УДК 004.4'233

ОРГАНИЗАЦИЯ СБОРА И ПОДГОТОВКИ ДАННЫХ ДЛЯ АНАЛИЗА ПРОИЗВОДИТЕЛЬНОСТИ UPC-ПРОГРАММ

Н. Е. Андреев, К. Е. Афанасьев

AN APPROACH TO TRACING AND PREPARING DATA FOR PERFORMANCE ANALYSIS OF UPC APPLICATIONS

N. E. Andreev, K. E. Afanasiev

Программная модель с разделенным глобальным адресным пространством (Partitioned Global Address Space - PGAS) является относительно новым направлением в разработке параллельных программ. Новая модель требует новых средств поддержки процесса разработки приложений. В статье представлен подход к организации сбора и подготовки данных для анализа производительности программ, написанных на языке Unified Parallel C (UPC), наиболее «взрослом» представителе модели PGAS.

Partitioned Global Address Space (PGAS) programming model is a relatively new area of parallel applications development. New model demands new tools to support applications development process. Paper describes approach to tracing and preparing data for performance analysis of parallel applications written in Unified Parallel C which is a most mature PGAS language.

Ключевые слова: PGAS, UPC, инструментовка, трассировка, анализ производительности, параллельное программирование.

Keywords: PGAS, UPC, instrumentation, tracing, performance analysis, parallel programming.

Введение

Разработка комплексов параллельных программ для высокопроизводительных вычислительных систем является сложной задачей. Суперкомпьютеры могут иметь разную архитектуру и коммуникационную сеть, существует ряд алгоритмов распараллеливания, а также технологий и средств для написания параллельных программ. Стандартом де-факто среди таких средств на сегодня является MPI. Несмотря на сравнительно высокую производительность MPI-программ, процесс разработки приложений с использованием данной библиотеки трудоемок и подвержен ошибкам. Ограничения программной модели передачи сообщений широко признаны, а саму библиотеку MPI иногда называют «ассемблером параллельного программирования». Альтернативой MPI является набирающая популярность модель программирования PGAS - Partitioned Global Address Space (разделенное глобальное адресное пространство) [1]. 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 [2]. К группе PGAS относятся такие языки как: Unified Parallel C (UPC), Co-Array Fortran (CAF), Titanium, Cray Chapel, IBM X10, Sun Fortress. Язык UPC [3] является наиболее «взрослым» представителем модели. Компиляторы для языка UPC разработаны всеми основными вендорами суперкомпьютерного рынка. Существуют реализации от компаний IBM, HP, Cray, SGI. Компания TotalView – ведущий разработчик в области отладки параллельных приложений, поддерживает язык UPC. Компания IBM включила поддержку языка UPC в интегрированную среду разработку Eclipse. Серия суперкомпьютеров Cray Baker, анонсированная во втором квартале 2010 года, построена на основе коммуникационной сети Gemini, которая специально разработана для поддержки программной модели PGAS на аппаратном уровне. IBM планирует к 2011 году выпустить машину Blue Waters, которая также поддерживает парадигму PGAS на аппаратном уровне. Все это говорит о том, что в ближайшее время данная программная модель получит широкое распространение. Однако, ввиду относительной молодости модели PGAS, для нее существует не так много инструментальных средств. Хотя определенная работа в этом направлении была выполнена, такие известные инструменты, как Intel Trace Analyzer, TAU, Cray Apprentice и другие, работают с ограниченным набором программных моделей, преимущественно моделью передачи сообщений. В результате разработчики, использующие новые модели параллельного программирования, зачастую вынуждены самостоятельно выполнять трудоемкий анализ при оптимизации своей программы. В связи с этим, актуальной является задача разработки инструмента анализа производительности, который бы в полной мере поддерживал модель PGAS.

Процесс анализа производительности

Adam Leko в статье «Performance Analysis Strategies» [4] приводит эталонный процесс анализа и оптимизации программы (рис. 1). 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. В данной статье будут рассмотрены только этапы оснащения и измерения и их реализация на примере инструмента, разработанного автором для языка UPC.

