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

Автореферат
диссертации на соискание ученой степени кандидата
физико-математических наук

Москва – 2016

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования Российской академии наук.

Научный руководитель:

Гайсарян Сергей Суренович, кандидат физико-математических наук, доцент, заведующий отделом компиляторных технологий

Официальные оппоненты:

Лацис Алексей Оттович, доктор физико-математических наук, заведующий сектором Федерального государственного бюджетного учреждения науки Институт прикладной математики им. М.В. Келдыша Российской академии наук

Биктимиров Марат Рамилевич, кандидат технических наук, ВРИО директора Федерального государственного бюджетного учреждения науки Всероссийский институт научной и технической информации Российской академии наук (ВИНИТИ РАН)

Ведущая организация:

Межведомственный суперкомпьютерный центр Российской академии наук - филиал Федерального государственного учреждения «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук»

Защита состоится “17” _марта_2016 г. в 16 часов на заседании диссертационного совета Д 002.087.01 при Институте системного программирования РАН по адресу: 109004, Москва, ул. А. Солженицына, д.25

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Институт системного программирования Российской академии наук.

Автореферат разослан “__”_____ 2016 г.

Ученый секретарь
диссертационного совета Д 002.087.01,
кандидат физико-математических наук Зеленов С. В.

Общая характеристика работы

Актуальность работы.

Современные высокопроизводительные вычислительные системы строятся по так называемой кластерной архитектуре (все системы из списка Top500). Узлами таких систем являются серверы, состоящие из нескольких процессоров, имеющих многоядерную архитектуру.

Практическое использование возможностей современной аппаратуры для организации высокоэффективных вычислений требует решения сложных оптимизационных проблем на всех уровнях параллельного выполнения вычислений. В настоящее время задача автоматического распараллеливания программ успешно решается лишь для узкого класса программ, допускающих параллельное выполнение без синхронизации. 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*). 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 и затраченными ресурсами будем называть продуктивностью. Таким образом, продуктивность достигается минимизацией затрачиваемых ресурсов при достижении необходимого

качества, в частности, посредством автоматизации процесса разработки параллельных программ.

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 среды 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, 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.

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.

Целю диссертационной работы является 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.

Краткое содержание работы

В разделе один - Введение обоснованы цели и задачи работы, ее актуальность, научная новизна и практическая значимость. Приводится краткий обзор работы.

Во втором разделе приводится обзор существующих инструментальных сред, поддерживающих разработку параллельных программ. В таких средах широко используется модель разрабатываемой процедуры. Использование модели позволяет ускорить интерпретацию параллельного приложения, так как в модели рассматриваются только те структуры, которые влияют на исследуемые характеристики параллельного приложения (например, время выполнения, границу

области масштабируемости). Известны два подхода к моделированию параллельных приложений: модели, основанные на симуляции трасс и модели, основанные на симуляции дискретных событий. Первый подход используется в средах (системах) разработки параллельных приложений *DiP, TAU, PERC, DVM*. Второй подход применяется в средах (системах) *POEMS и WARPP*.

Инструментальные среды, использующие модели, основанные на симуляции трасс, работают 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. 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. Анализируя проблемные фрагменты, прикладной программист может внести изменения в разрабатываемую программу, после чего повторить описанный процесс для выявления новых проблемных фрагментов.

Следует отметить отечественную систему DVM. DVM-система предназначена для создания переносимых оптимизированных вычислительных приложений на языках C-DVM и Fortran-DVM для многопроцессорных компьютеров с общей и распределенной памятью. В состав системы входят множество инструментов анализа и отладки параллельных приложений (компиляторы, отладчик, предсказатель производительности, трассировщик и.т.д.).

Важной особенностью DVM является то, что система поддержки выполнения DVM-программы позволяет по трассам выполнения программы на m процессорах определять характеристики выполнения программы при использовании n

процессоров. В начале исследуемая DVM-программа выполняется на инструментальном компьютере на некотором количестве процессоров с целью сбора трассы. По трассе DVM-программы и заданным пользователем параметрам целевой вычислительной системы интерпретатор трассы вычисляет временные характеристики выполнения данной программы на целевой вычислительной системе. Выполнение программы рассматривается не с точностью до выполнения операторов, а в виде последовательности работы более крупных фрагментов программы. Участки программы между вызовами функций системы поддержки выполнения считаются терминальными и характеризуются только временем их выполнения. Для оценки времен выполнения фрагментов программы 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 их пересчет для целевой системы.

