Обеспечение высокопродуктивного программирования для современных параллельных платформ

А.И. Аветисян, В.В. Бабкова, А.В. Монаков
{arut, barbara, amonakov}@ispras.ru
http://www.ispras.ru/groups/ctt/parjava.html

Аннотация. В настоящей статье описываются некоторые перспективные направления исследований по высокопродуктивному программированию для параллельных систем с распределенной памятью, проводимые в отделе компиляторных технологий Института системного программирования РАН. Обсуждаются текущие исследования и направления будущих работ, связанных с высокопродуктивным программированием многоядерных и гетерогенных систем.

1. Введение

Развитие компьютерных и сетевых технологий привело к тому, что одним из основных свойств современных вычислительных систем является параллелизм на всех уровнях. Происходит широкое внедрение кластерных систем (распределенная память) с тысячами процессоров. Началось широкое производство многоядерных процессоров общего назначения, Современные многоядерные процессоры имеют не более 16 ядер, однако производители уже серьезно говорят о нескольких сотнях и даже тысячах ядер [1]. Кроме того, выпускаются специализированные процессоры, содержащие сотни параллельно работающих ядер на одном чипе (графические акселераторы компаний *AMD* и *nVidia*). Высокая производительность, низкое энергопотребление и низкая стоимость специализированных многоядерных процессоров (как правило, это процессоры для компьютерных игр) способствовали стремлению использовать их не только по их прямому назначению. Начались исследования возможностей широкого применения гетерогенных архитектур, состоящих из процессора общего назначения и набора специализированных многоядерных процессоров (акселераторов) для решения вычислительных задач общего назначения. Акселератор имеет доступ как к своей собственной памяти, так и к общей памяти гетерогенной системы. Примерами таких архитектур являются: архитектура *IBM Cell*, архитектуры, использующие графические акселераторы компаний *AMD* и *nVidia,* многоядерный графический ускоритель *Larrabee* компании *Intel*.

Остро встал вопрос о языках параллельного программирования, которые могли бы обеспечить достаточно высокую производительность труда программистов, разрабатывающих параллельные приложения. Однако языки, разработанные в 90-е годы (*HPF* [2], *UPC* [3] и др.) не смогли решить эту проблему [4]. Это привело к тому, что промышленную разработку прикладных параллельных программ, обеспечивающих необходимое качество, приходится вести, на так называемом «ассемблерном» уровне, на последовательных языках программирования (*C/C++*, *Fortran*), разработанных в 60-70 гг., с явным использованием обращений к коммуникационной библиотеке *MPI* (для систем с распределенной памятью), явным указанием прагм *OpenMP* (для систем с общей памятью), с использованием технологии программирования CUDA [5] (расширение языка C для акселераторов *Nvidia*), которая точно отражает организацию оборудования, что позволяет создавать эффективные программы, но требует высокого уровня понимания архитектуры акселератора и др.

Таким образом, в настоящее время параллельное программирование связано с ручной доводкой программ (распределение данных, шаблоны коммуникаций, либо синхронизации доступа к критическим данным и т.п.). Это связано со значительными затратами ресурсов и требует высокой квалификации прикладных программистов. Цена, которую нужно заплатить, чтобы добиться хорошей производительности и требуемой степени масштабируемости приложений, часто оказывается непомерно высокой. Поэтому целью современных исследований является фундаментальная проблема высокой продуктивности [6] разработки параллельных приложений, когда обеспечивается достаточный уровень производительности при приемлемом уровне затрат на разработку. Это особенно актуально в связи с тем, что параллельное программирование становиться массовым.

Исследования ведутся по многим направлениям: изучаются свойства приложений, делаются попытки классификации приложений, в том числе для выявления в них общих ядер; исследуются свойства аппаратуры с целью максимального их использования и развития; ведутся исследования и разработки по целому спектру средств программирования.

Одним из направлений исследований является разработка языков нового поколения (*X10* [7], *Chapel* [8], *Fortress* [9], *Cilk* [10], Brook+ [11] и др.). Несмотря на то, что эти разработки опираются на опыт предыдущих лет, пока они не привели к успеху, прежде всего, из-за недостаточного уровня современных компиляторных технологий.

Реализуются как промышленные, так и исследовательские, системы, поддерживающие доводку программ разрабатываемых на «ассемблерном» уровне. К настоящему времени известно несколько таких систем: отладчики DDT [12], TotalView [13], система TAU [14], разработанная в университете штата Орегон и др.

Одним из таких средств является интегрированная среда *ParJava* [15], разработанная в ИСП РАН, которая предоставляет прикладному программисту набор инструментальных средств, поддерживающих разработку параллельных программ для вычислительных систем с распределенной памятью (высокопроизводительных кластеров) на языке *Java*, расширенном стандартной библиотекой передачи сообщений *MPI*.

