Федеральное государственное бюджетное учреждение науки
Институт системного программирования
Российской академии наук

На правах рукописи

Акопян Манук Сосович

# Инструментальные средства поддержки автоматизированной разработки параллельных программ

05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
к. ф.-м. н. С.С. Гайсарян

Москва – 2015

## Оглавление

# 1. Введение

**Актуальность работы**

Современные высокопроизводительные вычислительные системы строятся по так называемой кластерной архитектуре (все системы из списка Top500), то есть являются системами с распределенной памятью. Узлами таких систем являются серверы, состоящие из нескольких процессоров, имеющих многоядерную архитектуру. Кроме того, в составе таких узлов могут использоваться акселераторы.

Таким образом, аппаратура современных вычислительных систем обеспечивает возможность параллельного выполнения вычислений на всех уровнях: на уровне команд (за счет возможности одновременной выдачи нескольких команд, дублирования функциональных устройств, конвейеризации и векторизации вычислений), на уровне параллельно выполняющихся потоков (каждый поток запускается на отдельном ядре), на уровне параллельно выполняющихся процессов (каждый многопоточный процесс запускается на отдельном узле кластера). Потенциально это может обеспечить очень высокую производительность вычислительных систем.

Однако практическое использование возможностей современной аппаратуры для организации высокоэффективных вычислений требует решения сложных оптимизационных проблем на всех уровнях параллельного выполнения вычислений. В результате задача автоматического распараллеливания программ оказывается слишком сложной. В настоящее время она успешно решается лишь для узкого класса программ, допускающих параллельное выполнение без синхронизации. Как следствие, несмотря на активные исследования по новым высокоуровневым языкам параллельного программирования (X10, Chapel, и др.), фактическим стандартом разработки приложений для высокопроизводительных систем является последовательный язык программирования, расширенный стандартной библиотекой передачи сообщений (как правило, это библиотека *MPI*), которая содержит разнообразные возможности организации обмена

данными между параллельными ветвями программы. В частности, это связано с тем, что для организации эффективных вычислений требуются различные процедуры обмена для различных классов приложений. Таким образом, эффективность прикладной *MPI*-программы существенно зависит от того, какие процедуры библиотеки *MPI* в ней использованы и как размещены в программе обращения к этим процедурам. В отсутствие оптимизирующих компиляторов выбор процедур библиотеки MPI и расстановка обращений к ним осуществляются разработчиком вручную, то есть фрагменты параллельной программы, от которых в наибольшей степени зависят ее качества (масштабируемость, эффективность), разрабатываются практически на ассемблерном уровне. Поэтому разработка параллельных программ необходимого качества требует затрат квалифицированного ручного труда. Компромисс между необходимым качеством разрабатываемой программы и затраченными на ее разработку ресурсами будем называть продуктивностью. Таким образом, продуктивность достигается минимизацией затрачиваемых ресурсов при достижении необходимого качества, в частности, посредством автоматизации процесса разработки параллельных программ.

При этом в отличие от разработки последовательных программ, где достаточно хорошо развиты технологические средства, в настоящее время не существует устоявшихся технологических процессов, позволяющих разрабатывать параллельные программы требуемого уровня эффективности при минимизации ресурсов, затрачиваемых на разработку и сопровождение, то есть обеспечить необходимый уровень продуктивности.

Проблема повышения продуктивности параллельных вычислений становится особенно острой в связи с необходимостью использования гибридных моделей программирования (MPI+OpenMP, MPI+OpenACC, MPI+Cuda и др.) из-за усложнения аппаратуры узлов высокопроизводительных систем.

В этих условиях повышение продуктивности может быть достигнуто путем разработки инструментальных средств и соответствующих технологических процессов, позволяющих разработчику сократить время разработки и

сопровождения прикладной программы. Один из возможных наборов таких инструментов предлагает интегрированная среда ParJava. Она позволяет автоматически построить модель разрабатываемой JavaMPI-программы и исследовать ее свойства (например, получать необходимые оценки времени выполнения), используя интерпретацию модели. Причем интерпретатор может работать как на целевой системе, так и на инструментальном компьютере с минимальным использованием целевой системы лишь для отладочного прогона. В последнем случае, время и ресурсы, затрачиваемые не интерпретацию, сокращаются.

Однако среда ParJava обладает рядом ограничений: модель параллельной программы в ней не учитывает многоядерности современных процессоров; в реализации существует ограничение на размер памяти моделируемой параллельной программы; метод оценки времени выполнения базовых блоков не учитывает динамическую компиляцию (JIT) в Java.

Существует множество инструментов профилирования и визуализаторов трасс, в результате работы которых создаются таблицы и графики с различными статистиками. Однако в большинстве случаев предлагаемые инструменты требуют ручной обработки выходных данных. Количество обрабатываемой вручную информации увеличивается существенно с количеством ядер, процессов и объемом данных параллельной программы. Назовем коммуникационным шаблоном MPI наличие в трассе параллельной программы событий (например, send, recv), связанных определенными соотношениями (например, допустимыми величинами разности временных меток событий). В настоящее время известно несколько шаблонов, наличие которых свидетельствует о возможной потере производительности в параллельной программе. Автоматизация поиска коммуникационных шаблонов MPI позволила бы повысить продуктивность разработки.

Таким образом, актуальной является задача создания инструментальных средств, обеспечивающих сокращение доли ручной работы при разработке

прикладных параллельных программ для современных высокопроизводительных систем с многоядерными узлами.

**Цель диссертационной работы.**

Разработка методов и реализация соответствующих инструментов автоматизированного обнаружения коммуникационных шаблонов, как во время реального выполнения программ, так и на основе моделирования программ.

**Научная новизна.**

В диссертационной работе получены следующие основные результаты, обладающие научной новизной:

1. Разработана интерпретируемая модель параллельной программы, позволяющая оценивать границы масштабируемости параллельных Java-программ для современных высокопроизводительных вычислительных систем с распределенной памятью, строящихся на основе многоядерных узлов (взаимодействие между процессами осуществляется посредством MPI, а внутри процесса используются Java-потоки).
2. Разработан и реализован метод интерпретации модели, обеспечивающий интерпретацию реальных параллельных приложений за приемлемое время, как на целевой вычислительной системе, так и на инструментальном компьютере. В том числе обеспечивается учет изменений, вносимых динамическим компилятором времени выполнения.
3. Разработан метод автоматизированного обнаружения коммуникационных шаблонов MPI (как на основе реальной, так и модельной трассы), приводящих к потере производительности.

**Теоретическая и практическая значимость.**

Разработанные инструментальные средства реализованы в среде ParJava, которая интегрирована в виде подключаемого модуля в открытую среду разработки Eclipse. Интегрированная среда позволяет разрабатывать параллельные приложения на языке Java с использованием библиотеки MPI на

системах с распределенной памятью (для обмена сообщениями между процессами параллельной программы) и с использованием библиотеки потоков Java. Это обеспечивает платформо-независимость разрабатываемых приложений, возможность использования общей памяти между потоками процесса на одном узле вычислительной платформы и существенно уменьшает накладные расходы на разработку, доводку и сопровождение параллельного приложения.

**Апробация работы**

Основные результаты диссертации обсуждались на следующих конференциях и семинарах:

1. V Всероссийская межвузовская конференция молодых ученых, Санкт-Петербург, Апрель 15-18, 2008 г.
2. Научно-исследовательский семинар Института системного программирования РАН.
3. 10th International Conference on Computer Science and Information Technologies (CSIT 2015), September 28- October 02, Yerevan, Armenia

**Публикации**

Основные результаты работы опубликованы в статьях [1] – [7]. Работы [1] – [6] опубликованы в изданиях по перечню ВАК. Статья [7] опубликована в сборнике трудов конференций. В совместных работах [2, 3] личный вклад автора состоит в разработке и реализации предлагаемой в диссертации новой модели параллельной программы и ее интерпретатора, обеспечивающего интерпретацию реальных параллельных приложений за приемлемое время. В совместной работе [5] вклад автора состоит в разработке предлагаемого в диссертации метода автоматизированного обнаружения коммуникационных шаблонов MPI.

**Личный вклад автора**

Все представленные в диссертации результаты получены лично автором.

**Структура и объем диссертации**

Работа состоит из введения, шести разделов, заключения, списка литературы и одного приложения. Общий объем диссертации составляет 129 страниц, включая 33 иллюстрации. Библиография содержит 71 наименование.

# 2. Обзор инструментальных сред, поддерживающих разработку параллельных программ

При разработке каждой процедуры (функции, метода) параллельной программы для распределенной вычислительной системы прикладной программист сталкивается со следующими вопросами: допускает ли разрабатываемая процедура параллельное выполнение, масштабируемое на произвольное число узлов (процессоров)? Если нет, то каковы границы ее области масштабируемости? Какое ускорение можно получить на n узлах? Какие изменения внести в программный код, чтобы обеспечить наибольшее ускорение?

Получить ответы на перечисленные и другие подобные вопросы помогают инструментальные среды, поддерживающие разработку параллельных программ. Такие среды разрабатываются в различных странах с начала 90-х годов. Рассмотрим некоторые особенности организации, функционирования и применения инструментальных сред.

В инструментальных средах, в отличие от отладчиков, широко используются модель разрабатываемой процедуры (или всей программы). Использование модели позволяет ускорить интерпретацию параллельного приложения, так как в модели рассматриваются только те структуры, которые влияют на исследуемые характеристики параллельного приложения (например, на время выполнения, границу области масштабируемости). Интерпретатор модели позволяет анализировать модель, выдавая значения исследуемых характеристик.

Модель можно определить таким образом, что ее интерпретацию можно будет осуществлять не только на целевой вычислительной платформе (кластере), но и на инструментальном компьютере. Это создает более комфортные условия разработчику параллельного приложения, так как позволяет вести разработку в автономном режиме. Интерпретация модели требует значительно меньших ресурсов и выполняется гораздо быстрее, чем реальное выполнение

параллельного приложения. Кроме того, результаты интерпретации легко параметризировать по количеству используемых узлов целевой вычислительной платформы, что позволяет сократить количество интерпретаций.

Известны два подхода к моделированию параллельных приложений: модели, основанные на симуляции трасс (*trace driven simulation)* и модели, основанные на симуляции дискретных событий (*discrete event driven simulation*). Первый подход используется в средах (системах) разработки параллельных приложений *DiP (Dimemas)* [8], *TAU* [9], *PERC* [10, 11], DVM [12]. Второй подход – в средах (системах) *POEMS* [13] и *WARPP* [14, 15].

Рассмотрим инструментальные среды, использующие модели, основанные на симуляции трасс. Такие системы работают примерно по следующей схеме: сначала моделируемое приложение инструментируется и выполняется на целевой платформе для сбора его трассы. *Трасса параллельной программы* представляет собой набор трасс ее процессов. *Трасса процесса* представляет собой последовательность событий разного типа. В каждой системе определяется свой набор типов событий. Исходная трасса получается при выполнении инструментированной программы на целевой вычислительной платформе. Анализ трассы приложения позволяет оценить значения его динамических характеристик и оценить качество разрабатываемой параллельной программы.

Полученная трасса переносится на инструментальную машину и вместе с конфигурационным файлом, определяющим контекст, в котором должна выполняться интерпретация, передается интерпретатору трассы. В конфигурационном файле описываются параметры, влияние которых на выполнение программы исследуется в данной конкретной среде. Интерпретатор трассы переопределяет трассу с учетом контекста, определенного конфигурационным файлом. Полученная при выходе интерпретатора трасса передается на вход модулю визуализации трасс, что позволяет пользователю вручную обнаружить узкие места (проблемные фрагменты) разрабатываемой прикладной программы. Анализируя проблемные фрагменты, прикладной программист может внести изменения в разрабатываемую программу, после чего

повторить описанный процесс для выявления новых проблемных фрагментов. Такой итеративный подход позволяет вести разработку параллельного приложения, постепенно улучшая его качество.

Следует рассмотреть, что представляет собой трасса в исследуемых системах.

В системе DiP трасса каждого процесса представляет собой события работы процессора (CPU burst) и коммуникаций. Это дает возможность исследовать сбалансированность вычислительного узла с коммуникационной сетью. Также предусмотрена возможность создания пользовательских событий. В системе PERC [10] трасса представляет собой частотный профиль для каждой инструкции.

Следует отметить среду TAU, которая предоставляет набор программных средств для инструментирования приложения. Благодаря развитой библиотеке инструментирования среда TAU позволяет получить богатый набор типов событий в трассе – пользовательские события (интересующие пользователя фрагменты обрамляются инструментальными вставками вручную), коммуникационные события, события на основе компонентов (Component-Based) прозрачные с точки зрения языка и местонахождения и.т.д.

Построение трассы параллельного приложения состоит из следующих этапов:

1. Инструментирование программы. Инструментирование программы представляет собой добавление в определенных позициях оригинальной программы вызовы к инструментальной библиотеке. Во время выполнения программы эти вызовы регистрируют наступление определенного события и производят запись в трассу.
2. Выполнение инструментированной программы на целевой платформе. Инструментированная программа переносится на целевую платформу и производится запуск параллельной программы. В результате для каждого процесса программы создается его трасса.
3. Компенсация накладных расходов. Во время выполнения программы вызовы к инструментальной библиотеке, вставляемые в оригинальную программу, добавляют накладные расходы. Для исключения из модели

времени затраченное на накладные расходы, вычисляется время работы вызовов функций инструментальной библиотеки и вычитается из модельного времени.

TAU позволяет разбить большие события на более мелкие (вплоть до инструкций). Степень детализации может регулировать пользователь - пользователь может определять события на уровне исходного кода, включать и выключать группы событий и.т.д. После этого производится автоматическое инструментирование параллельной программы во время ее препроцессирования. В состав системы входит компилятор, который позволяет производить инструментирование параллельной программы во время ее компиляции, учитывая изменения, вносимые оптимизирующими фазами компилятора. Кроме того, в системе предусмотрена возможность динамического инструментирования бинарного кода без изменения оригинальной программы. Библиотека DyninstAPI [16] позволяет инструментировать работающую программу, вставляя в выполняемый код программы вызовы к инструментальной библиотеке на лету. Для языка Java реализован аналогичный модуль, который взаимодействует с интерфейсом JVMTI [17], который позволяет с помощью динамических функций профилирования инструментировать приложение без изменения исходного кода или байт-кода.

Интерпретатор (симулятор) трассы получает на вход трассу параллельного приложения и конфигурационный файл. Работа интерпретатора представляет собой сопоставление трассы на инструментальной машине с конфигурационным файлом. Конфигурационный файл содержит параметры, описывающие характеристики аппаратуры и коммуникационной сети на вычислительной платформе. Каждая среда разработки определяет свой набор параметров. Меняя эти параметры, на интерпретаторе трассы можно исследовать поведение данной трассы на различных процессорах, коммуникационных сетях и.т.д. Результатом симуляции (интерпретации) трассы является новая трасса, отражающая особенности выполнения программы на архитектуре, описываемой конфигурационным файлом.

В интерпретаторе DIMEMAS для вычисления времени коммуникаций используется простая линейная модель (латентность и пропускная способность). Моделируются блокирующие и не блокирующие передачи сообщений. В моделируемой архитектуре узел представляет собой SMP-машину с одним или несколькими процессорами и локальной памятью. Конфигурационный файл содержит такие параметры как скорость процессора, локальные коммуникации (латентность и пропускная способность при передаче сообщений между процессорами в рамках одного узла), латентность (пропускная способность) сетевых коммуникаций (латентность сети при передаче сообщений между узлами), количество входных и выходных связей (максимальное количество параллельных коммуникаций с узлом) и.т.д.

Симулятор DIMEMAS был использован для исследования топологии сети в проекте, разработанном суперкомпьютерным центром в Барселоне совместно с IBM [18, 19]. Модель сети включает в себя модули адаптера и коммутатора. Модули коммутаторов упорядочены в топологии «толстое дерево» с изменяемым количеством уровней. Детальное моделирование сети позволяет решать такие задачи как определение оптимального количества коммутаторов, количества уровней толстого дерева, размера буферов в коммутаторах, оптимального протокола маршрутизации пакетов, пропускной способности канала для каждого уровня в иерархии при использовании многоуровневых каналов. Основная цель данного проекта - исследование и оптимизация аппаратных характеристик коммуникационной сети.

Результаты, полученные во время симуляции, передаются модулю визуализации. Инструменты визуализации позволяют отобразить трассы, результаты статического анализа трассы в графическом виде. Трассы обычно визуализируют в виде MSC-диаграммы. В рассматриваемых системах либо реализованы собственные средства визуализации (Paraver [20], ParaProf [21]), либо реализованы конвертеры к известным форматам трасс, что позволяет использовать сторонние инструменты для анализа и визуализации трасс (например, Vampir [22]).

Система Vampir переводит трассу программы во множество графических представлений, таких как диаграмма состояний, отображение временной шкалы, статистики разного рода. Система поддерживает набор фильтров, которые позволяют выделять интересующие пользователя фрагменты (события). К достоинствам Vampir относится демонстрация временной шкалы, которая графически отображает частичную упорядоченность событий в разных процессах на горизонтальной оси времени; представление матрицы сообщений, (i,j)-ая ячейка которой представляет собой суммарное время пересылки сообщений от процесса i к процессу j.

Система визуализации Paraver [20] позволяет вначале использовать глобальное представление поведения приложения с последующим детализированным представлением задачи. Paraver предлагает единое представление данных большого объема. Модуль статического анализа Paraver позволяет увидеть такие статистические данные как время использования процессора, время блокирования в ожидания сообщения, время блокирования в ожидании свободных процессоров, количество и размер посланных (полученных) сообщений и.т.д.

При разработке параллельных приложений с помощью систем на основе симуляции трассы часто генерируется значительный объем данных. Это обусловлено тем, что при итеративной разработке приложения постепенно накапливается большое количество трасс и результатов работы анализаторов. Возникает необходимость в хранении и дальнейшем использовании этих данных. В TAU для решения этой проблемы был разработан инструмент PerfDMF, который обеспечивает базу для разбора, хранения, запроса и анализа данных производительности из разных экспериментов, версий приложения, инструментов профилирования и.т.д. На данный момент реализована поддержка таких СУБД как PostgreSQL, MySQL, Oracle и DB2.

Наиболее полно технология доводки параллельных программ с помощью моделей, основанных на интерпретации трасс, реализована в открытой среде разработки *TAU* (*Tuning and Analysis Utilities*), разрабатываемой в Университете

штата Орегон, Лос Аламосской Национальной Лаборатории и Научно-исследовательском Центре *Jülich, ZAM* (Германия). Среда позволяет исследовать динамические и статические характеристики параллельных программ, написанных на языках C++, C, Fortran, Java, Python. В среде реализован собственный анализатор и визуализатор трасс ParaProf. Трассы генерируются в собственном формате TAU, однако существуют трансляторы на известные форматы трасс. Это позволяет использовать сторонние инструменты для анализа и визуализации трасс (Vampir, JumpShot, Paraver, Expert и.т.д.).

Система DiP [8] разработана в суперкомпьютерном центре Барселоны. Основные компоненты системы это интерпретатор DIMEMAS и визуализатор трасс Paraver. Интерпретатор DIMEMAS восстанавливает поведение параллельной программы, основываясь на множестве трасс приложения и конфигурационном файле. DiP был разработан в 1993 г, а с 1996 г доступен как коммерческий инструмент от Pallas. Разработка приложения ведется итеративным образом: вначале параллельная программа инструментируется и производится отладочный запуск программы на целевой платформе для генерации трассы. Полученные трассы передаются на вход интерпретатору DIMEMAS на инструментальной машине. Интерпретатор восстанавливает поведение целевой платформы на инструментальной машине и создает новые трассы. Полученные трассы передаются инструменту Paraver, который производит анализ и визуализацию трасс. На основе полученных данных пользователь может изменить исходный код программы. Фазы симуляции и визуализации повторяются до тех пор, пока не будет достигнут желаемый результат (ускорение).

Симуляция трассы позволяет определить, как повела бы себя данная программа на вычислительной платформе с другими параметрами. Однако при исследовании масштабируемости параллельного приложения данный подход не совсем удобен. Все данные, которые можно получить в результате симуляции, жестко привязаны к данной трассе (точнее к количеству процессов и входным данным). При исследовании масштабируемости по Амдалю входные данные

остаются неизменными, меняется лишь количество процессов. В некоторых работах пытаются прогнозировать трассу при запуске m процессов, основываясь на трассе при запуске n процессов (m > n). При этом применяются в основном методы регрессионного анализа, однако, точность предсказания не очень высокая. Метод симуляции на основе трассы больше подходит для исследования характеристик создаваемой вычислительной платформы при ее построении.

DVM-система, разработанная в Институте прикладной математики им. М.В. Келдыша РАН [12], предназначена для создания переносимых оптимизированных вычислительных приложений на языках C-DVM и Fortran-DVM для многопроцессорных компьютеров с общей и распределенной памятью, включая и гибридные системы, в узлах которых вместе с универсальными многоядерными процессорами используются в качестве ускорителей и графические процессоры. В состав системы входят множество инструментов анализа и отладки параллельных приложений (компиляторы, отладчик, предсказатель производительности, трассировщик и.т.д.).

Система поддержки выполнения DVM-программы позволяет по трассам выполнения программы на m процессоров определять характеристики выполнения программы при использовании n процессоров.

Вначале исследуемая DVM-программа выполняется на инструментальном компьютере, на некотором количестве процессоров с целью сбора трассы. Модуль прогнозирования состоит из двух компонент: интерпретатора трассы и оценщика времени. По трассе DVM-программы и заданным пользователем параметрам целевой вычислительной системы интерпретатор трассы вычисляет временные характеристики выполнения данной программы на целевой вычислительной системе, вызывая функции оценщика времени.

Выполнение программы рассматривается не с точностью до выполнения операторов, а в виде последовательности работы более крупных фрагментов программы. Параллельное выполнение DVM-программы организуется с помощью функций системы поддержки выполнения, которая является частью DVM-системы. Вызовы этих функций вставляются при компиляции DVM-

программы. Участки программы между вызовами таких функций считаются терминальными и характеризуются только временем их выполнения.

Моделирование параллельного выполнения программы опирается на модель оценки времени выполнения фрагментов программы и на модель оценки времени коммуникаций. Для оценки времен выполнения предлагается использовать одну из двух моделей. Первая модель основывается на измерении времен выполнения на инструментальном компьютере и пересчете этих времен для целевой системы. Вторая модель использует аналитические оценки времен и последующий пересчет этих времен для целевой системы.

Замеры времен на инструментальном компьютере представлены в трассах выполнения программы на некотором количестве процессоров. В трассе собрана информация о последовательности вызовов функций системы поддержки, времена их работы и времена работы участков программ между вызовами [23].

Оценка времен коммуникаций опирается на многоуровневую модель. На самом нижнем уровне находятся ядра одного процессора. Нижние уровни соединяются между собой через новый уровень, который представлен несколькими каналами связи с заданными для каждого характеристиками коммуникационной сети. В DVM-программе могут использоваться несколько видов коммуникационных операций: обмен данными через теневые грани массивов, редукционные операции, копирование секций массивов, удаленный доступ к секциям массива, обмен данными при организации конвейерного выполнения витков цикла.

Следует отметить, что в системе DVM при отладке производительности программы пользователь не обязательно должен запускать ее с таким объемом исходных данных, какой будет характерен при решении реальных задач. Он может, например, ограничить количество регулярно повторяющихся внешних итераций одной-двумя итерациями.

Таким образом, в системе DVM реализован один из вариантов преобразования трассы: трасса собирается на инструментальном компьютере и пересчитывается для целевой вычислительной системы, как и в средах TAU и DiP.

Ниже рассматриваются инструментальные среды, использующие модели, основанные на симуляции дискретных событий и использующие метод прямого выполнения.

Разработка параллельного приложения в таких системах в основном ведется на инструментальной машине. Процесс моделирования прикладного приложения организован по следующей схеме: 1) построение модели и инструментирование параллельной программы; 2) отладочный запуск инструментированной программы для получения временного профиля; 3) анализ результатов 2-ой фазы и внесение отладочных данных в модель программы; 4) интерпретация модели с учетом конфигурационного файла, описывающего коммуникационную сеть; 5) построение графика масштабируемости. Для повышения продуктивности все фазы, кроме третьего, можно выполнить на инструментальной машине. Отладочный запуск проводится на целевой вычислительной платформе, что обеспечивает более точные результаты прогнозирования.

Модель параллельного приложения формируется из модели вычислений, модели потока управления и модели коммуникаций программы. Модель вычислений представляет собой участки линейных инструкций (базовые блоки). Модель потока управления отражает логическую структуру программы (например, граф потока управления, абстрактное синтаксическое дерево и.т.д.). Также модель потока управления обогащается данными, полученными при отладочном запуске инструментированной программы на целевой платформе. Модель коммуникаций представляет собой набор базовых примитивов и правил, которые моделируют функции MPI.

В среде POEMS модель программы представляет собой статический граф задачи (СГЗ), где каждая вершина представляет множество параллельных задач (task). Две вершины могут быть соединены несколькими ребрами разных типов – ребра «предшествования», описывающие поток управления либо синхронизацию; «коммуникационные» ребра, описывающие явную передачу данных между процессами. СГЗ строится с помощью компилятора dHPF.

В системе WARPP [14] в модели параллельной программы явно выделяются

базовые блоки, вызовы коммуникационных функций и операции ввода вывода. Вначале проводится анализ исходного кода, выделяются базовые блоки программы, и строится граф потока управления для каждого процесса параллельного приложения. Помимо этого, для каждого базового блока встраивается инструментальный код, измеряющий его время выполнения. На этой стадии построения модели строится инструментированная версия исходного кода и базовая модель приложения, описывающая поток управления.

Симуляция (интерпретация) модели программы осуществляется созданием в интерпретаторе множества «виртуальных процессоров», соответствующих процессорам на целевой системе. «Виртуальный процессор» поддерживает поток управления, вычисление выражений, локальные часы и.т.д. «Виртуальный процессор» представляет ядро в многоядерной системе, либо процессор в однопроцессорной системе. Во время симуляции «виртуальный процессор» ходит по потоку управления выполняет инструкции (вычисляет выражения) и генерирует события. В модели программы различаются несколько видов событий (вычислительное событие, коммуникационное событие, событие 'wait' и.т.д.). Выполнение симуляции продолжается переключением потока управления между одним из «виртуальных процессоров» и обработчиком событий внутри интерпретатора. «Виртуальный процессор» выполняет поток управления модели симуляции, останавливаясь, когда достигнуто событие, и передает поток управления обратно интерпретатору для обработки.

Обработка вычислительного события представляет собой реальное выполнение соответствующего фрагмента на инструментальной машине с добавлением времени выполнения этого события к счетчику времени виртуального процессора. Для события “wait” интерпретатор генерирует следующее событие в «виртуальном процессоре» и, определив как долго процессор оставался незанятым (idle), добавляет это время к счетчику времени. В разных системах вычислительное событие может представлять как отдельную инструкцию, так и базовый блок инструкций.

Коммуникационные события передаются специальным обработчикам в

интерпретаторе, которые проверяют, что отправитель и получатель готовы к передаче сообщения и заданные ими параметры сообщения корректны (размер сообщения, тэг, адресат). Для вычисления времени передачи сообщения вначале определяется тип сетевого взаимодействия (core2core, Infiniband …). После этого используются значения соответствующих параметров сетевого взаимодействия (латентность, ширина полосы пропускания и.т.д.). Передача данных не производится в этой точке, интерпретатор останавливает соответствующий процессор до тех пор, пока он не потребуется, записывая это время как “wait” событие. Время каждого события записывается по отношению к локальному времени виртуального процессора.

Симуляция заканчивается, когда все виртуальные процессоры полностью завершили симуляцию.

Моделирование программы с помощью симуляции дискретных событий позволяет с минимальным использованием целевой аппаратуры исследовать задачу масштабируемости параллельного приложения. Однако, поскольку симуляция модели ведется на инструментальной машине, очень важным фактором является то, какие технологии применяются для решения проблем интерпретации программ с большим объемом данных, и для обеспечения высокой скорости интерпретации. В проекте POEMS для решения проблемы моделирования задач с большим объемом данных предложили симулировать не всю программу, а только поток управления программы. Время вычислительных фрагментов измеряется во время отладочного запуска приложения на целевой платформе и используется при симуляции на инструментальной машине. Таким образом, можно существенно сократить ресурсы, требуемые при симуляции модели. В системе WARPP эта идея была реализована.

Интерпретация модели позволяет предсказать время выполнения параллельного приложения на вычислительной платформе. Интерпретация модели параллельного приложения проводится много раз (для разного количества используемых процессов). После этого полученные результаты используются для построения графика масштабируемости параллельного

приложения.

Для моделирования параллельного приложения в POEMS интегрирован инструмент MPI-Sim [24]. В MPI-Sim модель приложения представляется как чередование блоков локального кода и коммуникационных функций. Блоки локального кода интерпретируются с помощью метода прямого выполнения, а для коммуникационных функций реализована библиотека MPI-LITE. В библиотеке MPI-LITE реализовано несколько основных не блокирующих коммуникационных функций MPI точка-точка (базовые примитивы). MPI-Sim не интерпретирует на прямую все коммуникационные функции MPI - коллективные функции транслируются в коммуникации точка-точка, а они, в свою очередь, реализованы в виде комбинации базовых примитивов. Время выполнения коммуникаций в MPI-LITE определяется с помощью модели LogP [25].

В среде WARPP время для каждого вычислительного события получается посредством инструментирования исходного кода счетчиками времени [26]. Для циклов, содержащих один блок (однородные циклы), счетчики ставятся непосредственно перед и после цикла, и время события, соответствующего телу цикла, получается делением разности счетчиков на количество итераций. Для обычных блоков время измеряется разностью счетчиков. Поскольку блок может выполняться много раз во время выполнения приложения, то его время усредняется. В модели MPI-функции заменяются «оберточными» функциями, которые останавливают учет модельного времени и снова запускают часы после обработки вызова. Это позволяет отделить время коммуникаций от времени вычислений.

Также следует отметить систему Scalasca [27], предоставляющую набор инструментов, предназначенных для анализа производительности. Scalasca поддерживает измерение и анализ некоторых конструкций MPI, OpenMP и гибридных программных конструкций. Система поддерживает инкрементальный анализ производительности, который совмещает информацию времени выполнения с глубоким анализом характера параллелизма с помощью трассы событий.

Вначале параллельное приложение инструментируется. Пользователь может выбрать из двух вариантов: либо создание итогового отчета (профиль) на основе различных метрик для вызовов функций, либо сбор трассы событий времени выполнения. Итоговый отчет представляет собой набор различных представлений статистических данных о производительности параллельной программы. Когда возможность трассировки включена, каждый процесс создает файл трассы, содержащий записи для локальных событий данного процесса. После этого локальные трассы разных процессов сливаются в единую трассу. В трассе могут встречаться только терминальные события (SEND, RECV и.т.д.). Каждое событие помимо других свойств содержит также временную метку. Инструмент EXPERT можно использовать для выявления коммуникационных шаблонов. Инструмент EXPERT последовательно сканирует события в глобальной трассе и пытается отыскать коммуникационные шаблоны.

На основе проведенного анализа существующих систем, поддерживающих разработку параллельных программ, были выработаны требования к среде разработки параллельных программ.

Среда должна позволять собирать временной профиль параллельной программы на целевом кластере.

Среда должна позволять интерпретировать параллельную программу, как на целевом кластере, так и на инструментальном компьютере.

Среда должна адекватно отражать поведение параллельной программы для современных кластеров с многоядерными узлами.

Среда должна позволять отлаживать параллельные программы с существенно бóльшим объемом данных, чем доступная физическая память.

Среда должна предоставить возможность автоматизированного поиска коммуникационных шаблонов MPI в трассе программы.

Использование целевого кластера для сбора отладочных данных позволяет обеспечить достаточную точность предсказания, а перенос большей части разработки на инструментальный компьютер позволяет повысить продуктивность разработки.

# 3. Модель параллельной многопоточной Java-программы

В данном разделе определена новая интерпретируемая модель параллельной многопоточной Java-программы для вычислительных систем с распределенной памятью. Взаимодействие между процессами осуществляется посредством коммуникационной библиотеки mpiJava.mpi. В настоящее время известно несколько реализаций библиотеки MPI для окружения Java [28, 29, 30, 31]. Коммуникационная библиотека mpiJava.mpi, реализующая MPI, основана на пакете mpiJava [28], который представляет собой привязку языка Java через интерфейс JNI к существующей реализации MPI, например MPICH [32], OpenMPI [33]. В библиотеке реализованы «оберточные» функции для стандарта MPI 1.2 [34]. Внутри процесса могут использоваться Java потоки, для чего была разработана библиотека времени выполнения mpiJava.threads (описание приведено в Приложении 1), основанная на пакете java.util.concurrent [35].

## 3.1 Определение новой модели параллельной многопоточной Java-программы

В данном разделе описывается новая модель параллельной Java-программы. Расширения, введенные в новую модель, призваны решить следующие задачи:

- Уменьшить время интерпретации модели.
- Корректно учитывать специфику современных высокопроизводительных кластеров, содержащих узлы с многоядерными процессорами.

