 32]. MPI – это стандарт, использующий модель передачи сообщений между совместно протекающими процессами [44], реализациями которого являются библиотеки функций. Спецификация MPI содержит множество коммуникационных процедур различного характера: посылки типа точка-точка, широковещательные рассылки, редукционные операции, функции для организации процессов в группы и др. На сегодняшний день существуют как свободно-

распространяемые (MPICH, MPI-LAM, OpenMPI, MVAPICH), так и коммерческие реализации данного стандарта, адаптированные под конкретную вычислительную или сетевую архитектуру (IntelMPI, HP-MPI, Microsoft MPI, ScaliMPI). Благодаря тому, что при использовании MPI прикладной исследователь оперирует с памятью на низком уровне, то, создавая MPI-приложение, можно добиться высокой производительности параллельного кода.

К сожалению, параллельное программирование в MPI-стандарте – сложная и трудоемкая задача. Пользователю приходится проектировать свое приложение таким образом, чтобы загрузить работой несколько вычислительных узлов кластера. Причем для того, чтобы программа была эффективной, она должна быть способной работать с любым количеством процессоров, предоставленным в ее пользование в момент запуска и равномерно распределять нагрузку между ними. Наибольшее количество проблем в большинстве случаев возникает на этапе отладки приложения. Даже из практики разработки последовательного программного обеспечения известно, что примерно 2/3 времени, затрачиваемого на создание системы, приходится на отладку [50]. 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 стандартные техники тестирования и отладки неэффективны применительно к параллельным программам, потому как из-за взаимодействия параллельно работающих процессов на разных узлах кластерной системы возникает широкое 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 [85].

Это множество дополняется еще и специфическими ошибками, источником которых является применение интерфейса MPI, обладающего своими типами данных и правилами использования внутренних объектов.

В построении высокопроизводительных вычислительных комплексов за последние годы наметилась тенденция резкого увеличения среднего числа вычислительных узлов на один комплекс. Соответственно увеличивается количество процессов параллельных программ, запускаемых на суперкомпьютерах. В связи с этим все более востребованными становятся автоматизированные средства отладки. Существенной проблемой создания систем данной категории является то, что организовать проверки на все возможные ошибки в пользовательской программе чрезвычайно сложно. Причина этого заключается, во-первых, в том, что MPI-стандарт накладывает огромное количество ограничений на аргументы и последовательность вызовов функций, а несоблюдение любого ограничения является ошибкой. Во-вторых, MPI-стандарт регулярно претерпевает изменения (до настоящего момента были выпущены версии 1.0, 1.1, 1.2, 1.3, 2.0, 2.1, 2.2, 3.0 стандарта), и каждая версия привносит новые функции, типы данных, константы и пр. В то же время пользователю далеко не всегда требуется выполнение системой всех возможных проверок на его программе – зачастую он приблизительно представляет, в каких местах и функциях может возникнуть ошибка, а излишние проверки создают дополнительную нагрузку на вычислительную систему.

Таким образом, актуальной является задача разработки системы автоматизированного контроля корректности, предоставляющей пользователю возможность самому задавать проверки, которые должна произвести система при исполнении программы. При этом подразумевается не выбор подмножества из имеющихся вариантов, а самостоятельное определение пользователем такого поведения программы, которое должно быть выявлено системой отладки. Это

позволит не только обнаруживать ошибки, но и осуществлять пользовательские проверки.

Цель работы

Разработка и реализация эффективного метода отладки параллельных программ, базирующегося на принципах автоматизированного контроля корректности с использованием текстовых шаблонов ошибочного поведения.

Для достижения поставленной цели были сформулированы следующие **задачи:**

1) Выполнить обзор и анализ методов и инструментальных средств отладки параллельных программ.
2) Разработать систему описания шаблонов ошибочного поведения параллельных программ, использующих интерфейс MPI.
3) С помощью разработанной системы описания формализовать наиболее распространенные семантические ошибки в MPI-приложениях.
4) Создать инструментальное средство отладки, выполняющее поиск по заранее составленным текстовым шаблонам ошибочных ситуаций, возникающих при исполнении MPI-приложений.
5) Провести тестирование инструментального средства отладки и сопоставить его с аналогами.

Методы исследований

Для решения поставленных задач использовались методы системного анализа, объектно-ориентированного проектирования, парадигмы структурного и объектно-ориентированного программирования, методы отладки последовательных и параллельных программ, методы функционального проектирования, системного программирования в ОС семейства Windows и тестирования программного обеспечения.

Научная новизна диссертационной работы состоит в следующем:

1) Предложен метод автоматизированного контроля корректности параллельных приложений на основе шаблонов ошибочного поведения, позволяющий легко увеличивать количество выявляемых программным средством отладки ошибочных ситуаций.
2) Предложена система описания шаблонов ошибочного поведения, позволяющая формализовать различные ситуации, возникающие в процессе исполнения MPI-приложений. Данная система описания может быть адаптирована для других технологий параллельного программирования, использующих модель передачи сообщений.
3) Разработана библиотека шаблонов ошибочного поведения, позволяющая проводить проверку MPI-программ на наличие семантических ошибок, в том числе тех, которые не обнаруживаются популярными средствами автоматизированного контроля корректности.

Достоверность результатов проведенных исследований подтверждается адекватностью используемого подхода к отладке параллельных программ, доказанной применением в известных инструментальных средствах; проведением тестирования созданной программной системы на множестве ошибочных MPI-приложений; применением результатов работы при отладке параллельных программ, вошедших в учебное пособие, и в учебном процессе на математическом факультете Кемеровского государственного университета.

Практическая ценность работы заключается в том, что:

1) Разработанные шаблоны ошибочного поведения применимы для анализа MPI-программ на наличие семантических ошибок.
2) Созданное программное средство предоставляет прикладным программистам инструментарий для выявления семантических ошибок в MPI-программах в автоматизированном режиме, что существенно сокращает время разработки параллельных приложений и увеличивает эффективность

использования дорогостоящих высокопроизводительных вычислительных ресурсов.

3) Рекомендации, разработанные для пользователей программного средства, применимы как при описании ошибочного поведения, так и при проведении автоматизированного контроля корректности MPI-программ.

Первая глава данной работы содержит классификацию семантических ошибок в параллельных программах для систем с распределенной памятью. Большинство выделенных в рамках классификации типов ошибок являются независимыми от конкретной технологии программирования и присущи любой организации параллельных вычислений на подобной архитектуре. Некоторые типы ошибок специфичны для программирования в модели SPMD (Single Program Multiple Data) [5] (интерфейс MPI использует именно эту модель). Особое внимание уделено описанию типов ошибок, свойственных MPI-приложениям [6]. Даны результаты социологического опроса MPI-программистов относительно ранжирования семантических ошибок по степени сложности их отыскания и важности обнаружения инструментальными средствами. После систематизации и приведения примеров на несколько типов ошибок в первой главе следует описание применяемых методов отладки. Даны преимущества и недостатки каждого подхода к выявлению ошибок. Наиболее популярные программные системы отладки параллельных приложений классифицированы по приведенным подходам. В конце главы сделан вывод о том, что автоматизированный анализ корректности, по мнению автора, является наиболее перспективным методом обнаружения семантических ошибок. Детально разобраны инструментальные средства, использующие данный подход.

Во второй главе приведены правила системы описания шаблонов ошибочного поведения MPI-программ. Приведена структура файлов шаблонов и подробно описаны все ключевые элементы таких файлов. Показано, что включен-

ных в систему элементов достаточно для описания всех ошибок, приведенных в классификации первой главы. В файле шаблона пользователь может детализировать ошибочную ситуацию, например, указав конкретный номер MPI-процесса, на котором необходимо следить за возникновением ситуации. Более того, при помощи введенной системы можно описать шаблоны для пользовательских проверок на возникновение ситуаций, не являющихся ошибками в полном смысле этого слова. Даны шаблоны для ряда ошибок, фигурирующих в классификации первой главы.

Третья глава посвящена описанию системы отладки собственной разработки [6, 7, 8, 9, 10, 14, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]. Подробно рассмотрена структура и выделены конкурентные преимущества системы по сравнению с альтернативными средствами автоматизированного контроля корректности параллельных программ. Даны назначение и принципы работы компонентов системы. Описано назначение потоков данных, циркулирующих между компонентами. Выводы о наличии в программе пользователя семантических ошибок производятся путем сбора некоторой информации об MPI-процессах и анализа собранных данных по ходу работы. Приведено подробное описание используемых структур данных, которыми оперируют компоненты системы. Особенное внимание уделено описанию алгоритма обработки системой пользовательских шаблонов и обнаружению семантических ошибок в программах пользователей. Разобраны примеры обработки программ, содержащих наиболее типичные ошибки при MPI-программировании, которые, тем не менее, достаточно трудны для обнаружения без использования специализированных систем отладки.

Четвертая глава посвящена описанию проведенных тестов системы на MPI-программах, содержащих семантические ошибки. Был составлен ряд простых программ, каждая из которых проверялась при помощи разработанного инструментального средства и еще двух средств (MARMOT и ITAC), также ис-

пользующих подход автоматизированного контроля корректности. Для каждого теста приведена выходная информация от каждой из систем отладки. Результаты тестов сведены в таблицу и проведен их анализ. Система автоматизированного контроля корректности была применена для отладки параллельных программ, вошедших в учебное пособие «Параллельные вычислительные алгоритмы», разрабатываемое на кафедре ЮНЕСКО по НИТ КемГУ. Процесс обнаружения ошибок в этих программах также описан в четвертой главе.

В приложениях приведено полученное свидетельство о государственной регистрации программы для ЭВМ на разработанное программное средство, акт о внедрении в деятельность центра коллективного пользования «Высокопроизводительные параллельные вычисления» КемГУ, определение системы описания шаблонов в расширенной форме Бэкуса-Наура, текст составленных шаблонов и исходные коды параллельных программ, для отладки которых было применено программное средство.

На защиту выносятся:

1) Метод автоматизированного контроля корректности параллельных программ на основе шаблонов ошибочного поведения.
2) Система описания шаблонов ошибочного поведения MPI-приложений для использования в процессе автоматизированного контроля корректности.
3) Библиотека шаблонов ошибочного поведения для распространенных семантических ошибок в MPI-приложениях.
4) Программное средство отладки параллельных приложений, использующее метод автоматизированного контроля корректности на основе шаблонов ошибочного поведения.

Реализация результатов исследования. Система автоматизированного контроля корректности, являющаяся одним из результатов работы, была внедрена в центре коллективного пользования «Высокопроизводительные параллель-

ные вычисления» КемГУ (http://icp.kemsu.ru/CKP). Система установлена на входящем в состав центра вычислительном кластере, используется преподавателями и студентами математического факультета КемГУ для отладки MPI-программ, была применена при подготовке учебного курса «Параллельные вычислительные алгоритмы», нашедшего отражение в учебном пособии.

Апробация работы.

По мере их получения результаты диссертационного исследования были представлены на следующих конференциях и семинарах: VI Международный научно-практический семинар «Высокопроизводительные параллельные вычисления на кластерных системах» (Санкт-Петербург, 2006); XI Всероссийская научно-практическая конференция «Научное творчество молодёжи» (Анжеро-Судженск, 2007); VI Международная научно-практическая конференция «Информационные технологии и математическое моделирование» (Анжеро-Судженск, 2007); VII Всероссийская научно-практическая конференция «Инновационные недра Кузбасса. IT-технологии - 2008» (Кемерово, 2008); VII Международная научная конференция «Наука и образование» (Белово, 2008); III (XXXV) Международная научно-практическая конференция «Образование, наука, инновации - вклад молодых исследователей» (Кемерово, 2008); XLVI Международная научная студенческая конференция «Студент и научно-технический прогресс» (Новосибирск, 2008); XV региональный смотр-конкурс молодежных IT-проектов «Soft-Парад 2008» (Красноярск, 2008); XV всероссийская научная конференция студентов-физиков и молодых ученых (Кемерово-Томск, 2009); V сибирская конференция по параллельным и высокопроизводительным вычислениям (Томск, 2009); IX международная конференция-семинар «Высокопроизводительные параллельные вычисления на кластерных системах» (Владимир, 2009); XVII Всероссийская научно-методическая конференция «Телематика'2010» (Санкт-Петербург, 2010); Международная суперкомпьютерная

конференция «Научный сервис в сети Интернет: экзафлопсное будущее» (Абрау-Дюрсо, 2011); VI сибирская конференция по параллельным и высокопроизводительным вычислениям (Томск, 2011); международная научная конференция «Параллельные вычислительные технологии - 2012» (Новосибирск, 2012); всероссийская молодежная конференция «Информационно-телекоммуникационные системы и технологии» (ИТСиТ, 2012); международная научная конференция «Информационно-вычислительные технологии и математическое моделирование (ИВТ&ММ)» (Кемерово, 2013); расширенный семинар ССКЦ, НГУ и Центра Компетенции СО РАН - INTEL «Архитектура, системное и прикладное программное обеспечение кластерных суперЭВМ» под руководством профессора Глинского Б.М. (Новосибирск, 2010); научный семинар «Информационные технологии и математическое моделирование» под руководством профессора Афанасьева К. Е. (Кемерово, 2006 – 2012).

Работа содержит 167 страниц основного текста без приложений, 11 рисунков и 4 таблицы. Список литературы состоит из 90 наименований.

ГЛАВА 1. СЕМАНТИЧЕСКИЕ ОШИБКИ В ПАРАЛЛЕЛЬНЫХ ПРОГРАММАХ И ПОДХОДЫ К ИХ ОБНАРУЖЕНИЮ

1.1 Семантические ошибки в параллельных программах

Ошибка, которую можно определить как появление одного или более некорректных результатов на каком-либо из этапов преобразований программы из исходного кода в исполняемые файлы и последующей работы, может появиться из-за неверно составленного кода (программная ошибка) или аппаратных сбоев [24]. Программные ошибки, в свою очередь, делятся на (рисунок 1):

- *синтаксические* – ошибки, обнаруживаемые компилятором на стадии перевода программы в машинный код;
- и *семантические* (также называемые логическими или алгоритмическими).

Семантические ошибки можно разделить на:

- *сильные* – в этом случае программа проходит логическую последовательность состояний, которая приводит к результату, отличному от ожидаемого, или к тому, что программа не способна выполнять свои функции;
- и *слабые,* которые не являются как таковыми ошибками в смысле данного выше определения, а представляют собой причины неэффективного поведения программы пользователя (не полное использование вычислительных ресурсов, большой процент расхода времени на коммуникационные операции в общем времени работы и т.д.)

Рисунок 1. Программные ошибки

В данной работе наибольшее внимание уделено вопросам обнаружения различных семантических ошибок, в связи с тем, что все новые по сравнению с последовательными программами проблемы относятся именно к данной категории. Из множества семантических подробно разобран класс сильных ошибок, потому как поиск слабых – задача оптимизации, а не отладки параллельного приложения. При этом будут рассматриваться только такие ошибки и методы их обнаружения, которые присущи параллельным программам, т.е. таким проблемам, которые не присущи последовательному программированию. Таким образом, к ошибкам, которые останутся за гранью рассмотрения, относятся:

- переполнение буфера;
- использование неинициализированных объектов;
- утечки памяти;
- арифметическое переполнение и деление на ноль;
- недостижимый код;
- и т.д.

Некоторые из описанных ниже семантических ошибок приводятся в работах [42, 66, 73, 76].

Среди сильных семантических ошибок выделяют:

- *локальные* - для их обнаружения каждому процессу не требуется информация от других процессов (рисунок 2);
- *глобальные*, которые включают 2 и более процессов. Для их нахождения нужно анализировать данные из нескольких ветвей параллельного приложения (рисунок 3).

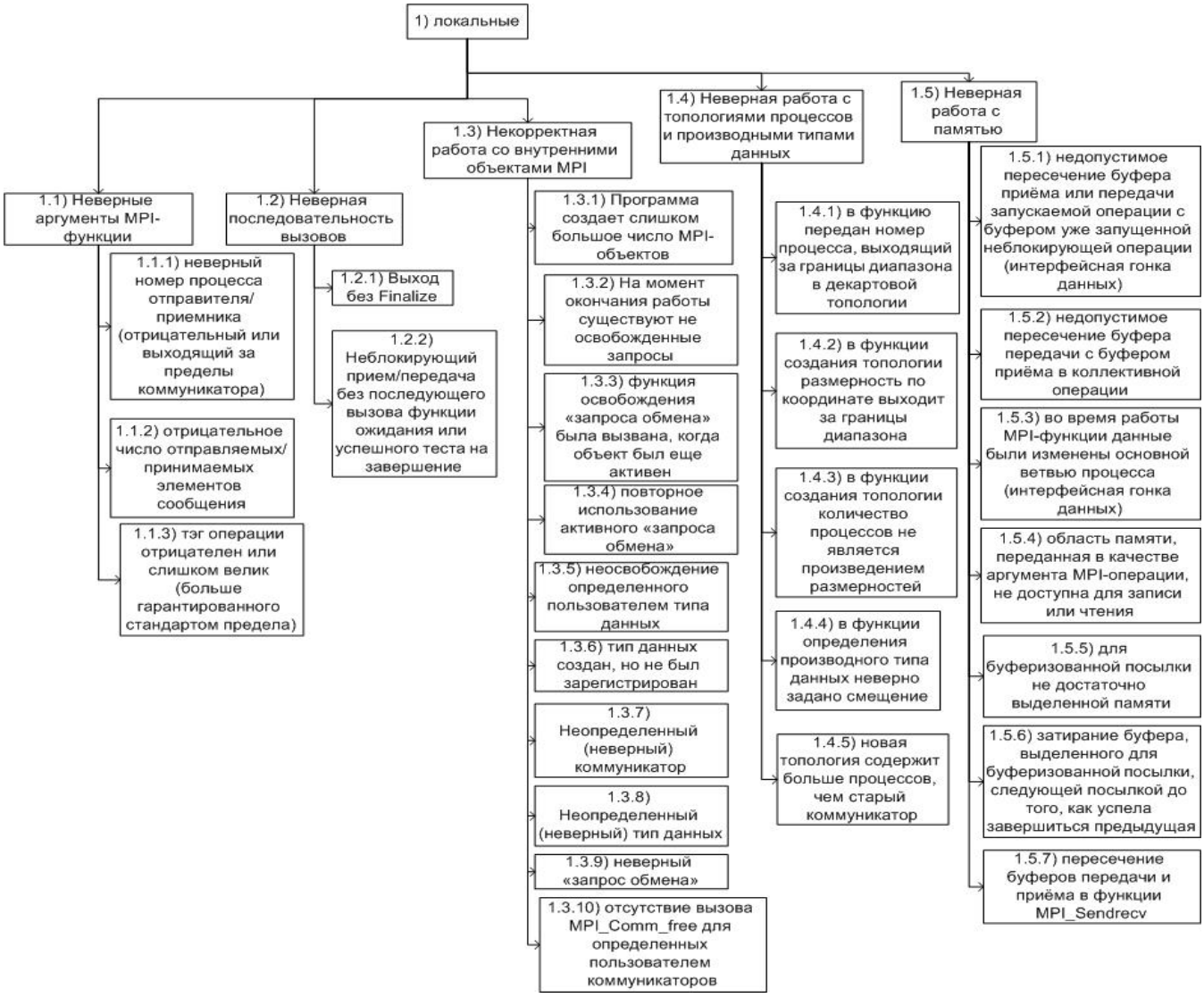

Рисунок 2. Локальные ошибки

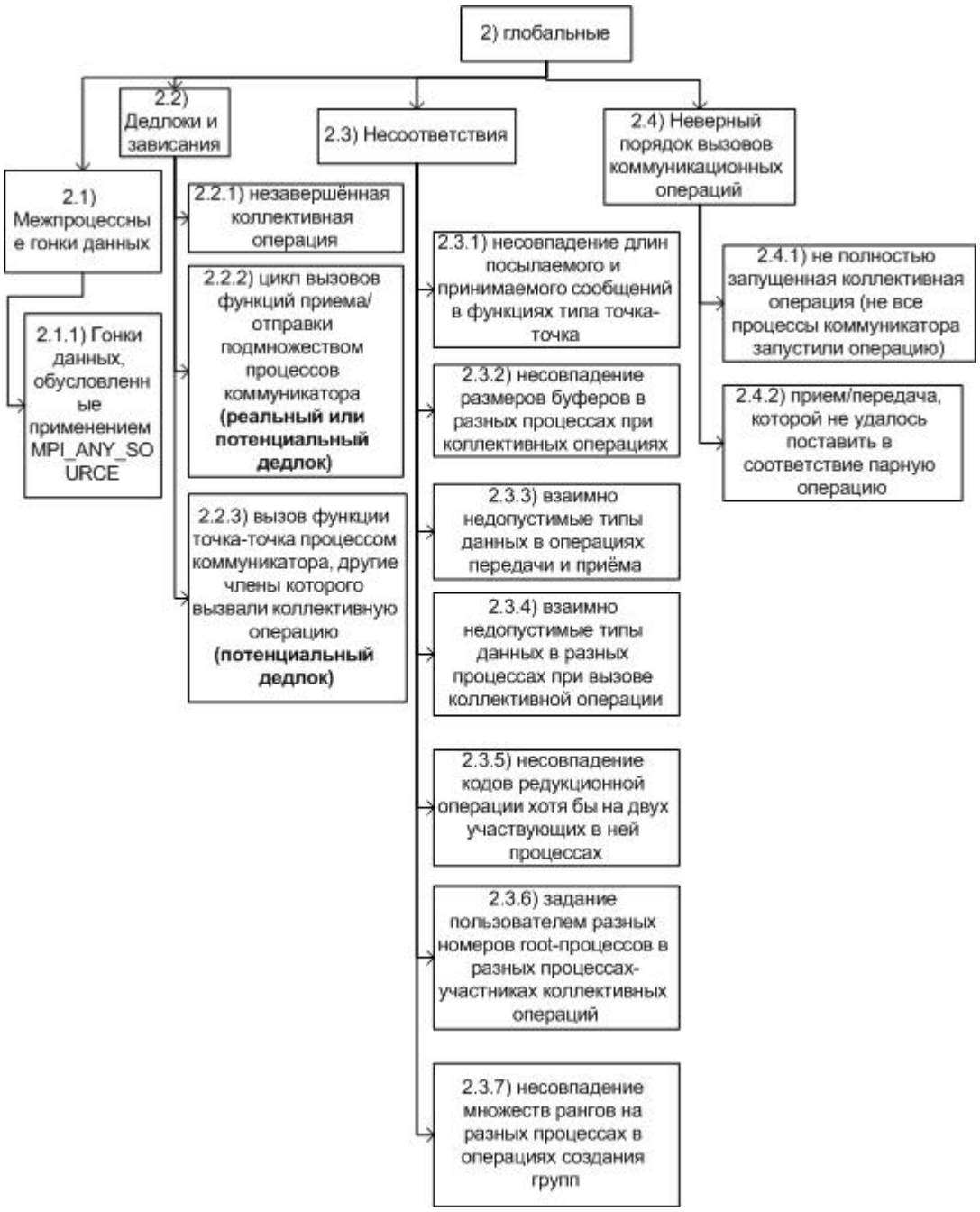

Рисунок 3. Глобальные ошибки

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. Проблемы этого сорта можно разбить на следующие группы:

Общие ошибки:

- процесс завершил свою работу, не вызвав MPI_Finalize (завершающую функцию в любой MPI-программе);
- зависание в MPI-функции (incomplete call).

Некорректная работа со внутренними объектами:

- программа создает слишком большое число «внутренних объектов» или типов данных (их лимит ограничен в MPI Standard);
- на момент окончания работы в программе существуют не освобожденные запросы;
- функция освобождения объекта request была вызвана, когда объект был еще активен;
- повторное использование активного «запроса обмена» (объекта типа MPI_Request);
- отсутствие вызова MPI_Type_free для определенного пользователем типа данных;
- тип данных создан, но не был зафиксирован операцией MPI_Type_commit;
- неопределенный (неверный) коммуникатор;
- неопределенный (неверный) тип данных;
- отсутствие вызова MPI_Comm_free для определенного пользователем коммуникатора;
- неверный «запрос обмена».

Неверная работа с топологиями процессов и производными типами данных:

- в функцию передана отрицательная координата (отрицательный номер процесса в декартовой топологии);
- в функцию передана слишком большая координата (номер процесса, выходящий за границу по данной координате);

- размерность по координате (в функции создания топологии) отрицательна или равна 0;
- размерность слишком велика (больше общего числа процессов);
- кол-во процессов не является произведением размерностей;
- неверно задано смещение (смещение затирает данные или посылает данные дважды);
- новая топология содержит больше процессов, чем старый коммуникатор.

Неверная работа с памятью:

- недопустимое пересечение буфера приёма или передачи запускаемой операции с буфером приёма или передачи какой-либо уже запущенной неблокирующей операции (имеет место как для функций типа точка-точка, так и для коллективных операций) – интерфейсная гонка данных;
- недопустимое пересечение буфера передачи с буфером приёма в коллективной операции;
- во время работы MPI-функции данные были изменены основной ветвью процесса;
- область памяти, переданная в качестве аргумента MPI-операции не доступна для записи или чтения;
- для буферизованной посылки не достаточно выделенной памяти;
- затирание буфера, выделенного для буферизованной посылки следующей посылкой (Bsend) до того, как успела завершиться предыдущая;
- пересечение буферов передачи и приёма в функции MPI_Sendrecv.

Неверная последовательность вызовов или неверные аргументы MPI-функции (имеются ввиду аргументы, не упомянутые в предыдущих типах ошибок – буфер данных, внутренние объекты MPI-реализации):

- неблокирующий прием/передача без последующего вызова MPI_Wait или успешного MPI_Test;
- неверный номер процесса отправителя/приемника (отрицательный или выходящий за пределы коммуникатора);
- отрицательное число отправляемых/принимаемых элементов сообщения;
- тэг операции отрицателен или слишком велик (больше гарантированного стандартом предела).

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

Неверный порядок вызовов коммуникационных операций:

- не полностью запущенная коллективная операция (не все процессы коммуникатора запустили операцию);
- прием/передача, которой не удалось поставить в соответствие парную операцию.

Несоответствия:

- несовпадение длин посылаемого и принимаемого сообщений в функциях типа точка-точка;
- несовпадение размеров буферов в коллективных операциях (например, разное кол-во элементов в разных процессах при вызове MPI_Bcast);
- взаимно недопустимые (несогласованные) типы данных в операциях передачи и приёма;
- взаимно недопустимые типы данных в разных процессах при вызове коллективной операции;
- несовпадение кодов редукционной операции хотя бы на двух участвующих в ней процессах;

- задание пользователем разных номеров root-процессов в разных процессах-участниках коллективных операций.

Неверные аргументы MPI-функций:

- в операциях создания групп неверный диапазон номеров процессов (MPI-рангов).

Дедлоки и зависания:

- незавершённая коллективная операция (все процессы коммуникатора запустили операцию, но не все по какой-либо причине её завершили);
- цикл вызовов блокирующих функций приема/синхронных функций отправки (MPI_Recv/MPI_Ssend) подмножеством процессов коммуникатора – реальный дедлок, или цикл вызовов простой функций отправки (MPI_Send) – потенциальный дедлок;
- вызов функции отправки или приема процессом коммуникатора, все остальные члены которого вызвали коллективную операцию – потенциальный дедлок.

Межпроцессные гонки данных.

Приведенная классификация ошибок не претендует на абсолютную полноту всех возможных проблем, добавляемых в процесс программирования использованием MPI-стандарта. Например, классификация не содержит ошибок, обусловленных использованием функций, которые описаны в стандартах MPI-2 и MPI-3 (параллельный файловый ввод/вывод, односторонние коммуникации и пр.) Ниже разъясняются некоторые термины и ошибки, фигурирующие в классификации.

Дедлоки возникают, когда набор коммуникационных функций, вызванных в различных MPI-процессах, создаёт условия продолжения работы этих процессов, которые никогда не могут быть удовлетворены некоторыми корректными MPI-реализациями [59]. В итоге каждый из таких процессов находится в ожида-

нии сигнала о том, что в другом процессе завершено некоторое действие (на основании этого дедлоки также называют взаимными блокировками). Коммуникационная операция создаёт зависимость, когда MPI-стандарт позволяет реализации блокировать процесс до тех пор, пока при работе другого процесса не произойдёт некоторое событие. Например, MPI-стандарт позволяет реализовать MPI_Send (функцию отправки сообщения) как синхронную операцию, и процесс, вызвавший эту функцию, может находиться в ожидании до тех пор, пока процесс-получатель не вызовет MPI_Recv (функцию приема сообщения). С другой стороны, функция MPI_Send может быть реализована как буферизованная посылка, и завершаться, не дожидаясь вызова MPI_Recv на принимающей стороне. Поэтому в случае, когда 2 процесса обоюдно вызывают MPI_Send, а затем MPI_Recv (листинг **1**), то в зависимости от реализации MPI, дедлок может произойти или нет. Более того, некоторые реализации меняют поведение данной функции в зависимости от объема передаваемых данных – для посылок небольших размеров может использоваться буферизация, а для сообщений большого объема – синхронная передача.

Листинг 1. Пример дедлока

```
if (myrank == 0) {
   MPI_Send ( &a, 1, MPI_INT, 1, 333, MPI_COMM_WORLD );
   MPI_Recv (&b, 1, MPI_INT, 1, 444, MPI_COMM_WORLD,&mpiStatus);
} else if (myrank == 1) {
   MPI_Send ( &c, 1, MPI_INT, 0, 444, MPI_COMM_WORLD );
   MPI_Recv (&d, 1, MPI_INT, 0, 333, MPI_COMM_WORLD,&mpiStatus);
}
```

Пример демонстрирует, что одна и та же параллельная программа в некоторых случаях может приводить к возникновению дедлока, а в некоторых – нет. На

основании этого дедлоки разделяют на стандартные (*реальные*), которые случаются всегда, и *потенциальные*, которые не проявляются на данной архитектуре или реализации MPI, но могут возникнуть при переносе приложения на другую платформу.

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

Листинг 2. Пример межпроцессной гонки данных

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

Выделяют *межпроцессные* и *интерфейсные* ошибки соревнования [59]. Межпроцессная гонка данных происходит, когда несколько процессов пытаются послать одинаковое по логике, но разное по содержанию сообщение. Интерфейсные гонки данных возникают при параллельной работе процесса и вызванной им MPI-функции. Примерами могут служить попытки записи в буфер до окончания операции MPI_Isend или попытки чтения до окончания MPI_Irecv.

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

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

1.2 Опрос пользователей

Привести полную классификацию всех возможных семантических ошибок, которые могут возникнуть в MPI-программе – крайне сложная и вряд ли разрешимая задача. В связи с этим возникает потребность в определении тех типов ошибок, обнаружение которых представляется пользователям наиболее затруднительным. Сотрудниками института прикладной математики имени М.В. Келдыша совместно с Межведомственным суперкомпьютерным центром РАН, научно-исследовательским вычислительным центром МГУ имени М.В. Ломоносова и интернет-сайтом parallel.ru проводилось анкетирование пользователей, занимающихся разработкой MPI-программ [35]. Результаты анкетирования приведены в таблицах 1 и 2.

Эти ответы дают представление о типах специфических ошибок в MPI-программах и важности их обнаружения с точки зрения пользователей.

Таблица 1.

Какие типы ошибок должны обнаруживать средства отладки? Оцените важность от 0 (наименее важно) до 10 (наиболее важно)		**Сумма в баллах**
Неверный порядок коммуникационных операций		
1.	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**
2.	Несогласование send/receive из-за неверного задания отправителя или получателя сообщения	**244**
3.	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**
4.	Различная последовательность вызова коллективных операций и редукционных операций в разных процессах (MPI_Bcast, MPI_Gather, MPI_Allreduce, …)	**202**

Ошибки недетерминированного поведения		
5.	Неповторяющиеся/недетерминированные ошибки, зависящие от времени или порядка принимаемых сообщений	**241**
6.	Неповторяющиеся/недетерминированные ошибки из-за выхода индекса за границу массива	**238**
7.	Неповторяющиеся/недетерминированные ошибки из-за записи в буфер отправки или приема, когда неблокирующие send/receive не завершились	**226**
8.	Неповторяющиеся/недетерминированные ошибки из-за чтения из буфера приема до того, как MPI_Irecv завершилась	**221**
9.	Неповторяющиеся/недетерминированные ошибки из-за неинициализированных переменных	**209**
Специфические для MPI ошибки		
10.	Ошибки, возникающие при изменении среды компиляции или выполнения (например, другая параллельная платформа, компилятор или библиотека MPI)	**222**
11.	Ошибки точности вычисления в редукционных функциях из-за другого порядка вычислений (по сравнению с последовательным)	**207**
12.	Ошибки при создании и использовании коммуникаторов, производных типов и топологий	**181**
13.	Ошибки, возникающие при изменении объема рабочей загрузки	**158**
Неверно заданные параметры		
14.	Ошибочное задание длины сообщения в send/receive	**216**
15.	Неверный адрес буфера пересылаемых/принимаемых данных	**206**
16.	Неверное число и/или тип параметров MPI-функций	**195**
17.	Неверные типы данных в send/receive	**179**
18.	Несовпадение параметров для коллективных операций и редукционных операций в разных процессах	**170**
19.	Ошибки, возникающие при изменении числа процессоров для выполнения и связанные с неверной параметризацией по числу процессоров, размерам массивов и т.п.	**159**

Неверное управление ресурсами		
20.	Исчерпание памяти при использовании неблокирующих операций посылки без задания операции wait	**208**
21.	Исчерпание памяти из-за порождения коммуникаторов, производных типов и т.п., без их освобождения	**184**

Таблица 2.

