Разработка параллельных Java программ для высокопроизводительных вычислительных систем с распределенной памятью*

В.П. Иванников, С.С. Гайсарян, А.И. Аветисян, В.В. Бабкова, В.А. Падарян

Аннотация. В работе рассматривается интегрированная среда ParJava, поддерживающая разработку и сопровождение программ, параллельных по данным. При разработке и модификации параллельной программы необходимо убедиться не только в ее правильности, но также в ее эффективности и масштабируемости. Однако анализ динамических свойств программы (профилей, трасс, слайсов и т.п.), позволяющий установить ее эффективность и масштабируемость, как правило, бывает связан с необходимостью многочисленных прогонов еще не полностью отлаженной программы на целевом вычислительном комплексе (высокопроизводительном кластере). Среда ParJava предоставляет разработчику программы, параллельной по данным, широкий набор инструментов, позволяющих анализировать динамические свойства программы в процессе ее разработки. Эти инструменты позволяют получить достаточно точные оценки времени выполнения программы как функции числа узлов параллельной вычислительной системы и границ области масштабируемости. В среду ParJava включен символьный интерпретатор модели параллельной программы, который позволяет осуществлять ее динамический анализ на инструментальном компьютере, сокращая время разработки программы и затраты на ее отладку.

1. Введение

В работе рассматривается интегрированная среда ParJava [1], поддерживающая разработку и сопровождение программ, параллельных по данным. Программы, параллельные по данным (в частности, SPMD-программы) характеризуются сравнительно высоким уровнем обмена данными между узлами параллельной вычислительной системы во время выполнения программы. Поэтому обеспечение таких свойств программы, как ее эффективность и масштабируемость требует дополнительной «доводки» программы, во время которой в программе обнаруживаются и устраняются «узкие места», мешающие достичь необходимого уровня ее масштабируемости. При «доводке» программы полезно знать ее динамические свойства (профили, трассы, слайсы и т.п.).

* Работа поддержана грантами РФФИ 02-01-00961, 02-07-90302, 03-07-90198 и грантом МК-2371.2003.01 Президента Российской Федерации

Среда ParJava предоставляет разработчику программы, параллельной по данным, широкий набор инструментов, позволяющих анализировать динамические свойства программы в процессе ее разработки. Эти инструменты позволяют получить достаточно точные оценки времени выполнения программы как функции числа узлов параллельной вычислительной системы и границ области масштабируемости. В среду ParJava включен символьный интерпретатор модели параллельной программы, который позволяет осуществлять ее динамический анализ не на целевой вычислительной системе, а на инструментальном компьютере, сокращая время разработки программы и затраты на ее отладку («доводку»).

Использование языка Java для разработки высокопроизводительных параллельных программ вызывает большой интерес у программистского сообщества. Большая часть работ в этом направлении предлагает реализацию стандартного интерфейса MPI в окружении Java. Причина этого в том, что большая часть SPMD-программ реализуется с использованием этого интерфейса, несмотря на сложности, связанные с необходимостью в явном виде задавать все обмены. Языки высокого уровня (например, HPF [2]) используются редко, так как не обеспечивают возможности «доводки» программы и повышают накладные расходы при обмене данными (это связано, в частности, с тем, что функции MPI вызываются не непосредственно из программы, а из системы поддержки соответствующего компилятора). Поэтому в среде ParJava разработка прикладных программ ведется с явным использованием вызовов функций MPI.

Будучи не в состоянии облегчить разработку параллельных программ с помощью языковых средств высокого уровня, мы разработали ряд инструментов, применение которых может помочь прикладному программисту при разработке, «доводке» и модификации его программы. Некоторые из этих инструментов обсуждаются в данной работе.

Работа состоит из восьми разделов. В разделе 2 содержится краткое описание среды ParJava. В разделе 3 рассматриваются инструменты, поддерживающие разработку параллельных программ: в основном это средства сбора профилей и трасс. Средства, позволяющие определить возможность распараллеливания циклов, и средства, помогающие программисту преобразовать цикл таким образом, чтобы он допускал параллельное выполнение, в работе не рассматриваются, хотя ряд таких средств включен в среду ParJava. В разделе 4 описывается иерархическая модель параллельной программы и интерпретатор этой модели. Использование иерархической модели позволяет предсказать время выполнения программы на заданном числе узлов и оценить границы ее масштабируемости. В разделе 5 описывается подсистема мониторинга параллельной программы. В разделе 6 обсуждаются результаты экспериментальных расчетов с использованием среды ParJava. В разделе 7 среда ParJava сравнивается с другими инструментальными системами, поддерживающими разработку параллельных программ.

2. Краткое описание среды ParJava

2.1. Реализация интерфейса MPI в среде Java

В настоящее время коммуникационные библиотеки окружения Java обеспечивают модель «клиент-сервер» и не поддерживают симметричных коммуникаций, которые необходимы для организации высокопроизводительных параллельных вычислений. Поэтому в среде ParJava был реализован стандартный интерфейс MPI, обеспечивающий симметричные коммуникации.

Реализация MPI не должна увеличивать латентность и другие накладные расходы на организацию коммуникаций и уменьшать пропускную способность коммуникационного оборудования. Реализация MPI на «чистой» Java не может выполнить это условие, так как в этом случае коммуникации будут управляться JVM, и накладные расходы на организацию коммуникаций не смогут быть уменьшены ниже некоторого порога, определяемого особенностями работы JVM. Поэтому, в среде ParJava, MPI был реализован в виде «привязки» (binding) к реализации на языке C. Такой подход позволил использовать высокоэффективную реализацию MPI, которая учитывает аппаратную специфику используемого коммуникационного оборудования. Это позволяет сосредо-точиться на оптимизации кода «привязки», размер которого существенно меньше размера реализации MPI на Java.

Для реализации MPI в среде ParJava используется аппарат прямых буферов, поддерживаемый пакетом `java.nio` в реализациях Java начиная с версии 1.4 [3].

2.2. Основные возможности среды ParJava (графический интерфейс)

В среде ParJava взаимодействие с прикладным программистом осуществляется через графический интерфейс пользователя (GUI). Главное меню графического интерфейса (Рис. 1) обеспечивает доступ к следующим группам инструментов среды ParJava: *File, Edit, View, Search, Run, Analyze, Transform, Visualize, Help.*

Меню *File, Edit, View, Search* содержат традиционные инструменты, обеспечивающие стандартные возможности. Остальные меню обеспечивают доступ к специальным инструментам среды ParJava.

Меню *Run* позволяет начать выполнение программы или ее символьную интерпретацию.

Меню *Analyze, Transform* и *Visualize* соответственно обеспечивают доступ к *анализаторам*, которые осуществляют различные виды статического и динамического анализа разрабатываемой программы, *преобразователям,* выполняющим требуемые преобразования программы, и *визуализаторам*, отображающим на монитор результаты работы анализаторов и других инструментов среды ParJava.