Новая модель была построена на базе существующей модели Java-программы [36], разработанной в ИСП РАН. В разделе приводится краткое описание предыдущей версии модели. Модель строилась по АСД Java-программы. Предыдущая версия модели содержала определение модели класса, функции, внутренних вершин, соответствующих операторам языка Java. Было дано определение базового блока – это семерка B=<id, τ, P, I, O, C, A>, где id –

идентификатор базового блока, τ – тип базового блока, P – последовательность инструкций байт-кода, I – список входных переменных, O – список выходных переменных, C – список управляющих переменных, A – список атрибутов базового блока. Определены базовые блоки следующих типов: вычислительный блок (cb), вызов пользовательской функции (ufc), вызов внешней функции (efc), вызов коммуникационной функции MPI (cfc), редуцированный блок (rb). Было введено понятие редукции вершины. Коммуникационные функции MPI моделировались с помощью следующих восьми базовых операций обмена: **`Init`**, **`Free`**, **`Pack`**, **`Unpack`**, **`Post`**, **`Get`**, **`Copy`**, **`Wait`**.

Однако существующая модель обладала рядом ограничений: не поддерживалась работа с потоками, в реализации существовало ограничение на размер памяти моделируемой параллельной программы, метод оценки времени выполнения базовых блоков не учитывал динамическую компиляцию в Java.

В новой модели ParJava процессы многопроцессно-многопоточной (МПМП) программы моделируются логическими процессами, а Java потоки логическими потоками. Узлы современных кластеров содержат многоядерные процессоры. При распараллеливании программы на узле для каждого ядра можно использовать отдельный Java поток. Поскольку кластер должен быть однородным (необходимо чтобы все узлы были одинаковые), то вводятся следующие ограничения:

- Каждый процесс параллельной МПМП программы использует одинаковое количество потоков.
- Потоки могут находиться в ожидании новых заданий, однако общее количество потоков не меняется (запрещается динамическое порождение потоков).

В предложенную новую модель добавлены атрибуты в основные определения для поддержки моделирования Java потоков, учета влияния JIT компиляции, также добавлены определения приведенной модели базового блока и.т.д.

Для увеличения производительности МПМП программы нужно сделать так, чтобы в потоках основную часть времени занимали вычисления над изолированными данными (приватизация), а обмены между этими потоками устраивались в определенные моменты.

В ParJava процессы МПМП программы моделируются логическими процессами, а Java потоки логическими потоками.

**Замечание 3.1.1**. Количество логических процессов и логических потоков определяется пользователем во время запуска интерпретации модели.

Определения 3.1.1 – 3.1.3, 3.1.7-3.1.9 были взяты из старой модели [36] и приведены здесь для полноты.

***Определение* 3.1.1**. *Моделью параллельной Java-программы* называется множество моделей всех классов этой программы. ■

***Определение* 3.1.2**. *Моделью класса* `C` *параллельной Java*-программы называется пара $\langle C_C, L_C \rangle$, где $C_C$ – модель общих ресурсов класса `C`, $L_C$ – список моделей методов класса `C`. ■

***Определение* 3.1.3**. *Модель общих ресурсов* $C_C$ класса `C` представляет собой список, который включает:

- Описания переменных объекта класса `C`, объявленных в классе (общедоступных, приватных и защищенных переменных объекта класса `C`);
- Описания статических переменных класса `C`;
- Описания общедоступных статических переменных других классов пакета, в который входит класс `C`, используемых в классе `C`;
- Описания общедоступных статических переменных классов других пакетов, которые используются в классе `C`. ■

***Определение* 3.1.4**. *Моделью метода (функции)* `C.f()` класса `C` параллельной Java-программы называется пара $M_{C.f} = \langle Md_{c.f}, Bd_{C.f} \rangle$, где $Md_{c.f}$ дескриптор метода c.f, а $Bd_{C.f}$ – *модель тела метода*. ■

**Определение 3.1.5**. *Дескриптором метода c.f()* называется пара $Md_{c.f} = \langle S_{c.f}, V_{c.f}\rangle$, где $S_{c.f}$ сигнатура метода c.f согласно стандарту [37], а $V_{c.f}$ – список описаний локальных переменных моделируемого метода. ■

***Определение* 3.1.6**. *Модель тела метода* (функции) `c.f()` представляет собой пару ⟨модель потока управления, модель вычислений⟩, где модель потока управления это дерево управления этого метода, а модель вычислений – множество вычислений, производимых в телах базовых блоков данного метода. Внутренние узлы модели тела метода соответствуют операторам языка *Java*, листовые узла – базовым блокам моделируемого метода. ■

Внутренний узел $St_\theta$ модели тела метода, соответствующий оператору *St* типа $\theta$, описывается дескриптором оператора *St*.

***Определение* 3.1.7**. *Моделью оператора St* называется кортеж $St = \langle D_{St}, [B_{CE}], Succ_{St}\rangle$, где $D_{St}$ – дескриптор оператора *St*, $B_{CE}$ – базовый блок, в котором вычисляется значение управляющего выражения оператора *St*, $Succ_{St}$ – список ссылок (*id*) на непосредственные потомки узла $St_\theta$.

***Определение* 3.1.8**. *Управляющим выражением* называется выражение, значение которого определяет ссылку на непосредственный потомок узла *St*, выполняемый после него. ■

***Определение* 3.1.9**. *Дескриптором оператора St* называется кортеж $D_{St} = \langle id, \theta, ref(Succ_{St}), ref(B_{CE}), val(CE), Time, Freq, Targets\rangle$. Элементами кортежа $St_\theta$ являются:

*id* – идентификатор внутреннего узла (экземпляра оператора типа $\theta$), используемый для ссылок на узел $St_\theta$,

$\theta$ – тип узла $St_\theta$; для моделирования программ на языке *Java* необходимо определить следующие типы внутренних узлов модели: *последовательность* ($\theta$ = `{}`), *переключатель* ($\theta$ = `switch`): *ветвление* ($\theta$ = `if`, либо $\theta$ = `if-else`), *цикл* `for` ($\theta$ = `for`), *цикл* `while` ($\theta$ = `while`), *цикл* `do-while` ($\theta$ = `do-while`).

*ref*($Succ_{St}$) – ссылка на список ссылок (*id*) на непосредственные потомки узла $St_\theta$,

*ref*($B_{CE}$) – ссылка на базовый блок $B_{CE}$ (*id* базового блока $B_{CE}$), в котором вычисляется значение управляющего выражения (либо пустая ссылка, если оператор $St_\theta$ не содержит управляющего выражения),

*val*(*CE*) – значение управляющего выражения (номер элемента списка $Succ_{St}$, элементы списка нумеруются, начиная с 0),

*Time* – время выполнения поддерева модели (дерева управления) с корнем *St*.

*Freq* – частота выполнения поддерева модели (дерева управления) с корнем *St*.

*Targets* – список динамических атрибутов, отличных от *Time* и *Freq*, которые необходимо определить в результате интерпретации; эти атрибуты называются *целевыми атрибутами интерпретации*. ■

***Замечание* 3.1.2**. Если интерпретация выполняется только для оценки времени выполнения программы и/или ее фрагментов и/или частоты выполнения фрагментов программы, список *Targets* пуст и может не включаться в дескрипторы. Если список *Targets* не пуст, то во время интерпретации вычисляется оценка значения каждого атрибута из этого списка, а вычисленные оценки присваиваются соответствующим элементам списка *Targets*; эти оценки характеризуют выполнение поддерева с корнем *St*. Большой интерес обычно представляет такой динамический атрибут, как частота выполнения поддерева с корнем *St* ($Freq_{St}$). Значения этого атрибута вместе с оценками времени выполнения позволяют оценить последовательную часть исследуемой программы, а также выявить «горячие» фрагменты программы, то есть фрагменты (методы, циклы, гнезда циклов), выполнение которых занимает большую часть времени выполнения программы. ■

***Замечание* 3.1.3**. В случае, когда *St* – оператор ветвления или цикла значением управляющего выражения является одно из логических значений

`true` или `false`, в случае, когда *St* – переключатель, значением управляющего выражения будет *id* соответствующей альтернативы. ■

***Замечание*** **3.1.4**. В процессе интерпретации модели можно определять значения и других динамических атрибутов модели (например, количества регистров, необходимых для выполнения поддерева с корнем *St*, количества промахов кэша при выполнении поддерева с корнем *St* и т.п.). ■

Значения всех атрибутов узла $St_\theta$, кроме входящих в список целевых атрибутов интерпретации *Targets*, определяются при построении модели. Оценки значений атрибутов из списка *Targets* вычисляются в процессе интерпретации.

Листовой узел модели, соответствующий базовому блоку *B*, описывается парой $B = \langle D, Bd \rangle$, где *D* – дескриптор базового блока, а *Bd* – тело базового блока *B*.

***Определение*** **3.1.10** *Дескриптор базового блока B* представляет собой кортеж $D = \langle id, \tau, ref(Bd), I, O, C, Time, Freq, Targets \rangle$, где:

*id* – идентификатор базового блока *B*, используемый для ссылок на него,

$\tau$ – тип базового блока *B*; в модели определены базовые блоки следующих типов $\tau$: вычислительный блок ($\tau = cb$), вызов пользовательской функции ($\tau = ufc$), вызов внешней функции ($\tau = efc$), вызов коммуникационной функции ($\tau = cfc$) и возврат из текущего метода или функции ($\tau = ret$),

*ref*(*Bd*) – ссылка на тело *Bd* базового блока *B*,

*I* – список определений *входных* переменных блока *B*,

*O* – список определений *выходных* переменных блока *B*,

*C* – список определений *управляющих переменных*, то есть определений переменных, входящих в множество *I* какого-либо базового блока, вычисляющего одно из управляющих выражений, или переменных, участвующих в вычислении фактических параметров коммуникационной функции;

*Time* – время выполнения блока *B*;

*Freq* – частота выполнения блока *B*;

*Targets* – список целевых атрибутов интерпретации. ■

***Замечание* 3.1.5**. Время выполнения вычислительных базовых блоков (блоков типа *cb*) определяется в процессе построения модели, время выполнения базовых блоков остальных типов определяется во время интерпретации. Значения *Freq* и динамических атрибутов, входящих в список *Targets* базового блока любого типа, определяются во время интерпретации. ■

***Замечание 3.1.6***. Каждый элемент списков *I*, *O* и *C* базового блока *B* представляет собой пару ⟨*name*, *id*⟩, где *name* – имя переменной, *id* – идентификатор базового блока, содержащего определение переменной *name*, достигающее базового блока *B*. Если базовый блок достигается несколькими определениями одной и той же переменной *name*, то для объединения этих определений используется новое определение переменной *name* с помощью $\phi$-функции [38, 39]. ■

***Определение* 3.1.11**. *Тело базового блока B* типа *cb* – это линейная последовательность вычислений, выполняемых в блоке *B* (вычисления представляются инструкциями байт-кода, но могут быть представлены и выражениями исходного языка или инструкциями любого другого внутреннего представления ассемблерного уровня). ■

***Определение* 3.1.12**. *Тело базового блока B* одного из типов *ufc*, *efc*, или *cfc* содержит линейную последовательность вычислений фактических параметров вызываемого метода или функции, представленных выражениями. За указанными вычислениями следует сам вызов соответствующего метода или функции (вычисления параметров представляются так же, как и в определении 3.1.9; значения параметров запоминаются в списке $V_{\texttt{C.f}}$ в системных переменных $prm_i$, где *i* – номер формального параметра). ■

С целью сокращения времени интерпретации модели принято решение моделировать только поток управления методов и функций. Для этого модели базовых блоков заменяются их приведенными моделями.

***Определение* 3.1.13**. *Приведенной моделью* базового блока *B* типа *cb* с дескриптором $D = \langle id, cb, ref(Bd), I, O, C, Time, Freq, Targets\rangle$ и телом *Bd* называется блок $B_C = \langle D_C, Bd_C\rangle$, где $D_C = \langle id, cb, ref(Bd_C), I_C, O_C, C, TimeVal, Freq, Targets\rangle$, $I_C = I \cap C$, $O_C = O \cap C$, а $Bd_C$ получается из *Bd* исключением определений переменных, не входящих в список $O_C$, а также мертвого кода, порожденного таким исключением. Атрибут *Time* заменяется на его значение *TimeVal*, которое измеряется при построении приведенной модели (замечание 3.1.5). ■

***Замечание* 3.1.7**. Если для некоторого базового блока *B* типа *cb* $O \cap C = \varnothing$, то тело $Bd_C$ приведенной модели $B_C$ блока *B* оказывается пустым. Это означает, что для оценки значений времени выполнения и других динамических атрибутов процедуры (метода или функции), в состав которой входит базовый блок *B*, достаточно информации, содержащейся в дескрипторе приведенной модели этого блока. Если тело $Bd_C$ приведенной модели блока *B* пусто, его дескриптор имеет вид $D_C = \langle id, cb, nil, I, O, \varnothing, TimeVal, Freq, Targets\rangle$. ■

***Замечание* 3.1.8**. Из определения 3.1.12 следует, что в модели учитывается зависимость выполнения программы от потока управления, но не учитывается зависимость от контекста. Для вычислений, не содержащих вызовы методов (функций), это не существенно, так как оценки времени выполнения базовых блоков согласно замечанию 3.1.5 определяются до интерпретации. Но это существенно для вызовов методов (функций), так как, интерпретируя только приведенные базовые блоки, невозможно адекватно отслеживать изменение контекста. Для учета влияния контекста вызова метода (функции) предлагается в процессе построения модели определять наборы значений фактических параметров каждого вызова, формируя наборы контекстов вызова каждого метода (функции). При этом каждому отдельному контексту присваивается *номер контекста cxt*. ■

***Определение* 3.1.14**. *Приведенной моделью* базового блока $B = \langle id, \tau, ref(Bd), I, O, C, Time, Freq, Targets\rangle$ одного из типов *ufc*, *efc*, или *cfc* для метода *F*

называется блок $B_F = \langle id,\ \tau, cxt, ref(Bd_{F(cxt)}), TimeVal, Freq, Targets\rangle$, где *cxt* – номер контекста вызова метода *F*, а $Bd_{F(cxt)}$ получается из *Bd* заменой вычислений значений фактических параметров на последовательность присваиваний фактическим параметрам значений, соответствующих контексту вызова *cxt*. Значение *TimeVal* определяется во время интерпретации модели. ■

Таким образом, определения, приведенные в данном разделе, позволяют ввести в модель функции, обеспечивающие работу с Java потоками, а также корректно ввести операцию удаления вычислительных инструкций, что позволяет сократить время интерпретации при оценке границы масштабируемости МПМП Java программ.

### 3.2 Моделирование коммуникационных функций

Коммуникационные функции (MPI и для работы с потоками), предоставляемые пользователю в параллельной Java программе, составляют библиотеку времени выполнения, функции из которой работают на целевой вычислительной платформе. В данном разделе описывается, как моделируются функции MPI и функции взаимодействия потоков в среде ParJava.

#### 3.2.1 Моделирование коммуникационных функций MPI

Подробное описание моделирования коммуникационных функций MPI приведено в [40]. В данном разделе приводятся лишь описание процесса моделирования.

Каждый MPI-процесс моделируемой программы представляется в ее модели с помощью логического процесса (класс LProc). Логический процесс определен как последовательность действий (примеры действий: выполнение базового блока, выполнение операции обмена и т.п.). Каждое действие имеет определенную продолжительность. В логическом процессе определено понятие модельных часов. Начальное показание модельных часов каждого логического процесса равно нулю. После интерпретации очередного действия к модельным часам соответствующего логического процесса добавляется значение времени,

затраченного на выполнение этого действия (продолжительности). Продолжительность каждого действия и значения исследуемых динамических параметров базовых блоков измеряются заранее на целевой платформе.

### 3.3.2 Моделирование коммуникационных функций обеспечивающих многопоточное выполнение

Пользовательские потоки моделируются с помощью логического потока (класс ELProc). Каждый объект этого класса обладает локальными модельными часами, время которых хранится в переменной **time**. Пусть основной поток сформировал группу из M заданий. При установлении нового задания, основной поток вместе с параметрами задания передает новому потоку также свое текущее время. Свободный поток из пула потоков, получая новое задание, устанавливает локальное время своих модельных часов, равное времени модельных часов основного потока. При интерпретации модели интерпретатор обновляет переменную **time** потока для каждого фрагмента модели (базовый блок, сбалансированное гнездо циклов и.т.д.), выполненного в рамках данного потока. После выполнения задания завершившийся поток передает основному потоку показания локальных модельных часов и блокируется в ожидании новых заданий. Как только все задания из группы выполнились, интерпретатор пробуждает основной поток, обновляет показание его локальных модельных часов максимальным временем выполненных потоков и продолжает интерпретацию основного потока.

В ParJava семафоры создаются в основном потоке и могут использоваться в порожденных потоках. Семафоры используются для взаимодействия потоков и моделируются следующим образом: каждый семафор обладает служебным полем $\mathbf{sem_{ts}}$ указывающий на момент времени, начиная с которого семафор разблокирован. Начальное значение $\mathbf{sem_{ts}}$ устанавливается равным текущему времени при создании семафора основным потоком. При входе в критическую область порожденные потоки вызывают метод **lock()**, при этом все потоки кроме первого будут заблокированы. Перед блокированием текущее время потока

сохраняется в локальной переменной **sem_start$_i$**. При выходе из критической области поток **th$_i$** вызывает функцию **unlock()**, которая присваивает переменной **sem$_{ts}$** значение, равное показанию модельных часов **th$_i$**, и разблокирует семафор. Пусть следующим пробуждается поток **th$_{i+1}$**. В этом случае время блокировки потока **th$_{i+1}$** определяется разницей (**sem$_{ts}$ - sem_start$_{i+1}$**). Интерпретатор модели обновляет локальное время потока **th$_{i+1}$** и переходит к следующей вершине.

В среде ParJava пакет mpiJava.threads, обеспечивающий коммуникации между потоками, моделируется с помощью следующих базовых примитивов:

pr_CreateThreadPool – создание пула потоков,

pr_SetTask – установка нового задания,

pr_CreateAtomic – создание атомарной переменной

pr_CompareAndSet – атомарное сравнение и присвоение переменной,

pr_CreateSemaphore – создание семафора,

pr_Lock – блокирующий вызов для входа в регион ограниченного доступа,

pr_TryLock – неблокирующий вызов для входа в регион ограниченного доступа,

pr_Unlock – выход из региона ограниченного доступа,

pr_Copy – копирование сообщения.

Ниже описывается, как моделируются основные методы из пакета mpi.threads.

Методам createThreadPool, setTask, createAtomic, compareAndSet, lock, tryLock, unlock, конструктору Semaphore соответствуют примитивы pr_CreateThreadPool, pr_SetTask, pr_CreateAtomic, pr_CompareAndSet, pr_Lock, pr_TryLock, pr_Unlock, pr_CreateSemaphore соответственно.

Методы getAndSet, getAndAdd описываются примитивом pr_CompareAndSet.

Методы CS_Enter, CS_Try_Enter, CS_Exit, конструктор CriticalSection описывают функциональность бинарного семафора. Бинарный семафор можно описать общим семафором (что обеспечивается примитивами pr_Lock, pr_TryLock, pr_Unlock, pr_CreateSemaphore).

Таким образом, функции из библиотеки времени выполнения для работы с

потоками, описанные в разделе 3.1, моделируются в интерпретаторе с помощью вышеописанных примитивов.

### 3.3 Построение модели параллельной *Java*-программы

Модель параллельной *Java*-программы представляет собой список моделей ее классов, а модель каждого класса `c` – список $L_c$ моделей методов этого класса.

Для построения модели метода `c.f()` класса `c` необходимо построить модель $Body_{c.f}$ тела этого метода.

Для построения модели тела $Body_{c.f}$ метода `c.f()` выполняются следующие шаги.

1. Выбираются начальные данные анализируемой параллельной *Java*-программы. В качестве начальных данных следует выбирать «наиболее типичные» значения входных данных программы.
2. С помощью *Java*-компилятора строится байт-код (*JavaBC*) метода `c.f()`.
3. Для построенного байт-кода строится граф потока управления. В результате выделяются базовые блоки метода `c.f()`.
4. Для каждого базового блока *B* метода `c.f()` строится его дескриптор $D = \langle id, \tau, ref(Bd), I, O, C, Time, Targets \rangle$. Состав множества динамических атрибутов *Targets* определяется пользователем, идентификатор *id* блока *B* и его тип $\tau$ вычисляются при построении графа потока управления, тело *Bd* блока *B* (множество инструкций на *JavaBC*) берется из графа потока управления. Для построения множеств *I, O, C* над графом потока управления выполняются различные виды статического анализа потоков данных (в частности, вычисляются достигающие определения, строятся *Du*-цепочки и т.п.).
5. Для каждого базового блока *B* определяется атрибут *Time*. Для этого задаются значения переменных из множества *I*, и блок *B* интерпретируется на *JavaVM*, выполняемой на узле целевого вычислительного комплекса. Полученные в результате интерпретации

значения переменных из множества *O* передаются соответствующим базовым блокам. Базовые блоки (листовые вершины модели тела $Body_{\texttt{c.f}}$ метода `c.f()`) интерпретируются в порядке их вхождения в модель слева направо. Атрибут *Time*, полученный в результате интерпретации, запоминается в соответствующем поле его дескриптора.

6. Для каждого базового блока *B* типа *cb* строится его приведенная модель $B_C$ (см. определение 3.1.13). Для каждого базового блока *B*, имеющего тип *ufc*, *efc*, или *cfc* строится его приведенная модель $B_F$ (см. определение 3.1.14).
7. Выполняется структурный анализ графа потока управления метода `c.f()` и строится его дерево управления.
8. Байт-код параллельной *Java*-программы выполняется в инструментальном режиме на *JavaVM* с отключенным *JIT*-компилятором, используя начальные данные, выбранные на шаге 1.

   Инструментальный режим включает обращения к инструментальным функциям в точках программы, соответствующих входу и выходу каждого базового блока. В процессе выполнения измеряются частота и «статическое» время выполнения каждого базового блока и записываются в дескриптор соответствующего базового блока в качестве значений атрибутов *Time* и *Freq* соответственно.

Следует отметить, что модель каждого метода параллельной *Java*-программы строится не по исходному тексту метода, а по его байт-коду (*JavaBC*). Для построения дерева управления метода сначала строится его граф потока управления, выявляющий базовые блоки, входящие в состав анализируемого метода, а затем выполняется структурный анализ [41] построенного графа потока управления, выявляющий управляющие структуры метода: ветвления и гнезда циклов.

*Язык Java* не содержит явного оператора перехода (`goto`). Поэтому граф потока управления любого метода *Java*-программы может содержать только

ветвления и естественные циклы. Картина может нарушаться операторами передачи управления (`break` и `continue`), но операторы перехода, соответствующие на байт-коде указанным операторам, можно специальным образом пометить при генерации байт-кода, что позволит при построении модели внести в граф потока управления изменения, которые позволят сохранить древовидную структуру модели.

Предлагаемый метод построения модели обеспечивает соответствие модели байт-коду, автоматически учитывая все оптимизации, выполняемые компилятором с языка *Java* на *JavaBC*, включая, в частности, такую глобальную оптимизацию, как вынесение вычислений, инвариантных относительно цикла, в предзаголовок цикла. Все базовые блоки модели, построенной таким образом, соответствуют базовым блокам байт-кода анализируемой *Java*-программы. Таким образом, модель параллельной *Java*-программы полностью соответствует версии этой программы, выполняемой на *JavaVM* в режиме интерпретации.

### 3.3.1 Моделирование «горячих» участков параллельной *Java*-программы

Как известно, в процессе интерпретации *Java*-программы на *JavaVM* выявляются «горячие» участки программы («горячими» называются участки программы, на выполнение которых тратится большая часть времени выполнения). Как правило, в качестве участков программы рассматриваются отдельные методы или большие гнезда циклов. Эти участки с помощью оптимизирующего *JIT*-компилятора заменяются эквивалентными «агрессивно» оптимизированными программами на машинном языке («агрессивной» называется оптимизация, вносящая существенные изменения в структуру оптимизируемой программы). Как показывает практика, «горячие» участки обычно составляют не более 10-15% всех участков программы [42]. Атрибуты *Time* и *Freq* базовых блоков позволяют выявить «горячие» участки анализируемой программы. В параллельных *Java*-программах это обычно гнезда циклов, либо методы, большую часть которых занимают гнезда циклов.

Учет влияния *JIT*-компилятора на выполнение параллельной программы производится, исходя из следующих предпосылок:

1. *JIT*-компилятор оптимизирует только «горячие» участки программы.
2. *JIT*-компилятор существенно изменяет отдельные базовые блоки и весь граф потока управления «горячих» методов и других «горячих» участков программы.
3. *JIT*-компилятор выполняется только один раз во время выполнения программы.

Отметим, что последняя предпосылка выполняется во всех штатных *JIT*-компиляторах, хотя известно несколько экспериментальных *JIT*-компиляторов, в которых перекомпиляция «горячих» участков осуществляется несколько раз (например, [43]).

Поэтому в режиме анализа производительности параллельной *Java*-программы после окончания работы *JIT*-компилятора строятся новые модели ее «горячих» методов и методов, содержащих «горячие» участки. Такая перестройка модели параллельной *Java*-программы является частью ее интерпретации.

Модели «горячих» методов и методов, содержащих «горячие» участки, строятся по бинарному представлению параллельной *Java*-программы. Использование бинарного представления не вызывает сложностей, так как при интерпретации параллельной *Java*-программы используются не инструкции *JavaVM*, а дескрипторы узлов дерева управления интерпретируемой программы.

# 4. Интерпретация модели

В разделе рассматривается разработанный в рамках работы интерпретатор новой модели. Описывается интерпретация модели потока управления и коммуникационных функций (как MPI, так и для работы с Java потоками). Часть интерпретатора пересекается с предыдущей версией, поскольку новая модель является развитием предыдущей модели. Приводится механизм, обеспечивающий возможность интерпретации реальных прикладных параллельных приложений, объем памяти которых превышает доступную физическую память. Описываются методы оценки времени выполнения базовых блоков и более крупных фрагментов, последующая редукция которых позволяет сократить время интерпретации. Также в разделе приводятся оценки погрешности прогнозирования времени выполнения сверху.

*Интерпретация модели параллельной Java-программы* состоит в интерпретации модели метода `main()` одного из классов, входящих в состав программы. Имя этого класса указывается пользователем. При этом пользователь может задать значения параметров метода `main()`. *Интерпретация модели функции (метода)* состоит в интерпретации ее корня.

Цель интерпретации – получение оценок динамических атрибутов, входящих в список *Targets, Time* и *Freq,* для каждой *интерпретируемой единицы* – базового блока, внутреннего узла дерева управления, метода (функции). При этом оценкой времени выполнения всей программы будет оценка времени выполнения метода `main()`.

**Интерпретация внутренних узлов** производится по следующим правилам:

***Последовательность***. Интерпретация нетерминального узла *St* с дескриптором $D_{St} = \langle id, \{\}, ref(Succ_{St}), nil, \varnothing, Time, Freq, Targets\rangle$ состоит в последовательной интерпретации узлов последовательности $Succ_{St}$ и вычислении атрибутов *Time* и *Freq* узла *St* по соответствующим атрибутам узлов последовательности $Succ_{St} = St_1, St_2, \ldots, St_k$ по формулам:

$$Time_{St} = \sum_{i=1}^{k} Time_{St_i}\ ,\ Freq_{St} = Freq_{St_1}\ .$$

Если интерпретация очередного узла последовательности $Succ_{St}$ завершается с состоянием возврата `break` или `continue`, значение *id* которого не совпадает с *id* узла *St*, то последующие узлы последовательности $Succ_{St}$ не интерпретируются, полученное состояние возврата и значение *id* передаются непосредственному предку узла *St*.

Если интерпретация очередного узла последовательности $Succ_{St}$ завершается с состоянием возврата `break`, значение *id* которого совпадает с *id* узла *St*, то последующие узлы последовательности $Succ_{St}$ не интерпретируются и состояние возврата узла *St* выставляется в `ok` (состояние возврата `continue`, значение *id* которого совпадает с *id* узла *St* в данном случае невозможно согласно семантике языка *Java*).

***Переключатель***. Интерпретация нетерминального узла *St* с дескриптором $D_{St} = \langle id, \texttt{switch}, ref(Succ_{St}), ref(B_{CE}), val(CE), Time, Freq, Targets\rangle$, где $Succ_{St} = St_1, St_2, \ldots, St_k$, состоит в интерпретации управляющего выражения *CE* и интерпретации узла $St_i$ из списка $Succ_{St}$, где $i = val(CE)$.

Атрибуты *Time* и *Freq* узла *St* вычисляются по значениям соответствующих атрибутов узла *CE* и интерпретированных узлов из списка $Succ_{St}$ (обозначим их $St_p, St_{p+1}, \ldots, St_q$) по формулам:

$$Time_{St} = \sum_{i=p}^{q} Time_{St_i}\ ,\ Freq_{St} = Freq_{St_p}\ .$$

Если интерпретация узла $St_i$ завершается с состоянием возврата `ok`, то последовательно интерпретируются следующие узлы $St_{i+1}, St_{i+2}, \ldots$ до тех пор, пока состояние возврата не станет равным `break` со значением *id*, совпадающим с идентификатором узла *St*, или не будут интерпретированы все оставшиеся узлы из списка $Succ_{St}$. Состояние возврата узла *St* устанавливается в `ok`.

Если интерпретация узла $St_i$ завершается с состоянием возврата `break` или

`continue`, значение *id* которого не совпадает с идентификатором узла *St*, то узлы $St_{i+1}$, $St_{i+2}$, … не интерпретируются, и интерпретация узла *St* завершается с соответствующим состоянием возврата.

Интерпретация узла $St_i$ с состоянием возврата `continue`, значение *id* которого совпадает с идентификатором узла *St*, приводит к возникновению исключительной ситуации «ошибка в модели», так как такое состояние недопустимо в силу семантики языка *Java*.

***Ветвление `if`***. Интерпретация нетерминального узла *St* с дескриптором $D_{St} = \langle id, \texttt{if}, ref(Succ_{St}), ref(B_{CE}), val(CE), Time, Freq, Targets\rangle$, где $Succ_{St} = St_1$ состоит в интерпретации управляющего выражения *CE*, после чего выполняется: интерпретация узла $St_1$, если $val(CE) = \texttt{True}$.

Атрибуты *Time* и *Freq* узла *St* вычисляется по значениям соответствующих атрибутов узлов $B_{CE}$ и $St_1$ по формулам:

$$Time_{St} = \begin{cases} Time_{B_{CE}} + Time_{St_1}, & \text{если } val(CE) = True \\ Time_{B_{CE}}, & \text{иначе} \end{cases}$$

$$Freq_{St} = Freq_{St} + 1$$

$$Freq_{St_1} = \begin{cases} Freq_{St_1} + 1, & \text{если } val(CE) = True \\ Freq_{St_1}, & \text{иначе} \end{cases}$$

Если интерпретация узла $St_1$ завершилась с состоянием возврата `break` (или `continue`), значение *id* которого не совпадает с идентификатором узла *St*, то это же состояние возврата выставляется для узла *St*.

***Ветвление `if-else`***. Интерпретация нетерминального узла *St* с дескриптором $D_{St} = \langle id, \texttt{if-else}, ref(Succ_{St}), ref(B_{CE}), val(CE), Time, Freq, Targets\rangle$, где $Succ_{St} = St_1, St_2$ состоит в интерпретации управляющего выражения *CE*, после чего выполняется: интерпретация последовательности узлов $\{St_1\}$, если $val(CE) = \texttt{True}$, или интерпретация последовательности узлов $\{St_2\}$, если $val(CE) = \texttt{False}$.

Атрибуты *Time* и *Freq* узла *St* вычисляется по значениям соответствующих атрибутов узлов $B_{CE}$ и $St_1$ по формулам:

$$Time_{St} = \begin{cases} Time_{B_{CE}} + Time_{St_1}, \text{если } val(CE) = True \\ Time_{B_{CE}} + Time_{St_2}, \text{если } val(CE) = False \end{cases}$$

$$Freq_{St} = Freq_{St} + 1$$

$$Freq_{St_1} = \begin{cases} Freq_{St_1} + 1, \text{если } val(CE) = True \\ Freq_{St_1}, \text{иначе} \end{cases}$$

$$Freq_{St_2} = \begin{cases} Freq_{St_2} + 1, \text{если } val(CE) = False \\ Freq_{St_2}, \text{иначе} \end{cases}$$

Если интерпретация узла из списка $St_1$ $(St_2)$ завершилась с состоянием возврата `break` (или `continue`), значение *id* которого не совпадает с идентификатором узла *St*, то это же состояние возврата выставляется для узла *St*.

***Цикл* `while`.** Интерпретация нетерминального узла *St* с дескриптором $D_{St}$ = ⟨*id*, `while`, *ref*($Succ_{St}$), *ref*($B_{CE}$), *val*(*CE*), *Time*, *Freq*, *Targets*⟩, где $Succ_{St} = St_1$ состоит в интерпретации управляющего выражения *CE*, после чего выполняется: интерпретация тела цикла (вершины $St_1$), если *val(CE) = True*, или интерпретация оператора, следующего за циклом (вершины $St_2$), если *val(CE) = False*.

Атрибуты *Time* и *Freq* узла *St* вычисляется по значениям соответствующих атрибутов узлов $B_{CE}$ и $St_1$ по формулам:

$$Time_{St} = \begin{cases} Time_{B_{CE}} + Time_{St_1}, \text{если } val(CE) = True \\ Time_{B_{CE}}, \text{если } val(CE) = False \end{cases}$$

$$Freq_{St} = Freq_{St} + 1$$

$$Freq_{St_1} = \begin{cases} Freq_{St_1} + 1, \text{если } val(CE) = True \\ Freq_{St_1}, \text{иначе} \end{cases}$$

Если интерпретация узла $St_1$ завершилась с состоянием возврата `break` (или `continue`), значение *id* которого не совпадает с идентификатором узла *St*, то это же состояние возврата выставляется для узла *St*. Если интерпретация узла $St_1$ завершилась с состоянием возврата `continue`, значение *id* которого совпадает с идентификатором узла *St*, продолжается интерпретация узла *St*. Если интерпретация узла $St_1$ завершилась с состоянием возврата `break`, значение *id* которого совпадает с идентификатором узла *St*, интерпретация узла *St* прекращается.