В настоящее время среда Java представляет значительный интерес с точки зрения высокопроизводительных вычислений. Это связано как с положительными свойствами Java как среды разработки прикладных программ (переносимость, простота отладки и др.), так и с тем, что использование инфраструктуры Java существенно упрощает разработку инструментальных средств. Можно упомянуть такие системы как: ProActive Parallel Suite [16] (*INRIA*), MPJ Express [17] (University of Reading and University of Southampton), Distributed Parallel Programming Environment for Java [18] (*IBM*) и др. Кроме того, добавлена поддержка *Java + MPI* в известной среде разработки параллельных программ на языках *C/C++* и *Fortran* 77/90 TAU.

В проекте *ParJava* решались две задачи: обеспечить возможность эффективного выполнения параллельных программ на языке Java с явными обращениями к MPI на высокопроизводительных кластерных системах и разработать технологический процесс реализации параллельных программ, обеспечивающий возможность переноса как можно большей части разработки на инструментальный компьютер.

В настоящей статье описываются некоторые перспективные направления исследований по высокопродуктивному программированию для параллельных систем с распределенной памятью, проводимые в отделе компиляторных технологий Института системного программирования РАН. Обсуждаются текущие исследования и направления будущих работ, связанных с высокопродуктивным программированием многоядерных и гетерогенных систем.

Статья состоит из 4 разделов. В разделе 2 описывается среда *ParJava*, модель параллельной *Java*-программы и возможности ее интерпретации, технологический процесс разработки программ в среде ParJava, механизмы времени выполнения. В разделе 3 приводятся результаты применения среды при разработке программ моделирования интенсивных атмосферных вихрей (ИАВ) и моделирования теплового движения молекул воды и ионов в присутствии фрагмента ДНК. В разделе 4 обсуждаются направления дальнейших исследований.

2. Средства разработки параллельных приложений в среде Java

Среда *ParJava* позволяет выполнять большую часть разработки параллельной Java-программы на инструментальном компьютере. Для этого используется модель параллельной *Java*-программы [19], интерпретируя которую на

инструментальном компьютере можно получить оценки времени выполнения программы на заданном кластере (кластер определяется числом узлов, параметрами платформы, используемой в качестве его узлов, и параметрами его коммуникационной сети), а также оценки других динамических атрибутов программы, построить модели ее профилей и трасс. Полученная информация о динамических свойствах параллельной программы позволяет оценить границы ее области масштабируемости, помогает прикладному программисту вручную оптимизировать программу, проверяя на интерпретаторе, как отразились произведенные изменения на ее масштабируемости. Возможность использования инструментального компьютера для оптимизации и доводки параллельной программы избавляет программиста от большей части отладочных запусков программы на целевой вычислительной системе, сокращая период отладки и доводки программы.

2.1. Модель параллельной *Java*-программы и ее интерпретация

Модель SPMD-программы представляет собой совокупность моделей всех классов, входящих в состав моделируемой программы. *Модель* каждого *класса* – это множество моделей всех методов этого класса; кроме того, в модель класса включается модель еще одного дополнительного метода, описывающего поля класса, его статические переменные, а также инициализацию полей и статических переменных. *Модель* метода (функции) состоит из списка описаний локальных и глобальных переменных метода и модифицированного абстрактного синтаксического дерева метода: внутренние вершины модели соответствуют операторам языка *Java*, а листовые – базовым блокам. В качестве базовых блоков рассматриваются не только линейные последовательности вычислений (вычислительные базовые блоки), но также и вызовы библиотечных функций, вызовы пользовательских методов и функций и вызовы коммуникационных функций. Для обеспечения возможности интерпретации модели по частям введено понятие *редуцированного блока*, который представляет значения динамических атрибутов уже проинтерпретированных частей метода.

Каждый *MPI*-процесс моделируемой программы представляется в ее модели с помощью *логического процесса*, который определен как последовательность *действий* (примеры действий: выполнение базового блока, выполнение операции обмена и т.п.). Каждое действие имеет определенную *продолжительность*. В логическом процессе определено понятие *модельных часов*. Начальное показание модельных часов каждого логического процесса равно нулю. После интерпретации очередного действия к модельным часам соответствующего логического процесса добавляется значение времени, затраченного на выполнение этого действия (продолжительности). Продолжительность каждого действия, а также значения исследуемых динамических параметров базовых блоков, измеряются заранее на целевой платформе.

