УДК 004.4.233

РЕАЛИЗАЦИЯ ИНСТРУМЕНТАЛЬНОГО СРЕДСТВА АВТОМАТИЗИРОВАННОГО АНАЛИЗА ПРОИЗВОДИТЕЛЬНОСТИ UPC-ПРОГРАММ

Н. Е. Андреев[1], **К. Е. Афанасьев**[1]

Рассмотрена реализация программного инструмента анализа производительности программ, написанных на языке Unified Parallel C, который принадлежит к программной модели PGAS — новому направлению в разработке параллельных приложений. Главной особенностью инструмента по сравнению с существующими разработками является поддержка автоматизированного анализа. Для реализации инструмента использовались части пакета Scalasca.

Ключевые слова: PGAS, Unified Parallel C, Scalasca, инструментовка, трассировка, анализ производительности, параллельное программирование.

Введение. Основные методы, которые предлагаются сегодня инструментами анализа производительности параллельных программ, — это профилирование и анализ временны́х шкал. Выходной информацией профилировщика является трасса — список событий, произошедших в приложении (исполненные инструкции, вызовы функций, вызовы операционной системы). После получения трассы профилировщик обрабатывает собранную информацию и строит профиль приложения — статистическую сводку о работе программы. Затем профиль представляется в графическом виде, удобном для анализа. Разработчик изучает его, определяет проблемные места и ищет способы оптимизации своего приложения. Трудность работы со статистическими сводками заключается в том, что зачастую их не хватает для выявления истинной проблемы недостаточной производительности. Полагаясь только на профилировку, программист рискует упустить действительные причины, которые привели к низкой эффективности его приложения.

Другим типичным подходом является анализ временны́х шкал. Временна́я шкала (timeline) отражает цепочку событий, возникших в процессах (или нитях) параллельного приложения за время его выполнения. Каждому процессу на такой диаграмме соответствует строка, которая разбита на последовательность вызовов функций. Линиями или стрелками отмечаются связи между событиями. Для работы с временно́й шкалой обычно предлагаются средства масштабирования и поиска. Такие шкалы предоставляют исчерпывающую информацию о работе программы и с годами получили широкое распространение. Однако использование таких инструментов становится менее эффективным с ростом количества ядер в современных кластерах. Объем данных на шкалах может стать настолько большим, что поиск проблем производительности таким способом становится достаточно сложным. Инструменты такого типа зачастую позволяют осуществлять фильтрацию, чтобы ограничить объем отображаемых данных. Например, многие визуализаторы трасс MPI (Message Passing Interface) позволяют пользователю ограничить отображаемые передачи сообщений при помощи выбора тега или источника и назначения, но это не решает проблему полностью.

Подходом, который позволяет в той или иной степени решить ряд поставленных проблем, является автоматизированный анализ производительности. Большинство стратегий используют приемы и методы из области искусственного интеллекта, такие как экспертные системы на основе баз знаний и методы автоматической классификации. Одним из таких методов является метод поиска шаблонов неэффективного поведения, предложенный в [1]. Суть метода заключается в следующем. Различным моделям параллельного программирования присущ ряд типичных проблем производительности или, другими словами, шаблонов неэффективного поведения. Разработчик как эксперт в определенной модели может разработать набор таких шаблонов и в процессе анализа трассы выполнять их поиск, последовательно проверяя все события трассы на соответствие каждому из них. По сути шаблон — это набор найденных в трассировочном файле событий, которые удовлетворяют условиям возникновения некоторой ситуации, которую он описывает. Такой способ представления проблем производительности позволяет фиксировать сложные ситуации, не охватываемые профилировочными инструментами и визуализаторами трасс.

На основе этого метода был разработан инструментарий для анализа производительности Scalasca (SCalable performance Analysis of LArge SCale Applications) [2], который работает с моделями передачи

[1] Кемеровский государственный университет, математический факультет, ул. Красная, 6, 650043, г. Кемерово; Н. Е. Андреев, аспирант, e-mail: nik@kemsu.ru; К. Е. Афанасьев, профессор, проректор, e-mail: keafa@kemsu.ru

сообщений MPI и общей памяти OpenMP. Сегодня набирает популярность новая модель программирования PGAS (Partitioned Global Address Space — разделенное глобальное адресное пространство) [3]. В модели PGAS используются односторонние коммуникации, где при передаче данных нет необходимости в явном отображении на двухсторонние пары send и receive — трудоемкий, подверженный ошибкам процесс, серьезно влияющий на продуктивность программиста. Обычное присвоение значения переменной массива $x[i] = a$ автоматически порождает необходимые коммуникации между узлами кластера. Программы, написанные в этой модели, проще для понимания, чем MPI-версии, и имеют сравнительную или даже более высокую производительность [4]. К группе PGAS относятся такие языки, как Unified Parallel C (UPC), Co-Array Fortran (CAF), Titanium, Cray Chapel, IBM X10 и Sun Fortress. Язык UPC [5] является наиболее "взрослым" представителем модели и применяется для решения ряда практических задач как в России [15, 16], так и за рубежом [17–19]. Компиляторы для языка UPC разработаны всеми основными вендорами суперкомпьютерного рынка. Существуют реализации от компаний IBM, HP и Cray. Однако ввиду относительной молодости модели параллельного программирования PGAS для нее существует не так много инструментальных средств. Такие известные пакеты, как Intel Trace Analyzer, TAU, Cray Apprentice и др., работают с ограниченным набором программных моделей, преимущественно с моделью передачи сообщений. Те инструменты, которые на данный момент существуют, в частности для языка UPC, такие как upc_dump, upc_trace [6], обладают ограниченной функциональностью, а PPW (Parallel Performance Wizard) [7] позволяет получить лишь статистические данные о работе программы. В результате разработчики, использующие новые модели параллельного программирования, зачастую вынуждены вручную выполнять трудоемкий анализ в процессе оптимизации своей программы.

Таким образом, актуальной является задача разработки инструмента автоматизированного анализа производительности параллельных программ, написанных в модели PGAS, в частности на языке UPC. Пакет Scalasca хорошо зарекомендовал себя в вопросах анализа производительности параллельных программ, поэтому логичным было бы попытаться адаптировать метод автоматизированного поиска шаблонов неэффективного поведения, на котором основан пакет, для модели PGAS. Метод предполагает анализ сложных межсобытийных связей. Для этого необходимы высокоуровневые структуры данных, которые способны отслеживать и предоставлять в нужный момент такую информацию. Поэтому для работы с трассой удобно использовать некий инструмент доступа к трассировочным файлам, который бы позволял упростить описание шаблонов, обеспечивал произвольный доступ к различным событиям, а также предоставлял некоторые абстракции, отражающие различные аспекты общего состояния выполнения программы, и связи между событиями. Инфраструктура пакета Scalasca предоставляет такие возможности, но так как он предназначен для работы с моделями передачи сообщений и общей памяти и не поддерживает модель разделенного глобального адресного пространства, то значительная часть пакета требует переработки. Настоящая статья посвящена разработке нового инструмента анализа производительности для языка UPC с использованием частей пакета Scalasca.