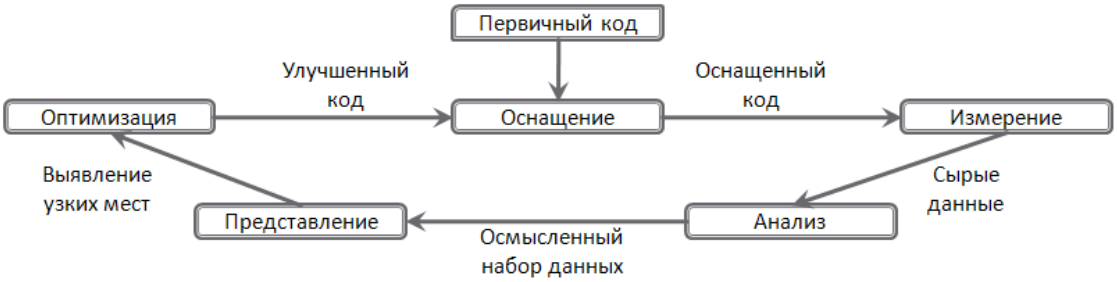

Рис. 1. ***Цикл анализа и оптимизации программы***

Инструментовка

На этапе инструментовки инструментовщик (вспомогательная программа или разработчик) добавляет дополнительный код в исходное приложение, который будет сообщать о том, когда произошли те или иные интересующие события, например, коммуникации или синхронизация. Инструментовка может быть выполнена тремя разными способами: инструментовкой исходного кода, с использованием библиотек-оберток или бинарной инструментовкой. Хотя инструменты обычно используют какой-то один из методов, можно использовать комбинацию данных подходов.

Инструментовка исходного кода помещает измерительный код прямо в исходный текст программы разработчика. Хотя это позволяет инструментам в дальнейшем легко связать полученную информацию со строками кода приложения, изменение исходного кода может повлиять на оптимизацию уровня компилятора. Кроме того, инструментовка исходного кода имеет ограничения. Она позволяет измерять только те части приложения, для которых доступен исходный код, что может стать проблемой, если пользователь захочет проанализировать внешние библиотеки, которые распространяются в скомпилированном варианте. Также, инструментовка исходного кода обычно требует полной перекомпиляции программы, что не очень удобно для больших комплексов программ.

Библиотеки-обертки являются посредниками между программой пользователя и конкретной библиотекой, поэтому они могут быть использованы только для записи информации о вызовах функций соответствующих библиотек, таких как MPI. Вместо линковки с исходной библиотекой, программист сначала линкует программу с библиотекой, которая поставляется вместе с инструментом анализа, а затем с исходной. Таким образом, библиотечные вызовы перехватываются инструментом, который записывает необходимую для анализа информацию, а затем сам выполняет вызов функции аутентичной библиотеки. На практике такое посредничество обычно реализуется на этапе линковки при помощи так называемых «слабых символов». Библиотеки-обертки удобны для разработчиков, так как достаточно перелинковать программу с новой библиотекой, что также означает снижение влияния на оптимизирующий компилятор. Однако библиотеки-обертки ограничены захватом информации только о библиотечных вызовах. Кроме того, многие инструменты, пользующиеся такими обертками, не могут соотнести собранные данные с исходным кодом программы (например, места обращений к библиотеке). Обертки используются для реализации профилировочного интерфейса MPI (PMPI), которым пользуются большая часть инструментов анализа производительности для записи информации о коммуникациях MPI.

Бинарная инструментовка наиболее удобный для программиста метод. Однако он вызывает большие трудности у разработчиков инструментов. Данный подход помещает инструментовочный код прямо в исполняемый файл и не требует перекомпиляции или перелинковки. Инструментовка может выполняться перед запуском или прямо во время запуска. Кроме того, так как в перекомпиляции и перелинковке необходимости нет, все оптимизации, выполненные компилятором, сохраняются. Главная проблема с бинарной инструментовкой в том, что она требует серьезных изменений для поддержки новых платформ, так как каждая платформа обычно имеет совершенно отличный формат исполняемых файлов и набор инструкций. Как и с библиотеками-обертками, сопоставление собранных данных с исходным кодом программы сложно или вообще невозможно, особенно если в исполняемом файле отсутствует отладочная информация.