Для идентификации логических процессов используются номера моделируемых процессов, использованные в моделируемой программе при описании коммуникатора *MPI* (будем называть их *пользовательскими* номерами процессов). Как известно, для удобства программирования приложений в стандарте *MPI* реализована возможность задавать *коммуникаторы*, позволяющие задавать виртуальные топологии сети процессоров, описывая группы процессов. В среде *ParJava* коммуникаторы, заданные программистом, отображаются на **`MPI_COMM_WORLD`**, получая, тем самым, наряду с пользовательскими номерами *внутренние* номера. Все инструменты среды *ParJava* работают с внутренними номерами процессов, но при выдаче сообщений пользователю эти номера переводятся в пользовательские. В дальнейшем, при упоминании номера логического процесса будет подразумеваться его внутренний (системный) номер. Внутренний номер используется для доступа к логическому процессу при моделировании коммуникационных функций.

Для сокращения времени интерпретации и обеспечения возможности выполнения интерпретации на инструментальном компьютере в среде *ParJava* моделируются только потоки управления процессов моделируемой программы и операции обмена данными между процессами. Это допустимо, так как значения времени выполнения и других исследуемых динамических атрибутов базовых блоков определяются на целевой вычислительной системе до начала интерпретации модели. Интерпретация модели лишь распространяет значения указанных динамических атрибутов на остальные узлы модели. Такой подход позволяет исключить из моделей базовых блоков переменные, значения которых не влияют на поток управления моделируемой программы. В результате часть вычислений, в которых определяются и используются указанные переменные, становится «мертвым кодом» и тоже исключается, что ведет к сокращению, как объема обрабатываемых данных, так и общего объема вычислений во время интерпретации. В некоторых базовых блоках описанный процесс может привести к исключению всех вычислений, такие базовые блоки заменяются редуцированными блоками.

Внутреннее представление модели *SPMD*-программы разрабатывалось таким образом, чтобы обеспечить возможность возложить как можно большую часть функций интерпретатора на *JavaVM*. Такое решение позволило существенно упростить реализацию интерпретатора и обеспечить достаточно высокую скорость интерпретации, однако для его реализации потребовалось внести некоторые структурные изменения в модель параллельной программы.

Внутреннее представление модели базового блока B представляет собой пару $\langle Descr_B, Body_B \rangle$, где $Descr_B$ – дескриптор блока B (т.е. семерка $\langle id, \tau, P, I_C, O_C, Time, A \rangle$, где id – уникальный идентификатор базового блока, присваиваемый ему при построении модели, τ – тип базового блока, P – ссылка на модель его тела, I_C – множество входных управляющих переменных, O_C – множество выходных управляющих переменных, $Time$ – время выполнения базового

блока, A – ссылка на список остальных его атрибутов), а $Body_B$ – модель тела блока B (список выражений на байт-коде, вычисляемых в блоке B).

Внутреннее представление модели метода определяется как тройка ⟨*дескриптор метода*, *модель потока управления, модель вычислений*⟩. Дескриптор метода содержит сигнатуру метода, список генерируемых методом исключений, список дескрипторов формальных параметров и ссылки на модель потока управления и модель вычислений. Модель потока управления – это модифицированное АСД, описанное в [19], в котором базовые блоки представлены своими дескрипторами. Модель вычислений – это преобразованный байт-код интерпретируемой *Java*-программы: все базовые блоки модели вычислений включены в состав переключателя, значение управляющей переменной которого определяет номер очередного интерпретируемого базового блока.

Интерпретация модели состоит в выполнении *Java*-программы, определяющей модель вычислений на *JavaVM*: в интерпретаторе модели потока управления определяется очередное значение управляющей переменной переключателя модели вычислений, после чего интерпретируется модель соответствующего базового блока. Интерпретация модели базового блока определяется его типом. Для блоков типа «вычислительный блок» (время выполнения таких базовых блоков определяется заранее на целевой платформе) вносится соответствующее изменение во временной профиль метода, и управление возвращается в модель вычислений. Для блоков типа «вызов пользовательской функции», управление возвращается в модель вычислений, где вызывается модель вычислений этого метода. Во время выполнения вызванной модели вычислений в стек помещается текущее состояние, и подгружаются необходимые структуры данных. После интерпретации метода из стека извлекается состояние на момент вызова пользовательской функции и продолжается выполнение модели вычислений первой функции. Для блоков типа «вызов коммуникационной функции», управление возвращается в модель вычислений, где вызывается модель соответствующей коммуникационной функции, которая помимо выполнения передачи данных между логическими процессами обеспечивает вычисление оценки времени выполнения коммуникации и внесение соответствующих изменений во временной профиль. При интерпретации блоков типа «редуцированный блок» (динамические атрибуты таких блоков уже определены и они не интерпретируются), возврат в модель вычислений не происходит; вносятся изменения во временной профиль, и выполняется поиск следующего базового блока.

