Один из подходов к трассировке параллельных программ в модели разделенного глобального адресного пространства

Н.Е. Андреев

Кемеровский государственный университет
nik@kemsu.ru

Введение

Компьютер – пожалуй, самая сложная машина, созданная когда-либо человеком, а параллельный компьютер, в свою очередь – самая сложная компьютерная система. Она была создана для поддержки одновременных вычислений, но несмотря на то, что одновременные события - явление обычное для окружающего нас мира, запрограммировать одновременность на компьютере не просто, даже для простых на первый взгляд задач. Главное преимущество параллельных систем заключается в поддержке одновременного выполнения параллельных операций с целью достижения высокой производительности. Получение высокой эффективности в процессе выполнения программы еще более усложняет использование параллельных систем. Согласно отчету Межведомственной комиссии по развитию сверхмощных вычислений США эффективность современных параллельных систем находится ниже отметки в 10%. [1] Предложения в области высокопроизводительных вычислений представляют собой совокупность большого количества процессоров, разработанных для более мелких систем. Такие массивные мультипроцессорные системы исключительно сложны в программировании и достижении высокого уровня производительности для определенного класса приложений. Постоянные технологические улучшения микропроцессоров, приводят к резкому росту теоретической пиковой производительности. Тем не менее, в результате мы получаем все менее сбалансированные мультипроцессоры, в которых с каждым годом растет дисбаланс между скоростью процессоров и пропускной способностью памяти. Этот дисбаланс приводит к низкому росту производительности на реальных приложениях.

В условиях резкой нехватки производительности программистам необходимы инструменты анализа эффективности и оптимизации программ. Разработка и реализация таких инструментов для параллельных компьютеров очень сложна, ввиду как архитектурной, так и эксплуатационной сложности таких систем.

Инструменты анализа производительности, такие как KOJAK [2] и TAU [3] доказали свою полезность в вопросах анализа критичных ко времени приложений. Упрощая процесс оснащения, систематизируя данные о производительности в информативных диаграммах, давая возможность выявлять узкие места программы, эти инструменты значительно сокращают время, необходимое для анализа и оптимизации параллельных программ. Тем не менее, большинство таких инструментов работают с ограниченным набором программных моделей, преимущественно моделью передачи сообщений. В результате разработчики, использующие новые модели параллельного программирования зачастую вынуждены вручную выполнять трудоемкий анализ, при оптимизации своей программы.

Хотя определенная работа в этом направлении была выполнена, подавляющее большинство новых моделей программирования не поддерживаются инструментами анализа производительности ввиду объема работы, необходимого для включения поддержки новой модели. В частности модели глобального адресного пространства (GAS), которые, не смотря на рост популярности, в данный момент не представлены среди инструментов анализа производительности. В данный класс входят языки Unified Parallel C (UPC) [4], Titanium [5], SHMEM и Co-Array Fortran (CAF) [6].

Описание метода

Данная работа посвящена разработке инструмента автоматического поиска шаблонов неэффективного поведения параллельных программ, для программной модели разделенного глобального адресного пространства (PGAS). В частности для языка Unified Parallel C (UPC). Традиционная модель передачи сообщений, реализованная в MPI, доминирует сегодня в области крупномасштабных параллельных комплексов. Тем не менее, ограничения этой модели, которые значительно снижают продуктивность разработки, широко признаны, поэтому PGAS модели набирают популярность. [7] Обеспечивая абстракцию общего адресного пространства для разнообразных системных архитектур, эти модели позволяют разработчикам выражать межпроцессные коммуникации подобно традиционному программированию с общей памятью, но с явным семантическим указанием локальности (locality), что позволяет добиться высокой производительности на системах с распределенной памятью. Модель GAS делает акцент на использовании односторонних коммуникационных операций, где при передаче данных нет необходимости в явном отображении на двухсторонние пары send и receive – трудоемкий подверженный ошибкам процесс, серьезно влияющий на продуктивность программиста. В результате, программы, написанные в этой модели, проще для понимания, чем MPI версии, и имеют сравнительную или даже более высокую производительность [8, 9].

