Набор шаблонов неэффективного поведения для программной модели PGAS на примере языка UPC

Н. Е. Андреев, К. Е. Афанасьев
Кемеровский государственный университет, Россия
e-mail: nik@kemsu.ru, keafa@kemsu.ru

Представлен набор шаблонов неэффективного поведения для программной модели разделенного глобального адресного пространства. Данные шаблоны позволяют выполнять автоматизированный анализ производительности приложений, написанных в новой модели параллельного программирования, в частности на языке Unified Parallel C.

Ключевые слова: PGAS, UPC, параллельное программирование, трассировка, оценка производительности, эффективность, автоматизация, оптимизация.

Введение

Современные высокопроизводительные кластеры преодолели рубеж в петафлопс, а количество ядер, которое насчитывают такие системы, превышает десятки и сотни тысяч. Вместе с тем эффективность параллельных приложений на подобных системах по разным оценкам составляет не более 15 или даже $3-5\,\%$ [1] от пиковой производительности. Для более рационального использования потенциала дорогостоящих вычислительных установок сообществу специалистов в области высокопроизводительных вычислений необходимы инструменты анализа производительности, которые, с одной стороны, помогают увеличить эффективность и масштабируемость параллельных программ, а с другой — позволяют в большей мере сконцентрироваться на научной составляющей приложения, чем на кропотливом процессе оптимизации.

Современная суперкомпьютерная индустрия ставит перед собой рубеж в экзафлопс. Разработка комплексов параллельных программ для этих систем является весьма сложной задачей. Стандартом *де-факто* среди таких средств параллельного программирования 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) [2]. В модели PGAS используются односторонние коммуникации, где при передаче данных нет необходимости в явном отображении на двухсторонние пары send и receive, представляющем собой трудоемкий, подверженный ошибкам процесс, существенно влияющий на эффективность программирования. Обычное присвоение значения переменной массива $x[i] = a$ автоматически вызывает

необходимые коммуникации между узлами кластера. Программы, написанные в этой модели, проще для понимания, чем версии MPI, и имеют сравнимую с последними или даже более высокую производительность [3]. К группе PGAS относятся такие языки как SHMEM, Unified Parallel C (UPC), Co-Array Fortran (CAF), Titanium, Cray Chapel, IBM X10, Sun Fortress. Язык UPC [4] является наиболее "взрослым" представителем модели и применяется для решения ряда практических задач как в России [5 – 7], так и за рубежом [8 – 10]. Компиляторы для языка UPC разработаны всеми основными вендорами суперкомпьютерного рынка. Существуют реализации от компаний IBM, HP, Cray.

Среди множества задач, решаемых при помощи языка UPC и других языков модели PGAS, можно выделить следующие. А.А. Корж рассматривает распараллеливание на языке SHMEM бенчмарка Unstructured Adaptive из пакета NAS Parallel Benchmarks [5, 6]. В бенчмарке NPB UA решается задача Дирихле для уравнения теплопереноса в трехмерной кубической области на нерегулярной декартовой сетке. Источник тепла представляет собой шар, движущийся с постоянной скоростью. Для решения задачи применяется спектральный метод конечных элементов (SEM) с использованием метода конечных мортаров. Д.В. Андрюшин и А.С. Семенов решают задачу проверки выполнимости и поиска выполняющего набора функции в конъюнктивной нормальной форме (SAT-задача, Satisfiability problem) [7]. К SAT-задачам сводятся многие практически важные проблемы: синтез и верификация дискретных управляющих схем, теоретическое программирование, вычислительная биология, а также многочисленные криптографические задачи. При помощи языка UPC разработаны конечно-элементные решатели задач вычислительной гидродинамики для статических неструктурированных сеток BenchC [8] и динамических неструктурированных сеток XFlow [9], которые были использованы для решения таких задач как численное моделирование потоков воздуха за наземным военным транспортным средством, грузовым самолетом и беспилотным летательным аппаратом (БЛА), исследование "трепещущего полета" колибри для моделирования сверхмалых БЛА, моделирование аэродинамики парашюта. B. Gordon и N. Nguyen реализовали на языке UPC параллельный алгоритм дифференциального криптоанализа для взлома DES-подобных шифров [10].

Вместе с тем набор инструментальных средств модели параллельного программирования PGAS на сегодня небольшой. Такие известные инструменты как Intel Trace Analyzer, TAU, Cray Apprentice и др. работают с небольшим набором программных моделей, преимущественно с моделью передачи сообщений. Инструменты, применяемые, в частности, для языка UPC — upc_dump, upc_trace [11], имеют ограниченную функциональность, а PPW [12] (Parallel Performance Wizard) позволяет получать лишь статистические данные о работе программы. В результате разработчики, использующие новые модели параллельного программирования, зачастую в процессе оптимизации своих программ вынуждены вручную выполнять трудоемкий анализ.

1. Методы автоматизированного анализа

В большинстве существующих инструментов анализа производительности параллельных программ применяется ручной метод анализа, при котором пользователь самостоятельно выполняет поиск узких мест производительности приложения на основе диаграмм, графиков, таблиц и методов манипулирования ими (масштабирование и поиск). Представление информации о производительности программы в графическом ви-

де может быть эффективно, но большой объем отображаемых данных без сложных методов фильтрации снижает полезность подобных инструментов. В условиях постоянного роста количества ядер в современных суперкомпьютерах особенно актуальным становится вопрос разработки средств автоматизированного анализа, способных в той или иной степени самостоятельно находить узкие места в приложениях. Среди многих публикаций в этом направлении можно выделить [15 – 18, 20].

Анализ указанных работ был проведен в [13, 14]. B.P. Miller [15] предлагает две автоматические методики — анализ критического пути (АКП) и анализ фазового поведения (АФП). Первая позволяет найти самый длинный по времени путь, через который параллельная программа проходит в процессе своего выполнения. Предполагается, что оптимизация критического пути имеет максимальное влияние на сокращение времени выполнения программы. Анализ фазового поведения позволяет выделить различные фазы выполнения программы — инициализации, вычислений, вывода результатов и т. д. Каждая фаза имеет свои характеристики производительности. Задача АФП — автоматически выделить эти фазы, что дает возможность в дальнейшем выполнять оптимизацию, сконцентрировавшись на затратных по времени частях приложения.