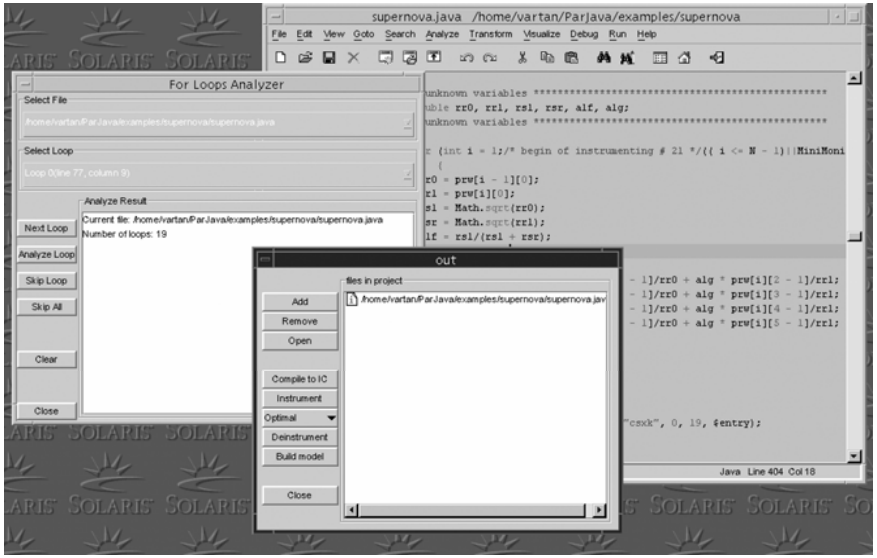

Рис. 1. Главное меню графического интерфейса пользователя среды ParJava.

В текущей версии среды ParJava меню *Analyze* обеспечивает доступ к следующим инструментам: *Slice* – построение обратного динамического слайса для заданного набора переменных в заданной точке программы; *Sequential* – выделение последовательной части параллельной программы; *AmdahlRatio* – вычисление отношения Амдаля [4]; *ForLoop* – анализ возможности распараллеливания заданного цикла *for* с помощью теста расстояний [5]. Через меню *Transform* доступны следующие инструменты: *Transform to IC* – построение внутреннего представления параллельной программы; *Compile* – компиляция внутреннего представления отдельного фала параллельной программы в байт-код; *Build Project* – сборка всего параллельного приложения; *Parallelize* – преобразование гнезда циклов к виду, допускающему распараллеливание [6]; *Instrumentate* – вставка обращений к инструментальным функциям в базовые блоки программы. Результаты анализов помогают прикладному программисту исследовать разрабатываемую программу, в результате символьной интерпретации получаются оценки границ области масштабируемости (по Амдалю [4], или по Густафсону [7]).

3. Инструменты, поддерживающие разработку параллельных программ

3.1. Внутреннее представление параллельной Java-программы

В среде ParJava разрабатываемая прикладная программа может иметь три представления: исходный текст (Java + MPI), внутреннее представление (IC) и байт-код (BC).

Представление IC используется в большей части анализаторов, для символьной интерпретации и в других инструментах, доступных прикладному программисту через графический интерфейс.

Во внутреннем представлении каждый метод, описанный в классах параллельной программы, представляется своим графом потока управления: вершинами этого графа являются базовые блоки, дуги задаются списком вершин. Каждый базовый блок B задается шестеркой $B=<\tau, P, I, O, L, A>$, где τ – тип базового блока, P – последовательность операторов, I и O – списки входных и выходных переменных, соответственно, L – список базовых блоков, следующих за текущим (дуги графа потока управления), элементы списка L называются последователями блока B, A – список дополнительных атрибутов. В среде ParJava определены следующие типы базовых блоков: линейный участок – линейная последовательность операторов Java-программы (последним оператором такого базового блока может быть оператор ветвления), вызов метода, вызов коммуникационного метода MPI, вызов служебного метода MPI, вызов метода JDK. Каждый оператор в списке P представляется четвёркой: код операции, список операндов (от одного до трёх). Если список L содержит более одного элемента, то каждый его элемент снабжается селектором (селектор – это пара: ссылка на переменную, значение которой позволяет выбрать последующий блок, значение).

Если блок B представляет вызов метода, P содержит две строки, определяющие имя переменной, на которой метод вызван и имя метода, I содержит не более одной переменной (возможно временной), в которой сохраняется значение, возвращаемое методом, L содержит только один элемент, селектор отсутствует.

Внутреннее представление используется для построения модели параллельной программы. Дополнительные атрибуты (например, время выполнения, статистика ветвлений) вырабатываются анализаторами и используются при интерпретации модели. Когда у блока B больше одного последователя, список A может содержать статистическую информацию о том, как это ветвление выполняется при определённых начальных данных.

Таким образом, внутреннее представление программы содержит информацию о потоке управления в каждом методе параллельной Java-программы.

3.2. Инструментирование и профилирование параллельной Java-программы

Определить эффективность и масштабируемость параллельной программы во время ее разработки (отладки) помогают динамический анализ и/или символьная интерпретация ее текущей версии. Динамический анализ программы состоит в определении ее профилей и трасс. Они зависят от входных данных, и не могут быть определены во время компиляции. Профили и трассы параллельной программы могут быть получены при ее выполнении на одном процессоре в специальном окружении, обеспечивающем адекватную

интерпретацию операторов обмена данными между узлами кластера (обращений к интерфейсу MPI).

Для исследования динамических свойств параллельной программы в каждый ее базовый блок добавляются *инструментальные операторы*, которые, не меняя алгоритма программы, обеспечивают сбор данных о свойствах программы в процессе ее выполнения (или интерпретации). Каждый инструментальный оператор – это обращение к библиотеке мониторинга, возвращающее значение атрибутов соответствующего базового блока. Для того, чтобы инструмен-тальные операторы действительно не меняли свойств программы, их количество по возможности минимизируется (для этого можно использовать, например, отношение доминирования, определенное на графе потока управления программы). Кроме того, в системе предусмотрена возможность компенсации влияния инструментальных операторов.

Подсистема компенсации является частью библиотеки мониторинга. Она решает следующие две задачи: 1) корректная расстановка временных меток в записываемых событиях и 2) поддержка правильного порядка обработки сообщений.

Необходимость поддержки порядка обработки сообщений возникает потому, что функции MPI не анализируют принимаемых сообщений и не следят за порядком приема сообщений. Выполнение инструментальных операторов может изменить временные промежутки между коммуникациями, в результате порядок обработки сообщений может измениться. Во многих системах динами-ческого анализа программ эту проблему обходят, обрабатывая трассы после завершения работы программы, когда имеется вся необходимая информация для перестройки трассы. В среде ParJava для поддержки исходного порядка выполнения программы она пересобирается после инструментирования. При этом в нее добавляются компенсационные операторы.