В данном инструменте был использован альтернативный подход, который заключается в использовании стандартного интерфейса между компилято-

ром и инструментом анализа производительности. В таком случае ответственность за вставку подходящего кода оснащения лежит на разработчиках компиляторов, которые лучше других знают среду времени выполнения (runtime). Перемещение этой обязанности с разработчика инструмента на разработчика компилятора резко снижает шанс непреднамеренного изменения поведения программы. Простота интерфейса снижает усилия, которые необходимо приложить разработчику компилятора для добавления поддержки программ анализа к своей системе. Кроме того этот простой интерфейс позволяет добавлять новые PGAS языки в уже существующие инструменты с минимальными усилиями.

Исходя из этого, при разработке инструмента анализа для языка UPC был выбран интерфейс производительности глобального адресного пространства (GASP) [5]. Это событийный интерфейс, который указывает как PGAS компиляторы и среды времени выполнения должны обмениваться информацией с инструментами анализа (рис. 2). Наиболее важная, входная точка GASP интерфейса - это функция обратного вызова (callback) *gasp_even t_notify()* (листинг 1), посредством которой реализации PGAS уведомляют 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, извлеченные PAPI, для более детального мониторинга последовательных аспектов вычислений и производительности подсистемы памяти.

Рис. 2. ***Высокоуровневая организация PGAS приложения, выполняющегося в GASP реализации***

Обратный вызов *gasp_event_notify()* включает в себя уникальный для каждой нити и каждой модели указатель на контекст, который является непрозрачным (opaque) объектом, создаваемым в процессе инициализации, где инструмент хранит локальные для нити данные о производительности.

Интерфейс GASP хорошо расширяем, что позволяет инструменту анализа получать характерные для языка и реализации сообщения, которые несут в себе информацию о производительности, с разной степенью детализации. Кроме того, интерфейс позволяет инструменту перехватывать только то подмножество сообщений, которое необходимое для выполнения текущей задачи анализа.

```
typedef enum {
GASP_START,
GASP_END,
GASP_ATOMIC
} gasp_evttype_t;

void gasp_event_notify(gasp_context_t con
text, unsigned int evttag, gasp_evttype_t
evttype, const char *filename, int linenum,
int colnum, ...);

void gasp_event_notifyVA(gasp_context_t con
text, unsigned int evttag, gasp_evttype_t
evttype, const char *filename, int linenum, int colnum, va_list va-
rargs);
```

Листинг 1. ***Структура уведомлений о событиях GASP***

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 (массовые (bulk) передачи данных и библиотечные вызовы асинхронных коммуникационных функций) 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 запуска и завершения, имеющие дело с инициализацией и завершением каждой нити; события вызова коллективных коммуникационных функций, которые фиксируют такие операции, как *broadcast*, *scatter* и им подобные; события управления памятью в общей и приватной кучах (heap).

Выборочная инструментовка

Компиляция программы в разработанном инструменте осуществляется при помощи скрипта-обертки tm_upcc. При помощи ключей данного скрипта, программист может выбрать 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.

Заметим, что в случае возникновения больших накладных расходов из-за частых вызовов коротких функций, можно воспользоваться встраиванием (inlining) этих функций при помощи макросов или директив/ключевых слов компилятора, таких как *inline*. Вызовы функций могут генерировать достаточно большие накладные расходы, если тело функции состоит из малого набора инструкций, поэтому можно ожидать большого увеличения производительности при использовании встраивания.

Кроме управления инструментовкой на уровне строк при помощи *#pragma pupc*, можно указать флаг – *noinst* при компиляции приложения. Это позволяет игнорировать целые части программы, если, например, они уже были оптимизированы или они практически не влияют на общую производительность приложения.