J. Vetter [16] разработал метод автоматической классификации неэффективных коммуникаций для программ, написанных на MPI. Основа метода — подход, заимствованный из области машинного обучения. Новизна подхода в применении классификации на базе дерева решений. Перед началом анализа программы пользователь вручную обучает дерево решений на заранее подготовленных тестах эффективного и неэффективного поведения MPI-программ. В своей классификации автор уделяет внимание блокирующим и не блокирующим операциям отправки и приема сообщений, для которых формулируются семь типов неэффективных коммуникаций.

В основе подхода, предлагаемого H.L. Truong [17], — поиск и автоматическая классификация накладных расходов. Сначала программист работает с последовательной версией алгоритма, который разбирается на синтаксические блоки кода и для которого рассчитываются эталонные параметры производительности. Далее запускается параллельная версия. Методика позволяет выявить такие накладные расходы на выполнение программы как перемещение данных, синхронизация, операторы управления параллелизмом и др. Полученные данные можно проанализировать в сравнении с последовательной версией.

B.R. Helm [18] предлагает платформу для интеграции различных методов анализа параллельных программ и их автоматическую подборку в зависимости от используемых в программе пользователя алгоритмов. Разработчик может интегрировать свой метод в базу знаний, а пользователь выбрать наиболее подходящий для его задач инструмент. Строго говоря, Helm разработал не метод автоматизированного анализа, а организационный подход к анализу параллельных программ, который позволяет снизить нагрузку на разработчика.

Наиболее перспективным, по мнению авторов, является метод поиска шаблонов неэффективного поведения, предложенный F. Wolf и B. Mohr [20], которые совместно с рядом других ученых в рамках рабочей группы APART Esprit IV попытались собрать и классифицировать проблемы производительности для программных моделей передачи сообщений и общей памяти [21], что привело к формированию четкой методологии, позволяющей формализовать эти проблемы и описать алгоритмы их поиска в параллельных программах.

2. Идея метода поиска шаблонов неэффективного поведения

Основными методами, предлагаемыми сегодня инструментами анализа производительности параллельных программ, являются профилирование и ручной анализ временных шкал. Поиск узких мест в параллельных программах позволяет упростить автоматизированный анализ производительности. Большинство стратегий используют приемы и методы из области искусственного интеллекта — экспертные системы на основе баз знаний и методы автоматической классификации. Одним из последних является метод поиска шаблонов неэффективного поведения [20]. Суть его 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 представляет собой набор найденных в трассировочном файле событий, удовлетворяющих условиям возникновения некоторой ситуации, описываемой шаблоном. Такой способ представления шаблона позволяет фиксировать сложные ситуации, не охватываемые профилировочными инструментами и визуализаторами трасс. Поскольку подобные составные события обычно включают в себя сложные межсобытийные связи, необходимы высокоуровневые структуры данных, способные отслеживать и предоставлять в нужный момент такую информацию. Поэтому для работы с трассой удобно использовать некий инструмент доступа к трассировочным файлам, позволяющий упростить описание шаблонов, обеспечивающий произвольный доступ к различным событиям, а также представляющий некоторые абстракции, отражающие различные аспекты общего состояния выполнения программы и связи между событиями.

В [20] был предложен набор шаблонов неэффективного поведения для моделей передачи сообщений и общей памяти. Авторами настоящей статьи разработаны инструмент анализа производительности для языка Unified Parallel C и набор шаблонов неэффективного поведения для модели разделенного глобального адресного пространства, который был положен в основу инструмента. В работе представлены уже существующие шаблоны, адаптированные для программной модели PGAS, а также новые шаблоны, характерные только для этой программной модели. Реализация самого инструмента описана в [19].

3. Основа алгоритма

Компьютер изменяет свое состояние за дискретные интервалы времени (такты), поэтому процессы, происходящие в программе, можно выразить при помощи последовательности событий. Событие характеризует атомарное действие, произошедшее в определенном месте (процесс/нить) и в определенное время. Это позволяет описать динамику параллельного приложения. Однако анализ производительности обычно основывается на неатомарных событиях (например, коллективная операция), требующих множества событий для их описания. Чаще всего это события входа и выхода, которые можно легко связать с конкретным моментом времени. Место возникновения события определяется нитью, его сгенерировавшей. Событию каждого конкретного типа t присущ ряд атрибутов $\{a_1, ..., a_n\}$. Часто разные события имеют несколько общих атрибутов, поэтому удобно сформировать иерархию, состоящую из событий, которые наследуют свои атрибуты от абстрактных типов.

3.1. Типы событий

Минимальным набором атрибутов любого события *Event* будут время *time* и место *loc* его возникновения (процесс/нить). События собираются в трассы — конечное пронумерованное множество событий $E := \{e_1, ..., e_n\}$, где нумерация задает временной порядок событий в трассе $i < j \Rightarrow e_i.time \leq e_j.time$, $i < j \wedge e_i.loc = e_j.loc \Rightarrow e_i.time < e_j.time$.

В процессе работы PGAS-приложения один и тот же код программы выполняется в нескольких нитях. Идентификатор нити можно узнать из константы MYTHREAD, определенной в каждом экземпляре исполняемого кода. Таким образом, нить *loc*, в которой произошло событие, — это число из $L := \{0, ..., n_t - 1\}$, где n_t — общее количество нитей, с которым была запущена программа.

Программу, написанную на одном из PGAS-языков, можно представить в виде множества блоков кода. В отличие от спецификации языка UPC, где под блоком подразумевается множество элементов массива данных, принадлежащих одной нити, здесь блок — это некоторый набор операторов языка UPC, которым могут быть функция, цикл или произвольная часть программы. Предполагается, что блок кода может быть покинут только тогда, когда произошел выход из всех блоков кода, заключенных в него. Иначе говоря, исполнение программы в каждой нити представляет собой некоторую вложенную структуру вызовов. События *Enter* и *Exit* означают соответственно вход и выход из блока кода (рис. 1). Оба эти события являются потомками абстрактного типа *RegionEvent*, имеющего атрибут *reg*, который идентифицирует именем блок кода, куда был осуществлен вход или выход. Кроме того, событие *Enter* несет атрибут *csite*, позволяющий узнать место в программе, из которого был осуществлен вход в блок кода, т. е. в какой строке программы была вызвана та или иная функция.