При построении частотного профиля инструментальная программа получает данные о работе исследуемой программы через вызовы инструментальных методов, вставленные при ее инструментировании. Профиль хранится в виде одномерного массива целого типа, каждый элемент которого соответствует одному из базовых блоков. Каждый вызов увеличивает на единицу значение соответствующего элемента массива. Если в процессе инструментирования производилась оптимизация количества инструментальных операторов, профиль достраивается после завершения работы программы.

Профили могут обрабатываться анализатором статистики. В этом случае каждый профиль рассматривается как многомерная выборка. На множестве выборок определяется расстояние между двумя элементами этого множества $\rho = \rho(a_i, a_j)$.

Репрезентативным набором профилей, относительно определяющего критерия $K = (\delta, p, Q), 0 < p < 1, 0 < \delta, Q > 1, Q \in N$, понимается следующее множество профилей S:

$$\exists SBS = \{a_i\} : SBS \subset S, \forall a_i, a_j \in SBS \Rightarrow \rho = \rho(a_i, a_j) < \delta, p \le |SBS| / |S|, |S| \ge Q .$$

3.3. Выполнение (интерпретация) инструментированной программы

Для выполнения инструментированной программы она транслируется в исходное представление на языке Java. Это позволяет, не меняя средств сборки проекта, снова скомпилировать байт-код, который теперь содержит вызовы инструментальных функций. Результаты сохраняются в отдельных файлах, каждый файл соответствует одному процессу параллельной программы. Помимо динамических характеристик программы, в файлы записывается служебная информация: номер процесса, параметры и время запуска и др.

3.4. Компенсация влияния инструментальных операторов

Инструментальные операторы могут изменить работу программы и исказить ее временной профиль. Кроме того, может произойти искажение трасс параллельных процессов. Для иллюстрации искажения трасс рассмотрим следующий пример: первый процесс получает сообщения от двух других процессов, причем ее работа зависит от того, какой процесс прислал данные.

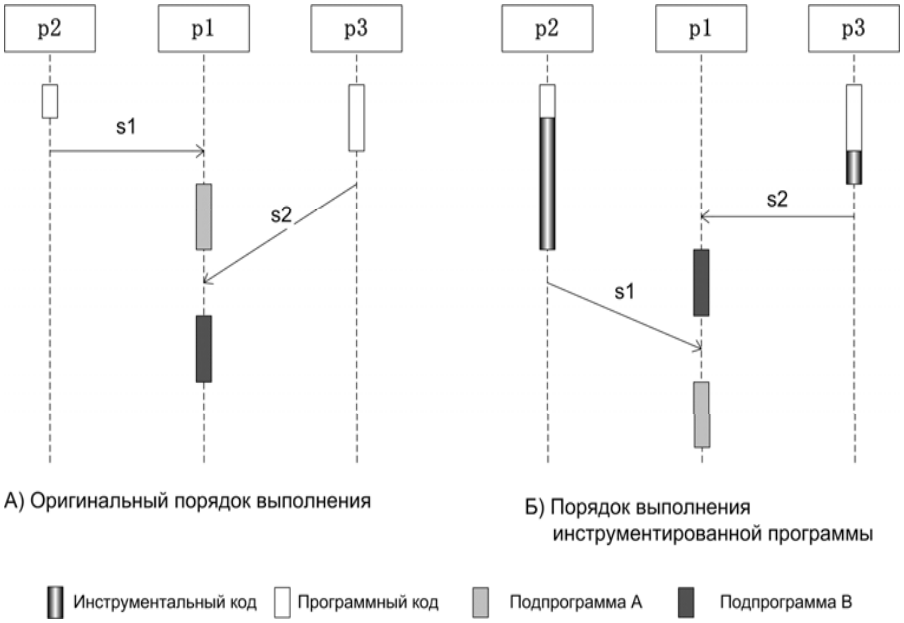

Рис. 2. Влияние инструментального кода на трассу параллельной программы

Пусть, в случае неинструментированного кода последовательность действий параллельной программы была следующей (Рис. 2А): сначала процесс `p1`

получает сообщение `s1` от процесса `p2` и выполняет подпрограмму `A`; затем процесс `p1` получает сообщение `s2` от процесса `p3` и выполняет подпрограмму `B`. После инструментирования программы может оказаться, что выполнение инструментальных операторов в процессе `p2` занимает больше времени, чем их выполнение в процессе `p3`. Тогда процесс `p3` успеет послать сообщение `s2` процессу `p1` раньше, чем процесс `p2` пошлет сообщение `s1` (Рис. 2Б); в результате процесс `p1` сначала получит сообщение `s2` от процесса `p3` и выполнит подпрограмму `B`, а потом, получив сообщение `s1` от процесса `p2`, выполнит подпрограмму `A`. Таким образом, трасса инструментированной программы будет отличаться от трассы исходной программы.

Для предотвращения искажений временного профиля необходимо, чтобы каждый из параллельных процессов инструментированной программы поддерживал три таймера: CT_i (показание внутренних часов узла, на котором выполняется процесс i), Δ_i (время, которое процесс i потратил на выполнение инструкций, добавленных в код программы в процессе ее инструментирования) и LT_i (локальное время процесса i, т.е. то время, в которое неинструмен-тированная программа находилась бы в той же точке выполнения, что и инструментированная). Легко видеть, что $LT_i = CT_i - \Delta_i$. Методика синхронизации часов и технология работы с таймерами описаны в [8] и [9].

Для предотвращения искажения профиля достаточно вместо CT_i заносить в профиль значение LT_i.

4. Иерархическая модель параллельной Java-программы

4.1. Построение иерархической модели

Наиболее точной моделью программы является сама программа, но такой подход не пригоден, так как предусматривает использование целевой платформы или средств симулирующих весь набор интерфейсов доступных выполняющейся программе (центральный процессор, системные и библиотечные вызовы). Поэтому в модели должны отражаться только те свойства программы, обладание которыми позволит оценить её время работы.

Иерархическая модель программы определяется как набор направленных графов $\{G_i\}$, где каждой функции (методу) соответствует единственный граф G_i – граф потока управления i-ой функции. Представление каждого G_i описано в разделе 3.1. Каждая вершина каждого графа снабжена заполненным списком атрибутов A.

Рассмотрим некоторые проблемы, которые приходится решать при построении модели. Если параллельная программа написана на объектно-ориентированном языке вызовы методов могут происходить с динамическим определением типа объекта (механизм виртуальных методов в C++). В общем случае, невозможно

статически определить переход управления из точки вызова такого метода, и при построении модели не удастся построить ребро вызова для вершины представляющей точку вызова. Решать эту проблему можно либо статически определяя тип переменной, на которой метод вызывается, либо профилируя программу – у отдельно взятого места вызова может быть всегда один и тот же динамический тип объекта, на котором происходит вызов метода.