2.2. Технологический процесс разработки параллельных программ в среде ParJava

В рамках среды ParJava разработан и реализован ряд инструментальных средств, которые интегрированы с открытой средой разработки Java-программ Eclipse [20]. После подключения этих инструментальных средств к Eclipse,

получилась единая среда разработки SPMD-программ, включающая как инструменты среды ParJava, так и инструменты среды Eclipse: текстовый редактор, возможность создания и ведения проектов, возможность инкрементальной трансляции исходной программы во внутреннее представление. Интеграция в среду *Eclipse* осуществлена с помощью механизма «подключаемых модулей».

При разработке параллельной программы по последовательной программе сначала оценивается доля последовательных вычислений, что позволяет (с помощью инструмента "Speed-up") получить оценку максимального потенциально достижимого ускорения в предположении, что все циклы, отмеченные прикладным программистом, могут быть распараллелены. Затем с использованием инструмента "Loop Analyzer" среды *ParJava* циклы исследуются на возможность их распараллелить.

Для распараллеленных циклов с помощью инструмента "DataDistr" подбирается оптимальное распределение данных по узлам вычислительной сети. В частности, для гнезд циклов, в которых все индексные выражения и все границы циклов заданы аффинными формами, инструмент "DataDistr" позволяет с помощью алгоритма [21] найти такое распределение итераций циклов по процессорам, при котором не требуется синхронизаций (обменов данными), если, конечно, требуемое распределение существует (см. ниже пример 1). В этом случае инструмент "DataDistr" выясняет, можно ли так распределить итерации цикла по узлам, чтобы любые два обращения к одному и тому же элементу массива попали на один и тот же процессор. Для этого методом ветвей и сечений находится решение задачи целочисленного линейного программирования, в которую входят ограничения на переменные циклов, связанные с необходимостью оставаться внутри границ циклов, и условия попадания на один и тот же процессор для всех пар обращений к одинаковым элементам массива. Задача решается относительно номеров процессоров, причем для удобства процессоры нумеруются с помощью аффинных форм (т.е. рассматривается многомерный массив процессоров).

Если оказывается, что для обеспечения сформулированных условий все данные должны попасть на один процессор, это означает, что цикл не может выполняться параллельно без синхронизации. В последнем случае инструмент "DataDistr" может в диалоговом режиме найти распределение данных по узлам, требующее минимального числа синхронизаций при обменах данными. Для этого к условиям сформулированной задачи линейного программирования добавляются условия на время обращений к одним и тем же элементам массива: например, в случае прямой зависимости, требуется, чтобы обращение по записи выполнялось раньше, чем обращение по чтению. В частности, при решении дополнительных временных ограничений, может оказаться, что они могут быть выполнены, если обрабатываемые в программе массивы будут разбиты на блоки. При этом смежные блоки должны быть распределены по процессорам «с перекрытием», чтобы все необходимые

данные были на каждом из процессоров. При этом возникают так называемые теневые грани (т.е. части массива, используемые на данном процессоре, а вычисляемые на соседнем процессоре). Ширина теневых граней определяется алгоритмом решения задачи и определяет фактический объем передаваемых в сети сообщений. Количество теневых граней зависит выбора способа нумерации процессоров: априорно выгоднее всего, чтобы размерность массива процессоров совпадала с размерностью обрабатываемых массивов данных. Однако в некоторых случаях оказывается более выгодным, чтобы размерность массива процессоров была меньше, чем размерность обрабатываемых массивов данных.

Пример 1. В качестве примера работы инструмента "DataDistr" рассмотрим цикл:

```
for (i = 1; i <= 100; i++)
for(j = 1; j <=100; j++){
      X[i,j] = X[i,j] + Y[i – 1,j]; /*s1*/
      Y[i,j] = Y[i,j] + X[i,j - 1]; /*s2*/
      }
```

Для приведенного примера инструмент "DataDistr" выдаст следующее распределение:

```
X[1,100] = X[1,100] + Y[0,100];                /*s1*/
for (p = -99; p <= 98; p++){
 if (p >= 0)
  Y[p+l,l] = X[p+l,0] + Y[p+l,l];              /*s2*/
  for (i = max(l,p+2); i <= min(100,100+p); i++) {
    X[i,i-p-l] = X[i,i-p-l] + Y[i-l,i-p-l]; /*s1*/
    Y[i,i-p] = X[i,i-p-l] + Y[i,i-p];         /*s2*/
    }
  if (p <= -1)
   X[101+p,100]=X[101+p,100]+Y[101+p-l,100];/*s1*/
}
Y[100,l] = X[100,0] + Y[100,l];                /*s2*/
```

где p – номер вычислителя, а цикл по p определяет распределение данных по вычислителям. Таким образом, исходный цикл расщепился на 200 цепочек вычислений, каждая из которых может выполняться независимо.