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 нестабильному/недетерминированному поведению программы (например, два запуска программы дают различные результаты вычислений либо различное завершение)	**334**
2.	Ошибки, приводящие к зависанию программы и завершению по истечении времени	**289**
3.	Ошибки, приводящие к неверным результатам (неверные *значения* расчетных переменных)	**271**
4.	Ошибки, приводящие к авостам в MPI-функциях (плохая диагностика)	**211**
5.	Ошибки, приводящие к недостатку ресурсов (например, если не освобождаются порождаемые ресурсы)	**194**
6.	Ошибки, приводящие к авосту в программе (деление на нуль, переполнение, обращение к чужой памяти)	**182**
7.	Зацикливание программы	**160**

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.

1.3 Методы и программные средства обнаружения семантических ошибок

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

- диалоговая отладка (отладчики TotalView [53], Distributed Debugging Tool [0], mpishim [31, 44]);
- верификация модели программы (MPI-Spin);
- сравнительная отладка (Distributed Virtual Machine);
- автоматизированный анализ корректности:
  - анализ во время работы программы (ITAC - Intel Trace Analyzer and Collector, MARMOT [72, 73], Umpire [87], MPI-CHECK [75], MUST [64, 65, 82]);
  - анализ по трассе (Intel Message Checker, Distributed Virtual Machine, GEPARD [39, 40]).

1.3.1 Диалоговая отладка

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

Самым распространенным и технически развитым средством этой категории является коммерческий отладчик TotalView компании Etnus [53, 49]. Поскольку системы данной группы обладают очень схожим функционалом, то в настоящей работе будет рассмотрен только этот интерактивный отладчик.

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

- процессы и потоки внутри каждого процесса могут быть запущены, остановлены, перезапущены, просмотрены и удалены;
- значения переменных в программе можно изменять во время сеанса отладки, поэтому появляется возможность проверять различные сценарии исполнения без перекомпиляции;
- распределенная архитектура TV позволяет отлаживать удаленные программы во всей локальной сети;
- несколько процессов на разных процессорах могут быть сгруппированы, и когда одна ветвь параллельного приложения достигает точки останова, все сгруппированные процессы будут остановлены.

Для реализации указанных возможностей в TV имеется три окна:

- Root Window – корневое окно;
- Process Window – окно процесса;
- Variable Window – вспомогательное окно.

Root Window дает обзор состояния выполняемой программы. Его также можно использовать как навигационный инструмент. Окно имеет следующие четыре выделенные страницы:

- Attached (подключенные процессы) - показывает список всех отлаживаемых процессов и нитей.
- Unattached (неподключенные процессы) - показывает процессы, над которыми TV имеет контроль. В случае если нельзя подключиться к какому-то процессу, то он изображается серым.
- Groups (группы) - представляет список MPI-групп, используемых выполняемой программой.
- Log (регистрация) - окно отображает собранную отладочную информацию о процессе и потоке внутри этого процесса. Панели внутри этого окна показывают трассу стека, границу стека и код для избранной нити.

Process Window имеет пять информационных панелей:

- Source Pane (панель исходного кода). Левый край этой панели, называемый областью поля тэга, показывает номера строк и иконки, указывающие определенные свойства программы. Можно разместить точку останова на любой строке программы. При этом TV заменяет номер строки на иконку STOP. Стрелка в поле тэгов указывает текущее положение программного счетчика.
- Threads Pane (панель нитей) – показывает список нитей, которые существуют в процессе. Когда выбирается некоторая нить в этом процессе, TV изменяет содержимое Stack Trace Pane, Stack Frame Pane и Source Pane, показывая информацию по этой нити.
- Stack Trace Pane (панель трассы стека) - показывает стек вызова процедур, которые выполняет выделенная нить.
- Stack Frame Pane (панель фрейма стека) показывает функциональные параметры, локальные переменные и регистры для избранного фрейма стека. Информация, представляемая панелями Stack Trace и Stack Frame, отражает состояние процесса на момент его последней оста-

новки. Соответственно, эта информация не корректируется во время выполнения нити.

- Action Points Pane (панель активных точек) показывает список точек останова, точек вычислений и точек наблюдения за процессом.

Variable Window (окно переменных) содержит список адресов, типов данных, значения локальных переменных, регистров или глобальных переменных. Окно также показывает значения, хранимые в памяти.

Следует заметить, что в числе средств обнаружения логических ошибок, использующих диалоговую отладку, присутствуют не только коммерческие системы. Скрипт mpigdb, входящий в состав пакета MPICH2, запускает каждый из процессов MPI-программы под управлением стандартного UNIX-отладчика gdb, предоставляя пользователю возможность воздействовать на определенные процессы в сеансе отладки.

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.

1.3.2 Верификация модели программы

Верификация модели программы (проверка на модели) – это один из подходов к решению проблемы автоматизации отладки и проверки правильности программ [50]. Появление этой техники произошло совсем недавно (в 80-х годах XX в. в работах Кларка, Эмерсона, Квайла и Сифакиса), но, несмотря на это, она уже нашла широкое применение при разработке параллельных приложений.

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

Главным плюсом такого подхода является то, что решается одна из основных проблем предыдущего класса инструментальных средств отладки– большая сложность обнаружения потенциальных ошибок, возникающих из-за недетерминированного поведения параллельной программы. Каждый раз при запуске приложения под управлением диалогового отладчика оно проходит лишь один из возможных маршрутов в графе состояния и проблемы, которые могут появиться на других маршрутах остаются «в тени». Основная же трудность, которую при-

ходится преодолевать в ходе проверки на модели, обусловлена эффектом «комбинаторного взрыва» в пространстве состояний, суть которого заключается в том, что с ростом числа процессов параллельной задачи, участвующих в проверке, число состояний растет, в общем случае, экспоненциально.

Некоторыми исследователями были разработаны подходы по применению метода «верификации программы на модели» к параллельным приложениям [81, 85, 86]. К инструментальным средствам, использующим данный подход, относится система MPI-Spin [85] – расширение известного верификатора Spin. Языком, на котором составляется абстрактная модель параллельной программы, является PROMELA (PRocess MEta LAnguage). Программа на PROMELA переводится в программу на C при помощи Spin (Simple Promela INterpreter), а затем компилируется и запускается обычным способом. Вершины дерева состояний перебираются по алгоритму обхода в глубину. Наряду с полным перебором состояний в графе, Spin имеет также режимы выполнения случайно выбранного маршрута и выполнения маршрута, заданного пользователем (при достижении системой каждой вершины, на которой может происходить разветвление, пользователю выдается сообщение и предлагается сделать выбор из нескольких альтернатив). При использовании MPI-Spin язык Promela дополняется конструкциями, упрощающими моделирование MPI-программ.

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

1.3.3 Сравнительная отладка

Основная идея этого подхода заключается в том, чтобы сравнить поведение данной (отлаживаемой) версии программы с поведением эталонной и выдать пользователю информацию о расхождениях. Часто за эталонную версию принимают последовательный вариант данного параллельного приложения [37]. Для этого выбираются определенные точки программы и в них сравниваются значения переменных. Выбор точек может осуществляться как пользователем (таким образом, например, это реализовано в отладчике Guard [52]), так и самой программной системой поддержки параллельной отладки (как в системе DVM [22]). 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 [2, 37]. При отладке больших параллельных вычислительных задач полные трассы при этом могут достигать очень больших размеров и не помещаться в имеющуюся оперативную память. В связи с этим разработаны различные методы определения тех моментов в программе, где целесообразно устанавливать контрольные точки записи состояния в файл трассы [2]. К ним относятся сбор трассы с граничных итераций вложенных друг в друга циклов, сбор трассы только на угловых значениях переменных циклов (например, если в программе есть двойной цикл *(i, j), i=a..b, j=c..d*, то угловыми будут пары *(a, c), (a, d), (b, c) и (b, d)* соответственно). Такой подход базируется на тех соображениях, что при граничных значениях переменных цикла высока вероятность появления таких ошибок, как неинициализированные переменные, выход за границы массива и пр. Естественно, что из-за применения таких стратегий многие расхождения могут остаться незамеченными для инструментального средства.

Интегрированный в систему DVM сравнительный отладчик OPENMP работает по следующей схеме (для анализа MPI-программ может быть применен ана-

логичный алгоритм). Сначала производится первый прогон параллельной программы со сбором трассы с запущенных потоков, затем автономной утилитой полученная трасса приводится к последовательному виду, т.е. записи трассы располагаются в таком порядке, в котором они были бы выданы при последовательном выполнении. Затем запускается инструментированная последовательная программа и полученная на предыдущем шаге трасса сравнивается динамически с реальным вычислением. В случае OPENMP-приложений для получения последовательной программы из параллельной достаточно всего лишь не принимать во внимание OPENMP-директивы при компиляции.

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

1.3.4 Автоматизированный анализ корректности

Автоматизированный анализ корректности MPI-программ подразумевает под собой выявление поведения параллельной программы, способного привести к семантическим ошибкам. Чаще всего анализируются аргументы и последовательность вызовов MPI-функций. Эта информация сравнивается с данными о вызовах в других процессах (в случае проверки на глобальную ошибку) или с данными о ранее вызванных MPI-функциях в том же процессе (при проверке на локальную ошибку), а затем на основании некоторых алгоритмов делается вывод о существовании реальных или потенциальных ошибок. Для сбора информации об MPI-вызовах применяется профилировочный интерфейс MPI. Этот инстру-

ментарий основан на том, что в любой реализации MPI каждая функция стандарта должна иметь возможность быть вызванной двумя способами: при помощи приставок MPI_ и PMPI_. Таким образом, становится возможным разработать собственную библиотеку MPI-функций, каждая из которых выглядит следующим образом (листинг **3**):

Листинг 3. Профилировка MPI-функции

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

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

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

1.3.5 Анализ по трассе

На основании двух подходов к обнаружению ошибок в MPI-программах, изложенных в приведенном примере, автоматический анализ корректности разделяют на анализ по трассе («посмертный анализ») и динамический анализ. При применении посмертного анализа каждый процесс параллельного приложения записывает данные об MPI-вызовах в локальный файл трассы, после работы файлы с каждого процесса собираются в единую трассу, которая на следующем шаге анализируется отдельной утилитой. Конкретное содержание файлов трассы зависит от формата. Обычно записываются имя MPI-функции, время и аргументы вызова. Метод сбора трассы параллельной программы для последующего анализа [61] часто используется инструментальными средствами оптимизации, в числе которых можно назвать Scalasca[74, 89] и Vampir [58]. Существуют также средства, предназначенные только для сбора трассы (например, VampirTrace

[84]). Файлы трассы при этом могут быть в форматах OTF(2) [60, 88]; Epilog [90], STF [66]и др. В последнее время разрабатываются также интеграционные решения, способные работать с различными типами вычислительных систем, «сборщиками трасс» и базами данных для хранения собранной информации. В качестве примера стоит назвать разработку Dataheap [70].

Данный подход использовался при разработке программной системы Intel Message Checker [59]. Средство состояло из трех компонент:

- Trace Collector – подсистема для сбора информации об MPI-вызовах в файле трассы во время выполнения параллельной программы.
- Analyzer Engine – утилита, которая проводит анализ файла трассы. Диагностическую информацию Analyzer Engine записывает назад в файл трассы для третьего компонента Visualizer.
- Visualizer – загружает файл трассы и позволяет пользователю просматривать ошибки с автоматическим перемещением к строкам кода, где они возникли.

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

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

Программная система в состоянии обнаружить следующие ошибки:

- несоответствия функций посылки и приема из-за некорректного указания посылающего или принимающего процесса в вызове коммуникационной функции;
- потенциальные и реальные дедлоки;
- ошибки, вызванные записыванием в буфер посылки перед завершением функции MPI_Isend;
- разный порядок вызовов коллективных и редукционных операций;
- ошибки, вызванные перекрыванием буфера посылки или приема с другой коммуникационной операцией (обнаружение данной ошибки не реализовано для производных типов данных);
- ошибки несоответствия размеров сообщений в операциях посылки и приема;
- несоответствия контрольных сумм посланных и принятых сообщений;
- аварийное завершение работы во время вызова MPI-функции;
- несоответствия типов передаваемых и принимаемых данных.

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

В настоящее время корпорация Intel отказалась от дальнейшей разработки данного программного средства и сейчас многие идеи, заложенные в нем, реализованы в системе Intel Trace Analyzer and Collector, которое будет рассмотрено ниже. Один из недостатков инструментальных средств посмертного анализа

описан выше при обсуждении систем сравнительной отладки – неконтролируемое разрастание файла трассы при работе с параллельной программой, ветви которой интенсивно обмениваются сообщениями. Другой слабой стороной является непозволительная продолжительность работы компонента Analyzer Engine на больших трассах – данная утилита не была распараллелена.

В отечественную систему DVM также входит подсистема отладки MPI-программ, использующая подход автоматизированного анализа по трассе [42, 45]. Пользователь компилирует свою программу одной из команд *dtcc, dtf77, dtf90*. Эти команды отличаются от команд *mpicc, mpif77, mpif90* тем, что программа компилируется с построением отладочных таблиц и линкуется с трассировщиком. Откомпилированная программа запускается при помощи команды *dtrun*, имеющей те же аргументы, что и *mpirun* из пакета MPICH. При этом по ходу работы MPI-программы для каждого процесса собирается трасса вызовов MPI-функций. По завершении работы программы пользователь командой *dtan* загружает компонент «анализатор корректности», который производит анализ накопленных файлов трасс и делает вывод о возникших семантических ошибках. Результаты анализа записываются в файл с именем <*task*>*.tran*, где <*task*> - имя пользовательской MPI-программы.

Однако описанная подсистема отладки не входит в состав версий системы DVM для ОС Windows, поэтому она не была подвергнута тестированию для сопоставления с созданным в рамках настоящей работы программным средством.

1.3.6 Анализ во время исполнения

Рассмотрению существующих программных средств, относящихся к данному направлению, и используемых ими алгоритмов будет уделено большее внимание, поскольку предлагаемая в данной работе информационная система также принадлежит к этому классу. Отличие данного подхода от предыдущего заклю-

чается 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 [71, 72] – разработке Штуттгартского центра высокопроизводительных вычислений, существует выделенный сервер отладки, который собирает данные об MPI-вызовах от процессов на рабочих узлах кластера. Архитектуру MARMOT можно представить следующим образом (рисунок 4):

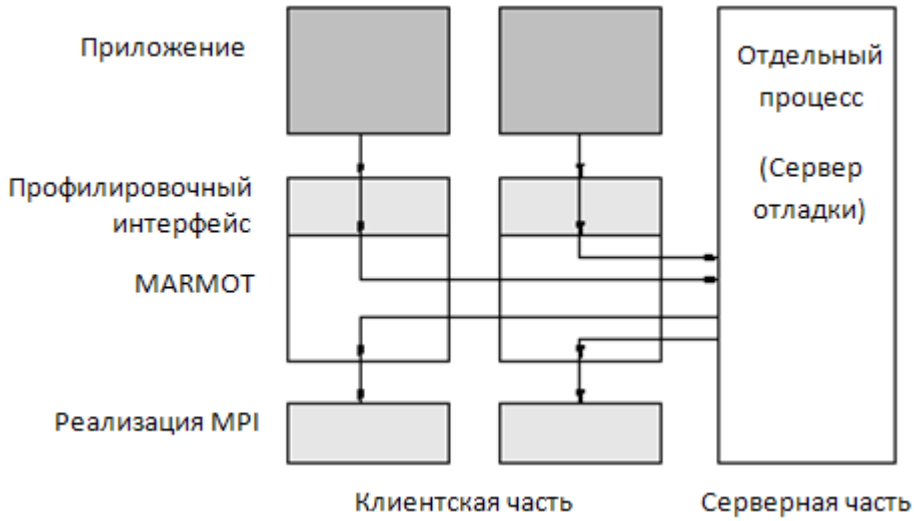

Рисунок 4. Архитектура системы MARMOT

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

Ниже подробно рассмотрены алгоритмы обнаружения семантических ошибок, применяемые в MARMOT.

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

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

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

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

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

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

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

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 проверяет сразу же, а для обнаружения глобальных осуществляется дополнительная посылка с параметрами вызова, имеющая тот же коммуникационный характер, что и проверяемая MPI-операция, т.е. посылка тому же адресату в случае, если проверяемой операцией является MPI-функция типа точка-точка и коллективная в случае проверки групповой коммуникации.

На все ситуации некорректного поведения ITAC выдает реакции в виде предупреждений или ошибок. 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 ITAC для поиска семантических ошибок.

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

Ошибки, когда процесс пытается изменить содержимое области памяти, которая в данный момент участвует в неблокирующей коммуникационной операции, обнаруживаются путем вычисления контрольной суммы сразу после того, как была инициализирована коммуникационная операция и после окончания работы данной функции. Если вычисленные контрольные суммы разные, значит, произошла модификация. Такие ошибки могут произойти также в случае блокирующих посылок (например, в случае если программа многопоточная и один поток занимается коммуникационной операцией, в то время как другой изменяет значение буфера).

Использование неинициализированной памяти. В реализации для Linux-систем обнаружение производится при помощи обращения к программе valgrind - если valgrind не установлен, то данная опция отключена.

Управление внутренними ресурсами. Контроль использования объектов типа «запрос обмена» (MPI_Request) проводится следующим образом. Каждый раз, когда число активных запросов превышает CHECK-MAX-REQUESTS – задаваемую пользователем константу, выдается предупреждение. По окончании работы параллельной программы выдается сообщение, если хотя бы 1 запрос обмена не был освобожден. Если приложение освободило такой объект до того, как была закончена соответствующая операция, то выдается предупреждение. В случае, если запущена операция с активным в текущий момент запросом, выводится отчет с ошибкой.

Неосвобождение типов данных может привести к тем же последствиям, что и неосвобождение объектов «запрос обмена», поэтому как только число таких типов данных превышает константу CHECK-MAX-DATATYPES, то выдается предупреждение.

Гонки данных. Одной из причин возникновения гонок данных является некорректное использование буферизованных посылок. Если подряд следуют несколько вызовов функции буферизованной отправки сообщения, то может возникнуть ситуация, когда 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 участков памяти, выводящихся в виде *[start_addr, end_addr]*.

Дедлоки. Дополнительная нить для каждого процесса обменивается информацией с оберткой для MPI-функции (MPI wrapper) чтобы обнаружить, что процесс «завис» в выполнении некоторой функции. Если все процессы, участ-

вующие в коммуникации, зависли, то тогда выдается сообщение о дедлоке. Тайм-аут, по истечении которого выдается такое сообщение, задается при помощи параметра DEADLOCK-TIMEOUT. 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 заменяется на MPI_Ssend.

Дедлок может также возникнуть при применении буферизованных посылок во время работы функции MPI_Buffer_detach - она блокирует процесс до тех пор, пока все сообщения внутри буфера не будут посланы. Для обнаружения таких дедлоков также применяется механизм временных тайм-аутов.

Ошибки несоответствия. Перед каждой коллективной операцией при помощи MPI_Bcast всем процессам рассылается следующая информация: тип операции, root-процесс операции, тип редукции, после этого все процессы сравнивают эти параметры со своими значениями. Для предопределенных типов данных выводится ошибка. Об ошибках в параметрах пользователю сообщается как GLOBAL:COLLECTIVE:INVALID_PARAMETER с описанием неверных параметров.

Архитектура ITAC хорошо подходит для обнаружения локальных ошибок в связи с тем, что в настоящее время абсолютное большинство производимых процессоров являются многоядерными, а при работе MPI-приложения далеко не всегда все ядра бывают загружены. Таким образом, дополнительный анализирующий поток на каждом вычислительном узле кластера работает без серьезного ущерба для производительности. 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, вызванной пользователем, 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.

Таким образом, среди других подходов к отладке параллельных программ автоматизированный анализ корректности во время исполнения выделяется тем, что посредством применения эвристических алгоритмов средство отладки может обнаружить ошибки, обусловленные недетерминизмом (потенциальные дедлоки и гонки данных), 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 пока инструментальное средство произведет перебор всех вариантов исполнения параллельного приложения. Единственное, что требуется от пользователя – слинковать свою программу с профилировочной библиотекой во время компиляции, запустить на необходимом числе узлов вычислительного кластера процессы-серверы (если они предусмотрены в системе), а после этого – запустить MPI-программу обычным образом. Однако заложить в систему автоматизированного контроля корректности проверки на все возможные ошибки, обусловленные некорректным использованием MPI-стандарта, практически невозможно. Сложность организации всех возможных проверок является следствием не только очень большого количества таких ошибок, но и того, что поддерживать такую систему в актуальном состоянии чрезвычайно не просто, поскольку MPI-стандарт регулярно претерпевает обновления (MPI Forum [77] постоянно вносит изменения в наборы функций, MPI-типов, констант и пр.) В то же время пользователю далеко не всегда требуется выполнение системой всех возможных проверок на его программе – зачастую он приблизительно представляет, в каких местах и функциях может возникнуть ошибка, а излишние проверки создают дополнительную нагрузку на вычислительную систему. К тому же известные про-

граммные системы, реализующие данный подход, используют довольно грубые методы для поиска семантических ошибок.

ГЛАВА 2. ШАБЛОНЫ ОШИБОЧНОГО ПОВЕДЕНИЯ

Для решения обозначенной выше проблемы систем автоматизированного контроля корректности, состоящей в сложности организации проверок на все возможные ошибки, в данной работе предлагается ввести возможность определения пользователем такого поведения параллельной программы, которое должно быть обнаружено системой.

2.1 Система описания шаблонов ошибочного поведения.

Для определения пользователем ошибок, на наличие которых требуется проверить программу, необходимо разработать некоторую систему описания. Такое описание, естественно, можно было бы выполнить с использованием какого-либо существующего языка представления знаний, например XML. Однако введение собственной схемы XML в некотором смысле эквивалентно введению правил новой системы описания. К тому же в этом случае пользователю пришлось бы изучать также и правила формирования XML-документов. При разработке системы описания авторы стремились к простоте записи, удобочитаемости и, по возможности, малому объему текста в файле.

Для начала нужно выяснить состав системы. Поскольку разрабатываемое программное средство использует подход автоматизированного контроля корректности, то оно получает информацию о вызываемых MPI-функциях при помощи профилировочного интерфейса. Следовательно, все, чем может оперировать средство отладки – это имена вызываемых функций, их аргументы и номера MPI-процессов (в информационную систему добавлен модуль анализа исходного кода приложения, благодаря чему становится возможным получить некоторую дополнительную информацию типа номеров строк кода и пр.) Именно эти параметры и легли в основу системы описания. Для описания ситуации пользователь должен указать:

- процессы, участвующие в данной ситуации;

– функции, вызываемые (или наоборот – не вызванные) каждым из процессов;
– значения, принимаемые аргументами функций.

При этом отождествлять описываемую пользователем ситуацию с семантической ошибкой будет не совсем верно. Дело в том, что указанная ситуация может возникнуть вследствие нескольких ошибок. Например, пользователь указывает, что один процесс должен вызвать функцию отправки типа точка-точка, а во втором должен отсутствовать вызов соответствующей операции приема. Такая ситуация может возникнуть из-за того, что в программном коде, выполняемом вторым процессом либо в принципе отсутствует функция приема от первого, либо не дойдя до нее, второй процесс завис в некоторой другой операции (например, коллективной коммуникации). В связи с этим ситуации, описываемые при помощи данного системы, авторы назвали «шаблонами», а саму систему – «системой описания шаблонов».

Шаблоны могут быть более или менее детализированы. Так, пользователь должен иметь возможность указать, что в некоторой ситуации могут участвовать любые два процесса или процессы с определенными номерами, если он знает, что ошибка возникает при взаимодействии именно этих двух.

Итак, перейдем к подробному описанию общей схемы составления шаблона. Каждый шаблон пользователь заполняет в отдельном файле, содержащем следующие разделы:

1) Название шаблона (параметр «Name»).
2) Количество (параметр «K») и, опционально, номера процессов («p1», «p2»,…). При этом можно указать, что в описываемой ситуации принимают участие все процессы запущенной программы, применив обозначение «n(MPI_COMM_WORLD)».

Пример 1: K=2 p1=0 p2=1 (два процесса с номерами «0» и «1» в коммуникаторе MPI_COMM_WORLD).

Пример 2: K=n(MPI_COMM_WORLD) (все процессы MPI-программы).

3) Функции, которые должны вызвать или, наоборот, не вызвать указанные процессы для возникновения описываемой ситуации. Каждая функция обозначается как «Fi», где i – номер функции. Указывается обозначенный в предыдущем разделе номер процесса (pj) и через двоеточие – имя MPI-функции без префикса «MPI».

Пример 3: F1=p1:Bsend F2=p2:Recv

Для обозначения любой функции из некоторого множества существуют специальные макросы. Так, запись «F1=p1:Send_any» обозначает, что первый процесс вызывает любую из блокирующих операций отправки. Существуют также макросы «(I)PTP» - любая (не)блокирующая операция типа точка-точка; «Coll_data» - любая коллективная коммуникация, каким-либо образом перераспределяющая данные (MPI_Bcast, MPI_Gather, MPI_Scatter, MPI_Alltoall); «Coll_reduc» - любая коллективная редукционная операция (MPI_Reduce, MPI_Scan, MPI_Allreduce); «Recv_any» - любая блокирующая операция приема типа точка-точка и некоторые другие.

Пример 4: F1=p1:Send_any F2=p2:Recv_any

Для логического объединения нескольких условий необходимо было ввести обозначения для конъюнкции (&&), дизъюнкции (||) и отрицания (!). Так, для указания того, что для возникновения ситуации процесс может вызвать операцию типа точка-точка или коллективную применяется следующая запись.

Пример 5: F1=p1:PTP||Coll_data||Coll_reduc

Кроме того, можно указать сразу несколько функций при помощи одного выражения. Так, строка «FA=pA:Coll_reduc» в файле шаблона обозначает, что все процессы, участвующие в описываемой ситуации (их номера или количество указывается в предыдущем блоке), должны вызвать коллективную редукционную функцию.

4) Следующим блоком в файле шаблона является блок описания условий. Поскольку в предыдущем разделе все участвующие в ситуации функции были введены при помощи символа «F» с номером, то в этом блоке обращение к ним идет посредством данных обозначений. Общее правило составления условий на аргументы состоит в том, что пользователь указывает функцию, номер ее аргумента и значение, которое должен принимать этот аргумент. При этом в качестве значения можно употреблять конкретное число: «F1(2)=7»; номер процесса: «F1(4)=p1»; значение какого-либо аргумента другой функции: «Fi(2)!=Fj(2)» или макрос, определенный в стандарте MPI: «F1(4)=MPI_ANY_SOURCE». Кроме того, пользователь может задавать рекурсивные условия.

Пример 7: Fj(4)=pj+1, j:=1,K-1

FK(4)=p1

Так, в примере 7 указано, что четвертый параметр всех функций с номерами от 1 до K-1 («K» - общее количество процессов), участвующих в ситуации, равен номеру процесса, следующего за тем, который вызвал данную функцию. Для замыкания цикла отдельно задается условие для последнего по номеру процесса (в примере приведен фрагмент шаблона, описывающего «дедлок», обусловленный циклом операций отправки или приема). Условия, фигурирующие в данном блоке, базируются на логических отношениях равенства «=», неравенства «!=», больше «>» и меньше «<».

В MPI-программе возможны также ошибки изменения передаваемых данных во время коммуникации. Для их обнаружения необходимо сравнивать контрольные суммы передаваемых буферов до начала коммуникации и после. Для этой цели в систему описания введены конструкции «CRC_BEFORE», «CRC_AFTER».

Пример 8: CRC_BEFORE(F1(1)) != CRC_AFTER(F2(1))

В примере 8 F1 – может быть неблокирующей функцией отправки типа точка-точка (MPI_Isend, MPI_Issend и пр.), а F2 – функцией приема. Ошибка состоит в том, что CRC-код, посчитанный для принятого буфера, не совпадает с CRC-кодом буфера, который был изначально отправлен. Это говорит о том, что данные были изменены основной ветвью процесса во время неблокирующей отправки.

Кроме того, в данном блоке можно записывать несколько условий при помощи обозначений «&&» и «||». Запись одного условия под другим эквивалентна записи их на одной строке через «&&».

Пример 9: F1(5)<0 || F1(5)>MPI_TAG_UB

Кроме того, любая строка файла шаблона может начинаться с символов «REM», что означает начало строчного комментария.

Более формально введенная система описания шаблонов определена в расширенной форме Бэкуса-Наура в приложении 1. Ниже (листинг 4) рассмотрен пример полного шаблона ошибочного поведения, с которым может работать создаваемое программное средство автоматизированного контроля корректности. Описываемая шаблоном ошибка «Повторное использование активного запроса обмена» заключается в том, что некоторый процесс вызывает неблокирующую функцию типа точка-точка, а затем, не освобождая объект «запрос обмена» (re-

quest) при помощи функций MPI_Wait и MPI_Test вызывает еще одну неблокирующую функцию, использующую тот же самый «запрос обмена». В результате такой последовательности информацию о завершении первой операции получить позднее будет уже невозможно. Здесь необходимо отметить, что существуют и иные функции, способные освободить объект «запрос обмена» (MPI_Waitsome, MPI_Waitany, MPI_Testsome,…). Однако, если исследователь не использует в своей программе эти функции, то прописывать в шаблоне все возможные варианты нет необходимости.

Листинг 4. Шаблон «Повторное использование активного запроса обмена»

```
Name=Repeated using of active request
1block
K=1
2block
F1=p1:IPTP
F2=p1:Wait! && Test!
F3=p1:IPTP
3block
F1(7)=F2(1)
F1(7)=F3(7)
```

Количество процессов, участвующих в ошибке – 1 (K=1). Последующие три записи в блоке функций означают, что для возникновения ситуации необходимо два вызова одним и тем же процессом неблокирующих коммуникационных функций типа точка-точка (необязательно следующих подряд друг за другом), между которыми отсутствует вызов функции ожидания или теста на завершение. Третий блок говорит о том, что параметр «запрос обмена» должен быть одинаковым в обеих неблокирующих коммуникациях и операции ожидания/теста на

завершение. Это необходимо потому, что даже если между вызовами функций F1 и F3 присутствует вызов какой-либо операции освобождения «запроса обмена», то совсем не обязательно эта операция должна относиться к F1 (она может освобождать совершенно другой «запрос обмена», задействованный в одной из предыдущих неблокирующих операций).

Таким образом, несмотря на то, что введенная система описания обладает очень небольшим набором элементов, этого вполне достаточно для описания любой ошибки, которую можно обнаружить при помощи подхода автоматизированного контроля корректности. Более того, можно конкретизировать ошибочную ситуацию, указав конкретные номера MPI-процессов или дополнительные условия на аргументы функций, и тогда средство отладки не будет осуществлять лишние проверки. Кроме того, при помощи системы описания пользователь может определить собственные проверки, не являющиеся ошибками, которые должно выполнить инструментальное средство во время работы программы. Стоит добавить, что если бы для описания ошибочных ситуаций была бы разработана соответствующая схема в XML, то тексты шаблонов были бы существенно более громоздкими.

2.2 Примеры шаблонов

В этом пункте приведены составленные при помощи введенной системы описания шаблоны для некоторых семантических ошибок, фигурирующих в классификации первой главы. Некоторые ошибки MPI-реализации обнаруживают самостоятельно. Так, например, если при исполнении параллельного приложения встречается функция отправки, номер адресата которой больше числа запущенных процессов, то пакет MPICH2 аварийно завершает приложение и сообщает об ошибке и ее локализации пользователю. Тем не менее, даже для таких

ошибок были разработаны соответствующие шаблоны. Причиной этого, во-первых, является то, что MPI-реализации не обязаны предоставлять такой функционал и пользователь не может полагаться на то, что ошибки могут быть найдены без помощи специального средства отладки. А во-вторых, этим подтверждаются широкие возможности созданной системы описания шаблонов, при помощи которой можно формализовать практически любую ошибочную ситуацию, вызванную неверным использованием MPI-стандарта.

Ниже приведен общий список из 22 ошибок, для которых разработаны шаблоны. При этом 20 ошибок обусловлены неверным использованием функций стандарта MPI-1 и 2 – MPI-2. Шаблоны для последних двух ошибок были добавлены для демонстрации того, что введенная система описания пригодна для записи ошибок, где фигурируют операции и объекты любой версии стандарта. При этом работа созданного в рамках диссертации программного средства с функциями, привнесенными стандартами MPI-2 и MPI-3 производится таким же способом, как и с функциями MPI-1.