В программной модели PGAS присутствуют операции релокализации данных — аналог коллективных коммуникационных операций MPI, использующихся в ситуациях, когда в программе происходит изменение свойств локальности и необходимо перераспределение данных. Моделирование таких операций включает в себя два аспекта. Во-первых, операция релокализации — коллективная, охватывающая события из нескольких нитей. Во-вторых, коллективная операция подразумевает обмен данными, но детали этого обмена скрыты в реализации компилятора и среды времени выполнения (runtime), что не позволяет явно описать происходящие события. Поэтому вводится составное событие *CExit*, которое наследует событие *Exit* и имеет дополнительные атрибуты — количества отправленных *sent* и полученных *recvd* каждой нитью *loc* данных, а также корневая *root* нить операции, если таковая имеется. В итоге коллективная операция моделируется одним событием *Enter* и одним событием *CExit* в каждой нити.

Программная модель PGAS также включает в себя синхронизацию на основе блокировок. В этом случае для захвата и освобождения блокировки используются события

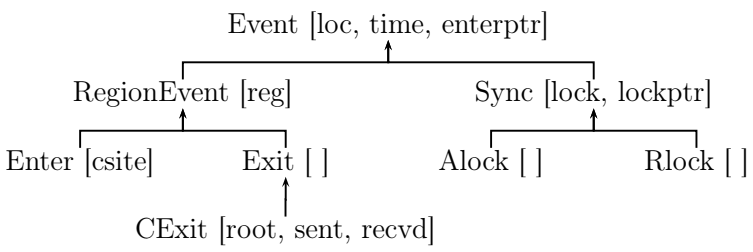

Рис. 1. Иерархия типов событий

Alock и *Rlock*. Очевидно, что событие *Alock* происходит всегда до соответствующего ему события *Rlock*, причем оба события включают в себя атрибут *lock*, содержащий идентификатор блокировки, над которой произошли действия. Этот атрибут наследуется от общего типа *Sync*, объединяющего события, оперирующие блокировками. События *Sync* обычно помещаются между событиями *Enter* и *Exit* соответствующей библиотечной функции PGAS-языка. В *Event* и *Sync* также присутствуют атрибуты-указатели — *enterptr* и *lockptr* соответственно. Определения этим атрибутам даны далее.

3.2. Состояния и атрибуты-указатели

Единичные события, происходящие в разных нитях параллельного приложения, сами не позволяют понять, в чем причина неэффективной работы программы. Например, параллельный профилировщик способен показать, что программа затратила большой объем времени на синхронизацию нитей в операциях захвата и освобождения блокировок, но было ли это вызвано просто большим количеством обращений к соответствующим функциям языка программирования или нити действительно соревновались за одну и ту же блокировку — не понятно. Чтобы выявить источник проблемы, инструмент анализа производительности должен основываться на составных событиях, т. е. выполнять пространственно-временной анализ трассы. Здесь под пространством понимается множество нитей, в которых возникают события, а под временем — время их возникновения в каждой конкретной нити. Для выражения сложных зависимостей между компонентами составного события введем две категории абстракций: последовательности состояний и атрибуты-указатели.

Составное событие представляет собой 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. Начальное состояние δ_0 — всегда пустое множество, $\delta_i,\ i > 0$, — состояние программы на момент возникновения события e_i, содержащее предшествующие e_i события $\delta_i \subseteq \{e \in E \mid e \leq e_i\}, 1 \leq i \leq n_e$. Для анализа PGAS-приложений достаточно поддерживать три состояния программы: стек блоков кода, очередь операции релокализации и статус владения.

Для любой программы в процессе ее выполнения обычно выделяются следующие области памяти: программного кода, 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. Если в трассе встречается событие входа в функцию, оператор перехода r^l_{Enter} добавляет это событие в состояние. Когда в трассе встречается событие выхода из функции, соответствующее ему событие входа удаляется из состояния оператором r^l_{Exit}. Таким образом, для оператора r^l_{Enter} областями определения и значений будут события типа *Enter*. Для оператора r^l_{Exit} областью определения будут события *Exit*, а областью значений — события *Enter*.

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 закончат их выполнение. События выхода каждой нити из коллективной операции накапливаются в состоянии

оператором c_{CExit}. Областью определения и значений для него будут события $CExit$. Как только в состоянии оказываются все события выхода, выполняется анализ и состояние очищается оператором c_{Event}. Областью определения оператора c_{Event} является множество событий $Event$. Область значений оператора — множество событий $CExit$.

Последнее состояние "Статус владения" необходимо для анализа синхронизации на основе блокировок. Блокировка (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. Если в трассе встречается событие изменения статуса $Alock$ или $Rlock$, то оператор перехода o^k_{Sync} помещает это событие в состояние, которое всегда содержит только последнее событие изменения статуса. Так как все события типа $Sync$ связаны между собой атрибутом-указателем $lockptr$, то это является достаточным. Определение атрибута $lockptr$ дано ниже. Областью значений и определения для оператора o^k_{Sync} является множество событий $Sync$.

Прежде чем перейти к формальному описанию рассматриваемых состояний дадим следующее определение. Пусть $F \in E$ — подмножество трассы, а $l \in L$ — место вызова. Тогда

$$\begin{aligned}
mostrcnt(F) &:= \{e \in F \mid \neg\exists f \in F : f.time > e.time\},\\
leastrcnt(F) &:= \{e \in F \mid \neg\exists f \in F : f.time < e.time\},\\
eqloc(F,l) &:= \{e \in F \mid e.loc = l\},\\
gtloc(F,l) &:= \{e \in F \mid e.loc > l\},\\
ltloc(F,l) &:= \{e \in F \mid e.loc < l\}.
\end{aligned}$$

Первые две функции возвращают те события множества F, которые произошли раньше всех и позже всех; $eqloc()$, $gtloc()$ и $ltloc()$ в свою очередь возвращают события, произошедшие в нити/нитях с идентификатором, равным, большим и меньшим l соответственно. Для удобства множество событий, возвращаемое перечисленными функциями, можно рассматривать как одно, если оно содержит единственный элемент. Перейдем к описанию состояний.