***Цикл do-while***. Интерпретация нетерминального узла *St* с дескриптором $D_{St}$ = ⟨*id*, `do-while`, *ref*($Succ_{St}$), *ref*($B_{CE}$), *val*(*CE*), *Time*, *Freq*, *Targets*⟩, где $Succ_{St}$ = $St_1$ состоит в интерпретации узла тела цикла (вершины $St_1$), последующей интерпретации управляющего выражения *CE*, после чего выполняется: интерпретация узла $St_1$, если *val*(*CE*) = `True`, или интерпретация оператора, следующего за циклом (вершины $St_2$), если *val(CE) = False*.

Атрибуты *Time* и *Freq* узла *St* вычисляется по значениям соответствующих атрибутов узлов $B_{CE}$, $St_1$ по формулам аналогичным соответствующим формулам для цикла `while`.

***Цикл for***. Интерпретация узла *St* = ⟨*id*, `for`, ({*init*, *body; update*}, *next*), *ref*($B_{CE}$), *val*(*CE*), *Time*, *Freq*, *Targets*⟩ производится следующим образом: {*init;* ⟨*id*, `while`, ({*body; update*}, *next*), *ref*($B_{CE}$), *val*(*CE*), *Time*, *Freq*, *Targets*⟩}.

Атрибут *Time* узла *St* вычисляется по значениям соответствующих атрибутов узлов *init*, *E*, а также *body* и *update*.

**Интерпретация базовых блоков.**

При построении модели метода (функции) модели базовых блоков заменяются их приведенными моделями (см. определения 3.1.13 и 3.1.14).

***Вычислительный блок***. Интерпретация базового блока типа «*вычислительный блок*» $B_{cb}$ = ⟨*id*, *cb*, *ref*(*Bd*), *I*, *O*, *C*, *Time*, *Freq*, *Targets*⟩ состоит в выполнении подмножества инструкций из списка *ref*(*Bd*), достаточного для определения его атрибутов из списка *Targets*, и значений переменных из списка *C*.

Указанное подмножество инструкций *ref*(*Bd*)′ ⊆ *ref*(*Bd*) определяется в результате статического анализа программы во время построения модели.

Атрибут *Time* вычислительного блока всегда определяется заранее (во время предварительного анализа программы), так что во время интерпретации его значение известно. Остальные атрибуты узла $B_{cb}$ либо определяются во время предварительного анализа программы, либо вычисляются во время интерпретации узла $B_{cb}$.

***Вызов пользовательского метода или функции***. Интерпретация базового блока типа «*вызов пользовательского метода или функции*» `c.f()` $B_{umc}$ = ⟨*id, umc, ref*(*Bd*), *I, O, C, Time, Freq, Targets*⟩ состоит из следующих этапов:

1. Интерпретируются выражения из списка *ref*(*Bd*), определяющие значения фактических параметров метода `c.f()`,

2. Интерпретируется корневой узла модели метода `c.f` для вычисленных значений фактических параметров,

3. Значения атрибутов корневого узла модели метода `c.f` (включая атрибут *Time*) присваиваются соответствующим атрибутам блока $B_{umc}$.

***Вызов библиотечного метода или функции.*** Интерпретация базового блока типа «*вызов внешней (библиотечной) функции*» `c.f()` зависит от доступности этой функции (библиотеки). Если байт-код класса, содержащего соответствующую функцию (метод), доступен (может быть загружен) во время интерпретации, то интерпретируются выражения из списка *ref*(*Bd*), определяющие значения фактических параметров, и выполняется соответствующая функция (метод) для вычисленных значений фактических параметров. Определение атрибутов такого узла производится по тем же правилам, что и для базового блока типа «*редуцированный блок*». Если байт-код соответствующего класса недоступен, то интерпретация аналогична интерпретации базового блока типа «редуцированный блок».

***Вызов коммуникационной функции.*** Интерпретация базового блока типа «*вызов коммуникационной функции*» заключается в определении атрибутов базового блока согласно выбранной модели коммуникаций. Подробности интерпретации коммуникационных функций для оценки времени выполнения программы рассматриваются в разделе 4.1.

Как уже отмечалось, редуцированный блок представляет поддерево модели программы, значения атрибутов блока во время интерпретации уже известны (они могли быть получены во время предварительного анализа программы или сообщены пользователем). Поэтому интерпретация редуцированного блока не

производится.

***Замечание.*** Из определения интерпретации видно, что интерпретация циклов может занимать много времени. Ниже будет показано, что во многих случаях это время можно сократить без ущерба для точности интерпретации.

Несмотря на то, что при интерпретации приведенной модели параллельной программы выполняется меньший объем вычислений, чем при выполнении самой программы, интерпретация может занимать значительное время. Время интерпретации увеличивается и в связи с тем, что интерпретация выполняется на инструментальной машине, производительность которой меньше, чем у целевой системы. В процессе интерпретации параллельной программы можно выбрать модель одного из операторов исследуемой программы, указав соответствующую вершину в модели метода, в состав которого входит указанный оператор, проверить с помощью анализатора модели, удовлетворяет ли он условиям редуцируемости, и если да, то редуцировать модель этого оператора. Редукция модели оператора заключается в том, что поддерево модели с корнем в рассматриваемом операторе заменяется одной вершиной типа редуцированный базовый блок. В дальнейших интерпретациях модели эта часть программы интерпретироваться не будет, а значения динамических параметров, необходимые для интерпретации модели, будут содержаться в редуцированном блоке, на который была заменена соответствующая часть программы во время редукции. Для удобства пользователей системы реализован режим интерпретации, при котором условия редуцируемости проверяются в процессе интерпретации каждой вершины модели. Если оказывается, что для какой-либо вершины модели эти условия выполняются, указанная вершина помечается как редуцируемая.

Операция редукции позволяет интерпретировать программу по частям, заменяя интерпретированные фрагменты редуцированными блоками. Следовательно, указав значения входных переменных из списка $I$ внутреннего узла $St$, допускающего корректную редукцию, можно выполнить его интерпретацию отдельно от остальной программы. В этом случае полученные

значения атрибутов из списка *Targets* присваиваются соответствующим атрибутам редуцированного блока, сформированного в результате редукции узла *St* и его потомков. В результате редукции количество узлов модели сокращается, что позволяет сократить длительность текущего сеанса интерпретации, если редукция выполнялась внутри гнезда цикла, а также сократить длительность последующих сеансов интерпретации.

## 4.1 Интерпретация коммуникационных функций при оценке времени работы программы

### 4.1.1 Интерпретация коммуникационных функций MPI при оценке времени работы программы

Как уже отмечалось, коммуникационные функции MPI можно моделировать с помощью следующих восьми базовых операций обмена: **`Init`**, **`Free`**, **`Pack`**, **`Unpack`**, **`Post`**, **`Get`**, **`Copy`**, **`Wait`**. Время выполнения этих примитивов оценивается с помощью таблицы зависимости между объемом передаваемых данных и временем передачи данных, получаемой для используемой коммуникационной сети на тесте «пинг-понг». Таблица зависимости определяет характеристики коммуникационной сети и строится при ее установке.

### 4.1.2 Оценка времени выполнения коммуникационных функций обеспечивающих многопоточное выполнение

Все методы из пакета mpiJava.threads моделируются базовыми примитивами, приведенными в разделе 3.3.2. Следовательно, для оценки времени выполнения методов взаимодействия между потоками достаточно оценить время выполнения базовых примитивов.

Время выполнения базовых примитивов, моделирующих операции между потоками, определяется следующим образом:

Примитив **pr_CreateThreadPool(n)** – время создания одного потока обозначается как $\mathbf{time}_{\mathbf{tCreate}}$, количество потоков в пуле – n. В таком случае время выполнения данного примитива определяется как

Time(pr_CreateThreadPool(n)) = n*$\mathbf{time_{tCreate}}$

Примитив **pr_SetTask(task, isBlocking)** – пусть $\mathbf{time_{tStart}}$ время запуска потока, а $\mathbf{time_{tRun}}$ время выполнения потока с заданием **task**. В этом случае

Time(pr_SetTask(task, isBlocking)) = $\mathbf{time_{tStart}}$ + + (isBlocking ? $\mathbf{time_{tRun}}$: 0)

где **isBlocking** - признак блокирования, величина переменной $\mathbf{time_{tRun}}$ определяется во время интерпретации и равна разности показаний модельных часов пользовательского и основного потока.

Примитив **pr_Lock(sem,n)** – **sem** это семафор, а **n** количество запрашиваемых слотов (для бинарного семафора это n - единица). Пусть $\mathbf{time_{lock}}$ время блокировки потока и переменная $\mathbf{sem_{ts}}$ представляет собой временную метку семафора. Когда поток освобождает регион с ограниченным доступом (примитив pr_Unlock), переменной семафора $\mathbf{sem_{ts}}$ присваивается показание модельных часов этого потока. В этом случае $\mathbf{time_{lock}}$ определяется следующим образом

$\mathbf{time_{lock}}$ = (**time** < $\mathbf{sem_{ts}}$) ? ($\mathbf{sem_{ts}}$ - **time**) : 0

В этом случае

Time(pr_Lock(sem,n)) = Time(pr_TryLock(sem,n)) + (pr_TryLock(sem,n)>0 ? $\mathbf{time_{lock}}$: 0)

Примитив **pr_Copy(n)** – пусть $\mathbf{time_{copy}}$ время выполнения операции копирования одного байта на вычислительном узле, **n** – количество копируемых байтов. В этом случае

Time(pr_Copy()) = n $\mathbf{time_{copy}}$

Время работы примитивов **pr_CreateAtomic(atomic,initValue)**, **pr_CompareAndSet(atomic,expect,update), pr_CreateSemaphore(initN, mode), pr_TryLock(sem,n), pr_Unlock(sem,n)** константно и определяется посредством следующих параметров: $\mathbf{time_{aCreate}}$ время создания атомарной переменной, $\mathbf{time_{aCAS}}$ время атомарного сравнения и присвоения переменной, $\mathbf{time_{sCreate}}$ время создания семафора, $\mathbf{time_{tryLock}}$ время, за которое поток предпринял попытку блокирования семафора, $\mathbf{time_{release}}$ время разблокировки семафора.

Параметры $\mathbf{time_{tCreate}}$, $\mathbf{time_{tStart}}$, $\mathbf{time_{aCreate}}$, $\mathbf{time_{aCAS}}$, $\mathbf{time_{sCreate}}$, $\mathbf{time_{release}}$,

$\mathbf{time_{tryLock}}$, $\mathbf{time_{copy}}$ определяются с помощью тестов на узле вычислительной платформы. В ParJava разработаны тестовые программы на уровне библиотечных функций Sun JDK1.6 с учетом особенностей библиотеки mpi.threads, определяющие соответствующие параметры.

***Замечание***. Значение параметров $\mathbf{time_{tStart}}$, $\mathbf{time_{sCreate}}$, $\mathbf{time_{release}}$, $\mathbf{time_{aCreate}}$, $\mathbf{time_{aCAS}}$ сильно меньше чем значение остальных параметров и вклад этих параметров в определении времени выполнения коммуникаций незначителен, поэтому в дальнейшем будем считать их равными нулю.

Время выполнения функций из групп создания и управления потоками, методов атомарных операций, методов для работы с критическими секциями элементарно выражаются через времена базовых примитивов приведенных выше.

## 4.2 Моделирование программ с большим объемом данных

В данном разделе приводится описание механизма (приведение модели), обеспечивающего моделирование программ с существенно бóльшим объемом данных, чем доступная физическая память на инструментальном компьютере.

В рассматриваемом классе задач (плотные матрицы) производится много вычислений на больших матрицах. Обычно инструментальная машина по ресурсам (например, оперативная память) уступает вычислительной платформе и для обеспечения моделирования программы на ней модель программы преобразуется в приведенную модель (в терминах определения 3.1.13). С точки зрения прогнозирования времени выполнения, результаты, которые считаются в прикладной программе, не являются существенными.

Приведение модели состоит в удалении из модели всех инструкций и данных, которые не влияют на поток управления программы. Для этого вначале строятся дерево доминаторов, DU-UD цепочки, SSA представление программы, проводится анализ живых переменных, и помечаются управляющие переменные (выражения). После этого удаляются все вычислительные инструкции. При интерпретации приведенной модели, интерпретатор, встретив вычислительный базовый блок, который не содержит инструкций, учитывает время выполнения и

переходит к следующему событию.

**Определение.** Управляющая переменная это переменная, значение которой может влиять на поток управления.

**Определение**. Переменная, которая используется в управляющей конструкции (оператор ветвления, цикла), называется непосредственной управляющей переменной.

Например, в следующем фрагменте *a* это непосредственная управляющая переменная, а *b* нет.

```
a=b;
if(a){BasicBlock;}
```

**Определение**. Управляющей инструкцией назовем инструкцию, в которой определяются управляющие переменные или используются непосредственные управляющие переменные.

**Определение**. Формальный параметр функции называется управляющим параметром, если он является управляющей переменной в рамках данной функции.

Приведение или чистка модели включает в себя три этапа:

1. Нахождение управляющих инструкций. Для этого необходимо сначала построить дерево доминаторов [44, 45] и DU-UD цепочки, SSA представление [46]. После чего необходимо построить два множества: 1)множество непосредственных управляющих переменных, 2)множество управляющих переменных. Для построения этого множества был использован итеративный алгоритм поиска управляющих переменных на основе поиска живых переменных.
2. Замена не управляющих фактических параметров в вызовах методов на константу по умолчанию для соответствующего типа параметра. В модели программы, в каждом вызове пользовательского метода все параметры, которые не являются управляющими, заменяются константными выражениями.
3. Удаление инструкций и связанных с ними данных из модели.

**Нахождение управляющих инструкций.** Глобальный алгоритм поиска управляющих переменных получает на входе для каждой инструкции языка программы S множество переменных, которые определяются в инструкции S и множество переменных, которые используются в ней. На выходе для каждой S вычисляются множества переменных, которые являются управляющими перед началом инструкции, после нее, множество глобальных управляющих переменных.

Введем следующие обозначения In[S] – множество локальных переменных, которые являются управляющими перед началом инструкции, Out[S] – множество локальных переменных, которые являются управляющими после выхода из инструкции, defS – множество переменных, которые определяются в инструкции, useS – множество переменных, которые используются в инструкции, GCVariables – множество, содержащее глобальные управляющие переменные, ControlParameters – множество, содержащее управляющие параметры, C – множество непосредственных управляющих переменных, используемых в инструкции.

После того как глобальный алгоритм поиска завершил свою работу, инструкция S будет управляющей, если выполнено одно из условий: $In[S] \cap Out[S] \neq \varnothing$ либо $defS \cap GCVariables \neq \varnothing$

Алгоритм является итеративным [47], то есть продолжает выполнять итерации, пока в ходе очередной итерации не будет внесено ни одного изменения в множество управляющих переменных. На каждой итерации обходятся все методы программы и для каждого из них 1) если возвращаемое значение метода является управляющим, то все переменные, определяющие это значение, помечаются как управляющие, 2) вызывается локальный алгоритм поиска управляющих переменных для этого метода.

**Вход**: Множество графов потока управления, где для каждого графа для каждой инструкции S вычислены множества defS и useS.

**Выход**: множества переменных, которые являются управляющими на входе, выходе каждой инструкции S базового блока B графа потока, множество

глобальных управляющих переменных (GCVariables) .

| Область определения | Множество переменных |
|---|---|
| Направление | Обратное |
| Граничное условие | Out[Выход] = ∅ |
| Оператор сбора Λ | ∪ |
| Уравнения | $In[S] = f_S(Out[S])$<br>$Out[S] = \Lambda_{P \in Succ[S]} In[P]$ |
| Инициализация | In[S] = C |

Псевдокод алгоритма представлен на Рисунке 1.

```
boolean gchange=false;
Init();
while(gchange)
{
  gchange = false;
  for(каждый граф потока управления)
  {
   if(метод M, содержится в ControlReturnMethods)
   {
    for(каждое return-выражение ret из M)
    for(каждая переменная var из ret)
    {
     gchange = true;
     Пометить var как управляющий.
    }
   }
   LocalSearch(M);
}
```

Рисунок 1. Псевдокод алгоритма поиска управляющих переменных.

На Рисунке 1 gchange это флаг, содержащий состояние изменения всех методов, ControlReturnMethods это множество методов, возвращаемые значения которых

определяют управляющие переменные.

В функции Init() (Рисунок 2) производится инициализация необходимых данных.

```
for(каждый граф потока управления)
{
In[Выход] = ∅;
for (каждая инструкция, отличная от выходной) In[S] = C
}
```

Рисунок 2. Инициализация данных.

Псевдокод функции LocalSearch(M), приведенный на Рисунке 3, представляет собой локальный итеративный алгоритм поиска управляющих переменных в рамках метода M.

```
boolean change=false;
while(change)
for(каждая инструкция, отличная от выходной)
{
Out[S] = Λ_{P∈Succ[S]} In[P]
In[S] = transferS(Out[S])
}
```

Рисунок 3. Локальный поиск.

Здесь change – флаг, содержащий состояние изменения текущего метода.

Передаточная функция transfers(Out[S]) задает соотношение между значениями потока данных до и после инструкции. Так как здесь рассматривается алгоритм обратного потока данных, то эта функция для инструкции S вычисляет множество In[S] по Out[S]. Помимо этого функция строит множество GCVariables. Для этого проверяется, существует ли в инструкции определение управляющей переменной (локальной, глобальной или являющейся фактическим параметром, при условии, что формальный параметр, соответствующий ей, находится в множестве ControlParams). В этом случае все переменные, использованные в этой инструкции, в зависимости от их типа добавляются в соответствующие

множества (In[S], GCVariables, ControlParams). Также, если в инструкции есть вызов метода, и этот метод возвращает значение, определяющее управляющую переменную, то такой метод добавляется в ControlReturnMethods.

**Замена не управляющих фактических параметров и удаление инструкций.** Производится обход модели, и в каждом вызове пользовательского метода каждая переменная (при условии, что формальный параметр, соответствующий ей, не содержится во множестве ControlParams) заменяется константой, соответствующей типу переменной.

**Удаление инструкций и связанные с ними данных из модели**. На последнем этапе из модели удаляются все инструкции, которые не являются управляющими или вызовами пользовательских методов. Пример удаления инструкций приводится на Рисунке 4.

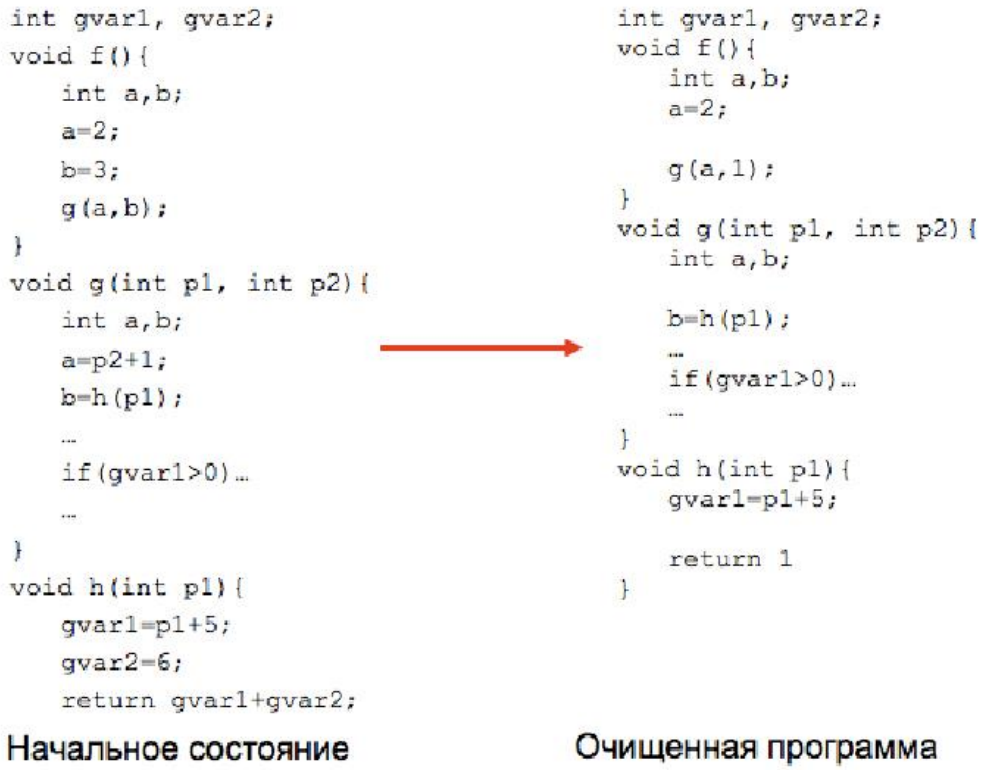


Рисунок 4. Удаление инструкций.

Приведение модели позволяет достичь следующих результатов:

1)появляется возможность интерпретировать реальные прикладные приложения с большим объемом данных (поскольку из модели удалены большие

матрицы),

2)увеличивается скорость интерпретации модели за счет сокращения количества интерпретируемых инструкций.

### 4.3 Оценка времени выполнения модели Java-программы

Время выполнения параллельной Java-программы определяется во время интерпретации модели. Интерпретатор использует значения целевых атрибутов Time терминальных узлов модели и вычисляет время выполнения для всех узлов модели. Терминальными узлами модели являются либо базовые блоки, либо более крупные фрагменты. Как показали исследования, оценка времени выполнения более крупных фрагментов программы приводит к ускорению интерпретации модели и повышению точности прогноза за счет учета влияния JIT-компилятора (динамический компилятор проводит агрессивную оптимизацию более крупных фрагментов программы, являющихся горячими участками).

#### 4.3.1 Оценка времени выполнения базовых блоков программы

Пользователь имеет возможность сообщить оценку времени выполнения вычислительного базового блока, удовлетворяющего необходимым условиям редукции, или вызова библиотечной функции. Для этого ему необходимо заменить соответствующий узел редуцированным блоком и присвоить требуемое значение времени его атрибуту *Time*.

Обычно при временном профилировании оценка времени выполнения участка кода (базового блока) получается как разность показаний системных часов в начале и конце этого участка кода. Начиная с версии 1.5, в стандартной библиотеке Java реализован метод **`System.nanoTime()`**, который возвращает время, пройденное с определенного момента в прошлом в наносекундах. Однако на большинстве платформ показания системных часов выражаются в микросекундах. Поскольку на современных компьютерах время работы базового блока обычно меньше микросекунды, временное профилирование программы требует применения специальной методики.

Время выполнения базовых блоков определяется с помощью аппаратного счётчика TSC (Time Stamp Counter). Обращение к счетчику производится через ассемблерную вставку с использованием интерфейса JNI (в настоящее время такие ассемблерные вставки реализованы для процессоров *Intel* 32, *Intel* 64 и *PowerPC*).

Для получения временного профиля программы необходимо выяснить время работы каждого базового блока. Для получения времени работы вычислительного базового блока перед началом и после базового блока вставляются инструментальные вставки, которые измеряют время выполнения и регистрируют показания часов в служебной структуре. Время выполнения базового блока измеряется несколько раз, после этого в модели базового блока сохраняется усредненное значение.

### 4.3.2 Измерение времени работы фрагментов MPI-программы на целевой вычислительной системе для ускорения интерпретации ее модели

При построении временного профиля моделируемой программы не следует ограничиться оценкой времени выполнения базовых блоков, интерпретация может быть существенно ускорена, если оценивать и время выполнения более крупных фрагментов: отдельных итераций циклов, циклов в целом, а в некоторых случаях – и гнезд циклов.

Временной профиль строится на целевой вычислительной системе и уточняется в процессе интерпретации программы. С помощью операции редукции, введенной в [2], модель преобразуется к форме, обеспечивающей ее быструю интерпретацию. Методам разбиения модели программы на достаточно крупные фрагменты, для которых возможно получить и использовать достаточно точные оценки их времени выполнения, посвящен данный раздел.

#### 4.3.2.1 Вычисление оценки времени выполнения методов

Модель каждого метода *Java*-программы рекурсивно строится из моделей его фрагментов – базовых блоков и их комбинаций (последовательностей, ветвлений, переключателей и циклов).

***Замечание***. Здесь рассматриваются фрагменты, которые в литературе обычно называются областями (регионами), то есть подграфы графа потока управления метода, имеющие единственный входной базовый блок, доминирующий над всеми остальными блоками фрагмента.

Измерив значение времени выполнения каждого базового блока, можно в процессе интерпретации модели вычислять оценки времени выполнения различных комбинаций базовых блоков и более крупных фрагментов, получая в конце интерпретации метода оценку времени его выполнения. При этом время выполнения последовательности фрагментов всегда равно сумме времен выполнения составляющих фрагментов. Но если в состав фрагмента входит ветвление, то время выполнения фрагмента зависит от времени выполнения текущей ветви, то есть определяется значениями параметров метода. Если время выполнения фрагмента не зависит от параметров метода (например, фрагмент не содержит ветвлений, или время выполнения каждой ветви примерно одинаково), будем называть такой фрагмент *простым*. В терминах раздела 3.1 каждый простой фрагмент может быть редуцирован.

Если тело цикла является простым фрагментом, то и сам цикл является простым фрагментом и может быть редуцирован. Мы будем называть такие циклы *простыми*. Подобные соображения применимы и к соответствующим гнездам циклов, которые тоже будут называться *простыми*. Если можно доказать, что гнездо циклов является простым, то можно при определении временного профиля метода измерить время выполнения такого гнезда и заменить его редуцированным блоком с соответствующим временем выполнения.

Следует отметить, что гнездо циклов, содержащее операции обмена данными через *MPI*, не может быть простым, хотя внутренние циклы, не содержащие операций обмена, могут быть простыми и допускать редукцию.

Для параллельных *MPI*-программ, написанных на одном из традиционных языков (*C*/*C++, Fortran*), модель должна строиться не над исходным кодом программы, а над ее промежуточным представлением с учетом результатов

статической оптимизации, так как оптимизаторы современных компиляторов могут существенно изменить структуру программы, следовательно и ее модель.

Для параллельных *Java*-программ, использующих *MPI*, структура программы, выполняемой на *Java VM*, практически не отличается от структуры исходной программы: статическая оптимизация, как правило, производится только внутри базовых блоков и не влияет на структуру программы. В процессе выполнения *Java*-программы на *Java VM* определяются наиболее часто выполняемые фрагменты программы (*«горячие» методы*) и для них выполняется динамическая компиляция с агрессивной оптимизацией (разработчики динамических компиляторов считают, что такие фрагменты составляют менее 20% всей программы [48]). Таким образом, большая часть методов программы продолжает интерпретироваться на *Java VM*, причем такие методы обычно удовлетворяют условиям редукции и при интерпретации модели вносят некоторый постоянный вклад во время выполнения соответствующих фрагментов (время интерпретации таких методов в среде *ParJava* сравнимо со временем интерпретации одного базового блока).

Что касается «горячих» методов, то для ускорения интерпретации их моделей с сохранением точности оценок времени выполнения необходимо уметь выделять как можно более крупные фрагменты, в пределах которых обычно сосредоточена большая часть структурных изменений кода, связанных с оптимизацией, и при профилировании измерять время выполнения таких фрагментов в целом, по возможности игнорируя их структуру. В параллельных программах такими крупными фрагментами являются гнезда циклов.

#### 4.3.2.2 Измерение времени выполнения сбалансированных гнезд циклов

Гнездо циклов **for** может быть представлено в виде дерева, корнем которого является самый внешний цикл **for**, а поддеревьями являются циклы **for** глубины вложенности 1, далее по рекурсии. Если у одного из циклов гнезда нет вложенных в него циклов, то соответствующий ему узел дерева не имеет поддеревьев и является терминальным в дереве, представляющем гнездо циклов.

В *MPI*-программах, как правило, распараллеливается один из внешних циклов гнезда (часто это бывает самый внешний цикл). В дальнейшем будут рассматриваться только распараллеливаемые подгнезда, где распараллеливается самый внешний цикл.

Интерпретация цикл **for**, количество повторений которого равно *n,* на интерпретаторе среды *ParJava* (либо на *JavaVM*) позволит измерить:

(1) время выполнения цикла $T_{Loop}$;

(2) время выполнения тела цикла на *i*-ой итерации $T^{\mathrm{i}}_{\mathrm{LoopBody}}$.

Сравнение $T^{\mathrm{i}}_{\mathrm{LoopBody}}$ со средним временем выполнения тела цикла $T^{\mathrm{mean}}_{\mathrm{LoopBody}} = \frac{T_{\mathrm{Loop}}}{\mathrm{n}}$ позволит определить степень сбалансированности цикла $D = \max_i \left| T^{\mathrm{i}}_{\mathrm{LoopBody}} - T^{mean}_{\mathrm{LoopBody}} \right|$. Будем говорить, что рассматриваемый цикл *сбалансирован*, если его степень сбалансированности *D* не превышает $\alpha \cdot T^{\mathrm{mean}}_{\mathrm{LoopBody}}$ (поскольку допустимая погрешность оценки времени выполнения фрагментов программы порядка 10% [49], можно принять $\alpha$ = 0,1). В противном случае цикл будет считаться *разбалансированным*.

Поскольку оптимизирующие преобразования цикла (такие, как вынесение из цикла вычислений, инвариантных относительно цикла, или минимизация числа индуктивных переменных) не влияют на его степень сбалансированности *D*, измерение *D* можно выполнять для исходного кода программы при его интерпретации на интерпретаторе среды *ParJava*, либо на *JavaVM*.

Гнездо циклов будем называть сбалансированным, если каждый цикл, входящий в его состав, сбалансирован. Анализ сбалансированного гнезда циклов выполняется, начиная с самых внутренних («листовых») циклов. Для каждого такого цикла измеряется время его выполнения, после этого указанный цикл редуцируется. Поэтому на каждом этапе анализа рассматривается цикл, у которого нет вложенных циклов (после редукции каждый вложенный цикл заменяется «эквивалентным» редуцированным базовым блоком). Если сбалансированы как сам цикл, так и все подциклы, входящие в его тело, то при

снятии временного профиля получается оценка времени выполнения всего цикла, а оценка времени выполнения отдельной итерации цикла определяется делением оценки времени выполнения цикла на число его итераций.

В случае сбалансированного цикла, число итераций которого определяется во время выполнения программы (например, в объемлющем цикле), нужно получить оценку времени выполнения цикла для достаточно большого числа итераций, потом получить оценку времени выполнения одной итерации цикла, причем последняя будет использоваться для вычисления оценки времени выполнения рассматриваемого цикла на каждой итерации объемлющего цикла. Аналогичный метод применим и к вычислению оценок времени выполнения циклов, распределяемых среди узлов вычислительной системы для параллельного выполнения. Результаты численных расчетов (см. раздел 7) подтвердили правомерность описанного подхода к оценке времени выполнения сбалансированных циклов.

#### 4.3.2.3 Измерение степени разбалансированности гнезда циклов и времени выполнения разбалансированных гнезд циклов в среде ParJava

Рассмотрим цикл вида:

```
for(i = 0; i ≤ n; i++) {
    {fr1};
    if(c(i)) {fr2};
    else {fr3};
    {fr4}
}
```

Интерпретация фрагментов **{fr1}**, **{fr2}**, **{fr3}** и **{fr4}** и условия **c(i)** на интерпретаторе среды *ParJava* (либо на *JavaVM*) позволяет получить предварительные оценки времени их выполнения: $T_{fr1}$, $T_{fr2}$, $T_{fr3}$, $T_{fr4}$, $T_{c(i)}$. При этом оценка времени выполнения тела цикла может быть вычислена по формуле $T_{LoopBody} = T_{fr1} + T_{c(i)} + T_{Cond} + T_{fr4}$, где $T_{Cond}$ определяется следующим образом: на

интерпретаторе оцениваются частоты выполнения ветвей условного оператора $F_{fr2}$ и $F_{fr3}$, после чего $T_{Cond}$ определяется по формуле

$$T_{Cond} = \frac{F_{fr2} \cdot T_{fr2} + F_{fr3} \cdot T_{fr3}}{F_{fr2} + F_{fr3}}$$

Если фрагменты **`{fr1}`**, **`{fr2}`**, **`{fr3}`** и **`{fr4}`** содержат ветвления, оценки времена их выполнения должны быть получены заранее с помощью описанного метода.

Для определения степени разбалансированности цикла (или гнезда циклов) интерпретатор модели запускается не менее чем в $P$ потоках ($P$ – достаточно большое число, например $P \geq 16$), и время интерпретации гнезда циклов измеряется в каждом потоке. Пусть $T_{Cond}^{p}$ – оценка $T_{Cond}$ в $p$-ом потоке ($1 \leq p \leq P$). Тогда степень разбалансированности вычисляется по формуле:

$$D' = \max_{p}(T_{Cond}^{p}) - \min_{p}(T_{Cond}^{p})$$

Рассмотренный метод применим для оценки разбалансированности циклов общего вида.