1. Инфраструктура пакета Scalasca. В [8] рассматривается эталонный процесс анализа и оптимизации программы. Сначала пользователь добавляет в исходную программу дополнительный код, который в процессе работы программы будет записывать информацию о вызовах всех функций. Этот этап называется инструментовкой или оснащением. Чаще всего он выполняется автоматически самим инструментом анализа производительности в момент компиляции программы. Далее, оснащенное таким образом приложение запускается на выполнение. После его завершения сырые данные о работе программы, собранные с каждого процесса/нити, подаются в модуль анализа. Здесь выполняется анализ производительности при помощи методов, реализованных в данном конкретном инструменте, после чего результаты анализа визуализируются в графическом интерфейсе пользователя. На основе полученных данных программист самостоятельно выполняет оптимизацию своей программы, и цикл повторяется до тех пор, пока не будет достигнут приемлемый уровень производительности.

Рассмотрим, как эти этапы реализуются в пакете Scalasca (рис. 1). Инструментовка выполняется либо автоматически при помощи ключей компилятора, либо вручную при помощи специального интерфейса прикладных программ (EPIK API), либо в полуавтоматическом режиме при помощи директив интерфейса под названием POMP. Дальше программа запускается на выполнение и на выходе для каждой нити формируется трассировочный файл. Для хранения записей трассы Scalasca использует собственный формат EPILOG [9]. Перед выполнением анализа все файлы объединяются и передаются в модуль EARL [12]. EARL — Event Analysis and Recognition Language (язык распознавания и анализа событий) представляет собой высокоуровневый интерфейс для доступа к записям трассы. При помощи EARL API разработчик может работать с трассой без необходимости ручного разбора текстового файла. На этом интерфейсе написаны непосредственно алгоритмы поиска шаблонов неэффективного поведения для MPI и OpenMP. Визуализация результатов проводится в утилите Cube, которая также входит в состав пакета.

Интерфейсы прикладных программ Scalasca, такие как EPILOG API, EARL API и Cube API, достаточно хорошо документированы, что позволяет использовать пакет для разработки инструментов анализа производительности для других языков. Однако это потребует замены всей цепочки до этапа “Анализ”. С другой стороны, исчезает необходимость разработки модуля визуализации, а также появляется возможность воспользоваться языком EARL для написания алгоритмов поиска шаблонов неэффективного поведения для языка UPC. Все это позволяет сократить время разработки инструмента автоматизированного анализа производительности для языка UPC и сконцентрировать большее внимание на адаптации оригинального метода для модели PGAS.

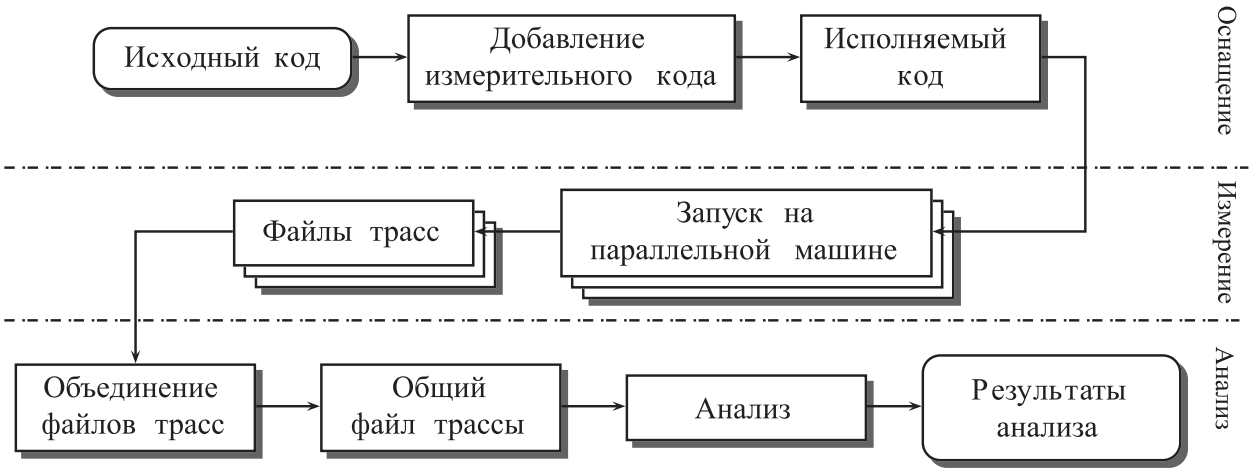

Рис. 1. Алгоритм работы пакета Scalasca

2. Интерфейс GASP. GASP [10] — это событийный интерфейс, который указывает, как PGAS-компиляторы и системы времени выполнения должны обмениваться информацией с инструментами анализа. Наиболее важная, входная точка GASP-интерфейса — это функция обратного вызова (callback) gasp_event_notify(), посредством которой компилятор уведомляет измерительный инструмент, когда возникает событие, вызывающее потенциальный интерес. Вместе с идентификатором события передается место вызова в исходном коде и аргументы, связанные с данным событием. Дальше инструмент “сам” решает, как обработать информацию и какие дополнительные метрики записать. Кроме того, можно вызвать функции исходного языка или воспользоваться его библиотекой, чтобы запросить специфичную для данной модели информацию, которую невозможно получить иным способом. Инструменты также могут обращаться к другим источникам информации о производительности, таким как счетчики CPU, прочитанные при помощи PAPI, для более детального мониторинга последовательных аспектов вычислений и производительности подсистемы памяти.

В процессе трассировки собирается исчерпывающий набор данных о работе UPC-приложений, который включает в себя следующие основные категории. События доступа к общим переменным, фиксирующие неявные (манипуляции с общими переменными) и явные (блочные передачи данных) односторонние коммуникационные операции. События синхронизации, такие как решетки (fence), барьеры и блокировки, которые служат для регистрации операций синхронизации между нитями. События распределения работы (work-sharing), обрабатывающие явно параллельные блоки кода, определенные пользователем. События инициализации и завершения работы нитей. Кроме того, существуют коллективные события, которые фиксируют такие операции, как broadcast, scatter и им подобные, и события управления памятью в общей и приватной кучах (heap). Наконец, в интерфейсе предусмотрены средства для пользовательских, явно срабатываемых событий, что позволяет разработчику задавать контекст получаемым данным. Это облегчает фазовую профилировку и выборочное оснащение определенных сегментов кода.