Состояние "Стек блоков кода" задается для каждой нити $l \in L$ и содержит события входа для активных блоков кода нити. Оператор перехода для данного состояния можно задать при помощи функций

$$\begin{aligned}
r^l_{Enter} &: R^l_i := \begin{cases} R^l_{i-1} \bigcup \{e_i\}, & \text{если } e_i.loc = l,\\ R^l_{i-1}, & \text{иначе,}\end{cases}\\
r^l_{Exit} &: R^l_i := \begin{cases} R^l_{i-1} \backslash mostrcnt(R^l_{i-1}), & \text{если } e_i.loc = l,\\ R^l_{i-1}, & \text{иначе.}\end{cases}
\end{aligned}$$

Стек представляет собой типичную структуру как для последовательных, так и для параллельных программ. Задача функций r^l_{Enter} и r^l_{Exit} добавлять и удалять из состояния блоки кода (обычно функции), которые нить посещает в процессе своего выполнения.

Состояние "Очередь операции релокализации" содержит события выхода из активной в настоящий момент коллективной операции релокализации данных. В каждый момент времени в параллельной программе может выполняться только одна такая

операция, что позволяет всегда поддерживать целостность состояния. Когда в очередь попадут события выхода всех нитей, операция считается завершенной и выполняется ее анализ. Операторы перехода задаются следующим образом:

$$c_{CExit} \quad : \quad C_i := C_{i-1} \bigcup \{e_i\},$$

$$c_{Event} \quad : \quad C_i := \begin{cases} C_{i-1} \backslash Inst, & \text{если } \forall l \in L :\mid eqloc(C_{i-1}, l) \mid \geq 1, \\ C_{i-1}, & \text{иначе,} \end{cases}$$

$$\text{где } Inst := \bigcup_{l \in L} leastrcnt(eqloc(C_{i-1}, l)).$$

В модели передачи сообщений MPI есть понятие коммуникаторов, которые разбивают процессы на коммуникационные домены. В модели PGAS такое понятие отсутствует, поэтому задается одно общее для всех нитей состояние, в которое посредством оператора c_{CExit} попадают все события выхода из коллективной операции реалокализации. После того как все события выхода попали в C, состояние очищается при помощи c_{Event}. Заметим, что $Event$ в отличие от $CExit$ — общее событие, поэтому оператор c_{Event} должен применяться раньше c_{CExit}. При достижении последнего события e_i выхода сначала применяется c_{Event}. Так как последнее событие еще не стало частью C_i, c_{Event} не производит никакого эффекта. После попадания e_i в C_i состояние полностью содержит экземпляр операции, и условие, позволяющее c_{Event} очистить C_i, становится верным. Наконец, после перехода к следующему событию снова вызывается c_{Event} и весь экземпляр удаляется.

Для доступа к событиям, принадлежащим экземпляру операции релокализации, удобно задать функцию, которая позволяет выделить эти события:

$$coll(e) := \begin{cases} Inst, & \text{если } type(e) = CExit \wedge \\ & \wedge \forall l \in L :\mid eqloc(C_i, l) \mid \geq 1, \\ \varnothing, & \text{иначе,} \end{cases}$$

$$\text{где } Inst := \bigcup_{l \in L} leastrcnt(eqloc(C_{i-1}, l)).$$

Если e — событие типа $CExit$, завершающее операцию релокализации, то $coll()$ возвращает все события $CExit$ операции. Иначе возвращается пустое множество.

Состояние “Статус владельца” позволяет отслеживать историю владения блокировкой. Пусть K — множество всех блокировок параллельной программы. Тогда статус владения O^k для блокировки $k \in K$ представляет собой состояние, которое содержит последнее событие изменения статуса владения k. Оператор перехода задается следующим образом:

$$o^k_{Sync} : O^k_i := \begin{cases} \{e_i\}, & \text{если } e_i.lock = k, \\ O^k_{i-1}, & \text{иначе.} \end{cases}$$

Здесь $Sync$ — событие захвата или освобождения блокировки. Это состояние вспомогательное и не используется напрямую. Далее определение будет использовано при описании атрибута-указателя, связывающего события, которые вместе составляют историю изменения владения блокировкой.

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 $e_i \in E$ указывает на событие входа в блок кода, где произошло событие e_i:

$$e_i.enterptr := \begin{cases} mostrcnt(R_{i-1}^{e_i.loc}), & \text{если } R_{i-1}^{e_i.loc} \neq \varnothing, \\ null, & \text{иначе.} \end{cases}$$

Указатель lockptr — это атрибут события синхронизации $e_i \in E_{Sync}$, указывающий на предыдущее событие синхронизации, оперировавшее той же самой блокировкой:

$$e_i.lockptr := \begin{cases} e_j, & \text{если } O_{i-1}^{e_i.lock} = \{e_j\}, \\ null, & \text{иначе.} \end{cases}$$

Очевидно, что атрибут *lockptr* события захвата блокировки всегда указывает на событие ее освобождения и наоборот. Только первое событие захвата блокировки указывает на *null*, поэтому оно начинает историю владения. Определив все необходимые состояния, операторы перехода и атрибуты-указатели, перейдем к описанию шаблонов.

4. Описание шаблонов

Метод поиска шаблонов неэффективного поведения предназначен для поиска проблем производительности, которые возникают из-за задержек в процессе взаимодействия нитей. Синтаксис языка UPC содержит следующие операции, требующие взаимодействия двух и более нитей: upc_put(), upc_get(), upc_barrier(), upc_notify(), upc_wait(), upc_memget(), upc_memput(), upc_memcpy(), upc_fence(), upc_lock(), upc_unlock(), а также все коллективные коммуникационные операции. Чтобы сформулировать набор шаблонов, полностью покрывающий задержки UPC-программ, необходимо проанализировать рассмотренные выше операции согласно их семантике на предмет потенциального возникновения задержек.

Программная модель PGAS представляет собой некий синтез моделей передачи сообщений и общей памяти. С точки зрения программиста память — это единое адресное пространство, но на низком уровне работа с памятью реализована при помощи передачи блоков данных через коммуникационную сеть кластера. Однако проблем модели передачи сообщений, связанных с блокирующими посылками, удалось избежать. Односторонние коммуникационные операции upc_put()\upc_get(), лежащие в основе языка UPC и модели PGAS в целом, не блокируют выполнение программы, как это часто бывает в приложениях, написанных на MPI. Это решает ряд проблем производительности программ, которые хорошо описаны в методе поиска шаблонов неэффективного поведения для библиотеки MPI, и позволяет исключить операции upc_put()\upc_get() из анализа.