Ввиду того, что при вычислении времени выполнения цикла и его отдельных итераций измерение времени производится лишь в начале и конце итерации, оценка получается правильной, несмотря на то, что структура выполняемого цикла после динамической компиляции может быть оптимизирована, в результате чего она может существенно отличаться от структуры исходного цикла. Может измениться даже число итераций цикла из-за его частичной раскрутки. Интерактивный сценарий устранения обнаруженной разбалансированности гнезда циклов состоит в распределении по потокам групп итераций цикла таким образом, чтобы времена выполнения указанных групп итераций стали близки (различались на достаточно малую величину). В случае недостаточно низкой (с точки зрения пользователя) степени разбалансированности (это выясняется с помощью интерпретатора) пользователь вручную меняет распределение итераций цикла и снова с помощью интерпретатора выясняет степень разбалансированности, после чего либо

процесс заканчивается, либо проверяется новое распределение. Если цикл разбалансирован, оценка времени его выполнения получается как сумма оценок времен выполнения всех его итераций. При этом время измеряется только в начале и в конце итерации: интерпретации итераций не требуется, так что оптимизация не влияет на оценку.

#### 4.3.2.4 Оценка времени выполнения циклов, требующих синхронизации

В предыдущих разделах рассматривались циклы, целиком выполняемые на одном из узлов кластера, т.е. циклы, которые во время своего выполнения не требуют периодических обменов данными между узлами кластера (и связанной с такими обменами синхронизации вычислений на разных узлах). В данном разделе рассматриваются оценки, необходимые для правильного (оптимального) размещения вызовов функций библиотеки *MPI* в теле цикла, выполняемого параллельно на нескольких узлах кластера. Указанные вызовы должны быть размещены таким образом, чтобы

(1) передача данных велась параллельно с расчетами,

(2) передача данных заканчивалась прежде, чем потребуется начать их обработку [50].

Таким образом, оптимизация (ее обычно называют настройкой параллельного цикла) состоит в нахождении такого взаимного расположения фрагментов тела цикла, при котором выполняются условия (1) и (2), и для ее решения необходимо знать оценки времени выполнения каждого из рассматриваемых фрагментов тела цикла.

Следует отметить, что методы, содержащие параллельные циклы, входят в число наиболее часто выполняемых методов программы и, следовательно, оптимизируются во время динамической компиляции. Значит для оптимизации (настройки) такого метода необходимо уточнить его модель, отразив в ней оптимизирующие преобразования, выполненные при его динамической компиляции. Основная трудность в том, что не все динамические компиляторы языка *Java* (в частности, [51]) предоставляют доступ к коду оптимизированной

программы. Тем не менее, один из наиболее современных динамических компиляторов [48], разработанный в IBM, обеспечивает такой доступ, а тем самым и принципиальную возможность уточнения модели рассматриваемого метода после его динамической компиляции. Анализируя оптимизированный код, можно построить его граф потока управления, а затем и дерево управления, лежащее в основе модели параллельной *Java*-программы среды *ParJava*. Используя эту возможность, в среде *ParJava* строится уточненная модель тела параллельного цикла, позволяющая оценить время выполнения каждого его фрагмента и тем самым выбрать оптимальные точки расположения вызовов функций обмена данными. Такой подход является трудоемким, но он позволяет оценить задержки, вызванные асинхронностью выполнения частей циклов на узлах кластера, и добиться их минимизации.

Для сбалансированного цикла, различные итерации которого требуют примерно одинаковых объемов вычислений, планы вычислений на различных узлах кластера одинаковы. Время $t$, требуемое на обмен данными между такими узлами, складывается из времени $t_{dt}$, необходимого на передачу требуемого объема данных через канал связи, и *времени синхронизации* $t_s$, которое идет на компенсацию асинхронности работы узлов кластера (и ядер указанных узлов). Зная параметры коммуникационной сети кластера, можно достаточно точно определить $t_{dt}$. Для времени $t_s$ возможны лишь вероятностные оценки, так как в это время входит ожидание в случае, когда к моменту обращения к данным оказывается, что обмен еще не завершен. Исследование детального профиля сбалансированного цикла позволяет настроить цикл, обеспечив приемлемые значения ускорения и масштабируемости.

В случае разбалансированного цикла, время вычисления различных итераций которого может отличаться на большие значения, настройка программы существенно усложняется. Но и в этом случае детальный профиль тела цикла может помочь найти методом проб приемлемое распределение данных по узлам кластера.

## 4.4 Оценка погрешности времени выполнения

В отличие от предыдущей версии модели, где время выполнения базовых блоков оценивалось вне контекста и использовались отдельные оберточные циклы для каждого базового блока, в текущей версии модели время выполнения базовых блоков оценивается при отладочном запуске оригинальной программы (сохраняется контекст) как в режиме Java интерпретатора, так и при включенном JIT.

Пусть получена оценка среднего времени работы базового блока $\bar{t}_{bb}$ согласно разделу 4.3.1. Пусть время выполнения базового блока измеряется N раз на узле вычислительной платформы. Пусть при каждом измерении $t_{bb}$ отличается от измеренного машинного времени $\tau_{bb}$ на случайную величину ε, представляющую собой гаусовский белый шум. То есть ε имеет стандартное нормальное распределение с нулевым математическим ожиданием и стандартным отклонением σ. При i-ом измерении:

$$t_{bb}^{i} = \tau_{bb}^{i} + \varepsilon_{bb}^{i} \ , \ \varepsilon_{bb}^{i} \sim \text{iid}(0,\sigma) \qquad (1)$$

$\varepsilon^{i}$ независимы и одинаково распределены. В этом случае среднее время выполнения базового блока определяется как

$$\bar{t}_{bb} = \frac{1}{N}\sum_{i=1}^{N}\tau_{bb}^{i} + \frac{1}{N}\sum_{i=1}^{N}\varepsilon_{bb}^{i} = \bar{\tau}_{bb} + \frac{1}{N}\sum_{i=1}^{N}\varepsilon_{bb}^{i} \qquad (2)$$

**Теорема 1**. Абсолютная погрешность $\Delta_{bb}$ оценки среднего времени работы базового блока $\bar{t}_{bb}$ удовлетворяет неравенству

$$|\Delta_{bb}| \le \frac{3}{\sqrt{N}}|\sigma| \qquad (3)$$

**Доказательство**. Из формулы (2) следует, что

$$|\Delta_{bb}| = |\bar{t}_{bb} - \bar{\tau}_{bb}| = \frac{1}{N}|\sum_{i=1}^{N}\varepsilon_{bb}^{i}| \qquad (4)$$

Согласно центральной предельной теореме случайная величина $\eta_N = \sqrt{N}(\bar{X} - \mu)$, где $\bar{X}$ выборочное среднее первых n величин, сходится по

распределению к нормальному $N(0,\sigma^2)$. Согласно правилу трех сигм, 99,7% случайных величин $\eta_N$ попадет в интервал $[-3\sigma, 3\sigma]$ (мат. ожидание равно нулю). Следовательно, для большинства случаев

$$|\eta_N| \le |3\sigma|$$

$$|\eta_N| = \sqrt{N}\,|\overline{X} - \mu| = \sqrt{N}\,\left|\frac{1}{N}\sum_{i=1}^{N}\varepsilon_{bb}^{i}\right| \le |3\sigma|$$

Подставляя полученный результат в формулу (4), получим

$$|\Delta_{bb}| = \frac{1}{N}\left|\sum_{i=1}^{N}\varepsilon_{bb}^{i}\right| \le \frac{|3\sigma|}{\sqrt{N}} = \frac{3}{\sqrt{N}}|\sigma|. \blacksquare$$

**Теорема 2**. Относительная погрешность оценки среднего времени работы базового блока $\bar{t}_{bb}$ удовлетворяет неравенству

$$\delta_{bb} \le \frac{3|\sigma|}{\sqrt{N}(\bar{\tau}_{bb} - |\sigma|)} \qquad (5)$$

**Доказательство**. Относительная погрешность среднего времени базового блока bb определяется как $\delta_{bb} = \frac{|\Delta_{bb}|}{\bar{t}_{bb}}$. Из теоремы 1 следует, что

$$\delta_{bb} \le \frac{\frac{3|\sigma|}{\sqrt{N}}}{\bar{t}_{bb}} = \frac{3|\sigma|}{\sqrt{N}(\bar{\tau}_{bb} + \frac{1}{N}\sum_{i=1}^{N}\varepsilon_{bb}^{i})} \qquad (6)$$

Поскольку $\varepsilon_{bb}^{i}$ ~iid(0,σ), то, применив правило трех сигм и подставив минимальное значение для $\varepsilon_{bb}^{i}$=M($\varepsilon_{bb}^{i}$)-|σ|=-|σ| в формулу (6), получим

$$\delta_{bb} \le \frac{3|\sigma|}{\sqrt{N}(\bar{\tau}_{bb} + (-|\sigma|))} = \frac{3|\sigma|}{\sqrt{N}(\bar{\tau}_{bb} - |\sigma|)}. \blacksquare$$

Поскольку программа представляет собой суперпозицию терминальных вершин, то абсолютная погрешность оценки времени работы программы

$$\Delta_P \le C_P \max_{bb \in P}(|\Delta_{bb}|) \qquad (7)$$

где $\Delta_{bb}$ – абсолютная погрешность времени работы базового блока, $C_P$ – константна для фиксированной программы P.

Когда оцениваются не базовые блоки, а более крупные фрагменты, в формуле (7) $\Delta_{bb}$ можно заменить на $\Delta_{fr}$, которая представляет собой либо $\Delta_{bb}$ (если оценивался базовый блок), либо $\Delta_{red}$ (при оценке крупного фрагмента и его последующей редукции). Такая замена правомерна, поскольку время выполнения крупного фрагмента оценивается тем же механизмом что и время базовых блоков.

Таким образом, предложенная методика интерпретации параллельной программы позволяет точно предсказать время ее работы.

# 5. Автоматизированное обнаружение коммуникационных шаблонов MPI

Существует множество инструментов профилирования и визуализаторов трасс, в результате работы которых создаются таблицы и графики с различными статистиками. Такие инструменты анализа производительности оказывают помощь разработчикам параллельных приложений в оптимизации параллельного кода. Однако в большинстве случаев предлагаемые инструменты требуют ручной обработки выходных данных - разработчик вручную анализирует предоставленные статистические данные и графики в поисках узких мест и возможностей повышения производительности программы. Количество обрабатываемой вручную информации увеличивается существенно с количеством ядер, процессов и объемом данных параллельной программы. Назовем коммуникационным шаблоном MPI наличие в трассе параллельной программы событий (например, send, recv), связанных определенными соотношениями (например, допустимых величин разности временных меток событий). В настоящее время известно несколько шаблонов, наличие которых свидетельствует о возможной потере производительности в параллельной программе. Автоматизация поиска коммуникационных шаблонов MPI позволило бы повысить продуктивность разработки. В разделе описывается метод автоматизированного обнаружения коммуникационных шаблонов MPI приводящих к потере производительности параллельного приложения, а также приводится описание типов выявляемых шаблонов: десять типов для блокирующих коммуникационных функций и семь типов для не-блокирующих функций [7].

## 5.1 Описание метода выявления коммуникационных MPI шаблонов

Метод автоматизированного обнаружения коммуникационных шаблонов, приводящих к потере производительности в параллельных MPI-программах для

вычислительных систем с распределенной памятью, базируется на анализе данных, полученных во время исполнения параллельной программы в режиме сбора информации, так называемый post-mortem анализе [27]. Для автоматизированного обнаружения коммуникационных шаблонов необходимо вначале получить информацию времени выполнения о критических конструкциях-функциях, которые потенциально могут привести к потере производительности. После этого проводится анализ собранной информации на предмет наличия шаблонов.

Рассмотрим характерные особенности шаблонов в параллельной программе, которые могут привести к потере производительности. Пусть на входе имеется параллельная SPMD программа для вычислительных систем с распределенной памятью. В какой-то момент работы программы в процессе $pid_i$ потребуются данные процесса $pid_j$ . Если такой необходимости нет, то данная параллельная программа в каждом процессе производит только локальные вычисления, и она хорошо масштабируема. Однако в реальных прикладных задачах такое поведение не встречается. Таким образом, процессы выполняют определенную работу, после чего по какой-то схеме передают данные друг другу и снова приступают к локальным вычислениям. Этот цикл повторяется до тех пор, пока в конце очередной итерации не будет достигнута определенная точность, либо не выполнится некий предикат. Потери производительности могут возникнуть во время обмена данными между процессами. Происходит это из-за асинхронного поведения разных процессов параллельного приложения.

Метод автоматизированного обнаружения коммуникационных MPI шаблонов в параллельных MPI-программах состоит из следующих этапов (Рисунок 5):

Этап 1. Сбор данных времени выполнения параллельной MPI-программы.

Этап 2. Анализ данных, полученных на Этапе 1, и выявление шаблонов в параллельной MPI-программе.

Этап 3. Создание отчета о выявленных шаблонах с привязкой к исходному коду параллельной программы.

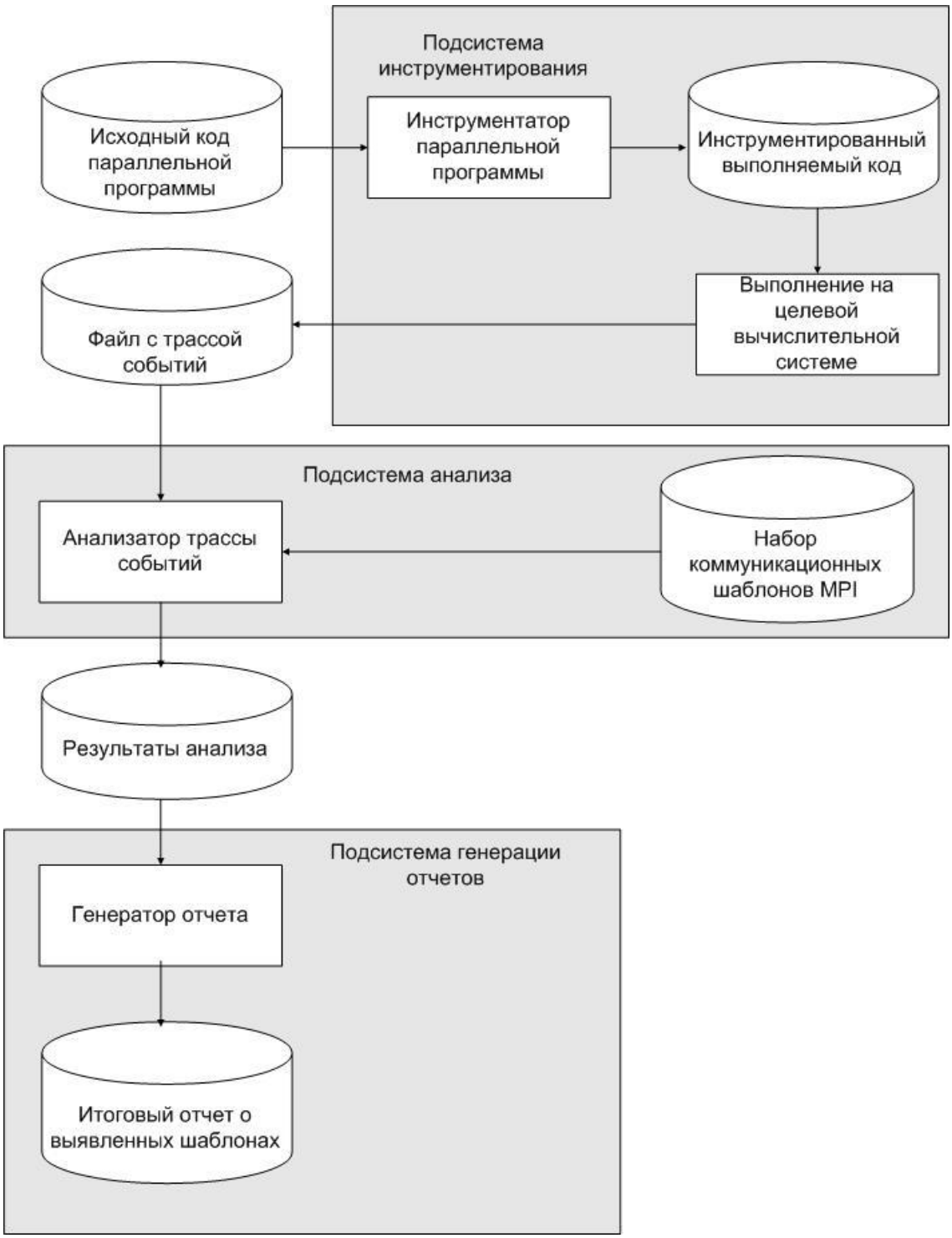


Рисунок 5. Схема метода автоматизированного выявления коммуникационных шаблонов в параллельной программе.

Построение трассы параллельного приложения состоит из следующих этапов:

а) Инструментирование программы. Инструментирование программы представляет собой добавление в определенных позициях оригинальной программы вызовы к инструментальной библиотеке. Во время выполнения

программы эти вызовы регистрируют наступление определенного события и производят запись в трассу.

б) Выполнение инструментированной программы на целевой платформе. Инструментированная программа переносится на целевую платформу и производится запуск параллельной программы. В результате для каждого процесса программы создается его трасса.

Инструментирование параллельного приложения можно проводить на уровне исходного кода приложения, на уровне компоновки, динамически. Проанализировав описанные подходы, в данной работе было выбрано инструментирование на уровне компоновки.

При использовании подхода инструментирования на уровне компоновки необходимо, чтобы либо компилятор, либо библиотека, которая является «узким местом» с точки зрения производительности, поддерживала интерфейс профилирования. Например, для библиотеки MPI разработчики поддерживают стандартный интерфейс PMPI [52]. Согласно стандарту для всех функций интерфейса MPI реализуются две версии: с префиксом MPI_ и PMPI_ (например, MPI_Ssend и PMPI_Ssend). Инструменты, предназначенные для сбора трассы, реализуют оболочки для функций (например, все коммуникационные точка-точка функции MPI), данные времени выполнения которых необходимо собрать. Рисунок 6 представляет пример функции-оболочки для стандартной функции MPI_Ssend(…). Инструкции в прологе и эпилоге производят необходимые измерения и запись в трассу. А между ними вызывается оригинальная функция синхронной посылки данных.

При запуске параллельного приложения профилирующая библиотека компонуется с бинарным кодом параллельного приложения и помимо вызовов к функциям MPI производятся также и вызовы к профилирующей библиотеке, тем самым обеспечивается измерение производительности и сбор трассы. Естественно в такой профилирующей библиотеке-оболочке можно реализовать не все функции стандарта MPI, а лишь те, которые разработчики библиотеки считают критичными при исследовании производительности параллельной

программы. Более того, реализуемые функции-оболочки можно разбить на группы и, с помощью опций при инструментировании, включить профилирование той или иной группы, тем самым концентрируя внимание на критических на данный момент функциях и уменьшая размер собираемой трассы.

```
MPI_Ssend(…)
{
  // исходный код пролога
  PMPI_Ssend(…);
    // исходный код эпилога
}
```

Рисунок 6. Функция-оболочка для стандартной функции MPI_Ssend(…).

На втором этапе, после получения трассы событий применяется post-mortem анализ - трасса параллельной программы анализируется для выявления предопределенных шаблонов. Каждому шаблону соответствует определенный критерий (предикат от временной метки события, временной метки соответствующего парного события и.т.д.). Для выявления шаблонов перебираются события из трассы и, при выполнении определенного критерия, регистрируется наличие соответствующего шаблона.

На третьем этапе собранные данные передаются генератору отчетов, который создает итоговый отчет в удобном формате. Итоговый отчет о выявленных шаблонах содержит список описателей шаблонов. Каждый элемент в списке представляет собой кортеж <type, node, process/thread, file, line, comment,…>, где type – тип шаблона, comment – диагностическое текстовое сообщение о шаблоне. node, process/thread, file, line определяют адрес узла, идентификатор процесса/потока, имя файла, номер строки в файле, где проявляется данный шаблон.

## 5.2 Применение метода выявления коммуникационных шаблонов MPI на модельной трассе в среде ParJava

С целью повышения продуктивности разработки параллельных приложений

в ParJava большая часть разработки переносится на инструментальный компьютер. Описанный в разделе 5.1 метод был адаптирован для выявления шаблонов в MPI программах, написанных на языке Java. Гибкость модели ParJava позволила применить данный подход на инструментальном компьютере. Трасса параллельной программы собирается на инструментальном компьютере, при минимальном использовании дорогостоящей целевой вычислительной платформы. Для этого на инструментальном компьютере производится построение модели параллельной программы. Модель обогащается данными, полученными при отладочном запуске инструментированной программы на целевой платформе. Получение трассы на инструментальном компьютере обеспечивается системой, в которой используется модель параллельной программы, основанная на симуляции дискретных событий, и применяется метод прямого выполнения [13].

На Рисунке 7 представлен метод автоматизированного выявления шаблонов в MPI-программах, написанных на языке Java, с использованием трассы, полученной посредством моделирования. Метод состоит из следующих этапов:

Этап 1. Сбор модельной трассы параллельной MPI-программы.

Этап 2. Анализ данных, полученных на Этапе 1, и выявление шаблонов в параллельной MPI-программе.

Этап 3. Создание отчета о выявленных шаблонах с привязкой к исходному коду параллельной программы.

На первом этапе используется моделирование MPI-программы для получения модельной трассы дискретных событий. Интерпретатор модели обходит вершины модели параллельной программы, вычисляет динамические атрибуты (время выполнения) и для каждой терминальной вершины модели, которая может привести к образованию шаблонов (в частности это вершины, соответствующие коммуникационным функциям MPI), записывает событие в трассу.

Естественно, трассу можно собрать не для всех функций стандарта MPI, а лишь для тех, которые разработчики считают критичными при исследовании

производительности параллельной программы. Для таких функций в интерпретаторе модели создаются функции-оболочки.

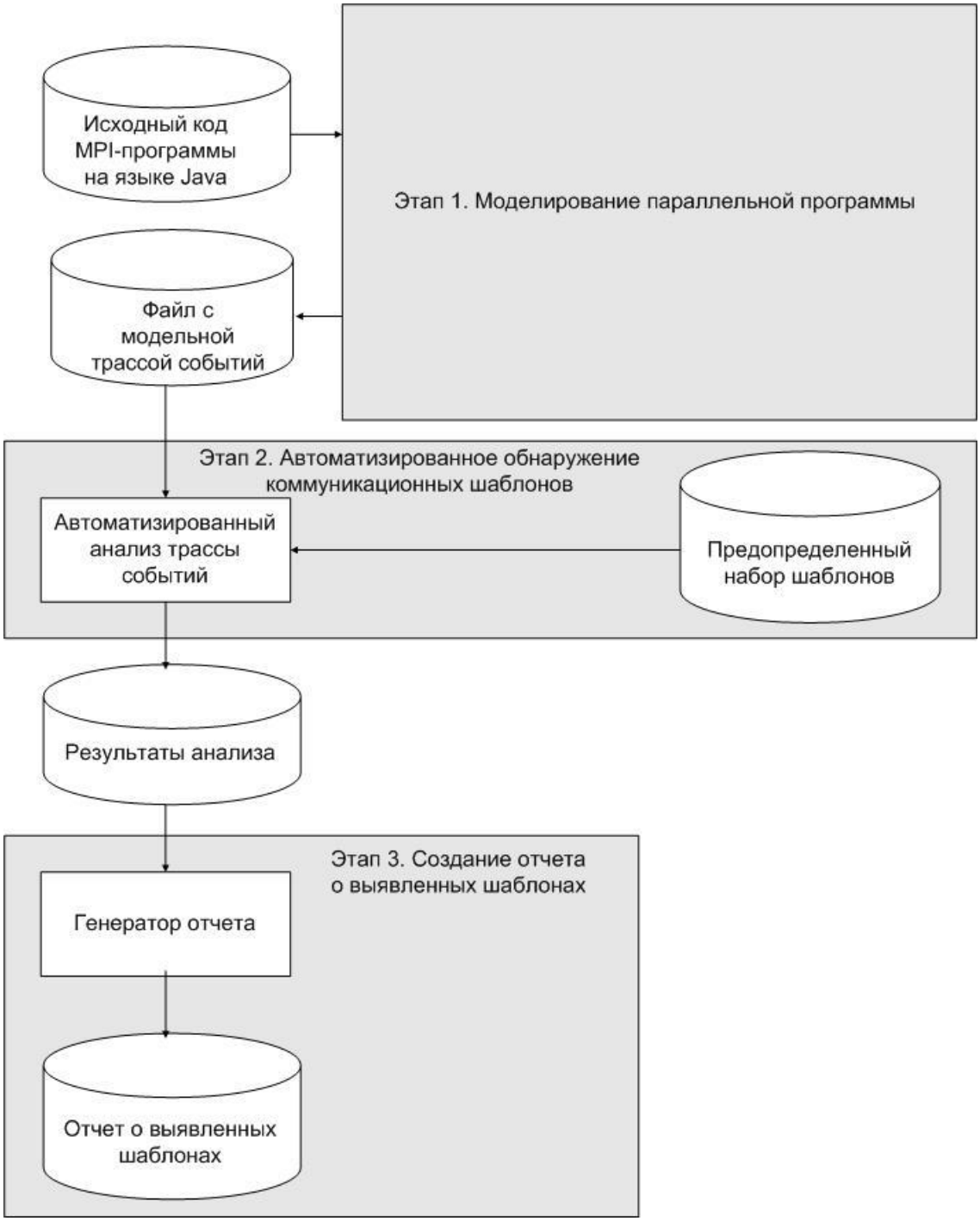


Рисунок 7. Метод автоматизированного выявления шаблонов посредством моделирования.

Реализуемые функции-оболочки можно разбить на группы и с помощью опций во время построения модели задать необходимость профилирования той или иной группы, тем самым концентрируя внимание на критических на данный момент функциях и уменьшая размер собираемой трассы.

Степень детализации трассы имеет большое значение. Чем больше степень детализации, тем больше информации о выполнении параллельной программы можно собрать, что в свою очередь позволяет анализировать различные аспекты производительности параллельного приложения. С другой стороны при большей детализации увеличивается размер трассы событий, что приводит к проблеме хранения трассы, а также к увеличению времени анализа трассы.

Данные времени выполнения параллельной программы представляют собой трассу определенных событий параллельного приложения. Событие представляет собой кортеж <T,TS,…>, где T - тип события, TS – временная метка начала события. А также в зависимости от типа события в кортеж могут быть включены дополнительные поля (например, для типа “send” кортеж содержит также S – размер сообщения, D – идентификатор процесса получателя, Tag – таг отправляемого сообщения).

Собранная модельная трасса передается post-mortem анализатору, который проводит поиск шаблонов.

### 5.3 Описание типов выявляемых шаблонов

В разделе рассматриваются варианты шаблонов с применением коммуникаций точка-точка в параллельных MPI-программах.

При пересылке сообщений от одного процесса к другому возникают моменты простоя (idle) либо на одной, либо на другой стороне. Данный эффект на корректность вычислений не влияет, однако он отрицательно отражается на скорости выполнения программы. Устранение этих простоев приведет к увеличению производительности параллельной программы.

Обозначим через Time_start(F) временную метку непосредственно перед началом выполнения функции F, где F представляет собой функцию MPI.

Обозначим через Time_end(F) временную метку непосредственно после выполнения функции F, где F представляет собой функцию MPI.

Пусть ε пороговое значение (на данный момент получено экспериментальным путем).

Обозначим через I_T(pid, $p_i$, $c_j$) время простоя процесса с идентификатором pid в результате коммуникации $c_j$={sendId,recvId} вызванное обнаруженным шаблоном $p_i$. sendId и recvId представляют собой внутренние идентификаторы посылки и приема соответственно.

**Замечание**. Если время простоя для коммуникации $c_j$ меньше порогового значения ε, то будем считать, что при коммуникации $c_j$ шаблон $p_i$ не проявился ($I\_T(pid,p_i,c_j)=0$).

### 5.3.1 Шаблоны, связанные с запаздыванием посылки/приема при использовании блокирующих точка-точка коммуникаций

**Ранняя стандартная посылка (E_MPI_Send→MPI_Recv)**. Рассмотрим случай, когда посылка начинается раньше, чем прием (Рисунок 8). В этом случае процесс-отправитель теряет время (idle).

Критерий шаблона:

$$I\_T(pid,p_i,c_j)=\begin{cases}0, & \text{если } (Time\_start(MPI\_Recv)-Time\_start(MPI\_Send))<\varepsilon\\ Time\_start(MPI\_Recv)-Time\_start(MPI\_Send), & \text{иначе}\end{cases} > 0$$

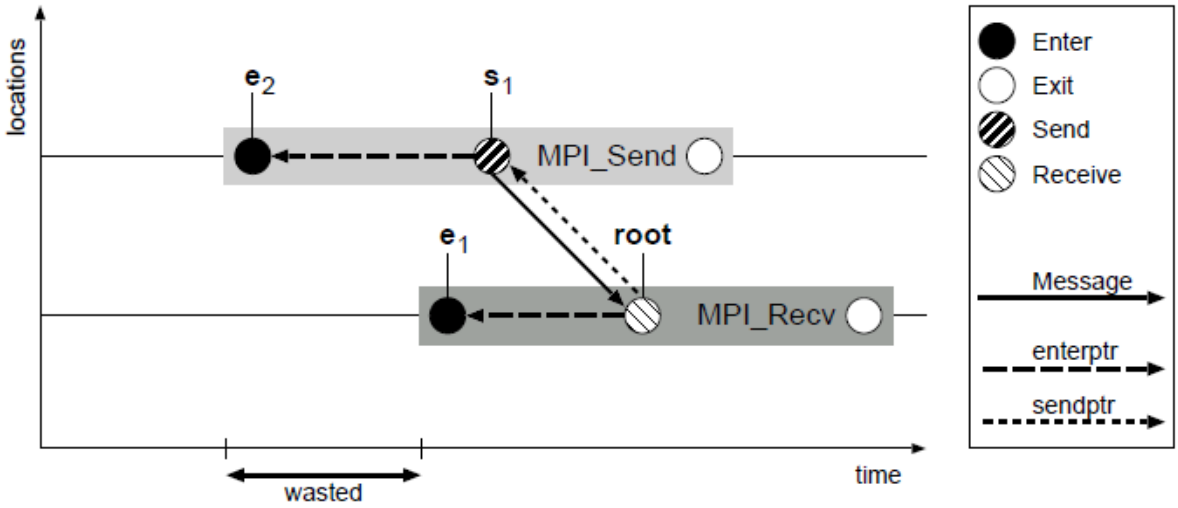


Рисунок 8. Поздний прием с передачей MPI_Send.

**Замечание**. Здесь следует учесть особенности, связанные с реализацией MPI (также об этом упоминается в стандарте MPI). Если реализация библиотеки MPI использует протокол опережающей посылки (например, MPICH, MPICH2, MVAPICH [53], MVAPICH2,…), то функция MPI_Send, пересылающая сообщение, размер которого меньше, чем определенная константа, будет

локальной (не блокирующей). И, следовательно, простоя в этом случае наблюдаться не будет и необходимо исключить этот случай из поиска.

**Поздняя стандартная посылка (L_MPI_Send→MPI_Recv).** Рассмотрим случай, когда получение сообщения начинается раньше, чем посылка (Рисунок 9.). В этом случае простаивает процесс-получатель.

Критерий шаблона:

$$I\_T(pid, p_i, c_j) = \begin{cases} 0, \text{если } (Time\_start(MPI\_Send) - Time\_start(MPI\_Recv)) < \varepsilon \\ Time\_start(MPI\_Send) - Time\_start(MPI\_Recv), \quad \text{иначе} \end{cases} > 0$$

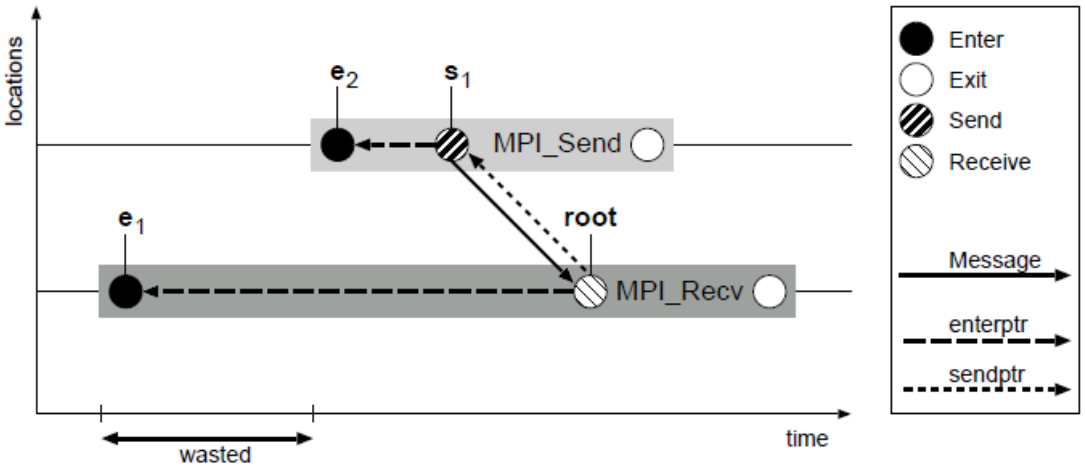


Рисунок 9. Ранний прием с передачей MPI_Send.

**Поздняя буферизированная посылка (L_MPI_Bsend→MPI_Recv).** Функция MPI_Bsend является локальной функцией – отправитель (процесс 0) копирует сообщение в буфер и возвращает управление, а система времени выполнения MPI занимается отправкой сообщения из буфера. Прием сообщения начинается раньше, чем соответствующая посылка MPI_Bsend (Рисунок 10). В этом случае принимающая сторона (процесс 1) простаивает в ожидании.