После того, как данные распределены по вычислителям, прикладному программисту необходимо выбрать такие операции обмена данными и так трансформировать свою программу, чтобы добиться максимально возможного перекрытия вычислений и обменов данными. Среда *ParJava* позволяет решать этот вопрос в диалоговом режиме, используя интерпретатор параллельных программ (инструмент "Interpreter").

В тривиальных случаях даже использование стандартных коммуникационных функций (`MPI_send`, `MPI_receive`) позволяет достичь достаточного уровня масштабируемости. Однако, в большинстве случаев это невозможно, так как приводит к большим накладным расходам на коммуникации, а это в соответствии с законом Амдаля существенно урезает масштабируемость. Достичь перекрытия вычислений и обменов для некоторых классов задач позволяет использование коммуникационных функций `MPI_Isend`, `MPI_Ireceive` совместно с функциями `MPI_Wait` и `MPI_Test`. Это поясняется примером 2.

Пример 2. Как видно из схемы на рис. 1, необходимо добиться, чтобы во время передачи данных сетевым процессором (промежуток между моментами времени t_1^s и t_2^s) вычислитель был занят обработкой данных, а не простаивал в ожидании окончания пересылки.

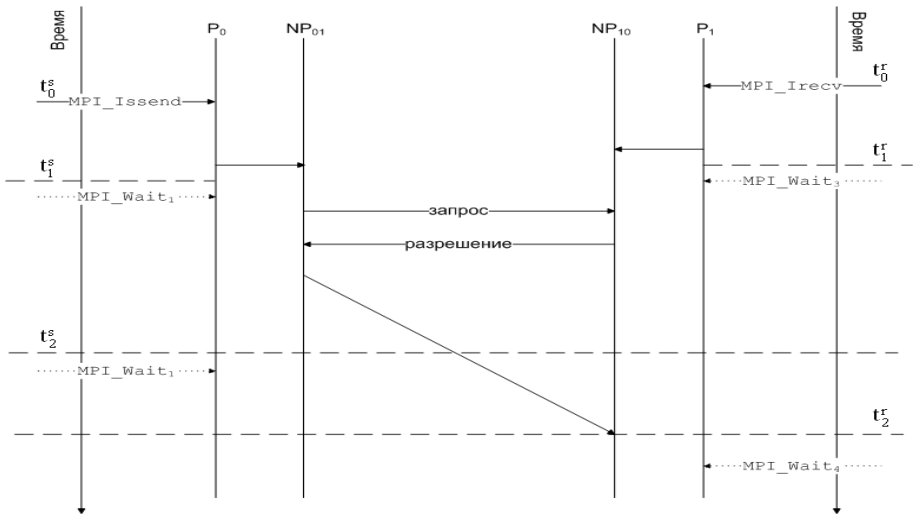

Рис. 1. Схема передачи данных MPI

Подбор оптимальных коммуникационных функций требует многочисленных интерпретаций различных версий разрабатываемой программы. Для ускорения этого процесса строится «скелет» программы и все преобразования делаются над ним. После достижения необходимых параметров параллельной программы автоматически генерируется вариант программы в соответствии с полученным коммуникационным шаблоном.

Проиллюстрировать важность оптимального выбора операций пересылок можно на следующем «скелете» реальной программы моделирования торнадо

(рис. 2а). Если в рассматриваемом «скелете» заменить блокирующие операции *Send* и *Recv* на неблокирующие и установить соответствующую операцию *Wait* после гнезда циклов получится «скелет», представленный на рис. 2б. На рис. 3 приведены графики ускорения «скелетов» программы представленного на рис. 2а и 2б. Как видно, такая замена существенно расширила область масштабируемости программы.

При этом окончательная версия «скелета», удовлетворяющая требованиям прикладного программиста, используется для построения трансформированной исходной программы.

```
//sending
Send
Recv
//calculating
for (i = beg_i; i < end_i;
i++)
    for (j = 0; j < N; j++)
        B[i][j] = f(A[i][j]);
```

(а)

```
//sending
ISend
IRecv
//calculating
for (i = beg_i + 1; i < end_i - 1;
i++)
    for (j = 0; j < N; j++)
        B[i][j] = f(A[i][j]);
//waiting
Wait();
//calculating last columns
if (myid != 0)
    for (j = 0; j < N; j++)
        B[0][j] = f(tempL[j]);
if (myid != proc_size - 1)
    for (j = 0; j < N; j++)
        B[N - 1][j] = f(tempR[j]);
```

(б)

Рис. 2. Схематичное изображение алгоритмов с блокирующими и неблокирующими пересылками

К сожалению, в большинстве реальных программ такими простыми преобразованиями не удается достичь необходимого уровня перекрытия, либо такое преобразование невозможно. Использование предлагаемой технологии обеспечивает возможность применения различных преобразований программы, и достигать необходимых параметров программы в приемлемое время.