Современные инструменты анализа используют несколько подходов для оснащения параллельных программ метками, снимающими метрики производительности. К сожалению, ни один из этих методов не подходит для программной модели GAS. Оснащение исходного кода может помешать оптимизирующему компилятору, а также не позволяет работать с релаксированной (relaxed) моделью памяти, где некоторые семантические детали коммуникаций намерено недоопределены на уровне исходного кода, давая возможность выполнять агрессивную оптимизацию. Оснащение на уровне двоичного кода реализовано не на всех архитектурах. Также данный метод не всегда позволяет скоррелировать данные о производительности с исходным кодом, особенно в системах использующих компиляцию из исходного кода в исходный код (source-to-source compilation). Подход с промежуточной библиотекой, использующей обертки вокруг функций, реализующих интересующие операции, не работает для компиляторов генерирующих код, который напрямую использует инструкции аппаратного обеспечения или низкоуровневые проприетарные интерфейсы. Наконец, разные компиляторы и среды выполнения могут использовать различные способы реализации (даже для одного и того же языка), что еще более усложняет процесс сбора данных. Например, существующие реализации UPC включают прямые, монолитные компиляторы (GCC-UPC, Cray UPC) и трансляторы исходного кода, дополненные библиотеками времени выполнения (Berkeley UPC, HP UPC и MuPC). Разнообразные подходы приводят к таким существенным различиям процессов компиляции и запуска, что не существует единого подхода оснащения, который был бы эффективен для всех реализаций. Естественный выход из данной ситуации – просто выбрать один из существующих методов оснащения, который работает для конкретной реализации. К сожалению, этот подход потребовал бы глубоко изучить внутренние и часто изменчивые или проприетарные детали реализации, и привел к разработке системо-зависимого, непереносимого инструмента, во вред пользователю.

Понятно, что для того чтобы справится с моделью GAS необходим новый подход. Альтернативой является использование стандартного интерфейса между компилятором и инструментом анализа производительности. В таком случае ответственность за вставку подходящего кода оснащения лежит на разработчиках компиляторов, которые лучше других знают среду выполнения. 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 GAS 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). Это событийный интерфейс, который указывает как GAS компиляторы и системы времени выполнения должны обмениваться информацией с инструментами анализа (рисунок 1). Наиболее важная, входная точка GASP интерфейса – это функция обратного вызова (callback) gasp_event_notify (рисунок 2), посредством которой реализации GAS 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 [10], для более детального мониторинга последовательных аспектов вычислений и производительности подсистемы памяти.

Рисунок 1. Высокоуровневая организация GAS приложения, выполняющегося в GASP реализации.

Callback 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 по-

зволяет 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 context,
                       unsigned int event_id,
                       gasp_evttype_t event_type,
                       const char *source_file,
                       int source_line, int source_col, ...);
```

Рисунок 2. 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. Также есть коллективные события, которые фиксируют такие операции, как broadcast, scatter и им подобные и события управления памятью в общей и приватной кучах (heap). Наконец, в инструменте предусмотрены средства для пользовательских, явно срабатываемых событий, что позволяет пользователю задавать контекст получаемым данным. Это облегчает фазовую профилировку и выборочное оснащение определенных сегментов кода.

Для тонкого контроля над оснащением и накладными расходами измерений предусмотрены несколько управляемых пользователем параметров. Во-первых, флаги компиляции --inst и --inst-local, использующиеся для запроса оснащения операций, исключая или включая события, генерируемые локальными обращениями к общей памяти. Так как обращения к локальной общей памяти зачастую такие же быстрые, как обычные операции доступа к памяти, оснащение этих событий может значительно увеличить накладные расходы на выполнения приложения. Тем не менее, информация о таких операциях в некоторых случаях полезна, например, при оптимизации локальности данных и выполнении оптимизации приватности, и может стоить дополнительных накладных расходов. Также имеют место директивы #pragma, которые позволяют пользователю давать инструкции компилятору избегать накладных расходов на оснащение для определенных лексических блоков кода в процессе компиляции. И наконец, функция программного контроля для управления сбором данных о работе программы в процессе ее выполнения.

Литература

1. Federal Plan for High-End Computing: Report of the High-End Computing Revitalization Task Force (HECRTF), 2004, http://www.nitrd.gov/pubs/2004_hecrtf/20040702_hecrtf.pdf.

2. Mohr B., Wolf F.: KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications. Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003). Klagenfurt, Austria (September 2003).
3. Shende S., Malony A.D.: TAU: The TAU Parallel Performance System. International Journal of High Performance Computing Applications. 20:2 (2006) 287-331.
4. UPC Consortium: UPC Language Specifications v1.2. Lawrence Berkeley National Lab Tech Report LBNL-59208 (2005).
5. Yelick K.A., Semenzato L., Pike G., Miyamoto C., Liblit B., Krishnamurthy A., Hilfinger P.N., Graham S.L., Gay D., Colella P., Aiken A.: Titanium: A High-Performance Java Dialect. Concurrency: Practice and Experience, 10:11-13 (1998).
6. Numrich B., Reid J.: Co-Array Fortran for Parallel Programming. ACM Fortran Forum. 17:2 (1998) 1-31.
7. DARPA High Productivity Computing Systems (HPCS) Language Effort http://www.highproductivity.org/.
8. Bell C., Bonachea D., Nishtala R., Yelick K.: Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. 20th International Parallel & Distributed Processing Symposium (IPDPS), 2006.
9. Datta K., Bonachea D., Yelick K.: Titanium Performance and Potential: an NPB Experimental Study. Languages and Compilers for Parallel Computing (LCPC), 2005.
10. Browne S., Dongarra J., Garner N., Ho G., Mucci P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications (IJHPCA), 14:3 (2000) 189-204.