Измерение

Непосредственный сбор сообщений, возникающих в процессе работы приложения, и их запись в файлы трасс выполняет разработанная и входящая в состав инструмента трассировочная библиотека, которая связывается (link) с программой пользователя в процессе компиляции и становится неотъемлемой частью общего приложения. Любой код, добавленный к основной программе, вносит накладные расходы на ее выполнение. Большие накладные расходы могут сильно исказить результат трассировки, что в свою очередь приведет к некорректным результатам при анализе. Поэтому главным требованием к таким библиотекам являются низкие вносимые накладные расходы. Основным источником накладных расходов при трассировке являются медленные операции дискового ввода/вывода.

В данной системе используется асинхронный неблокирующий В/В, как наиболее эффективный метод параллельного выполнения вычислений и записи данных на диск. Трассировочная библиотека оперирует двумя буферами, в которые по мере выполнения программы записывается информация о возникающих событиях. Когда первый буфер полностью заполняется, библиотека продолжает записывать данные 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 скрипта tm_upcc может самостоятельно управлять размером буферов, снижая, таким образом, накладные расходы до минимума.

Кроме механизмов двойной буферизации и асинхронного ввода/вывода в системе поддерживается запись на локальные диски вычислительных узлов и последующего перемещения всех файлов трасс на сетевую файловую систему после завершения параллельной программы. Это позволяет избежать задержек на передачу данных по сетевому протоколу NFS.

После того, как все события записаны в отдельные файлы трасс для каждой нити, они объединяются в единый файл для последующего анализа. На данном этапе необходим механизм точной синхронизации времени, так как 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. В процессе объединения, записи файлов трасс упорядочиваются по времени при помощи алгоритма программной синхронизации времени, который будет описан далее, и передаются в модуль анализа.

Выборочное измерение

Разработанный автором инструмент также поддерживает простой интерфейс прикладных программ для включения и отключения измерений для определенных частей программы. Этот интерфейс не влияет на инструментовку, это значит, что он не так эффективен с точки зрения снижения накладных расходов, как выборочная инструментовка. Однако так как подход реализован в виде интерфейса, это обеспечивает большую гибкость.

Функция *pupc_control(int on)* позволяет программисту включить или отключить сбор данных в процессе работы программы. Если параметр *on* равен нулю, то инструмент перестанет записывать какие-либо данные в файл трассы, до тех пор, пока сбор не будет включен другим вызовом функции *pupc_control()* с ненулевым значением *on*. Эта функция позволяет ограничить сбор данных о производительности до отдельных частей программы, что может быть полезно, если вы заинтересованы в оптимизации только определенных блоков кода.

Важно помнить, что функция *pupc_control()* никак не влияет на инструментовочный код, добавленный в программу. Вместо этого он заставляет инструмент игнорировать данные о производительности для определенных частей программы во время ее выполнения.

Анализ фаз программы

Хотя такие ключи, как *--inst-local* и *--inst-functions*, позволяют в определенной степени управлять тем, какие регионы кода программы будут инструментованы, информации только уровня функций может быть недостаточно для анализа параллельного приложения. Иногда необходимо знать время, потраченное в определенной фазе программы.

Если программа была скомпилирована с ключом *--inst-functions*, то при просмотре собранных данных, можно будет узнать, сколько времени каждая нить провела в отдельных функциях, но не будет никакой информации о том, сколько времени было потрачено на каждую из фаз программы. Например, для приложений линейной алгебры фазами могут быть: инициализация матрицы, вычисление собственных значений, произведение матрицы на вектор, сбор результатов со всех нитей, форматирование результатов и операции ввода/вывода для записи всех вычислений на диск. Каждая фаза обычно имеет собственные характеристики производительности и поэтому удобно в процессе оптимизации рассматривать каждую фазу, как отдельную сущность. Если фаза вычислений имеет неравномерную нагрузку, то это будет сложно выявить, анализируя лишь производительность функций целиком. Подобным образом, если программа имеет сложную структуру, где фазы не разделены четко по вызовам функций, то будет очень сложно определить, к какой фазе относятся обнаруженные проблемы производительности.