Этот процесс отладки и оптимизации параллельной программы показывает, что задача сама по себе неформализована. На сегодняшний день не может

быть реализован компилятор, делающий оптимизацию автоматически, т.к. в некоторых точках процесса программист обязан принимать волевые решения.

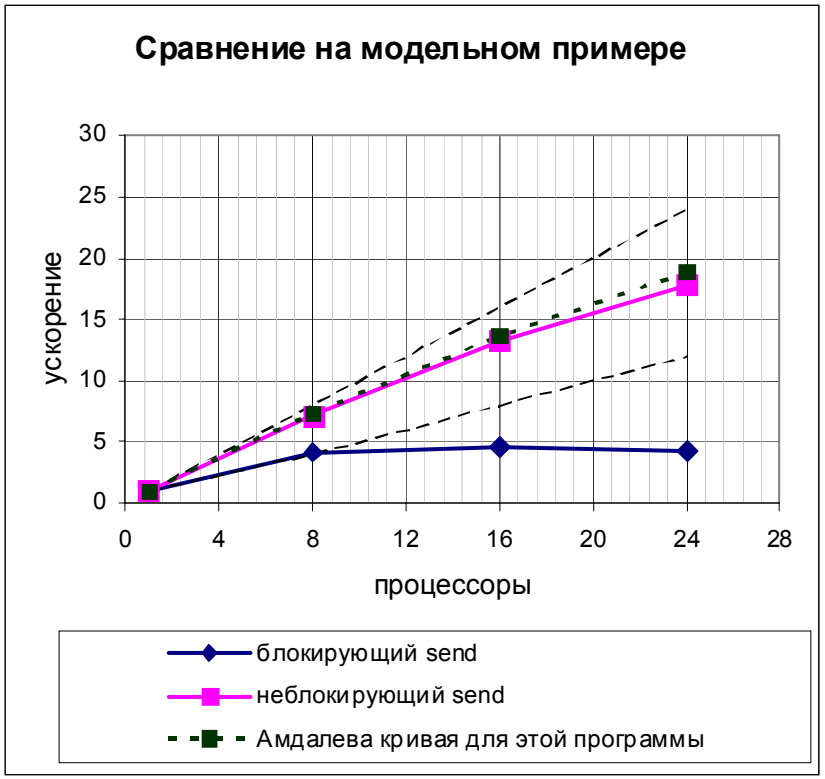

Рис. 3. Сравнение масштабируемости параллельных программ, использующих блокирующие и неблокирующие пересылки

На следующем этапе, необходимо проинтерпретировать полученную программу на реальных данных, для того чтобы оценить, какое количество вычислителей будет оптимальным для счета. Для этого снова используется инструмент «Inerpreter». Поскольку интерпретатор использует смешанную технику, включающую в себя элементы прямого выполнения, довольно остро стоит проблема нехватки памяти на инструментальной машине. Моделирование некоторых серьёзных программ требует использования довольно больших массивов данных, которые не могут быть размещены в памяти одного вычислительного узла. Для решения этой проблемы проводится преобразование модели, представляющее собой удаление выражений программы, значение которых не влияет на поток управления. Это

позволяет качественно изменить требования по памяти, в том числе существенно сократив время интерпретации.

Таким образом, инструментальные средства среды *ParJava* позволяют реализовать технологический процесс, обеспечивающий итеративную разработку параллельной программы. Отметим, что итеративная разработка предполагает достаточно большое число итераций, что может привести к большим накладным расходам по времени разработки и ресурсам. Итеративная разработка реальных программ, предлагаемая в рамках *ParJava*, возможна благодаря тому, что большая часть процесса выполняется на инструментальном компьютере, в том числе за счет использования инструментов, базирующихся на интерпретаторе параллельных программ.

Применение интерпретатора позволяет достаточно точно оценивать ускорение программы. Ошибка на реальных приложениях не превосходила 10%, и в среднем составила ~5% [36-44]. Окончательные значения параметров параллельной программы можно уточнить при помощи отладочных запусков на целевой аппаратуре.

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.