Помимо динамического определения типов, представляет сложность моделирование блоков `try{}catch(){}`. Выход из блока `try` может потен-циально произойти во время выполнения любого оператора. Попадая в блок `catch` невозможно определить место программы, из которого пришло управление, не анализируя стек вызова методов. Можно рассматривать блоки `try{}catch(){}` как обработку ситуаций, не происходящих при обработке корректных данных отлаженной программой на целевой системе. Тогда содержимое `try` становиться просто блоком, а блок `catch` исключается из рассмотрения.

При построении модели у каждого базового блока должны быть определены атрибуты из списка A (см. раздел 3.1). Сложности, возникающие при определении этих атрибутов, связаны с особенностями работы системы замера времени на аппаратуре. Типичный случай, когда аппаратный счётчик времени не обеспечивает измерения достаточно малых промежутков времени. Например, если точность измерения времени составляет 1 мкс, то за это время центральный процессор успевает выполнить 10^6 команд. Измерить время выполнения базового блока можно в этом случае одним из следующих двух способов.

Первый способ: создается искусственный контекст (программа), внутри которого рассматриваемый базовый блок может выполняться. Контекст, в частности, должен определять значения всех переменных из списка I. Измерить время работы искусственной программы не составляет труда: можно выпол-нить ее в цикле достаточно большое число раз и поделить суммарное время вы-полнения на число витков цикла. Второй способ измерения предусматривает статический подсчёт одинаковых операций выполняющихся в рассматри-ваемом коде, и последующее их сложение с весом, отражающим время выпол-нения этой операции на целевой машине. Такой подход требует однократной временной оценки операций JVM (сложение чисел, вычисление булевых выра-жений, создание массивов и т.д.) на целевой машине, после чего результатами можно пользоваться для статической оценки любого базового блока.

Для узлов описывающих вызовы функций (методов), измерение времени зависит от типа узла. Если вызывается метод, определенный прикладным программистом, время его выполнения определяется при интерпретации этого метода. Если вызывается коммуникационная функция MPI, время ее работы определяется с помощью модели коммуникационной сети (одной из наиболее распространённых моделей сети является модель LogGP [10]).Другим способом измерения времени выполнения коммуникаций является

использование специального тестового режима работы библиотеки MPI (“Test mode”). В этом режиме выполняются инструментированные методы MPI, которые позволяют вычислить требуемый атрибут для каждого вызова коммуникационных функций MPI.

Если вызывается библиотечная функция, считается, что время ее работы входит в спецификацию используемой библиотеки.

4.2. Интерпретация иерархической модели

Имея определённые атрибуты «время выполнения» во всех узлах можно определить порядок определения времени выполнения всего графа. Делать это можно через построение цепочки узлов, по которым пройдёт выполнение (интерпретация работы программы) или через правила свёртки подграфов.

Под цепочкой будем понимать мультимножество, под записью – добавление элемента в мультимножество, саму цепочку в дальнейшем будем назвать цепочкой выполнения. Первым записывается в цепочку корневой узел графа. Выполняются все четвёрки (если они есть) принадлежащие этому узлу. Если есть атрибут *next*, то следующий узел конец дуги с номером равным значению *next*. Если из узла не выходит никаких дуг, процесс построения цепочки окончен. Временем выполнения графа будем считать сумму атрибутов «время» всех узлов находящихся в цепочке.

Возникает вопрос, как можно модифицировать граф, чтобы процесс построения цепочки происходил за меньшее число шагов, и при этом результат (сумма атрибутов) сохранялся. Возможно рассмотреть более широкий класс изменений, когда результат построения цепочки выполнения модифицированного графа отличается от оригинального результат не более чем на δ. Поскольку рассматривается граф потока управления, то в нём существуют подграфы, сами являющиеся графами потока управления. В некоторых случаях подграф может быть заменён одной вершиной, с атрибутами, построенными по определённым правилам, при этом граф остаётся графом управления.

Если требуется провести свёртку, не изменяя результат построения цепочки управления, то выполнить её можно при выполнении следующих условий:

1. Подобие подграфа. Подграф является графом управления.
2. Замкнутость по результатам. Ни одна из определяемых в подграфе переменных не влияет на значения атрибутов next в узлах не входящих в подграф и над которыми доминирует выход из подграфа.
3. Статичность структуры. Для любых значений переменных при входе в подграф, цепочка управления подграфа одна и та же.

Если в подграфе есть узлы представляющие коммуникации, требуется выполнение ещё двух условий:

4. Замкнутость по коммуникациям. Если в подграфе есть узлы отвечающие вызовам коммуникационных функций, то всем ответным

функциям (в случае `send`/`recv`), или соответствующим групповым соответствуют узлы также принадлежащие этому подграфу.

5. Синхронность начала. Последняя коммуникация, перед входом в подграф произвела синхронизацию всех процессов.

Выделив подграф, удовлетворяющий перечисленным условиям, можно заменить его одним узлом с атрибутом время равным времени выполнения подграфа. Желательно привести свёртками граф к ацикличному виду, так как построение цепочки управления сводиться в таком случае к обходу дерева в глубину.

На практике, модели немногих программ можно будет привести к ациклическому виду, руководствуясь указанными условиями. Ослабление требований можно провести в третьем пункте. Пусть на множестве всех значений переменных минимальное время выполнения подграфа m, максимальное M, и узел являющийся точкой входа появляется в цепочке выполнения Q раз. Тогда, заменив подграф на вершину с временем M, можно сказать что подграф свёрнут с точностью $(M\text{-}m)Q$.

Данное преобразование может применяться к подграфам, относящимся к языковым конструкциям `if` и `switch`.

Граф программы можно изначально представить в ациклическом виде, и проводить свёртки более агрессивно. Введём дополнительные типы вершин – LOOP и IF. Для определения атрибута «время» можно воспользоваться рекурсивными формулами.

1. Если P – это последовательность блоков $p_1, p_2, \ldots, p_M$, а t_i – время выполнения блока p_i, то $T = \sum_i t_i$.
2. Ветвление. Если P – оператор if или switch, $T = \max_{i=1,n} t_i$. Если собрана статистика ветвлений и f_i – вероятность (частота) переходов по i-ой ветви, то $T = \sum_i f_i t_i$, $\sum_i f_i = 1$.
3. Последовательный цикл с I итерациями. $T = tI$, где t – время выполнения тела цикла.
4. Параллельный цикл. $T=tI$, where где t – время выполнения тела цикла, а $I = I(P, N)$ число итераций, зависящее от количества процессоров P и размера задачи N (например, $I \propto \frac{N}{P}$).