Инструментальные среды, использующие модели, основаны на симуляции дискретных событий и используют 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. С целью повышения продуктивности разработки параллельных программ, все фазы кроме второй можно выполнить на инструментальной машине. Отладочный запуск проводится на целевой вычислительной платформе, что обеспечивает более точные результаты прогнозирования.

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.

В третьем разделе определена новая интерпретируемая модель параллельной многопоточной Java-программы для вычислительных систем с распределенной памятью. Взаимодействие между процессами осуществляется посредством коммуникационной библиотеки mpiJava.mpi, которая представляет собой реализацию MPI на языке Java. Внутри процесса могут использоваться Java потоки, для чего была разработана библиотека времени выполнения mpiJava.threads.

В разделе 3.1 приводится формальное определение новой модели, которая призвана решить следующие задачи:

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

Новая модель была построена на базе существующей модели Java-программы 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 τ: 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 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.

Листовой узел модели, соответствующий базовому блоку B, описывается парой объектов $B = \langle D, Bd \rangle$, где D – дескриптор базового блока, а Bd – тело базового блока B. *Дескриптор базового блока* B представляет собой кортеж $D = \langle id, \tau, ref(Bd), I, O, C, Time, Freq, Targets \rangle$, где $ref(Bd)$ – ссылка на тело Bd базового блока B, $Time$ – время выполнения блока B, $Freq$ – частота выполнения блока B, $Targets$ – список целевых атрибутов интерпретации.

Тело базового блока B типа cb – это линейная последовательность вычислений, выполняемых в блоке B, которые могут быть представлены либо инструкциями байт-кода, либо выражениями исходного языка, либо инструкциями любого другого внутреннего представления ассемблерного уровня.

Базовому блоку типа cfc соответствует либо функция MPI, либо функция для работы с Java потоками (библиотека mpiJava.threads).

Каждый элемент списков I, O и C представляет собой пару $\langle name, id \rangle$, где $name$ – имя переменной, id – идентификатор базового блока, содержащего определение переменной name, достигающее рассматриваемого базового блока. Если базовый блок достигается несколькими определениями одной и той же переменной $name$, то для объединения этих определений используется новое определение переменной name с помощью ϕ-функции.

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. $Time$ заменяется на его значение $TimeVal$, которое измеряется при построении приведенной модели.

Приведенной моделью базового блока B с дескриптором $D = \langle id, \tau, ref(Bd), I, O, C, Time, Freq, Targets\rangle$ одного из типов *ufc*, *efc*, или *cfc* для метода F называется блок с дескриптором $D_F = \langle id, \tau, ref(Bd_{F(cxt)}), I_C, O_C, C, TimeVal, Freq, Targets\rangle$, 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.

Введение последних двух определений позволяет корректно удалять вычислительные инструкции, тем самым обеспечивая возможность интерпретации прикладных параллельных приложений с большим объемом данных на инструментальном компьютере.

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.

Как уже отмечалось, в новой модели взаимодействие между процессами осуществляется посредством MPI, а внутри процесса могут использоваться Java потоки. Для этого в среде ParJava была введена единая система моделирования коммуникаций. **В разделе 3.2** описывается, как моделируются MPI-процессы и пользовательские потоки, а также примитивы, обеспечивающие моделирование функций для работы с потоками в среде ParJava.

Каждый MPI-процесс моделируемой программы представляется в ее модели с помощью логического процесса. Логический процесс определен как последовательность действий. 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. Продолжительность каждого действия, а также значения исследуемых динамических параметров базовых блоков, измеряются заранее на целевой платформе. 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 (pr_SetTask), 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 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 среде ParJava библиотека времени выполнения mpiJava.threads обеспечивает взаимодействие между потоками. Для моделирования поведения функций из этой библиотеки был выбран следующий набор 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.

Приведенный набор базовых примитивов позволяет моделировать поведение всех функций взаимодействия между потоками из библиотеки mpiJava.threads.

Отметим, что, для улучшения точности прогнозирования времени выполнения программы, необходимо учитывать оптимизации, вносимые динамическим компилятором (JIT). **В разделе 3.3** приводится схема построения модели параллельной программы. Для каждого метода программы строится граф потока управления и дерево управления. Над графом потока управления выполняются различные виды статического анализа потоков данных (в частности, вычисляются достигающие определения, строятся Du-цепочки и т.п.).

Байт-код параллельной Java-программы выполняется в инструментальном режиме на JavaVM с отключенным JIT-компилятором. 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. Предлагаемый метод построения модели обеспечивает соответствие модели байт-коду, автоматически учитывая все оптимизации, выполняемые компилятором с языка *Java*.