Интерфейс GASP не совместим с форматом EPILOG и формирует данные в собственном формате, поэтому на этапе объединения файлов трасс потребуется выполнять конвертацию. Существует возможность в момент сбора трассировочной информации конвертировать ее в формат EPILOG “на лету”. Однако это внесет дополнительные накладные расходы. Так как сбор данных происходит в процессе работы программы, то высокий уровень накладных расходов может исказить результаты последующего анализа. Быстрая запись исходных, не трансформированных данных в промежуточный формат и их последующая конвертация позволяют избежать этой проблемы.

3. Формирование трассы. Этап инструментовки в компиляторе языка UPC на уровне пользова-

теля реализуется примерно так же, как и в компиляторах других языков. В автоматическом режиме инструментовка выполняется при помощи ключей компилятора. Ручная корректировка выполняется при помощи прагм. Ключами программист выбирает классы событий, которые его интересуют. По умолчанию программа компилируется с ключом *–inst*, который указывает, что программа пользователя должна сообщать обо всех системных событиях, поддерживаемых моделью PGAS. К ним относятся обращения к функциям языка UPC, а также односторонние коммуникационные операции, обращения к которым происходят неявно при изменении значений общих переменных. Если программист укажет ключ *–inst–functions*, то вместе с системными функциями в трассу попадет информация о вызовах пользовательских функций. Ключ *–inst–local* добавит сообщения о локальных обращениях к общим переменным программы.

Другой способ управления процессом инструментовки — это прагмы компилятора, которые позволяют избежать инструментовки отдельных частей приложения. Если в UPC-программе есть функция, которая с малой вероятностью может быть источником проблем производительности и программист не заинтересован в каждом вызове этой функции, то можно воспользоваться прагмами pupc. Они определены в спецификации GASP и позволяют указать компилятору не выполнять инструментовку определенного блока кода. Если заключить описание функции в прагмы $\#pragma\ pupc\ off$ и $\#pragma\ pupc\ on$, то инструмент не будет получать сообщения о вызове функции в процессе выполнения программы. Однако если используемый компилятор с поддержкой GASP выполняет инструментовку в местах вызова функций, а не их описания, то будет необходимо поместить *#pragma pupc* вокруг всех мест вызова функции. В современных компиляторах с поддержкой GASP, позволяющих выполнять инструментовку вызовов функций, необходимо помещать *#pragma pupc* вокруг описаний функций, а не мест их вызова.

Данные в EARL подаются в собственном формате EPILOG, поэтому трассировку UPC-приложения необходимо выполнять в соответствии с форматом его записей. Для формирования корректной трассы необходимо обязательное использование ряда записей, которые можно разбить на несколько групп. К первой группе относятся записи, описывающие архитектуру кластера и структуру исходного кода программы. Кластер состоит из узлов, узлы из процессов, а процессы, соответственно, могут состоять из одной или нескольких нитей. Четверка (кластер, узел, процесс, нить) уникально идентифицирует нить, в которой произошел вызов функции. Исходный код программы описывается в терминах регионов (region) и мест вызовов (call site). Регион — это некоторый блок исходного кода, обычно функция. Место вызова — это строка в программе, из которой был вызван регион. Ко второй группе относятся записи событий, такие как вход/выход из функции, вызов односторонних и коллективных коммуникационных операций, захват/освобождение блокировки. К третьей группе можно отнести служебные записи. Подробно с описанием формата EPILOG можно познакомиться в спецификации [9].

4. Метод поиска шаблонов. Основным методом для анализа производительности параллельных программ был выбран метод поиска шаблонов неэффективного поведения, который является наиболее развитым на сегодняшний день среди методов автоматизированного анализа [25]. Работа программы в этом методе представляется в виде последовательности событий. Событие характеризует атомарное действие, произошедшее в определенном месте (процесс/нить) и в определенное время. Событиями являются вход и выход из пользовательских функций и функций языка параллельного программирования, захват и освобождение блокировки и др. Однако единичные события, происходящие в разных нитях параллельного приложения, сами по себе не позволяют понять, в чем причина неэффективной работы программы. Чтобы выявить источник проблемы, инструмент анализа производительности должен основывать свой анализ на составных событиях. В свою очередь, составное событие — это не случайное подмножество трассы, а набор событий, возникающих в определенном контексте. Этот контекст можно представить в виде состояния параллельного приложения на момент возникновения составного события. Простые события, возникающие по мере выполнения программы, переводят ее из одного состояния в другое. Чтобы выразить сложные зависимости между компонентами составного события, используются две категории абстракций: последовательности состояний и атрибуты-указатели.

Для анализа PGAS-приложений достаточно поддерживать три состояния программы: стек блоков кода, очередь операции релокализации и статус владения. В процессе выполнения для любой программы обычно выделяются следующие области памяти: область программного кода, область статических данных, стек (stack) и куча (heap). Стек вызовов хранит информацию о функциях, вход в которые на текущий момент времени был выполнен приложением. Аналогичную структуру данных удобно использовать и в процессе анализа. Это позволяет на каждом шаге знать, какие функции в настоящий момент выполняет каждая из нитей. Состояние “Очередь операции релокализации” используется для анализа коллективных операций. Так как эти операции выполняются всеми нитями, то переход к их анализу целесообразно

начинать, после того как все нити заканчивают их выполнение. События выхода каждой нитью из коллективной операции накапливаются в этом состоянии. Последнее состояние “Статус владения” необходимо для анализа синхронизации на основе блокировок. Блокировка (или мьютекс) — это примитив синхронизации, используемый для обеспечения взаимно исключающего доступа к разделяемому ресурсу (данным или операторам). Если нить пытается захватить блокировку, которой уже владеет другая нить, то обычно она блокируется до момента освобождения. Чтобы находить подобные ситуации в трассе, необходимо отслеживать историю захватов и освобождений каждой блокировки.

Атрибуты-указатели, в свою очередь, позволяют связать между собой несколько событий трассы: событие выхода из функции с событием входа, операцию освобождения блокировки с операцией ее захвата. Указатель enterptr — это атрибут-указатель, который для произвольного события указывает на событие входа в блок кода, в котором оно произошло. Указатель lockptr — это атрибут события синхронизации, который указывает на предыдущее событие синхронизации, оперировавшее той же самой блокировкой. Указатель lockptr для операции захвата блокировки всегда указывает на операцию освобождения, и наоборот.