Пример. Программа решения системы алгебраических уравнений итерационным методом Якоби (см. раздел 7), имеет следующий приведённый граф для метода `main`.

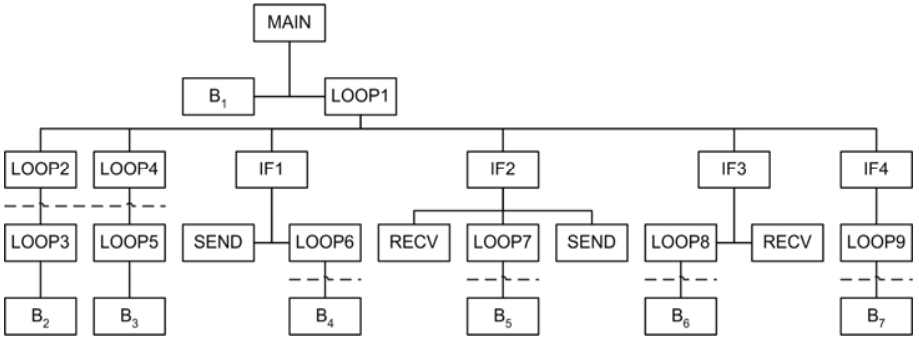

Рис. 3. Иерархическая модель для задачи решения линейных алгебраических уравнений методом Якоби.

Для рассматриваемой программы применение формул свёртки даёт следующий результат $T = t_1 + \frac{N^2}{P}(t_2 + t_3) + N(t_4 + t_5 + t_6 + t_7) + 2t_{send}(N) + 2t_{recv}(N)$. Как видно для определения времени выполнения всего метода `main` достаточно определить время выполнения базовых блоков и промоделировать выполнение операций `send` и `recv`.

5. Мониторинг параллельной Java-программы

Во время отладки параллельной программы важно добиться синхронности выполнения процессов на каждом узле, чтобы свести к минимуму их простои. Добиваться синхронности нужно путем корректировки алгоритма и эффективного распределения данных. Помочь в поиске мест в тексте программы, в которых необходимо улучшение кода, может мониторинг времени выполнения. С его помощью можно отслеживать согласованность работы параллельной программы в целом. Работы по стандартизации мониторов параллельных программ ведутся в Техническом Университете Мюнхена [11]. Опубли-кованный стандарт послужил основой для реализации в среде ParJava.

Основное окно монитора представлено на Рис. 4.

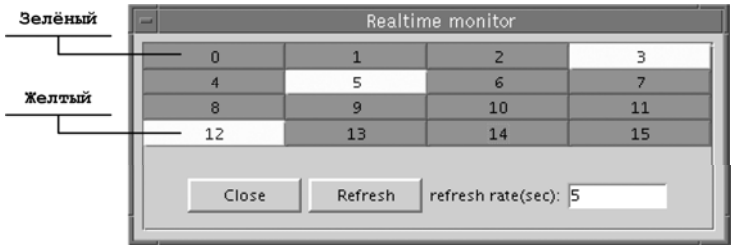

Рис. 4. Основное окно монитора времени выполнения.

Как уже отмечалось, процесс мониторинга может не только замедлить ее работу, но и изменить порядок обработки сообщений (Рис 2). В системе мониторинга для предотвращения этого используется специальная («отладочная») версия библиотеки MPI, поддерживающая исходный порядок обработки сообщений.

В реализации монитора использована архитектура распределённого монитора с центральным управляющим модулем [12]. Монитор состоит из *головной части*, расположенной на управляющей машине и «*сборщиков*», распределённых по узлам кластера. Головная часть обеспечивает интерфейс с пользователем и занимается приёмом данных от «сборщиков». Для передачи данных исполь-зуется протокол TCP/IP. При наличии в вычислительной системе двух сетей – вычислительной и служебной, трафик создаваемый монитором, не будет загружать вычислительную сеть, искажая работу параллельной программы.

Схема, по которой происходит сбор данных на узле, показана на Рис. 5 и состоит в следующем. При запуске параллельной программы под управлением монитора на каждом узле выполняется два процесса: процесс основной программы и процесс «сборщика». При этом выполняется код, устанавливающий связь между этими двумя процессами через стандартные средства IPC. Создаётся общий сегмент памяти, в котором размещается объект, реализующий кольцевой буфер. Для явного указания расположения объекта в памяти используются возможности пакета `java.nio`, доступного в JDK 1.4.

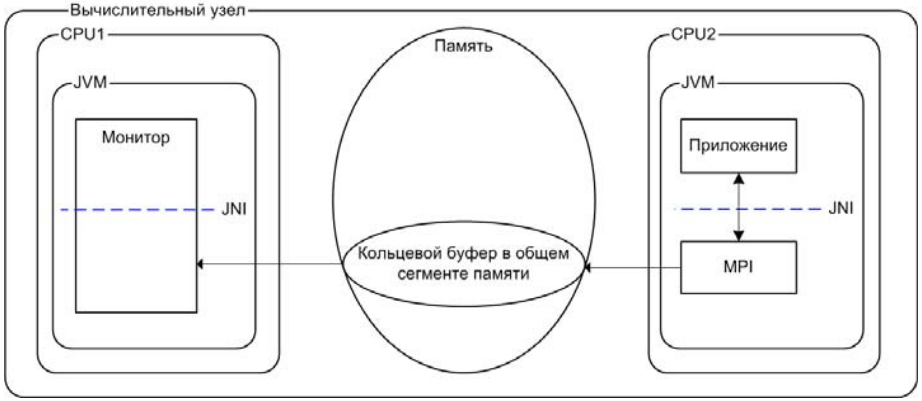

Рис. 5. Схема взаимодействия между параллельной программой и «сборщиком».

Система мониторинга параллельной программы должна позволять варьировать уровень подробности получаемых данных. Это свойство реализовано введением двухуровневого интерфейса системы с пользователем. Графический интерфейс в обычном состоянии показывает только текущее состояние процессов. Основная часть окна представляет собой набор пронумерованных

прямоугольников. Цвет прямоугольника отражает специфику действий, выполняющихся данным процессом. Зелёный цвет соответствует выполнению вычислений, жёлтый – выполнению обмена данными, красный – простою процесса в ожидании прихода данных. Информация о состоянии процессов обновляется с задаваемой пользователем частотой. Более подробную информацию о процессе можно получить «нажав» мышью на соответствующий ему прямоугольник. После нажатия выплывает новое окно, в котором выдаются данные о количестве переданных/принятых данных, времени потра-ченном на вычисления и коммуникации, а также показывается оценка замедления программы из-за мониторинга. Присутствие в программе массовых коммуникаций точка-точка или групповых коммуникаций влияет на текущую пропускную способность сети, доступную для каждого конкретного процесса. Система мониторинга подсчитывает время передачи, что позволяет оценить этот показатель. Доступ к данным о пропускной способности сети (график по времени, последнее значение, среднее значение), также становится возможным при активировании окна процесса.