Критерий шаблона:

$$I\_T(pid, p_i, c_j) = \begin{cases} 0, \text{если } (Time\_start(MPI\_Bsend) - Time\_start(MPI\_Recv)) < \varepsilon \\ Time\_start(MPI\_Bsend) - Time\_start(MPI\_Recv), \quad \text{иначе} \end{cases} > 0$$

**Ранняя буферизированная посылка (E_MPI_Bsend→MPI_Recv).** Посылка сообщения начинается раньше, чем соответствующий прием. Однако в данном случае процесс 0 не простаивает, потому что функция MPI_Bsend является локальной – функция копирует сообщение в буфер и возвращает управление, а система времени выполнения MPI занимается отправкой сообщения из буфера.

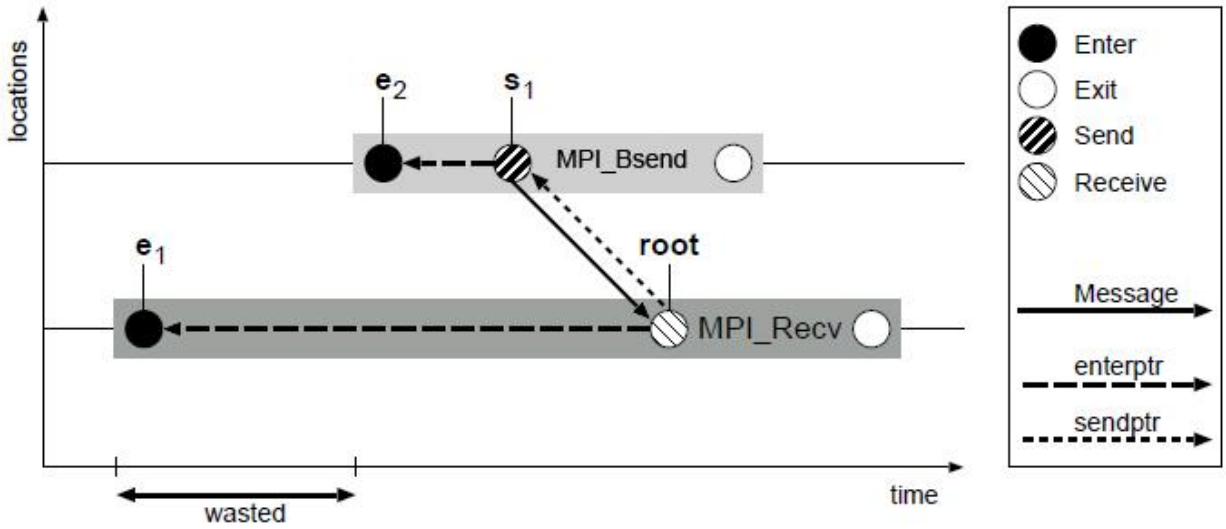


Рисунок 10. Ранний прием с передачей MPI_Bsend.

**Поздняя синхронная посылка (L_MPI_Ssend→MPI_Recv).** Прием сообщения начинается раньше, чем соответствующая посылка MPI_Ssend (Рисунок 11). В этом случае принимающая сторона (процесс 1) простаивает в ожидании.

Критерий шаблона:

$$I\_T(pid, p_i, c_j) = \begin{cases} 0, & \text{если } (Time\_start(MPI\_Ssend) - Time\_start(MPI\_Recv)) < \varepsilon \\ Time\_start(MPI\_Ssend) - Time\_start(MPI\_Recv), & \text{иначе} \end{cases} > 0$$

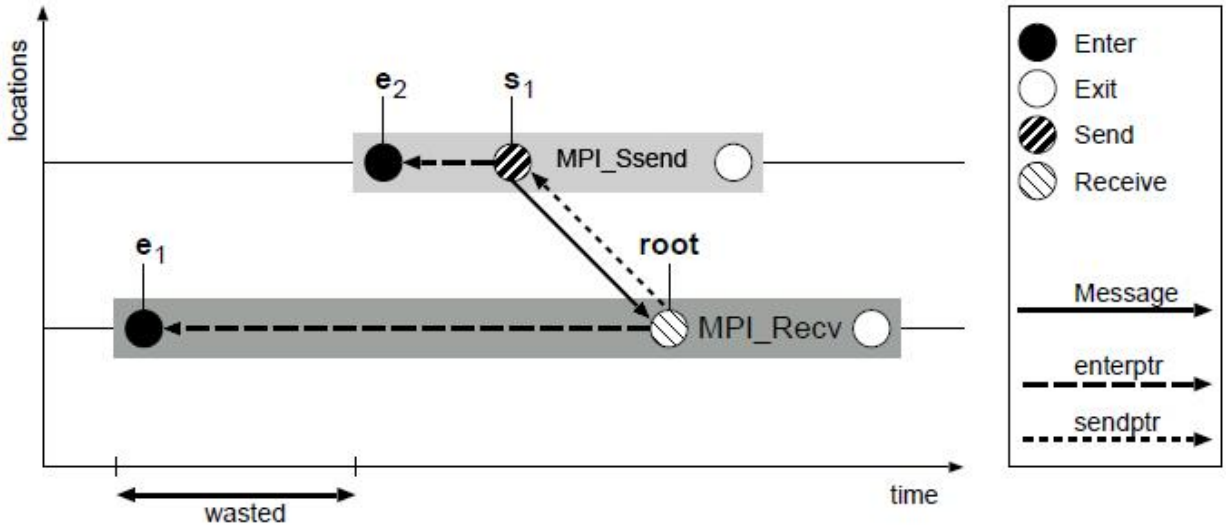


Рисунок 11. Ранний прием с передачей MPI_Ssend.

**Ранняя синхронная посылка (E_MPI_Ssend→MPI_Recv)**. Посылка сообщения начинается раньше, чем соответствующий прием. Синхронная посылка использует протокол рандеву – отправляющий ждет, пока сообщение будет принято на стороне принимающего и после чего возвращает управление.

То есть из-за того что MPI_Recv начинается позже, процесс-отправитель простаивает (Рисунок 12).

Критерий шаблона:

$$I\_T(pid, p_i, c_j) = \begin{cases} 0, & \text{если } (Time\_start(MPI\_Recv) - Time\_start(MPI\_Ssend)) < \varepsilon \\ Time\_start(MPI\_Recv) - Time\_start(MPI\_Ssend), & \text{иначе} \end{cases} > 0$$

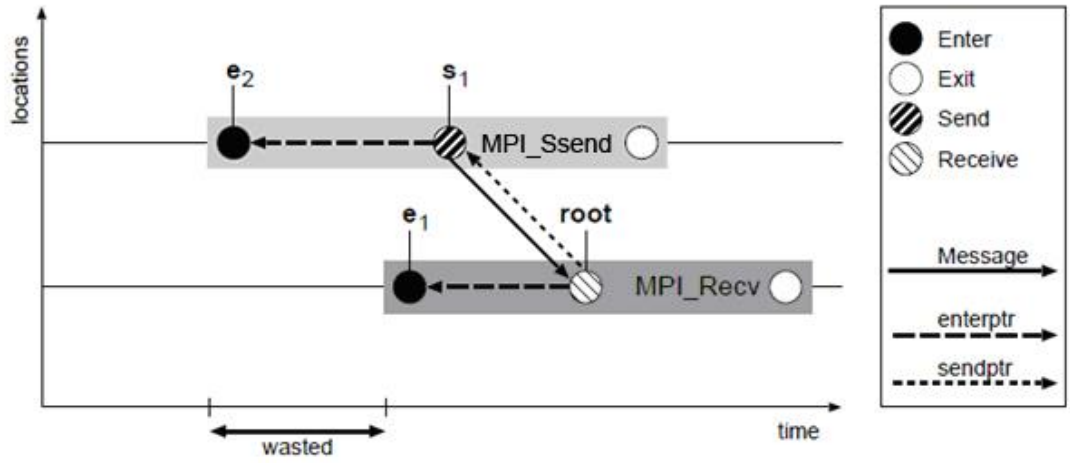


Рисунок 12. Поздний прием с передачей MPI_Ssend.

**Поздняя посылка по готовности(L_MPI_Rsend→MPI_Recv).** Прием сообщения начинается раньше, чем соответствующая посылка MPI_Rsend (Рисунок 13). В этом случае принимающая сторона (процесс 1) простаивает в ожидании.

Критерий шаблона:

$$I\_T(pid, p_i, c_j) = \begin{cases} 0, & \text{если } (Time\_start(MPI\_Rsend) - Time\_start(MPI\_Recv)) < \varepsilon \\ Time\_start(MPI\_Rsend) - Time\_start(MPI\_Recv), & \text{иначе} \end{cases} > 0$$

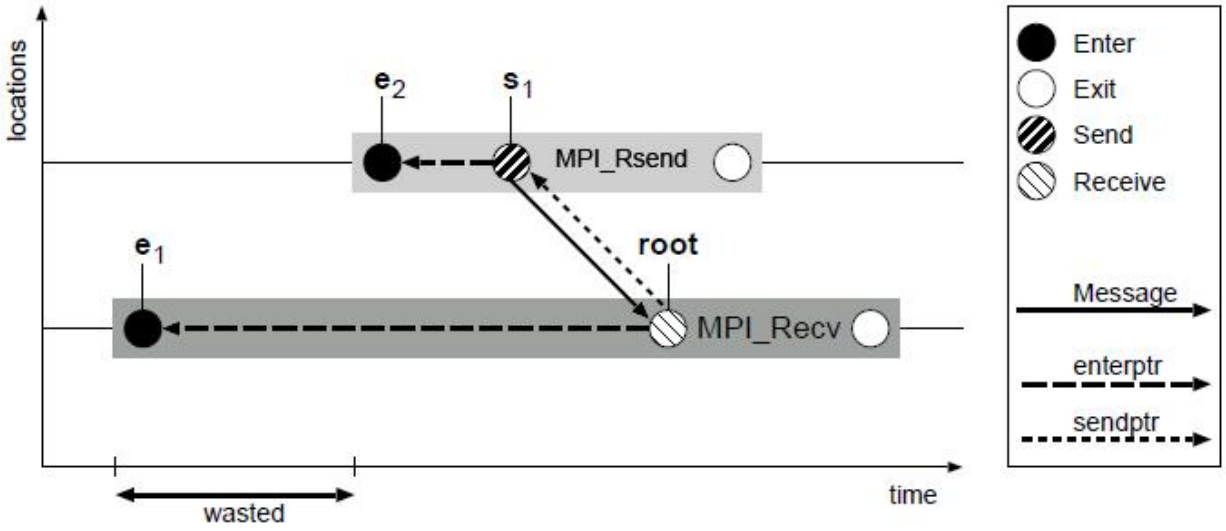


Рисунок 13. Ранний прием с передачей MPI_Rsend.

**Ранняя посылка по готовности(E_MPI_Rsend→MPI_Recv).** Посылка сообщения начинается раньше, чем соответствующий прием. Согласно стандарту MPI данная ситуация является ошибочной. Однако в текущей реализации MPICH, MVAPICH MPI_Rsend отображается на MPI_Send. Библиотека MPI не выдает сообщение об ошибке, а сообщение передается получателю. Основываясь на экспериментах, шаблон ранней посылки по готовности можно разбить на два случая:

а) Размер сообщения меньше определенной константы. Тогда применяется опережающая посылка, и операция MPI_Rsend является локальной. Следовательно, в этом случае простоя не будет и такая ситуация не приводит к снижению эффективности программы.

б) Размер пересылаемого сообщения больше определенной константы. В этом случае будет применяться протокол рандеву, что при поздней инициализации операции приема (MPI_Recv) приведет к простою процесса отправителя.

Рисунок 14 представляет графическое представление выше описанного шаблона.

Критерий шаблона:

$$I\_T(pid, p_i, c_j) = \begin{cases} 0, & \text{если } (Time\_start(MPI\_Recv) - Time\_start(MPI\_Rsend)) < \varepsilon \\ Time\_start(MPI\_Recv) - Time\_start(MPI\_Rsend), & \text{иначе} \end{cases} > 0$$

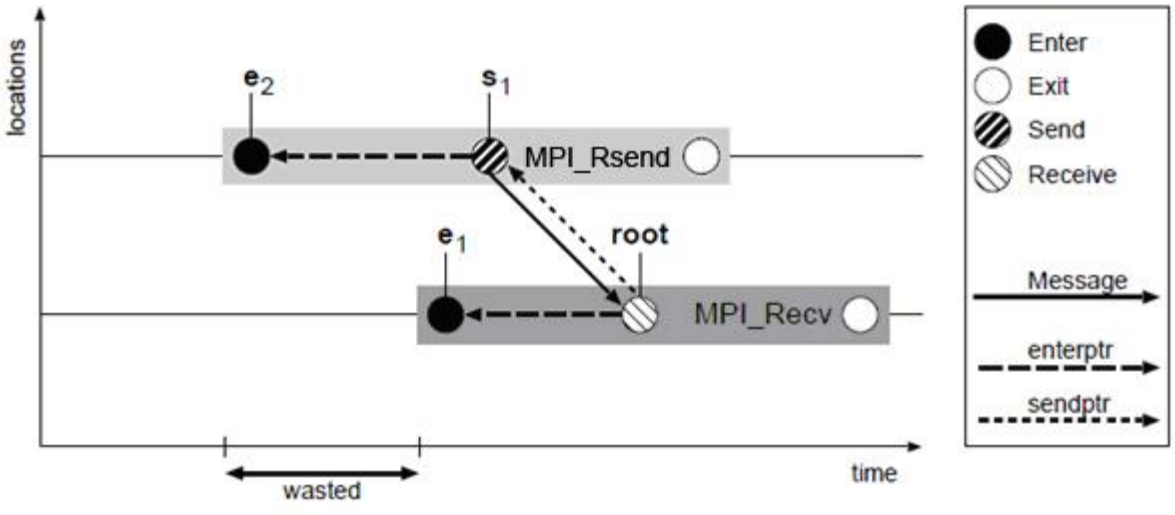


Рисунок 14. Поздний прием с передачей MPI_Rsend.

### 5.3.2 Шаблоны, связанные с «неправильным порядком сообщений»

Шаблон с «неправильным порядком сообщений» может возникнуть, когда в процессе-получателе сообщения ожидаются в одном порядке, а процесс-отправитель посылает сообщения в другом порядке (Рисунок 15).

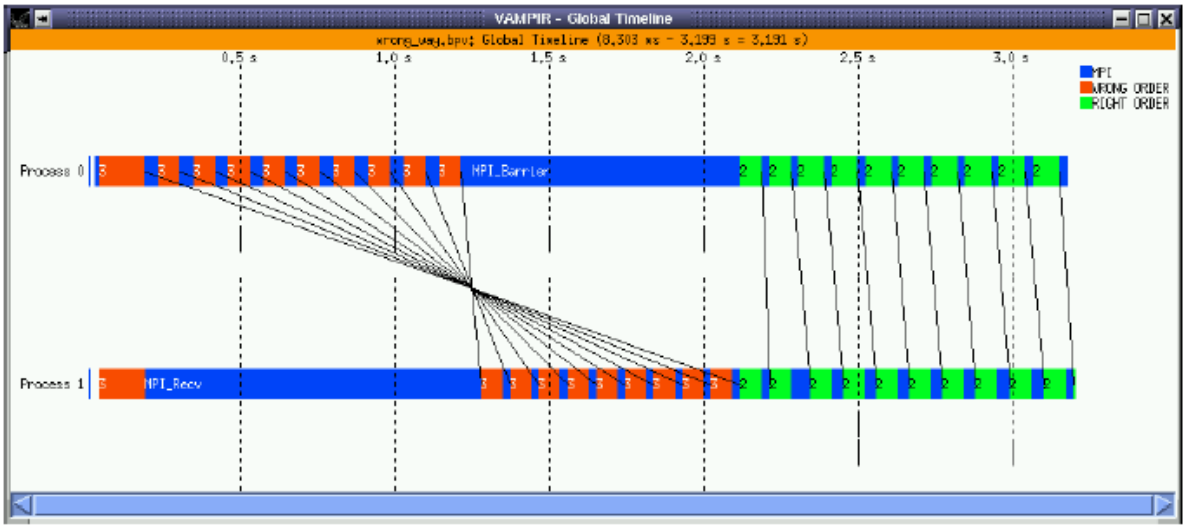


Рисунок 15. Шаблон «неправильный порядок».

Переупорядочив сообщения, можно не только добиться ускорения программы, но и уменьшения размера буфера для хранения необработанных сообщений.

Данный шаблон может возникнуть в двух случаях:

а) передача данных начинается раньше приема, и порядок отправленных сообщений отличается от порядка принимаемых сообщений.

б) прием данных начинается раньше передачи, и порядок отправленных сообщений отличается от порядка принимаемых сообщений.

Если при посылке/приеме используется протокол рандеву, то очевидно программа заблокируется (deadlock). Но если посылка локальная (буферизированная посылка, либо обычный send, но размер сообщения небольшой и сообщение в реализации MPI отправляется через внутренний MPI буфер), то блокировки не будет. В этом случае неэффективная организация коммуникаций приведет к шаблону с «неправильным порядком сообщений».

**Неправильный порядок с применением MPI_Send.** При применении пары функций {MPI_Send, MPI_Recv} возможны два варианта:

а) Размер пересылаемого сообщения небольшой и в этом случае используется протокол опережающей посылки, что приводит к тому, что вызовы MPI_Send не блокируются и можно получить шаблон «неправильный порядок». Верхняя граница размера для небольших сообщений отличается в разных реализациях MPI, также как и при разных настройках библиотеки MPI. В реализации MVAPICH2-1.7 при настройках по умолчанию верхняя граница составляет 64KB.

б) Размер пересылаемого сообщения больше границы, описанной в пункте а). В этом случае используется протокол рандеву. При использовании данного протокола для завершения функции MPI_Send необходимо, чтобы соответствующая функция MPI_Recv была начата. Но поскольку сообщения принимаются в другом порядке, то данная программа заблокируется.

На Рисунке 16 приведено графическое представление шаблона при пересылке сообщений в неправильном порядке с применением функции MPI_Send.

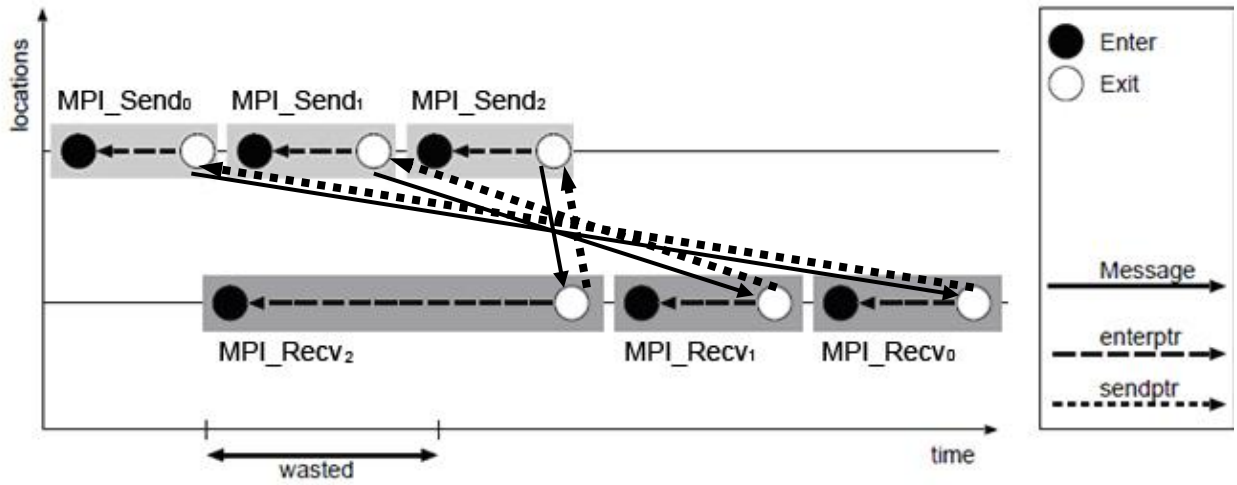


Рисунок 16. Шаблон «неправильный порядок» при применении пары функций {MPI_Send, MPI_Recv}.

**Неправильный порядок с применением MPI_Ssend**. Шаблон «неправильный порядок сообщений» невозможен при применении пары функций {MPI_Ssend, MPI_Recv}, поскольку в этом случае последовательные вызовы MPI_Ssend, не встретив соответствующих MPI_Recv-ов, заблокируются.

Критерий шаблона:

(Time_start(MPI_Send$_0$) < Time_start(MPI_Send$_1$)) &

(Time_start(MPI_Recv$_1$) < Time_start(MPI_Recv$_0$))

**Неправильный порядок с применением MPI_Bsend**. Шаблон «неправильный порядок» может возникнуть при использовании пары функций {MPI_Bsend, MPI_Recv} (Рисунок 17).

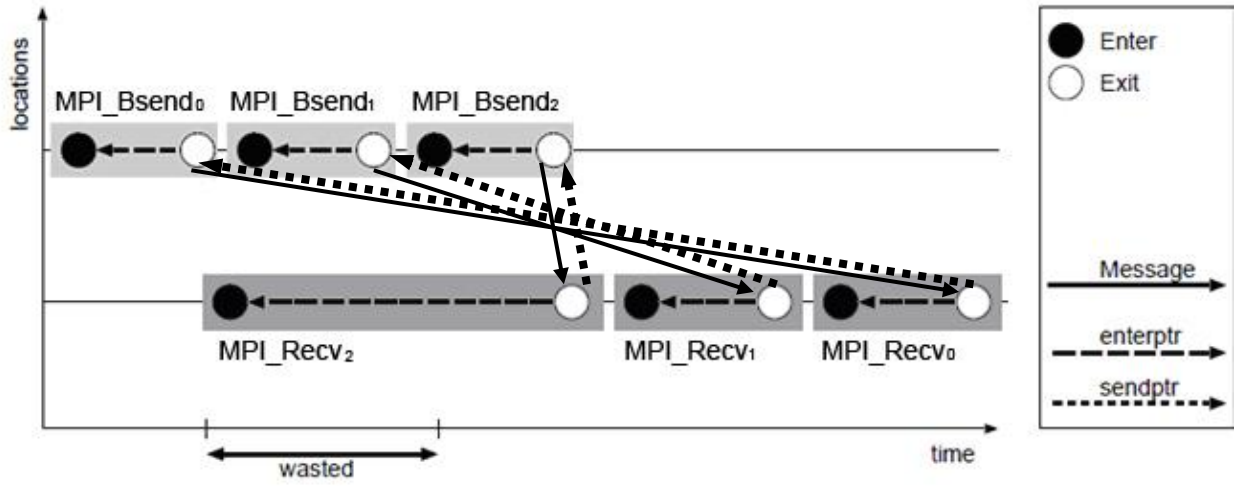


Рисунок 17. Шаблон «неправильный порядок» при применении пары функций {MPI_Bsend, MPI_Recv}.

Критерий шаблона:

(Time_start(MPI_Bsend$_0$) < Time_start(MPI_Bsend$_1$)) &

(Time_start(MPI_Recv$_1$) < Time_start(MPI_Recv$_0$))

**Неправильный порядок с применением MPI_Rsend**. Пусть на отправляющей и принимающей стороне используются функции MPI_Rsend и MPI_Recv. Как показали исследования в реализации MVAPICH-а при работе с {MPI_Rsend,MPI_Recv}-ом важно не абсолютное запаздывание соответствующего MPI_Recv-а, а относительный порядок остальных MPI_Recv-ов в очереди сообщений на стороне процесса-получателя. Рисунок 18 представляет схематическое описание коммуникаций в случае с последовательными посылками MPI_Rsend и последовательными приемами MPI_Recv. На принимающей стороне ($p_1$) существует очередь сообщений, где хранятся все сообщения, которые прибыли (либо прибыло служебное сообщение-

запрос на начало пересылки), но еще не были обработаны (не было вызова соответствующего MPI_Recv-а). После того как в $p_1$ был вызван $MPI\_Recv_2$ система времени выполнения берет первый элемент в очереди и поскольку $MPI\_Rsend_0$ не соответствует $MPI\_Recv_2$, то система регистрирует эту ошибку, но выдача ошибки откладывается. Далее подбирается следующий элемент из очереди ($MPI\_Rsend_1$) и поскольку соответствие найдено, производится прием сообщения в буфер $MPI\_Rsend_1$. Пользовательская программа продолжает выполнять инструкции. Если в программе не будет вызова $MPI\_Recv_0$, то программа завершится, и ошибки не будет. Если дальше по коду программы находится $MPI\_Recv_0$, то программа аварийно завершается с выдачей соответствующего сообщения.

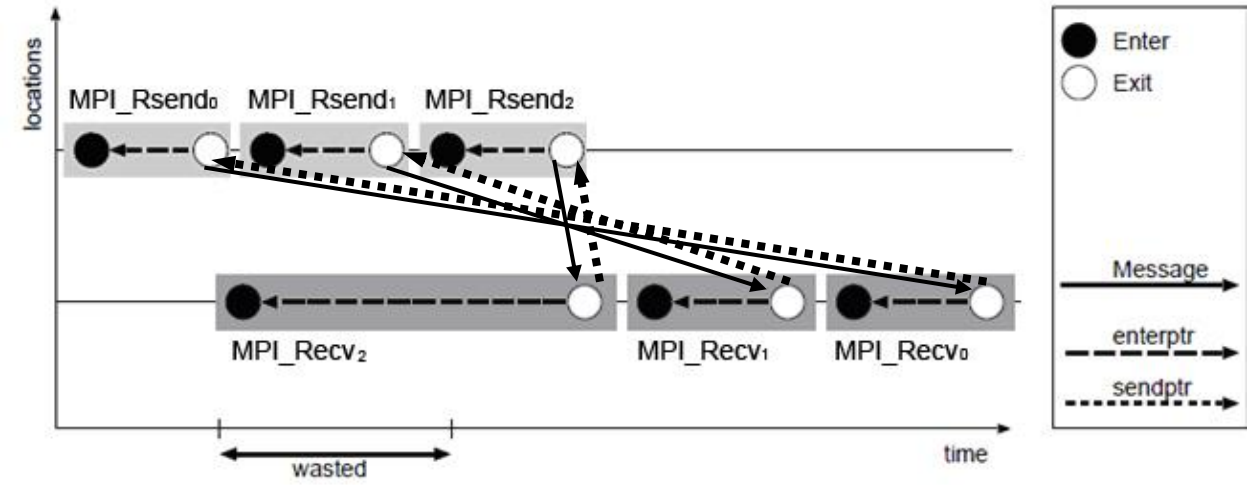


Рисунок 18. Шаблон «неправильный порядок» при применении пары функций {MPI_Rsend, MPI_Recv }.

Исходя из вышесказанного, шаблон «неправильный порядок» не применим к случаю, когда используется последовательность передачи сообщений посредством MPI_Rsend-ов.

### 5.3.3 Шаблоны, связанные с не-блокирующими точка-точка коммуникациями

Пусть имеется пара вызовов {MPI_Isend, MPI_Wait} в процессе $pid_0$ и {MPI_Irecv, MPI_Wait} в процессе $pid_1$.

**Ожидание на стороне отправителя при использовании {MPI_Isend, MPI_Irecv}**. Рассмотрим процесс $pid_0$. В этом случае после вызова функции

MPI_Isend управление возвращается в процесс $pid_0$ и выполняются вычислительные инструкции, после чего вызывается функция $MPI\_Wait_0$. Если вызов $MPI\_Wait_0$ был произведен слишком рано, то процесс блокируется и простаивает. В трассе событий содержатся временные метки для каждого события, следовательно, разница временных меток событий после и перед вызовом $MPI\_Wait_0$ позволит вычислить время простоя процесса.

Критерий шаблона:

$$I\_T(pid_0, p_i, c_j) = \begin{cases} 0, \text{ если } (Time\_end(MPI\_Wait_0) - Time\_start(MPI\_Wait_0)) < \varepsilon \\ Time\_end(MPI\_Wait_0) - Time\_start(MPI\_Wait_0), \quad \text{иначе} \end{cases} > 0$$

Помимо выявления шаблона, можно выдать диагностическое сообщение с оценкой оптимальной дистанции($O\_D(pid,p_i,c_j)$) для вызова $MPI\_Wait_0$.

$$O\_D(pid_0, p_i, c_j) = \begin{cases} \Delta Ti + T_{send}, \quad \text{если } Time\_start(MPI\_Isend) > Time\_start(MPI\_Irecv) \\ (Time\_start(MPI\_Irecv) - Time\_start(MPI\_Isend)) + \Delta Ti + T_{send}, \text{ иначе} \end{cases}$$ где

$\Delta T_i$ - время выполнения функции MPI_Isend, $T_{send}$ оценка времени реальной пересылки через коммуникационную сеть.

**Ожидание на стороне получателя при использовании {MPI_Isend, MPI_Irecv}**. При приеме сообщения в процессе $pid_1$ возникает аналогичная ситуация. После вызова функции MPI_Irecv управление возвращается в процесс $pid_1$ и выполняются вычислительные инструкции, после чего вызывается функция $MPI\_Wait_1$. Если вызов $MPI\_Wait_1$ был произведен слишком рано, то процесс блокируется и простаивает. Разница временных меток событий после и перед вызовом $MPI\_Wait_1$ позволит вычислить время простоя процесса.

Критерий шаблона:

$$I\_T(pid_1, p_i, c_j) = \begin{cases} 0, \text{ если } (Time\_end(MPI\_Wait_1) - Time\_start(MPI\_Wait_1)) < \varepsilon \\ Time\_end(MPI\_Wait_1) - Time\_start(MPI\_Wait_1), \quad \text{иначе} \end{cases} > 0$$

Оптимальная дистанция($O\_D(pid,p_i,c_j)$) для вызова $MPI\_Wait_1$ оценивается формулой:

$$O\_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, \quad \text{если } Time\_start(MPI\_Isend) < Time\_start(MPI\_Irecv) \\ (Time\_start(MPI\_Isend) - Time\_start(MPI\_Irecv)) + \Delta T_i + T_{send}, \text{ иначе} \end{cases}$$

где $\Delta T_i$ - время выполнения функции MPI_Irecv, $T_{send}$ оценка времени реальной

пересылки через коммуникационную сеть.

По аналогии с блокирующими точка-точка коммуникациями при использовании не-блокирующих коммуникаций можно выделить следующие шаблоны.

**Ожидание на стороне отправителя при использовании {MPI_Ibsend, MPI_Irecv}**. В этом случае шаблон не проявляется, потому что не-блокирующая функция MPI_Ibsend является локальной – функция копирует сообщение в буфер и возвращает управление, а система времени выполнения MPI занимается отправкой сообщения из буфера.

$I\_T(pid, p_i, c_j)=0$

**Ожидание на стороне получателя при использовании {MPI_Ibsend, MPI_Irecv}**. На стороне отправителя используется пара функций {MPI_Ibsend, $MPI\_Wait_0$}, на стороне получателя {MPI_Irecv, $MPI\_Wait_1$}. В этом случае шаблон может проявиться на стороне получателя. По аналогии с шаблоном при использовании пары **{MPI_Isend, MPI_Irecv}**:

Критерий шаблона:

$$I\_T(pid_1, p_i, c_j) = \begin{cases} 0, \text{ если } (\text{Time\_end}(MPI\_Wait_1) - \text{Time\_start}(MPI\_Wait_1)) < \varepsilon \\ \text{Time\_end}(MPI\_Wait_1) - \text{Time\_start}(MPI\_Wait_1), \quad \text{иначе} \end{cases} > 0$$

Оптимальная дистанция($O\_D(pid,p_i,c_j)$) для вызова $MPI\_Wait_1$ оценивается формулой:

$$O\_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, \quad \text{если Time\_start}(MPI\_Ibsend) < \text{Time\_start}(MPI\_Irecv) \\ (\text{Time\_start}(MPI\_Ibsend) - \text{Time\_start}(MPI\_Irecv)) + \Delta T_i + T_{send}, \text{ иначе} \end{cases}$$ гд

е $\Delta T_i$ - время выполнения функции MPI_Irecv, $T_{send}$ оценка времени реальной пересылки через коммуникационную сеть.

**Ожидание на стороне отправителя при использовании {MPI_Issend, MPI_Irecv}**. На стороне отправителя используется пара функций {MPI_Issend,$MPI\_Wait_0$}, на стороне получателя {MPI_Irecv, $MPI\_Wait_1$}. В этом случае после вызова не-блокирующей синхронной функции MPI_Issend управление возвращается в процесс $pid_0$. Если вызов соответствующей функции $MPI\_Wait_0$ был произведен слишком рано, то процесс блокируется и простаивает.

Критерий шаблона:

$$I\_T(pid_0, p_i, c_j) = \begin{cases} 0, \text{ если } (\text{Time\_end}(MPI\_Wait_0) - \text{Time\_start}(MPI\_Wait_0)) < \varepsilon \\ \text{Time\_end}(MPI\_Wait_0) - \text{Time\_start}(MPI\_Wait_0), \quad \text{иначе} \end{cases} > 0$$

Оптимальная дистанция($O\_D(pid,p_i,c_j)$) для вызова $MPI\_Wait_0$ оценивается формулой:

$$O\_D(pid_0, pi, cj) = \begin{cases} \Delta Ti + T_{send}, \quad \text{если Time\_start(MPI\_Issend)} > \text{Time\_start(MPI\_Irecv)} \\ (\text{Time\_start(MPI\_Irecv)} - \text{Time\_start(MPI\_Issend)}) + \Delta Ti + T_{send}, \text{ иначе} \end{cases}$$ где

$\Delta T_i$ - время выполнения функции MPI_Issend, $T_{send}$ оценка времени реальной пересылки через коммуникационную сеть.