Для учета влияния JIT компилятора, 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.

Такое построение модели позволяет учитывать изменения, вносимые JIT-компилятором и уменьшить погрешность предсказания.

В разделе 4 рассматривается разработанный в рамках работы интерпретатор модели. Описывается интерпретация коммуникационных функций (как MPI, так и для работы с Java потоками). Приводится механизм, обеспечивающий возможность интерпретации реальных прикладных параллельных приложений, объем памяти которых превышает доступную физическую память. Описываются методы оценки базовых блоков и более крупных фрагментов, последующая редукция которых позволяет сократить время интерпретации. Также в разделе приводятся оценки погрешности прогнозирования времени выполнения сверху.

Цель интерпретации – получение оценок целевых атрибутов, входящих в список T, для каждой интерпретируемой единицы – базового блока, внутреннего узла дерева управления, функции. Список динамических атрибутов T каждого интерпретируемого узла всегда содержит атрибут Time – время выполнения поддерева с корнем в этом узле.

В разделе 4.1 приводится описание интерпретации коммуникационных функций. Для моделирования коммуникационных функций выбирается набор базовых примитивов, с помощью которых можно описать все коммуникационные функции. Время выполнения базовых примитивов моделирования функций MPI 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.

Все функции из пакета mpiJava.threads моделируются базовыми примитивами, приведенными в разделе 3.2.2. Следовательно, для оценки времени выполнения функций взаимодействия между потоками достаточно оценить время выполнения базовых примитивов. Длительность базовых примитивов 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 потока и родительского 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

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 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. В ParJava разработаны программы на уровне библиотечных функций Sun JDK1.6 с учетом особенностей

библиотеки mpiJava.threads, определяющие соответствующие параметры.

В разделе 4.2 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. С точки зрения прогнозирования времени выполнения, результаты, которые вычисляются 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.

Приведение модели позволяет достичь следующих результатов: 1) появляется возможность интерпретировать прикладные приложения с существенно бóльшим объемом данных (поскольку из модели удалены большие матрицы), 2) увеличивается скорость интерпретации модели за счет сокращения количества интерпретируемых инструкций.

Раздел 4.3 описывает оценку 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. Время выполнения базовых блоков определяется с помощью аппаратного счётчика TSC (Time Stamp Counter). Обращение к счетчику производится через ассемблерную вставку с использованием интерфейса JNI (ассемблерные вставки реализованы для процессоров Intel 32, Intel 64 и PowerPC). В ParJava также оценивается время выполнения более крупных фрагментов программы (отдельных итераций циклов, циклов в целом, а в некоторых случаях – и гнезд циклов). Например, для сбалансированного гнезда циклов анализ выполняется, начиная с самых внутренних («листовых») циклов. Для каждого такого цикла измеряется время его выполнения, после чего указанный цикл редуцируется. Поэтому на каждом этапе анализа рассматривается цикл, у которого нет вложенных циклов. В итоге, гнездо циклов заменяется на редуцированный базовый блок. Это приводит к существенному ускорению интерпретации модели и повышению точности прогноза за счет учета влияния JIT компилятора.

В разделе 4.4 приводится оценка погрешности прогнозирования. 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 $\overline{t_{bb}}$ согласно разделу 4.3. Пусть время выполнения базового блока измеряется N раз на узле вычислительной платформы. Пусть при каждом измерении t_{bb} отличается от измеренного машинного времени τ_{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:

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

где ε_i независимы и одинаково распределены. В этом случае среднее время

выполнения базового блока определяется как

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

Теорема 1. Абсолютная погрешность Δ_{bb} оценки среднего времени работы базового блока $\bar{t}_{bb}$ удовлетворяет неравенству $|\Delta_{bb}| \le \frac{3}{\sqrt{N}}|\sigma|$

Теорема 2. Относительная погрешность оценки среднего времени работы базового блока $\bar{t}_{bb}$ удовлетворяет неравенству $\delta_{bb} \le \frac{3|\sigma|}{\sqrt{N}(\bar{\tau}_{bb} - |\sigma|)}$

Поскольку программа представляет собой суперпозицию терминальных вершин, то абсолютная погрешность оценки времени работы программы $\Delta_P \le C_P \max_{bb \in P}(|\Delta_{bb}|)$, где Δ_{bb} – абсолютная погрешность времени работы базового блока, C_P – константна для фиксированной программы P.