Функции пакета MPI, используемые параллельной программой записывают в кольцевой буфер сообщения о коммуникациях, происходящих в программе. Монитор читает данные из кольцевого буфера, используя для навигации по буферу временные метки, связанные с каждым сообщением.

6. Результаты экспериментальных расчетов

В этом разделе приводятся результаты экспериментальных расчетов, демонстрирующих такие свойства среды ParJava, как точность интерпретации модели программы, сравнение производительности Java+MPI по сравнению с C+MPI, переносимость параллельных программ.

В качестве первого примера рассматривается программа численного решения трехмерной системы дифференциальных уравнений гидродинамики, моделирующих конвекционные потоки при взрыве сверхновой звезды. Система уравнений предложена в работе [13]. Она имеет вид:

$$\rho\frac{\partial v}{\partial t} = -\mathrm{grad}P - \frac{\rho GM}{r^3}\mathrm{r}, \frac{\partial \rho}{\partial t} + \rho \mathrm{div} v = 0,$$
$$\frac{\partial E}{\partial t} = T\frac{\partial S}{\partial t} + \frac{P}{\rho^2}\frac{\partial \rho}{\partial t}, \frac{\partial S}{\partial t} = 0.$$

где r – плотность вещества, v – скорость вещества, P – давление, E – энергия, S – энтропия.

Для численного решения системы была применена консервативная разностная схема Годуновского типа второго порядка точности. Область была разбита на три концентрические части, каждая часть была покрыта равномерной прямоугольной сеткой, причем шаг сетки каждой внутренней области (по каждому направлению) был вдвое меньше, чем шаг сетки в более внешней области.

Вычисления выполнялись на высокопроизводительном кластере ИСП РАН, имеющем следующие параметры: 8 двухпроцессорных узлов Athlon 1533 MHz с 256 Mb оперативной памяти на каждом, коммуникационные сети – Myrinet, полнодуплексный гиперкуб с пропускной способностью 2 Gb/sec, служебная сеть – Fast Ethernet.

На Рис. 6 ускорение параллельной программы (ускорением называется отношение времени выполнения программы на одном процессоре к ее времени выполнения на p процессорах), вычисленное с помощью ее символьной интерпретации (чёрная кривая), сравнивается с ускорением, полученным при ее фактическом выполнении (серая кривая).

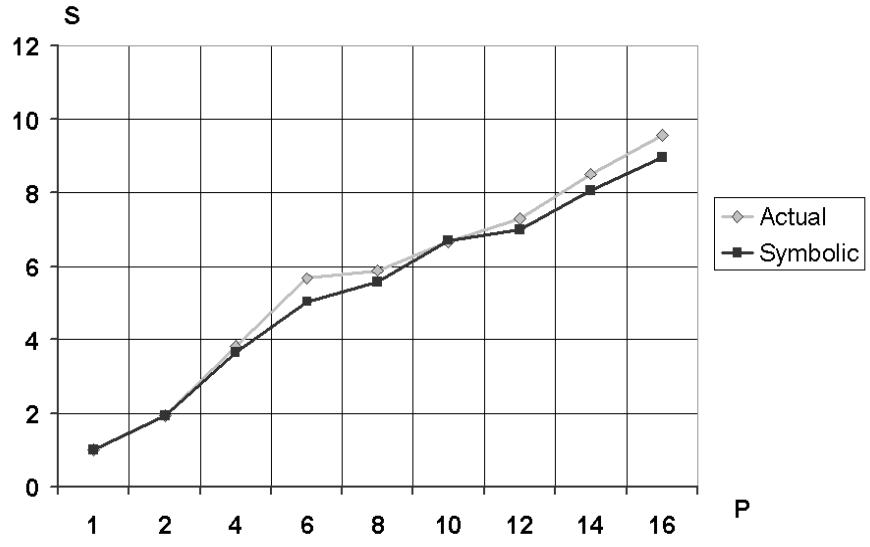

Рис. 6. Сравнение ускорения программы, вычисленного с помощью символьной интерпретации программы (чёрная кривая), с ускорением, полученным при ее фактическом выполнении (серая кривая).

Сравнение кривых показывает, что символьная интерпретация позволяет достаточно точно оценить характеристики параллельной программы. Ее преимущество перед фактическим выполнением в том, что вся чёрная кривая была получена на одном инструментальном процессоре, а для получения серой кривой потребовалось многократно выполнить ее на кластере. Таким образом, механизм символьной интерпретации позволяет сократить время, требуемое для «доводки» программы, так как инструментальный процессор значительно доступнее кластера.

На Рис. 7 время выполнения параллельной программы в среде ParJava, где она разрабатывается на языке Java с использованием библиотеки MPI (чёрная кривая) сравнивается с временем ее выполнения в системе программирования C+MPI (серая кривая). Из графиков видно, что параллельная программа на языке Java выполняется всего в полтора-два раза медленнее, чем аналогичная программа на языке C. Это означает, что существующее соотношение между временами выполнения последовательных программ на языках Java и C в среде ParJava сохраняется и для параллельных программ. Таким образом, график показывает, что реализация интерфейса MPI в среде ParJava действительно обеспечивает требуемое качество коммуникаций при выполнении параллельных программ.

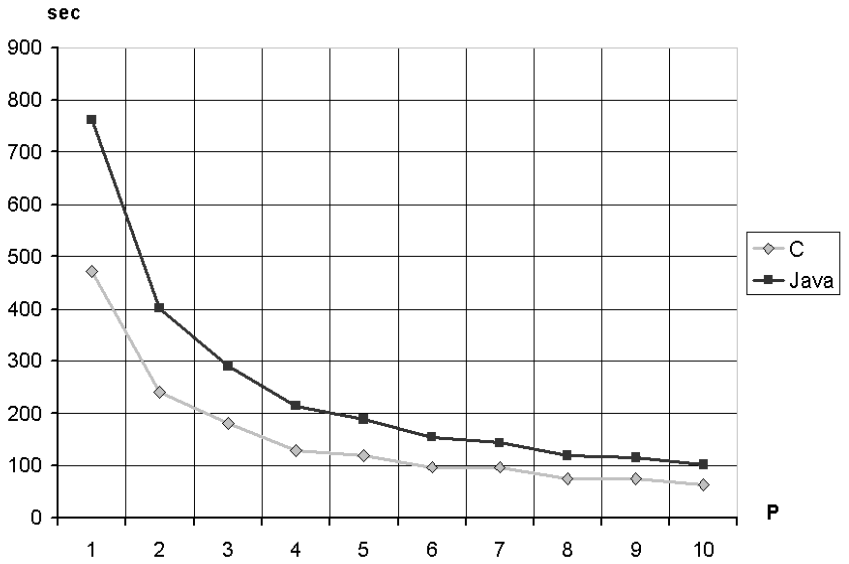

Рис. 7. Сравнение времени выполнения параллельной программы в среде ParJava (Java+MPI – чёрная кривая) и времени ее выполнения на C+MPI (серая кривая).

