**2.3. Механизмы времени выполнения среды ParJava**

Для обеспечения возможности использования среды Java на высокопроизводительных кластерных системах были реализованы стандартная библиотека MPI и механизм контрольных точек.

**2.3.1. Реализация стандартной библиотеки MPI для окружения Java**

Первой задачей, которую было необходимо решить при разработке среды *ParJava*, была эффективная реализация стандартной библиотеки *MPI* для окружения *Java*. В настоящее время известно несколько реализаций библиотеки *MPI* для окружения *Java*, но ни одна из этих реализаций не обеспечивает достаточно эффективных обменов данными. Кроме того, в них реализованы не все функции библиотеки*MPI*. Поэтому для среды *ParJava* была разработана оригинальная реализация библиотеки *MPI* для окружения *Java* – библиотека *mpiJava*.

В настоящее время в библиотеке *mpiJava* поддерживаются все функции стандарта *MPI* 1.1, а также параллельные операции ввода-вывода из стандарта *MPI* 2.

Библиотека *mpiJava* реализована путем «привязки» (*binding*) к существующим реализациям библиотеки*MPI* с помощью интерфейса *JNI* по аналогии с «привязкой» для C++, описанной в стандарте *MPI* 2.

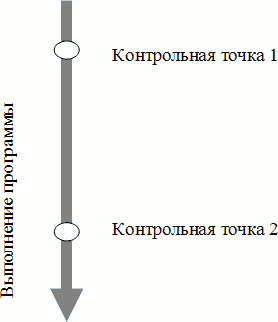
Начиная с версии 1.4, в *Java* поддерживаются прямые буферы, содержимое которых может находиться в памяти операционной системы (вне кучи *Java*). Использование прямых буферов при передаче данных позволяет избежать лишних копирований данных. Это позволяет сократить накладные расходы на передачу данных.

**2.3.2. Механизм контрольных точек**

В среде ParJava реализован инструмент “CheckPoints”, реализующий механизм контрольных точек, который позволяет существенно сократить объемы сохраняемых данных и время на их сохранение.

Реализация механизма контрольных точек для параллельной программы является нетривиальной задачей. Параллелизм усложняет процесс установки точек останова, т. к. сообщения порождают связи между отдельными процессами, и приходится обеспечивать так называемые консистентные состояния. Состояние двух процессов называется неконсистентным, если при передаче сообщения одного процесса другому, может возникнуть состояние, когда первый процесс еще не послал сообщение, а во втором уже сработала функция получения сообщения. Если в таком состоянии поставить точку останова, то восстановив впоследствии контекст задачи, мы не получим ее корректной работы.

Пользователь указывает в программе место сохранения данных с помощью директивы EXCLUDE\_HERE. В контрольной точке 1 (рис. 4) не сохраняются данные, которые будут обновлены до своего использования («мертвые» переменные). В контрольной точке 2 не сохраняются данные, которые используются только для чтения до этой контрольной точки. Результатом будет значительное уменьшение размеров сохраняемых данных в контрольной точке и уменьшение накладных расходов на их сохранение.

   
*Рис. 4. Контрольные точки*

Вначале граф потока управления *G*=(*N*, *E*) разбивается на подграфы *G*′. Корнем каждого подграфа *G*′ является директива EXCLUDE\_HERE. Подграф включает все пути, достижимые из этой директивы, не проходящие через другую директиву EXCLUDE\_HERE.

Для каждого подграфа вычисляются 2 множества переменных: *DE*(*G*′) - множество переменных, которые «мертвы» на каждой директиве EXCLUDE\_HERE в *G*′ и *RO*(*G*′) - множество переменных, предназначенных только-для-чтения по всему подграфу *G*′.

Для нахождения множеств *DE*(*G*′) и *RO*(*G*′) используется консервативный анализ потока данных. В каждом состоянии S в программе вычисляются два множества характеризующие доступ к памяти: *use*[*S*] – множество переменных, которые могут быть использованы вдоль некоторого пути в графе, и *def*[*S*] – множество переменных, которым присваиваются значения до их использования вдоль некоторого пути в графе, начинающегося в S, или множество определений переменных.