Были проанализированы синтаксис и семантика языка Unified Parallel C и выделены 12 шаблонов неэффективного поведения. Семь шаблонов из OpenMP и MPI были адаптированы для языка UPC, среди них: конкуренция за блокировку, ожидание в барьере, завершение барьера, поздняя рассылка, ранняя сборка, синхронизация на входе в коллективную операцию “многие ко многим”, синхронизация на выходе из коллективной операции “многие ко многим”. Кроме того, были разработаны пять новых шаблонов, характерных только для языка UPC, среди них: синхронизация на входе в коллективную операцию, синхронизация на выходе из коллективной операции, ранняя префиксная редукция, ожидание внутри коллективной операции, ожидание в операции динамического выделения памяти.

Приведем для примера один из шаблонов. Полное описание метода и всех шаблонов представлено в [23, 24].

Шаблон “Ожидание в операции динамического выделения памяти” возникает в операции *upc_all_alloc*(). В языке UPC присутствует ряд операций динамического выделения общей памяти. Если память выделяется коллективно при помощи функции *upc_all_alloc*(), то возникают требования к синхронизации. В действительности выделение памяти происходит в нити 0, после чего результат операции рассылается по всем нитям аналогично операции *upc_all_broadcast*(). Если нить 0 входит в операцию позже других, то остальные нити обязаны ждать рассылки результата (рис. 2). Этот шаблон не встречается в программной модели передачи сообщений и характерен только для языка UPC.

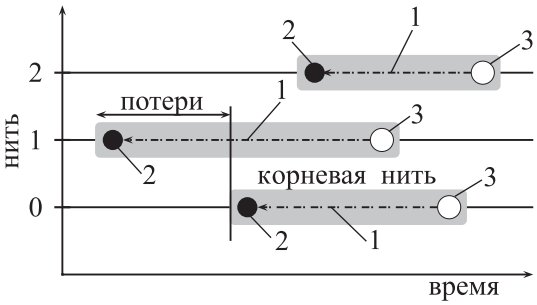

Рис. 2. Шаблон “Ожидание в операции динамического выделения памяти”: 1) enterptr, 2) начало, 3) конец

Шаблон имеет следующий алгоритм поиска. На первом этапе для всех событий трассы проверяется условие входа:

$$\begin{cases} type(last) = CExit \wedge \\ \wedge\ coll(last) \neq \varnothing \wedge \\ \wedge\ last.reg = upc\_all\_alloc. \end{cases}$$

Анализ коллективной операции начинается после того, как все нити завершили ее выполнение. Поэтому, если встретилось событие выхода из коллективной операции, это событие выхода последней нитью и функцией, из которой был выполнен выход, является *upc_all_alloc*(), то выполняется переход к анализу. Здесь функция *type*() возвращает тип события, *coll*() возвращает непустое множество, если все нити завершили выполнение коллективной операции, и пустое множество в обратном случае. Атрибут *reg* для каждого события содержит название функции, с которой связано событие. Далее проверяется условие срабатывания шаблона:

$$\begin{cases} r := last.root.enterptr, \\ E_1 := coll(last), \\ E_2 := \begin{cases} E_1.enterptr, & \text{если } \exists e \in E_1.enterptr : e.time < r.time, \\ \text{неудача}, & \text{иначе.} \end{cases} \end{cases}$$

Сначала в переменную r записывается событие входа в операцию корневой нитью. В данном случае *last* — это событие выхода из операции последней нитью. Атрибут *root* любого события коллективной операции хранит идентификатор корневой нити, а атрибут-указатель *enterptr* события выхода указывает на событие входа. Множество E_1 содержит события выхода из функции *upc_all_alloc*() всех нитей. Да-

лее выполняется проверка: если хотя бы одна из нитей вошла в операцию раньше корневой, то шаблон сработал и в E_2 присваиваются все события входа.

Если шаблон сработал, то на последнем этапе алгоритма рассчитывается время, потерянное на синхронизацию: $wasted = \sum\limits_{\substack{e \in E_2 \\ e.time < r.time}} r.time - e.time.$

5. Реализация.

5.1. Трассировка. Трассировочный модуль для приложений, поддерживающих GASP, выполняется в виде отдельной библиотеки и компилируется вместе с параллельной программой. В библиотеке должна быть приведена реализация callback-функции $gasp\_event\_notify()$, а также некоторых других, описанных в стандарте GASP [10]. Внутри модуля также выполняется непосредственно вывод данных в файл. Каждая нить в процессе выполнения записывает данные в собственный трассировочный файл.

В процессе разработки трассировочного модуля было уделено большое внимание вносимым модулем накладным расходам. В исходном коде отсутствует какая-либо обработка данных, за исключением использования хэш-таблиц для хранения имен файлов и функций, что является целесообразным для сокращения размера трассировочных файлов. Данные записываются на локальные диски вычислительных узлов и перемещаются на сетевую файловую систему после завершения параллельной программы. Это позволяет избежать задержек на передачу данных по сетевому протоколу NFS (Network File System). Кроме того, были использованы два подхода, которые позволяют значительно сократить размер трассировочных файлов при сохранении прежнего уровня накладных расходов — это асинхронный ввод/вывод с двойной буферизацией и сжатие данных “на лету”.

Существует несколько подходов к выполнению ввода/вывода (В/В). Один из самых распространенных — синхронный блокирующий В/В. В данном подходе приложение из пространства пользователя (user-space) выполняет системный вызов, который приводит к блокированию программы. Это означает, что приложение блокируется до тех пор, пока не завершится системный вызов (данные будут переданы или возникнет ошибка). Программа находится в состоянии, когда она не потребляет ресурсов процессора и просто ожидает ответа. Второй вариант синхронного В/В — синхронный неблокирующий В/В. В данной модели устройство открывается в неблокирующем режиме. Операции чтения и записи возвращают ошибку о том, что команда не может быть немедленно выполнена, а приложение должно периодически проверять завершение операции. Между проверками программа может выполнять вычисления. Другой разновидностью блокирующего подхода является метод неблокирующего В/В с блокирующими уведомлениями. В этой модели инициируется неблокирующий вызов, а для проверки состояния дескриптора В/В используется блокирующая функция $select()$. Эта функция интересна тем, что она может использоваться для нескольких дескрипторов одновременно. Для каждого дескриптора можно запросить уведомление о возможности записи данных, чтения или возникновении ошибки. Проблема с вызовом $select()$ в том, что он не очень эффективен сам по себе и не подходит для высокопроизводительного В/В. Наконец, в модели асинхронного неблокирующего В/В вызов немедленно возвращает выполнение программе. Дальше приложение может выполнять вычисления, пока операция выполняется в фоновом режиме. Для уведомления о завершении операции может быть использован механизм сигналов или обратных вызовов (callback). Данный метод позволяет в одной и той же нити одновременно выполнять вычисления и несколько операций В/В. Пока выполняются одна или несколько медленных операций В/В, процессор может продолжать выполнять параллельную программу.