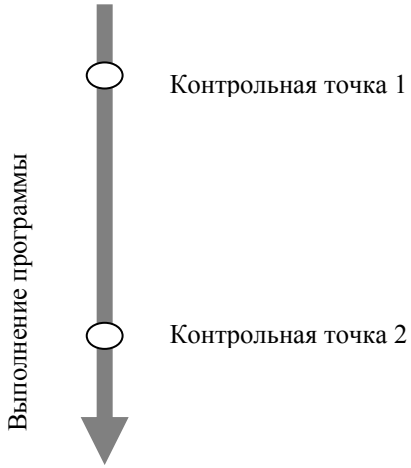

Рис. 4. Контрольные точки

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 $l \in use[S']$ и для каждого S'' на этом пути $l \notin def[S'']$. 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 $l \notin use[S]$. Поэтому $l \in RO(G')$ тогда и только тогда, когда $l \notin gen[S]$ 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:

$$out[S] = \cap_{S'} in[S']$$
$$in[S] = FS,$$

где $out[S] = \cap_{S'} DEAD[S']$ – 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 $FS = out[S] \cup gen[S] - use[S]$ 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.

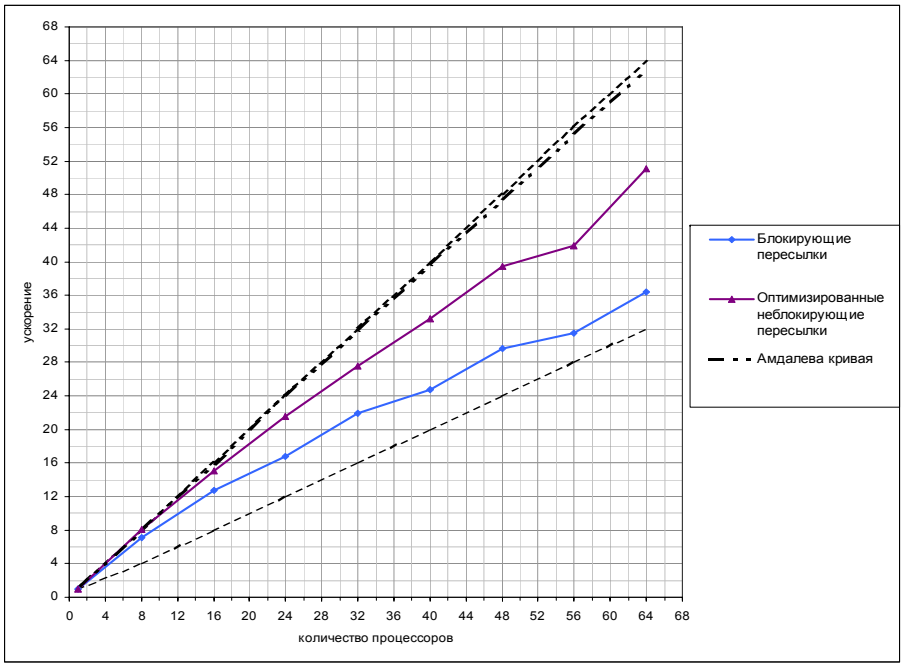

Рис. 5. График ускорения.

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].

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]. 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 x 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.

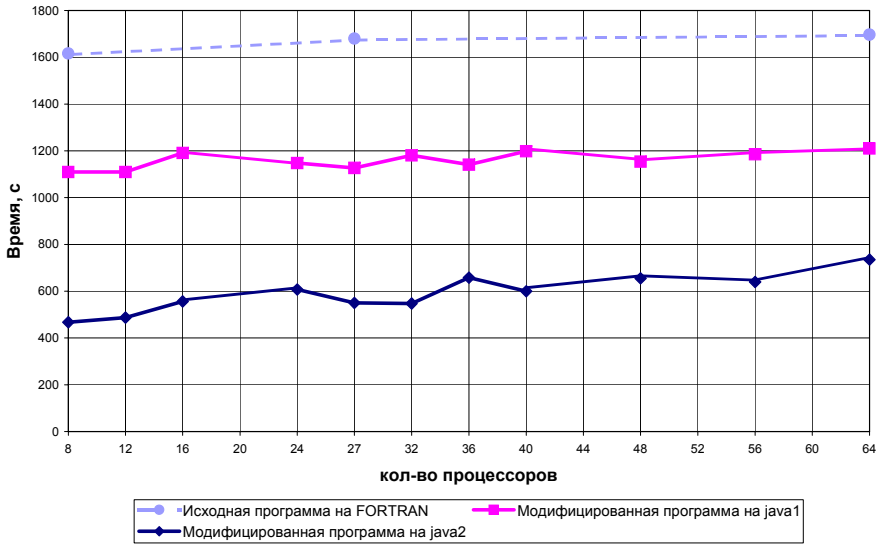

Рис. 6. Сравнение исходной программы на языке FORTRAN77 с модифицированными программами на языке Java.