**Ожидание на стороне получателя при использовании {MPI_Issend, MPI_Irecv}**. При приеме сообщения в процессе $pid_1$ возникает аналогичная ситуация. После вызова функции MPI_Irecv управление возвращается в процесс $pid_1$ и выполняются вычислительные инструкции, после чего вызывается функция $MPI\_Wait_1$. Если вызов $MPI\_Wait_1$ был произведен слишком рано, то процесс блокируется и простаивает.

Критерий шаблона:

$$I\_T(pid_1, p_i, c_j) = \begin{cases} 0, \text{ если } (\text{Time\_end}(MPI\_Wait_1) - \text{Time\_start}(MPI\_Wait_1)) < \varepsilon \\ \text{Time\_end}(MPI\_Wait_1) - \text{Time\_start}(MPI\_Wait_1), \quad \text{иначе} \end{cases} > 0$$

Оптимальная дистанция($O\_D(pid,p_i,c_j)$) для вызова $MPI\_Wait_1$ оценивается формулой:

$$O\_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, \quad \text{если Time\_start(MPI\_Issend)} < \text{Time\_start(MPI\_Irecv)} \\ (\text{Time\_start(MPI\_Issend)} - \text{Time\_start(MPI\_Irecv)}) + \Delta T_i + T_{send}, \text{ иначе} \end{cases}$$ гд

е $\Delta T_i$ - время выполнения функции MPI_Irecv, $T_{send}$ оценка времени реальной пересылки через коммуникационную сеть.

**Ожидание на стороне отправителя при использовании {MPI_Irsend, MPI_Irecv}**. На стороне отправителя используется пара функций {MPI_Irsend, $MPI\_Wait_0$}, на стороне получателя {MPI_Irecv, $MPI\_Wait_1$}. Данный случай является пересечением шаблона с использованием блокирующих функций {MPI_Rsend,MPI_Recv} и шаблона «Ожидание на стороне получателя» при использовании {MPI_Isend, MPI_Irecv}. Следовательно, шаблон может

проявиться только тогда, когда размер пересылаемого сообщения больше определенной константы. В этом случае будет применяться протокол рандеву, что при поздней инициализации операции приема (MPI_IRecv) приведет к простою процесса отправителя. Таким образом:

Критерий шаблона:

$$I\_T(pid_0, p_i, c_j) = \begin{cases} 0, \text{ если } (\text{Time\_end}(MPI\_Wait_0) - \text{Time\_start}(MPI\_Wait_0)) < \varepsilon \\ \text{Time\_end}(MPI\_Wait_0) - \text{Time\_start}(MPI\_Wait_0), \quad \text{иначе} \end{cases} > 0$$

Оптимальная дистанция (O_D(pid,$p_i$,$c_j$)) для вызова MPI_$Wait_0$ оценивается формулой:

$$O\_D(pid_0, p_i, c_j) = \begin{cases} \Delta Ti + T_{send}, \quad \text{если Time\_start}(MPI\_Irsend) > \text{Time\_start}(MPI\_Irecv) \\ (\text{Time\_start}(MPI\_Irecv) - \text{Time\_start}(MPI\_Irsend)) + \Delta Ti + T_{send}, \text{ иначе} \end{cases}$$ гд

е $\Delta T_i$ - время выполнения функции MPI_Irsend, $T_{send}$ оценка времени реальной пересылки через коммуникационную сеть.

**Ожидание на стороне получателя при использовании {MPI_Irsend, MPI_Irecv}**. На стороне отправителя используется пара функций {MPI_Irsend, MPI_$Wait_0$}, на стороне получателя {MPI_Irecv, MPI_$Wait_1$}. В этом случае шаблон может проявиться на стороне получателя.

Критерий шаблона:

$$I\_T(pid_1, p_i, c_j) = \begin{cases} 0, \text{ если } (\text{Time\_end}(MPI\_Wait_1) - \text{Time\_start}(MPI\_Wait_1)) < \varepsilon \\ \text{Time\_end}(MPI\_Wait_1) - \text{Time\_start}(MPI\_Wait_1), \quad \text{иначе} \end{cases} > 0$$

Оптимальная дистанция (O_D(pid,$p_i$,$c_j$)) для вызова MPI_$Wait_1$ оценивается формулой:

$$O\_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, \quad \text{если Time\_start}(MPI\_Irsend) < \text{Time\_start}(MPI\_Irecv) \\ (\text{Time\_start}(MPI\_Irsend) - \text{Time\_start}(MPI\_Irecv)) + \Delta T_i + T_{send}, \text{ иначе} \end{cases}$$ где

$\Delta T_i$ - время выполнения функции MPI_Irecv, $T_{send}$ оценка времени реальной пересылки через коммуникационную сеть.

### 5.3.4 Близкая посылка, передача сообщений

Рассмотрим программу, где в процессе $pid_i$ применяется посылка сообщения и прием сообщения с процессом $pid_j$, и операции посылки и приема находятся близко. Если было найдено такое использование функций и логика программы

позволяет, то можно объединить функции MPI_Send, MPI_Recv в функцию MPI_Sendrecv. В этом случае получится серьезный выигрыш по времени выполнения. Происходит это из-за того, что в реализациях MPI достаточно хорошо реализована функция MPI_Sendrecv. Помимо этого, на нижнем уровне современные высокопроизводительные коммуникационные сети (например, Infiniband [54]) поддерживают дуплексную связь, что позволяет одновременно посылать и получать сообщение одному HCA (Host Channel Adapter). И в этом случае, вместо того, чтобы процесс $pid_i$ ждал окончания операции MPI_Send и только после этого начал ждать окончания MPI_Recv, посылка и получение производятся одновременно.

Пусть $\Delta_{SR}$ некоторая предопределенная константа.

Критерий шаблона:

(Time_start(MPI_Recv) - Time_end(MPI_Send)) < $\Delta_{SR}$

При нахождении этого шаблона выдается диагностическое сообщение. И пользователь, убедившись, что не существует зависимости по данным для буферов send_buf и recv_buf, может заменить пару вызовов функций {MPI_Send, MPI_Recv} на вызов функции MPI_Sendrecv.

# 6. Описание программного обеспечения среды ParJava

Среда ParJava интегрирована с открытой средой разработки Java-программ Eclipse [55]. Среда Eclipse обеспечивает графический интерфейс для инструментов среды ParJava. Eclipse – это свободно распространяемая интегрированная среда разработки программного обеспечения, охватывающая все этапы разработки: создание файлов с исходным кодом, ведение проектов, средства для работы с системами контроля версий, отладчик и.т.д. Среда Eclipse предоставляет возможность подключения к ней различных модулей по специфицированным интерфейсам.

После подключения к Eclipse инструментальных средств ParJava, получилась единая среда разработки SPMD-программ, включающая как инструменты среды ParJava, так и инструменты среды Eclipse: текстовый редактор, возможность создания и ведения проектов, возможность инкрементальной трансляции исходной программы во внутреннее представление, интерпретатор модели и.т.д. Интеграция ParJava в среду Eclipse осуществляется с помощью механизма «подключаемых модулей». Подключаемый модуль (plug-in) представляет собой jar-файл, содержащий классы модуля.

При разработке параллельной Java-программы сначала необходимо средствами среды Eclipse создать Java-проект. После этого, чтобы получить возможность пользоваться инструментами ParJava, необходимо конвертировать Java-проект в ParJava-проект (Рисунок 19).

После того как проект был создан, пользователь может приступить к построению модели программы. С этой целю Java-файлы, для которых требуется построить модель, помечаются в директории, в которой они хранятся. При генерации модели параллельно генерируется модуль для оценки времени выполнения базовых блоков. Java-файлы пользователя, которые необходимо моделировать, помечаются с помощью контекстного подменю “ParJava->Model” в окне Package Explorer или Navigator. С помощью подменю “ParJava-> Don’t

model“ можно отменить моделирование файла. На иконках файлов, для которых модель уже построена, появляется буква “m”. Сначала из исходного кода получается дерево разбора программы, которое передается генератору внутреннего представления (ГВП). ГВП основан на свободных пакетах [56, 57] и создает модифицированное абстрактное синтаксическое дерево (МАСД) программы, внутреннее представление базовых блоков.

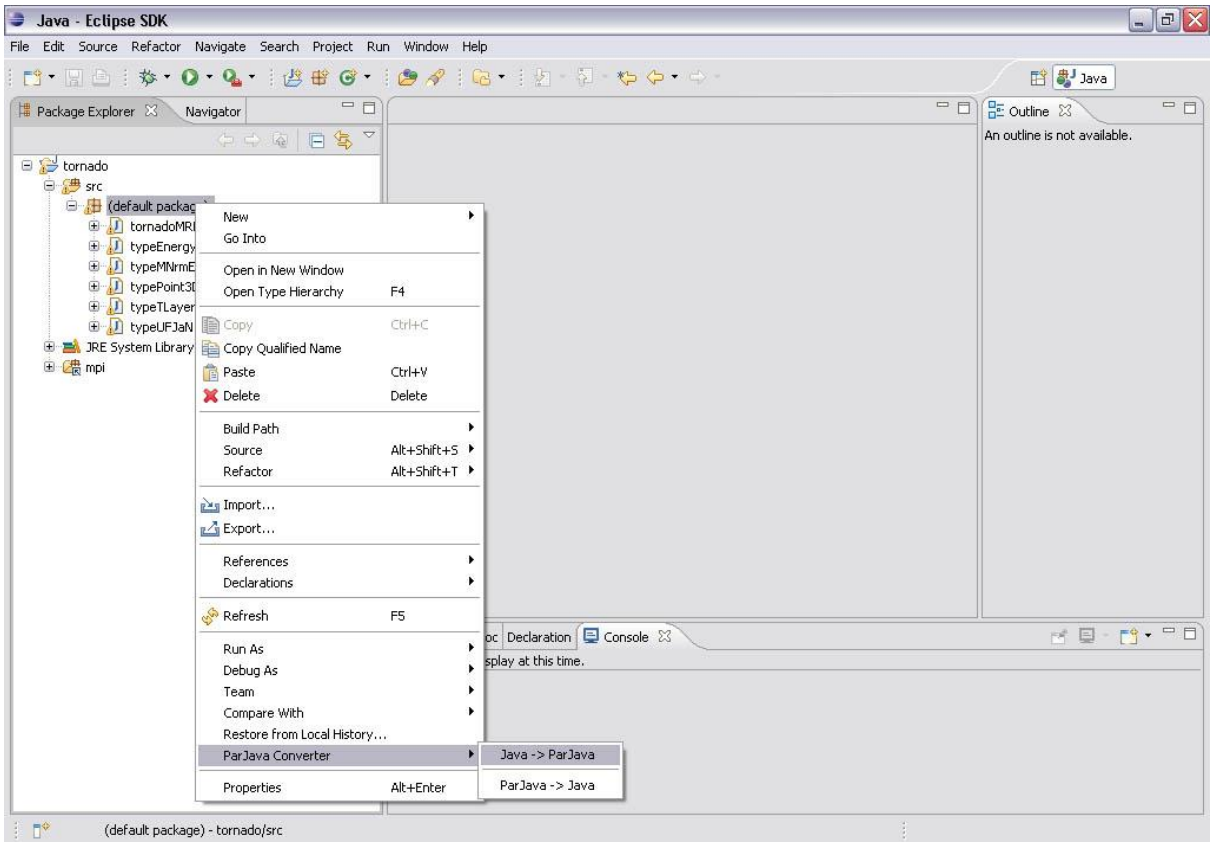


Рисунок 19. Конвертация Java-проект в ParJava-проект.

Модель программы в среде описывается иерархией классов. На Рисунке 20 приведены классы верхнего уровня. Класс Vertex является родителем для всех классов модели. Класс Statement является общим предком для всех управляющих операторов языка Java (if, if-else, for, while итд). Класс Expression представляет собой общий родитель для всех выражений языка Java (унарное выражение, бинарное выражение, доступ к переменной и.т.д.). DescriptorVertex это общий родительский класс для описателей разных сущностей: описатель файла, описатель класса, описатель поля, описатель метода и описатель переменной. DescriptorVertex содержит два поля: имя и битовое поле, описывающее модификаторы описываемой структуры (уровень доступа, видимость,

волатильность, синхронизированность и.т.д.). Эти два атрибута наследуются потомками и используются при необходимости. Каждому java файлу соответствует объект класса FileDescriptor. Согласно стандарту JLS [37] в Java файле могут содержаться больше одного класса, поэтому в FileDescriptor хранится массив описателей класса (ClassDescriptorVertex), имя пакета и массив импортируемых пакетов. Описатель класса содержит массив описателей методов (MethodDescriptorVertex), массив описателей полей (FieldDescriptorVertex), имя базового класса (суперкласс), список реализуемых интерфейсов. Описатель метода содержит сигнатуру метода согласно спецификации Java [37], массив описателей переменных (VariableDescriptorVertex), описывающих формальные параметры метода, массив выбрасываемых исключений и ссылку на тело метода. Описатель поля (FieldDescriptorVertex) помимо наследуемых полей содержит еще тип поля и ссылку на инициализирующее выражение.

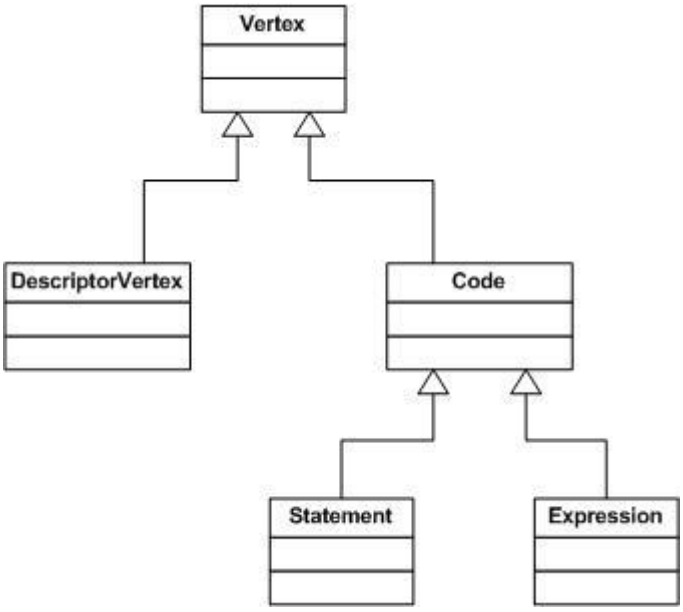


Рисунок 20. Внутреннее представление - классы верхнего уровня.

На Рисунке 21 приведена иерархия классов описывающих управляющие операторы. Родительский класс Statement содержит общие поля для всех потомков: id – уникальный идентификатор оператора (инструкции), labels – метки оператора, e – управляющее выражение оператора, r – список потомков оператора.

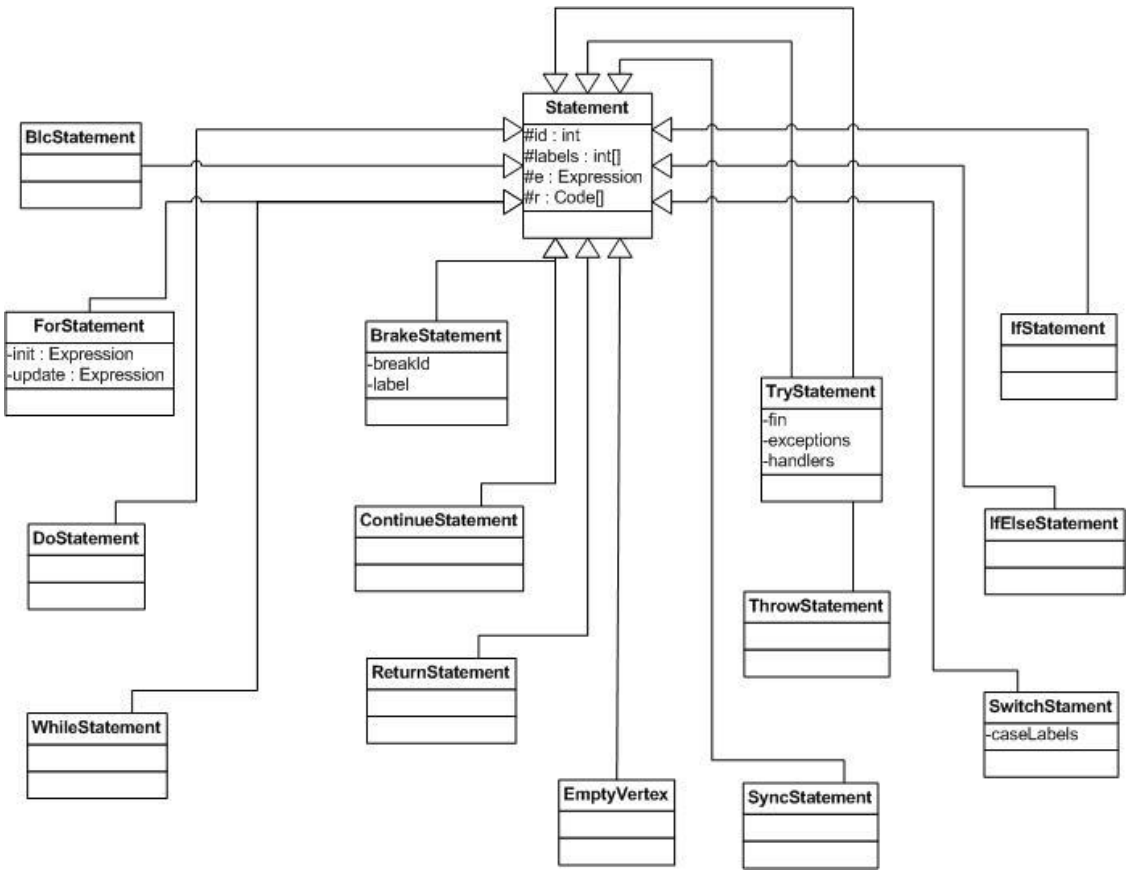


Рисунок 21. Внутреннее представление - иерархия классов управляющих операторов.

На Рисунке 22 приведена схема получения оценок базовых блоков (фрагментов). Вначале из МАСД получается оценочный код (ОК). ОК вместе с тестовым набором начальных данных моделируемой параллельной программы (эти данные предоставляются пользователем) выполняется на узле целевой вычислительной системы. В ходе двух проходов на тестовом наборе данных на узле целевой вычислительной системы получаются оценки времени выполнения базовых блоков и фрагментов (горячих участков) программы. Файл с оценками времени выполнения базовых блоков переносится на инструментальный компьютер, где полученные оценки вносятся в дескрипторы базовых блоков приведенной модели, образуя модель потока управления. На этом завершается генерация модели метода и подготовка ее к интерпретации.

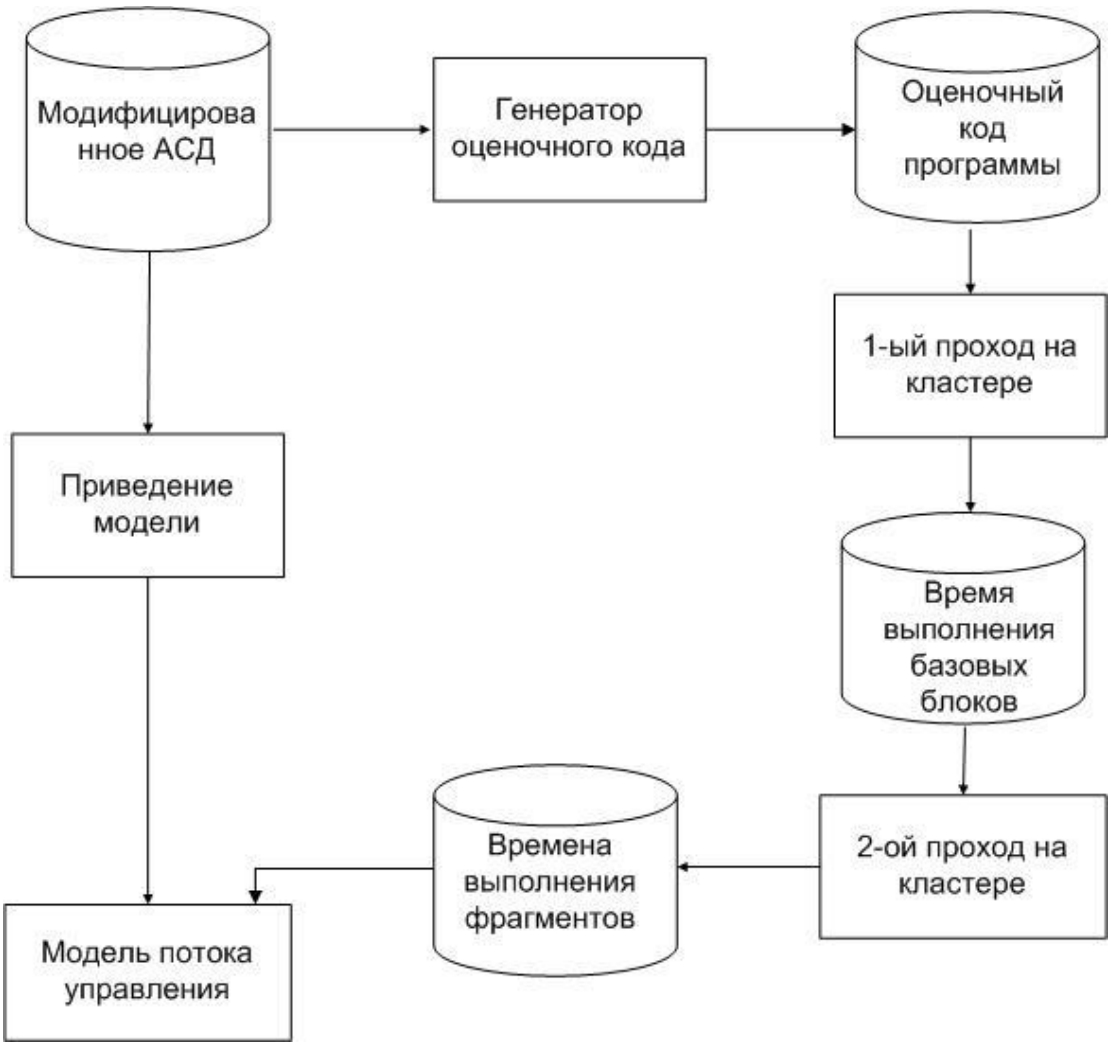


Рисунок 22. Схема получения оценок времени выполнения.

Для отображения модели программы используется окно ParJava Model view. Чтобы его открыть, необходимо выбрать пункты меню Window->Show View->Other, в появившемся окне выбрать ParJava->ParJava Model. В результате откроется окно ParJava Model view (Рисунок 23). В этом окне отображаются файлы моделей текущего проекта. Текущий проект определяется по открытому в данный момент в редакторе файлу. В данном случае в редакторе открыт файл torhomo.java, который находится в проекте tornado. Поэтому текущим проектом является проект tornado. При изменении моделей программы содержимое окна будет автоматически обновлено.

Для блоков модели программы используется следующая цветовая маркировка: зеленым выделены базовые блоки модели, желтым выделены внутренние вершины модели.

Если для блока определены координаты текста, соответствующего этому блоку, то при выборе этого блока будет автоматически произведена подсветка текста.

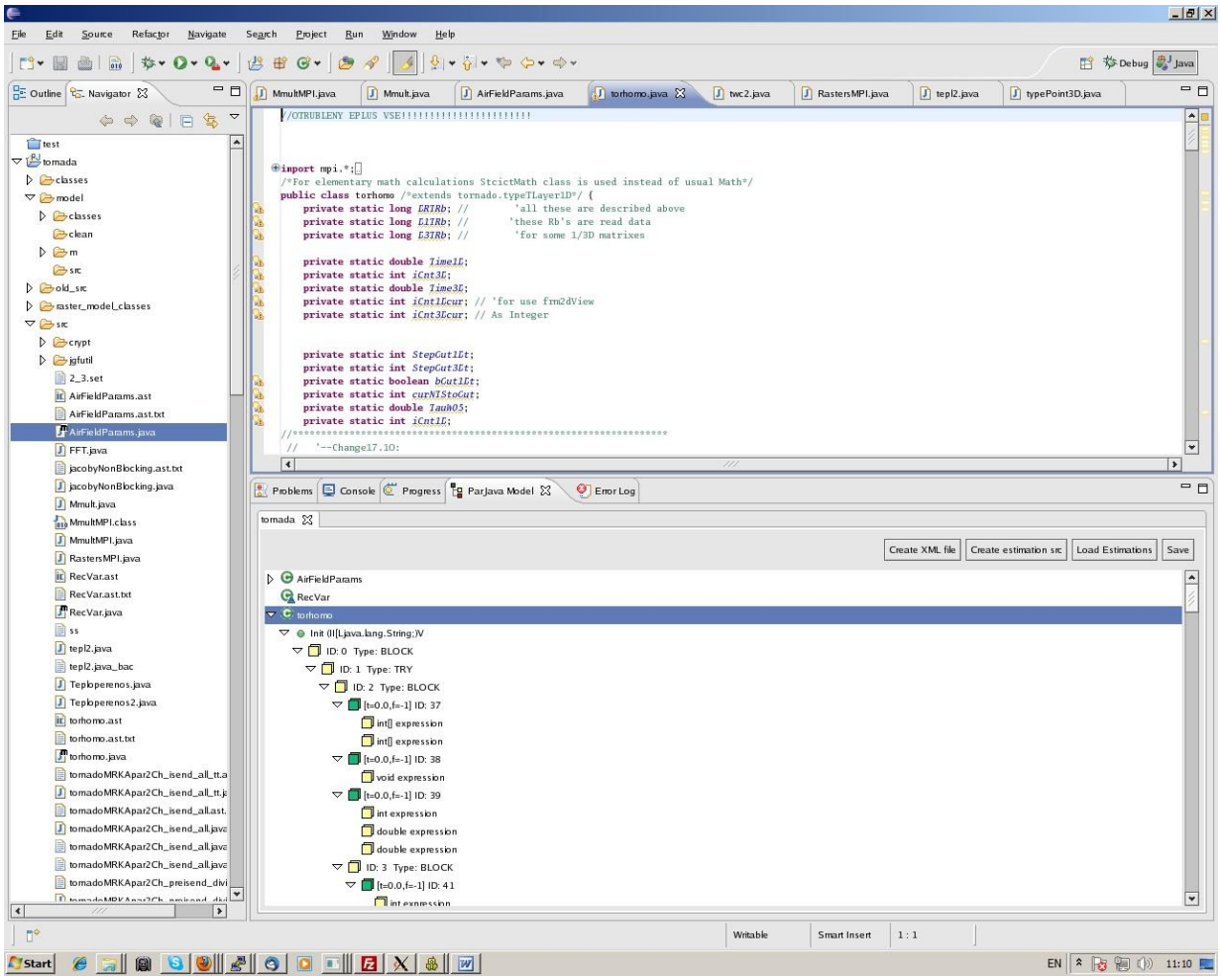

Рисунок 23. Отображение модели программы.

Для хранения файлов модели используется папка model в корне проекта (например, для проекта tornado это tornado/model). При этом в папке model создаются папки:

1. classes – в данной папке хранится откомпилированный деструктурированный код.
2. clean – в данной папке хранится очищенный исходный код (используется для отладочных целей), а в папке est – оценочный код (ОК).
3. m – данная папка содержит .m-файлы, в которых содержится модель соответствующего класса.
4. src – в папке хранится деструктурированный код.

Для интерпретатора ParJava файлы моделей из папки m копируются в папку classes.

ОК необходимо перенести на узел целевого вычислительного комплекса и запустить. Причем необходимо подобрать параметры (количество процессов на узле итд) запуска ОК в соответствии с параметрами реального запуска параллельного приложения. В результате работы ОК создает файл data.est, который необходимо перенести обратно на инструментальный компьютер и с помощью кнопки Load Estimations в ParJava Model view (Рисунок 23) добавить в модель, после этого модель готова к интерпретации.

Помимо этого необходимо получить конфигурационный файл, содержащий параметры, необходимые для моделирования коммуникационных примитивов. Конфигурационный файл имеет определенный формат и содержит такие параметры вычислительной платформы как время копирования на узле, латентность, таблица полосы пропускания, таблица времен инициализации, время создания потока, время запуска потока и.т.д. Конфигурационный файл можно получить, запустив класс ru.ispras.ParJava.instr.NetworkTester на двух узлах вычислительной платформы. Данный тест предполагает, что каждый узел кластера содержит восемь ядер.

Ниже приведен пример скрипта запуска этого теста с помощью системы Torque [58].

```
#PBS -l walltime=02:30:00,nodes=2:ppn=8
#PBS -N run_network_test
#!/bin/sh
cd /home/manuk/estim
/home/manuk/workspace/mpiJava/src/scripts/prunjava -
Xbootclasspath/p:/home/manuk/workspace/mpiJava/lib/bootclasses -
Djava.library.path=/home/manuk/workspace/mpiJava/lib:. –d64 -Xmx200M -Xrs
ru.ispras.ParJava.instr.NetworkTester
```

Полученный на выходе тестовой программы конфигурационный файл переносится на инструментальный компьютер и при интерпретации модели указывается в параметрах запуска интерпретатора. Для запуска интерпретатора модели выбираем в главном меню Eclipse пункты Run -> Run…

В диалог запуска добавлена возможность создавать конфигурации для запуска интерпретатора – ParJava Interpreter. В конфигурации необходимо указать следующие параметры:

Закладка *Main* (Рисунок 24)

1. *Name* – имя конфигурации
2. *Project* – имя проекта, содержащего интерпретируемую программу,
3. *Main class* – имя запускаемого класса,
4. *Program arguments* – входные аргументы программы
5. *Java VM arguments* – ключи к Java VM.

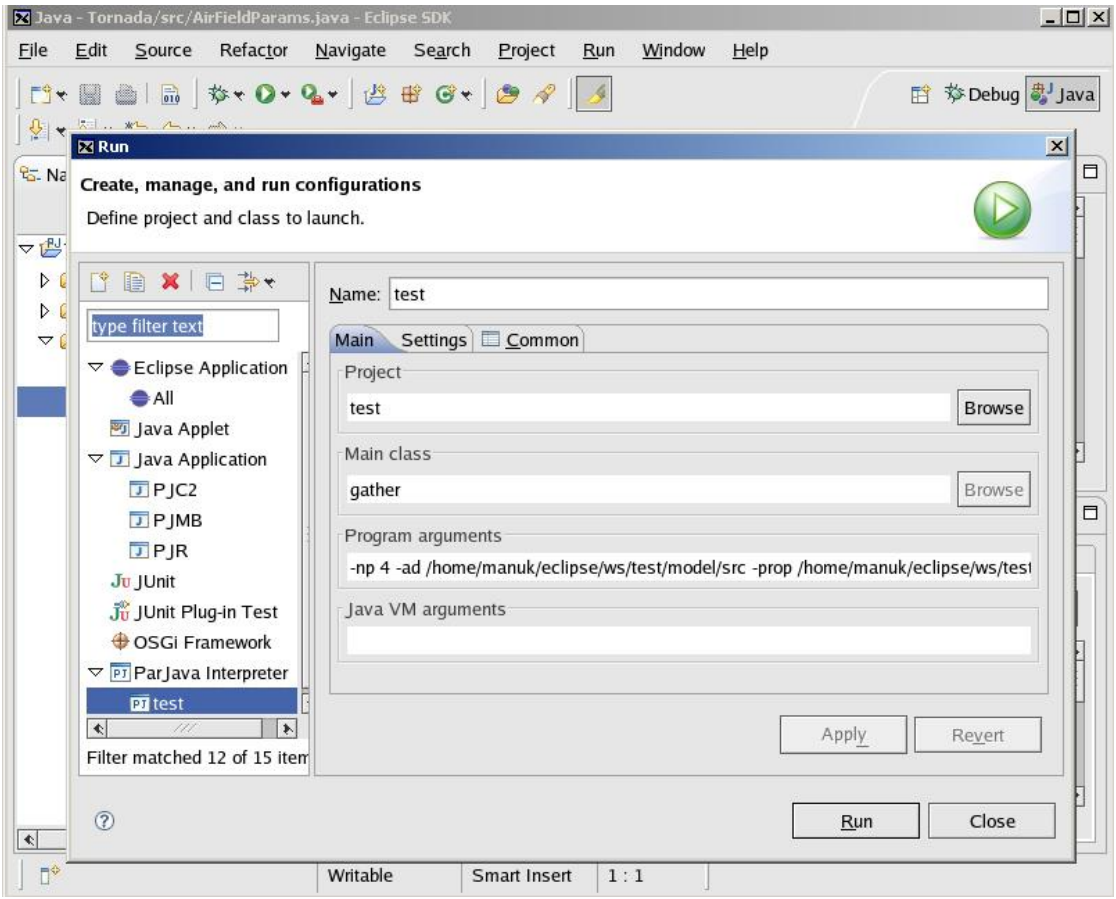


Рисунок 24. Диалоговое окно запуска интерпретатора.

**Замечание**. По умолчанию в системе выключена реальная передача данных. Коммуникации моделируются корректно, время вычисляется, но реального копирования данных не происходит. Для того, чтобы включить реальное копирование при интерпретации в параметрах «Java VM arguments», необходимо добавить опцию “-DTRANSFER=ON”.

Закладка Settings

1. *Processors number* – количество моделируемых процессов.
2. *Interpreter arguments* – параметры к интерпретатору (например, “–prop имя_конфигурационного_файла”).

Результаты интерпретации выводятся в окно консоли.

# 7. Практическое применение среды ParJava

В данном разделе приведены практические рекомендации по улучшению производительности параллельной многопоточной Java-программы с явными обращениями к библиотеке MPI и результаты численных экспериментов. Исследованные в 7.1 особенности позволили существенно улучшить производительность параллельных SPMD программ приведенных разделе 7.2.