В разделе 5 приводится описание метода автоматизированного обнаружения коммуникационных шаблонов MPI, приводящих к потере производительности. Причем для поиска можно использовать как реальную, так и модельную трассу. Разработанный метод основан на post-mortem анализе данных времени выполнения. Для автоматизированного обнаружения шаблонов необходимо вначале получить информацию времени выполнения о критических функциях, которые потенциально могут привести к потере производительности. После этого проводится анализ собранной информации на предмет наличия шаблонов.

В разделе 5.1 приводится описание метода, состоящего из следующих этапов:

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

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

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

Построение трассы параллельного приложения состоит из следующих этапов:

а) Инструментирование параллельной программы.

б) Выполнение инструментированной программы на целевой платформе.

Для получения данных времени выполнения (трасса) параллельного приложения применяется подход инструментирования на уровне компоновки библиотеки MPI с применением интерфейса PMPI.

На втором этапе анализатор трассы производит сканирование собранной трассы и поиск шаблонов (для каждого шаблона определены события и связанные с ними критерии).

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 коммуникационных шаблонов MPI в программах, написанных на языке Java (**раздел 5.2**). Гибкость модели ParJava позволила применить данный подход на инструментальном компьютере. Трасса параллельной программы собирается на инструментальном компьютере, с минимальным использованием 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. Собранная модельная трасса передается post-mortem анализатору, который проводит поиск шаблонов.

В разделе 5.3 описываются типы выявляемых шаблонов. Метод позволяет выявить 17 типов шаблонов, связанных с точка-точка коммуникационными функциями MPI: семь типов с блокирующими функциями, два типа, связанных с неправильным порядком сообщений, семь типов, связанных с неблокирующими функциями и один тип с применением близкой посылки-получения.

В разделе 6 приводится описание программного обеспечения. Среда ParJava интегрирована с открытой средой разработки Java-программ Eclipse. 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.

Описывается взаимодействия пользователя со средой для создания и доступа к различным инструментариям системы. Приводится диаграмма иерархии классов, описывающих модель параллельной программы в среде.

В разделе 7 приводится описание эффектов, связанных с реализацией JVM и влияющих на производительность параллельной программы и результаты численных экспериментов. Использование специфических факторов, описанных в разделе 7.1, прикладным программистом может привести к потенциальному росту производительности параллельной программы.

Далее приводится описание результатов численных экспериментов для двух программ-примеров, использовавшихся при реализации среды ParJava:

1. 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.

2. Программа быстрого преобразования Фурье (БПФ) на 3D сетке из набора NAS Parallel Benchmarks (NPB).

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. На Рисунке 1 приведен график зависимости ускорения программы моделирования ИАВ от количества используемых процессов для кластера МСЦ РАН. На Рисунке 1 «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. Как видно из графика, начиная со 120 процессов, реальное ускорение программы идет на спад. Интерпретатор также фиксирует падение производительности программы, хотя и с некоторым запаздыванием.

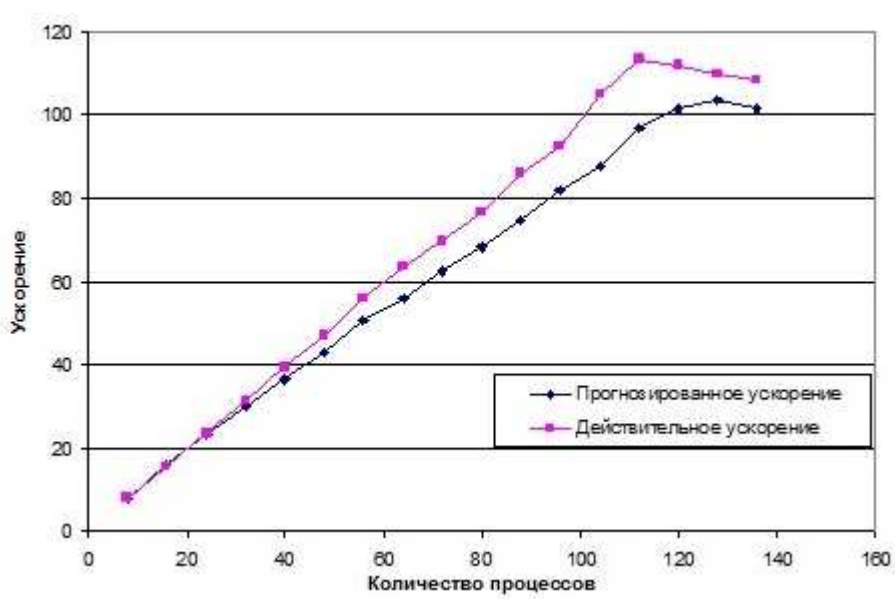

Рисунок 1. Ускорение программы моделирования ИАВ на кластере МСЦ РАН.