- Несовпадение длин посылаемого и принимаемого сообщений в функциях точка-точка.
- Взаимно недопустимые типы данных в операциях передачи и приема.
- Несовпадение размеров буферов в разных процессах при коллективной операции:
    a. для функций, перераспределяющих данные;
    b. для редукционных операций.
- Взаимно недопустимые типы принимаемых и отправляемых данных в векторном варианте коллективной операции:
    a. для функций сбора данных (MPI_Gatherv, MPI_Allgatherv);
    b. для функции рассылки данных (MPI_Scatterv);

c. для функции перераспределения данных (MPI_Alltoallv).

- Несовпадение кодов редукционной операции в разных процессах.
- Задание пользователем разных номеров root-процессов в разных процессах-участниках коллективной операции.
- Несовпадение количества процессов во вновь создаваемой группе.
- Незавершенная коллективная операция.
- Выход без вызова функции MPI_Finalize.
- Неблокирующий прием/передача без последующего вызова MPI_Wait или успешного MPI_Test.
- Повторное использование активного запроса обмена.
- Неосвобождение определенного пользователем типа данных (MPI_Type_contiguous).
- Межпроцессные гонки данных (гонки данных, обусловленные применением MPI_ANY_SOURCE).
- Дедлоки, обусловленные циклом отправок или получений.
- Вызов коммуникационной функции точка-точка процессом, парный для которого вызвал коллективную операцию.
- Неверный номер процесса приемника/отправителя (отрицательный или выходящий за пределы коммуникатора).
- Тэг операции точка-точка отрицателен или слишком велик.
- Передача, которой не удалось поставить в соответствие парную операцию.
- Отрицательное число отправляемых/принимаемых элементов сообщения.

- Изменение данных, участвующих в неблокирующей коммуникации, основным потоком MPI-процесса.
- Неверный режим доступа при открытии файла.
- Отсутствие закрывающей функции для использованного файла.

Также созданы 2 шаблона для пользовательских проверок:

- Превышение количества передаваемых элементов заданного значения.
- Коммуникации между процессами, разность номеров которых более единицы при пересылке данных по кольцу.

Далее приведен текст некоторых шаблонов с пояснениями. Полный список шаблонов представлен в приложении 4. Для начала будут описаны локальные ошибки. К этому множеству, в частности, относится приведенный выше шаблон ошибки «Повторное использование активного запроса обмена». По определению локальной ошибки, ее причина локализована в пределах одного процесса, поэтому во всех таких шаблонах параметр «K» равен единице.

2.2.1 Шаблоны для локальных ошибок

1) Выход без вызова функции MPI_Finalize

В качестве первого простейшего примера приведен шаблон для ошибки «Выход без вызова функции MPI_Finalize» (листинг 5). Согласно MPI-стандарту вызов MPI_Finalize является обязательным для каждого процесса параллельной программы, и игнорирование данного правила может привести к некорректному завершению работы остальных процессов.

Листинг 5. Шаблон «Выход без вызова MPI_Finalize»

```
Name=Exit without MPI_Finalize
1 block
```

```
K=1
2 block
F1=p1:Finalize!
```

Для обозначения ошибочной ситуации, состоящей в том, что существует процесс, который во время своей работы не вызывает MPI_Finalize, достаточно строки «F1=p1:Finalize!» во втором блоке шаблона. Поскольку в шаблоне нет уточнения относительно того, какой MPI-процесс должен участвовать в ситуации, то в качестве «p1» может выступить любой процесс.

2) Неблокирующий прием/передача без последующего вызова MPI_Wait или MPI_Test

Следующая ошибочная ситуация включает в себя оперирование уже двумя функциями — вызов неблокирущей операции точка-точка и отсутствие вызова фукции, освобождающей «запрос обмена», используемый в первой операции.

Листинг 6. Шаблон «Неблокирующий прием/передача без последующего вызова Wait или Test»

```
Name=Unblock send/receive without freeing request
1 block
K=1
2 block
F1=p1:IPTP
F2=p1:Wait! && Test!
3 block
F1(7) = F2(1)
```

В качестве первой функции используется введенный в систему описания макрос «IPTP» для обозначения любой неблокирующей коммуникационной операции типа точка-точка. Для указания того, что некоторый процесс после вызова неблокирующей операции не должен вызывать ни MPI_Wait, ни MPI_Test при-

менена запись, содержащая конъюнкцию. В тексте шаблона в качестве примера указаны только 2 функции для освобождения «запроса обмена». Поскольку пользователь составляет шаблоны сам, то он может перечислить только те процедуры, которые использует в своей программе. Если, например, он никогда не употребляет функцию MPI_Waitany, то перечислять ее в шаблоне не имеет смысла. Это служит еще одним подтверждение гибкости созданной системы описания ошибочных ситуаций — средства, в которых набор проверок фиксирован, в рассматриваемом случае обязаны включать в себя проверки на вызов абсолютно всех функций освобождения «запроса обмена».

3) Повторное использование активного запроса обмена

Текст данного шаблона был приведен выше (листинг 4). Необходимо заметить, что принципиальное отличие от шаблона «неблокирующий прием/передача без последующего вызова Wait или Test» состоит в том, что в данном случае вызов функции освобождения запроса обмена должен отсутствовать только между двумя неблокирующими пересылками. Это важно потому, как функция *F2*, фигурирующая в шаблоне, должна быть вызвана до функции *F3* в программе пользователя.

4) Неосвобождение определенного пользователем типа данных

Следующая ошибка относится к подгруппе «Некорректная работа со внутренними объектами MPI» (рисунок 2). Существует несколько конструкторов типов данных: MPI_Type_contiguous, MPI_Type_vector, MPI_Type_hvector, MPI_Type_indexed и др. Здесь в качестве примера рассмотрен шаблон для первого из них – MPI_Type_contiguous.

Листинг 7. Шаблон «Неосвобождение определенного при помощи MPI_Type_contiguous типа данных пользователем»

```
Name=Unfreed user defined datatype
```

```
1 block
K=1
2 block
F1=p1:Type_contiguous
F2=p1:Type_free!
3 block
F1(3)=F2(1)
```

Данный шаблон содержит условие на равенство третьего аргумента MPI_Type_contiguous и второго аргумента MPI_Type_free. Этот аргумент – новый тип данных. По аналогии с шаблоном «Неблокирующий прием/передача без последующего вызова Wait или Test» (листинг 6) данное условие нужно для указания того, что MPI_Type_free должен освобождать именно тот тип, который был создан конструктором *F1*. Шаблоны для других функций-конструкторов пользовательских типов данных строятся путем замены названия функции в строке для *F1* и номера аргумента *F1* в условии.

5) Тэг операции точка-точка отрицателен или слишком велик

Стандарт MPI также накладывает ограничения на диапазон значений некоторых объектов. Одним из них является тэг операций типа точка-точка. Несмотря на то, что в качестве этого аргумента функции передается целочисленная переменная (int), ее значение должно быть не менее 0 и не более определенной в MPI константы – MPI_TAG_UB.

Листинг 8. Шаблон «Тэг операции отрицателен или слишком велик»

```
Name=Message tag is negative or too large
1 block
K=1
2 block
F1=p1:PTP||IPTP
```

```
3 block
F1(5)<0 || F1(5)>MPI_TAG_UB
```

Данный шаблон демонстрирует применение логической операции (дизъюнкции) к условиям шаблона. Тэг является пятым аргументом, как в блокирующих, так и в неблокирующих коммуникациях точка-точка, поэтому в рамках одного шаблона объединены оба множества функций, для чего используются определенные в системе описания макросы «PTP», «IPTP» и дизъюнкция.

6) *Неверный номер процесса приемника/отправителя (отрицательный или выходящий за пределы коммуникатора)*

В коммуникациях типа точка-точка четвертый аргумент является номером партнера по операции. Так, для функции отправки указывается номер процесса-получателя, а для приема – номер процесса-отправителя. Поскольку процессы в MPI нумеруются с нуля, и их количество при запуске указывается пользователем, то поставить в качестве четвертого аргумента произвольное число нельзя.

Листинг 9. Шаблон «неверный номер процесса отправителя/получателя»

```
Name=Illegal number of sender/receiver
1 block
K=1
2 block
F1=p1:PTP||IPTP
3 block
F1(4)<0 || F1(4)>=n(MPI_COMM_WORLD)
```

Шаблон для данной ошибки сходен с шаблоном «тэг операции отрицателен или слишком велик». Здесь продемонстрировано использование введенного в системе обозначения для общего количества всех запущенных процессов параллельной программы — «n(MPI_COMM_WORLD)».

2.2.2 Шаблоны для глобальных ошибок

Поскольку глобальные ошибки, по определению, возникают в результате взаимодействия нескольких процессов, то во всех шаблонах данного пункта параметр K >= 2. Здесь также будут приведены два шаблона для пользовательских проверок на ситуации, не являющиеся нарушением MPI-стандарта. Рассмотрение начнем с ошибок несоответствия в коммуникациях точка-точка, где участвуют отправляющий и принимающий процессы.

1) Несовпадение длин посылаемого и принимаемого сообщений в функциях точка-точка.

Листинг 10. Шаблон «Несовпадение длин посылаемого и принимаемого сообщений в функциях точка-точка»

```
Name=Sender’s and receiver’s messages length mismatch
1 block
K=2
2 block
F1=p1:Send_any
F2=p2:Recv_any
3 block
F1(2)!=F2(2)
```

Ошибочная ситуация заключается в том, что один из процессов вызывает какую-либо из функций отправки, а его партнер по коммуникации – функцию приема. Для обозначения этого применены макросы «Send_any» и «Recv_any». Поскольку длина сообщения – второй аргумент любой операции точка-точка, то ошибочную ситуацию в шаблоне можно задать, применив условие «*F1(2)!=F2(2)*».

2) Межпроцессная гонка данных, обусловленная применением макроса MPI_ANY_SOURCE.

Гонки данных – ошибки, обусловленные недетерминизмом исполнения параллельной программы. В большей мере они присущи технологиям, обеспечивающим работу на вычислительных системах с общей памятью. Однако в некоторых случаях гонки данных встречаются и в MPI-программах. Один из них – когда два процесса отправляют третьему сообщение, а тот выполняет прием с макросом «MPI_ANY_SOURCE». При нескольких запусках такой программы даже на одной вычислительной системе функция приема может получать информацию то от одного, то от другого процесса. Семантика программы может быть построена таким образом, что в любом случае приложение продолжит работу корректно. Но может быть и так, что пользователь всегда рассчитывает на определенный порядок получения и не учитывает другую возможность. Как раз в этом случае может иметь место действительная ошибка.

Листинг 11. Шаблон «Межпроцессная гонка данных, обусловленная применением макроса MPI_ANY_SOURCE»

```
Name=Data race caused by using “MPI_ANY_SOURCE” in receive function
1 block
K=3
2 block
F1=p1:Send_any
F2=p2:Send_any
F3=p3:Recv_any
3 block
F1(4)=p3
F2(4)=p3
F3(4)=MPI_ANY_SOURCE
```

В шаблоне участвуют 3 процесса – два отправляющих и один принимающий. Соответственно, во втором блоке прописаны 3 функции (по одной для каждого из процессов). Третий блок демонстрирует возможность системы описания

задавать в качестве значения аргумента MPI-функции номер процесса (первое и второе условие) и определенный в MPI-стандарте макрос (третье условие). Еще раз отметим, что в качестве «*p1*», «*p2*» и «*p3*», фигурирующих в шаблоне, могут выступать процессы с любыми номерами в коммуникаторе MPI_COMM_WORLD.

3) Дедлоки, обусловленные циклом отправок или получений.

«Мертвые блокировки» или дедлоки в MPI-программе могут быть вызваны различными причинами. Одна из них – цикл, образованный операциями типа точка-точка. Такой цикл могут образовать 2 и более процессов.

Листинг 12. Шаблон «Дедлок, вызванный циклом операций отправки или получения»

```
Name=Deadlock caused by cycle of sends or receives
1 block
K<=n(MPI_COMM_WORLD)
2 block
FA=pA:Send||Ssend||Recv
3 block
Fj(4)=p(j+1), j:=1,K-1
FK(4)=p1
```

Поскольку количество процессов, участвующих в ошибочной ситуации, может быть различным, то единственное ограничение – чтобы оно было не больше общего количества запущенных процессов. Второй блок говорит о том, что цикл блокировок может быть образован функциями «MPI_Send», «MPI_Ssend» или «MPI_Recv». Неблокирующие функции или буферизованный вариант отправки («MPI_Bsend») не ожидают до выполнения ответного действия со стороны партнера по коммуникации, поэтому не могут вызвать подобный дедлок. Третий блок демонстрирует возможность объединения нескольких условий при помощи одного выражения. Четвертый аргумент функций точка-точка –

номер процесса-отправителя или получателя соответственно. Для образования цикла каждый процесс должен отправлять/принимать сообщение следующему. Для замыкания цикла последнее условие выписано отдельно.

4) Несовпадение размеров буферов в разных процессах при коллективной операции.

Ряд глобальных ошибок имеет место также и при применении коллективных операций. Поскольку такие функции должны быть вызваны всеми процессами коммуникатора, то в шаблонах, описывающих ошибки групповых коммуникаций, характерным значением параметра «K» является «n(MPI_COMM_WORLD)».

Распространенным классом ошибок, как при использовании коммуникаций точка-точка, так при групповых операциях, являются ошибки несоответствия. В нижеследующих листингах приведен шаблон для ситуации несоответствия в размере буфера в различных процессах, вызывающих любую коллективную функцию, перераспределяющую данные (листинг 13) и любую редукционную операцию (листинг 14).

Листинг 13. Шаблон «Несовпадение размеров буферов в разных процессах при коллективной операции (функция, перераспределяющая данные)»

```
Name=Buffer sizes mismatch in different processes during collective operation (Bcast, Gather, Scatter, Alltoall)

1 block

K=n(MPI_COMM_WORLD)

2 block

FA=pA:Coll_data

3 block

F1(2)!=F2(2)
```

Листинг 14. Шаблон «Несовпадение размеров буферов в разных процессах при коллективной операции (редукционная функция)»

Name=Buffer sizes mismatch in different processes during collective reduction operation (Reduce, Scan, Allreduce)

1 block

K=n(MPI_COMM_WORLD)

2 block

FA=pA:Coll_reduc

3 block

F1(3)!=F2(3)

Также, как и в предыдущем шаблоне «дедлок, вызванный циклом операций отправки или получения» здесь используется обозначение «FA=pA», поскольку все процессы, участвующие в ситуации, вызывают одну и ту же функцию. Представленные шаблоны очень схожи между собой, но их нельзя объединить в один, поскольку для первой группы функций ошибка состоит в том, что среди общего множества процессов существуют два, у которых неодинаков второй аргумент, а для второй группы – третий аргумент.

5) Вызов функции отправки типа точка-точка процессом, парный для которого вызвал коллективную операцию.

Следующая ошибка сочетает в себе как коллективную функцию, так и операцию точка-точка. Некоторый процесс (*p1*) вызывает операцию отправки по отношению к процессу *p2*. Процесс *p2* в это время вызывает какую-либо из коллективных функций. Поскольку в большинстве случаев коллективные функции реализуются как блокирующие, то, если операция отправки отработает как синхронная, возникнет ситуация дедлока.

Листинг 15. Шаблон «Вызов функции отправки типа точка-точка процессом, парный для которого вызвал коллективную операцию»

Name=Call of point-to-point function by process, pair of which has called collective operation

1 block

K=2

2 block

F1=p1:Send||Ssend

F2=p2:Recv! && Irecv!

F3=p2:Coll_data||Coll_reduc

3 block

F1(4)=p2

В шаблоне в качестве функции *F1* указаны «MPI_Send» или «MPI_Ssend», поскольку тольку они могут работать как синхронные. Очень важным является также то, что процесс *p2* не должен вызывать «MPI_Recv» или «MPI_Irecv», соответствующий операции отправки, до вызова коллективной операции. Если убрать строку с функцией *F2*, то под шаблон будут попадать вполне корректные программы, где сначала процессы обмениваются сообщениями точка-точка, а затем вызывают коллективную операцию. В блоке условий обязательно нужно прописать в качестве процесса-адресата *p2* для отправки, выполняемой функцией *F1*.

6) Превышение количества передаваемых элементов заданного значения (пользовательская проверка).

Простейшим примером пользовательской проверки является ситуация, когда при вызове некоторой коммуникационной функции некоторый процесс передает большее количество данных, чем на то рассчитывает пользователь.

Листинг 16. Шаблон «Превышение количества передаваемых элементов заданного значения»

Name=Amount of transmited elements more than defined value

1 block

K=1

2 block

F1=p1:Send_any

3 block

F1(2)>15

В шаблоне для примера задана функция семейства «Send_any», а в третьем блоке указано, что количество элементов больше определенного пользователем значения (в данном случае - 15).

7) Коммуникации между процессами, разность номеров которых более единицы при пересылке данных по кольцу (пользовательская проверка).

Предположим, что в пользовательской программе реализуется передача данных по кольцу. Соответственно, каждый процесс может взаимодействовать только с соседними (по номерам отличающимися не более чем на единицу). То есть процесс номер *k* может обмениваться сообщениями только с процессами *(k-1)* и *(k+1)*. Исключение составляют процессы с номерами *0* и *(n(MPI_COMM_WORLD)-1)*, которые могут передавать сообщения друг другу. В такой ситуации коммуникации процессов, номера которых отличаются друг от друга более чем на единицу, являются ошибкой. Для контроля этого пользователь может создать следующий шаблон.

Листинг 17. Шаблон «Коммуникации между процессами, разность номеров которых более единицы при пересылке данных по кольцу»

Name=Communications between processes, which difference of numbers more than 1 during transmit of data by ring

1 block

K=n(MPI_COMM_WORLD)

2 block

FA=pA:Send_any

3 block

Fj(4)=p(j+1), j:=0,K-2

F(K-1)(4)=p1

p2-p1>1

Программа, для проверки поведения которой создан данный шаблон, предполагает передачу каждым процессом данных процессу с номером на единицу большим. Аналогично шаблону «дедлок, вызванный циклом операций отправки или получения», при помощи рекурсии (конструкции «*Fj(4)=p(j+1)*») здесь задается кольцо. Семантическая ошибка определяется последним выражением в блоке условий: «*p2-p1>1*», которое говорит о том, что среди общего множества должны найтись два процесса, номера которых отличаются более, чем на единицу.

ГЛАВА 3. МОДЕЛЬ И АЛГОРИТМЫ ОБНАРУЖЕНИЯ СЕМАНТИЧЕСКИХ ОШИБОК СИСТЕМОЙ АВТОМАТИЗИРОВАННОГО КОНТРОЛЯ КОРРЕКТНОСТИ

3.1 Требования к системе автоматизированного контроля корректности. Анализ требований

В связи с приведенными в первой главе недостатками подходов диалоговой отладки, сравнительной отладки, верификации программы по модели и анализа корректности по трассе для параллельных приложений на кафедре ЮНЕСКО по новым информационным технологиям КемГУ решено было начать разработку собственной информационной системы отладки MPI-программ.

К системе был предъявлен ряд приведенных ниже требований, которые разбиты на функциональные и нефункциональные [51].

Функциональные требования:

1. Минимальное количество действий со стороны пользователя для получения результата от системы отладки, содержащего перечень найденных ошибок.

2. Обнаружение поведения системы, соответствующее шаблонам, составленным при помощи системы описания, представленной во второй главе.

3. Представление результатов анализа в понятном для пользователя виде и дающем полную информацию о типе ошибки и ее локализации в исходном коде.

4. Предоставление пользователю возможности управлять параметрами системы.

Нефункциональные требования:

1. Поддержка кластерной платформы Windows HPC Server 2008.
2. Поддержка реализации MPICH2 стандарта MPI.
3. Поддержка компиляторов Visual C++ compiler и gcc из пакета MinGW.

Анализ выдвинутых требований приводит к ограничениям, которые заведомо должны быть наложены на инструментальное средство.

Так, минимальное количество действий пользователя для того, чтобы система отладки выполнила свою работу, приводит к необходимости использования подхода автоматизированного контроля корректности, поскольку, как было показано в первой главе, остальные методы требуют значительных временных и трудозатрат со стороны пользователя.

Второе функциональное требование означает, что для анализа пользовательской программы система отладки может использовать определенный в стандарте MPI профилировочный интерфейс [78], посредством которого система получает информацию о вызываемых MPI-функциях и благодаря которому получает управление для проведения анализа на возникновение ошибок.

Условие выдачи информации в понятном для пользователя виде означает, что в результате своей работы инструментальное средство должно создать представление найденных ошибок в каком-либо традиционном, привычном для пользователя формате. Поскольку работа с высокопроизводительным вычислительным ресурсом может осуществляться по консольному доступу (telnet, ssh), то в качестве такого формата выбран простой текст (txt), так как, например, работа с html-страницами не может быть адекватно осуществлена в таком режиме. При этом инструментальное средство должно сообщать место локализации ошибки в исходном коде. Но, в связи с тем, что профилировочный интерфейс MPI не

предоставляет информации о номере строки, где была вызвана обрабатываемая функция, то необходимо применение какого-либо специализированного инструментария для этой функции. В качестве такого инструментария может быть применена утилита-препроцессор, описанная ниже.

К нефункциональным относятся те требования к системе отладки, которые не описываются каким-либо вариантом использования, а накладывают ограничения на используемую операционную систему или другое программно-аппаратное обеспечение, описывают внутренние свойства системы и пр.[51]

Целевой платформой для разрабатываемого инструментального средства является Windows HPC Server 2008. Такой выбор обусловлен тем, что в текущее время для этой операционной системы выпускается немного программного обеспечения поддержки высокопроизводительных вычислений, а тем более – свободно-распространяемого ПО. Тем не менее, эта кластерная платформа более легка в настройке и использовании, чем UNIX-решения. К тому же ОС данного семейства установлены на подавляющем большинстве домашних и рабочих компьютеров исследователей, где зачастую производится разработка и отладка параллельных программ, для чего необходимы специальные средства.

Причина того, что инструментальное средство отладки не может работать с произвольной MPI-библиотекой является индивидуальная реализация скрытых (opaque) объектов (группы, коммуникаторы, типы данных), которые фигурируют в MPI-функциях, но их конкретная структура не закреплена в MPI-стандарте. Так, в библиотеке MPICH тип «MPI_Comm» определен как «int», в то время как в OpenMPI это структура с несколькими полями. Поскольку система автоматизированного контроля корректности работает со значениями аргументов MPI-функций, то знание внутреннего представления этих объектов необходимо. В качестве целевой была выбрана MPI-реализация MPICH2 как одна из наиболее широко распространенных и бесплатных. Требование к компилятору обусловле-

но тем, что перевод пользовательского исходного кода в исполняемые файлы должно производить инструментальное средство путем вызова консольной команды, вид которой зависит от конкретного компилятора.

3.2 Моделирование поведения системы автоматизированного контроля корректности

Итак, в ходе анализа требований к инструментальному средству отладки MPI-программ, об устройстве и принципе работы средства сделаны следующие выводы:

- Для обнаружения семантических ошибок в пользовательских программах система использует подход автоматизированного контроля корректности во время исполнения.
- Информацию о вызываемых MPI-функциях система получает посредством профилировочного интерфейса MPI-реализации.
- Отдельным модулем системы является препроцессор исходного кода, посредством которого системе становится известно о месте локализации последней вызванной функции.
- Данные об анализируемых функциях система должна хранить в каких-либо динамических структурах, память из которых освобождается по мере того, как содержащиеся в них объекты становятся ненужными для дальнейшей работы.
- Результатом работы системы является текстовый файл, содержащий список найденных ошибок и информацию о них.

Таким образом, работу системы первично можно представить следующей диаграммой бизнес-процессов (стандарт IDEF0) [34]:

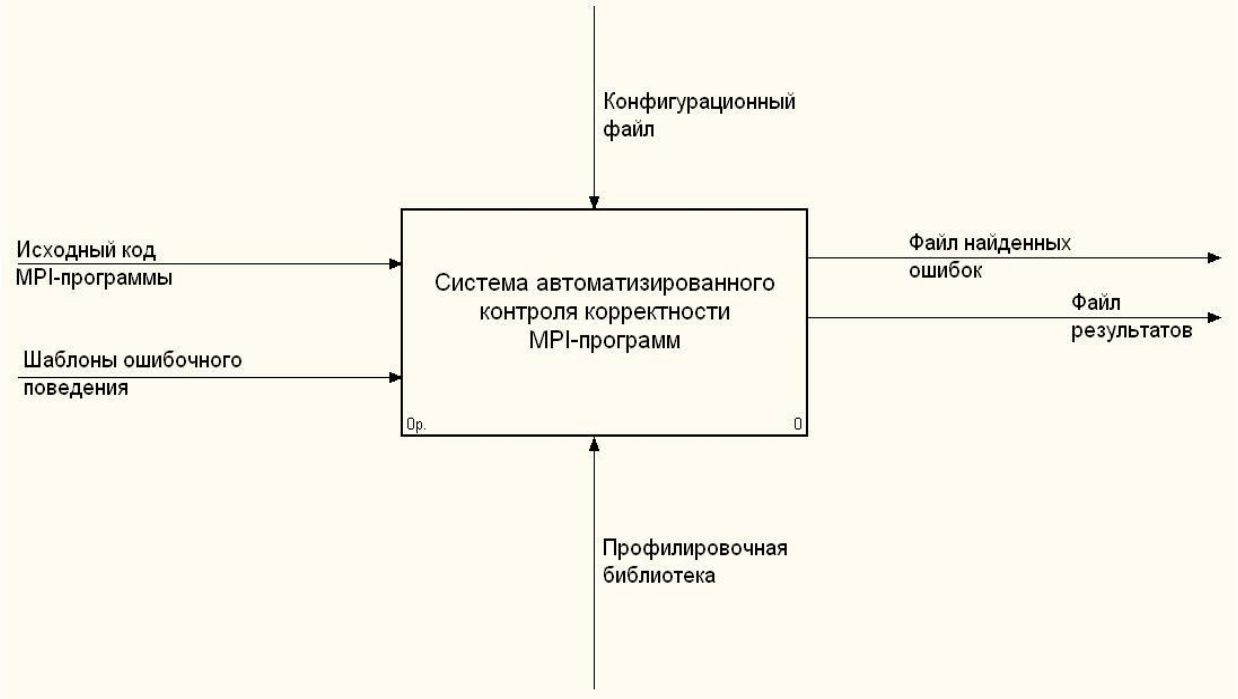

Рисунок 5. Диаграмма бизнес-процессов нулевого уровня

Далее разобраны элементы, представленные на рисунке 5. Так как препроцессор является одним из компонентов системы, она должна принимать файл исходного кода пользовательской программы. Препроцессор вставляет в код необходимые операторы, директивы и пр., в результате чего получается преобразованный файл исходного кода. Поскольку система использует профилировочный интерфейс MPI-реализации, то во время компиляции полученного файла необходимо применить связывание со статической библиотекой. При этом можно поступить двояко: отдать компиляцию с необходимыми опциями на откуп пользователя или включить в разрабатываемую систему некоторую утилиту, которая будет делать это автоматически. Для удобства пользователя решено было создать такую утилиту. После этого шага в системе получен исполняемый файл, который можно запустить на высокопроизводительной вычислительной системе. Вполне логичным было решение также внести этот запуск в функционал утилиты.

Добавление конфигурационного файла, указанного на диаграмме в качестве управления, обусловлено тем, что пользователь должен иметь возможность управлять автоматизированными действиями системы. Например, поскольку компиляцию производит сама система, то пользователь должен иметь возможность задать имя получаемого исполняемого файла. Кроме того, имя результирующего текстового файла с найденными ошибками также может быть задано пользователем. В последующем будут определены другие параметры, входящие в данный конфигурационный файл.

На рисунке 5 в качестве выходного параметра системы обозначен также файл результатов. Это связано с тем, что система отладки управляет запуском параллельной программы (в утилиту внесена соответствующая команда) и обрабатывает вызываемые MPI-функции. На выходе этого процесса получается файл найденных ошибок и файл результатов.

Следующим важным моментом в структуре и принципах работы системы является то, как и где будет проводиться анализ на возникновение ошибок. В первой главе было обозначено, что для этих целей в системе MARMOT существует выделенный сервер-анализатор, запускаемый вместе с другими процессами MPI-программы при помощи команды mpirun или mpiexec, но только выполняющий совершенно другую работу, чем MPI-процессы пользовательского приложения. В создаваемой системе решено было применить тот же подход с той лишь разницей, что сервер-анализатор не является MPI-процессом, а, значит, коммуникации между ним и процессами параллельной программы происходят не через MPI-функции, а посредством более низкоуровневого функционала библиотеки WinSocks.

Причиной введения в архитектуру системы процесса-сервера является то, что глобальные ошибки могут быть обнаружены только при помощи анализа функций, вызванных в различных MPI-процессах (из определения глобальной

ошибки). Схема обнаружения может быть следующей: во время работы параллельной программы процессы отправляют информацию о вызываемых MPI-процедурах процессу-серверу, который помещает эти данные в определенные структуры и по ходу работы проводит анализ этих структур. По результатам анализа сервер производит заключения о появлении в программе глобальных ошибок. Поскольку с процессом-сервером (далее будет применяться термин «сервер-анализатор») необходимо обмениваться информацией множеству MPI-процессов, которые могут располагаться на различных узлах вычислительного кластера, то в качестве инструмента межпроцессного взаимодействия [48] были выбраны windows-сокеты (Windows Sockets API). В связи с этим необходимо было определить сетевые порты, по которым будет производиться взаимодействие между сервером и MPI-процессами. Выбор этого параметра работы системы также решено было возложить на пользователя, а не задавать статическим образом, поскольку на сетевом оборудовании кластера могут быть установлены некоторые ограничения, не позволяющие передавать информацию по всем портам, кроме ограниченного числа. В связи с этим в конфигурационный файл также был включен данный параметр.

На основе приведенного выше анализа можно детализировать диаграмму нулевого уровня (рисунок 5). На рисунке 6 изображена функциональная модель системы автоматизированного контроля корректности, на которой отражены описанные выше этапы работы.

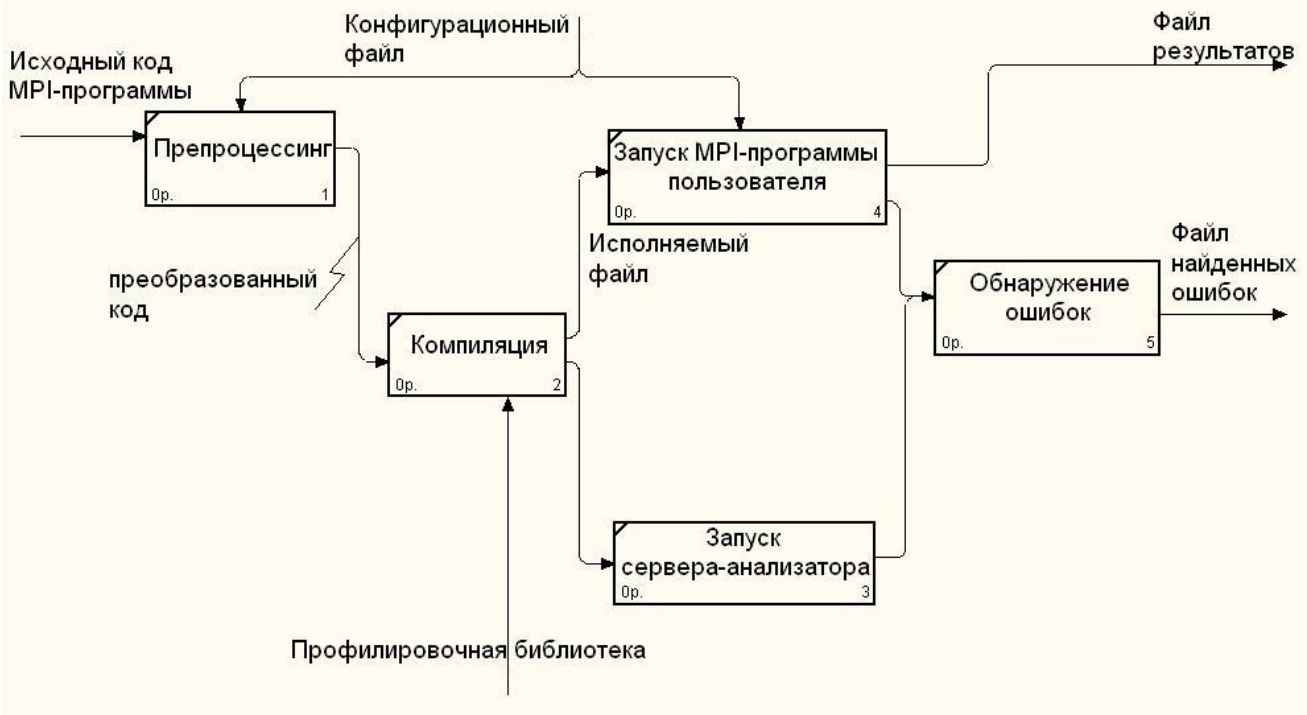

Рисунок 6. Диаграмма бизнес-процессов первого уровня

3.3 Архитектура и основные структуры данных

В данном пункте диссертации подробно разобраны основные архитектурные элементы системы автоматизированного контроля корректности и базовые принципы работы этих элементов.