В качестве второго примера рассматривалась программа решения системы линейных алгебраических уравнений $Ax = b$ итерационным методом Якоби.

На Рис. 8 представлены графики зависимости ускорения от числа узлов параллельной вычислительной системы. Видно, что с увеличением порядка матрицы область масштабируемости расширяется. Вычисления выполнялись на высокопроизводительном кластере ИСП РАН (см. выше).

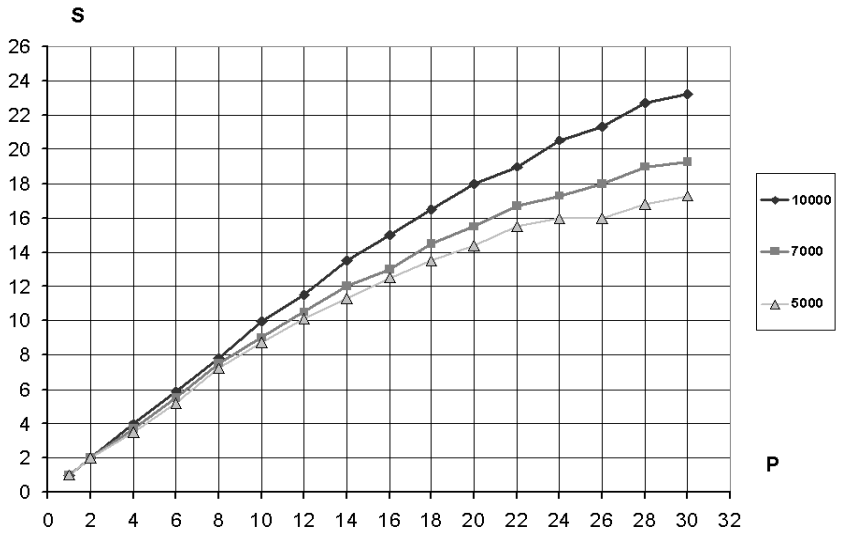

Рис. 8. Зависимость ускорения от числа узлов параллельной вычислительной системы для матриц различного порядка.

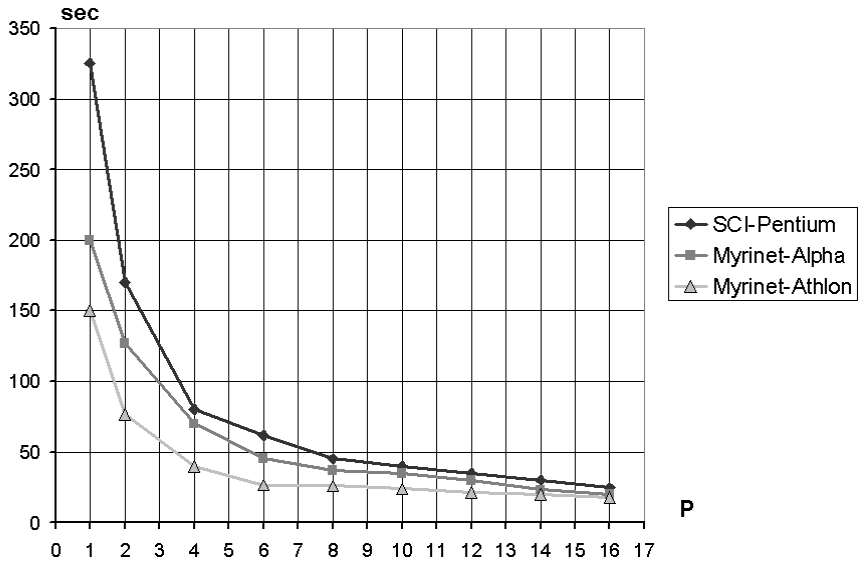

Рис. 9. Переносимость параллельной программы, разработанной в среде ParJava.

На Рис. 9 демонстрируется переносимость параллельной программы, разработанной в среде ParJava. Видно, что при выполнении на разных кластерах время счета ведет себя сходным образом. Для получения этих графиков были использованы кластеры:

- 18-ти узловой кластер Pentium (500MHz, два процессора на узле) с коммуникационной сетью SCI,
- 16-ти узловой кластер Alpha 21264 (667MHz, два процессора на узле) с коммуникационной сетью Myrinet,
- 16-ти узловой кластер Athlon XP (1533MHz, два процессора на узле) с коммуникационной сетью Myrinet.

7. Близкие работы

В работе рассмотрены следующие две проблемы: 1) применение средств анализа программ (как статического, так и динамического) для разработки, «доводки» и модификации параллельных программ; 2) обеспечение возможности разработки программ, параллельных по данным, в окружении Java.

Первая проблема актуальна не только для разработки параллельных программ в окружении Java. Опубликовано несколько работ по средам, поддерживающим разработку параллельных программ на языках C/C++ и Fortran. Можно отметить такие исследовательские проекты, как AIMS (NASA Advanced Supercomputing Division) [14], TAU (University of Oregon) [15], Pablo (University of Illinois) [16], а также коммерческие системы Vampir (Pallas GmbH) [17], PGPROF (The Portland Group™ Compiler Technology) [18] и др.

Большая часть этих систем ограничивается сбором статистики и построением профилей, проводя анализ параллельной программы после ее выполнения (post-mortem analysis). Наиболее близкой к ParJava является система AIMS.

Сравнивая символьную интерпретацию параллельной программы в системе AIMS и в среде ParJava, можно отметить, что иерархическая модель программы, принятая в ParJava, более точно отражает специфику параллельных вычислений и допускает поэтапную частичную интерпретацию, что позволяет применять ее для моделирования высокопроизводительных распределенных вычислительных систем (кластеров) с большим числом узлов. Иерархическая модель позволяет менять уровень детализации интерпретируемой программы, позволяя в случае необходимости повысить точность интерпретации.

Кроме того, в среде ParJava в отличие от AIMS более точно решена проблема символьной интерпретации циклов. В системе AIMS цикл рассматривается как целый базовый блок, тело цикла не интерпретируется, а заменяется набором атрибутов, вычисляемых в результате статического анализа тела цикла. В среде ParJava атрибуты тела цикла вычисляются в результате его автономной интерпретации, что дает гораздо более точные результаты, особенно для гнезд циклов.