Для реализации инструмента был выбран асинхронный В/В как наиболее эффективный. Трассировочная библиотека оперирует двумя буферами, в которые по мере выполнения программы записывается информация о возникающих событиях, сжатая при помощи библиотеки zlib. Когда первый буфер полностью заполняется, запись продолжается во второй буфер, в то время как при помощи механизма асинхронного неблокирующего В/В данные из первого буфера записываются в фоновом режиме на диск. Программист с помощью ключа компиляции может самостоятельно управлять размером буферов, снижая таким образом накладные расходы до минимума. Возможность сжатия данных “на лету” выгодно отличает разработанный инструмент от других инструментов анализа производительности UPC-программ, таких как Parallel Performance Wizard (PPW) [7]. В PPW такая возможность отсутствует, что негативно сказывается на размере файлов трасс.

Процесс слияния трасс всех нитей в единый файл, а также их конвертация в формат EPILOG были вынесены в отдельный модуль, который занимает значительную (более 2000 строк кода) часть инструментального средства. Так как записи формата GASP не имеют прямого соответствия записям формата EPILOG, то имеет место расчет дополнительных параметров и генерация вспомогательных записей. В процессе преобразования между форматами параллельно выполняется корректировка временны́х меток.

Часы каждого из узлов могут показывать разное время. Даже небольшое смещение часов может привести к неправильным результатам при объединении, так как события возникают в программе с очень высокой частотой. Синхронизация временны́х меток всех записей выполняется согласно алгоритму удаленного считывания времени [20]. Этот алгоритм не является лучшим, но оказался достаточным для проведения необходимых экспериментов. Сегодня существуют более точные подходы [21, 22], использование которых предполагается в следующих версиях инструмента.

Конвертация — это дополнительный шаг, вносимый в цепочку, в сравнении с оригинальным подходом Scalasca. Сам процесс анализа трассы, которая может достигать десятков гигабайт, занимает значительный объем времени. Выполнение конвертации такого количества данных можно вынести в отдельную задачу. Записи EPILOG имеют большое количество ссылок друг на друга. Поиск на каждом шаге записей, на которые необходимо сослаться, дополнительно усложняет задачу. Чтобы сократить время поиска, были использованы эффективные структуры данных, такие как хэш-таблицы и сбалансированные двоичные деревья с оптимально заданными функциями упорядочивания узлов дерева.

5.2. Анализ и визуализация. Анализ трассы EPILOG выполняется при помощи EARL API [13]. EARL API предоставляет удобный интерфейс для доступа к трассе. Места вызовов, регионы, записи трассировочного файла и так далее — это классы C++, которые создаются автоматически при создании объекта "EventTrace" и последующей итерации по нему. Таким образом все данные извлекаются вызовом методов соответствующих классов. При помощи языка EARL авторами были разработаны и реализованы 12 шаблонов неэффективного поведения UPC-программ.

Инструмент выполняет три вида анализа. Во-первых, трасса проверяется на наличие соответствий шаблонам неэффективного поведения. Во-вторых, подсчитывается ряд профилировочных данных, таких как количество вызовов каждой из функций, объем полученных/отправленных каждой нитью данных, количество операций синхронизации и коммуникаций и др. В-третьих, инструмент выполняет анализ разбалансировки нагрузки. Данный метод исходит из предположения, что сумма времени, проведенного в каждой функции приложения, должна быть примерно одинаковой для каждой нити. Пользователь может посмотреть, какие нити и в каких функциях провели времени больше и меньше среднего.

Для визуализации результатов используется инструмент Cube из пакета Scalasca. После того как трасса проанализирована, все данные, такие как структура кластера, список функций приложения и характеристики производительности, выгружаются в формат XML. Выгрузка осуществляется при помощи Cube API [14]. Дальше сформированный файл подается в программу Cube, которая непосредственно визуализирует результаты. С описанием функций, используемых для создания XML-файла, можно ознакомиться в спецификации [14].

6. Результаты.

6.1. Сравнение накладных расходов. Одной из важнейших характеристик инструментов анализа производительности является объем вносимых накладных расходов. Подавляющее большинство современных инструментов анализа выполняет сбор информации о работе программы в процессе ее выполнения. Это неизбежно влияет на работу самой программы. Если влияние слишком большое, то результаты анализа могут быть подвергнуты сомнению. В [11] средний процент вносимых в работу программы накладных расходов современными инструментальными средствами оценивается в 10%. Чтобы оценить работу разработанного авторами инструмента с позиции данной характеристики, проведены ряд экспериментов для пяти тестов (ядер) пакета NAS Parallel Benchmarks для 32 нитей, среди них были следующие.

1. Ядро EP — "Embarrassing Parallel" — основано на порождении пар псевдослучайных нормально распределенных чисел (гауссово распределение). Получаются числа $r_j \in (0,1)$, где $0 \leqslant j \leqslant 2n$ и значение n определяется классом теста. По замыслу разработчиков этот тест позволяет оценить максимальную производительность кластера при операциях с плавающей точкой при сведении к минимуму межузловых взаимодействий. Эти взаимодействия сводятся к окончательному объединению результатов, рассчитанных на каждом узле независимо от всех остальных. Этот тест может быть полезен, если на кластере будут решаться задачи, связанные с применением метода Монте-Карло. В алгоритме также учитывается время на форматирование и вывод данных.

2. MG — simple 3D MultiGrid benchmark. Приближенное решение трехмерного уравнения Пуассона $\bigtriangledown^2 u = v$ на сетке $N \times N \times N$ с периодическими граничными условиями (функция на всей границе равна 0 за исключением заданных 20 точек). Здесь значение N определяется классом теста. Функция v — кусочно-постоянная и равная нулю по всей границе за исключением определенных 20 точек. Этот тест полезен для оценки межузловых соединений.

3. CG — solving an unstructured sparse linear system by the Conjugate Gradient method (решение неструктурированной разреженной линейной системы методом сопряженных градиентов). Матрица систе-

мы является положительно определенной и симметричной. Метод сопряженных градиентов используется для нахождения приближенного значения наименьшего собственного значения матрицы. В тесте используется обратный степенной метод для нахождения наибольшего собственного значения матрицы.

4. FT — A 3-D Fast-Fourier Transform partial differential equation benchmark — численное решение уравнения в частных производных $\dfrac{\partial u(x,t)}{\partial t} = \alpha \bigtriangledown^2 u(x,t)$, $x \in \mathbb{R}^3$, с использованием прямого и обратного быстрых преобразований Фурье. Этот тест включает в себя большое количество действий, оказывающих значительную нагрузку на сетевое окружение (перемещение массивов данных).