3.3.1 Конфигурационный файл

Пользователь управляет работой системы посредством установки значений в конфигурационном файле. Данный файл имеет следующее содержание:

Листинг 18. Структура конфигурационного файла (MDS.cfg)

```
Source_files =
Proc_num =
Computelist =
Analyze_server =
Port_data =
```

Port_manage =

Executable_file =

Error_file =

Templates_path =

ProfileLib_path =

Preprocessor_path =

Compiler_type =

Compiler_dir =

MPICH_dir =

Working_dir =

Additional_args =

- *Source_files* — файлы исходного кода, составляющие пользовательскую MPI-программу.
- *Proc_num* — количество процессов параллельной программы.
- *Compute_list* — список имен или ip-адресов вычислительных узлов кластера, на которых будет запущена MPI-программа пользователя.
- *Analyze_server* — имя или ip-адрес узла кластера, играющего роль сервера-анализатора.
- *Port_data* — TCP-порт, через который устанавливается соединение сервера-анализатора с вычислительными узлами кластера для передачи аргументов MPI-функций.
- *Port_manage* — TCP-порт для передачи управляющей информации от сервера служебным потокам на вычислительных узлах. Введение этого параметра обусловлено тем, что в некоторых случаях при работе системы возникает необходимость в передаче асинхронного сообщения

от сервера MPI-процессу (например, команды о срочном завершении работы). Такие сообщения принимают дополнительные потоки. Для передачи этой служебной информации в систему был введен еще один сокет, работающий по порту *Port_manage*.

- *Executable_file* — имя исполняемого файла, получаемого в результате компиляции пользовательской программы. Данный файл в дальнейшем запускается на кластере для анализа системой.

- *Error_file* — имя файла, в который после окончания работы программы помещаются все записи о найденных ошибках с сервера-анализатора.

- *Templates_path* — путь к каталогу с файлами шаблонов.

- *ProfileLib_path* — полный путь к статической профилировочной библиотеке, подключаемой к программе пользователя.

- *Preprocessor_path* — полный путь к препроцессору пользовательского кода.

- *Compiler_type* — тип компилятора (mingw или Visual C++). Данный параметр может принимать 2 значения: gcc и msvc.

- *Compiler_dir* — путь к директории, содержащей компилятор, заголовочные файлы и статические библиотеки.

- *MPICH_dir* — путь к директории, в которую установлена библиотека MPICH2.

- *Working_dir* — рабочая директория, в которую будет помещен исполняемый файл, файл найденных ошибок и служебные файлы.

- *Additional_args* — параметры командной строки, с которыми должна быть запущена MPI-программа. Так же, как и в консольной команде запуска приложения, данные аргументы следуют через пробел.

Стоит заметить, что если профилировочная библиотека и препроцессор расположены в том же каталоге, что и утилита запуска, то указывать полный путь необязательно – можно указать только имена соответствующих файлов.

Таким образом, текст файла *MDS.cfg* может быть следующим:

Листинг 19. Пример конфигурационного файла

```
Source_files = D:\myprog\calc.cpp D:\myprog\grid.cpp
Proc_num = 3
Computelist = n1, n2, n3
Analyze_server = n1
Port_data = 27015
Port_manage = 27016
Executable_file = mympiprog.exe
Error_file = mympiprog_logic_err.txt
Templates_path = C:\ Program Files\MDS\MDS_templates
ProfileLib_path = C:\Program Files\MDS\lib_profile\libprofile.lib
Preprocessor_path = C:\Program Files\MDS\preprocessor\preprocessor.exe
Compiler_type = msvc
Compiler_dir = C:\Program Files\Microsoft Visual Studio 9.0
MPICH_dir = C:\Program Files\MPICH2
Working_dir = \\n1\sharefolder\vlasenko_progs\myprog
Additional_args =
```

3.3.2 Утилита запуска

Для получения результата в виде файла, содержащего список найденных ошибок, пользователю необходимо только заполнить конфигурационный файл и запустить описанную далее утилиту, передав ей *MDS.cfg* в качестве аргумента. Считывая параметры конфигурационного файла, утилита запуска выполняет следующую последовательность действий.

Посредством вызова командной строки с передачей в качестве аргумента консольной команды утилита запускает препроцессор. Получив в качестве ключа имена файлов исходного кода (*source_files*), препроцессор вводит в них дополнительные операторы и выдает файл с преобразованным кодом, который помещается в рабочую директорию – ту, которая указана параметром «Working_dir» конфигурационного файла.

Далее утилита запуска формирует строку компиляции. При этом принципиальное значение имеет то, какой компилятор выбран (из среды Visual Studio или пакета компиляторов MinGW), что указывается параметром *Compiler_type*. Данные компиляторы имеют различные ключи, и строки вызова компилятора и линковщика будут существенно различаться. Проанализировав значение переменной окружения PROCESSOR_ARCHITECTURE, утилита определяет, в какой среде происходит работа – 32-х или 64-х битной операционной системы. В строку компиляции записываются пути к необходимым заголовочным файлам и устанавливаются флаги компилятора. Одним из них является флаг, говорящий о том, что результатом работы компилятора должен стать объектный файл программы. Строка вносится в bat-файл компиляции, который также помещается в рабочую директорию. После этого формируется строка для вызова линковщика, где устанавливается имя исполняемого файла параллельной программы и подключаются необходимые статические библиотеки, в том числе – профилировоч-

ная библиотека, входящая в состав системы. Одним из ключей задается разрядность операционной системы. Эта строка также вносится в bat-файл компиляции.

Записав в bat-файл 2 строки для компиляции и линковки соответственно, утилита запускает данный файл. Результат каждого процесса записывается в свой текстовый файл, также помещаемый в рабочую директорию. Утилита запуска анализирует вывод компилятора и линковщика. В случае, если компиляция прошла неуспешно, то пользователю необходимо сообщить о синтаксических ошибках в программе. Но, поскольку компиляции подвергается преобразованная программа, то указывать на ошибки в этом исходном коде было бы неверным. В связи с этим следующим шагом утилита запускает компиляцию оригинальной пользовательской программы с выводом результатов в файл err_compilation_*current_time*, где current_time – текущее время в формате год_месяц_число_часы_минуты_секунды. На этом работа утилиты завершается.

Если ошибок компиляции не возникло, то утилита запускает процесс сервера-анализатора. Поскольку этот процесс должен быть запущен на определенном узле (*Analyze_server*), то требуется использовать какой-либо механизм удаленного запуска. Для удаленного запуска была выбрана системная утилита *clusrun*, входящая в среду Windows HPC Server 2008. Параметрами вызова являются имя узла (*Analyze_server*), рабочая директория, порты *Port_manage* и *Port_data*, а также имя файла с найденными семантическими ошибками (*Error_file*). При этом требуется, чтобы на том вычислительном узле, который будет исполнять роль сервера-анализатора, полный путь к исполняемому файлу сервера был включен в переменную окружения PATH. Если работа с системой осуществляется в другой ОС семейства Microsoft Windows, то утилита сообщит о том, что запуск процесса-сервера требуется произвести «вручную».

Последним действием, выполняемым утилитой, является запуск MPI-программы пользователя. Для этого используется утилита mpiexec, входящая в

подавляющее большинство реализаций MPI-стандарта. Этому предшествует формирование файла с именами вычислительных узлов кластера, на которых будет работать параллельная программа. Данные имена извлекаются из значения параметра *Computelist* конфигурационного файла. Затем формируется строка с вызовом *mpiexec*, где в качестве параметров указываются:

- количество MPI-процессов;
- файл с именами хостов кластера для запуска параллельной программы;
- имя сервера-анализатора через переменную окружения MDS_SERVER;
- порты *Port_manage* и *Port_data* через переменные окружения MANAGE_PORT и DATA_PORT соответственно;
- дополнительные аргументы пользовательской программы, указанные в строке *Additional_args* конфигурационного файла.

Данная строка записывается в новый bat-файл, который затем запускается и параллельное приложение начинает работать, обмениваясь информацией с сервером-анализатором. Если утилита работает не в среде Windows HPC Server 2008, то запуск bat-файла не производится. Это делается преднамеренно, поскольку запуск сервера-анализатора должен предшествовать MPI-программе, а выполнить это действие должен пользователь. Тем не менее, формирование данного bat-файла является небесполезным, поскольку после запуска сервера-анализатора, пользователю остается только загрузить этот файл и не нужно прописывать все параметры *mpiexec* самостоятельно.

На этом этапе рассмотрения требований к системе и ее структуры можно привести диаграмму вариантов использования (рисунок 7), отражающую возможные действия основных действующих лиц по отношению к системе.

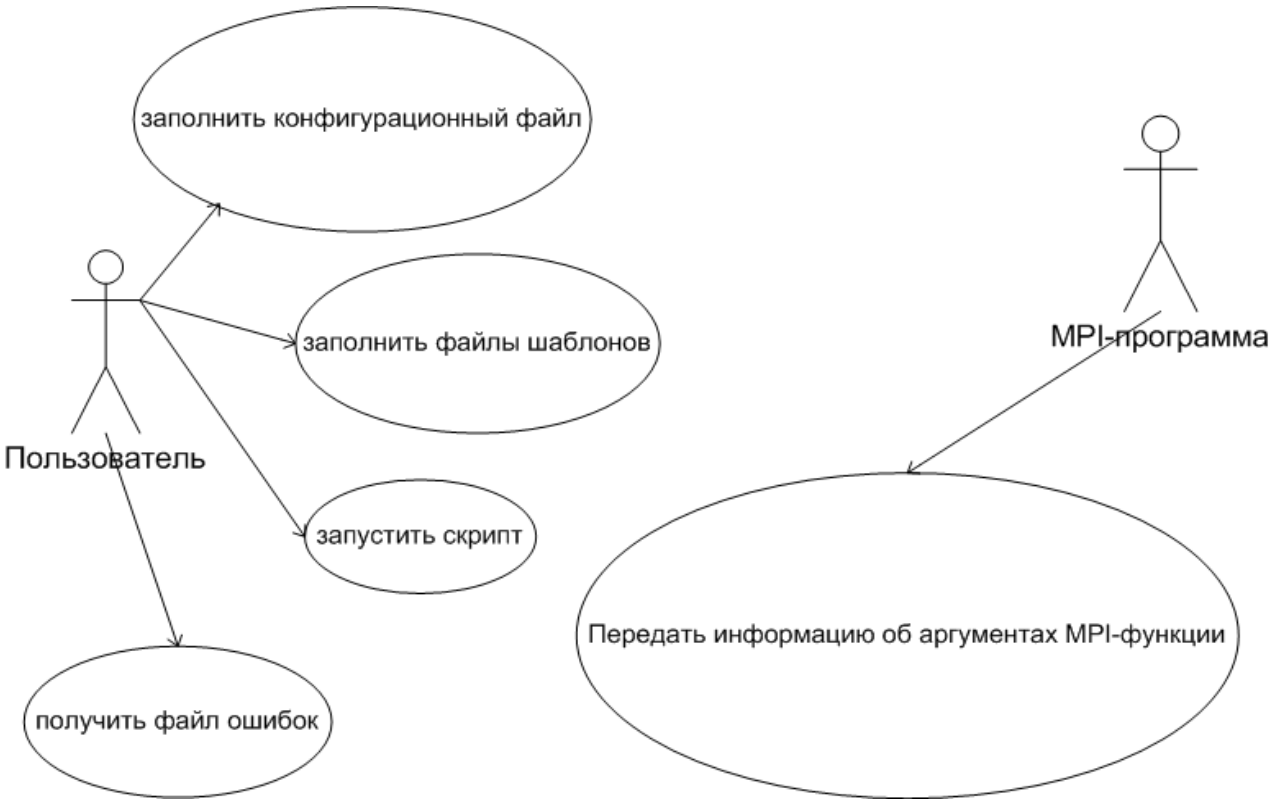

Рисунок 7. Модель вариантов использования

Главным действующим лицом является пользователь. Поскольку интерфейс системы достаточно прост, что обусловлено применением подхода автоматизированного контроля корректности, то количество вариантов использования у данного действующего лица не велико. В качестве второго действующего лица можно выделить MPI-программу, как отдельное приложение, взаимодействующее с системой. Поскольку данное взаимодействие осуществляется через профилировочную библиотеку, то единственным вариантом использования является передача параметров вызываемых операций системе.

3.3.3 Препроцессор

Препроцессор представляет собой небольшое консольное приложение. Его аргументами являются имена файлов исходного кода, которые препроцессор обрабатывает один за другим и на выходе выдает преобразованные файлы,

названия которых формируются путем прибавления к имени строки *«_MDSprep»*, например calc_MDSprep.cpp.

Как было обозначено выше (см. пункт 2.2), препроцессор введен в систему для получения «информационного контекста» вызываемых MPI-функций (имени файла и строки кода). Поэтому в первой строке каждой функции сразу после фигурной скобки, открывающей тело функции, препроцессор вставляет оператор “strcpy(vlas::file_name, “*FileName*”);” (в дальнейшем он будет обозначаться как оператор1), где:

- *vlas* — пространство имен, определенное в профилировочной библиотеке, которая при компиляции будет подключена к программе пользователя. Данное пространство имен введено для безопасности функционирования системы – если в профилировочной библиотеке и в пользовательской программе глобальные переменные будут иметь одинаковое имя, то может возникнуть конфликт.
- *file_name* — глобально определенный массив, хранящий имя файла, содержащего данную функцию.
- *FileName* — имя исходного файла, который обрабатывается препроцессором в данный момент

Добавление данной строки в начало каждой функции в пределах одного файла необходимо в связи с тем, что по ходу работы программы функция, реализованная в одном файле, может вызывать функции из других файлов. Поэтому, если в переменной *vlas::file_name* будет находиться старое значение, то при обращении к другой функции, в которой существует семантическая ошибка, сообщение о ней в Error_file будет содержать неверную локализацию.

Весь программный код в обрабатываемых файлах просматривается построчно и в начале каждой строки, содержащей хотя бы один раз подстроку «MPI_», вставляется оператор «vlas::num_str = num_str» (в дальнейшем он будет обозначаться как оператор2), где num_str – номер текущей строки. Операторы вставляются только в такие строки, потому что система автоматизированного контроля корректности анализирует только MPI-функции. Вставка этих операторов в другие строки повлечет лишь увеличение накладных расходов на программу пользователя, поскольку дополнительные операторы требуют дополнительного процессорного времени.

При этом препроцессор учитывает ряд синтаксических правил языка C/C++. Ниже приведены некоторые проверки, которые выполняет препроцессор перед вставкой в код обозначенных операторов:

1) Строка кода должна быть не закомментирована, то есть в ее начале (после, может быть нескольких символов пробела или табуляции) не должно быть символов «//», а также она не должна находиться внутри блока «/*…*/». Если закомментирована только часть строки, то подстрока «MPI_» ищется только до начала комментария.

2) Если строка начинается со слова «else», то вставлять оператор в ее начало неверно, потому как будет нарушена конструкция условного оператора «if - else if - else». Поэтому оператор2 вставляется первым в дальнейшей последовательности операторов (в блоке «else if» или «else»). При этом учитывается следующее правило 3.

3) Если оператор2 необходимо вставить в сложный блок (if, else, else if, for, while, do), то проверяется наличие фигурных скобок для ограничения этого блока. Если фигурных скобок нет, значит блок содержит только один оператор, и чтобы вставить оператор2 необходимо поставить открывающую фи-

гурную скобку, внести оператор 2, затем присутствующий оператор и после символа «;» - закрывающую фигурную скобку.

3.3.4 MPI-процессы

Выше уже было упомянуто, что, как и в других инструментальных средствах автоматического контроля корректности в разработанной системе к MPI-программе прилинковывается профилировочная библиотека, перехватывающая вызовы MPI-функций.

При обращении в пользовательской программе к первой же процедуре, которая обязана присутствовать в любом MPI-приложении – MPI_Init, устанавливается взаимодействие с сервером-анализатором по порту *Port_data*, переданному MPI-программе в качестве аргумента. Затем процесс отправляет через созданный сокет свой номер (ранг) и обратно ожидает поступления сообщения «GO!». После этого устанавливает соединение по порту Port_manage и отправляет сообщение с номером процесса на сервер через этот сокет. В этой же функции происходит прием от сервера имен функций, которые необходимо профилировать и информации о необходимости передачи контрольной суммы. Список данных функций формируется сервером во время парсинга файлов шаблонов. Наконец, на сервер делается еще одна посылка с системным идентификатором MPI-процесса (PID) и их общим количеством.

В дальнейшем, при вызове коммуникационных функций (в случае, если от сервера передано сообщение о необходимости вычисления CRC-кода для данной операции) вычисляется контрольная сумма буфера передачи (приема) по алгоритму CRC16 [47] и делается по 2 посылки на сервер: до обращения к PMPI-функции и после. Для функций типа точка-точка передаются следующие данные, упаковываемые в одну текстовую строку:

— сокращенное имя операции;

— номер вызвавшего процесса в коммуникаторе MPI_COMM_WORLD;

— процесс-партнер (приемник для операций типа «Send» и отправитель для операций типа «Recv»);

— число элементов буфера приема/передачи;

— тип данных MPI;

— тэг операции;

— (опционально) CRC-код;

— «запрос обмена» (request) для неблокирующих операций и определенную в системе константу NULL_REQUEST для блокирующих;

— имя файла, из которого вызвана данная функция (vlas::file_name);

— номер текущей строки кода (vlas::num_str);

— флаг завершенности операции («f», если посылка на сервер отправляется до вызова PMPI-функции и «t» - если после).

При вызове функции MPI_Finalize на сервер передается завершающее сообщение, которое говорит о том, что работа данного процесса завершена нормальным образом.

В пределах каждого из MPI-процессов запускается по одному служебному потоку, задача которого – «прослушивать» порт *Port_manage*, принимать сигналы от сервера и выполнять соответствующие действия. Соединение по порту Port_manage проходит в асинхронном режиме

Работа MPI-процесса завершается по наступлению одного из следующих событий (при этом не рассматриваются случаи снятия процесса пользователем

или завершение операционной системой ввиду какой-либо внештатной ситуации):

— нормальный выход после вызова MPI_Finalize;

— принудительное завершение служебным потоком в результате приема соответствующего сигнала от сервера-анализатора.

3.3.5 Сервер-анализатор

Начав свою работу, сервер-анализатор осуществляет парсинг файлов шаблонов из директории, указанной в конфигурационном файле. При этом создаются и заполняются соответствующие содержимому шаблонов структуры данных (подробнее эти структуры описаны ниже). Во время парсинга выполняются следующие проверки корректности составленных шаблонов:

— Первая значащая строка (за исключением пустых и строк-комментариев) должна начинаться с символов «*Name*=» (между «*Name*» и «=» допустимы пробелы).

— Второй значащей строкой должна быть «*1block*» (между «*1*» и «*block*» допустимы пробелы).

— Третья значащая строка должна иметь вид «*K op i*», где op – один из символов «>, <, =», а *i* – натуральное число или «*n(MPI_COMM_WORLD)*».

— После этого должна следовать строка «2block» (аналогично, между «2» и «*block*» допустимы пробелы).

— Строки второго блока должны иметь вид «*Fi=pj:Func*», где *i* и *j* – натуральные числа или «*A*», а «*Func*» - либо одна из поддерживаемых системой MPI-функций без префикса «MPI_» (возможно со знаком «!»), либо комбинация таких функций через «&&» или «||».

— Функции во втором блоке должны вводиться, начиная с номера «1» и следуя подряд. То есть, если заданы функции «*F1*» и «*F3*», а «*F2*» отсутствует, то данный шаблон некорректен. Исключение составляют случаи, когда во втором блоке вводится «*FA*».

— Также во втором блоке номер процесса, который должен вызывать функцию «*pi*» не должен превышать общее число процессов «*K*». Такая проверка выполняется, если первый блок имеет вид «*K=j*» или «*K<j*», где *j* – натуральное число.

— В третьем блоке каждая строка должна иметь вид «*Fi(j) op R*», где j – натуральное число, *i* – также натуральное число или арифметическое выражение в круглых скобках, зависящее от K; *op* – один из символов «>, <, =, !=»; *R* – правая часть условия в виде целого числа, номера процесса (*pk*, где *k* – натуральное число), значения аргумента другой функции (*Fl(m)*, где *l* и *m* – натуральные числа), MPI-макрос – строка из прописных букв и подчерков, начинающаяся с «MPI_». Еще одним возможным видом третьего условия является «*pi op pj*», где *i* и *j* – натуральные числа, *op* – один из символов логического отношения. После этого в строке могут присутствовать только символы «&&» или «||».

Если в шаблоне не соблюдено хотя бы одно из данных правил, сервер-анализатор заносит сообщение об этом в файл ошибок и завершает свою работу и работу MPI-процессов, послав соответствующий управляющий сигнал.

После парсинга файлов шаблонов сервер создает 2 TCP-сокета по портам, номера которых принимает в качестве аргументов запуска, после чего пассивно ожидает подключения от одного из MPI-процессов (достигнувшего соответствующей инструкции в MPI_Init первым). Стоит заметить, что данным процессом может быть любая ветвь параллельной программы. Приняв номер (MPI-ранг в коммуникаторе MPI_COMM_WORLD) данного процесса, сервер порождает

первый поток, служащий для обработки сообщений от MPI-процессов. Основной поток после этого переходит в ожидание, пока переменная, обозначающая количество процессов MPI-приложения не станет отличной от 0. Порожденная нить отправляет сообщение «GO!» по созданному основной нитью сокету, создает соединение по порту *Port_manage* и принимает через этот сокет номер процесса. Затем принимает PID процесса и общее их количество.

После того, как проинициализирована переменная с общим количеством MPI-процессов, основная нить процесса-сервера создает недостающие соединения по портам *Port_data* (общее_число_процессов−1) и запускает столько же потоков.

На этом работа основной нити приостанавливается до того, как завершатся все порожденные потоки. Затем закрываются все сокеты по обоим портам и вызываются процедуры поиска глобальных ошибок, которые требуется выполнять только по окончанию работы параллельной программы (подробнее в следующем пункте диссертации).

Основная работа каждого из порожденных потоков заключена в цикл, содержащий:

— прием данных через сокет, связывающий этот поток с соответствующим MPI-процессом;

— обработку полученной информации в зависимости от типа функции, параметры которой переданы (блокирующая отправка/прием, MPI_Init, MPI_Finalize, функция ожидания завершения неблокирующей операции точка-точка, коллективная операция, сообщение о зависании MPI-процесса в некоторой функции) при этом для некоторых типов функций различаются действия в зависимости от того в какой момент отправлено сообщение на сервер (до PMPI-функции или после).

Условием завершения цикла является прием сообщения о вызове MPI_Finalize или установка переменной принудительного завершения MPI-процесса в единицу.

Одной из основных структур данных на сервере является динамический список, содержащий информацию о ситуациях, возникновение которых необходимо проверить (на рисунке 8 элемент №1). Содержание этого списка соответствует данным, внесенным в файлы шаблонов (для каждой ситуации - количество процессов и их номера; имена MPI-функций; условия на их аргументы; двумерный массив ссылок, значение которого будет разъяснено ниже). Еще один список (на рисунке 8 элемент №2), формируемый из списка шаблонов, предназначен для рассылки MPI-процессам. Каждый его элемент содержит информацию о функции, фигурирующей в каких-либо файлах шаблонов. Для каждой функции составляется список номеров аргументов, которые извлекаются из третьего блока шаблонов. Кроме того, каждый элемент данного списка содержит информацию о номерах MPI-процессов, которые, согласно первому блоку шаблона, должны ее вызывать. Если хотя бы в одном шаблоне, где участвовала функция, не указаны конкретные номера процессов в коммуникаторе MPI_COMM_WORLD, то в данное поле списка записывается макрос «ANY». По окончании парсинга файлов шаблонов каждому MPI-процессу через TCP-сокет отправляются имена функций из этого списка, причем каждая функция отправляется только тем процессам, которые указаны в соответствующем поле данного элемента списка функций для профилирования. Если в этом поле содержится «ANY», то имя данной функции рассылается всем процессам.

Также на сервере требуется хранить информацию о вызываемых программой MPI-функциях для проверки сложившейся ситуации на соответствие шаблонам. Основной структурой данных на сервере, хранящей информацию о вызываемых коммуникационных функциях типа точка-точка, по которой прово-

дится анализ на наличие семантических ошибок, является динамический список (элемент №3 на рисунке 8). Каждый элемент данного списка – структура, основные поля которой приведены далее.

— номер строки, содержащей данную MPI-функцию;

— сокращенное наименование операции;

— число передаваемых/принимаемых элементов (имеется в виду количество элементов буфера – второй аргумент функций типа точка-точка);

— MPI-тип передаваемых/принимаемых данных;

— процесс-приемник сообщения;

— процесс-отправитель;

— тэг сообщения;

— CRC-код буфера;

— объект MPI_Request – для блокирующих функций выставляется равным константе NULL_REQUEST, определенной в приложении-сервере;

— «флаг завершенности» коммуникации. Это переменная типа bool, принимающая значение false, если было принято только первое сообщение о рассматриваемой коммуникации (до вызова PMPI-функции). После приема второго сообщения данный параметр принимает значение true.

В данный список добавляются элементы по мере поступления сообщений от MPI-процессов. Принятая через сокет строка проходит разбор (парсинг), а затем полям пустой структуры присваиваются соответствующие значения. По-

скольку потоки сервера работают независимо, то для синхронизации их действий по модификации списка необходимо применять какой-либо специальный инструментарий, в качестве которого выбраны мьютексы. В зависимости от типа коммуникации поток, обрабатывающий сообщение, производит тот или иной набор проверок на соответствие шаблонам возникшей в программе ситуации. Если никаких ошибок, связанных с коммуникацией, информация о которой занесена в некоторый элемент списка, не найдено и система сделала вывод о том, что при дальнейшей работе параллельной программы с этой коммуникацией больше не может возникнуть, то данный элемент списка удаляется. Таким образом, после того, как все MPI-процессы нормально или аварийно завершили свою работу, в динамическом списке остаются только те элементы, с которыми либо связаны какие-либо из уже найденных ошибок, либо такие, которые могут содержать ошибку, и для ее выявления нужен дальнейший анализ.

В отличие от операций точка-точка, которые имеют практически одинаковый набор аргументов и любая функция отправки может быть сопоставлена любой функции приема, у каждой из коллективных операций свой набор параметров. В связи с этим для операции каждого типа на сервере-анализаторе ведется свой динамический список, содержание структур которого зависит от конкретного типа операции (элемент 4 на рисунке 8). При вызове коллективной функции в некотором процессе по аналогии с операциями точка-точка в данном списке добавляется элемент.

В качестве примера ниже приведено содержание структур для списков функций MPI_Bcast и MPI_Gather.

MPI_Bcast:

— порядковый номер элемента в списке;

— номер строки исходного кода, содержащей данную MPI-функцию;

— число передаваемых/принимаемых элементов (количество элементов буфера);

— MPI-тип передаваемых/принимаемых данных;

— root-процесс, рассылающий данные;

— процесс, вызвавший данную операцию;

— «флаг завершенности» коллективной операции (поле «fin»). Это переменная типа bool, принимающая значение *false*, если сервером было принято только первое сообщение о начале работы рассматриваемой операции (до вызова PMPI-функции). После приема второго сообщения о завершении данный параметр принимает значение true.

MPI_Gather:

— порядковый номер элемента в списке;

— номер строки исходного кода, содержащей данную MPI-функцию;

— число передаваемых элементов;

— число принимаемых элементов root-процессом;

— MPI-тип передаваемых данных;

— MPI-тип принимаемых данных root-процессом;

— root-процесс, принимающий данные;

— процесс, вызвавший данную операцию;

— «флаг завершенности». Данное поле имеет тот же смысл, что и в операции MPI_Bcast.

Для хранения информации об ошибках на сервере используется еще один список (элемент №6 на рисунке 8). Каждый элемент этого списка содержит номер шаблона в соответствующем списке и номера элементов списка операций точка-точка или коллективных, которые соответствуют вызовам функций, повлекших описанную в шаблоне ситуацию. Стоит отметить, что, несмотря на то, что из динамического списка могут удаляться элементы, номера оставшихся не изменяются. Это происходит потому как номер элемента списка - это еще одно поле структуры, которое заполняется при добавлении данной структуры в список и остается неизменным на всем протяжении работы процесса-сервера.

Анализируя элементы динамических списков коммуникаций, потоки сервера не только помещают в массивы номера элементов, которые отвечают коммуникациям, содержащим ошибки определенного вида. Также сервер может воздействовать на процессы параллельного приложения через управляющие сигналы, которые представляют собой сообщения, отправляемые через асинхронные сокеты по порту *Port_manage*.

В конце своей работы, после того, как завершены все MPI-процессы и соответствующие потоки сервера-анализатора, основной его поток проводит анализ на соответствие определенному виду шаблонов. Это те шаблоны, которые содержат условия не вызова некоторых функций в программе вообще. Примерами таких шаблонов являются «Выход без вызова MPI_Finalize», «Неблокирующий прием/передача без последующего вызова MPI_Wait или MPI_Test», «Неосвобождение определенного пользователем типа данных». Поскольку вызов функций, указанных в этих шаблонах с символом «!», может произойти в любой момент до окончания работы параллельной программы, то вывод о том, сработал ли шаблон или нет, можно сделать только после завершения приложения. В качестве материала для анализа также применяются динамические списки коммуникаций.

В результате своей работы сервер-анализатор формирует текстовый файл, куда для каждого шаблона записываются найденные по ходу работы приложения совпадения, и для каждого из них указывается полная информация обо всех функциях, вызов которых повлек данную ошибку. Для этого из списка найденных ошибок извлекаются номера элементов, информация о которых берется из списка сообщений.

По итогам проведенного анализа архитектуры и взаимодействия структурных элементов системы автоматизированного контроля корректности в соответствии с нотацией UML построена совмещенная диаграмма размещения и компонентов (рисунок 8), отражающая расположение основных программных модулей по узлам кластера. На этой же диаграмме отражены основные структуры данных, используемые в системе. Создание конфигурационного файла и исполнение утилиты запуска должно происходить на головном узле кластера, поскольку именно этот узел предоставляет внешний интерфейс взаимодействия с пользователем. Сервер-анализатор запускается на одном из вычислительных узлов кластера (на диаграмме он обозначен как «Вычислительный узел 0»). MPI-программа пользователя запускается на других узлах, на каждом из которых должна быть размещена профилировочная библиотека.

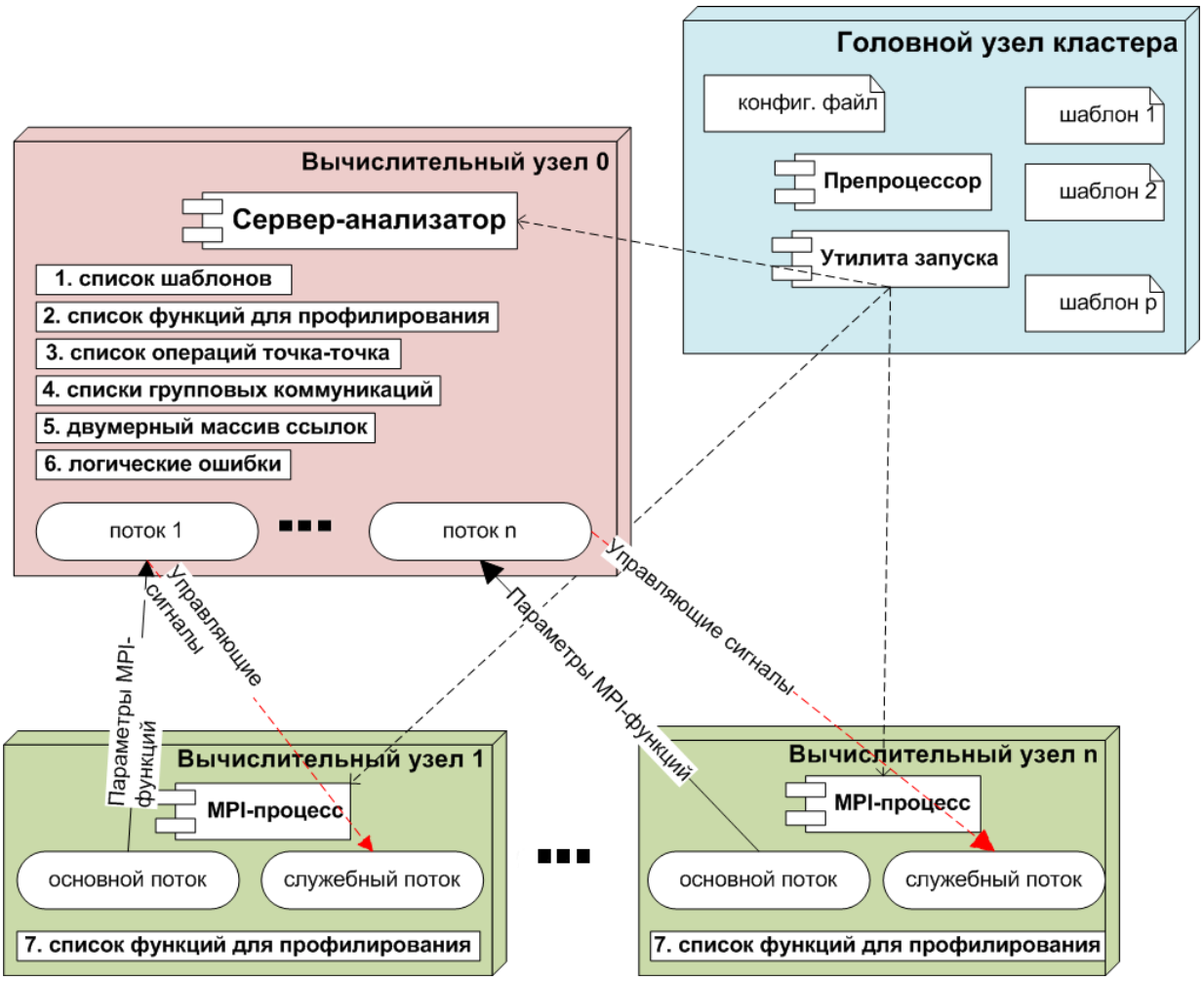

Рисунок 8. Модель размещения и компонентов

Ниже приводится алгоритм одного сеанса работы системы автоматизированного контроля корректности (рисунок 9). Цифры на рисунке соответствуют номерам действий в нижеследующем алгоритме.