## 7.1 Доводка производительности параллельной программы в среде Java

В разделе приводится описание особенностей, призванных повысить производительность параллельной программы. Производительность Java программы зависит от многих факторов[59], в частности, от размера задачи. Это может привести к неожиданным и неоднократным «сборкам мусора», JIT перекомпиляции.

В данном разделе вначале сравнивается время выполнения «чтение данных» соседнего процесса программы и время выполнения «чтение данных» соседнего потока программы. После этого приводится описание нескольких особенностей связанных с реализацией JVM, влияющих на производительность параллельной программы. Использование специфических факторов, описанных в данном разделе, позволили повысить производительность параллельных SPMD программ, разрабатываемых в среде ParJava.

### 7.1.1 Операция «чтение данных» соседнего процесса/потока Java

Рассмотрим время выполнения операции «чтение данных соседнего процесса/потока» для многопроцессной программы, в котором используются не-блокирующие и блокирующие коммуникаций MPI, и для многопоточной программы.

Пусть имеется многопроцессная программа P1 и многопоточная программа P2. Для программы P1 (P2) в каждом процессе (потоке) производятся вычисления и в определенный момент процесс (поток) 0 становится потребителем ($p_0$) и

должен получить доступ к данным процесса (потока) 1, который в этом взаимодействии является производителем ($p_1$).

В многопроцессной программе P1 при использовании неблокирующих коммуникаций время выполнения операции «чтение данных соседнего процесса» определяется следующей формулой:

$$\text{time}_{\text{wait}} = \begin{cases} (t_{ready} - t_0) + time_{comm}, t_{ready} > t_0 \\ (t_0 - t_{ready}) > time_{comm} \,?\, 0 : time_{comm} - (t_0 - t_{ready}), t_{irecv} < t_{ready} < t_0 \\ (t_0 - t_{irecv}) > time_{comm} \,?\, 0 : time_{comm} - (t_0 - t_{irecv}), t_{ready} < t_{irecv} \end{cases}$$

где $t_{irecv}$ - момент времени вызова IRecv в процессе $p_0$, $t_0$ - момент времени вызова wait в процессе $p_0$, $t_{ready}$ - момент времени, когда в процессе $p_1$ готовы данные, $time_{comm}$ - время, необходимое для пересылки данных из процесса-производителя к процессу-потребителю и копирования в буфер памяти в процессе-потребителе, $time_{wait}$ – время выполнения функции wait.

При использовании блокирующих коммуникаций:

$$\text{time}_{\text{wait}}=\text{time}_{\text{recv}}=\begin{cases} (t_{ready} - t_0) + time_{comm}, t_{ready} > t_0 \\ time_{comm}, t_{ready} < t_0 \end{cases}$$

где $time_{recv}$ – время выполнения функции Recv, $t_0$ - момент времени вызова Recv в процессе $p_0$, $t_{ready}$ - момент времени, когда в процессе $p_1$ готовы данные, $time_{comm}$ - время, необходимое для пересылки данных из процесса $p_1$ к процессу $p_0$ и копирования в буфер памяти в процессе $p_0$.

В случае с **многопоточной** программой P2 поток-производитель $p_1$ вызывает функцию блокирования lock и начинает вычислять данные. Как только данные вычислены, поток разблокирует критическую секцию. Поток-потребитель $p_0$ не может считать данные до тех пор, пока $p_1$ не разблокирует критическую секцию ($p_0$ блокируется в lock-е). Время выполнения операции «чтение данных» соседнего потока определяется следующей формулой:

$$\text{time}_{\text{lock}}=\begin{cases} 0; t_{ready} \le t_0 \\ t_{ready} - t_0, t_{ready} > t_0 \end{cases}$$

где $t_0$ - момент времени вызова lock в потоке-потребителе $p_0$, $t_{ready}$ - момент времени, когда в потоке-производителе $p_1$ готовы данные, $time_{lock}$ – время выполнения функции lock.

Таким образом, в многопроцессной программе P1 на операцию «чтение данных соседнего процесса» тратится больше времени, чем в многопоточной программе P2.

Когда процесс вызывает функцию wait (явно при не-блокирующей посылке или посредством recv), то при определенных условиях (не готовы данные или производится пересылка) вызов блокируется. Блокирование происходит внутри реализации MPI и в зависимости от того как реализована операция блокирования в MPI, это может возыметь определенные последствия:

1. если lock реализован как lock-sleep, то произойдет переключение контекста и когда данные прибудут, процесс пробудится и опять будет произведено переключение контекста
2. если lock реализован посредством spin-lock, то процесс останется активным и переключений контекстов не произойдет.

В случае с **многопоточной** программой операцию lock определяет пользователь. Если он явно организует spin-lock, то очевидно контекст переключаться не будет. Когда пользователь вызывает функцию, то какой именно подход блокирования будет выбран, зависит от реализации JVM.

**Переключение контекста** не является бесплатной операцией – управление CPU передается JVM и операционной системе, которые сохраняют данные потока и отправляют его в режим ожидания (sleep). Когда поток должен пробудиться, то система (JVM) загружает его фрейм и данные в память. И скорее всего это переключение повлечет за собой кэш-промахи, что в свою очередь также отразиться на производительности.

### 7.1.2 Синхронизация потоков при обращении к данным в общей памяти Java

При использовании общей памяти потоками параллельной программы значительный вес в доводке производительности программы получают особенности, связанные с синхронизацией доступа потоков к общим данным. В Java синхронизация данных ведет к появлению дополнительных накладных расходов: синхронизация обеспечивается специальными инструкциями барьеров

памяти (memory barriers). Эти инструкции могут сбрасывать кэш, аппаратные буферы, а также повлиять на другие оптимизации компилятора (например, предотвратить переупорядочивание инструкций). Синхронизация бывает «оспариваемая» (**contended**) и «неоспариваемая» (**uncontended**). «Неоспариваемая» синхронизация имеет низкую стоимость [60]. «Неоспариваемая» синхронизация может обрабатываться внутри JVM, в то время как «оспариваемая» требует вмешательства операционной системы, что увеличивает ее стоимость. Динамический компилятор Java может сократить стоимость синхронизации, заменив «оспариваемую» синхронизацию «неоспариваемой», когда докажет, что за данный барьер никогда не производится спор между потоками (contention). Существует три способа сократить «спор»:

- Сократить продолжительность блокирования
- Сократить частоту запросов блокирования
- Заменить исключающую блокировку механизмами позволяющими улучшить параллелизм

### 7.1.3 Особенности, связанные со «сборщиком мусора» в Java

В **многопроцессной** программе каждый процесс запускается на отдельной JVM. Необходимая память для работы процесса выделяется системой управления памяти **своей JVM**. В этом случае, для каждого процесса работает **свой сборщик мусора(gc)** [61]. В случае же с многопоточной программой, память, необходимая всем потокам программы, выделяется на одной JVM, и это нужно учесть при запуске программы в опциях интерпретатора (xmx,…). Помимо этого в этом случае работает только **один gc**, который обслуживает все потоки программы. Следовательно, давление на gc возрастает пропорционально количеству потоков программы.

В JVM сборщик мусора время от времени запускается в отдельном потоке и может стать узким местом в параллельной программе. Для улучшения производительности необходимо уменьшить время работы gc, а в идеальном случае и вовсе свести к нулю. Сборщик мусора перебирает динамическую память

(куча) и выделяет (маркирует) те области, на которые есть живые ссылки. После этапа маркировки производится чистка всех немаркированных областей. Обычно большинство объектов, выделенных в куче, используются недолго (умирают рано), они хранятся в молодом поколении. Другие объекты могут жить до конца работы программы, и они хранятся в долгоживущем поколении. Процесс сборки мусора можно оптимизировать, используя эффект «ранней смерти». Общая память в куче разбивается на несколько областей:

1. Молодое поколение(young generation) – хранятся молодые объекты.
2. Долгоживущее поколение(tenured) – хранятся долгоживущие объекты.
3. Перманентная область (permanent) –хранятся служебные данные JVM.

Как только в поколении не остается места, производится сборка мусора для данного поколения: минорная сборка(minor collection) для молодого поколения и мажорная сборка (major collection) для долгоживущего поколения. Большинство выделенных объектов умирают молодыми, следовательно, минорная сборка занимает меньше времени, потому что время сборки прямо пропорционально количеству живых объектов. При создании объекта в программе, производится попытка выделить память в молодом поколении, если этого не удается (молодое поколение переполнено), то производится минорная сборка (minor collection), чистящая молодое поколение. Во время минорных сборок, объекты, которые прожили достаточно долго, переносятся в долгоживущее поколение. Как только долгоживущее поколение переполняется, производится мажорная сборка, которая работает намного медленнее из-за большого количества живых объектов в нем.

Для уменьшения влияния сборщика мусора на производительность программы рекомендуется:

1. Большие матрицы и общие данные выделить в самом начале программы.
2. Вычислить объем данных выделенных в первом шаге, и в соответствии с этим задать минимальный и максимальный размер кучи (Xmx,Xms).
3. Задать размеры поколений (NewRatio, NewSize, MaxNewSize).

Чем больше размер молодого поколения, тем меньше будет минорных сборок. С другой стороны, чем больше размер молодого поколения, тем меньше

размер долгоживущего поколения, что в свою очередь приведет к частым мажорным сборкам. Поэтому размеры поколения нужно выбрать так, что большие матрицы, выделенные в начале программы, попадали сразу в долгоживущее поколение, а так же оценить размер памяти, который будет выделяться по ходу выполнения программы, и задать соответствующий размер молодого поколения во избежание минорных сборок.

### 7.1.4 Особенности, связанные с буфером локальной памяти потоков в Java

В многопоточной программе выделение памяти потоками производится параллельно. В общем случае все потоки выделяют память в общей куче. Каждый раз при выделении памяти в общей куче (heap) система управления памяти ставит блокировку (lock), выделяет память и снимает блокировку (отпускает lock), что может повлиять на производительность общей программы. Поэтому рекомендуется большую матрицу и общие данные выделить в самом начале программы, а локальные данные для каждого потока выделять в локальной области, что позволит избежать lock-ов на общей куче.

Для улучшения производительности при запуске каждого потока JVM создает для него локальный буфер для выделения памяти (TLAB). Во время работы потока при выделении памяти вначале производится попытка выделить память из области TLAB-а и, если эта попытка оказывается неудачной, то память выделяется в общей куче со всеми вытекающими последствиями. При запуске программы необходимо знать, сколько памяти будет выделено во время работы потока, и задать соответствующие корректировки относительно работы с TLAB-ами с помощью опций интерпретатора Java (ResizeTLAB, TLABSize,…).

## 7.2 Результаты численных экспериментов

В данном разделе приводится описание результатов численных экспериментов для двух программ-примеров, использовавшихся при реализации среды ParJava. Эти примеры позволили выявить некоторые особенности кластеров с многоядерными узлами и показали применимость среды ParJava при разработке реальных прикладных программ.

1. Программа численного решения системы уравнений, моделирующей процессы и условия генерации интенсивных атмосферных вихрей (ИАВ) [62] в трехмерной сжимаемой атмосфере, исходя из теории мезомасштабных вихрей В.Н. Николаевского.

2. Программа быстрого преобразования Фурье на 3D сетке из набора NAS Parallel Benchmarks (NPB) [63, 64].

Интерпретация этих примеров на инструментальной машине стало возможным благодаря разработанной в работе модели и интерпретатора к ней. В частности, приведенная модель позволила интерпретировать модель параллельной программы, у которой объем памяти превышает доступную физическую память; удаление вычислительных инструкций, оценка времени выполнения и интерпретация крупных фрагментов (итерации, *простые* циклы) сократили время интерпретации и повысили точность предсказаний; разработанная в работе модель потоков позволила интерпретировать многопроцессно-многопоточные программы.

Действительное ускорение обоих приложений были получены на кластере МСЦ РАН МВС-100К, на каждом узле которого два 4-ядерных процессора Intel® Xeon® CPU X5365 с частотой 3.00GHz и с интерфейсной платой HP Mezzanine Infiniband DDR. Количество узлов на кластере МВС-100К 1410. Использовалась 64-разрядная версия среды Java.

**Пример 1**. Прикладная параллельная программа численного решения системы уравнений, моделирующей процессы и условия генерации интенсивных атмосферных вихрей (ИАВ) в трехмерной сжимаемой атмосфере, исходя из теории мезомасштабных вихрей В.Н. Николаевского.

Математическая модель описывается нелинейной многокомпонентной трехмерной системой уравнений в частных производных смешанного типа [62]. Принимая гипотезу сухоадиабатической атмосферы [65], из общей теории мезомасштабной турбулентности [66] можно получить следующую систему уравнений движений воздуха, представляющих собой законы сохранения массы, импульса, а также их первых моментов:

$$\frac{da}{dt} = -(U_3 a_{0z} + Div) \qquad (1)$$

$$\frac{dU_i}{dt} = A_1(f(\Delta U_i + \frac{\partial Div}{\partial X_i}) + U_{ij}\phi_j) + A_2\varepsilon_{ijk}\hat{A}_j[f\omega_k] + \delta_{i3}g\cdot 0.4a(1+0.2a) - c^2\frac{\partial a}{\partial X_i}, \qquad (2)$$

$$\frac{dJ}{dt} = A_4(f\Delta J + \phi_j\frac{\partial J}{\partial X_j}), \qquad (3)$$

$$\frac{dF_i}{dt} = A_3(f\Delta F_i + \phi_j\frac{\partial F_i}{\partial X_j}) + \frac{f}{J}\left((A_3 + A_4)\frac{\partial F_i}{\partial X_j}\frac{\partial J}{\partial X_j} - 2A_2\omega_i\right) - \frac{g}{J}\varepsilon_{ij3}\hat{A}_j[J], \qquad (4)$$

где i, j = 1, 2, 3, ось z=$X_3$ направлена вертикально вверх; $U_i$ скорость ветра, $Div = \frac{\partial U_i}{\partial X_i}$, $\Delta = \frac{\partial^2}{\partial X_i^2}$; $a = \ln(\rho/\rho_0) - a_0(z)$ - возмущение логарифма плотности, $a_0(z) = \frac{5}{2}\ln(1-\gamma_a z)$, $\gamma_a = \frac{g}{C_p T_0}$ - сухоадиабатический градиент температуры, $C_p = \frac{7}{2}R_B \approx 1005$ Дж/(кг К) – удельная теплоемкость воздуха при постоянном давлении, $T_0$ =288.15 К, $\rho_0$ =1.23 кг/м3, - температура и плотность воздуха на поверхности земли в стандартной атмосфере, $g$ - ускорение свободного падения; $f = f(\omega) = \frac{\omega_{bk} + \omega}{\omega_0}$, $\omega_0$ - начальный модуль мезовихря, $\omega_{bk}$ - его фоновое значение; $A_j$ - модули коэффициентов турбулентной вязкости в начальный момент [67]; $J$ – момент инерции мезовихря, $F_i = \Omega_i + \omega_i$ - суммарная завихренность, $\Omega_k = \frac{1}{2}\varepsilon_{kij}\frac{\partial U_j}{\partial X_i}$ - вихрь поля скорости ветра, $\omega_k$ - мезозавихренность; $U_{ij} = 2e_{ij} = \frac{\partial U_i}{\partial X_j} + \frac{\partial U_j}{\partial X_i}$, $\hat{A}_j[B] \equiv \frac{\partial B}{\partial X_j} + B\left(\frac{\partial a}{\partial X_j} + \delta_{j3}a_{0z}\right)$, $\phi_j = \hat{A}_j[f]$, $c^2 = c_0^2\cdot(1+0.4a)$ - квадрат скорости звука, $c_0^2 = \frac{\partial P_a}{\partial \rho_a} = \frac{7}{5}R_B T_0\cdot(1-\gamma_a z)$ - квадрат скорости звука в начальный момент времени.

Аналитические решения математической модели неизвестны, а численное решение впервые было получено в работе [68].

Соответствующая параллельная программа была разработана и реализована в среде ParJava. В процессе разработки инструменты ParJava использовались для определения области масштабируемости и оптимального числа параллельных процессов, а также для настройки параллельной программы с целью ее оптимизации. Исследования показали, что при выбранной схеме вычислений доля последовательных вычислений f = 0,83%. Разработанная программа показала достаточно высокую производительность и хорошую масштабируемость.

Разработанная в диссертации модель позволила интерпретировать параллельную программу моделирования ИАВ, где объем памяти больше чем доступная физическая память (5GB). Также, благодаря приведенной модели (удаление вычислительных инструкций), оценке времени выполнения более крупных фрагментов (итерации цикла, *простые* циклы) повысилась точность прогноза и сократилась время интерпретации модели.

На Рисунке 25 приведен график зависимости ускорения программы моделирования ИАВ от количества используемых процессов для кластера МСЦ РАН.

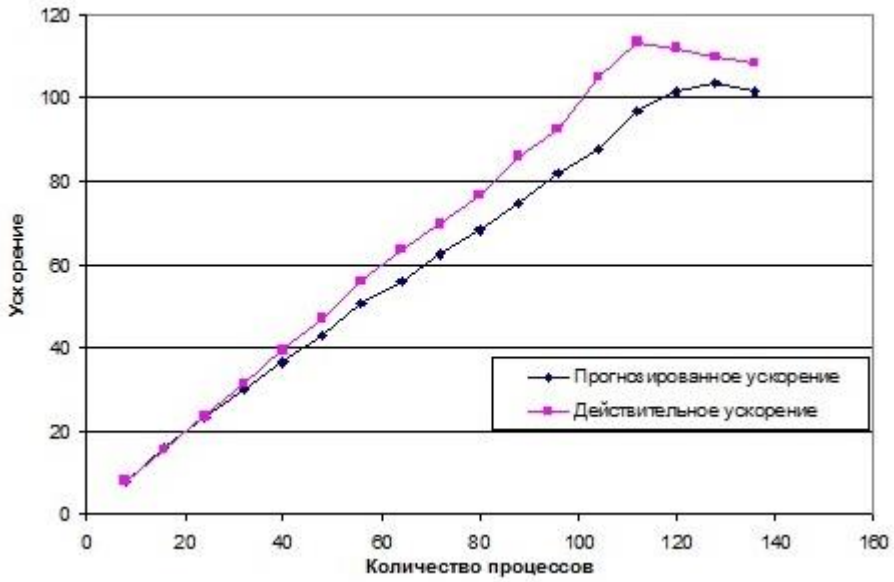


Рисунок 25. Ускорение программы моделирования ИАВ на кластере МСЦ РАН.

На Рисунке 25 «Действительное ускорение» представляет собой ускорение программы, измеренное при выполнении программы на вычислительной платформе, «Прогнозированное ускорение» - ускорение, предсказанное интерпретатором ParJava на инструментальной машине. Графики показывают хорошее совпадение предсказанных и реальных значений производительности программы. Объем памяти требуемой в задаче 5Gb. Как видно из графика, начиная со 120 процессов, реальное ускорение программы идет на спад.

Интерпретатор также фиксирует, что в данной точке производительность программы падает. Отметим, что такое падение производительности связано с асимптотикой формулы Амдаля ($\lim_{p\to\infty} S_A(p) = \frac{1}{f}$) [69]. В самом деле, 1/0,0083 ≈ 120. Процент погрешности предсказания составил в среднем 10%.

**Пример 2**. Рассмотрим параллельную программу быстрого преобразования Фурье FT из набора NAS Parallel Benchmarks. Первоначальная версия NPB на языке Java была разработана и реализована в университете Coruna [70].

В рамках данной работы, был реализован многопоточный вариант программы FT: коммуникации между процессами программы обеспечиваются функциями MPI, а для взаимодействия между потоками в каждом процессе используются функции из библиотеки времени выполнения mpiJava.threads. Проведена серия экспериментов по сравнению параллельной программы основанной только на библиотеке MPI и параллельной программы на основе MPI и потоков Java.

Приведем описание параллельной программы FT. В данной работе рассматривается 3D версия программы FT (расчетная область представляет собой трех мерную матрицу комплексных чисел). Пусть имеется последовательность $u=\{u_0, u_1,\dots u_{N-1}\}$. Дискретное преобразование Фурье (задача 1D FFT) преобразует последовательность u в другую последовательность U, где

$$U_k = \sum_{n=0}^{N-1} u_n e^{\frac{-2\pi i k n}{N}}$$

В реализованном алгоритме FT дискретное преобразование Фурье

применяется на 3D сетке размерностью LxMxN. В этом случае формула преобразование Фурье принимает следующий вид:

$$F_{q,r,s}(u) = \sum_{l=0}^{L-1}\sum_{k=0}^{M-1}\sum_{j=0}^{N-1} u_{j,k,l} e^{\frac{-2\pi ijq}{L}} e^{\frac{-2\pi ikr}{M}} e^{\frac{-2\pi ils}{N}}$$

Пусть исходные данные представляют собой куб $N^3$. В этом случае при применении 1D декомпозиция параллельная программа масштабируется O(N), а при 2D декомпозиция $O(N^2)$.

В приведенных ниже графиках исследуется масштабируемость по Амдалью. Размер рассчитываемой матрицы 512x512x256, объем памяти требуемой в задаче, примерно 5Gb, количество внешних итераций 20.

Разработанная в работе модель позволила интерпретировать параллельную программу БПФ на 3D сетке, где объем памяти больше чем доступная физическая память (5GB). Благодаря приведенной модели (удаление вычислительных инструкций), оценке времени выполнения более крупных фрагментов (итерации цикла, *простые* циклы) повысилась точность прогноза и сократилась время интерпретации модели. Разработанная в диссертации модель и интерпретатор к ней позволили интерпретировать не только многопроцессную параллельную программу БПФ, но и многопроцессно-многопоточную версию программы.

В приведенных ниже рисунках FT соответствует параллельному приложению БПФ на языке Java с применением интерфейса MPI. В данном случае рассматривается параллельная многопроцессная программа без применения многопоточности. FT запускается с применением 2D декомпозиции: для запуска с N(8,16,32,64) ядрами применяется декартовое разбиение пространства процессов [N1,N2], где N=N1*N2 и для запуска используется N1 узлов, на каждом узле N2 процессов.

FT_T представляет собой МПМП программу и является модификацией FT. В данном случае применяется 1D декомпозиция: для запуска с N(8,16,32,64) ядрами используется линейное пространство процессов [N1], где N1 количество узлов и на каждом узле в определенных точках программы запускаются TSize=N2 потоков (N=N1*N2).

На Рисунке 26 представлены графики зависимости времени выполнения параллельной программы БПФ от числа используемых ядер.

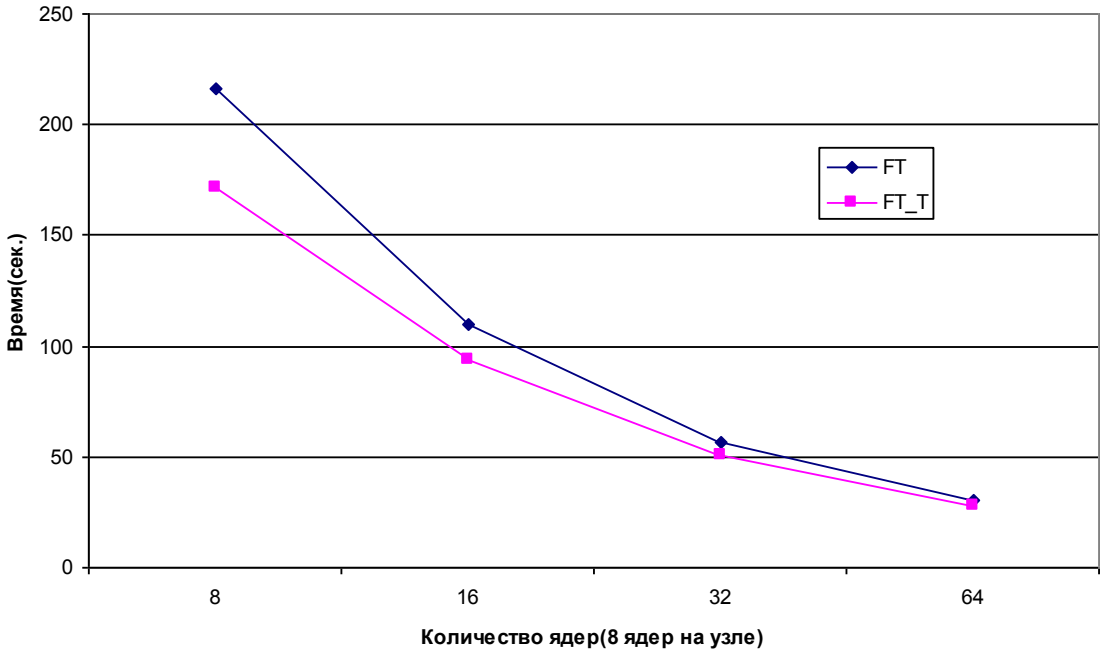


Рисунок 26. Сравнение времени выполнения многопроцессной программы БПФ с МПМП версией.

На каждом узле используются по восемь ядер: для программы FT N2=8, для программы FT_T TSize=8. МПМП программа FT_T выполняется быстрее, чем многопроцессная программа FT, на 9,5%-20%. В FT_T были распараллелены на потоки функции, связанные с локальным транспонированием и расчетами. Инициализация же данных в каждом процессе выполняется последовательно (используется один поток). То есть степень параллелизма в данном случае меньше, чем для программы FT. Если распараллелить на потоки и инициализационные функции, то время выполнения программы FT_T сократиться еще больше.

Нужно заметить, что с увеличением количества используемых ядер преимущество FT_T над FT уменьшается. Профилирование программы показало, что связано это с глобальным транспонированием по оси x, которое производится посредством коллективной коммуникационной функции AlltoAll между процессами программы.

На Рисунке 27 приведен график зависимости ускорения МПМП программы

(FT_T) от количества используемых ядер для кластера МСЦ РАН. На Рисунке 27 «Действительное ускорение» представляет собой ускорение программы FT_T, измеренное при выполнении программы на вычислительной платформе, «Прогнозированное ускорение» - ускорение, предсказанное интерпретатором ParJava на инструментальной машине. Графики показывают хорошее совпадение предсказанных и реальных значений производительности программы (3-7%).

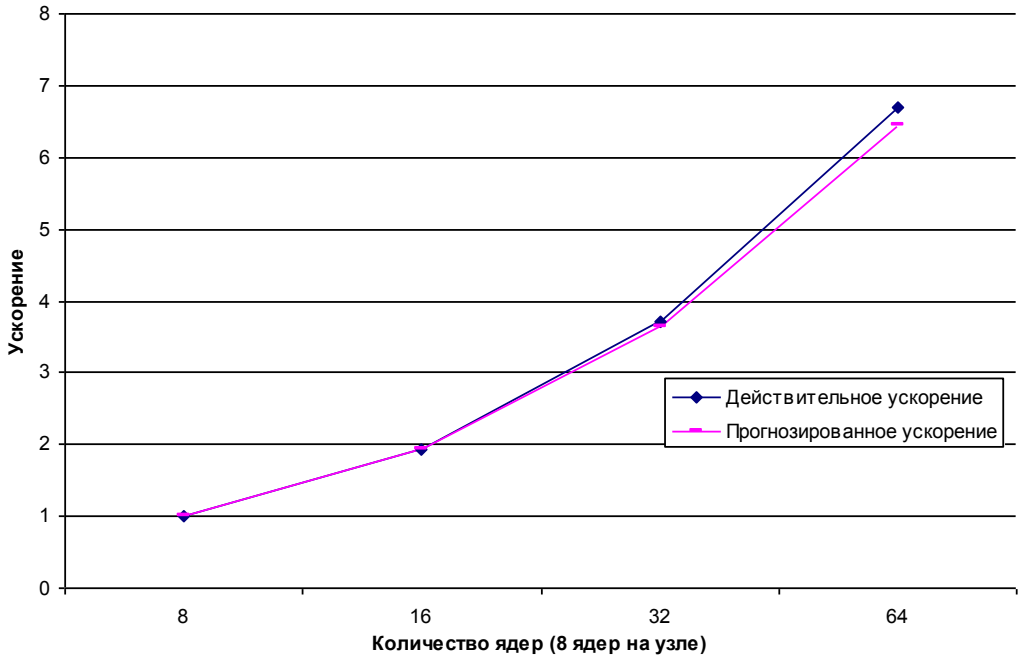


Рисунок 27. Ускорение МПМП программы БПФ на кластере МСЦ РАН.

На Рисунке 28 приведены графики времени выполнения функции транспонирования и ее составных частей для программы FT. На данном рисунке FT_1Loc и FT_1Fin относятся к подготовке транспонированию по оси x, FT_1Glo относится к транспонированию по оси x (глобальная коммуникация между узлами кластера), FT_2Loc и FT_2Fin относятся к подготовке транспонированию по оси z, FT_2Glo относится к транспонированию по оси z (глобальная коммуникация между процессами внутри узла кластера).

В начальных точках (8 ядер, 16 ядер) время выполнения коммуникаций в программе FT_T заметно меньше, чем для коммуникаций в FT, но с увеличением количества ядер разрыв уменьшается. Это связано с тем, что в FT_T количество обменивающихся процессов TSize раз меньше чем в FT, а размер пересылаемых сообщений больше. В случае с коллективной коммуникацией AlltoAll время

выполнения функции меньше, когда задействованы больше процессов. В этом случае задействованы больше сетевых карт, больше каналов связи и соответственно сокращаются простои по ожиданию тех или иных процессов.

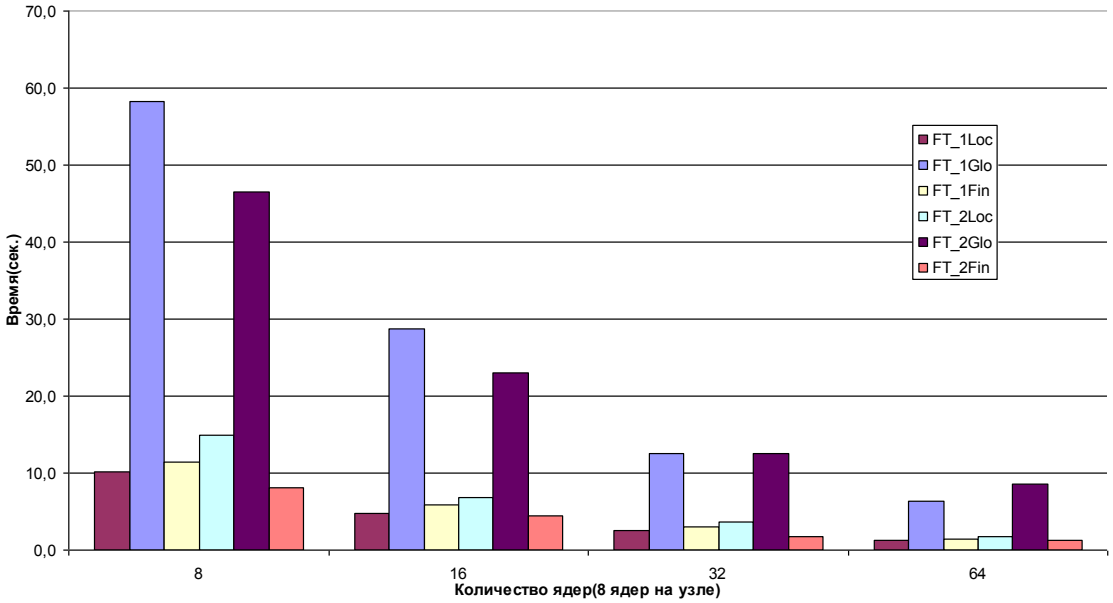


Рисунок 28. Временной профиль транспонирования в многопроцессной программе БПФ.

На Рисунке 29 приведены графики времени выполнения функции транспонирования и ее составных частей для программы FT_T.

На данном рисунке FT_T_1Loc и FT_T_1Fin относятся к транспонированию по оси z, FT_T_1Glo относится к транспонированию по оси x (глобальная коммуникация между узлами кластера). В программе FT_T отсутствует необходимость коммуникаций при транспонировании по оси z поскольку все потоки в рамках одного процесса имеют доступ к необходимым данным благодаря общей памяти. В этом случае, для повышения производительности (кэш попадания) производится лишь копирование в последовательный массив.

Из Рисунков 28, 29 видно, что основная причина деградации производительности программы FT_T это эффект связанный с графиком FT_T_1Glo. В программе FT производятся две коммуникации FT_1Glo и FT_2Glo. При увеличении количества ядер в FT_2Glo не меняется количество обменивающихся процессов, но сокращается размер сообщений. А в FT_1Glo увеличивается количество обменивающихся процессов, одновременно

уменьшается размер сообщений. Ситуация с FT_T_1Glo похожа на FT_1Glo, однако в данном случае размер сообщений больше и в функции FT_1Glo параллельно выполняются N2 AlltoAll коммуникаций.