Одной из типичных причин задержек выполнения программы является барьерная синхронизация, присутствующая практически во всех языках параллельного программирования, в том числе в UPC, где она представлена операциями upc_barrier() и upc_notify()\upc_wait(). В первом случае программа блокируется до тех пор, пока все нити не достигнут операции upc_barrier(), во втором случае, который называется барьером с расщепленной фазой, синхронизация разбивается на два этапа. По окончании вычислений, требующих синхронизации, нить выполняет операцию upc_notify(),

чтобы проинформировать другие нити о своем состоянии. Дальше она может заняться локальными вычислениями, и когда последние также будут готовы, нить выполняет upc_wait(), чтобы остановиться и дождаться, пока остальные нити выполнят upc_notify(). Когда все остальные нити выполнят upc_notify(), нить, ожидающая в операции upc_wait(), освобождается и может перейти к следующей фазе вычислений, не дожидаясь остальных. Несмотря на то что барьер с расщепленной фазой более эффективен с точки зрения производительности программ, в обоих случаях возникает задержка либо в функции upc_barrier(), либо в upc_wait(). Поэтому данные операции требуют анализа и представлены в виде шаблонов "Ожидание в барьере" и "Завершение барьера".

Операции upc_memget(), upc_memput() и upc_memcpy() в отличие от upc_put()\ upc_get(), которые выполняют пересылку переменных, предназначены для передачи непрерывных массивов данных. Недостаток этих операций с точки зрения производительности в том, что они блокирующие. Нить останавливает свое выполнение, до тех пор, пока передача данных не завершится. Однако такие операции односторонние и не требуют явного участия второй нити, а значит отсутствуют и какие-либо связи между событиями, поэтому время, потерянное в них, можно найти, используя обычный механизм профилировки.

От модели общей памяти в UPC остались проблемы, связанные с использованием синхронизации на основе блокировок. Основным источником задержек в UPC являются операции релокализации данных, среди которых upc_all_broadcast(), upc_all_scatter(), upc_all_gather(), upc_all_gather_all(), upc_all_exchange(), upc_all_permute(), upc_all_reduce() и upc_all_prefix_reduce(). В отличие от односторонних коммуникационных операций в коллективных операциях присутствуют сложные зависимости по данным, требующие синхронизации. Если программа написана неудачно и четкая координация между нитями нарушена, то неизбежно возникнут дополнительные задержки. Чтобы свести эту проблему к минимуму, в UPC для всех операций релокализации были выделены три типа синхронизации, которые указываются последним аргументом в вызове функций отдельно для синхронизации на входе в операцию и на выходе из нее. Дадим определение каждому из них: UPC_IN_NOSYNC — коллективная функция имеет право считывать или записывать любые данные, но только после того как в вызов коллективной операции вошла хотя бы одна нить; UPC_IN_MYSYNC — коллективная функция имеет право считывать или записывать данные только тех нитей, которые уже вошли в операцию; UPC_IN_ALLSYNC — коллективная функция имеет право считывать или записывать данные только после того, как все нити вошли в коллективную операцию; UPC_OUT_NOSYNC — коллективная функция имеет право считывать и записывать данные до тех пор, пока последняя нить не вышла из операции; UPC_OUT_MYSYNC — нить может выйти из коллективной функции только тогда, когда все операции чтения и записи с данными этой нити завершены; UPC_OUT_ALLSYNC — прежде чем выйти из коллективной операции, нить должна дождаться завершения всех операций чтения и записи данных.

Если алгоритм программы составлен таким образом, что результат работы операции релокализации не требует координации нитей или для синхронизации используются барьеры, то можно полностью отказаться от синхронизации, указав флаги UPC_IN_NOSYNC и UPC_OUT_NOSYNC. Если требуется, чтобы функция могла оперировать данными только тех нитей, которые уже вошли в операцию релокализации, а эти нити не могли выйти из операции до тех пор, пока остальные не завершили работу

с их данными, то используются флаги UPC_IN_MYSYNC и UPC_OUT_MYSYNC. UPC_IN_ALLSYNC и UPC_OUT_ALLSYNC представляют собой самый неэффективный тип синхронизации, когда все нити должны дождаться друг друга на входе или на выходе. С учетом указанных особенностей языка Unified Parallel C авторами были разработаны 12 шаблонов неэффективного поведения, позволяющих выявлять сложные межпроцессные зависимости, не анализируемые профилировщиками и визуализаторами трасс.

Шаблон "Конкуренция за блокировку" возникает, если одна из нитей пытается захватить блокировку, которой уже владеет другая нить (рис. 2). Шаблон является типичным для модели общей памяти, однако он хорошо прилагается к программной модели PGAS, так как в последней используется модель распределенной общей памяти. Сложное составное событие в данном случае состоит из трех простых событий $\{\{e_1\},\{a_1\},\{r_0\}\}$ — входа в функцию upc_lock() в нити 1, захвата блокировки в нити 1 и освобождения в нити 0. Используя атрибуты-указатели, можно найти все компоненты составного события. Поскольку событие захвата блокировки всегда указывает на предыдущее событие освобождения, то a указывает на r. В свою очередь $a.enterptr$ указывает на e, так как e — событие входа в соответствующий блок кода. Однако чтобы полностью описать шаблон, необходимо сформулировать ограничение на временной порядок компонентов. Иначе говоря, чтобы классифицировать ситуацию как неэффективное поведение, e должно произойти раньше r. Формально условие срабатывания шаблона выглядит следующим образом:

$$r := a.lockptr,$$
$$e := \begin{cases} a.enterptr, & \text{если } a.enterptr.time < r.time, \\ \text{неудача}, & \text{иначе.} \end{cases}$$

Потерянное время определяется вычитанием временной метки e из r. Кроме того, спецификация позволяет выяснить идентификатор нити, владевшей блокировкой $r.loc$, в то время, когда нить $a.loc$ пыталась ее захватить. Это дает возможность получить более конкретную информацию об обстоятельствах возникновения данной ситуации.