1. Пользователь формирует файлы шаблонов.
2. Затем заполняет конфигурационный файл системы, в котором указывает ряд обязательных параметров (исходные файлы своего проекта; число MPI-процессов и др.) и может указать дополнительные параметры (имя узла сервера-анализатора и др.)

3. После этого пользователь загружает утилиту запуска, считывающую параметры из конфигурационного файла и выполняющую ряд действий, первое из которых - запуск препроцессора.
4. Программа-препроцессор вносит в пользовательские файлы исходного кода дополнительные операторы для получения «информационного контекста» вызова MPI-функций.
5. Далее утилита запуска вызывает компилятор для измененных препроцессором файлов пользовательской программы. Ключами компилятора являются параметры, указанные в командном файле.
6. Если компиляция прошла без ошибок, то на указанном в конфигурационном файле вычислительном узле запускается процесс сервера-анализатора с номером TCP-порта в качестве параметра.
7. Последним действием, выполняемым утилитой запуска, является вызов “mpiexec” для пользовательской программы. Ключами данного вызова являются пользовательские аргументы из конфигурационного файла, а также номер TCP-порта для связи с сервером.
8. Сначала сервер-анализатор считывает файлы шаблонов, выполняет проверки шаблонов на корректность и заносит имеющуюся в них информацию в специальные структуры данных.
9. Затем сервер передает MPI-процессам список функций для профилирования, которые фигурируют в файлах шаблонов.
10. Далее во время работы параллельной программы вызовы MPI-функций перехватываются профилировочной библиотекой, и их аргументы передаются серверу.

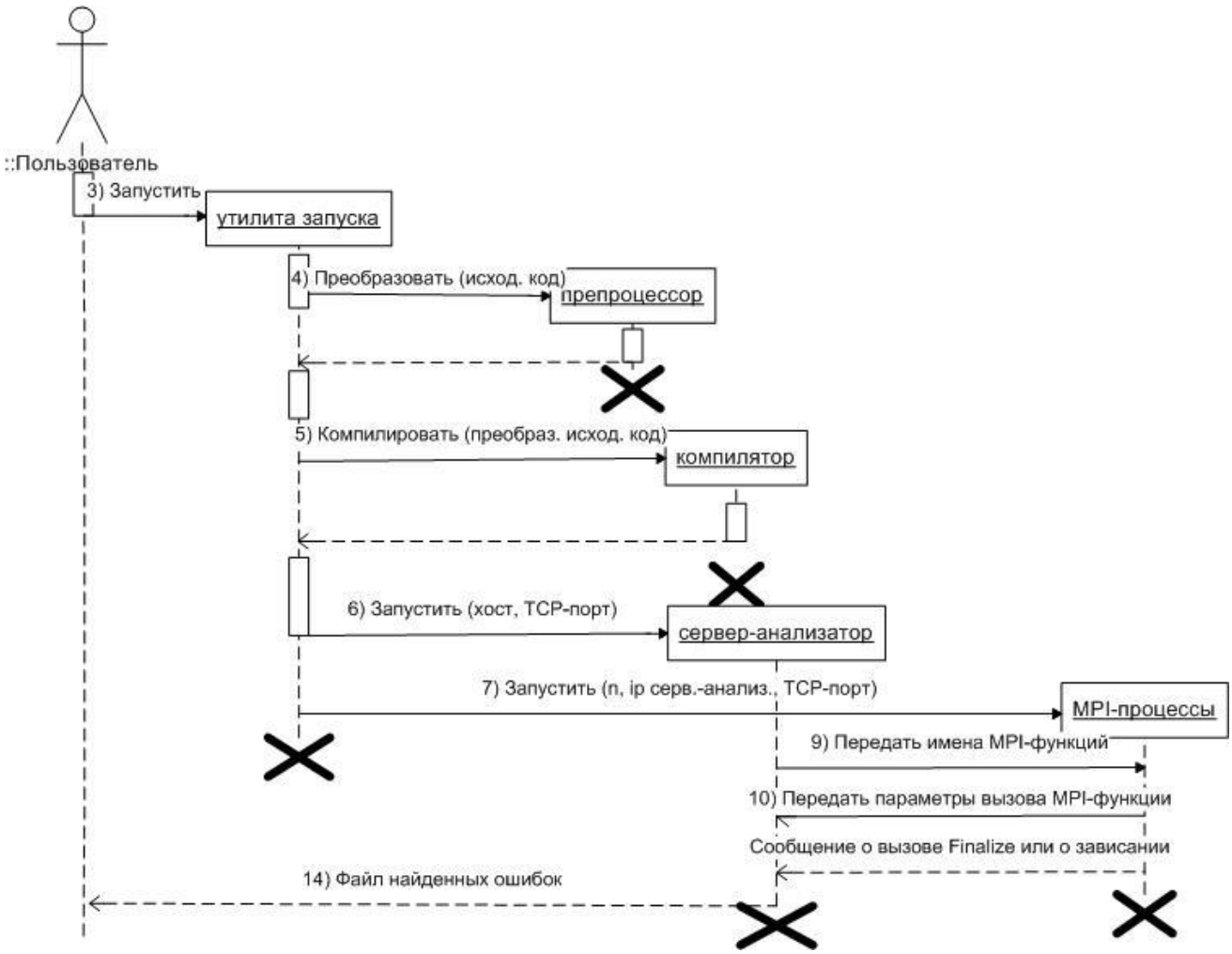

Рисунок 9. Диаграмма последовательности

11. На сервере для каждого подключения от MPI-процесса запускается по одному потоку, принимающему сообщения от узлов кластера и помещающему полученные данные в соответствующие структуры.
12. После каждого приема информации о вызове некоторой MPI-функции сервер проверяет возникшую в программе ситуацию на соответствие шаблонам. При выявлении соответствия в список ошибок заносится подробная информация о совокупности функций, вызов которых стал причиной возникновения шаблонной ситуации.
13. Во время работы может возникнуть ситуация, когда некоторому «стороннему наблюдателю» необходимо вмешаться в работу MPI-процессов (например, когда некоторые ветви параллельного приложения «зависли» во взаим-

ной блокировке). В этом случае сервер-анализатор передает управляющие сигналы служебным потокам процессов через предназначенные для этого TCP-порты.

14. По завершению работы MPI-программы, найденные сервером ошибки, заносятся в результирующий файл ошибок, который предназначается для анализа пользователем.

3.4 Процесс выявления соответствия поведения MPI-программы шаблонам ошибочного поведения

При вызове коммуникационной функции MPI-процесс через TCP-сокет отправляет серверу-анализатору текстовое сообщение, содержащее параметры вызванной операции. В связи с этим в профилировочной библиотеке все целые и вещественные числа, являющиеся аргументами MPI-функции переводятся в формат текстовой строки. Из таких строк формируется общее сообщение, отправляемое на сервер.

В случае коммуникации точка-точка в состав сообщения, передаваемого с MPI-процесса на сервер-анализатор, входят:

— Номер строки в исходном коде MPI-программы, где была вызвана функция.

— Сокращенное имя операции (Send, Recv, Isen, Isse, Bsen и т.д.).

— Ранг MPI-процесса, вызвавшего данную операцию.

— Номер процесса-партнера (приемника или отправителя).

— Число передаваемых/принимаемых элементов.

— Тип данных. Данный параметр передается такой же строкой, какое имя носит тип данных. Например, «MPI_INT».

— Тэг сообщения.

— «Запрос обмена».

— Контрольная сумма сообщения – CRC, в случае, если среди шаблонов присутствует проверка на появление «грязных данных».

— Флаг, указывающий, закончилась ли операция. Под данную строку отводится 2 байта, поскольку достаточно передачи одного символа (*t* или *f* для значений «истина» и «ложь» соответственно).

После получения сообщения с параметрами функции от MPI-процесса поток сервера-анализатора производит разбор (парсинг) данной строки. В случае если функция имеет тип точка-точка, то полученными параметрами заполняется запись, которая становится новым элементом динамического списка, добавляемым в конец. Затем просматривается список функций для профилирования (элемент №2 на рисунке 8) и отыскивается элемент, соответствующий рассматриваемой операции. По найденному элементу списка функций определяются шаблоны, в которых участвует данная функция. Для каждого шаблона сверяется номер вызвавшего процесса с номером процесса, который должен вызывать данную функцию согласно первому блоку шаблона (если первый блок шаблона задан как *«K=m p1=q1 p2=q2 …»*). Если в шаблоне номер процесса не задан (первый блок имеет вид *«K=m»*) или если задан и совпадает с номером вызвавшего, то просматриваются все условия шаблона (третий блок), где участвует данная функция, и выполняются соответствующие проверки.

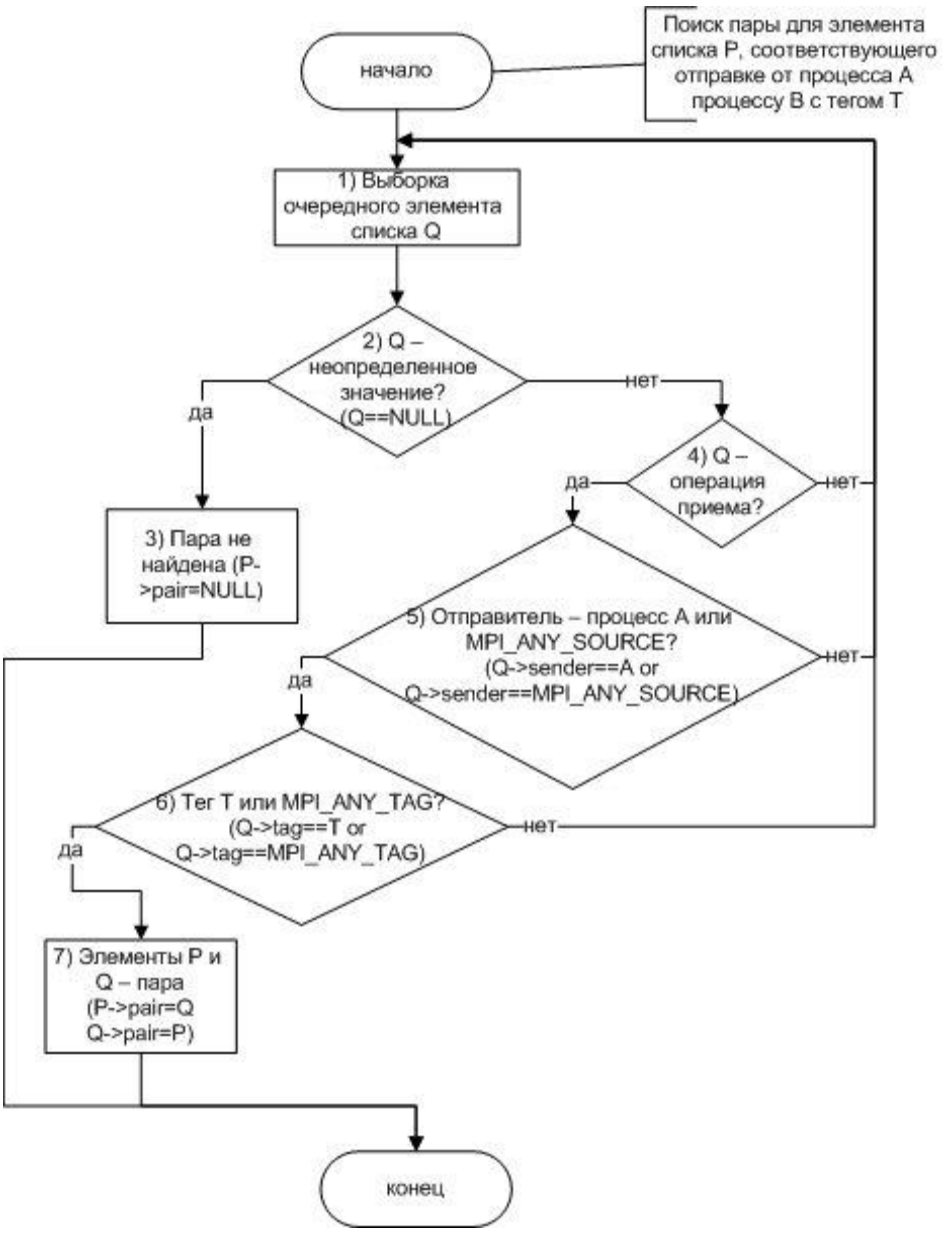

Рисунок 10. Поиск парного элемента при операции приема

Допустим, что в некотором условии данная функция фигурирует как «*Fi(a)*», где *i* и *a* – номера функции в шаблоне и аргумента данной функции соответственно. Если другая часть условия (вторая часть отношения) является константой или макросом, определенным в MPI, то условие проверяется сразу. Если вторая часть – также запись вида «*Fj(b)*», то выполняется проверка на то, не являются ли *Fi* и *Fj* парой Send-Recv. Во время данной проверки список коммуни-

каций точка-точка просматривается с конца к началу и ищется элемент, соответствующий такой функции точка-точка, что:

1) она имеет противоположное направление передачи информации по отношению к вновь добавляемой (то есть, если новая операция – отправка, то искомая функция должна выполнять прием и наоборот – решение 4 на рисунке 10);
2) процессы-отправители и процессы-приемники обеих операций равны или в операции приема фигурирует макрос MPI_ANY_SOURCE (решение 5 на рисунке 10);
3) тэги операций одинаковы или в операции приема фигурирует MPI_ANY_TAG (решение 6).

Поиск именно такого элемента выполняется в связи с тем, что для пользователя важно выявить несовпадение аргументов только для тех пар, которые соответствуют друг другу, то есть данные, переданные функцией *Fi*, должны приниматься функцией *Fj* или наоборот.

После того, как пара найдена, выполняется проверка, заданная в условии шаблона. Другие условия шаблона проверяются аналогичным образом. Если все условия выполняются, то в список семантических ошибок (элемент 6 на рисунке 8) вносится новая запись, где указывается номер шаблона в соответствующем списке и номера элементов списка операций точка-точка, которые соответствуют вызовам функций, повлекших описанную в шаблоне ситуацию. В последующем, по окончанию работы программы, в итоговый файл ошибок записывается подробная информация об ошибке, для чего извлекаются данные из всех трех динамических списков. В случае, когда данный вызов функции не повлек «срабатывания» хотя бы одного шаблона, соответствующий элемент списка операций точка-точка удаляется.

Поскольку групповая функция, согласно MPI-стандарту, должна быть вызвана на всех процессах, входящих в коммуникатор, то для шаблонов с участием групповых функций характерен следующий вид второго блока: «*FA=pA:Coll_data*» или «*FA=pA:Coll_reduc*», где *Coll…* – некоторая коллективная операция. Для анализа MPI-вызовов на соответствие таким шаблонам был введен двумерный массив ссылок {*Fij*} на соответствующие элементы списка вызовов данной функции. Для шаблона со вторым блоком, содержащим запись «*FA=pA…*» создается массив ссылок с количеством столбцов, равным общему количеству запущенных MPI-процессов. Каждая строка массива соответствует группе {*F11, F12, …, F1p*}, {*F21, F22, …, F2p*}, где *Fij* – ссылка на элемент списка функции *Fj*, вызванной в *i*-ый раз. Каждая проверка для шаблона состоит в анализе вызовов функций, входящих в одну группу.

В качестве примера рассмотрим шаблон «Несовпадение кодов редукционной операции в различных процессах». Во всех таких функциях код редукции – это пятый аргумент.

Листинг 20. Шаблон «Несовпадение кодов редукционной операции в различных процессах»

```
Name=Reduction code mismatch in different processes
1block
K=n(MPI_COMM_WORLD)
2block
FA=pA:Reduce||Allreduce||Reduce_scatter||Scan
3block
F1(5)!=F2(5)
```

Предположим, что первым редукционную операцию MPI_Reduce вызвал процесс с номером 3. В списке данной операции появляется элемент с параметрами данного вызова. Затем в первую строку двумерного массива ссылок в по-

ложение 3 записывается ссылка на вновь добавленный элемент списка коллективной операции. Когда другие процессы будут вызывать MPI_Reduce, для этих вызовов также будут заводиться новые элементы в списке операции Reduce, и в первой строке двумерного массива будут появляться соответствующие ссылки. Когда один из процессов (например, пятый) вызовет MPI_Reduce во второй раз, ссылка на этот вызов будет записана во вторую строку массива и т.д. Фрагмент динамического списка функции MPI_Reduce изображен на рисунке 11. Содержимое двумерного массива ссылок для момента, когда пятый процесс раньше остальных вызывает редукционную операцию во второй раз представлено в виде таблицы 3. В двумерный массив записываются ссылки на элементы списка коллективной операции MPI_Reduce. В таблице 3 это изображено номерами элементов. Каждый столбец соответствует своему процессу, а каждая стока – очередному вызову MPI_Reduce. Поскольку пятый процесс первым вызывает операцию во второй раз, то ссылка на соответствующий элемент списка появляется во второй строке первой.

После того, как полностью заполняется строка, а значит, все процессы коммуникатора вызвали коллективную операцию, выполняются проверки условий третьего блока шаблона. Для этого путем полного перебора предпринимается попытка таким образом занумеровать вызовы функций в списке коллективной операции, чтобы выполнялись все условия. В приведенном примере всем элементам списка операции Reduce присваиваются подряд номера 1 и 2, и проверяется условие неравенства пятого аргумента. Если удается найти такую пару, то информация об этих элементах заносится в список логических ошибок. Если для элементов, ссылки на которые составляют одну строку массива, не срабатывает ни один из шаблонов, то все эти элементы удаляются из списка.

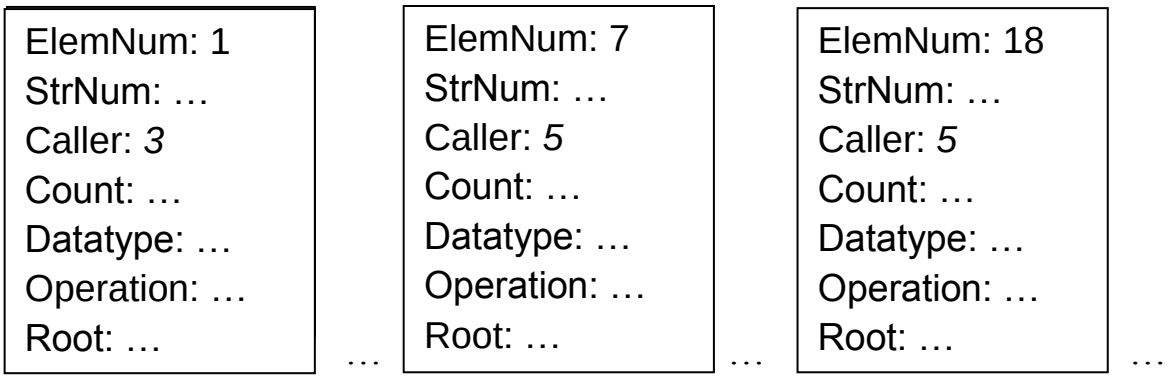

Рисунок 11. Фрагмент списка операции MPI_Reduce

Таблица 3. Двумерный массив ссылок для шаблона «Несовпадение кодов редукционной операции в различных процессах»

№ строки	*…*	*Ссылки процесса 3*	*…*	*Ссылка процесса 5*	*…*	*Ссылка процесса n(MPI_COMM_WORLD)-1*
1		1		7		
2				18		

В общем случае двумерный массив ссылок применяется не только для анализа коллективных функций. Такой массив составляется для всякого шаблона, содержащего более одной функции – то есть, когда условие шаблона не может быть проверено сразу. Если во 2-м блоке шаблона указаны функции *F1,…,Fp,* то для этого шаблона инициализируется двумерный массив ссылок с количеством столбцов, равным p. Каждая строка массива соответствует группе *{F11, F12, …, F1p}, {F21, F22, …, F2p}*, где *Fij* – ссылка на элемент списка функции *Fj*, вызванной в *i*-ый раз. Каждая проверка для шаблона состоит в анализе вызовов функций, входящих в одну группу.

В качестве еще одного примера далее приводится алгоритм обработки шаблона «Неосвобождение производного типа данных» (листинг 7).

Сервер-анализатор, принимая от MPI-процесса информацию о вызове MPI_Type_contiguous, добавляет в список этой функции элемент с параметрами вызова, создает в двумерном массиве ссылок новую строку и в первый элемент строки записывает ссылку на созданный элемент списка. При вызове MPI_Type_free, удовлетворяющего условию «F1(3)=F2(1)», параметры вызова также записываются в новый элемент списка данной функции, а в строку двумерного массива, куда внесена ссылка на соответствующий вызов функции MPI_Type_contiguous – ссылка на этот элемент списка. Поскольку для срабатывания шаблона необходимо, чтобы вызов MPI_Type_free для нового типа данных отсутствовал, то такие строки массива *Fij* не попадают в итоговый файл ошибок. Если по ходу работы MPI-процесса отсутствовал подходящий вызов MPI_Type_free, то после завершения всех процессов сервер-анализатор обнаружит данную ошибочную ситуацию во время финальных проверок. При этом просматриваются все не полностью заполненные строки двумерного массива ссылок, и в файл ошибок записывается подробная информация о тех функциях, которые были вызваны и перечислены функции, вызов которых отсутствует.

В некоторых случаях функция *Fj*, вызванная в i-ый раз может быть включена не в i-ую группу (одним из примеров этому являются функции MPI_Wait и MPI_Test из приведенного шаблона «Повторное использование активного запроса обмена» - листинг 4). Причиной таких исключений из общего правила является то, что в третьем блоке описания шаблона на *Fj* могут быть наложены ограничения, которым не удовлетворяет данный вызов функции.

Система отладки завершает свою работу после завершения всех MPI-процессов и потоков сервера-анализатора, связанных с данными процессами. Сервер считает MPI-процесс завершенным, если поступило сообщение о вызове MPI_Finalize или в течение 30 секунд он не получает от MPI-процесса ни одного сообщения. В последнем случае окончание работы MPI-процесса происходит

ненормально, и перед тем как завершить свою работу сервер записывает в файл найденных ошибок информацию о таких процессах.

После того, как MPI-приложение завершено, сервер записывает в файл обнаруженные шаблонные ситуации. Для каждой шаблонной ситуации, выявленной системой в программе пользователя, в файл ошибок заносится следующее:

— Название шаблона.

— Номер конкретной ситуации, удовлетворяющей шаблону, поскольку таких ситуаций в программе пользователя может быть несколько.

— Информация обо всех функциях, участвующих в ситуации:
  - номер строки и имя файла исходного кода, где была вызвана функция;
  - имя функции;
  - номер MPI-процесса, сделавшего данный вызов;
  - значения аргументов данной функции (конкретный набор аргументов зависит от конкретной операции).

В качестве примера ниже приведен текст файла ошибок для программы, которая содержит дедлок, образованный встречными операциями отправки (MPI_Send) между двумя процессами (листинг 21). Представленный вывод является реакцией системы, обусловленной наличием в списке шаблонов «Дедлока, обусловленного циклом отправок или получений» (листинг 12).

Листинг 21. Файл ошибок для программы, содержащей дедлок

```
DEADLOCK CAUSED BY CYCLE OF SENDS OR RECEIVES:
Situation 1
Function1:
File: D:\mistake_progs\deadlock.cpp
```

String number: 34

Operation: Send

MPI rank: 0

Datatype: MPI_INT

Amount of elements: 1

Receiver rank: 1

Sender rank: 0

Tag: 255

Function2:

File: D:\mistake_progs\deadlock.cpp

String number: 46

Operation: Send

MPI rank: 1

Datatype: MPI_INT

Amount of elements: 1

Receiver rank: 0

Sender rank: 1

Tag: 125

Файл ошибок помещается в сетевую директорию на головном узле кластера.

ГЛАВА 4. ТЕСТИРОВАНИЕ СИСТЕМЫ АВТОМАТИЗИРОВАННОГО КОНТРОЛЯ КОРРЕКТНОСТИ. ПРИМЕРЫ АНАЛИЗА ПРОГРАММ

Для проверки созданной системы на применимость в практической разработке MPI-приложений необходимо было провести ряд тестов на параллельных программах, содержащих семантические ошибки. В данной главе описаны полученные результаты и сравнительный анализ с наиболее известными системами отладки, использующими подход автоматизированного контроля корректности.

4.1 Тестовая база

Тесты проводились на вычислительном кластере, организованном на базе компьютерного класса кафедры ЮНЕСКО по новым информационным технологиям Кемеровского Государственного Университета.

Параметры кластера:

— операционная система – Microsoft HPC Server 2008 64bit;

— количество вычислительных узлов – 5;

— CPU – Pentium Dual-Core E5300 @2.6 Ггц (2 ядра);

— оперативная память одного узла – 2 Гб;

— коммуникационная среда – Ethernet 100 Мб/с.

На данном кластере были установлены 2 реализации стандарта MPI: MPICH2 и MSMPI, входящий в пакет HPC Pack. Для разработки параллельных приложений использовалась интегрированная среда разработки Microsoft Visual

Studio 2008 со встроенным компилятором. Наряду с системой собственной разработки тестированию были подвергнуты инструментальные средства MARMOT_v2.2 и Intel Trace Analyzer and Collector 8.0.3.008.

В конфигурационном файле системы в качестве значения параметра «Templates_path» был прописан путь до директории, содержащей 22 файла шаблонов, перечисленных в пункте 2.2 настоящей работы.

4.2 Используемые тесты

Для тестирования разработанной системы и аналогов был разработан ряд простых MPI-программ, содержащих семантические ошибки. Все описанные ниже программы написаны на языке C++. Для краткости в приведенных ниже листингах программ опущен блок подключения заголовочных файлов.

1) Гонка данных

Листинг 22. Программа с «гонкой данных»

```
void main(int argc, char *argv[]) {
  int rank, nP;
  MPI_Status mpiStatus;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nP);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("\n  rank= %d,  nP= %d",rank,nP);
  int vartosend = rank;
  if(rank == 0) { MPI_Send(&vartosend, 1, MPI_INT, 2, 333, MPI_COMM_WORLD);}
  if(rank == 1) { MPI_Send(&vartosend, 1, MPI_INT, 2, 333, MPI_COMM_WORLD);}
  if(rank == 2) {
```

```
    int vartorecv[2];
    MPI_Recv(&vartorecv[0], 1, MPI_INT, MPI_ANY_SOURCE, 333, MPI_COMM_WORLD,
    &mpiStatus);
    MPI_Recv(&vartorecv[1], 1, MPI_INT, MPI_ANY_SOURCE, 333, MPI_COMM_WORLD,
    &mpiStatus);
    printf("\n I am 2. vartorecv[0]=%d  vartorecv[1]=%d", vartorecv[0], vartorecv[1]);
}
 MPI_Finalize();}
```

В данной программе (листинг 22) нулевой и первый процессы отправляют каждый по сообщению процессу с рангом 2. Каждое из сообщений содержит свой ранг. Второй процесс, в свою очередь, вызывает 2 функции приема с макросом MPI_ANY_SOURCE в качестве процесса-отправителя. В данной ситуации порядок приема сообщений процессом 2 не определен, следовательно, возникают условия «гонки данных». Существенно то, что тэги у всех трех операций одинаковы, поскольку иначе «гонка данных» не будет иметь место.

Проанализировав данную программу, система автоматизированного контроля корректности выявила соответствие шаблону «Межпроцессные гонки данных» (листинг 11), применяя алгоритм поиска, описанный в пункте 3.4. После завершения работы в файл найденных ошибок записывается информация, приведенная в листинге 23.

Листинг 23. Сообщение системы о гонке данных

```
DATA RACE CAUSED BY USING “MPI_ANY_SOURCE” IN RECEIVE FUNCTION
Situation 1:
Function1:
File: D:\mistake_progs\datarace.cpp
String number: 16
```

Operation: Recv

MPI rank: 2

Datatype: MPI_INT

Amount of elements: 1

Receiver rank: 2

Sender rank: MPI_ANY_SOURCE

Tag: 333

CRC16: 45057

Function2:

File: D:\mistake_progs\datarace.cpp

String number: 13

Operation: Send

MPI rank: 1

Datatype: MPI_INT

Amount of elements: 1

Receiver rank: 2

Sender rank: 1

Tag: 333

CRC16: 45057

Function3:

File: D:\mistake_progs\datarace.cpp

String number: 12

Operation: Send

MPI rank: 0

Datatype: MPI_INT

Amount of elements: 1

Receiver rank: 2

Sender rank: 0

Tag: 333

CRC16: 24576

Вносимыми данными являются тип ошибки (извлекается из параметра «Name» шаблона), номер данной ошибки среди всех ошибок этого типа (Situation 1) и «метаданные» каждой функции – номер строки, в которой вызвана данная функция, имя операции без префикса «MPI_», ранг в коммуникаторе MPI_COMM_WORLD, MPI-тип данных, количество элементов, ранги отправителя и получателя сообщения и тэг операции.

Система MARMOT после анализа данной программы в файл вывода записывает следующую строку: «Warning from rank 2(Thread: 0) with Text: WARNING: MPI_Recv: Use of MPI_ANY_SOURCE may cause race conditions!», предупреждая пользователя о том, что применение макроса MPI_ANY_SOURCE в качестве аргумента функции MPI_Recv может привести к возникновению «гонки данных». MARMOT выдает такое предупреждение во всех случаях при применении MPI_ANY_SOURCE и даже в том случае, когда реально условий для появления «гонки данных» нет. Так, система MARMOT была протестирована на программе, в которой один процесс посылает другому сообщение, а второй принимает его с аргументом MPI_ANY_SOURCE. Более в программе никаких действий не совершается и она завершает свое выполнение. В выходном файле, сформированном MARMOT, также присутствовала приведенная строка.

Intel Trace Analyzer and Collector по результатам анализа данной выдал строку «Error checking completed without finding any problems», никак не определив ситуацию «гонки данных».

2) Дедлок

Листинг 24. Дедлок из-за встречных посылок

```
void main( int argc, char *argv[]) {
  int rank, nP;
  MPI_Status mpiStatus;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nP);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("\n  rank= %d,  nP= %d", rank, nP);
  if(rank == 0) {
    int var0 = 998;
    MPI_Send( &var0, 1, MPI_INT, 1, 333, MPI_COMM_WORLD);
    MPI_Recv( &var0, 1, MPI_INT, 1, 222, MPI_COMM_WORLD, &mpiStatus);
     printf("\n I am 0. var0 = %d", var0);
  }
  if(rank == 1) {
    int var1 = 123;
    MPI_Send( &var1, 1, MPI_INT, 0, 222, MPI_COMM_WORLD);
    MPI_Recv( &var1, 1, MPI_INT, 0, 333, MPI_COMM_WORLD, &mpiStatus);
    printf("\n I am 1. var1 = %d", var1);
  }
  MPI_Finalize();}
```

Представленная в листинге 24 программа содержит потенциальный дедлок. Поскольку объем пересылаемых операциями Send данных очень мал, то MPI-реализация отрабатывает как буферизованная, то есть ни один из Send-ов не

ожидает вызова соответствующего Recv-а на принимающей стороне. В связи с этим программа не зависает. Но при переносе программы на другую платформу или реализацию MPI-стандарта ситуация может измениться.

Во время теста система обнаружила соответствие с шаблоном «Дедлок, вызванный циклом операций отправки или получения» (листинг 12). По описанному алгоритму (пункт 3.4) в системе автоматизированного контроля корректности данная ошибка обнаружена на сервере-анализаторе и в файл семантических ошибок записана следующая информация (листинг 25):

Листинг 25. Сообщение о дедлоке

```
DEADLOCK CAUSED BY CYCLE OF SENDS OR RECEIVES
Situation 1:
Function1:
File: D:\mistake_progs\deadlock.cpp
 String number: 19
 Operation: Send
 MPI rank: 1
 Datatype: MPI_INT
 Amount of elements: 1
 Receiver rank: 0
 Sender rank: 1
 Tag: 222
CRC16: 31493
------------------------------
Function2:
File: D:\mistake_progs\deadlock.cpp
 String number: 13
```

Operation: Send

MPI rank: 0

Datatype: MPI_INT

Amount of elements: 1

Receiver rank: 1

Sender rank: 0

Tag: 333

CRC16: 9216

Так же, как для ошибок «гонка данных», в файл выводится тип ошибки, номер ошибки и параметры MPI-функций, вызов которых привел к ее появлению.