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 $220\,\mathring{A}$, 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].

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.

Литература

[1] 1. David A. Patterson et al. The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View. Technical Report No. UCB/EECS-2008-23

[2] http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-23.html. March 21, 2008.

[3] 2. J.C. Adams, W.S. Brainard, J.T. Martin, B.T. Smith, J.L. Wagener. Fortran 95 Handbook. Complete ISO/ANSI Reference. Scientific and Engineering Computation Series. MIT Press, Cambridge, Massachusetts, 1997

[4] 3. W. Chen, C. Iancu, K. Yelick. Communication Optimizations for Fine-grained UPC Applications. //14th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005.

[5] 4. K. Kennedy, C. Koelbel, H. Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson // HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming,2007, San Diego, California, June 09 - 10, 2007, pp. 7-1 – 7-22

[6] 5. CUDA, среда для параллельного программирования на GPU. http://www.nvidia.com/object/cuda_home.html

[7] 6. The DARPA High Productivity Computing Systems. http://www.highproductivity.org/

[8] 7. K. Ebcioglu, V. Saraswat, V. Sarkar. X10: an Experimental Language for High Productivity Programming of Scalable Systems // Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05) Feb 13, 2005, San Francisco, USA pp. 45-52

[9] 8. B.L. Chamberlain, D. Callahan, H.P. Zima. Parallel Programmability and the Chapel Language // International Journal of High Performance Computing Applications, August 2007, 21(3): 291-312.

[10] 9. E. Allen, D. Chase, J. Hallett et al The Fortress Language Specification (Version 1.0) / cSun Microsystems, Inc., March 31, 2008 (262 pages)

[11] 10. Cilk++ Solution Overview. http://www.cilk.com/multicore-products/cilk-solution-overview/

[12] 11. Brook+ Streaming Compiler. http://ati.amd.com/technology/streamcomputing/sdkdwnld.html

[13] 12. Parallel Debugger: DDT. http://www.nottingham.ac.uk/hpc/html/docs/numerical/parallel_ddt.php

[14] 13. TotalView. http://www.totalviewtech.com/

[15] 14. Sameer S. Shende, Allen D. Malony. The TAU Parallel Performance System // The International Journal of High Performance Computing Applications,Volume 20, No. 2, Summer 2006, pp. 287–311

[16] 15. В.П. Иванников, А.И. Аветисян, С.С. Гайсарян, В.А. Падарян. Оценка динамических характеристик параллельной программы на модели. // «Программирование» 2006, №4 с. 21–37

[17] 16. Brian Amedro, Vladimir Bodnartchouk, Denis Caromel, Christian Delbé, Fabrice Huet, Guillermo L. Taboada. Current State of Java for HPC. Technical report N° 0353. August 2008.

[18] 17. Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: Towards Thread Safe Java HPC, Submitted to the IEEE International Conference on Cluster Computing (Cluster 2006), Barcelona, Spain, 25-28 September, 2006.

[19] 18. Distributed Parallel Programming Environment for Java. http://www.alphaworks.ibm.com/tech/dppej

[20] 19. В.П. Иванников, А.И. Аветисян, С.С. Гайсарян, В.А. Падарян. Прогнозирование производительности MPI-программ на основе моделей. // «Автоматика и телемеханика», 2007, №5, с. 8-17

[21] 20. Eclipse. http://www.eclipse.org/

[22] 21. A.W. Lim, M.S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. Proc. 24th ACM SIGPLAN-SIG-ACT Symposium on principles of programming languages. 1997, pp. 201-214.

[23] 22. Аветисян А.И., Бабкова В., Гайсарян С.С., Губарь А.Ю. Рождение торнадо в теории мезомасштабной турбулентности по Николаевскому. Трехмерная численная модель в ParJava. // Журнал «Математическое моделирование», 2008, №8.

[24] 23. Теплухин А. В. Многопроцессорное моделирование гидратации мезоскопических фрагментов ДНК. // Математическое моделирование, 2004г., том 16, номер 11, с.15-24.

[25] 24. Аветисян А.И., Гайсарян С.С., Калугин М.Д., Теплухин А.В. «Разработка параллельного алгоритма компьютерного моделирования водно-ионной оболочки ДНК»// Труды XIII Байкальской Всероссийской конференции «Информационные и математические технологии в науке и управлении». Часть I. - Иркутск: ИСЭМ СО РАН, 2008, с. 195-206.