Шаблон "Синхронизация на входе в коллективную операцию" справедлив для тех операций релокализации, в которых используется тип синхронизации UPC_IN_ALLSYNC (рис. 3, *a*). Предположим, что это операция широковещательной рассылки upc_all_broadcast(). Согласно указанному типу синхронизации, каждая нить обязана дождаться на входе всех остальных. Вхождение нити в операцию релокализации в разные моменты времени вносит нежелательные накладные расходы на синхронизацию. Шаблон не встречается в программной модели передачи сообщений и характерен только для языка UPC. Это объясняется тем, что стандарт MPI не накладывает строгих ограничений на порядок входа нитей в коллективную операцию. Будут ли нити

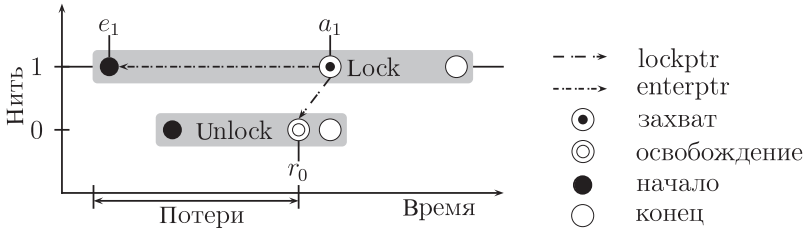

Рис. 2. Конкуренция за блокировку

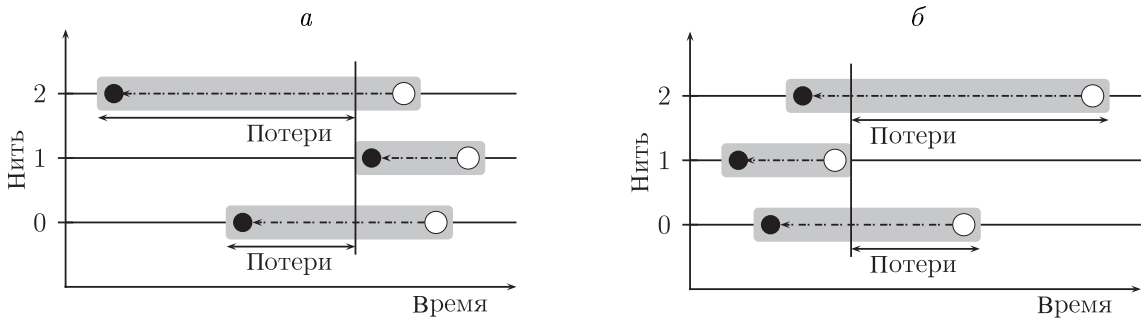

Рис. 3. Синхронизация на входе в коллективную операцию (*а*) и на выходе из нее (*б*)

ожидать друг друга, как при использовании барьерной синхронизации, или же будет использован более оптимальный алгоритм, определяется реализацией, и этого нельзя отследить. В случае с UPC синхронизация указывается программистом явно, что позволяет сформулировать новые более точные шаблоны. Отправной точкой здесь является событие выхода из операции последней нитью ($last$). Это отражено в следующем предикате:

$$type(last) = CExit \wedge$$
$$\wedge\ coll(last) \neq \varnothing \wedge$$
$$\wedge\ last.reg = upc\_all\_broadcast.$$

Составное событие $\{\{last\}, E_1, E_2\}$ включает в себя три части: отправное событие $last$, множество событий $CExit$ для операции upc_all_broadcast() E_1 и все события входа E_2. E_1 получено при помощи функции $coll()$. Затем находится E_2 путем отслеживания всех атрибутов-указателей $enterptr$ для множества E_1. Для удобства $E_1.attr$ используется как сокращение для $\{v \mid \exists e \in E_1 : e.attr = v\}$:

$$E_1 := coll(last),$$
$$E_2 := \begin{cases} E_1.enterptr, & \text{если } \exists e_i, e_j \in E_1.enterptr : e_i.time \neq e_j.time, \\ \text{неудача}, & \text{иначе.} \end{cases}$$

Очевидно, что шаблону будет соответствовать почти любая коллективная операция upc_all_broadcast(), так как нити никогда не входят в операцию одновременно. Однако шаблон позволяет точно узнать источник и объем накладных расходов в каждом конкретном случае. Время, потерянное на синхронизации, рассчитывается следующим образом:

$$wasted = \sum_{e \in E_2} \max(E_2.time) - e.time.$$

Шаблон “Синхронизация на выходе из коллективной операции” справедлив для тех операций релокализации, в которых используется тип синхронизации UPC_OUT_ALLSYNC (рис. 3, *б*), и также характерен только для языка UPC. Как и в шаблоне с синхронизацией на входе, все нити обязаны дождаться друг друга, но теперь на выходе из операции. Отправной точкой здесь также является событие выхода из операции последней нитью ($last$). Так как условия срабатывания шаблона на входе и выходе практически аналогичны, приведем лишь формулы для нахождения исходных данных и расчета времени, потерянного на синхронизацию:

$$E_1 := coll(last),$$
$$wasted = \sum_{e \in E_1} \max(E_1.time) - e.time.$$

Шаблон "Ожидание в барьере" возникает в ситуациях, когда в программе используется барьерная синхронизация. Если в приложении встретился барьер, то все нити должны остановиться и дождаться друг друга. Этот шаблон очень похож на "Синхронизацию на входе в коллективную операцию", поэтому его описание не приводится. Отметим, что шаблон типичен как для OpenMP и MPI, так и для UPC.

Шаблон "Завершение барьера" — достаточно специфический в том смысле, что в нормальной ситуации все нити должны выходить из барьера в один и тот же момент времени. Любое даже незначительное время, проведенное в этом шаблоне, может означать неэффективность реализации PGAS-языка либо наличие помех со стороны других процессов, работающих на том же расчетном узле. Формальное описание шаблона совпадает с описанием шаблона "Синхронизация на выходе из коллективной операции". Шаблон типичен для всех трех программных моделей.