5. IS — parallel sort of small integers. Параллельная сортировка N целых чисел. Тест не использует арифметические операции с плавающей точкой. На эффективность теста большое влияние оказывает первоначальное распределение чисел в памяти. Сортировка целых чисел является важной частью метода частиц (particle method).

Результаты показаны в сравнении с инструментами PPW и upc_trace. Утилита upc_dump не была включена в сравнение, так как по сути не является инструментом анализа и выполняет лишь сбор данных.

Таблица 1

Сравнение накладных расходов и размеров файлов трасс для тестов NPB

	Накладные расходы, %					Размер трассы, МБ				
Инструмент	CG	EP	FT	IS	MG	CG	EP	FT	IS	MG
upc_trace	24.77	0.26	8.76	5.07	14.05	940	542	479	506	674
PPW	9.97	0.09	3.12	6.73	0.33	4 524	17	775	17 401	114
TM	9.16	0.24	5.42	2.00	1.45	13 429	72	820	18 427	176
TM_Z	8.15	0.07	1.94	0.02	0.14	2 583	16	192	3 167	33

Из табл. 1 видно, что худшие результаты показывает upc_trace. Несмотря на то что это профилировщик и он не выполняет полной сборки трассы с программы, тесты показывают его низкую эффективность. В тесте CG инструмент upc_trace добавляет 25% ко времени работы программы и в тесте MG — 14%, что является достаточно высоким показателем.

PPW и инструмент, разработанный авторами (строка TM), имеют сравнимые показатели. Накладными расходами для тестов EP и MG можно пренебречь. В тесте FT TM несколько проигрывает PPW — 6% против 3%. В тесте IS TM показывает немного лучшие результаты, чем PPW — 2% против 7%. В тесте CG оба инструмента показывают схожие результаты. В целом, накладные расходы, вносимые в работу программы и для PPW, и для TM, не превышают порогового значения в 10%.

Если посмотреть на размеры результирующих трассировочных файлов, то видно, что TM показывает результаты хуже, чем PPW. Например, для теста CG общий объем файлов трасс составляет 4.41 ГБ для PPW и 13.11 ГБ для TM. Дисковые операции одни из самых медленных, поэтому объем накладных расходов для TM должен быть заметно выше. Однако применяемые в инструменте подходы асинхронного ввода/вывода и записи файлов трасс на локальные диски узлов вместо записи на сетевую файловую систему позволяют получить хорошие результаты.

Дополнительно была протестирована возможность инструмента сжимать данные файлов трасс “на лету”, что позволяет резко снизить объем файлов трасс. Из таблицы видно, что, например, для теста IS сжатие позволяет сократить размер трассы с 18 ГБ до 3.1 ГБ (строка TM_Z). PPW не позволяет выполнять сжатие на лету. Размер трассы PPW для теста IS составляет 17 ГБ. Такое значительное сокращение объема трасс снижает и накладные расходы на трассировку, что можно видеть из первой части таблицы.

Наряду с возможностью сжатия файлов трасс в инструменте реализован асинхронный ввод/вывод данных с регулируемым размером буфера. При помощи ключей скрипта запуска параллельного приложения, без перекомпиляции программы, разработчик может выбрать тип ввода/вывода — синхронный или асинхронный и размер буфера в килобайтах. По умолчанию используется буфер размером 16 МБ. Это означает, что для каждой нити в момент запуска будет выделен буфер размером 16 МБ.

Оба подхода были протестированы на тесте CG из пакета NPB. Размер буфера варьировался от 1 до 32 МБ. Результаты тестирования представлены в табл. 2, где t — общее время выполнения программы, t_o — объем накладных расходов на ввод/вывод в секундах и % — в процентах от времени работы приложения. Как видно из таблицы, при небольших размерах буфера оба подхода показывают примерно одинаковый процент возмущения, равный 26%, что является достаточно высоким показателем. При

постепенном увеличении размера практически сразу становится заметным преимущество асинхронного подхода. При 16 и 32 МБ синхронный метод показывает объем накладных расходов немного выше верхней допустимой планки в 10%, в то время как для асинхронного метода накладные расходы практически отсутствуют. Исходя из результатов тестирования можно рекомендовать использование асинхронного метода с размером буфера 16 МБ или выше.

Таблица 2

Сравнение накладных расходов для синхронного и асинхронного ввода/вывода на тесте CG из пакета NPB. Нагрузка класса C

Буф., МБ	Синхронный В/В			Асинхронный В/В		
	t_o, с	t, с	%	t_o, с	t, с	%
1	14.28	54.72	26.09	14.34	54.57	26.27
2	13.85	53.66	25.82	12.66	53.85	23.51
4	12.28	50.49	24.33	8.99	50.88	17.67
8	8.55	49.85	17.16	4.87	44.05	11.06
16	5.92	44.63	13.27	0.68	37.62	1.80
32	4.30	41.87	10.26	0.31	35.75	0.88

6.2. Пример анализа программы. Для тестирования инструментального средства были использованы параллельные алгоритмы сортировки целых чисел и быстрого преобразования Фурье из пакета NAS Parallel Benchmarks (NPB), а также реализация итерационного метода Якоби, предоставленная Dorian Krause из университета Лугано. На данных приложениях была показана адекватность разработанного инструмента проблемам, возникающим в реальных приложениях. Программы были оптимизированы в соответствии с найденными узкими местами. Увеличение производительности составило 30%, 12% и 70% соответственно. Результаты оптимизации для приложений блочной сортировки целых чисел и быстрого преобразования Фурье были совместно с сотрудниками университета Джорджа Вашингтона добавлены в пакет NPB. Приведем пример анализа и оптимизации алгоритма параллельной сортировки.

В задаче реализован алгоритм сортировки, часто использующийся в задачах, основанных на методе частиц. Такой тип сортировки характерен для приложений физики, где частицы принадлежат ячейкам и могут перемещаться между ними. Сортировка используется для переназначения частиц соответствующим ячейкам. В тесте проверяется скорость выполнения вычислений и выполнения коммуникационных операций. Отличительной особенностью задачи является то, что в ней не используются операции с плавающей точкой, однако присутствует большой объем коммуникаций.

Тест представляет интерес по двум причинам: во-первых, он используется как основа алгоритмов для решения реальных задач физики “частиц в ячейках” и во-вторых, пакет тестов NPB используется для анализа эффективности работы кластеров и хорошо оптимизирован. Вероятность нахождения проблем производительности в подобных приложениях низкая и вызывает особые трудности.