MARMOT не в состоянии обнаруживать ситуации потенциальных дедлоков, а выявляет только реальные по определенному тайм-ауту. В связи с этим при проверке программы из листинга 24 данная система не выдает каких-либо сообщений об ошибке.

ITAC производит поиск потенциальных дедлоков путем реализации всех вызовов MPI_Send как синхронных. В связи с этим потенциальный дедлок всегда проявляется как реальный, следовательно, вызвавшие встречные операции посылки процессы зависают. После этого ITAC определяет, что они не выходят из данных вызовов более определенного тайм-аута и выдает сообщение о дедлоке, включая параметры вызванных MPI-функций. Данный подход обладает тем недостатком, что в случае реализации встречных Send-ов как буферизованных, MPI-программа может еще продолжать свою работу и выдать верный результат. Алгоритм обнаружения этого типа ошибок, реализованный в рамках данной работы, не прерывает выполнение программы, а только предупреждает пользователя о возможном проявлении данной ошибки в других условиях.

3) Несоответствие в размере сообщений

В качестве следующего теста рассматриваемых систем отладки была взята программа, содержащая ошибку несоответствия в размере сообщений операций MPI_Send и MPI_Recv (листинг 26).

Листинг 26. Ошибка несоответствия в размере буфера

```
void main(int argc, char *argv[]) {
    int rank, nP;
    MPI_Status mpiStatus;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nP);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("\n  rank= %d, nP= %d", rank, nP);
  if(rank == 0) {
   const int size0 = 4;
   int vartosend[size0];
   for( int i = 0; i < size0; i++) { vartosend[i] = i;}
   MPI_Send( vartosend, size0, MPI_INT, 1, 333, MPI_COMM_WORLD);
 } else  if(rank == 1) {
   const int size1 = 7;
   int vartorecv[size1];
   MPI_Recv(vartorecv, size1, MPI_INT, 0, 333, MPI_COMM_WORLD, &mpiStatus);
   printf("\n source = %d tag = %d\n", mpiStatus.MPI_SOURCE, mpiStatus.MPI_TAG);
   for( int i = 0; i < size1; i++)
     printf("\n from 1: v[%d]=%d ", i, vartorecv[i]);
 }
MPI_Finalize();}
```

В данном случае «срабатывает» шаблон «Несовпадение длин посылаемого и принимаемого сообщений в функциях точка-точка» (листинг 10). В ходе анализа данной программы сервер-анализатор определяет, что MPI_Send, вызванный нулевым процессом и MPI_Recv первого процесса являются парой, но имеют разный размер передаваемого и принимаемого буферов соответственно, о чем заносит сообщение в файл ошибок по завершению работы (листинг 27).

Листинг 27. Сообщение об ошибке несоответствия

```
SENDER'S AND RECEIVER'S MESSAGES LENGTH MISMATCH
Situation 1:
Function1:
File: D:\mistake_progs\mismatch_size.cpp
String number: 12
 Operation: Send
 MPI rank: 0
 Datatype: MPI_INT
 Amount of elements: 4
 Receiver rank: 1
 Sender rank: 0
 Tag: 333
CRC16: 4853
------------------------------
Function2:
File: D:\mistake_progs\mismatch_size.cpp
 String number: 16
 Operation: Recv
```

MPI rank: 1

Datatype: MPI_INT

Amount of elements: 7

Receiver rank: 1

Sender rank: 0

Tag: 333

CRC16: 376

Несмотря на то, что данная ситуация не считается ошибкой по документу MPI Standard (считается ошибкой лишь отправка большего количества данных, чем получение), авторы разработанной системы автоматизированного контроля корректности считают необходимым выдавать пользователю предупреждение об этом. Такие соображения основаны на том, что если пользователь установил в функции MPI_Recv количество принимаемых элементов равным некоторому числу, то, возможно, он рассчитывает на прием именно такого объема данных, и в последующем в программе будет использовать не проинициализированные элементы массива приема, что приведет к появлению «грязных данных».

Системы MARMOT и ITAC никаким образом не диагностируют данную ошибку в представленной программе.

4) Несоответствие в типе данных

Другим типом ошибок, относящихся к ошибкам несоответствия, является несоответствие в типе данных между операциями получения и отправки. Тест, приведенный в листинге 28, содержит именно такой пример некорректного использования стандарта MPI.

Листинг 28. Ошибка несоответствия в типах данных

```
void main(int argc, char *argv[]) {
```

```
    int rank, nP;
    MPI_Status mpiStatus;
    const int size = 4;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nP);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(rank == 0) {
    int vartosend[size];
    for(int i=0; i<size; i++) {
      vartosend[i] = i;
      printf("\n from 0: v[%d] = %d", i, vartosend[i]);
    }
    MPI_Send(vartosend, size, MPI_INT, 1, 333, MPI_COMM_WORLD);
  } else if(rank == 1) {
    double vartorecv[size];
    MPI_Recv(vartorecv, size, MPI_DOUBLE, 0, 333, MPI_COMM_WORLD, &mpiStatus);
    printf("\n source = %d tag = %d\n", mpiStatus.MPI_SOURCE, mpiStatus.MPI_TAG);
    for(int i=0; i<size; i++)
      printf("\n from 1: v[%d]=%lf ", i, vartorecv[i]);
  }
MPI_Finalize();}
```

Поскольку отправляемые данные интерпретируются как целые числа, а принимаются – как действительные, то цикл вывода принятого массива в конце программы выдаст совсем не те данные, которые ожидал пользователь.

Проанализировав данную программу, разработанная система выявляет соответствие с шаблоном «Взаимно недопустимые типы данных в операциях передачи и приема» и выводит в файл найденных ошибок сообщение, аналогичное представленному в листинге 27, то есть тот же набор параметров для пары Send/Recv, но с другим названием группы ошибок, поскольку «срабатывает» другой шаблон. Необходимо заметить, что система в состоянии корректно обрабатывать такие ситуации лишь в случае использования предопределенных типов данных. При использовании производных типов возможно появление «ложных» сообщений об ошибках.

Система MARMOT никаким образом не сообщает о данной ошибке, поскольку не проводит анализ параметров парных операций посылки и получения.

ITAC обнаруживает данную ошибку и выводит сообщение, содержащее ту же информацию, что и разработанная система (листинг 29).

Листинг 29. Сообщение ITAC на несоответствие в типах данных

```
[1] ERROR: GLOBAL:MSG:DATATYPE:MISMATCH: error

[1] ERROR:    Datatype signature mismatch.

[1] ERROR:    Data was sent by process [0] at:

[1] ERROR: MPI_Send(*buf=0x00dcff34, count=4, datatype=MPI_INT, dest=1, tag=333, comm=MPI_COMM_WORLD)

[1] ERROR: main (D:\mistake_progs\mismatch_datatype.cpp:24)

 [1] ERROR: Data was received by process [1] at:

[1] ERROR: MPI_Recv(*buf=0x00dcff58, count=4, datatype=MPI_DOUBLE, source=0, tag=333, comm=MPI_COMM_WORLD, *status=0x00dcff18)

[1] ERROR: main (D:\mistake_progs\mismatch_datatype.cpp:30)
```

5) Непарная посылка

Последним тестом, где ошибка вызвана блокирующей функцией типа точка-точка, была программа (листинг 30), содержащая непарную операцию отправки (Send, для которого не было соответствующего Recv-а).

Листинг 30. Непарная посылка

```
void main(int argc, char *argv[]) {
  int rank, nP;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nP);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(rank == 0) {
      int send_var = 100;
      MPI_Send( &send_var, 1, MPI_INT, 1, 333, MPI_COMM_WORLD);
  }
  MPI_Finalize();}
```

Выполняя данную программу, MPI-реализация может запустить MPI_Send как буферизованную посылку, в результате чего ни один из MPI-процессов не «зависнет», но, тем не менее, отправленная посылка не будет принята. Именно так и отработал данный тест при анализе разработанной системой автоматизированного контроля корректности. Программа была завершена нормальным образом, но в файл семантических ошибок сервер-анализатор записал следующее сообщение (листинг 31). В данном случае сработал шаблон «Передача, которой не удалось поставить в соответствие парную операцию».

Листинг 31. Сообщение о непарной коммуникации

```
UNPAIR SEND OPERATION
```

Situation 1:

Function1:

File: D:\mistake_progs\unpair.cpp

String number: 11

Operation: Send

MPI rank: 0

Datatype: MPI_INT

Amount of elements: 1

Receiver rank: 1

Sender rank: 0

Tag: 333

CRC16: 12933

Система сообщает о том, что в программе найдена непарная операция и выдает все параметры данной функции.

Как уже было отмечено, ITAC преобразует все MPI_Send в синхронные коммуникации. За счет этого нулевой процесс зависает в данном вызове, о чем ITAC выдает сообщение после предопределенного тайм-аута (листинг 32).

Листинг 32. Сообщение ITAC о зависании процесса

[0] ERROR: GLOBAL:DEADLOCK:HARD: fatal error

[0] ERROR: Application aborted because no progress was observed for over 0:27 minutes,

[0] ERROR: check for real deadlock (cycle of processes waiting for data) or potential deadlock (processes sending data to each other and getting blocked because the MPI might wait for the corresponding receive).

[0] ERROR: [2] no progress observed for over 0:27 minutes, process is currently in MPI call:

[0] ERROR: MPI_Send(*buf=0x00deff70, count=1, datatype=MPI_INT, dest=1, tag=333, comm=MPI_COMM_WORLD)

[0] ERROR: main (D:\mistake_progs\unpair.cpp:11)

MARMOT, в отличие от инструментального средства компании Intel, не изменяет характер коммуникационной функции и, не обнаружив даже зависания, не выдает никаких предупреждений об имеющейся непарной посылке.

6) Изменение передаваемых в неблокирующей посылке данных

Одной из опасных особенностей неблокирующих функций точка-точка, не свойственных блокирующим аналогам, является возможность «порчи» данных, участвующих в коммуникации, основным потоком MPI-программы. Именно такая программа и была взята для следующего теста рассматриваемых инструментальных средств отладки (листинг 33). Нулевой процесс отправляет достаточно большой объем данных (более мегабайта) первому процессу посредством функции MPI_Isend. Сразу после этого значения первого, среднего и последнего элементов массива изменяются. Это является причиной того, что при выводе данных элементов после приема первым процессом выдаются измененные значения, а не те, которые были отправлены.

Листинг 33. Изменение данных во время неблокирующей посылки

```
void main(int argc, char *argv[]) {
    int rank, nP;
    MPI_Status mpiStatus;
    MPI_Request req;
    const int size = 400000;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nP);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("\n  rank= %d,  nP= %d",rank,nP);
```

```
if(rank == 0) {
    double buftosend[size];
    for(int i=0; i < size; i++) {buftosend[i] = (double) i;}
    printf("\n from 0: v[0] = %lf", buftosend[0]);
    printf("\n from 0: v[%d] = %lf", (int)((double)size/2.0), buftosend[ (int)((double)size/2.0) ]);
    printf("\n from 0: v[%d] = %lf", size-1, buftosend[size - 1]);
    MPI_Isend( buftosend, size, MPI_DOUBLE, 1, 333, MPI_COMM_WORLD, &req);
    buftosend[0] = -7;
    buftosend[(int)((double)size/2.0)] = -7;
    buftosend[size - 1] = -7;
    MPI_Wait( &req, &mpiStatus);
  } else if(rank == 1) {
   double buftorecv[size];
   MPI_Recv( buftorecv, size, MPI_DOUBLE, 0, 333, MPI_COMM_WORLD, &mpiStatus);
   printf("\n from 1: v[0] = %lf", buftorecv[0]);
   printf("\n from 1: v[%d] = %lf", (int)((double)size/2.0), buftorecv[ (int)((double)size/2.0) ]);
   printf("\n from 1: v[%d] = %lf", size-1, buftorecv[size - 1]);
}
MPI_Finalize();}
```

Данный вид ошибок обнаруживается созданной в рамках работы системой автоматизированного контроля корректности путем сравнения контрольной суммы, подсчитанной перед началом работы неблокирующей функции и после приема на другой стороне. Подсчет контрольной суммы производится в профилировочной библиотеке MPI-процессами для коммуникаций точка-точка до вызова соответствующей PMPI-процедуры в связи с тем, что в списке шаблонов присутствует «Изменение данных, участвующих в неблокирующей коммуника-

ции, основным потоком MPI-процесса» (именно в связи с этим пункт «CRC16» присутствует во всех сообщениях о найденных ошибках выше). Информация, записанная в файл найденных ошибок в результате обработки системой данной программой, представлена в листинге 34.

Листинг 34. Сообщение об ошибке «грязных данных» при неблокирующей посылке

```
DATA RACE DUE TO UNBLOCK POINT-TO-POINT FUNCTION
Situation 1:
Function1:
File: D:\mistake_progs\nonblock_change_data.cpp
String number: 19
Operation: Isend
 MPI rank: 0
 Datatype: MPI_DOUBLE
 Amount of elements: 400000
 Receiver rank: 1
 Sender rank: 0
 Tag: 333
 CRC16: 19456
 ------------------------------
Function2:
File: D:\mistake_progs\nonblock_change_data.cpp
String number: 26
 Operation: Recv
 MPI rank: 1
 Datatype: MPI_DOUBLE
```

Amount of elements: 400000

Receiver rank: 1

Sender rank: 0

Tag: 333

CRC16: 45057

Система предупреждает пользователя о том, что данные были изменены во время передачи от отправителя получателю, о чем можно сделать вывод по разным контрольным суммам пары операций.

MARMOT не проводит подобный анализ, поэтому в результирующем файле отстутствуют какие-либо сообщения о данной ошибке.

ITAC обнаруживает данное некорректное поведение программы и выводит следующую информацию (листинг 35), которая содержит тип найденной ошибки и параметры пары функций отправки/получения.

Листинг 35. Сообщение ITAC о появлении «грязных данных»

```
[1] ERROR: GLOBAL:MSG:DATA_TRANSMISSION_CORRUPTED: error

[1] ERROR:    Data was corrupted during transmission.

[1] ERROR:    Data was sent by process [0] at:

[1] ERROR:       MPI_Isend(*buf=0x00ae2b78, count=400000, datatype=MPI_DOUBLE, dest=1,
tag=333, comm=MPI_COMM_WORLD, *request=0x007d5768)

[1] ERROR:       main (D:\mistake_progs\nonblock_change_data.cpp:19)

 [1] ERROR:    Data was received by process [1] at:

[1] ERROR:          MPI_Recv(*buf=0x007d5778, count=400000, datatype=MPI_DOUBLE,
source=0, tag=333, comm=MPI_COMM_WORLD, *status=0x007d5728)

[1] ERROR:       main (D:\mistake_progs\nonblock_change_data.cpp:26)
```

7) Неблокирующая посылка без функции ожидания

Данный тест содержит ошибку управления внутренним объектом MPI-реализации. MPI-стандарт регламентирует, что после вызова любой неблокирующей коммуникационной функции должна следовать операция освобождения «запроса обмена», фигурирующего в данной функции. Данные объекты освобождают функции ожидания завершения неблокирующей посылки/приема (MPI_Wait…-функции), функции теста на завершение с результатом «успех» (MPI_Test…) или операция отмены коммуникации (MPI_Cancel). Нижеследующая программа (листинг 36) не содержит ни одной из перечисленных функций, несмотря на то, что данные от нулевого первому процессу отправляются при помощи MPI_Isend.

Листинг 36. Неосвобожденный «запрос обмена»

```
void main(int argc, char *argv[]) {
    int rank, nP;
    MPI_Status mpiStatus;
    MPI_Request req;
    const int size = 4;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nP);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(rank == 0) {
   int buftosend[size];
   for(int i=0; i<size; i++) {
     buftosend[i] = i;
     printf("\n from 0: v[%d] = %d", i, buftosend[i]);
   }
```

```
    MPI_Isend( buftosend, size, MPI_INT, 1, 333, MPI_COMM_WORLD, &req);
    Sleep( 5000);
  } else if(rank == 1) {
   int buftorecv[size];
   MPI_Recv( buftorecv, size, MPI_INT, 0, 333, MPI_COMM_WORLD, &mpiStatus);
   printf("\n source = %d tag = %d\n", mpiStatus.MPI_SOURCE, mpiStatus.MPI_TAG);
   for(int i=0; i<size; i++)
     printf("\n from 1: v[%d]=%d ", i, buftorecv[i]);
 }
MPI_Finalize();}
```

Система автоматизированного контроля корректности при вызове в программе MPI_Finalize обнаруживает, что, согласно шаблону «Неблокирующий прием/передача без последующего вызова MPI_Wait или MPI_Test» (листинг 6), нулевой процесс не вызвал MPI_Wait или MPI_Test (в двумерном массиве ссылок данного шаблона отсутствует соответствующая ссылка в первой строке), поэтому условие шаблона выполнено. В файл найденных семантических ошибок записывается следующая информация (листинг 37).

Листинг 37. Сообщение системы о неосвобожденном «запросе обмена»

```
UNBLOCK SEND/RECEIVE WITHOUT FREEING REQUEST
Situation 1:
Function1:
File: D:\mistake_progs\nonblock_change_data.cpp
String number: 18
 Operation: Isend
 MPI rank: 0
```

Datatype: MPI_INT

Amount of elements: 4

Receiver rank: 1

Sender rank: 0

Tag: 333

CRC16: 25173

Система MARMOT также отслеживает некорректную работу со внутренними объектами MPI-реализации и по окончании работы выдает следующее сообщение (листинг 38).

Листинг 38. Сообщение MARMOT о неосвобожденном «запросе обмена»

Text: WARNING: MPI_Finalize: There are still 1 active/non-freed Requests!

Listing of information for all remaining Requests:

Information for Resource of type MPI_Request:

created at D:\mistake_progs\nonblock_change_data.cpp line: 18

not yet freed.

Таким образом, MARMOT сообщает лишь строку исходного кода, где имел место вызов неблокирующей функции.

Система ITAC обнаруживает данную ошибку в программе и в поток вывода заносит следующую информацию (листинг 39).

Листинг 39. Сообщение ITAC о неосвобожденном запросе

[0] WARNING: LOCAL:REQUEST:NOT_FREED: warning

```
[0] WARNING:    When calling MPI_Finalize() there were unfreed requests:
[0] WARNING:    1 in this process.
[0] WARNING:    This may indicate that resources are leaked at runtime.
[0] WARNING:    To clean up properly MPI_Request_free() should be called
[0] WARNING:    for each persistent request and MPI_Wait() for normal
[0] WARNING:    requests.
[0] WARNING:    1. 1 time:
[0] WARNING:      MPI_Isend(*buf=0x00deff50, count=4, datatype=MPI_INT, dest=1, tag=333, comm=MPI_COMM_WORLD, *request=0x00deff64)
[0] WARNING:      main (D:\mistake_progs\nonblock_change_data.cpp:18)
```

8) Коллективная операция с различным root-процессом

Данный тест (листинг 40) содержит ошибку при использовании групповой функции широковещательной рассылки (Bcast). В одном из процессов коммуникатора, участвующих в операции, в качестве параметра «корневой процесс» указано другое целое число, чем во всех остальных процессах.

Листинг 40. Широковещательная рассылка с неодинаковым корневым процессом в разных ветвях MPI-приложения

```
void main(int argc, char *argv[]) {
  int rank, nP;
  int rootproc = -1;
  MPI_Status mpiStatus;
  const int length_of_ryad = 5;
  int i = 0;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nP);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
double mass_ryad[length_of_ryad];
if(rank == nP-1)  {
   for(i=0;i<length_of_ryad;i++) { mass_ryad[i] = (double)i/2;}
}
rootproc = nP - 1;
if (rank == 0) { rootproc = nP-2; }
  MPI_Bcast( mass_ryad, length_of_ryad, MPI_DOUBLE, rootproc, MPI_COMM_WORLD);
printf("\n I'm %d: ", rank);
for(i=0; i < length_of_ryad; i++)
  printf(" %4.2lf", mass_ryad[i]);
MPI_Finalize();}
```

Тестовые запуски проводились для трех процессов. Проанализировав строку в двумерном массиве ссылок для шаблона «Задание пользователем разных номеров root-процессов в разных процессах-участниках коллективной операции», после вызова всеми процессами функции MPI_Bcast, сервер-анализатор обнаруживает удовлетворение условию шаблона, в результате чего в итоговом файле найденных ошибок появляется следующее сообщение (листинг 41). Для групповых функций вывод несколько отличается от вывода для операций точка-точка. Поскольку в групповых функциях участвуют множество процессов, то все параметры одной функции выписываются в одну строку, чтобы файл ошибок не был слишком «вытянут» в высоту.

Листинг 41. Сообщение системы о разных root-процессах в одной коллективной операции

```
DIFFERENT ROOT-PROCESS IN DIFFERENT PROCESSES DURING COLLECTIVE OPERATION BCAST
```

Situation 1:

Operation - Bcast, MPI rank - 0, file – D:\mistake_progs\bcast_diff_root.cpp, string - 19, datatype - MPI_DOUBLE, amount of elements – 5, root - 1

Operation - Bcast, MPI rank - 1, file – D:\mistake_progs\bcast_diff_root.cpp, string - 19, datatype - MPI_DOUBLE, amount of elements – 5, root - 2

Operation - Bcast, MPI rank - 2, file – D:\mistake_progs\bcast_diff_root.cpp, string - 19, datatype - MPI_DOUBLE, amount of elements – 5, root - 2

Обо всех ошибках несоответствия в аргументах коллективной операции, в том числе несоответствия в корневом процессе, система сообщает таким же образом, как приведено в листинге 41. Для каждой ошибки несоответствия приводится имя коллективной операции и для каждого процесса коммуникатора – те параметры вызова, в которых может возникнуть несоответствие (тип данных, количество элементов и root-процесс).

Реакция MARMOT на данную программу состоит в выдаче следующей информации (листинг 42).

Листинг 42. Сообщение MARMOT о разных root-процессах в одной коллективной операции

12: Error global message with Text: WARNING: all clients are pending!

Last calls (max. 10) on node 0:

timestamp 3: MPI_Init(*argc, ***argv)

d:\mistake_progs\bcast_diff_root.cpp line: 10

…

timestamp 12: MPI_Bcast(*buffer, count = 5, datatype = non-predefined datatype, root = 1, comm = self-defined communicator)

d:\mistake_progs\bcast_diff_root.cpp line: 19

Last calls (max. 10) on node 1:

timestamp 1: MPI_Init(*argc, ***argv)

D:\mistake_progs\bcast_diff_root.cpp line: 10

…

timestamp 11: MPI_Bcast(*buffer, count = 5, datatype = non-predefined datatype, root = 2, comm = MPI_COMM_WORLD)

d:\mistake_progs\bcast_diff_root.cpp line: 19

Last calls (max. 10) on node 2:

timestamp 2: MPI_Init(*argc, ***argv)

D:\mistake_progs\bcast_diff_root.cpp line: 10

…

timestamp 10: MPI_Bcast(*buffer, count = 5, datatype = non-predefined datatype, root = 2, comm = MPI_COMM_WORLD)

D:\mistake_progs\bcast_diff_root.cpp line: 19

MARMOT обнаруживает лишь то, что все процессы параллельного приложения «зависли» и выдает последние вызовы MPI-функций каждым процессом (до 10 вызовов). В причинах зависания программы пользователь вынужден разбираться самостоятельно. Кроме того, как уже упоминалось в пункте 2.4.2, групповые коммуникации не обязательно должны быть блокирующими. В связи с этим операция MPI_Bcast в некоторой реализации стандарта MPI может не вызвать зависание процессов на данной программе. Поэтому такой метод обнаружения ошибки несоответствия нельзя считать удовлетворительным.

ITAC верно диагностирует данную ошибку и выдает соответствующее предупреждение (листинг 43), включающее отдельно параметры вызова MPI_Bcast процессом, аргументы которого разнятся от других процессов, а также параметры данного вызова в остальных процессах.

Листинг 43. Сообщение ITAC о разных root-процессах в одной коллективной операции

```
[0] ERROR: GLOBAL:COLLECTIVE:ROOT_MISMATCH: error
[0] ERROR:    Root process does not match in the same collective operation.
[0] ERROR:    Collective call by local rank [0] (same as global rank):
[0] ERROR:          MPI_Bcast(*buffer=0x031a8af0, count=5, datatype=MPI_DOUBLE, root=2, comm=MPI_COMM_WORLD)
[0] ERROR:       main (D:\mistake_progs\bcast_diff_root.cpp:19)
 [0] ERROR:    Collective call by the 3 processes with local ranks [1:2] (same as global ranks):
[0] ERROR:          MPI_Bcast(*buffer=..., count=5, datatype=MPI_DOUBLE, root=2, comm=MPI_COMM_WORLD)
[0] ERROR:       main (D:\mistake_progs\bcast_diff_root.cpp:19)
```

9) Коллективная операция, запущенная не всеми процессами коммуникатора

Тестом на еще одну логическую ошибку, связанную с применением групповой функции, была программа, где коллективная операция была вызвана не всеми процессами коммуникатора MPI_COMM_WORLD (листинг 44). Функцию широковещательной рассылки MPI_Bcast вызывают все процессы, за исключением нулевого. Данный тест также запускался на трех процессах.

Листинг 44. Широковещательная рассылка, вызванная не всеми процессами коммуникотора

```
void main(int argc, char *argv[]) {
  int rank, nP;
  int rootproc = -1;
  MPI_Status mpiStatus;
  int length_of_ryad = 5;
```

```
int i = 0;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nP);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
double *mass_ryad = new double[length_of_ryad];
if(rank == nP-1){
  for(i=0;i<length_of_ryad;i++) { mass_ryad[i] = (double)i/2;}
}
rootproc = nP - 1;
if (rank != 0) {
  MPI_Bcast(mass_ryad, length_of_ryad, MPI_DOUBLE, rootproc, MPI_COMM_WORLD);
}
printf("\n I'm %d: ", rank);
for(i=0; i < length_of_ryad; i++) {printf(" %4.2lf", mass_ryad[i]);}
MPI_Finalize();}
```

Данная ситуация попадает под шаблон «Незавершенная коллективная операция». Соответствие с этим шаблоном выявляется по первой строке двумерного массива ссылок, где отсутствует одна ссылка (иначе говоря, одна ячейка в строке таблицы остается пустой). По этой строке определяется номер процесса, не вызвавшего функцию. По завершении работы системы в файле найденных ошибок появляется следующее сообщение (листинг 45).

Листинг 45. Сообщение системы о не полностью запущенной коллективной операции

```
COLLECTIVE OPERATION CALLED BY NOT ALL PROCESSES
Situation 1:
```

Operation – Bcast, MPI rank – 1, file – D:\mistake_progs\bcast_not_all.cpp, string - 19, datatype - MPI_DOUBLE, amount of elements – 5, root - 2

Operation – Bcast, MPI rank – 2, file – D:\mistake_progs\bcast_not_all.cpp, string - 19, datatype - MPI_DOUBLE, amount of elements – 5, root - 2

WASN'T CALL: Operation – Bcast, MPI rank - 0

Система выводит параметры всех вызовов данной операции всеми процессами, запустившими эту функцию. Для процесса, не участвовавшего в вызове коллективной функции, выводится строка, начинающаяся на «WASN'T CALL».

Подобно предыдущему тесту (коллективная операция с различным root-процессом в разных ветвях MPI-приложения) MARMOT обнаруживает зависание программы и выводит последние вызовы MPI-функций каждым процессом с параметрами (листинг 42).

На данный тест ITAC реагирует аналогично MARMOT и обнаруживает зависание всех процессов (нулевого – в MPI_Finalize, первого и второго – в MPI_Bcast). Для каждого процесса выводятся параметры вызова и количество секунд, которые данный процесс провел в этой функции (листинг 46).

Листинг 46. Сообщение ITAC о не полностью запущенной коллективной операции

[0] ERROR: GLOBAL:DEADLOCK:HARD: fatal error

[0] ERROR: [0] no progress observed for over 0:25 minutes, process is currently in MPI call:

[0] ERROR: MPI_Finalize()

[0] ERROR: main (D:\mistake_progs\bcast_not_all.cpp:23)

[0] ERROR: [1] no progress observed for over 0:25 minutes, process is currently in MPI call:

[0] ERROR: MPI_Bcast(*buffer=0x003fbb50, count=5, datatype=MPI_DOUBLE, root=2, comm=MPI_COMM_WORLD)

[0] ERROR: main (D:\mistake_progs\bcast_not_all.cpp:19)

[0] ERROR: [2] no progress observed for over 0:25 minutes, process is currently in MPI call:

[0] ERROR: MPI_Bcast(*buffer=0x003fbb70, count=5, datatype=MPI_DOUBLE, root=2, comm=MPI_COMM_WORLD)

[0] ERROR: main (D:\mistake_progs\bcast_not_all.cpp:19)

10) Пользовательская проверка на превышение количества передаваемых элементов заданного значения

Последний тест был проведен для демонстрации возможности системы обнаруживать поведение параллельной программы, соответствующее описанному в шаблоне даже в том случае, если описанное поведение не является ошибкой. Была создана программа с приведенным ниже файлом исходного кода (листинг 47).

Листинг 47. Превышение количества элементов заданного пользователем значения

```
void main(int argc, char *argv[]) {
      int rank, nP;
      MPI_Status mpiStatus;
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &nP);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      printf("\n  rank= %d, nP= %d", rank, nP);
      const int messagesize = 100;
 if(rank == 0) {
    int vartosend[messagesize];
    for( int i = 0; i < messagesize; i++) { vartosend[i] = i;}
    MPI_Send( vartosend, messagesize, MPI_INT, 1, 333, MPI_COMM_WORLD);
  } else  if(rank == 1) {
```

```
    int vartorecv[messagesize];
    MPI_Recv(vartorecv, messagesize, MPI_INT, 0, 333, MPI_COMM_WORLD, &mpiStatus);
    printf("\n FROM 0");
    for( int i = 0; i < messagesize; i++)
      printf("\n v[%d]=%d ", i, vartorecv[i]);
  }
 MPI_Finalize();}
```

Программа не содержит ошибок, то есть правила MPI-стандарта в ней не нарушены. Но зато в программе возникает ситуация, удовлетворяющая шаблону «Превышение количества передаваемых элементов заданного значения» (листинг 16). В шаблоне пользователь специфицировал, что ни одна операция отправки не должна передавать более пятнадцати элементов. Поскольку в программе передается 100, то система автоматизированного контроля корректности обнаружила соответствие и в итоговый файл ошибок записала следующую информацию (листинг 48).

Листинг 48. Сообщение о «срабатывании» пользовательского шаблона

```
AMOUNT OF TRANSMITED ELEMENTS MORE THAN DEFINED VALUE
Situation 1:
Function1:
File: D:\mistake_progs\too_many_elems.cpp
String number: 12
 Operation: Send
 MPI rank: 0
 Datatype: MPI_INT
```

Amount of elements: 100

Receiver rank: 1

Sender rank: 0

Tag: 333

CRC16: 4108

Инструментальные средства MARMOT и ITAC, естественно, сообщений об ошибках не выдали, поскольку в эти системы нельзя заложить пользовательские проверки, а собственный набор проверок включает только ошибки некорректного использования стандарта MPI.

4.3 Анализ проведенных тестов

Результаты тестирования инструментальных средств отладки можно представить в виде таблицы 4. По каждому тесту для систем были выставлены следующие оценки:

– «++» - ошибка продиагностирована точно; указаны функции, приведшие к появлению ошибки, и их локализация в программе.

– «+−» - в сообщении не содержится четкого указания на существующую ошибку, но имеется какая-либо информация, сужающая область поиска ошибки (например, выявлено зависание процессов, как следствие ошибки).

– «−−» - отстуствуют сообщения, каким-либо образом способствующие обнаружению ошибки.

Таблица 4. Результаты тестирование инструментальных средств отладки

ОШИБКА	MARMOT	ITAC	Собственная система автоматизированного контроля корректности
1. **Гонка данных**, обусловленная применением макроса MPI_ANY_SOURCE	+ –	– –	++
2. **Потенциальный дедлок** (2 процесса выполняют встречные операции MPI_Send)	– –	+ –	++
3. **Несоответствие** в размере сообщений точка-точка (буфер процесса-приемника больше)	– –	– –	++
4. **Несоответствие** в типе (один процесс отправляет сообщение другого типа, чем принимает адресат)	– –	++	++
5. **Непарная посылка** (один процесс выполняет MPI_Send, но адресат не вызывает операцию приема)	– –	+ –	++
6. **Ошибка «грязных данных»** (основной поток изменяет данные, участвующие в неблокирующей коммуникации)	– –	++	++
7. Неблокирующая коммуникация без функции ожидания	++	++	++
8. **Несоответствие** в номере корневого процесса при коллективной коммуникации	+ –	++	++
9. **Не полностью запущенная коллективная операция** (MPI_Bcast, вызванный не всеми процессами коммуникатора)	+ –	+ –	++

10. **Пользовательская проверка** на превышение количества передаваемых элементов заданного значения	– –	– –	++

Тестирование показывает, что свободно-распространяемое инструментальное средство MARMOT плохо обнаруживает глобальные ошибки и применяет «грубые» и неточные алгоритмы поиска некорректного использования MPI-стандарта. Intel Trace Analyzer and Collector способен обнаружить большее число ошибок, однако данная система платная и требует также приобретения реализации Intel MPI, при этом цена данного программного решения прямо пропорциональна количеству вычислительных узлов.