Ячейка памяти *l* является «живой» в состоянии S, если существует путь из S в другое состояние *S*′ такой, что *l* http://citforum.ru/pictures/symbol/in.gif*use*[*S*′] и для каждого *S*′′ на этом пути *l*http://citforum.ru/pictures/symbol/notin.gif*def*[*S*′′]. Ячейка *l* является элементом *DE*(*G*′), если *l*«мертвая» во всех операторах сохранения контрольных точек в *G*′.

Ячейка памяти *l*является ячейкой только-для-чтения в операторе *S*, если *l* http://citforum.ru/pictures/symbol/notin.gif*use*[*S*]. Поэтому *l*http://citforum.ru/pictures/symbol/in.gif *RO*(*G*′) тогда и только тогда, когда lhttp://citforum.ru/pictures/symbol/notin.gif*gen*[*S*] для всех *S* в *G*′.

Эти определения консервативны, поскольку они ищут все возможные пути через граф потока управления, тогда как некоторые из них могут никогда не достигнуть выполнения в программе.

Анализ «живых» переменных обычно выражается в виде уравнений потока данных, по одному на каждое состояние в программе. Мы дадим уравнение потока данных, которое позволит нам определить *DE*(*G*′) и*RO*(*G*′) для каждого подграфа *G*′ в программе. Каждое из этих уравнений может быть решено обычным итеративным методом.

Для достижения нашей цели уравнение потока данных характеризуем его функцией обновления. Такая функция ассоциируется с каждым состоянием *S*. Определим *in*[*S*] как множество мертвых переменных в точке непосредственно перед блоком *S*, а *out*[*S*] – такое же множество в точке, непосредственно следующей за блоком *S*.

Вычисляем множество мертвых переменных в направлении обратном графу потока управления. Система уравнений выглядит следующим образом:

*out*[*S*] = ∩*S*′ *in*[*S*′]

*in*[*S*] = *FS*,

где *out*[*S*] = ∩*S*′ *DEAD*[*S*′] – это пересечение множества мертвых переменных для всех состояний S′, которые являются преемниками S в G, а FS – это функция обновления, которая в нашем случае равна *FS*= *out*[*S*]http://citforum.ru/pictures/symbol/cup.gif*gen*[*S*]*– use*[*S*]и свидетельствует о переменных, которые мертвы непосредственно перед *S* и тех переменных, которые стали мертвыми после *S*, плюс о тех, в которые производилась запись в состоянии*S*, минус любые ячейки, с которых происходило чтение в S.

При следующем запуске эти массивы и параметры загружаются, и по ним полностью восстанавливается контекст прерванной задачи.

**3. Приложения среды ParJava**

В этом разделе описываются результаты применения разработанной методологии поддержки разработки параллельных программ на примере создания программ моделирования интенсивных атмосферных вихрей (ИАВ) и моделирования теплового движения молекул воды и ионов в присутствии фрагмента ДНК.

**3.1. Моделирование процесса зарождения торнадо**

В Институте физики Земли РАН была разработана математическая модель развития торнадо в трехмерной сжимаемой сухоадиабатической атмосфере. Большой объем вычислений для получения численного решения потребовал реализации программы на высокопроизводительных вычислительных кластерах. Рассматриваемая система уравнений является сильно нелинейной системой смешанного типа. Для решения системы использовалась явная разностная условно-устойчивая схема второго порядка точности по времени и пространству; критерии ее устойчивости оказались близкими к явной схеме Маккормака.

Программа разработана в Институте системного программирования РАН в сотрудничестве с Институтом физики Земли РАН с использованием среды *ParJava* и предназначена для выполнения на кластерных вычислительных системах.