Помимо этого символьная интерпретация в среде ParJava допускает генерацию трасс интерпретируемой программы, ее мониторинг, определение тупиковых ситуаций.

Что касается второй проблемы, то все больше исследований посвящается параллельному программированию в окружении Java. Большая часть этих работ связана с реализациями интерфейса MPI для окружения Java как с помощью «привязки» к реализации MPI на языке C [19], так «чистые» реализации на языке Java [20]. Преимущество реализаций MPI, принятое в среде ParJava уже обсуждалось в разделе 2.1.

Необходимо отметить несколько работ, связанных с введением в язык Java средств высокоуровневого параллельного программирования (в стиле HPF). Наиболее известны работы HPJava [21] и Titanium [22]. К сожалению, эти работы обладают всеми недостатками HPF, которые, как известно, не позволяют достичь достаточной степени масштабируемости программ из-за высоких накладных расходов на организацию коммуникаций.

8. Заключение

В работе рассмотрена интегрированная среда ParJava, поддерживающая разработку и сопровождение программ, параллельных по данным. Показано, что для программ, параллельных по данным (в частности, SPMD-программ) среда ParJava помогает прикладному программисту обеспечить такие свойства разрабатываемой (модифицируемой) программы, как ее эффективность и масштабируемость, помогает осуществлять дополнительную «доводку» программы, во время которой в программе обнаруживаются и устраняются «узкие места», мешающие достичь необходимого уровня ее масштабируемости. «Доводке» программы существенно помогает выявление ее динамических свойств (профилей, трасс, слайсов и т.п.).

В среде ParJava удалось реализовать оригинальный механизм символьной интерпретации, которая позволяет достаточно точно оценить характеристики разрабатываемой параллельной программы. Ее преимущество перед фактическим выполнением в том, что символьная интерпретация позволяет получить достаточно точные оценки динамических свойств разрабатываемой прикладной программы на персональном компьютере (рабочей станции), а не путем многократного выполнения разрабатываемой программы на целевой вычислительной системе (кластере). Это дает возможность прикладному программисту использовать свой персональный компьютер не только для предварительной отладки, но и для «доводки» параллельной программы, что существенно удешевляет и ускоряет ее разработку (или модификацию).

Среда ParJava продолжает развиваться, в нее постоянно включаются новые инструменты.

Интегрированная среда ParJava установлена на вычислительных ресурсах ИСП РАН, ГУ МСЦ и НИВЦ МГУ. Среда ParJava используется в учебном процессе

кафедр системного программирования факультета ВМиК МГУ и факультета ПМЭ МФТИ.

Достоинством среды ParJava является также то, что параллельная программа, разработанная при помощи этой среды, может выполняться на любой масштабируемой вычислительной системе без каких-либо модификаций или преобразований, с сохранением условия масштабируемости. Это снимает многие проблемы по распространению параллельных программ.

Литература

1. Victor Ivannikov, Serguei Gaissaryan, Arutyun Avetisyan, Vartan Padaryan. Improving properties of a parallel program in ParJava Environment // The 10th EuroPVM/MPI conference, Venice, Sept. 2003, LNCS v. 2840, pp 491-494.
2. D. Loveman. High-Performance Fortran. //IEEE Parallel and Distributed Technology, v. 1(1), 1993.
3. Java™ 2 Platform, Standard Edition, v 1.4.1. API Specification. http://java.sun.com/j2se/1.4.1/docs/api/index.html.
4. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. // Proc. AFIPS Conference, Reston, VA, vol. 30, pp. 483-485, April 1967.
5. E. Walker. Extracting data flow information for parallelizing FORTRAN nested loop kernels. / Submission for the degree of Doctor of Philosophy. Department of Computer Science, Heslington, the University of York, England. 1994.
6. A. W. Lim, G. I. Cheong, M. S. Lam. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. // In Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, June 1999.
7. J. L. Gustafson. Reevaluating Amdahl's law. // Communications of the ACM, vol. 31, no. 5, pp. 532-533, May 1988.
8. K. Arvind. Probabilistic Clock Synchronization in Distributed Systems. // IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 5, May 1994.
9. R. Gupta and M. Spezialetti. Dynamic Techniques for Minimizing the Intrusive Effect of Monitoring Actions. // International Conference on Distributed Computing Systems, 1995. pp. 368-376.
10. R. Martin, A. Vahdat, D. Culler, T. Anderson. The Effects of Latency, Overhead and Bandwidth in a Cluster of Workstations. //In Proc. 24th Int. Symp. on Com. Arch. (ISCA'97), 1997, pp 85 – 97.
11. OMIS 2.0 -- On-line Monitoring Interface Specification. http://wwwbode.cs.tum.edu/~omis/.
12. K. Fürlinger and M. Gerndt. Distributed configurable application monitoring on SMP clusters. // Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface, Venice, 2003, p. 429-437.
13. S. D. Ustyugov and V. M. Chechetkin. Supernovae Explosions in the Presence of Large-scale Convective Instability in a Rotating Protoneutron Star. // Astronomy Reports, 1999, vol. 43, №11, p. 718-727.
14. J. C. Yan. Performance Tuning with AIMS – An Automated Instrumentation and Monitoring System for Multicomputers. // Proceedings of the 27th Hawaii International Conference on System Sciences, Wailea, Hawaii, January, 4 - 7, 1994. Vol. II. pp 625-633.

15. Sameer Shende and Allen D. Malony. Integration and application of the TAU performance system in parallel java environments. // Proceedings of the Joint ACM Java Grande - ISCOPE 2001 Conferenc, June 2001.
16. Luiz DeRose and Daniel A. Reed. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. // Proceedings of the International Conference on Parallel Processing (ICPP'99), Fukushima, Japan, September 1999.
17. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. // Supercomputer, 12(1): 69--80, January 1996.
18. B. Carpenter, G. Zhang, G. Fox, Xiaoming Li, Xinying Li, and Y. Wen. Towards a Java environment for SPMD programming. // D. Pritchard and J. Reeve, (eds.), 4th Intern. Europar Conf., LNCS vol. 1470, 1998.
19. S. Mintchev. Writing Programs in JavaMPI. // TR MAN-CSPE-02, Univ. of Westminster, UK, 1997.
20. Tong WeiQin, Ye Hua, Yao WenSheng. PJMPI: pure Java implementation of MPI. // Proceedings of the 4th International Conference on High Performance Computing in the Asia-Pacific Region, 2000.
21. B. Carpenter. Elements of the HPJava Language. //The HPJava Project. http://www.npac.syr.edu/projects/pcrc/HPJava/elements.html.
22. K. Yelick, L. Semenzato, G.Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, A. Aiken. Titanium: A High-Performance Java Dialect. // ACM 1998 Workshop on Java for High-Performance Network Computing. http://www.cs.ucsb.edu/conferences/java98.