В разработанном автором инструменте, реализован интерфейс прикладных программ, приведенный в листинге 2. Данный набор функций может быть использован для ручной инструментовки программы. Это позволит сообщить инструменту о специфичных для программы событиях, таких как вход в фазы коммуникации или вычислений.

```
#include <pupc.h>
unsigned int pupc_create_event(const char
*name, const char *desc);
void pupc_event_start(unsigned int evttag,
...);
void pupc_event_end(unsigned int evttag,
...);
void pupc_event_atomic(unsigned int evttag,
...);
```

Листинг 2. ***Интерфейс прикладных программ для пользовательских событий***

Если планируется использование программы в системах, которые могут не поддерживать данные функции (например, компиляторы, не поддерживающие интерфейс GASP), то пользователь может защитить все части кода, связанные с данными функциями, проверяя существование макроса *__UPC_PUPC__*. Любой компилятор UPC или инструмент анализа, поддерживающий описываемый здесь интерфейс прикладных программ, задает макрос *__UPC_PUPC__*. Таким образом, защита ручной инструментовки при помощи *#ifdef* позволит коду оставаться портирумым на системы, без поддержки данного интерфейса.

Функция *pupc_create_event()* приводит к генерации идентификатора для пользовательского сообщения с именем *name* и описанием *desc*. Идентификатор, возвращаемый функцией, может использоваться для блока кода или фазы программы. То, как этот идентификатор будет использоваться для сбора данных о производительности, полностью зависит от программиста.

После того, как новый идентификатор был сгенерирован при помощи функции *pupc_create_event()*, он может быть использован в семействе функций *pupc_event()* для обозначения атомарных операций или операций входа/выхода, связанных с данным событием. Функция *pupc_event_start()* означает вход в событие, а *pupc_event_end()* его завершение.

Строка *desc*, переданная в функцию *pupc_create_event()* может содержать строку в формате *printf*, при помощи которой в дальнейшем будут интерпретированы дополнительные аргументы, передаваемые в семейство функций *pupc_event()*, после идентификатора события *evttag*.

Если аргумент *desc* не NULL, то каждый вызов функции из семейства *pupc_event()* с данным идентификатором события, должен содержать аргументы согласно формату, переданному в аргументе *desc*, иначе инструмент анализа производительности может вести себя некорректно или аварийно завершить свое выполнение.

Функция *pupc_event_atomic()* - это специальная функция, которая генерирует атомарное событие. Это событие полезно, если необходимо записать ка-

кие-либо данные, в то время как программа находится в процессе выполнения данного события. Или, например, если нужно записать сколько раз произошло какое-либо событие, без использования каждый раз функций *pupc_event_start()* и *pupc_event_end()*.

По умолчанию, пользовательские события обрабатываются так же, как функции, поэтому все вызовы функций семейства *pupc_event()* должны быть правильно вложены. Другими словами, любой вызов *pupc_event_start()* должен сопровождаться вызовом *pupc_event_end()* с тем же самым идентификатором события.

Синхронизация времени

Далеко не все высокопроизводительные вычислительные системы имеют реализацию внутренней аппаратной синхронизации часов между узлами машины. В таких случаях часы каждого из узлов могут отличаться между собой на смещение (offset) и отклонение (drift). Данный инструмент решает данную проблему при помощи программной синхронизации, которая обеспечивает правильный порядок взаимосвязанных между собой сообщений.