Программу можно разделить на два блока: загрузка/инициализация данных и главный цикл. Сохранение результатов происходит во время выполнения главного цикла. Входные данные хранятся в файле, где перечислены физические параметры модели и вспомогательные данные для работы программы, например, количество выдач значимых результатов и пути.

Для выявления возможностей распараллеливания циклы были исследованы при помощи Омега теста, реализованного в среде *ParJava*, который показал отсутствие зависимостей по данным между элементами массивов, обрабатываемых в циклах. Это позволило разделить массивы на блоки и распределить полученные блоки по процессорам кластера. Поскольку разностная схема является трехточечной, возникают теневые грани ширины в один пространственный слой. На текущей итерации используются только данные, вычисленные на предыдущих итерациях. Это позволяет обновлять теневые грани один раз при вычислении каждого слоя, что снижает накладные расходы на пересылку. Исследования на интерпретаторе показали, что двумерное разбиение массивов наиболее эффективно, поэтому в программе использовалось двумерное разбиение. Вариант с трехмерным разбиением оказался не самым оптимальным из-за неоднородности вычислений по оси Z.

Как показано в разд. 2, для данного класса задач необходимо использовать коммуникационный шаблон с использованием неблокирующих коммуникаций.

Так как программа моделирования торнадо – это программа с большим временем счета: 300 секунд жизни торнадо на кластере МСЦ (64 вычислителя Power 2,2 GHz, 4 GB) рассчитывались около недели, − и большим объемом генерируемых данных, использовался механизм контрольных точек.

На рис. 5 приводятся графики ускорения программы при блокирующих и неблокирующих обменах данными. При тестировании производительности использовались начальные данные рабочей задачи, но вычислялись только первая секунда жизни торнадо.

Данные результаты вычислений использовались в исследовании процесса зарождения торнадо и они продемонстрировали адекватность используемой модели и возможность использования среды *ParJava*для разработки такого рода приложений.

Более подробно результаты моделирования торнадо рассматриваются в работе [[22](http://citforum.ru/programming/digest/high_productive/3.shtml#ref22)].

##### 3.2. Моделирование теплового движения молекул воды и ионов в присутствии фрагмента ДНК

Для моделирования теплового движения молекул воды и ионов в присутствии фрагмента ДНК методом Монте-Карло существовала параллельная программа, написанная на языке Fortran с использованием MPI. Биологические и математические аспекты задачи и программы описаны в работе [[23](http://citforum.ru/programming/digest/high_productive/3.shtml#ref23)]. В параллельной программе исходное пространство, заполненное молекулами, разделяется на равные параллелепипеды по числу доступных процессоров. Процессоры проводят испытания по методу Монте-Карло над каждой молекулой. Решение о принятии новой конфигурации принимается на основе вычисления изменения энергии. Для вычисления изменения энергии на каждом шаге производится выборка соседей, вклад которых учитывается при вычислении. После того как процессоры перебирают все молекулы, производится обмен данными между процессорами, моделирующими соседние области пространства.

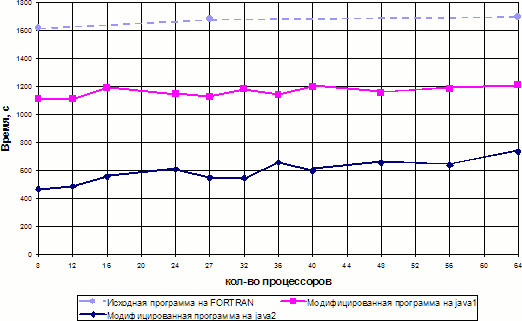
Исходная Fortran-программа требовала для моделирования реальных систем большого числа процессоров (от 512) и могла выполняться только на количестве процессоров, равному кубу натурального числа, то есть на 8, 27, 64 и т.д. процессорах. Это приводило к увеличению времени, необходимого для моделирования одной системы. Главная проблема заключалась в кубическом росте количества перебираемых молекул при выборе соседей, с увеличением количества молекул, моделируемых на одном процессоре.