Шаблон "Поздняя рассылка" возникает при использовании синхронизации UPC_IN_MYSYNC на входе в операции *один ко многим*, к которым относятся такие как upc_all_broadcast() и upc_all_scatter() (рис. 4, *а*). Если нить, рассылающая данные, входит в операцию позднее нитей, принимающих данные, то последние должны приостановить свое выполнение. Шаблон отражает время, потерянное в результате возникновения такой ситуации. Впервые он был описан для языка MPI, который имеет большой набор коллективных операций, в том числе операций *один ко многим*. Шаблон можно применить к языку UPC с учетом его собственных коллективных функций. В данном случае необходимо найти событие входа в операцию нитью источником данных. Для этого достаточно воспользоваться атрибутом *root* любого события, являющегося частью коллективной операции. Чтобы шаблон сработал, необходимо также проверить, есть ли нити, вошедшие в операцию раньше:

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

Для расчета потерянного времени необходимо учесть только те события входа, которые произошли раньше события входа нитью *root*:

$$
wasted = \sum_{\substack{e \in E_2 \\ e.time < r.time}} r.time - e.time.
$$

Шаблон "Ранняя сборка" присущ операциям, выполняющим сборку данных, таким как upc_all_gather() и upc_all_reduce(), если для синхронизации на входе ис-

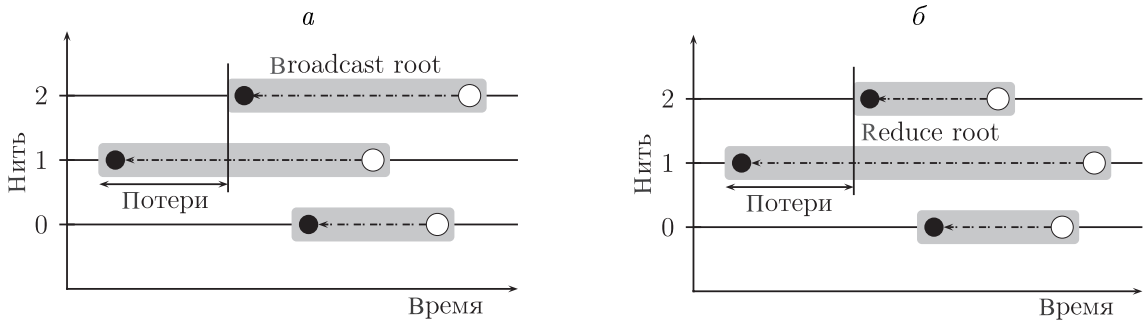

Рис. 4. Поздняя рассылка (*а*), ранняя сборка (*б*)

пользуется UPC_IN_MYSYNC (рис. 4, *б*). Этот шаблон аналогичен шаблону “Поздняя рассылка” за тем исключением, что причиной его возникновения является нить — получатель данных в случае, если она входит в операцию позднее других. Шаблон типичен для языков MPI и UPC.

Шаблон “Ранняя префиксная редукция” уникален для операции префиксной редукции upc_all_prefix_reduce() (рис. 5, *а*). Возникает при использовании синхронизации UPC_IN_MYSYNC и не встречается в языках MPI и OpenMP. В этой операции результат редукции в нити n зависит от редукции, выполненной в нити $n-1$. Если хотя бы одна из нитей $0, ..., n-1$ не вошла в операцию, нить n должна ждать. Чтобы в E_2 попали все события входа в коллективную операцию и потерянное время не равнялось нулю, достаточно существования хотя бы одного события входа, которое бы произошло позже события входа в нити с бóльшим идентификатором:

$$E_1 := coll(last),$$
$$E_2 := \begin{cases} E_1.enterptr, & \text{если } \exists e_i, e_j \in E_1.enterptr : e_i.time > e_j.time \ \wedge \ e_i.loc < e_j.loc, \\ \text{неудача}, & \text{иначе.} \end{cases}$$

Для расчета времени, потерянного в шаблоне, необходимо для события входа e каждой нити найти самое позднее событие входа среди нитей с меньшим идентификатором. С этой целью можно воспользоваться функцией $ltloc()$

$$wasted = \sum_{e \in E_2} max(Sub_{e.loc}.time) - e.time,$$
$$\text{где } Sub_{e.loc} = ltloc(E_2, e.loc).$$

Шаблон “Синхронизация на входе в коллективную операцию *многие ко многим*” в отличие от шаблона “Синхронизация на входе в коллективную операцию” возникает при использовании синхронизации UPC_IN_MYSYNC и только в операциях, отправляющих данные от многих нитей ко многим, к которым относятся upc_all_gather_all() и upc_all_exchange(). Если во втором случае все нити ждут друг друга, то здесь они имеют право работать с данными нитей, уже вошедших в коллективную операцию. Тем не менее, если некоторые нити еще не вошли в операцию, а с остальными обмен уже произошел, нить остановится. Шаблон описывает потерянное в такой ситуации время и встречается также в языке MPI. Искомые данные и потерянное время находятся так же, как в шаблоне “Синхронизация на входе в коллективную операцию”. Важно отметить, что шаблон не достаточно точен, так как часть времени, попадающего в категорию потерянного, является полезным, — это время, потраченное на обмен с нитями, уже вошедшими в операцию. На практике невозможно определить, какую часть

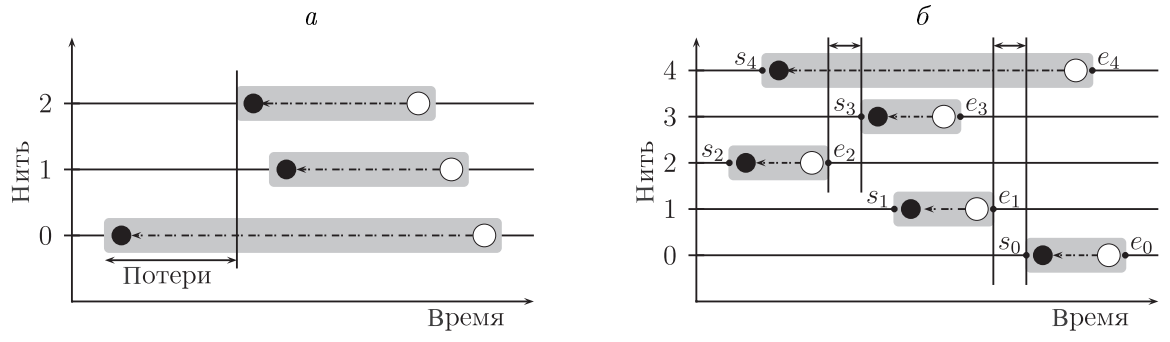

Рис. 5. Ранняя префиксная редукция (*а*), ожидание внутри коллективной операции (*б*)

времени нить обменивалась данными, а какую часть времени находилась в ожидании. Поэтому данный шаблон можно рассматривать лишь как подозрение на неэффективное поведение. Если программа на такую ситуацию затратила значительную часть времени, то нужно более внимательно проанализировать проблему.