Вместо того чтобы корректировать время в процессе работы программы, инструмент выполняет «посмертную» (post-mortem) синхронизацию локальных по отношению к каждому узлу временных меток во время объединения локальных файлов трасс в единый глобальный файл. Для этого при запуске программы и по ее завершению выполняются измерения смещений на основе алгоритма удаленного считывания времени. Синхронизация происходит асимметрично – одна главная нить сообщает время всем остальным подчиненным нитям. Подчиненная нить отправляет запрос главной в момент времени s_1, главная нить отвечает отправкой текущего значения своих часов m, и подчиненная получает это значение в момент времени s_2. Тогда время в подчиненной нити, соответствующее моменту времени m в главной нити, рассчитывается следующим образом: $s := s_1 + (s_2 - s_1)/2$. Таким образом, смещение рассчитывается, как разность $m - s$. Чтобы минимизировать эффект от асимметричных задержек при отправке и получении сообщений, в инструменте реализован следующий статистический подход. Каждая подчиненная нить выполняет серию пересылок. Для расчетов используется пересылка с минимальной разницей $s_2 - s_1$. Считается, что она имеет минимальную и, следовательно, симметричную задержку. После завершения работы программы, каждая нить имеет две пары (s_s, o_s) и (s_e, o_e), которые содержат локальное время, вместе со смещением относительно главной нити. Первая пара рассчитывается при запуске программы и вторая при завершении. В посмертном (post-mortem) алгоритме синхронизации времени предполагается, что все часы имеют постоянное отклонение, и могут быть представлены в виде линейной функции. В результате, время каждой подчиненной нити может быть выражено через время главной нити следующим образом:

$$m(s) := s + \frac{(o_e - o_s)}{(s_e - s_s)} \times (s - s_s) + o_s .$$

Чтобы обойти эффекты внешней синхронизации локальных часов при помощи NTP, измерения времени были реализованы при помощи таймеров высокой точности [6], использующих тики (tics) процессора, вместо системного времени.

Заключение

Цикл анализа и оптимизации программы состоит из пяти этапов. Четыре из них - оснащение, измерение, анализ и представление выполняет инструментальное средство, а последний этап – оптимизация, выполняется программистом. В статье представлено описание первых двух этапов и их реализации для инструментального средства анализа производительности UPC-программ. Описана реализация инструментовки при помощи интерфейса GASP, а также способы выборочной инструментовки, предоставляемые пользователю инструмента, главная цель которых, снизить накладных расходы. Дано описание измерительной библиотеки инструмента, выполняющей сбор пользовательских и системных сообщений, а также эффективных алгоритмов ввода/вывода, использованных в процессе реализации. Перечислены механизмы выборочного измерения, при помощи которых пользователь может управлять процессом сбора трассировочных данных. Приведен интерфейс прикладных программ, позволяющий программисту выделять фазы программы и в дальнейшем анализировать их как отдельные функции. Также описан алгоритм синхронизации, использованный для упорядочивания событий разных нитей по времени.

Литература

1. High Productivity Computing Systems Program [Электронный ресурс]. URL: http://www.highproductivity.org/ (дата обращения: 30.08.2010).

2. Bell, C. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap [Текст] / C. Bell, D. Bonachea, R. Nishtala, K. Yelick // 20th International Parallel & Distributed Processing Symposium, 2006.

3. El-Ghazawi, T. UPC Language Specifications [Электронный ресурс]. URL: http://www.gwu.edu~upc/documentation.html (дата обращения: 30.08.2010).

4. Leko, A. Performance Analysis Strategies [Электронный ресурс]. URL: http://www.hcs.ufl.edu/prj/upcgroup/upcperf/documents/20050302-AnalysisDraft.pdf (дата обращения: 30.08.2010).

5. Su, H. GASP! A Standardized Performance Analysis Tool Interface for Global Address Space Programming Models [Текст] / H. Su, D. Bonachea, A. Leko, H. Sherburne, M. Billingsley III, and A. George // Proc. of Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA06), Umea, Sweden, June 18–21, 2006.

6. Bonachea, D. Proposal for High-Performance Wall-Clock Timers in UPC [Электронный ресурс]. URL: https://upc-wiki.lbl.gov/images/9/9b/Upc_tick_0.2.pdf (дата обращения: 30.08.2010).