Сообщения, выдаваемые системой в результате анализа параллельных программ на соответствие шаблонам, содержат достаточно точную и подробную информацию о типе ошибки, ее локализации и причине ее возникновения. К тому же пользовательские проверки не способно производить ни одно из представленных средств, за исключением разработки диссертанта.

4.4 Отладка прикладных MPI-программ

После проведения тестов система автоматизированного контроля корректности была внедрена в центр коллективного пользования «Высокопроизводительные параллельные вычисления» КемГУ (http://icp.kemsu.ru/СКР), будучи установленной на вычислительном кластере, являющемся ресурсом центра. На данном кластере она была применена преподавателями кафедры ЮНЕСКО по новым информационным технологиям математического факультета КемГУ в ходе разработки серии учебных пособий «Основы высокопроизводительных вычислений» [11, 12, 13]. Третий том данной серии – «Параллельные вычислитель-

ные алгоритмы» [13]. В данном учебном пособии рассматриваются параллельные алгоритмы для классических задач линейной алгебры и программирования (умножение матриц и векторов, сортировка массивов действительных чисел, решение СЛАУ, параллельные методы на графах).

В ходе разработки пособия авторами был создан ряд параллельных программ с использованием библиотек MPI и OpenMP. Исходные коды этих программ вошли в текст пособия. Для проверки программ на корректность вызовов MPI-функций использовалась система, созданная в рамках данной диссертационной работы.

Сортировка массива методом чет-нечетной перестановки

Одной из протестированных программ был пример сортировки массива при помощи метода чет-нечетной перестановки. Составленная MPI-программа в нулевом процессе генерировала массив действительных чисел, рассылала его части по остальных процессам при помощи операции MPI_Scatterv, после чего производила сортировку массива с обменами элементами массива между процессами и собирала отсортированный массив снова в нулевом процессе. При запуске программы в конце работы нулевой процесс выводил собранный отсортированный массив, в результате чего на экране появлялись некорректные данные (значения с большим разбросом величин, существенно отличающиеся от тех, что изначально были сгенерированы для сортировки). Очевидно, что это являлось следствием ошибок, допущенных в программном коде. Программа была запущена под управлением системы автоматизированного контроля корректности. После завершения работы система сгенерировала файл ошибок, содержащий следующую информацию (листинг 49).

Листинг 49. Файл ошибок после запуска программы чет-нечетной сортировки

MISMATCHES OF SENDING AND RECEIVING DATA IN VECTOR VARIANT OF COLLECTIVE OPERATION FOR GATHERING FUNCTIONS

Situation 1:

Operation – Gatherv, MPI rank – 0, file - D:\Dissertaciya\from_igriva\oesort\oesort.cpp, string - 157, amount of sent elements – 2, sent datatype - MPI_DOUBLE, amounts of received elements – {2, 2, 2, 2}, displacements – {0, 2, 4, 6}, received datatype – MPI_INT, root – 0

Operation – Gatherv, MPI rank – 1, file - D:\Dissertaciya\from_igriva\oesort\oesort.cpp, string - 157, amount of sent elements – 2, sent datatype - MPI_DOUBLE, amounts of received elements – {2, 2, 2, 2}, displacements – {0, 2, 4, 6}, received datatype – MPI_INT, root – 0

Operation – Gatherv, MPI rank – 2, file - D:\Dissertaciya\from_igriva\oesort\oesort.cpp, string - 157, amount of sent elements – 2, sent datatype - MPI_DOUBLE, amounts of received elements – {2, 2, 2, 2}, displacements – {0, 2, 4, 6}, received datatype – MPI_INT, root – 0

Operation – Gatherv, MPI rank – 3, file - D:\Dissertaciya\from_igriva\oesort\oesort.cpp, string - 157, amount of sent elements – 2, sent datatype - MPI_DOUBLE, amounts of received elements – {2, 2, 2, 2}, displacements – {0, 2, 4, 6}, received datatype – MPI_INT, root – 0

Было обнаружено соответствие с шаблоном «Взаимно недопустимые типы принимаемых и отправляемых данных в векторном варианте коллективной операции (для функций сбора данных)», поскольку во всех вызвавших функцию MPI_Gatherv процессах тип данных в отправляемом буфере различается с типом данных в буфере приема. В исходном коде программы (приложение 2) строка 157, указанная в полученном файле ошибок, содержала следующий вызов (листинг 50).

Листинг 50. Вызов MPI_Gatherv в программе чет-нечетной сортировки

```
MPI_Gatherv(ALocal,nLocal,MPI_DOUBLE,A,ASendNum,ASendInd,MPI_INT,0,
MPI_COMM_WORLD);
```

После анализа вызова автор программы обнаружил несоответствие типов данных отправляемых блоков массива и типа данных итогового массива на 0-ом

процессе. После исправления найденной ошибки запуски данной программы проходили корректно.

Поблочное умножение матрицы на вектор

Следующей MPI-программой, для отладки которой использовалась разработанная система, являлась задача поблочного умножения матрицы на вектор, также вошедшая в учебное пособие «Параллельные вычислительные алгоритмы». Исходный код данной программы приведен в приложении 3. Подобно предыдущей задаче, исходная матрица и вектор генерируются на нулевом процессе, затем при помощи операции MPI_Scatterv рассылаются по другим процессам параллельной программы. Каждый из процессов вычисляет произведение своего блока матрицы на свой блок вектора, после чего результаты собираются на нулевом процессе при помощи глобальной редукции. Правильность полученного вектора произведения проверяется сравнением с результатом последовательного умножения исходной матрицы и вектора на нулевом процессе.

Запуск программы давал существенное различие между результатами, полученными при последовательном и параллельном умножении. После проверки программы системой автоматизированного контроля корректности был сгенерирован файл ошибок следующего содержания (листинг 51).

Листинг 51. Файл ошибок после запуска программы поблочного умножения матрицы на вектор

```
BUFFER SIZES MISMATCH IN DIFFERENT PROCESSES DURING COLLECTIVE OPERATION:

Situation 1:

Operation – Reduce, MPI rank – 0, file - D:\Dissertaciya\from_igriva\block_multiply\mvb.cpp, string - 199, amount of elements – 3, datatype - MPI_DOUBLE, reduction operation - SUM, root – 0
```

Operation – Reduce, MPI rank – 1, file - D:\Dissertaciya\from_igriva\block_multiply\mvb.cpp, string - 199, amount of elements – 2, datatype - MPI_DOUBLE, reduction operation - SUM, root – 0

Operation – Reduce, MPI rank – 2, file - D:\Dissertaciya\from_igriva\block_multiply\mvb.cpp, string - 199, amount of elements – 3, datatype - MPI_DOUBLE, reduction operation - SUM, root – 0

Operation – Reduce, MPI rank – 3, file - D:\Dissertaciya\from_igriva\block_multiply\mvb.cpp, string - 199, amount of elements – 2, datatype - MPI_DOUBLE, reduction operation - SUM, root – 0

Приведенное выше сообщение является следствием обнаружения системой ситуации, описанной в шаблоне «Несовпадение размеров буферов в разных процессах при коллективной операции» для редукционных функций (листинг 14). По данному файлу автор программы заключил, что в вызове функции MPI_Reduce (листинг 52) количество элементов, пересылаемое для суммирования на нулевой процесс, различается в разных ветвях параллельного приложения. В итоге был сделан вывод о том, что в данном вызове вместо *nLocalCol* необходимо использовать другую переменную (*n* – размерность исходной матрицы).

Листинг 52. Вызов MPI_Reduce в программе поблочного умножения матрицы на вектор

```
MPI_Reduce( pcLocal, pc, nLocalCol, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Заключение

В рамках данной работы приведена классификация семантических ошибок в параллельных программах для многопроцессорных вычислительных систем с распределенной памятью. Среди них детально разобраны те, которые вызваны некорректным использованием стандарта MPI. Описаны подходы к обнаружению таких ошибок, применяемые при построении инструментальных средств отладки MPI-приложений. В качестве особенно перспективного и удобного для пользователя выделен подход автоматизированного контроля корректности. Проведен анализ существующих систем, которые применяют данный подход. Анализ показал, что в этих системах набор проверок на семантические ошибки строго фиксирован и у пользователя нет возможности «дать задание» системе провести интересующую его проверку. Кроме того, существующие системы способны обнаружить далеко не все возможные ошибки, обусловленные нарушением правил MPI-стандарта — это доказывают проведенные в рамках работы тесты на ошибочных программах. Было сделано заключение о том, что реализовать в рамках какой-либо системы, использующей подход автоматизированного контроля корректности обнаружение всех логических ошибок в MPI-программах и поддерживать эту систему актуальной чрезвычайно сложно ввиду огромного количества различных возможных ошибок и периодического обновления MPI-стандарта (появления новых версий).

В связи с вышеизложенным была разработана система описания шаблонов ошибочного поведения MPI-программ, позволяющая легко формализовать ошибку для последующего обнаружения автоматизированными инструментальными средствами отладки. С помощью введенной системы описаны 22 распространенные ошибки, обусловленные некорректным использованием функций стандартов MPI-1 и MPI-2, и 2 пользовательских шаблона для ситуаций, не являющихся ошибками.

Также была разработана архитектура и принципы функционирования новой системы автоматизированного контроля корректности MPI-программ, проводящей анализ пользовательских приложений на соответствие шаблонам, поданным на вход системе. В настоящей работе описаны основные структуры данных и процесс их наполнения и освобождения во время работы системы. Приведен процесс обработки созданным инструментальным средством параллельных программ, содержащих некоторые шаблонные ситуации.

Даны результаты тестов, в которых одни и те же MPI-программы, содержащие различные ошибки, были проверены разработанной системой и известными инструментальными средствами, также использующими подход автоматизированного контроля коррекности. Результаты показали, что исследуемые средства отладки в нескольких случаях менее точно диагностируют причины некорректного поведения MPI-программы и выявляют потенциальные ошибки по сравнению с системой собственной разработки, когда имеется соответствующий шаблон с описанной ситуацией.

Созданная система была внедрена в деятельность центра коллективного пользования «Высокопроизводительные параллельные вычисления» Кемеровского государственного университета и использована преподавателями и аспирантами КемГУ для отладки создаваемых MPI-приложений.

Таким образом, в результате проведения работы была достигнута поставленная цель – разработана система описания шаблонов ошибочного поведения MPI-приложений и создано программное средство, выявляющее соответствие поведения параллельной программы описанным в шаблонах ситуациям.

В ходе выполнения работы были достигнуты следующие основные результаты:

– предложен метод автоматизированного контроля корректности параллельных приложений на основе шаблонов ошибочного поведения;
– создана система описания шаблонов ошибочного поведения параллельных программ, позволяющая формализовать пользовательские проверки и ошибки, возникающие при использовании функций стандарта MPI как выпущенных версий, так и будущих;
– разработана расширяемая библиотека шаблонов для распространенных семантических ошибок в MPI-программах;
– создано программное средство отладки, обнаруживающее шаблонные ситуации во время работы MPI-приложений;
– разработаны рекомендации по составлению текстовых шаблонов и применению созданного программного средства.

Список литературы

1. Абрамов, С.М. Т-система - среда программирования с поддержкой автоматического динамического распараллеливания программ. Пример реализации алгоритма построения изображений методом трассировки лучей [Текст] / С.М. Абрамов, А.И. Адамович, М.Р. Коваленко // Программирование. Т. 25 (2), 1999. С. 100-107.

2. Алексахин, В.А. Средства отладки OPENMP-программ в DVM-системе [Текст] / В.А. Алексахин, В.О. Баринова, В.А. Бахтин и др. // Материалы Всероссийской научной конференции «Научный сервис в сети Интернет: технология распределённых вычислений» 22-27 сентября 2008г., г.Новороссийск – М.: Изд-во МГУ, 2008. С. 281-285.

3. Андрианов, А.Н. Норма. Описание языка. Рабочий стандарт [Текст] / А.Н.Андрианов, А.Б.Бугеря, К.Н.Ефимкин и др.// Препринт ИПМ им.М.В.Келдыша РАН, N120, 1995. – 50 с.

4. Афанасьев, К.Е. Многопроцессорные вычислительные системы и параллельное программирование [Текст] / К.Е. Афанасьев, С.В. Стуколов // Кемерово, 2003. – 182 с.

5. Афанасьев, К.Е. Многопроцессорные системы: построение, развитие, обучение [Текст] / К.Е. Афанасьев, В.Г. Домрачев, И.В Ретинская и др. // Под ред. Тихонова А.Н. - М.: КУДИЦ-ОБРАЗ, 2005. – 224 с.

6. Афанасьев, К.Е. Семантические ошибки в параллельных программах для систем с распределенной памятью и методы их обнаружения современными средствами отладки. [Текст] / К.Е. Афанасьев, А.Ю. Власенко // Вестник Кемеровского государственного университета. №2. Кемерово: Изд-во КемГУ, 2009. С. 13-20.

7. Афанасьев, К.Е. Распределенная система автоматического обнаружения семантических ошибок в MPI-программах [Текст] / К.Е. Афанасьев, А.Ю. Власенко // Сборник статей участников Всероссийского конкурса научных работ студентов и аспирантов «Телематика' 2010: телекоммуникации, веб-технологии, суперкомпьютинг». – СПб: СПбГУ ИТМО, 2010. С. 167-172.

8. Афанасьев, К.Е. Реализация подхода автоматического контроля корректности в распределенной информационной системе [Текст] / К.Е. Афанасьев, А.Ю. Власенко // Программа и тезисы докладов шестой сибирской конференции по параллельным и высокопроизводительным вычислениям. Томск: изд-во Томского университета, 2011. С. 78-79.

9. Афанасьев, К.Е. Автоматизированный поиск шаблонов ошибочного поведения в параллельных программах, использующих модель передачи сообщений [Текст] / К.Е. Афанасьев, А.Ю. Власенко // Программа и тезисы докладов седьмой сибирской конференции по параллельным и высокопроизводительным вычислениям. Томск: изд-во Томского университета, 2013. С. 18-19.

10. Афанасьев, К.Е. Разработка системы поддержки высокопроизводительных вычислений в Web-среде [Электронный ресурс] / К.Е. Афанасьев, А.Ю. Власенко // Информационно-вычислительные технологии и математическое моделирование (ИВТ&ММ): сборник трудов конференции. Кемерово: Изд-во КемГУ, 2013. С. 1-2.

11. Афанасьев, К.Е. Основы высокопроизводительных вычислений. Т.1. Высокопроизводительные вычислительные системы: учебное пособие [Текст] / К.Е. Афанасьев, С.Ю. Завозкин, С.Н. Трофимов, А.Ю. Власенко; Кемеровский государственный университет. - Кемерово, 2013. – 245 с.

12. Афанасьев, К.Е. Основы высокопроизводительных вычислений. Т.2. Технологии параллельного программирования:учебное пособие: учебное пособие [Текст] / К.Е. Афанасьев, С.В. Стуколов, В.В. Малышенко, С.Н. Карабцев, Н.Е. Андреев; Кемеровский государственный университет. - Кемерово, 2013. – 434 с.

13. Афанасьев, К.Е. Основы высокопроизводительных вычислений. Т.3. Параллельные вычислительные алгоритмы: учебное пособие [Текст] / К.Е. Афанасьев, И.В. Григорьева, Т.С. Рейн; Кемеровский государственный университет. – Кемерово, 2012. – 185 с.

14. Афанасьев, К.Е. Автоматизированный анализ корректности MPI-программ на основе определенных пользователем шаблонов ошибочного поведения [Текст] / К.Е. Афанасьев, А.Ю. Власенко // Вестник Томского

Государственного Университета. Серия: Управление, вычислительная техника и информатика. №1 (26), 2014. С. 75-83.

15. Бахтин, В.А. Использование языка Fortran-DVM/OpenMP для разработки больших задач [Текст] / В.А. Бахтин, А.В. Воронков, А.С. Голубев и др.// Труды Всероссийской научной конференции «Научный сервис в сети Интернет: решение больших задач», сентябрь 2008 г., г. Новороссийск. – М.: Изд-во МГУ, 2008. С. 185-191.

16. Бычков, И.В. Методы и средства организации параллельных и распределенных вычислений на основе парадигмы модульного программирования [Текст] / И.В. Бычков, Г.А. Опарин, А.П. Новопашин и др. // Вестник Кемеровского государственного университета. № 4(52). Кемерово: Изд-во КемГУ, 2012. С. 22-29.

17. Власенко, А.Ю. Отладка MPI-преограмм на персональном компьютере через эмуляцию многопроцессорной вычислительной системы [Текст] / А.Ю. Власенко, А.В. Демидов // Материалы 6 Международного научно-практического семинара «Высокопроизводительные параллельные вычисления на кластерных системах» 12-17 декабря 2006г. Санкт-Петербург: Изд-во Санкт-Петербургского госуниверситета, 2007. С. 164-169.

18. Власенко, А.Ю. Система отладки MPI-программ [Текст] / А.Ю. Власенко, А.В. Демидов // Материалы 11 Всероссийской научно-практической конференции «Научное творчество молодёжи» 20-21 апреля 2007г. Томск: Изд-во Томского университета, 2007. С. 69-72.

19. Власенко, А.Ю. Отладчик параллельных программ [Текст] / А.Ю. Власенко, А.В. Демидов // Материалы третьей международной научно-практической конференции студентов, аспирантов и молодых учёных. Кемерово: ООО «ИНТ», 2008. С. 152-153.

20. Власенко, А.Ю. Система контроля корректности MPI-программ [Текст] / А.Ю. Власенко, А.В. Демидов // Материалы 6 Международной научно-практической конференции «Информационные технологии и математическое моделирование» 9-10 ноября 2007г. Томск: Изд-во Томского университета, 2007. С. 19-23.

21. Власенко, А.Ю. Отладка MPI-программ на персональной ЭВМ [Текст] / А.Ю. Власенко, А.В. Демидов // Материалы Всероссийской научной конференции «Инновационные недра Кузбасса. IT-технологии». Кемерово: ООО «ИНТ», 2008. С. 236-241.

22. Власенко, А.Ю. Система динамического контроля корректности MPI-программ. / А.Ю. Власенко, А.В. Демидов // Материалы 46 Международной научной студенческой конференции «Студент и научно-технической прогресс»: Психология / Новосибирск: Новосибирский гос. ун-т., 2008. С. 216-217.

23. Власенко, А.Ю. Intel Trace Analyzer and Collector как одно из наиболее распространенных программных средств автоматического контроля корректности MPI-приложений. / А.Ю. Власенко // Материалы Пятнадцатой Всероссийской научной конференции студентов-физиков и молодых ученых (ВНКСФ-15, Кемерово-Томск). Кемерово: издательство АСФ России, 2009. С. 646-648.

24. Власенко, А.Ю. Модель масштабируемой системы автоматического контроля корректности параллельных программ. [Текст] / А.Ю. Власенко // Вестник НГУ. Серия: Информационные технологии. Т.7, выпуск 4, 2009. С. 53-66.

25. Власенко, А.Ю. Подсистема автоматического обнаружения логических ошибок в MPI-программах информационно-вычислительного портала Кемеровского государственного университета. [Текст] /А. Ю. Власенко // Материалы пятой сибирской конференции по параллельным и высокопроизводительным вычислениям. Томск: изд-во Томского университета, 2009. С. 153-156.

26. Власенко, А.Ю. Архитектура и принципы работы масштабируемого инструментального средства динамического обнаружения семантических ошибок в MPI-программах. [Текст] / А.Ю. Власенко // Материалы IX международной конференции-семинара «Высокопроизводительные параллельные вычисления на кластерных системах». Владимир: изд-во Владимирского государственного университета, 2009. С. 73-76.

27. Власенко, А.Ю. Комплекс информационных систем поддержки параллельных вычислений в рамках информационно-вычислительного

портала [Электронный ресурс] / А.Ю. Власенко, Н.Н. Окулов // Труды международной суперкомпьютерной конференции «Научный сервис в сети Интернет: экзафлопсное будущее», 2011. С. 517–523.

28. Власенко, А.Ю. Система автоматического контроля корректности и виртуальная лаборатория как компоненты информационно-вычислительного портала [Текст] / А.Ю. Власенко, Н.Н. Окулов // Научно-технический вестник Поволжья. №6 2011г. – Казань: Научно-технический вестник Поволжья, 2011г. – 267 с.

29. Власенко, А.Ю. Программный комплекс для анализа, отладки и оптимизации параллельных приложений [Текст] / А.Ю. Власенко, Н.Н. Окулов, А.В. Демидов // Параллельные вычислительные технологии (ПаВТ'2012): труды международной научной конференции (Новосибирск, 26 – 30 марта 2012 г.). Челябинск: Издательский центр ЮУрГУ, 2012. – 715 с.

30. Воеводин, В.В. Параллельные вычисления [Текст] / В.В. Воеводин, Вл.В. Воеводин. - СПб: БХВ-Петербург, 2002. – 608 с.

31. Гергель, В.П. Введение в методы параллельного программирования. Лабораторная работа 3. Отладка параллельных MPI-программ в среде Microsoft Visual Studio 2005. [Электронный ресурс] / В.П. Гергель // http://www.software.unn.ac.ru/ccam/mskurs/LAB/RUS/PPT/CCS_Lab03.pdf (дата обращения: 12.03.2013).

32. Гергель, В.П. Теория и практика параллельных вычислений [Текст] / В.П. Гергель. – М.:БИНОМ. Лаборатория знаний, 2007. – 424 с.

33. Григорьева, И.В. Система удаленного доступа и управления распределенными вычислительными ресурсами [Текст] / И.В. Григорьева, А.В. Демидов // Вычислительные технологии. Т.13, специальный выпуск 5, 2008. С. 28-32.

34. Гудов, А.М. Технология разработки программного обеспечения [Электронный ресурс]/ А.М. Гудов, С. Ю. Завозкин, С. Н. Трофимов // http://unesco.kemsu.ru/study_work/method/po/UMK/index.html (дата обращения 5.07.2012).

35. Ефимкин, К.Н. Средства отладки MPI-программ. Анализатор корректности [Электронный ресурс]/ К.Н. Ефимкин, О.Ф. Жукова, В.А.Крюков// http://www.keldysh.ru/papers/2006/prep28/prep2006_28.html (дата обращения: 17.02.2013).

36. Коновалов, Н.А. C-DVM - язык разработки мобильных параллельных программ [Текст] / Н.А.Коновалов, В.А.Крюков, Ю.Л.Сазанов // Программирование N 1, 1999. С. 20-28.

37. Крюков, В.А. Разработка параллельных программ для вычислительных кластеров и сетей [Текст] / В.А. Крюков // Информационнные технологии и вычислительные системы, №1-2. - М.: ИМВС РАН, 2003. С. 42-61.

38. Лаевский, Ю.М. Оптимизация приложений с использованием библиотеки Intel MKL [Электронный ресурс] / Ю.М. Лаевский, А.А. Романенко // http://www.intuit.ru/studies/courses/663/519/info (дата обращения: 10.11.2013).

39. Малышкин, В.Э. Отладчик параллельных программ для многопроцессорных вычислительных систем [Текст] / В.Э. Малышкин, А.А. Романенко // Материалы научно-методической конференции "ТЕЛЕМАТИКА-2003". Санкт-Петербург: Изд-во Санкт-Петербургского государственного университета точной механики и оптики, 2003. С. 259-262.

40. Малышкин, В.Э. Базисные функции отладчика параллельных программ GEPARD и их реализация для МВС-1000/М [Текст] / В.Э. Малышкин, А.А. Романенко // Материалы второго научно-практического семинара "Высокопроизводительные параллельные вычисления на кластерах". Нижний Новгород, 2002г. С. 248-255.

41. Новиков, Р.Н. Многоядерные процессоры: теория (на примере Intel и AMD) [Электронный ресурс] / Р.Н. Новиков // http://novikov.ua/bnews/2007/07/20/5945/ (дата обращения: 16.12.2011).

42. Отладка корректности MPI-программ [Электронный ресурс] // http://keldysh.ru/dvm/dvmhtm1107/rus/deb_mpi/tranUG.html (дата обращения: 13.10.2012).

43. Руководство пользователя по библиотеке PETSc [Электронный ресурс] // http://www.cluster.bsu.by/download/Perevod_PDF.pdf (дата обращения: 17.02.2013).

44. Сердюк, Ю.П. Кластерные вычисления [Электронный ресурс] / Ю.П. Сердюк // http://www.intuit.ru/department/calculate/clusterexec/2/4.html#sect8 (дата обращения: 3.08.2012).

45. Средства отладки MPI-программ [Электронный ресурс] // http://www.kiam.ru/dvm/dvmhtm1107/rus/deb_mpi/deb_mpi.html (дата обращения: 12.09.2012).

46. Сущенко, С.П. Параллельные вычисления [Текст] / С.П. Сущенко. - Томск: Изд-во ТГУ, 2003. – 196 с.

47. Уоррен, Г. Алгоритмические трюки для программистов [Текст] / Г. Уоррен - Вильямс, 2004 г. – 288 с.

48. Фролов, А. В. Программирование для Windows NT [Текст] / А.В. Фролов, Г.В. Фролов - М.: Диалог-МИФИ, 1996. - 272 с.

49. Шпаковский, Г.И. Программирование для многопроцессорных систем в стандарте MPI. [Электронный ресурс] / Г.И. Шпаковский, Н.В. Серикова// http://window.edu.ru/window_catalog/redir?id=24068&file=shpakovski_bsu.pdf (дата обращения: 14.12.2011).

50. Эдмунд, М. Кларк. Верификация моделей программ: Model Checking. [Текст] / М. Кларк Эдмунд, О. Грамберг, Д. Пелед - М.: издательство московского центра непрерывного математического образования, 2002. – 416 с.

51. Якобсон, А. Унифицированный процесс разработки программного обеспечения [Текст] / А. Якобсон, А.Г. Буч, Дж. Рамбо - СПб.: Питер, 2002. – 496 с.

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

53. Balay, S. PETSc users manual. Technical Report ANL-95/11 - Revision 3.4 [Электронный ресурс] / S. Balay, J. Brown, K. Buschelman и др. // http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf (дата обращения: 03.02.2014).

54. Barney, B. TotalView [Электронный ресурс] / B. Barney // https://computing.llnl.gov/tutorials/totalview/ (дата обращения: 07.04.2012).

55. Beech-Brandt J. Applications of UPC [Электронный ресурс] / J. Beech-Brandt // http://www.nesc.ac.uk/talks/892/applicationsofupc.pdf (дата обращения: 30.08.2011).

56. BERT 77: Automatic and Efficient Parallelizer for FORTRAN [Электронный ресурс] // http://www.basement-supercomputing.com/content/view/24/50/ (дата обращения: 15.02.2011).

57. Bonachea, D. GASNet Specification, v1.1. - U.C. Berkeley Tech Report (UCB/CSD-02-1207), 2002. – 32 p.

58. Brunst, H. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach [Текст] / H. Brunst, H. Hoppe, W. Nagel и др. // Proceedings of International Conference on Computational Science - ICCS, San Francisco, CA, USA, May 2001. P. 751-760.

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

60. Eschweiler, D. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries [Текст] / D. Eschweiler, M. Wagner, M. Geimer // Advances in Parallel Computing. Vol.22: Applications, Tools and Techniques on the Road to Exascale Computing, 2012. P. 481-490.

61. Geimer, M. Scalable Parallel Trace-Based Performance Analysis [Текст] / M. Geimer, F. Wolf, B. Wylie и др. // Proceedings of the 13th European PVM/MPI Conference, Springer LNCS, Bonn, Germany, September 2006. P. 303-312.

62. Grelck, C. S-Net: A Typed Stream Processing Language [Текст] / C. Grelck, S. Scholz, A. Shafarenko // Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06), Budapest, Hungary, 2006. P. 81-97.

63. Guzev, V., Serdyuk Y. Asynchronous parallel programming language based on the Microsoft .NET platform [Текст] / V. Guzev, Y. Serdyuk // PaCT-2003, Lecture Notes in Computer Science, v. 2763, Springer, 2003. P. 236-243.

64. Hilbrich, T. MUST: A Scalable Approach to Runtime Error Detection in MPI Programs [Текст] / T. Hilbrich, M. Schulz, B. R. de Supinski и др. // Tools for High Performance Computing. Springer, Berlin, 2009. P. 53-66.

65. Hilbrich, T. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection [Текст] / T. Hilbrich, J. Protze, M. Schulz и др. // Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. Salt Lake City, UT, USA, 2012. P. 1-10.

66. Intel Trace Analyzer and Collector – Documentation [Электронный ресурс] // http://software.intel.com/en-us/articles/intel-trace-analyzer-and-collector-documentation/ (дата обращения: 23.11.2013).

67. Introduction to the Partitioned Global Address Space (PGAS) Programming Model [Электронный ресурс] // http://cnx.org/content/m20649/latest/ (дата обращения: 18.04.2013).

68. Introduction to the SHMEM programming model [Электронный ресурс] // http://docs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=linux&db=man&fname=/usr/share/catman/man3/intro_shmem.3.html&srch=intro_shmem (дата обращения: 24.07.2012).

69. Johnson, S. The ParaWise Expert Assistant – Widening Accessibility to Efficient, Scalable Tool Generated OpenMP Code [Текст] / S. Johnson, E. Evans, H. Jin и др. // Proceedings of WOMPAT 2004, LNCS 3349. P. 67-82.

70. Kluge, M. Collecting Distributed Performance Data with Dataheap: Generating and Exploiting a Holistic System View [Текст] / M. Kluge, D. Hackenberg, W. Nagel // ICCS, Vol. 9 of Procedia Computer Science, Elsevier, 2012. P. 1969-1978.

71. Krammer, B. MARMOT: An MPI Analysis and Checking Tool. [Текст] / B. Krammer, K. Bidmon, M.S. Muller и др. // PARCO. Vol. 13 of Advances in Parallel Computing, Elsevier, 2003. P. 493–500.

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

73. Krammer, B. MPI I/O Analysis and Error Detection with MARMOT /B. Krammer, M. Muller, M. Resch // Proceedings of 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19 - 22, 2004. P. 242-250.

74. Lorenz, D. Extending Scalasca's Analysis Features [Текст] / D. Lorenz, D. Böhme, B. Mohr и др.// Proceedings of the Tools for High Performance Computing 2012. Springer Berlin Heidelberg, 2013. P. 115–126.

75. Luecke, G. MPI-CHECK: A tool for checking Fortran 90 MPI programs [Текст] / G. Luecke, H. Chen, J. Coyle и др.// Concurrency and Computation:Practice and Experience, Vol. 15, 2003. P. 93-100.

76. MARMOT Windows Tutorial [Электронный ресурс] // http://fs.hlrs.de/projects/marmot/downloads/marmot-2.1.0/Marmot_Windows_Tutorial.pdf (дата обращения: 11.03.2012)

77. Message Passing Interface Forum [Электронный ресурс] // http:/www.mpi-forum.org/ (дата обращения: 24.10.2013).

78. MPI: A Message-Passing Interface Standard [Электронный ресурс] // http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-1.1/mpi-report.htm (дата обращения: 23.10.2013).

79. Nieplocha, J. High Performance Remote Memory Access Comunications: The ARMCI Approach [Текст] / J. Nieplocha, V. Tipparaju, M. Krishnan и др. // International Journal of High Performance Computing and Applications, Vol 20(2), 2006. P. 233-253.

80. O'Neal, D. CAPTools Project Final Report: Evaluation and Application of the Computer Aided Parallelisation Tools / D. O'Neal, R. Luczak, M. White //

Proceedings of the DoD High Performance Computing Modernization Program Users Group Conference, Albuquerque, New Mexico, June 5-8, 2000. P. 57-75.

81. Pervez, S. Practical Model Checking Method for Verifying Correctness of MPI Programs / S. Pervez, G. Gopalakrishnan, R. M. Kirby и др. // Proceedings of 14th European PVM/MPI User's Group Meeting, Paris, France, September/October 2007. P. 344-353.

82. Protze, J. MPI Runtime Error Detection with MUST: Advanced Error Reports [Текст] / J. Protze, T. Hilbrich, B.R. de Supinski и др.// Proceeding of Tools for High Performance Computing 2012, Stuttgart, Germany, September 2012. P. 25-38.

83. Scalable Offline Debugging [Электронный ресурс] / http://www.allinea.com/Portals/90122/docs/allinea_whitepaper_nov11.pdf (дата обращения: 9.05.2012).

84. Schöne, R. The VampirTrace Plugin Counter Interface: Introduction and Examples / R. Schöne, R. Tschüter, D. Hackenberg и др. // Proceedings of the Euro-Par 2010 Parallel processing Workshops, Berlin Heidelberg, Springer-Verlag, 2011. P. 501-511.

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

86. Vakkalanka, S. ISP: A Tool for Model Checking MPI Programs [Текст] / S. Vakkalanka, S. Sharma, G. Gopalakrishnan, R. M. Kirby // Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, New York, USA 2008. P. 285-286.

87. Vetter, J. Dynamic Software Testing of MPI Applications with Umpire [Электронный ресурс] / J. Vetter, B. de Supinski // Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), 2000. Article No.51. P. 70-79.

88. Wagner, M. Hierarchical Memory Buffering Techniques for an In-Memory Event Tracing Extension to the Open Trace Format 2 [Текст] / M. Wagner, A.

Knüpfer, Wolfgang E. Nagel // Proceedings of the 42nd International Conference on Parallel Processing (ICPP), Lyon, France, 2013. P. 970-976.

89. Wolf, F. Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications [Текст] / F. Wolf, B. Wylie, E. Abraham и др.// Proceedings of the 2nd Parallel Tools Workshop, Germany, 2008. P. 157–167.

90. Wolf, F. Epilog binary trace-data format [Текст] / F. Wolf, B. Mohr // Tech. rep. FZJ-ZAM-IB-2004-06 (May 2004), 23 p.

Приложение 1. Определение системы описания шаблонов в расширенной форме Бэкуса-Наура

Поскольку в различных работах существует несколько вариантов синтаксиса расширенной формы Бэкуса-Наура (РБНФ), то ниже приводятся принятые соглашения.

- “ ::= ” – определение
- “ ‘…’ ” - терминал
- “ , ” – конкатенация
- “ | ” – выбор
- “ ; ” – завершение правила
- “ [] ” – условное вхождение
- “ {} ” – повторение
- “ () ” – группировка элементов
- “ (*…*) ” – комментарий
- “ ; ” – завершающий символ для правила

шаблон ::= ‘Name’, ‘=’, строка_символов,
‘1block’,
‘K’, ‘=’, количество_процессов [, номера_процессов],
‘2block’,
{‘F’, натуральное_число, ‘=’, ‘p’, натуральное_число, ‘:’, функция [{, ‘||’, функция}] [{, ‘&&’, функция}] },
‘3block’,
{(‘F’, натуральное_число, ‘(’, натуральное_число, ‘)’, (‘=’ | ‘!=’ | ‘>’ | ‘<’), значение_аргумента) | (‘CRC_BEFORE’, ‘(’, ‘F’, натуральное_число, ‘(’, натуральное_число, ‘)’, ‘)’, ‘!=’, ‘CRC_AFTER’, ‘(’, ‘F’, натуральное_число, ‘(’, натуральное_число, ‘)’, ‘)’)}
;
строка_символов ::= {символ};
количество_процессов ::= натуральное_число | ‘n(MPI_COMM_WORLD)’;
номера_процессов ::= {‘ ’, ‘p’, натуральное_число, ‘=’, натуральное_число};
функция ::= макрос | имя_MPI_функции_без_префикса_MPI [, ‘!’];
макрос ::= PTP | IPTP | Coll_data | Coll_reduc | Send_any | Recv_any;
PTP ::= Send | Bsend | Ssend | Rsend | Recv;
IPTP ::= Isend | Ibsend | Issend | Irsend | Irecv;
Coll_data ::= Bcast | Gather | Scatter | Alltoall;
Coll_reduc ::= Reduce | Scan | Allreduce;
Send_any ::= Send | Bsend | Ssend | Rsend | Isend | Ibsend | Issend | Irsend;
Recv_any ::= Recv | Irecv;
значение_аргумента ::= ‘F’, натуральное_число, ‘(’, натуральное_число, ‘)’ |
  натуральное_число |
  ‘p’, натуральное_число |
  константа_MPI;
константа_MPI ::= MPI_ANY_SOURCE | MPI_ANY_TAG | MPI_TAG_UB | MPI_MAX | MPI_MIN | MPI_SUM| MPI_PROD | MPI_LAND | MPI_BAND | MPI_LOR | MPI_BOR | MPI_LXOR | MPI_BXOR | MPI_ MINLOC | MPI_MAXLOC | MPI_REPLACE;

Ввиду очевидности в правилах не описаны нетерминалы «натуральное_число», «символ» и «имя_MPI_функции_без_префикса_MPI».

Приложение 2. MPI-программа сортировки массива методом чет-нечетной перестановки