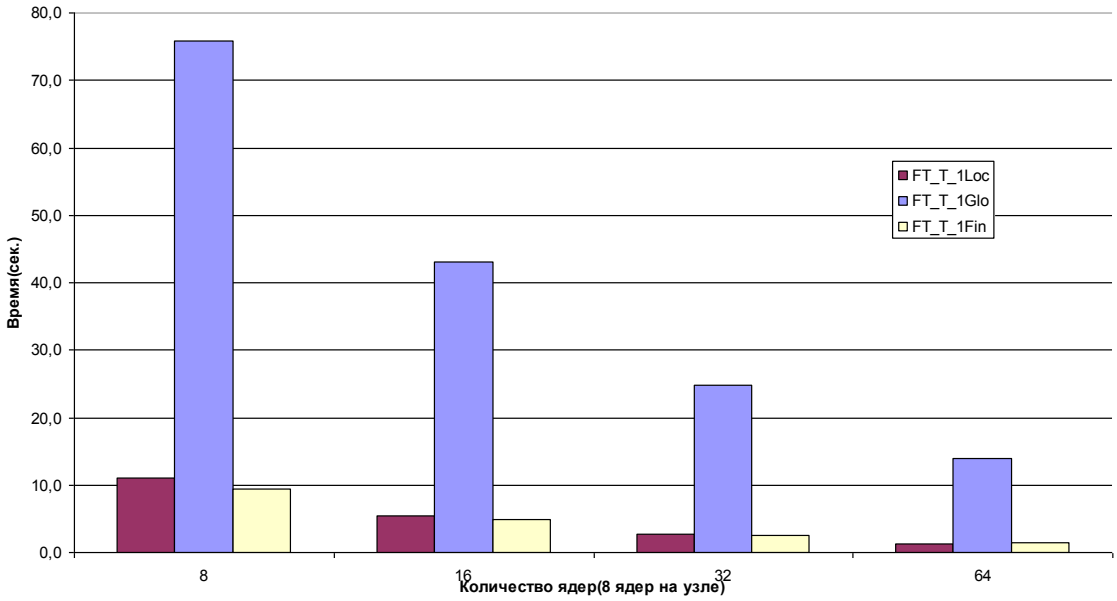


Рисунок 29. Временной профиль транспонирования в МПМП программе БПФ.

Увеличение количества используемых ядер приводит к сокращению времени выполнения параллельных участков программы, а последовательная часть программы остается почти таким же, что приводит к увеличению доли последовательной части. Поскольку степень параллелизма в FT_T меньше чем в FT, то ускорение FT_T меньше.

Таким образом, в текущей версии программ FT, FT_T при использовании от восьми до тридцати двух ядер МПМП программа FT_T работает в среднем на 10-20% быстрее, чем аналогичная многопроцессная программа FT. Но при дальнейшем увеличении количества ядер преимущество сокращается. В дальнейшем планируется замена схемы коммуникаций в МПМП с целью увеличения степени параллелизма, что приведет к ускорению программы.

## 7.3 Результаты экспериментов по обнаружению коммуникационных шаблонов MPI

Реализованный инструмент автоматизированного обнаружения коммуникационных шаблонов MPI был применен к прикладной параллельной программе расчета вязкого обтекания затупленной головной части (НПО машиностроение) [71]. Данная программа написана на языке C++ с использованием интерфейса MPI, объем параллельной программы 4500 строк. Поиск шаблонов проводился на реальной трассе параллельной программы, причем в трассу были включены только события, относящиеся к коммуникациям. Инструмент обнаружил в программе один шаблон типа «ранняя стандартная посылка (E_MPI_Send→MPI_Recv)», и четыре шаблона с использованием не-блокирующих функций - два шаблона «ожидание на стороне отправителя при использовании {MPI_Isend, MPI_Irecv}» и два шаблона «ожидание на стороне получателя при использовании {MPI_Isend, MPI_Irecv}». После исправления программы производительность улучшилась при оптимальном количестве процессов на 5 %, а в среднем на 3%.

Также разработанный инструмент был протестирован на параллельной программе моделирования ИАВ. Данная параллельная программа написана на языке Java с использованием интерфейса MPI. Объем программы составляет 5103 строки. При анализе данного приложения поиск шаблонов проводился на модельной трассе полученной интерпретатором ParJava на инструментальном компьютере. Как и в случае с анализом реальной трассы в модельную трассу были включены только события связанные с коммуникационными функциями. В программе было обнаружено два шаблона неэффективного использования не-блокирующих MPI-функций типа «ожидание на стороне отправителя при использовании {MPI_Isend, MPI_Irecv}» и один шаблон типа «близкая посылка, передача сообщений». После исправления программы производительность улучшилась на 1-2%.

# 8. Заключение

В работе получены следующие новые научные результаты:

1. Разработана интерпретируемая модель параллельной программы, позволяющая оценивать границы масштабируемости параллельных Java-программ для современных высокопроизводительных вычислительных систем с распределенной памятью, строящихся на основе многоядерных узлов (взаимодействие между процессами осуществляется посредством MPI, а внутри процесса используются Java-потоки).
2. Разработан и реализован метод интерпретации модели, обеспечивающий интерпретацию реальных параллельных приложений за приемлемое время, как на целевой вычислительной системе, так и на инструментальном компьютере. В том числе обеспечивается учет изменений, вносимых динамическим компилятором времени выполнения.
3. Разработан метод автоматизированного обнаружения коммуникационных шаблонов MPI (как на основе реальной, так и модельной трассы), приводящих к потере производительности.

Разработаны и реализованы следующие программные средства: построитель модели параллельной МПМП программы на языке Java, интерпретатор модели, модуль оценки времени выполнения фрагментов программы, модуль по удалению вычислительных инструкций модели (редукция), сборщик модельной трассы событий, модуль по выявлению коммуникационных шаблонов приводящих к потере производительности. Все программные средства интегрированы в среду ParJava, которая доступна как дополнение к среде разработки программ Eclipse.

Разработанные программные средства были применены при разработке следующих прикладных программ:

1. Программа численного решения системы уравнений, моделирующей процессы и условия генерации ИАВ в трехмерной сжимаемой атмосфере.

2. Программа быстрого преобразования Фурье на 3D сетке из набора NPB.

3. Программа расчета вязкого обтекания затупленной головной части. Поскольку программа написана на языке C++, то инструменты ParJava были применены к ней для выявления коммуникационных шаблонов.

Разработанная модель параллельной МПМП Java программы позволяет учитывать многоядерность современных процессоров. Приведение модели позволяет моделировать параллельные программы с существенно бóльшим объемом данных, чем доступная физическая память на инструментальном компьютере и при этом увеличивается скорость интерпретации модели за счет сокращения количества интерпретируемых инструкций. Точность предсказаний времени выполнения была улучшена за счет учета влияния JIT-компилятора и оценки времени выполнения более крупных фрагментов (итерации, *простые* циклы). Также разработчику прикладных программ предоставляется возможность по выявлению коммуникационных шаблонов MPI на основе как реальной, так и модельной трассы. Таким образом, разработанные программные средства позволяют минимизировать использование целевой вычислительной платформы, переведя большую часть разработки на инструментальный компьютер.

В прототипной реализации ParJava не поддерживается использование шаблонов языка Java. А библиотека времени выполнения поддерживает MPI на уровне 1.2.

В дальнейшем будет расширена библиотека времени выполнения до уровня 2.0, будет разработана новая модель программы, учитывающая особенности графических процессоров (GPU). Планируется реализация параллельной МПМП программы расчета вязкого обтекания затупленной головной части на языке Java и ее доводка производительности с использованием инструментальных средств среды ParJava.

## Список публикаций


1. М.С. Акопян. Интерпретация как средство исследования динамических свойств параллельной программы на инструментальном компьютере. // Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики, 2008, №54, с. 166-168
2. В.П. Иванников, А.И. Аветисян, С.С. Гайсарян, М.С. Акопян. Особенности реализации интерпретатора модели параллельных программ в среде ParJava. Журнал "Программирование". 2009, том 35, № 1, с. 10-25
3. А.И. Аветисян, М.С. Акопян, С.С. Гайсарян. Методы точного измерения времени выполнения гнезд циклов при анализе JavaMPI-программ в среде ParJava. Труды Института системного программирования РАН, том 21, 2011, с. 83-102
4. М.С. Акопян. Расширение модели ParJava для случая кластеров с многоядерными узлами. Труды Института системного программирования РАН, том 23, 2012, с.13-30
5. М.С. Акопян, Н.Е. Андреев. Исследование и разработка шаблонов неэффективного поведения в параллельных MPI, UPC приложениях. Труды Института системного программирования РАН, том 24, 2013, с. 49-70
6. М.С. Акопян. Использование многопоточных процессов в среде ParJava. Труды Института системного программирования РАН, том 27, выпуск 2, 2015, с. 5-22
7. Manuk Akopyan “Performance Penalty Detection in MPI Applications” // 10th International Conference on Computer Science and Information Technologies (CSIT 2015), September 28- October 02, Yerevan, Armenia

## Список литературы


8. J. Labarta et al., "DiP: A Parallel Program Development Environment," Proc. Euro-Par "96, Vol. II, Springer-Verlag, 1996, pp. 665-674.
9. Sameer S. Shende Allen D. Malony, “The Tau Parallel Performance System”, International Journal of High Performance Computing Applications, Volume 20 , Issue 2, Pages: 287 – 311, May 2006
10. L. Carrington, N. Wolter, and A. Snavely, “A Framework for Application Performance Prediction to Enable Scalability Understanding”, Scaling to New Heights Workshop, Pittsburgh, May 2002
11. Laura Carrington, Allan Snavely, Nicole Wolter: A performance prediction framework for scientific applications. Future Generation Comp. Syst. 22(3): 336-346 (2006)
12. Клинов М.С., Крюков В.А. Прогнозирование характеристик параллельного выполнения DVM-программ // Труды Всерос. науч. конф. «Научный сервис в сети Интернет: технологии параллельного программирования», Новороссийск, 18–23 сентября 2006. М.: Изд-во МГУ, 2006. С. 113–114.
13. Vikram S. Adve, Rajive Bagrodia, James C. Browne, Ewa Deelman, Aditya Dube, Elias Houstis, John Rice, Rizos Sakellariou, David Sundaram-Stukel, Patricia J. Teller and Mary K. Vernon, “POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems” , IEEE Transactions on Software Engineering, Vol. 26, No. 11, P. 1027-1048, NOVEMBER 2000
14. Hammond S.D., Mudalige G.R., Smith J.A., Jarvis S.A.: ‘WARPP – a toolkit for simulating high-performance parallel scientific codes’. Second Int. Conf. Simulation Tools and Techniques (SIMUTools09), Rome, Italy, March 2009
15. S.D. Hammond, G.R. Mudalige, J.A. Smith, A.B. Mills, S.A. Jarvis, J. Holt, I. Miller, J.A. Herdman and A. Vadgama, Performance Prediction and Procurement in Practice: Assessing the Suitability of Commodity Cluster Components for Wavefront Codes, IET Software, 2009

16. B. Buck and J. K. Hollingsworth, “An API for Runtime Code Patching,” The International Journal of High Performance Computing Applications, 14(4), pp. 317–329, Winter 2000.
17. Sun Microsystems, “The JVM Tool Interface Version” http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/
18. W. Denzel, J. Li, P. Walker, and Y. Jin. A framework for end-to-end simulation of high-performance computing systems, booktitle = Simutools ’08: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops. 2008.
19. Cyriel Minkenberg and Germán Rodriguez, “Trace-driven Co-simulation of High-Performance Computing Systems using OMNeT++”, Proceedings of the 2nd International Conference on Simulation Tools and Techniques, a. 65,Rome, March 2009
20. Paraver http://www.bsc.es/plantillaA.php?cat_id=488
21. R. Bell, A. D. Malony and S. Shende, “ParaProf: A Portable, Extensible and Scalable Tool for Parallel Performance ProfileAnalysis,” Proc. Europar 2003 Conference, LNCS 2790, Springer, Berlin, pp. 17–26, 2003.
22. A. Knupfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. M¨uller and W.E. Nagel, “The Vampir Performance Analysis Tool-Set”, in: Tools for High Performance Computing, pp 139-155, Springer Verlag, 2008
23. М.С. Клинов. Прогнозирование характеристик эффективности выполнения DVM-программ на кластере. Вестник Нижегородского университета им. Н.И. Лобачевского, 2009, № 4, с. 190–197
24. Sundeep Prakash and Rajive Bagrodia. MPI-Sim: Using Parallel Simulation to Evaluate MPI Programs. // Proceedings of the Winter Simulation Conference, 1998, pp. 467-474.
25. Thilo Kielmann, Sergei Gorlatch. Bandwidth-Latency Models (BSP, LogP). Springer US, 2011, pp 107-112

26. S.J. Pennycook, G.R. Mudalige, S.D. Hammond and S.A. Jarvis, Parallelising Wavefront Applications on General-Purpose GPU Devices, UK Performance Engineering Workshop 2010 (UKPEW 2010), July, 2010, Warwick, UK
27. Felix Wolf, Felix Freitag, Bernd Mohr, Shirley Moore, Brian J. N. Wylie: Large Event Traces in Parallel Performance Analysis. In Proc. 8th Workshop on Parallel Systems and Algorithms (PASA), Frankfurt, Germany, volume P-81 of Lecture Notes in Informatics, pages 264-273, Gesellschaft für Informatik, March 2006.
28. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. Revised version, August 1999
29. Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: Towards Thread Safe Java HPC, Submitted to the IEEE International Conference on Cluster Computing (Cluster 2006), Barcelona, Spain, 25-28 September, 2006.
30. S. Mintchev and V. Getov, Towards Portable Message Passing in Java: Binding MPI, Springer-Verlag, Proceedings of the 4th European PVM/MPI Users. Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 135—142, 1997, ISBN 3-540-63697-8.
31. Markus Bornemann , Rob V. Van Nieuwpoort , Thilo Kielmann. MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java. Euro PVM/MPI 2005, volume 3666
32. MPICH. [Электронный ресурс] — Электрон. дан. — США, [2012-?] Режим доступа: http://www.mpich.org. — Загл. с экрана. — Англ.
33. OpenMPI. [Электронный ресурс] — Электрон. дан. — США, [2012-?] Режим доступа: http://www.open-mpi.org. — Загл. с экрана. — Англ.
34. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI – The complete Reference, Volume 1, The MPI Core, Second edition. / The MIT Press. 1998
35. JavaConcurrent. [Электронный ресурс] — Электрон. дан. — США, [1995-2015] Режим доступа:

http://docs.oracle.com/javase/tutorial/essential/concurrency/. — Загл. с экрана. — Англ.

36. В.А. Падарян. Исследование и разработка методов оценки масштабируемости и производительности программ, параллельных по данным. [Текст] : дис. канд. физ.-мат. наук : 05.13.11. Москва, 2005.
37. James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, Alex Buckley. The Java Language Specification, Java SE 8 Edition (Java Series). Addison-Wesley Professional; 1 edition. May 16, 2014
38. John Aycock, Nigel Horspool. Simple Generation of Static Single-Assignment Form. 9th International Conference, CC 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000, pp. 110-125, Berlin, Germany, March
39. Marcelo Cintra, Guido Araujo. Array Reference Allocation Using SSAForm and Live Range Growth. // ACM SIGPLAN Workshop LCTES 2000, pp 48-62, Vancouver, Canada, June 18
40. В.П. Иванников, А.И. Аветисян, С.С. Гайсарян, В.А. Падарян. Прогнозирование производительности MPI-программ на основе моделей. // «Автоматика и телемеханика», 2007, №5, с. 8-17
41. Sharir, Micha. Structural Alalysis: A new approachto Flow Analysis in Optimizing Compilers, Computer Languages, Vol. 5, Nos. ¾, 1980, pp. 141-153
42. D. E. Knuth, "An empirical study of FORTRAN programs," Software Practice and Experience, vol. 1, 1971, pp. 105-133.
43. A. Matthew, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A Survey of Adaptive Optimization in Virtual Machines" // Proceedings of the IEEE, 93(2), 2005. Special issue on program generation, optimization, and adaptation
44. K.D. Cooper, T.J. Harvey, and K. Kennedy, "A Simple, Fast Dominator Algorithm," Rice CS TR 06-38870, January 2006.
45. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems, 1(1):115–120, July 1979.

46. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 13(4):451–490, Oct. 1991.
47. Steven Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1 edition, 1997, p. 856
48. Alpern A.B, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K.S. Mckinley, M. Mergen, J.E.B. Moss, T. Ngo, V. Sarkar. The Jikes Research Virtual Machine project: IBM Systems Journal, Vol. 44, No 2, 2005
49. В.П. Иванников, А.И. Аветисян, С.С. Гайсарян, В.А. Падарян. Оценка динамических характеристик параллельной программы на модели. «Программирование» 2006, №4, с. 21–37
50. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jaffrey D. Ullman. Compilers: principles, techniques, and tools. – 2nd ed., Pearson Education Inc., 2007, p. 1000
51. M. Paleczny, C. Vick, and C. Click. The Java HotSpotTM server compiler. In Proceedings of the Java Virtual Machine Research and Technology Symposium, pages 1–12, 2001
52. MPI profiling interface. [Электронный ресурс] — Электрон. дан. — США, [2012-?] Режим доступа: http://www.mpi-forum.org/docs/mpi-11-html/node152.html. — Загл. с экрана. — Англ.
53. MVAPICH. [Электронный ресурс] — Электрон. дан. — США, [2007-?] Режим доступа: http://mvapich.cse.ohio-state.edu. — Загл. с экрана. — Англ.
54. Infiniband. [Электронный ресурс] — Электрон. дан. — США, [2013-?] Режим доступа: http://www.infinibandta.org.— Загл. с экрана. — Англ.
55. Ed Burnette. Eclipse IDE Pocket Guide. O'Reilly Media. August 2005
56. Java tree builder home page. [Электронный ресурс] — Электрон. дан. — США, [1985-?] Режим доступа: . http://compilers.cs.ucla.edu/jtb/. — Загл. с экрана. — Англ.

57. JavaCC Grammar Files. [Электронный ресурс] — Электрон. дан. — США, [1997-?] Режим доступа: . https://javacc.java.net/doc/javaccgrm.html. — Загл. с экрана. — Англ.

58. TORQUE Resource Manager. [Электронный ресурс] — Электрон. дан. — США, [2002-?] Режим доступа: . http://www.adaptivecomputing.com/products/open-source/torque/. — Загл. с экрана. — Англ.

59. Rui Zhang, Zoran Budimlic, Ken Kennedy. Performance Modeling and Prediction for Scientific Java Applications.// IEEE International Symposium on Performance Analysis of Systems and Software, March 2006, pp. 199-210

60. Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. Java Concurrency In Practice. Addison Wesley Professional, May 19, 2006, p. 384

61. Brian Goetz. Java theory and practice: Garbage collection in the HotSpot JVM. IBM, 2003

62. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga. THE NAS PARALLEL BENCHMARKS. THE NAS PARALLEL BENCHMARKS. RNR Technical Report RNR-94-007, March 1994

63. H. Jagode, “Fourier Transforms for the BlueGene/L Communications Network”, Master’s thesis, University of Edinburgh, 2006.

64. Аветисян А.И., Бабкова В., Гайсарян С.С., Губарь А.Ю.. Рождение торнадо в теории мезомасштабной турбулентности по Николаевскому. Трехмерная численная модель в ParJava. Журнал «Математическое моделирование», том 20, №8, с. 28-40, 2008

65. Хргиан А.Х. Физика атмосферы. М: Изд-во Московского Университета, 1986. С.240.

66. Nikolaevskiy V.N. Angular Momentum in Geophysical Turbulence: Continuum. Spatial Averaging Method . Dordrecht: Kluwer (Springer). 2003. P. 245.

67. Арсеньев С.А., Губарь А.Ю., Николаевский В.Н. Самоорганизация торнадо и ураганов в атмосферных течениях с мезомасштабными вихрями. // ДАН, 2004, т.396, № 4, с.541-546.
68. Аветисян А.И., Бабкова В.В., Губарь А.Ю. «Моделирование интенсивных атмосферных вихрей в среде ParJava.» Всероссийская научная конференция «Научный сервис в сети Интернет: технологии параллельного программирования», г. Новороссийск, 2006. с. 109-112.
69. Amdahl G.M. Validity of single-processor approach to achieving large-scale computing capability, Proceedings of AFIPS Conference, Reston, VA. 1967. pp. 483-485
70. Dami´an A. Mall´on, Guillermo L. Taboada, Juan Touri˜no, and Ram´on Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. // Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09). Weimar, Germany, Feb 2009, pp. 181-190.
71. Максимов Ф.А., Чураков.Д.А., Шевелев Ю.Д. Д.А. Разработка математических моделей и численных методов для решения задач аэродинамического проектирования на многопроцессорной вычислительной технике, ЖВМиМФ, том 51, N2, 2011. с. 303-328

# Приложение 1. Краткое описание mpiJava.threads

Согласно стандарту MPI, в параллельной программе каждый MPI процесс может представлять собой многопоточную программу. В каждом потоке могут вызываться функции MPI, однако сами потоки не адресуемы. Сообщение, отправленное процессу с идентификатором rank, может быть получено любым потоком, выполняющимся в данном процессе (параметр rank в коммуникационных функциях MPI относится к процессу). Примитивы работы с потоками в стандарте не определены и оставлены на усмотрение пользователя.

Пакет mpiJava был расширен методами, поддерживающими использование потоков согласно стандарту MPI:

- Реализованы на Java функции из стандарта MPI MPI_INIT_THREAD, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN
- Разработан пакет mpi.threads, реализующий функции взаимодействия между потоками, основанный на пакете java.util.concurrent [35].

Многопоточное программирование посредством низкоуровневого интерфейса java.lang.Thread не считается лучшим решением в достижении параллелизма из-за проблем с производительностью. Начиная с версии 1.5, в Java введен пакет java.util.concurrent, предоставляющий высокоуровневый интерфейс к потокам в Java. Разработанный пакет mpi.threads использует и расширяет инструменты, предоставляемые пакетом java.util.concurrent.

Однако пакет java.util.concurrent был разработан не для высокопроизводительных вычислений, поэтому в реализации mpi.threads некоторые методы оригинального пакета были закрыты, некоторые адаптированы для применения в рамках модели SPMD. Пакет mpi.threads **не является** реализацией OpenMP, но многие подходы из OpenMP используются при работе с пакетом: модель выполнения потоков, распараллеливаемые инструкции оформляются в виде пользовательских заданий, локальные в рамках задания переменные, разделяемые переменные, операция редукции при формировании

распараллеливаемого фрагмента, критические секции, атомарные конструкции, барьеры и.т.д.

**Модель выполнения потоков** в библиотеке mpi.threads аналогична **модели выполнения потоков** в OpenMP. На Рисунке 30 приведена модель выполнения МПМП Java–программы, где N – количество процессов программы (на каждом узле запускается один процесс), n – количество ядер на каждом узле.

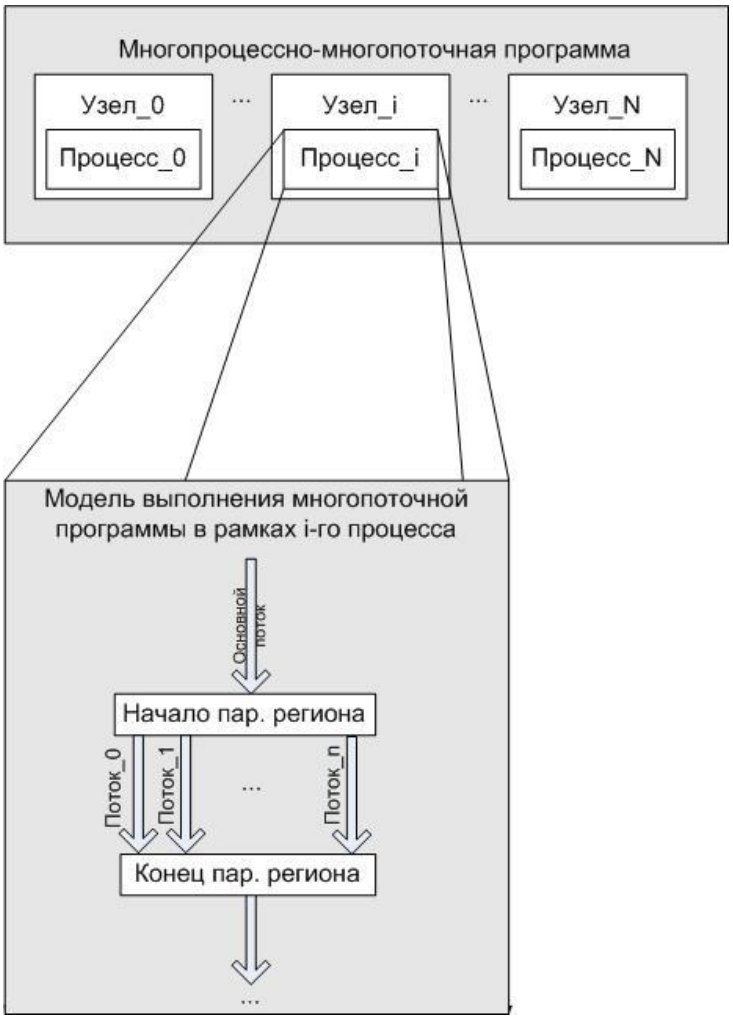


Рисунок 30. Модель выполнения МПМП Java–программы.

Рассмотрим поведение параллельной многопоточной программы в рамках одного MPI процесса в среде ParJava. В отличие от MPI и PGAS моделей, где изначально работают N вычислительных модулей (процессы в случае MPI и потоки в случае PGAS), при применении библиотеки mpi.threads в начале программы выполняется один поток (основной поток), который последовательно выполняет инструкции программы. В точках программы, где необходимо выполнить параллельно на нескольких потоках некий набор инструкций **Ins**

(параллельный регион), пользователь формирует задания, содержащие инструкции **Ins**, и передает ссылки на эти задания библиотеке времени выполнения. Полученные пользовательские задания выполняются каждый в отдельном потоке. Для обеспечения функциональности барьера после формирования заданий, пользователь должен явно вызвать соответствующую библиотечную функцию. В результате, основной поток блокируется и ждет завершения выполнения сформированных ранее заданий, выполняющихся в параллельных потоках. Синхронизация между потоками производится посредством критических секций, семафоров, атомарными операциями над переменными и.т.д.

**Замечание**. В библиотеке mpi.threads не поддерживается вложенные параллельные регионы.

В целях повышения производительности библиотека mpi.threads предоставляет возможность пользователю в рамках MPI-процесса создавать пул потоков. При создании пула создаются и инициализируются фиксированное количество рабочих потоков. После этого рабочие потоки блокируются в ожидании пользовательских заданий. Также в пуле создается очередь заданий. Распараллеливаемый фрагмент последовательной программы должен быть преобразован (заменен) в группу заданий. Во время выполнения программы, пользователь формирует новые задания и передает ссылки на них в очередь заданий, откуда задание передается первому свободному потоку, в котором и будет выполняться данное задание. Использование пула потоков позволяет избежать накладных расходов при создании и запуске новых потоков в ходе выполнения программы.

**Замечание**. В среде ParJava введено ограничение, согласно которому в каждом процессе МПМП программы используются одинаковое количество потоков.

**Замечание**. В среде ParJava введено ограничение, согласно которому в каждом процессе создается единственный пул потоков в последовательной части основного потока (main) и уничтожается в конце выполнения основного потока.

Пользовательское задание в ParJava представляет собой объект пользовательского класса, наследника от системного класса mpiJava.threads.PJTask. В пользовательском классе должен быть реализован метод run(), содержащий исходный код задания. Обычно метод run() содержит цикл (гнездо циклов) исходной программы, который нужно распараллелить между потоками. Параметры задания устанавливаются с помощью метода *setParams(…)*. При распараллеливании гнезда циклов пользователь должен распределить итерации цикла по заданиям с помощью параметров задания *setParams(loop_start,loop_end,…)* (ниже будет приведен пример распараллеливания цикла).

При работе с параллельными потоками используется модель памяти Java. Вся память, выделяемая в рамках задания, доступна потоку, который выполняет это задание. При формировании новых заданий память основного потока не наследуется автоматически. При необходимости использования переменных (или значений переменных) основного потока в задании, пользователь при формировании задания должен передать в метод *setParams(…)* формальную переменную по ссылке (или по значению). При передаче переменных по ссылке, они становятся разделяемыми (shared) переменными и доступ к таким переменным во избежание состязаний (race condition) нужно пользоваться семафорами, критическими секциями, если хотя бы один из обращений к данным переменным является операцией записи.

Функциональность операции *reduction* реализована посредством библиотечных функций setReductionOp(op), setReductionValue(init), getReductionValue(), getReducedValue(). Перед формированием группы заданий, пользователь задает операцию редукции с помощью метода setReductionOp(op). После этого, для каждого задания установливается начальное значение редукционной переменной. Обращение к редукционной переменной в исходном коде задания производится посредством методов getReductionValue() и setReductionValue(value). После завершения группы заданий (после метода

waitForAll()) основной поток может получить (вызов метода getReducedValue()) комбинированное значение редукционной переменной.

В пакете java.util.concurrent реализованы высокоуровневые классы и методы для работы с многопоточными программами, которые можно разбить на следующие группы:

1. Классы для работы с атомарными переменными. Классы из данного пакета предназначены для примитивных атомарных операцией с переменными соответствующего типа и обладают примерно одинаковым множеством методов (compareAndSet(…), getAndSet(…), getAndAdd(…)).

2. Классы для работы с блокировками (семафоры, блокировки). Классы из данной группы предоставляют методы для синхронизации (блокировки) потоков, когда в многопоточной программе производится обращения из разных потоков к общей памяти.

3. Классы для создания и управления потоками. Классы из данной группы предоставляют высокоуровневый интерфейс для создания и управления потоками. Также реализованы методы для работы с пулом потоков. Пул потоков предназначен для улучшения производительности при использовании большого количества асинхронных заданий за счет сокращения накладных расходов при выполнении задания.

4. Классы-коллекции для хранения объектов. Классы из данной группы представляют собой коллекции (по терминологии Java) для хранения объектов. Они обеспечивают безопасный интерфейс (thread-safe) для работы с коллекциями объектов при обращении к коллекции из нескольких потоков.

Все классы из первого пункта должны быть поддержаны в пакете **mpiJava.threads** в виде оболочек. На данный момент реализованы оболочки AtomicBoolean, AtomicInteger и AtomicLong.

Из второй группы в пакете **mpiJava.threads** реализовано:

1. Оболочка для класса **Semaphore**. Некоторые методы класса **Semaphore**, которые неприменимы при высокопроизводительных вычислениях,

были закрыты: методы для работы со временем ожидания (timeout), методы, реагирующие на прерывания из потока и.т.д.

2. Оболочка для класса **ReentrantLock**, делегирующего только функциональность критических секций. Остальные методы класса **ReentrantLock**, которые неприменимы при высокопроизводительных вычислениях, были закрыты: методы для работы со временем ожидания (timeout), методы, реагирующие на прерывания из потока и.т.д.

Из третей группы в пакете **mpiJava.threads** реализована оболочка для класса **ThreadPoolExecutor**. Методы класса **ThreadPoolExecutor**, которые неприменимы при высокопроизводительных вычислениях в рамках модели выполнения описанной в данном разделе, были закрыты:

- Методы для работы со временем ожидания (timeout),
- Методы, создающие пулы потоков с переменным количеством потоков,
- Методы, ограничивающие время жизни потока в пуле потоков
- Методы для работы с отверженными пользовательскими заданиями
- Методы, которые при вызове выполняют привилегированное действие (осуществляется проверка прав доступа данного действия к критическим ресурсам системы).

Методы для создания пула потоков и добавления новых заданий были изменены в пакете **mpiJava.threads** для повышения производительности параллельной многопоточной программы. Также были добавлены методы, обеспечивающие функциональность барьера и методы, обеспечивающие операцию редукции.

**Пример**. Приведем пример использования потоков посредством библиотеки mpi.threads при распараллеливании гнезда циклов. Пусть имеется гнездо циклов с независимыми итерациями, которое нужно выполнить параллельно в нескольких потоках. На Рисунке 31 приведен псевдо код такого цикла. Вначале массивы A и B инициализируются, после чего начинается цикл, где вычисляются значения массива B, используя массив A.

```
//инициализация массива A
//инициализация массива B
for(i=1; i <N; ++i)
{
  //тело цикла
  A[i] = B[i] + …;
}
//печать массива A
```

Рисунок 31. Последовательная версия исходного кода.

На Рисунке 32 приведен пример исходного кода задания. Пусть количество потоков в пуле равно M. Итерации цикла распределяются равномерно по заданиям. В этом случае пространство итераций цикла [1,N] разбивается на M равных частей и формируется группа из M заданий (задание представляет собой группу итераций основного цикла, которые будут выполняться в одном потоке).

```
class LoopTask extends PJTask{
private int loop_start,loop_end;
private double[] refA;
private double[] refB;
public void setParams(int loopStart, int loopEnd,double[] A,double[] B)
{
   loop_start = loopStart;
   loop_end = loopEnd;
   refA = A;
   refB = B;
}
public void run()
{
  for(i= loop_start; i < loop_end; ++i)
  {
    //тело цикла
    A[i] = B[i] + …;
  }
}
}
```

Рисунок 32. Исходный код задания.

Задания распределяются между свободными потоками из пула, после чего основной поток блокируется и ожидает окончания выполнения M заданий. На Рисунке 33 приведен фрагмент исходного кода основного потока многопоточной программы.

```
//создание пула потоков (tPool) с M потоками.
…
LoopTask[] loopTask;
//инициализация массива A
…
//инициализация массива B
…
int remaining = N%M;
int chunk = N/M;
int start = 1;
int end = start+(chunk-1) + (remaining >0?1:0);
for(int j = 0; j < M; ++j)
{
  loopTask[j].setParams(start,end,A,B);
  tPool.setTask(loopTask[j]);
  start = end + 1;
  end = start + (chunk-1) + (--remaining >0?1:0);
}
//Основной поток блокируется в ожидании выполнения заданий
tPool.waitForAll();
//печать массива A
…
```

Рисунок 33. Исходный код распараллеленной программы.

Как уже отмечалось, в данном примере итерации цикла независимы. Если же распараллеливаемый фрагмент содержит общие данные, то необходимо воспользоваться синхронизационными функциями (семафоры, барьеры и.т.д.).