В среде ParJava была реализована аналогичная программа на языке Java с использованием MPI. Исследование производительности Java-программы позволило выявить причину неэффективной работы Fortran-программы. Для сокращения объема вычислений при построении списка соседей были предложены и реализованы две модификации. Основная модификация заключалась во введении дополнительного разделения молекул по пространству. Это позволило уменьшить количество перебираемых молекул и сократить объем вычислений при построении списка соседей.

Для оценки эффективности произведенных модификаций сравнивались три программы. Исходная Fortran-программа с использованием *MPI*, сравнивалась с двумя *Java*-программами, также использующими *MPI*. Первая *Java*-программа включает в себя только одну модификацию, заключающуюся в использовании одного набора соседей при вычислении энергии для текущего и измененного положения молекулы. Во второй программе, дополнительно было реализовано разбиение на поддомены. Обе *Java-*программы можно запускать на произвольном числе процессоров, с одним ограничением: по каждому из направлений исходная ячейка должна быть разделена хотя бы на 2 домена.

Для сравнения производительности была проведена серия тестов с разным числом вычислителей. Объем задачи увеличивался с ростом числа вычислителей таким образом, чтобы на каждом вычислителе был одинаковый объем данных. На 8 вычислителях моделировалась кубическая ячейка с ребром 100 ангстрем, а на 64 вычислителях ребро ячейки составляло 200 ангстрем, при этом каждый вычислитель моделировал 4166 молекул. В процессе моделирования над каждой из молекул проводилось по 1000 испытаний.

Результаты измерения, приведенные на рис. 6, получены на кластере ИСП РАН, состоящем из 12 узлов (*2 х Intel Xeon X5355*, 4 ядра), объединенных сетью *Myrinet*. Число *MPI*-процессов соответствовало числу ядер, каждое из которых может рассматриваться как отдельный вычислитель. Для Fortran-программы измерения производились с использованием 8, 27 и 64 ядер. Для *Java*-программ измерения производились с использованием 8, 12, 16, 24, 27, 32, 36, 40, 48, 56, 64 ядер.

   
*Рис. 6. Сравнение исходной программы на языке FORTRAN77 c модифицированными программами на языке Java.*

Из графика видно, что при равных объемах данных первая модифицированная программа на *Java*работает в 1,5 раза быстрее, а вторая в 2-3 раза, при этом удалось сохранить точность расчета. Программа на *Java*, в точности соответствующая исходной программе на языке Фортран, требовала от 10 до 30% больше времени и имела аналогичную форму графика производительности.

Исследование и модификация программы в среде ParJava позволило увеличить объем решаемой задачи с полным сохранением свойств модели. Модифицированная программа позволила смоделировать на 128 вычислителях кластера МСЦ фрагмент В-ДНК, состоящий из 150 пар нуклеотидов (15 витков двойной спирали, 9555 атомов) и водную оболочку, содержащую ионы *Cl-* и *Na+*. Ячейки, включавшая этот фрагмент, имела размер в 220Å, и он содержал ~300 тысяч молекул воды. Время работы программы 9 часов, за это время над ионами и молекулами воды было произведено по 10000 элементарных испытаний.

Более подробно результаты моделирования теплового движения молекул воды и ионов в присутствии фрагмента ДНК приводятся в работе [[24](http://citforum.ru/programming/digest/high_productive/3.shtml#ref24)].

#### 4. Направления дальнейших исследований

В настоящем разделе рассматриваются дальнейшие работы по параллельному программированию. Ставится цель разработать методы и реализовать соответствующие инструментальные средства, позволяющие в автоматическом режиме выявлять потенциальный параллелизм, генерировать параллельный код и осуществлять доводку полученного кода с учетом особенностей выбранной аппаратуры (кластер, система с общей памятью, кластер с многоядерными узлами).