После запуска и анализа были получены первоначальные данные о производительности программы. Бóльшую часть времени (примерно 60%) программы занимают операции барьерной синхронизации (ветка Time–Synchronization–Collective–Wait at Barrier), передачи больших посылок данных (ветка Time–Communication–Point to Point–Bulk), а также операции управления общей памятью (ветка Time–Synchronization–Memory Management–Wait at Alloc). Данным классам соответствуют операции $upc\_wait()$, $upc\_memget()$ и $upc\_all\_alloc()$ соответственно.

Чтобы объяснить причину значительного времени, проведенного в операции управления общей памятью $upc\_all\_alloc()$, были изучены результаты анализа разбалансировки нагрузки. Данная метрика показала, что первая и последняя нити провели в операции $upc\_all\_alloc()$ времени значительно меньше среднего. Другими словами, они выполняли эту операцию быстрее других. Так как операция $upc\_all\_alloc()$ является коллективной и требует участия всех нитей, это означает, что все остальные нити ожидали, пока первая и последняя их догонят. Такое поведение привело к срабатыванию шаблона “Ожидание в операции динамического выделения памяти”, который показывает общее время, потерянное программой ввиду подобного неэффективного поведения.

Кроме того, интерес представляет операция $upc\_memget()$, которая выполняет крупные посылки данных между нитями и отнимает наибольшее время работы программы — 40%. На каждой итерации алгоритма выполняется передача данных от многих ко многим, т.е. каждая нить копирует в свою память порцию данных со всех других нитей. Так как в программе используется блокирующая выполнение функция $upc\_memget()$, то каждая операция копирования ожидает завершения предыдущей. Спецификация языка UPC описывает лишь минимальную библиотеку для выполнения крупных (bulk) передач данных. Функции $upc\_memput()$, $upc\_memget()$ и $upc\_memcpy()$ работают аналогично функции $memcpy()$ из стандарта C99. Каждая из них позволяет перемещать один непрерывный блок памяти в/из памяти, указанной при помощи указателей на локальное или общее пространство или между областями общей памяти.

Другие библиотечные функции, которые позволили бы выразить более сложные типы неколлективных коммуникаций, отсутствуют. К ним можно отнести крупные пересылки данных в/из прерывных областей памяти (например, столбец многомерного массива или произвольные элементы нерегулярной структуры данных). Такие сложные операции позволяют ускорить работу программы за счет агрегирования коммуникаций, т.е. объединения мелкозернистых операций (которые можно легко реализовать при помощи большого количества мелких сообщений, как это и реализовано в данной программе) в крупнозернистые операции, которые увеличивают эффективность сети передачи данных за счет пересылки более крупных сообщений и их запаковки и распаковки на каждом из концов. Более того, в оригинальной спецификации UPC отсутствует механизм, который бы позволил разработчику указать, что данная коммуникационная операция может выполняться параллельно по отношению к другим вычислительным или коммуникационным операциям, которые ее окружают.

Каждая отдельная передача в анализируемом алгоритме не зависит от всех других, поэтому естественным решением для ускорения данной программы видится выполнение этих операций параллельно. Для этого было использовано расширение стандарта UPC, которое реализовано в компиляторе Berkeley UPC и называется “Non-blocking memcpy extensions” [26]. Данное расширение содержит так называемые индексируемые функции *memcpy*(). Эти функции предоставляют общий механизм для выражения операции, собирающей и рассылающей данные в произвольные области памяти. Формирование такого перемещения данных в виде одной высокоуровневой операции вместо множества маленьких непрерывных позволяет выполнять агрессивную оптимизацию перемещения данных на уровне реализации UPC: например, подстройка механизма передачи под конкретную иерархию памяти или использование специфичных для платформы операций сборки и рассылки, реализованных в аппаратном обеспечении.

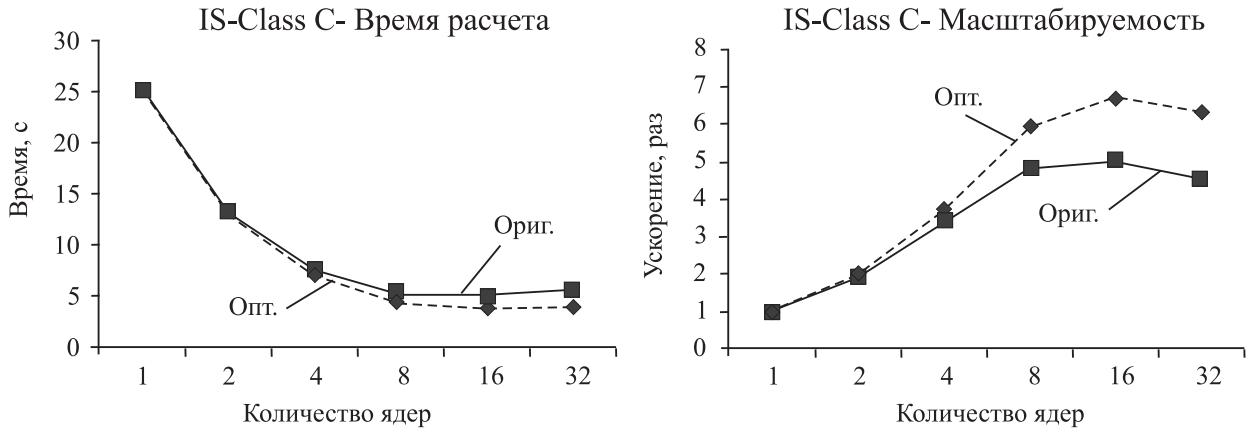

Рис. 3. Результаты оптимизации теста IS из пакета NPB

На рис. 3 и в табл. 3 приведены результаты оптимизации алгоритма для разного количества нитей. Из этих результатов следует, что эффект от оптимизации увеличивается с ростом количества нитей и составляет почти 30% для 32 нитей. Сам тест имеет достаточно небольшое время расчета, однако использование этого алгоритма в основе реальной задачи с большим количеством ячеек может дать ощутимый прирост в случае использования оптимизированной версии.