```
#include "mpi.h"
#include "math.h"
#include "stdio.h"
#include "stdlib.h"
#include "malloc.h"

void QuickSort(double*,int,int);
void CompareSplit(int AID,int BID,int nA,int nB,int nMax,double *ALocal);

int main(int argc, char* argv[])
{

double t1, t2, dt1,dt2;
int narray[]={8,1000,2000,3000,5000};

int ncount=sizeof(narray)/sizeof(int);

double *A,*ALocal;
int n,nLocal,MaxnLocal,TmpnLocal,NextnLocal,PrevnLocal;
int MyID,NextID,PrevID;
int i,P,Rest,rnk;

MPI_Status MPI_Stat;
char Pname[MPI_MAX_PROCESSOR_NAME];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &P);
MPI_Comm_rank(MPI_COMM_WORLD, &MyID);

printf ("I have %d proc. ncount = %d\n", P, ncount);
MPI_Get_processor_name(Pname,&rnk);
printf("I am proc num=%d, %d, name=%s\n",MyID,rnk,Pname);

//Количество процессоров четное
if (P%2!=0)
{
  printf("P=%d, number of processes must be odd.\n",MyID);
  return 0;
}

PrevID=MyID-1;
NextID=MyID+1;
if (MyID==0) PrevID=P-1;
if (MyID==P-1) NextID=0;
```

```
n=narray[0];
printf("\nMyID=%d n=%d\n",MyID,n);

if (MyID==0)
{
  A=new double [n];
  for(i=0;i<n;i++)
    A[i]=n-i;
  printf("\nBefore:\n");
  for(i=0;i<n;i++)
    printf (" a(%d)=%f ",i,A[i]);
}

MPI_Barrier(MPI_COMM_WORLD);

// Распределение массивов для рассылки массивов A
int *ASendInd = new int [P]; // массив начальных индексов для рассылки A
int *ASendNum = new int [P]; // массив количеств элементов при рассылке A

// Вычисление массивов количества и начальных индексов рассылаемых элементов
TmpnLocal=n/P;
Rest=n-TmpnLocal*P;

if (Rest==0)
  MaxnLocal=TmpnLocal;
else
  MaxnLocal=TmpnLocal+1;

ASendInd[0]=0;
if (Rest>0)
  ASendNum[0]=MaxnLocal;
else
  ASendNum[0]=TmpnLocal;

for(i=1;i<P;i++)
{
  if (i<Rest)
        ASendNum[i]=MaxnLocal;
  else
        ASendNum[i]=TmpnLocal;
  ASendInd[i] = ASendInd[i-1]+ASendNum[i-1];
}

nLocal=ASendNum[MyID];
PrevnLocal=ASendNum[PrevID];
NextnLocal=ASendNum[NextID];

if (MyID==0)
{  printf("\n************************************\n");
```

```
  for (i=0;i<P;i++)
    printf(" sn_%d=%i ",i,ASendNum[i]);
  for (i=0;i<P;i++)
    printf(" si_%d=%i ",i,ASendInd[i]);
}

ALocal= new double [MaxnLocal];
for (i=0;i<MaxnLocal;i++)
  ALocal[i]=0.0;

printf("\n I am %d, nLocal=%d\n",MyID,nLocal);

// Рассылка строк матрицы
MPI_Scatterv(A,ASendNum,ASendInd,MPI_DOUBLE,ALocal,nLocal,MPI_DOUBLE,0,MPI_CO
MM_WORLD);

QuickSort(ALocal,0,nLocal-1);
for(i=0;i<nLocal;i++)
printf(" as%1d_%d=%f ",MyID,i,ALocal[i]);
MPI_Barrier(MPI_COMM_WORLD);

for (i=0;i<P;i++)
{
  MPI_Barrier(MPI_COMM_WORLD);
  if (i%2==0) //четный шаг
  {
    if (MyID%2==0) //с процессором справа
    {
          printf ("\n%d: ODD1: I is %d contact with %d\n",i,MyID,NextID);
              CompareSplit(MyID,NextID,nLocal,NextnLocal,MaxnLocal,ALocal);
    }
        else  //с процессором слева
        {
            printf ("\n%d: ODD2: I is %d contact with %d\n",i,MyID,PrevID);
              CompareSplit(MyID,PrevID,nLocal,PrevnLocal,MaxnLocal,ALocal);
        }
  }
  else   //нечетный шаг
  {
    if (MyID%2==0)//с процессором слева
        {
              printf ("\n%d: EVEN1: I is %d(%d) contact with
%d(%d)\n",i,MyID,nLocal,PrevID,PrevnLocal);
              CompareSplit(MyID,PrevID,nLocal,PrevnLocal,MaxnLocal,ALocal);
        }
        else //с процессором справа
        {
              printf ("\n%d: EVEN2: I is %d(%d) contact with
%d(%d)\n",i,MyID,nLocal,NextID,NextnLocal);
```

```
            CompareSplit(MyID,NextID,nLocal,NextnLocal,MaxnLocal,ALocal);
      }
    }
}
printf("\n%d --  After CYCLE\n",MyID);
for (i=0;i<nLocal;i++)
  printf(" t%1d_%d=%f ",MyID,i,ALocal[i]);
if (MyID==0)
{
   for (i=0;i<P;i++)
     printf(" sn_%d=%i ",i,ASendNum[i]);
   for (i=0;i<P;i++)
     printf(" si_%d=%i ",i,ASendInd[i]);
}

// Сбор результирующего вектора на первом процессоре
MPI_Gatherv(   ALocal,  nLocal, MPI_DOUBLE,   A,  ASendNum,   ASendInd,  MPI_INT, 0,
MPI_COMM_WORLD);
printf("\n rank %d: sizeof(ASendNum)=%d, sizeof(ASendInd)=%d, _msize(ASendNum)=%d,
_msize(ASendInd)=%d,   ASENDNUM (recvcnts): ", MyID, sizeof(ASendNum), sizeof(ASendInd),
_msize(ASendNum), _msize(ASendInd));
for (i = 0; i < (_msize(ASendNum)/sizeof(int)); i++) {
                    printf(" %d", ASendNum[i]);
}
printf("\n rank %d: ASENDIND (displs): ", MyID);
for (i = 0; i < (_msize(ASendNum)/sizeof(int)); i++) {
                              printf(" %d", ASendInd[i]);
}

delete [] ALocal;
delete [] ASendInd;
delete [] ASendNum;

MPI_Finalize();
return 0;
}

void CompareSplit(int AID,int BID,int nA,int nB,int nMax,double *ALocal)
{
double *BLocal;
double *Temp;
int i,j,curA,curB,curTemp;
MPI_Status status;
//int ierr;

BLocal = new double [nMax];
Temp = new double [nMax];

//Пересылаем массивы с Other
```

```
printf("\nIn CompareSplit: A-%d, B-%d----------------------------------------------\n",AID,BID);
MPI_Sendrecv(ALocal,nMax,MPI_DOUBLE,BID,1,BLocal,nMax,MPI_DOUBLE,BID,1,MPI_CO
MM_WORLD,&status);

if (AID<BID)
{
   curA=0;
   curB=0;

         for(curTemp=0;curTemp<nA;curTemp++) {
           if (curA<nA && curB<nB) {
             if (ALocal[curA]<BLocal[curB]) {
           Temp[curTemp]=ALocal[curA++];
       } else {
                              Temp[curTemp]=BLocal[curB++];
                    }
     } else if (curA<nA) {
          Temp[curTemp]=ALocal[curA++];
     } else if (curB<nB) {
          Temp[curTemp]=BLocal[curB++];
     }
         }
   for (i=0; i<nA; i++)
     ALocal[i]=Temp[i];
   if (nA<nMax) ALocal[nMax-1]=0.0;

}
else
{
   curA=nA-1;
   curB=nB-1;

         for(curTemp=nA-1;curTemp>=0;curTemp--)
   {
           if (curA>=0 && curB>=0)
           {
             if (ALocal[curA]>BLocal[curB])
             {
           Temp[curTemp]=ALocal[curA--];
          }
          else
          {
            Temp[curTemp]=BLocal[curB--];
          }
        }
        else if (curA>=0)
          Temp[curTemp]=ALocal[curA--];
        else if (curB>=0)
          Temp[curTemp]=BLocal[curB--];
```

```
      }
   for (i=0;i<nA;i++)
     ALocal[i]=Temp[i];
   if (nA<nMax) ALocal[nMax-1]=0.0;
}

   for (i=0; i<nMax; i++)
   printf(" ar%1d_%d=%f ",AID,i,ALocal[i]);
   printf("\n%d -- ToFinish 3\n",AID);

         delete [] Temp;
         delete [] BLocal;
         return;
}

// Последовательный алгоритм быстрой сортировки
void QuickSort(double* A, int first, int last)
{
   int i = first, j = last;
         double pivot = A[(first + last) / 2];
         double temp;
   do {
      while (A[i] < pivot) i++;
      while (A[j] > pivot) j--;
      if(i <= j) {
         if (i < j) {
                              temp=A[i];
                              A[i]=A[j];
                              A[j]=temp;
                           }
         i++;
         j--;
      }
   } while (i <= j);

   if (i < last)
      QuickSort(A, i, last);
   if (first < j)
      QuickSort(A, first,j);
}
```

Приложение 3. MPI-программа поблочного умножения матрицы на вектор

```
#include "mpi.h"
#include "omp.h"
#include "stdio.h"
#include "math.h"

int main(int argc, char* argv[])
{
  double* pA;  // Первый аргумент – исходная матрица
  double* pb;  // Второй аргумент – исходный вектор
  double* pc;  // Результат умножения матрицы на вектор
  double* pcSerial;  // Результат умножения матрицы на вектор
  int n = -1;        // Размеры исходных матрицы и вектора
  int narray[]={5}; //{6, 150,200,250,400,500};
//,1000,2000,3000,5000,7500,10000,12500,15000,17500,20000,25000,30000,35000,40000,50000};
  int ncount=sizeof(narray)/sizeof(int);

  int P,MyID,MyIDStr,MyIDCol,rnk;
  int Q;
  int nLocal,min_nLocal,nLocalStr,nLocalCol,Rest;
  int strnum,colnum,elemnum=0;
  double* pALocal;
  double* pA_blocks;
  double* pbLocal;
  double* pcLocal;
  int i = -1, j = -1, k = -1;
  double t1 = -1.0, t2 = -1.0, dtSerial = -1.0, dtParallel = -1.0;
  char Pname[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &P);
  MPI_Comm_rank(MPI_COMM_WORLD, &MyID);

  MPI_Get_processor_name( Pname, &rnk );
  printf("\n I am proc %d, %d, name=%s",MyID,rnk,Pname);

  Q=(int) sqrt( (double)P );
  if (P!=Q*Q) {
          printf("\n ERROR!!! P is not a full square!");
          MPI_Finalize();
          return 0;
  }

  for (int iter=0; iter < ncount; iter++) {
    fflush( stdout);
         printf("\n\n");
         MPI_Barrier( MPI_COMM_WORLD);
```

```
        n=narray[iter];
        pA = new double [n*n];
        pA_blocks = new double [n*n];
        pb = new double [n];
        pc = new double [n];

        if (MyID == 0) {
    pcSerial = new double [n];

          //Инициализация pA
                printf("\n A============:\n");
                for (i=0; i<n; i++) {
                        for (j=0; j<n; j++) {
          pA[i*n+j] = j+i-n;
                                printf("  %d - %5.2lf", i*n+j, pA[i*n+j]);
                        }
                        printf("\n");
                }

          //Инициализация pB и pC
          for (i = 0; i < n; i++) {
                  pb[i] = i * 1.0;
                  pc[i] = 0.0;
    }

    t1 = MPI_Wtime();
                printf("\n pcSerial================\n");
                for (i=0;i<n;i++) {
                        pcSerial[i] = 0.0;
                        for (j=0;j<n;j++) {
                          pcSerial[i] += pA[i*n+j]*pb[j];
                        }
                        printf("\n %d - %lf", i, pcSerial[i]);
                }
    t2 = MPI_Wtime();
    dtSerial = t2 - t1;
        } // end of "if (MyID==0)"

  t1 = MPI_Wtime();
        nLocal = n/Q;
        min_nLocal = nLocal;
        Rest = n-Q*nLocal;
        nLocalStr = nLocal;
        nLocalCol = nLocal;
        MyIDStr = MyID/Q;
        MyIDCol = MyID%Q;
        if (MyIDStr < Rest) nLocalStr++;
        if (MyIDCol < Rest) nLocalCol++;
```

```
pALocal = new double [nLocalStr * nLocalCol];
      pbLocal = new double [nLocalCol];
      pcLocal = new double [n];

      for (i=0;i<n;i++)
  pcLocal[i]=0;

      //Перераспределение матрицы по блокам и рассылка по процессорам
      if (MyID==0) {
            int proc_num=0;
        int cur_num=0;
        int str_sum=0, cur_str = 0, col_sum=0, cur_col = 0;

            printf("\n pA_blocks=========\n");
            for (int hrz = 0; hrz < Q; hrz++) {
    cur_str = min_nLocal;
              if (hrz < Rest) cur_str++;
    col_sum = 0;
              for (int vrt = 0; vrt < Q; vrt++) {
                        cur_col = min_nLocal;
                 if (vrt < Rest) cur_col++;
                        for (i = 0; i < cur_str; i++) {
                          for (j = 0; j < cur_col; j++) {
                            pA_blocks[cur_num++] = pA[str_sum*n + col_sum + i*n + j];
                                printf("  %d - %5.02lf", cur_num-1, pA_blocks[cur_num-1]);
                          }
                          printf("\n");
                        }
                        col_sum += cur_col;
                    }
     str_sum += cur_str;
            }
      } //end of  "if (MyID==0)"

      t1 = MPI_Wtime();
      // Распределение массивов для рассылки массивов pA и pc
int *pASendInd = new int [P]; // массив начальных индексов для рассылки pA
int *pASendNum = new int [P]; // массив количеств элементов при рассылке pA
int *pbSendInd = new int [P]; // массив начальных индексов для рассылки pb
int *pbSendNum = new int [P]; // массив количеств элементов при рассылке pb

// Вычисление массивов количества и начальных индексов рассылаемых элементов
for (int pstr=0;pstr<Q;pstr++) {
            if (pstr<Rest)
    strnum=min_nLocal+1;
            else
    strnum=min_nLocal;

  for (int j=0;j<Q;j++) {
```

```
                pbSendNum[pstr+j*Q]=strnum;
        if (pstr > 0)
          pbSendInd[pstr+j*Q]=pbSendInd[pstr-1] + pbSendNum[pstr-1];
                else
                  pbSendInd[j*Q]=0;
      }

              for (int pcol=0;pcol<Q;pcol++) {
        if (pcol<Rest)
                        colnum=min_nLocal+1;
                else
                        colnum=min_nLocal;

                elemnum=strnum*colnum;
                pASendNum[pstr*Q+pcol]=elemnum;

                if (pstr>0 || pcol>0)
          pASendInd[pstr*Q + pcol] = pASendInd[pstr*Q + pcol - 1] + pASendNum[pstr*Q + pcol -
1];
                else
                  pASendInd[0]=0;
      }
        } //end of "for (int pstr=0;pstr<Q;pstr++)"

        // Рассылка строк матрицы и фрагментов вектора
    MPI_Scatterv( pA_blocks, pASendNum, pASendInd, MPI_DOUBLE, pALocal, nLo-
calStr*nLocalCol, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatterv( pb       , pbSendNum, pbSendInd, MPI_DOUBLE, pbLocal, nLocalCol,
MPI_DOUBLE, 0, MPI_COMM_WORLD);

        // pA and pb output of each process
        printf("\n Rank %d: pALocal and pbLocal control", MyID);
        for (i = 0, j = 0; i < pASendNum[MyID]; i++) {
                if (i%nLocalStr == 0) {
                        printf(" | %d - %lf\n", j, pbLocal[j]);
                        j++;
                }
                printf(" %lf", pALocal[i]);
        }

        delete [] pASendInd;
        delete [] pASendNum;
        delete [] pbSendInd;
        delete [] pbSendNum;

    // Параллельное выполнение умножения матрицы на вектор
    //-------------------------------------------------
    int displ;
    if (MyIDStr<Rest)
```

```
      displ = MyIDStr * (min_nLocal+1);
    else
      displ = Rest*(min_nLocal+1) + (MyIDStr-Rest)*min_nLocal;
    for (i=0; i<nLocalStr; i++)
      for (j=0; j<nLocalCol; j++)
        pcLocal[i+displ] += pALocal[i*nLocalCol+j] * pbLocal[j];

        // Сбор результирующего вектора на 0 процессоре
    //-------------------------------------------
        MPI_Reduce( pcLocal, pc, nLocalCol, MPI_DOUBLE, MPI_SUM,
0, MPI_COMM_WORLD);
        if (MyID == 0) {
                printf("\n pc FINAL: ==============");
                for (i = 0; i < n; i++) {
                        printf("\n %d - %lf", i, pc[i]);
                }
        }


        t2 = MPI_Wtime();
    dtParallel = t2 - t1;

        if (MyID==0) {
        // Проверка правильности параллельного умножения и вывод результатов
          double Mistake=0.0;
          for (i=0;i<n;i++)
            Mistake+=fabs(pc[i]-pcSerial[i]);

                printf("\n n=%d dtSerial=%f dtParallel=%f Mis-
take=%f\n",n,dtSerial,dtParallel,Mistake);
                delete [] pcSerial;
        } //end of "if (MyID==0)"
    delete [] pA;
        delete [] pA_blocks;
        delete [] pb;
        delete [] pc;
        delete [] pALocal;
    delete [] pbLocal;
    delete [] pcLocal;
  }
  MPI_Finalize();
}
```

Приложение 4. Текст шаблонов для распространенных ошибок MPI-программ

1) *Несовпадение длин посылаемого и принимаемого сообщений в функциях точка-точка*

Name=Sender’s and receiver’s messages length mismatch

1 block

K=2

2 block

F1=p1:Send_any

F2=p2:Recv_any

3 block

F1(2)!=F2(2)

2) *Взаимно недопустимые типы данных в операциях передачи и приема*

Name=Sender’s and receiver’s data type mismatch

1 block

K=2

2 block

F1=p1:Send_any

F2=p2:Recv_any

3 block

F1(3)!=F2(3)

3) *Несовпадение размеров буферов в разных процессах при коллективной операции*

   a. *для функций перераспределения данных (MPI_Gather,MPI_Scatter, MPI_Alltoall, MPI_Bcast)*

Name=Buffer sizes mismatch in different processes during collective operation (Bcast, Gather, Scatter, Alltoall)

1 block

K=n(MPI_COMM_WORLD)

2 block

FA=pA:Coll_data

3 block

F1(2)!=F2(2)

b. *для редукционных функций (MPI_Reduce, MPI_Scan, MPI_Allreduce)*

Name=Buffer sizes mismatch in different processes during collective reduction operation (Reduce, Scan, Allreduce)

1 block

K=n(MPI_COMM_WORLD)

2 block

FA=pA:Coll_reduc

3 block

F1(3)!=F2(3)

4) *Взаимно недопустимые типы принимаемых и отправляемых данных в векторном варианте коллективной операции*

a. *для функций сбора данных (MPI_Gatherv, MPI_Allgatherv)*

Name=Mismatches of sending and receiving data in vector variant of collective operation for gathering functions

1 block

K=1

2 block

F1=p1:Gatherv||Allgatherv

3 block

F1(3)!=F1(7)

b. для функции рассылки данных (MPI_Scatterv)

Name=Mismatches of sending and receiving data in vector variant of collective operation for scattering function

1 block

K=1

2 block

F1=p1:Scatterv

3 block

F1(4)!=F1(7)

c. для функции перераспределения данных (MPI_Alltoallv)

Name=Mismatches of sending and receiving data in vector variant of collective operation for function Alltoallv

1 block

K=1

2 block

F1=p1:Alltoallv

3 block

F1(4)!=F1(8)

5) Несовпадение кодов редукционной операции в разных процессах

Name=Reduction code mismatch in different processes

1block

K=n(MPI_COMM_WORLD)

2block

FA=pA:Reduce||Allreduce||Reduce_scatter||Scan

3block

F1(5)!=F2(5)

6) Задание пользователем разных номеров root-процессов в разных процессах-участниках коллективной операции (MPI_Bcast).

Name=Different root-process in different processes during collective operation Bcast

1block

K=n(MPI_COMM_WORLD)

2block

FA=pA:Bcast

3block

F1(4)!=F2(4)

7) Несовпадение количества процессов во вновь создаваемой группе

Name=Amount of processes mismatch in new group

1block

K= n(MPI_COMM_WORLD)

2block

FA=pA:Group_incl||Group_excl

3block

F1(2)!=F2(2)

8) Незавершенная коллективная операция

Name=Collective operation called by not all processes

1block

K=1

2block

F1=p1:Coll_data! && Coll_reduc!

9) Выход без вызова функции MPI_Finalize

Name=Exit without MPI_Finalize

1 block

K=1

2 block

F1=p1:Finalize!

10) Неблокирующий прием/передача без последующего вызова MPI_Wait или MPI_Test

Name=Unblock send/receive without freeing request

1 block

K=1

2 block

F1=p1:IPTP

F2=p1:Wait! && p1:Test!

3 block

F1(7) = F2(1)

11) Повторное использование активного запроса обмена

Name=Repeated using of active request

1block

K=1

2block

F1=p1:IPTP

F2=p1:Wait! && Test!

F3=p1:IPTP

3block

F1(7)=F2(1)

F1(7)=F3(7)

12) Неосвобождение определенного пользователем типа данных (MPI_Type_contiguous)

Name=Unfreed user defined datatype

1 block

K=1

2 block

F1=p1:Type_contiguous

F2=p1:Type_free!

3 block

F1(3)=F2(1)

13) Межпроцессные гонки данных, обусловленные применением MPI_ANY_SOURCE

Name=Data race caused by using “MPI_ANY_SOURCE” in receive function

1 block

K=3

2 block

F1=p1:Send_any

F2=p2:Send_any

F3=p3:Recv_any

3 block

F1(4)=p3

F2(4)=p3

F3(4)=MPI_ANY_SOURCE

14) Дедлоки, обусловленные циклом отправок или получений

Name=Deadlock caused by cycle of sends or receives

1 block

K<=n(MPI_COMM_WORLD)

2 block

FA=pA:Send||Ssend||Recv

3 block

Fj(4)=p(j+1), j:=1,K-1

FK(4)=p1

15) Вызов функции отправки типа точка-точка процессом, парный для которого вызвал коллективную операцию

Name=Call of point-to-point function by process, pair of which has called collective operation

1 block

K=2

2 block

F1=p1:Send||Ssend

F2=p2:Recv! && Irecv!

F3=p2:Coll_data||Coll_reduc

3 block

F1(4)=p2

16) Неверный номер процесса приемника/отправителя (отрицательный или выходящий за пределы коммуникатора)

Name=Illegal number of sender/receiver

1 block

K=1

2 block

F1=p1:PTP||IPTP

3 block

F1(4)<0 || F1(4)>=n(MPI_COMM_WORLD)

17) Тэг операции отрицателен или слишком велик

Name=Message tag is negative or too large

1 block

K=1

2 block

F1=p1:PTP||IPTP

3 block

F1(5)<0 || F1(5)>MPI_TAG_UB

18) Передача, которой не удалось поставить в соответствие парную операцию

Name=Unpair send operation

1 block

K=2

2block

F1=p1:Send_any

F2=p2:Recv_any!

3block

F1(4)=p2

19) Отрицательное число отправляемых/принимаемых элементов сообщения

Name=Negative number of transmited elements in collective or point-to-point operation

1block

K=1

2block

F1=p1:PTP||Coll_data

3block

F1(2)<0

20) Изменение данных, участвующих в неблокирующей коммуникации, основным потоком MPI-процесса

Name=Data race due to unblock point-to-point function

1block

K=2

2block

F1=p1:Isend||Issend||Ibsend||Irsend

F2=p2:Recv_any

3block

CRC_BEFORE(F1(1)) != CRC_AFTER(F2(1))

21) Неверный режим доступа при открытии файла

Name= Incorrect access mode for file opening

1block

K=1

2block

F1=p1:File_open

3block

F1(3)!=MPI_MODE_RDONLY && F1(3)!=MPI_MODE_RDWR && F1(3)!=MPI_MODE_WRONLY && F1(3)!=MPI_MODE_CREATE && F1(3)!=MPI_MODE_EXCL && F1(3)!=MPI_MODE_DELETE_ON_CLOSE && F1(3)!=MPI_MODE_UNIQUE_OPEN && F1(3)!=MPI_MODE_SEQUENTIAL && F1(3)!=MPI_MODE_APPEND

22) Отсутствие закрывающей функции для использованного файла

Name= Missing closing function for opened file

1block

K=1

2block

F1=p1:File_open

F2=p1:File_close!

23) Коммуникации между процессами, разность номеров которых более единицы при пересылке данных по кольцу (пользовательская проверка)

Name=Communications between processes, which difference of numbers more than 1 during transmit of data by ring

1 block

K=n(MPI_COMM_WORLD)

2 block

FA=pA:Send_any

3 block

Fj(4)=p(j+1), j:=0,K-2

F(K-1)(4)=p1

p2-p1>1

24) Превышение количества передаваемых элементов заданного значения (пользовательская проверка)

Name=Amount of transmited elements more than defined value (15)

1 block

K=1

2 block

F1=p1:Send_any

3 block

F1(2)>15

Приложение 5. Свидетельство о регистрации разработанного программного средства

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2012610206

«Распределенная автоматизированная система обнаружения логических ошибок в MPI-программах»

Правообладатель(ли): ***Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Кемеровский государственный университет» (КемГУ) (RU)***

Автор(ы): ***Афанасьев Константин Евгеньевич, Власенко Андрей Юрьевич (RU)***

Заявка № **2011618295**
Дата поступления **7 ноября 2011 г.**
Зарегистрировано в Реестре программ для ЭВМ
10 января 2012 г.

Руководитель Федеральной службы по интеллектуальной собственности, патентам и товарным знакам

Б.П. Симонов

Приложение 6. Акт о внедрении программного средства

МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Кемеровский государственный университет» (КемГУ)

650043, г.Кемерово, ул. Красная, 6
Телефон: (3842) 58-84-39. Факс: (3842) 58-53-50
E-mail: rector@kemsu.ru. http://www.kemsu.ru

18 фев 2014 № 257/05

УТВЕРЖДАЮ:

Проректор по научной работе,

д.б.н., профессор

_____ В.Г. Дружинин

"___" _____ 2014 г.

АКТ о внедрении программного обеспечения «Система автоматизированного контроля корректности MPI-программ» в опытную эксплуатацию

Комиссия в составе:

председатель: начальник управления информатизации, д.т.н., доцент Гудов А.М.;

члены комиссии: начальник научного управления, д.т.н., профессор Шадрин А.В., директор центра новых информационных технологий КемГУ; к.ф.-м.н., доцент Карабцев С.Н.
составили настоящий акт о том, что программное обеспечение «Система автоматизированного контроля корректности MPI-программ», созданное в рамках диссертационной работы Власенко Андрея Юрьевича, внедрено в деятельность центра коллективного пользования «Высокопроизводительные параллельные вычисления» ФГБОУ ВПО «Кемеровский государственный университет» (ЦКП по ВПВ КемГУ).

«Система автоматизированного контроля корректности MPI-программ» была внедрена в опытную эксплуатацию 20 января 2014 г.

Внедренная система обладает следующими основными характеристиками:

- Автоматизированное обнаружение семантических ошибок в параллельных программах, использующих коммуникационный интерфейс MPI.
- Возможность определения пользователем ситуаций в MPI-приложении, которые должны быть обнаружены системой во время работы данного приложения посредством создания собственных файлов шаблонов.
- Возможность ограничения пользователем числа ситуаций, на возникновение которых необходимо проверить параллельную программы, путем отключения любого количества имеющихся файлов шаблонов.

- Отсутствие ограничений на количество MPI-процессов и, соответственно, количество вычислительных узлов кластера, на котором работает параллельная программа под управлением системы.
- Выдача в текстовом файле по окончанию работы системы подробной информации о найденных в параллельной программе случаях соответствия поведения приложения с ситуациями, заданными в текстовых шаблонах.

В ходе эксплуатации системы автоматизированного контроля корректности MPI-программ подтверждено, что она соответствует предъявленным к ней требованиям. Система позволила уменьшить время отладки и локализации ошибок в параллельных программах пользователей ЦКП по ВПВ КемГУ.

Председатель комиссии:

Начальник управления информатизации КемГУ,

д.т.н., доцент А.М. Гудов

Члены комиссии:

начальник научного управления КемГУ,

д.т.н., профессор А.В. Шадрин

директор центра новых информационных технологий КемГУ,

к.ф.-м.н., доцент С.Н. Карабцев