Для анализа и трансформации гнезд циклов, в которых все индексные выражения и границы циклов задаются аффинными формами относительно индексов массивов, будет разработана инфраструктура, базирующаяся на декларативном представлении гнезд циклов в виде выпуклых многогранников в пространстве индексов. Такое представление существенно снижает накладные расходы на анализ и трансформацию гнезд циклов, так как позволяет выполнять их с помощью операций над матрицами. В составе инфраструктуры будет реализован набор методов (API), реализующих семь базовых трансформаций циклов (любая трансформация цикла является их суперпозицией), а также методы, позволяющие определять зависимости по данным, выявлять шаблоны доступа к памяти, вычислять границы циклов при изменении порядка циклов в гнезде и др. Кроме того, будет разработана и реализована подсистема преобразования императивного представления (байт код) программы в декларативное, а также подсистема преобразования декларативного представления в параллельные программы для систем с распределенной памятью (Java+MPI) или для систем с общей памятью (Java-треды). Будет исследован круг вопросов связанных с генерацией эффективного кода в модели, когда каждый процесс MPI является многотредовым. Будет проведен сравнительный анализ такой реализации с реализацией на Java+MPI на кластерах с многоядерными узлами.

Будет разработан инструмент, позволяющий в автоматическом режиме подбирать коммуникационные примитивов с использованием методики, рассмотренной в разделе 2. В основе инструмента многократная интерпретация модели разрабатываемой параллельной программы. Для обеспечения многократной интерпретации в приемлемое время будет реализована возможность автоматической генерации «скелета» реального приложения.

Будут исследованы произвольные (не аффинные) гнезда циклов и разработаны инструментальные средства, позволяющие распараллеливать их в диалоговом режиме: инструмент для выяснения наличия зависимостей по данным между итерациями цикла с помощью синтетического Омега-теста и инструменты для вычисления вектора направлений и вектора расстояний между, характеризующих зависимости между итерациями гнезда циклов.

В последнее время получили распространение специализированные устройства, обеспечивающие высокую степень параллелизма. Одним из классов таких устройств являются графические акселераторы. При стоимости и энергопотреблении, сравнимыми с процессорами архитектуры x86-64, они превосходят их по пиковой производительности на операциях с плавающей точкой и пропускной способности памяти приблизительно на порядок. Большой интерес вызывают исследования возможности использования неоднородных вычислительных архитектур, включающих универсальные процессоры и акселераторы для решения задач, не связанных непосредственно с обработкой графики.

Для разработки программ для таких гибридных систем в настоящее время используется модель программирования CUDA, первоначально разработанная для акселераторов Nvidia. Она точно отражает организацию оборудования, что позволяет создавать эффективные программы, но в то же время требует от разработчика хорошего понимания архитектуры акселератора, а перенос существующего кода для выполнения на акселераторе с помощью CUDA обычно требует значительных модификаций.

Соответственно, актуальной является задача разработки технологий компиляции, позволяющих упростить написание эффективных программ и перенос существующего кода на графические акселераторы. Для этого предлагается определить набор прагм, позволяющих выделить участки кода, которые должны быть скомпилированы для выполнения на акселераторе. Чтобы быть разумной альтернативой более низкоуровневым средствам, такой набор расширений должен быть достаточно гибким, чтобы позволять улучшать производительность кода за счёт тонкой настройки конфигурации потоков выполнения и распределения данных в иерархии памяти акселератора. В то же время, реализованные средства должны позволять последовательный перенос программного кода на акселератор с минимальными изменениями в исходных кодах и процессе компиляции.

Реализацию предлагается осуществить в компиляторе GCC, который является де-факто стандартным компилятором для операционной системы Linux, поддерживает несколько входных языков (C, C++, Fortran, Java, Ada и другие) и позволяет генерировать код для множества архитектур. В GCC уже реализованы OpenMP 3.0 и система анализа зависимостей и трансформации циклов GRAPHITE, что является существенной частью необходимой для такого проекта инфраструктуры.