Таблица 3

Сравнение с неоптимизированной версией IS

Нити	Ориг. версия			Опт. версия			Разн., %
	Время, с	Уск.	Эффект.	Время, с	Уск.	Эффект.	
1	25.15	1.00	1.00	24.84	1.00	1.00	1.22
2	13.17	1.91	0.95	12.83	1.94	0.97	2.58
4	7.50	3.35	0.84	6.80	3.65	0.91	9.28
8	5.18	4.86	0.61	4.24	5.86	0.73	18.13
16	5.04	4.99	0.31	3.73	6.66	0.42	25.94
32	5.51	4.56	0.14	3.96	6.27	0.20	28.10

7. Заключение. В настоящей статье рассмотрена реализация нового инструмента для анализа параллельных программ, написанных на языке Unified Parallel C. Отличительной особенностью разработанного средства от других существующих инструментов является поддержка автоматизированного анализа. Такой подход упрощает анализ приложений, сокращает время, затрачиваемое на оптимизацию, и позволяет автоматически находить проблемные места программы, которые могли остаться незамеченными при ручном анализе. Для реализации этапа инструментовки был использован интерфейс GASP. Этапы сбора, объединения и конвертации данных потребовали разработки собственных модулей. За основу для модуля анализа была взята библиотека EARL из пакета Scalasca. Сами алгоритмы анализа реализованы самостоятельно. Для визуализации результатов был использован Cube из пакета Scalasca. Сравнение вносимых накладных расходов показало, что инструмент не уступает, а в некоторых случаях показывает лучшие результаты по сравнению с существующими средствами PPW и upc_trace. Еще одним преимуществом инструмента стал значительно меньший объем итоговых файлов трасс.

СПИСОК ЛИТЕРАТУРЫ

1. *Wolf F., Mohr B.* Specifying performance properties of parallel applications using compound events // Parallel and Distributed Computing Practices. 2001. **4**, N 3. 301–317.
2. *Geimer M., Wolf F., Wylie B., Abraham E., Becker D., Mohr B.* The Scalasca performance toolset architecture // Concurrency and Computation: Practice and Experience. 2010. **22**, N 6. 702–719.
3. High Productivity Computing Systems Program [Electronic resource] (http://www.highproductivity.org/).
4. *Bell C., Bonachea D., Nishtala R., Yelick K.* Optimizing bandwidth limited problems using one-sided communication and overlap // Proc. 20th International Parallel & Distributed Processing Symposium. Rhodes Island, 2006. (DOI: 10.1109/IPDPS.2006.1639320).
5. *El-Ghazawi T.* UPC Language Specifications [Electronic resource] (`http://www.gwu.edu/~upc/documentation.html`).
6. LBNL, UC Berkeley. Berkeley UPC User's Guide [Electronic resource] (verb"http://upc.lbl.gov/docs/user/index.shtml").
7. *Su H.H., Billingsley M., George A.* Parallel performance wizard: a performance analysis tool for partitioned global-address-space programming // Conference on Supercomputing. Miami, 2006. 1–8.
8. *Leko A.* Performance Analysis Strategies [Electronic resource] (http://www.hcs.ufl.edu/prj/upcgroup/upcperf/documents/20050302-AnalysisDraft.pdf).
9. *Wolf F., Mohr B., Bhatia N., Hermanns M.A., Geimer M.* EPILOG binary trace-data format. Tech. Rep. FZJ-ZAM-IB-2004-06. Forschungszentrum Julich, University of Tennessee, 2004.
10. *Su H., Bonachea D., Leko A., Sherburne H., Billingsley III M., George A.* GASP! A standardized performance analysis tool interface for global address space programming models // Proc. of Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA06). Umea, Sweden, June 18-21, 2006. 450-459.
11. *Leko A., Sherburne H., Su H., Golden B., George A.D.* Practical Experiences with Modern Parallel Performance Analysis Tools: An Evaluation [Electronic resource] (http://www.hcs.ufl.edu/upc/archive/toolevals/WhitepaperEval-Summary.pdf).
12. *Wolf F., Mohr B.* EARL — a programmable and extensible toolkit for analyzing event traces of message passing programs // Proc. of the 7th International Conference on High Performance Computing and Networking Europe (HPCN). Amsterdam, 1999. 503–512.
13. *Wolf F., Bhatia N.* EARL — API documentation: high-level trace access library. Tech. Rep. ICL-UT-04-03. Forschungszentrum Julich, University of Tennessee, 2004.
14. *Wolf F., Song F.* CUBE — User Manual. Tech. Rep. ICL-UT-04-01. Forschungszentrum Julich, University of Tennessee, 2004.
15. *Корж А.А.* Результаты масштабирования бенчмарка NPB UA на тысячи ядер суперкомпьютера Blue Gene/P с помощью PGAS-расширения OpenMP // Вычислительные методы и программирование. 2010. **11**, № 1. 164–174.
16. *Андрюшин Д.В., Семенов А.С.* Исследование реализации алгоритма Survey Propagation для решения задачи выполнимости функций булевых переменных (SAT-задача) на языке UPC // Тр. Международной суперкомпьютерной конференции "Научный сервис в сети Интернет: суперкомпьютерные центры и задачи". М.: Изд-во Моск. ун-та, 2010. 133–135.
17. *Johnson A.* Unified parallel C within computational fluid dynamics applications on the Cray X1 // Proc. of the Cray User's Group Conference. Albuquerque, 2005. 1–9.
18. *Beech-Brandt J.* Applications of UPC [Электронный ресурс] (http://www.nesc.ac.uk/talks/892/applicationsofupc.pdf).
19. *Gordon B., Nguyen N.* Overview and Analysis of UPC as a Tool in Cryptanalysis. Tech. Rep. FL 32611. High-performance Computing and Simulation (HCS) Research Laboratory, Department of Electrical and Computer Engineering, University of Florida, 2003.
20. *Cristian F.* Probabilistic clock synchronization // Distributed Computing. Berlin: Springer Verlag, 1998. 146–158.

21. *Hoefler T., Schneider T., Lumsdaine A.* Characterizing the influence of system noise on large-scale applications by simulation // Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'10). New Orleans, 2010. 1–11.
22. *Rabenseifner R.* The controlled logical clock — a global time for trace based software monitoring of parallel applications in workstation clusters // Proc. of the Fifth Euromicro Workshop on Parallel and Distributed (PDP'97). London, 1997. 477–484.
23. *Андреев Н.Е., Афанасьев К.Е.* Автоматизированный анализ производительности параллельных программ как способ повышения продуктивности разработчика // Информационные технологии и математическое моделирование (ИТММ-2010). Материалы IX Всероссийской научно-практической конференции с международным участием. Анжеро-Судженск, 19–20 ноября 2010 г. Томск: Томский гос. ун-т, 2010. Ч. 2. 121–125.
24. *Андреев Н.Е., Афанасьев К.Е.* Набор шаблонов неэффективного поведения для программной модели PGAS на примере языка UPC // Вычислительные технологии (в печати).
25. *Андреев Н.Е.* Методы автоматизированного анализа производительности параллельных программ // Вестник Новосибирского государственного университета. 2009. **7**, № 1. 16–25.
26. *Bonachea D.* Proposal for Extending the UPC Memory Copy Library Functions, v2.0 [Electronic resource] (http://upc.lbl.gov/publications/upc_memcpy.pdf).

Поступила в редакцию
10.03.2011