Пример 2. На языке Java реализована параллельная программа БПФ из набора NAS Parallel Benchmarks (NPB). На Рисунке 2 представлен график зависимости времени выполнения от числа используемых ядер вычислительной платформы для программы БПФ. Размер рассчитываемой матрицы 512x512x256, объем памяти требуемой в задаче, примерно 5Gb, количество внешних итераций 20. На приведенном графике FT соответствует параллельной программе БПФ на языке Java с явными обращениями к MPI. В этом случае применялся 2D декомпозиция, на каждом узле кластера запускается по восемь процессов.

График FT_T соответствует версии параллельной программы БПФ с использованием и интерфейса MPI, и потоков Java (МПМП). В случае с FT_T

применяется 1D декомпозиция – на каждом узле запускается один процесс, внутри которого при обработке гнезд циклов программы порождаются по восемь потоков. 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. Были проведены исследования по сравнению ускорения МПМП программы БПФ, измеренное при выполнении программы на вычислительной платформе, и ускорению, предсказанного интерпретатором ParJava на инструментальной машине. Интерпретатор модели достаточно точно (3-7%) предсказал ускорение программы при увеличении количества процессов/потоков.

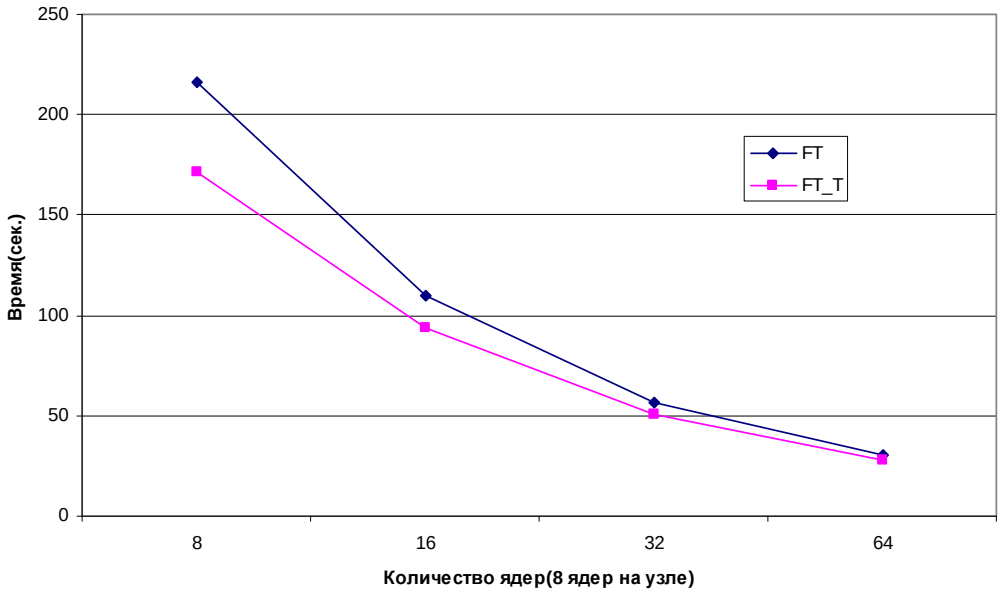

Рисунок 2. Зависимость времени выполнения программы быстрого преобразования Фурье от количества используемых ядер.

Реализованный инструмент автоматизированного обнаружения коммуникационных шаблонов MPI был применен к прикладной параллельной программе расчета вязкого обтекания затупленной головной части (НПО машиностроение). Данная программа написана на языке C++ с использованием MPI, объем исходного кода 4500 строк. Поиск шаблонов проводился на реальной трассе параллельной программы. Инструмент обнаружил в программе пять шаблонов.

После исправления программы производительность программы улучшилось 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 MPI. Объем исходного кода составляет 5103 строки. При анализе данного приложения поиск шаблонов проводился на модельной трассе полученной интерпретатором ParJava на инструментальном компьютере. В данной программе было обнаружено три шаблона. После исправления программы производительность программы улучшилась на 1-2%.

В разделе восемь - Заключение формулируются результаты диссертационной работы и приводятся направления дальнейших исследований. Одним из направлений дальнейших исследований является разработка новой модели программы, учитывающая особенности графических процессоров.

Основные результаты работы

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.

По теме диссертации опубликованы следующие работы

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