Шаблон "Синхронизация на выходе из коллективной операции *многие ко многим*" аналогичен предыдущему с тем отличием, что синхронизация и соответственно потеря времени происходят на выходе из операции. Шаблон встречается в языках MPI и UPC.

Шаблон "Ожидание внутри коллективной операции" возникает в операциях upc_all_broadcast(), upc_all_scatter(), upc_all_gather() и upc_all_reduce() при использовании синхронизации типа UPC_IN_MYSYNC (рис. 5, *б*). Этот шаблон является дополнением к шаблонам "Поздняя рассылка" и "Ранняя сборка". При выполнении коллективной операции в языке UPC может возникнуть ситуация, когда нить находится в коллективной операции одна. Это происходит, например, если нить входит в операцию первой либо выходит из нее последней или, если часть нитей уже выполнила вычисления и вышла, но некоторые нити еще не успели дойти до коллективной операции. Такая ситуация может возникнуть и в языке MPI, однако она не описана в оригинальном методе.

Предположим, что корневой нитью коллективной операции является нить с идентификатором 4 (см. рис. 5, *б*). Потерянное время для этой нити в данном случае рассчитывается как

$$wasted = (s_3.time - e_2.time) + (s_0.time - e_1.time).$$

Для нахождения потерянного времени в общем случае достаточно из времени выполнения операции корневой нитью вычесть время выполнения операции другими нитями, пересекающееся с корневой. Рассмотрим три ситуации взаимного расположения корневой нити относительно других нитей во времени:

1) если нить начинает выполнение раньше корневой (см. нить 2 на рис. 5, *б*), то это время $(s_4.time - s_2.time)$ не учитывается при нахождении разницы;

2) если времена выполнения операции двумя или более нитями пересекаются (см. нити 1 и 3 на рисунке), то они объединяются и далее вычитается совокупный отрезок $([s_3, e_1])$;

3) если нить начинает выполнение позже корневой (см. нить 0 на рисунке), то это время $(e_0.time - e_4.time)$ не учитывается при нахождении разницы.

Шаблон "Ожидание в операции динамического выделения памяти" возникает в операции *upc_all_alloc*() и характерен только для языка UPC. В UPC присутствуют ряд операций для 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 должны ждать рассылки результата, поэтому описание данного шаблона аналогично описанию шаблона "Поздняя рассылка".

5. Тестирование

Разработанный набор шаблонов был реализован в виде инструментального средства автоматизированного анализа производительности [19]. Инструмент состоит из набора

утилит для компиляции, запуска, анализа и визуализации результатов. Для компиляции и запуска приложения разработаны скрипты-обертки вокруг компилятора Berkeley UPC. Сбор данных о работе программы выполняется специально разработанной измерительной библиотекой, которая подключается к программе в момент компиляции. Модуль анализа выполняет поиск шаблонов неэффективного поведения, а также расчет профилировочных данных. Для визуализации результатов используется утилита Cube из пакета Scalasca.

Метод был протестирован на алгоритмах блочной сортировки целых чисел и быстрого преобразования Фурье из пакета NAS Parallel Benchmarks, разрабатываемого NASA. Версия для языка UPC разработана Университетом Джорджа Вашингтона. Тестирование проводилось на кластере Мельбурнского Технологического Института. Узлы кластера имеют по два четырехъядерных процессора AMD Opteron 2356, 32 ГБ оперативной памяти и объединены коммуникационной сетью InfiniBand. Для компиляции приложений использовались Berkeley UPC 2.10 и Sun Studio Express 12.

Алгоритм сортировки целых чисел применяется 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 коммуникационных операций. Задача отличается тем, что в ней не используются операции с плавающей точкой, однако присутствует большой объем коммуникаций.

В результате анализа алгоритма были обнаружены шаблоны “Ожидание в барьере” и “ Ожидание в операции динамического выделения памяти”. Оказалось, что первая и последняя нити провели в операции *upc_all_alloc*() времени значительно меньше среднего, т. е. выполняли ее быстрее других. Поскольку 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, который показывает общее время, потерянное программой ввиду подобного неэффективного поведения. Кроме того, в приложении было обнаружено большое количество операций передачи непрерывных массивов данных.

После оптимизации приложения выполнены расчеты и сравнение времени выполнения, ускорения и эффективности оригинальной и оптимизированной версий. В последнем столбце таблицы указана разница между временем их выполнения. Как видно, максимальный эффект от оптимизации приходится на 32 нити и составляет около 30 %.

Результаты оптимизации IS и FT

Количество нитей	IS			FT		
	Время выполнения оригинальной версии, с	Время выполнения оптимальной версии, с	Δ, %	Время выполнения оригинальной версии, с	Время выполнения оптимальной версии, с	Δ, %
1	25.15	24.84	1.22	592.15	580.33	2.00
2	13.17	12.83	2.58	373.55	370.02	0.94
4	7.50	6.80	9.28	187.32	181.99	2.85
8	5.18	4.24	18.13	111.00	103.18	7.05
16	5.04	3.73	25.94	83.60	74.94	10.36
32	5.51	3.96	28.10	73.43	64.50	12.17

При помощи алгоритма быстрого преобразования Фурье решается трехмерное дифференциальное уравнение. Алгоритм используется во многих областях науки — в физике, теории чисел, комбинаторике, обработке сигналов, теории вероятностей, статистике, криптографии, акустике, океанологии, оптике, геометрии и мн. др. Поэтому анализ и оптимизация реализации данного алгоритма для языка UPC, как и для любого другого, имеет практическую значимость.

Анализа предложенной программы показал, что код содержит излишнюю синхронизацию, на что указывает значительное время, полученное для шаблона "Ожидание в барьере". Существенные затраты времени приходятся также на операции перемещения данных.

В результате оптимизации приложения для 32 нитей было получено ускорение алгоритма на 12 %. В таблице приведены расчетные значения времени.

Версии алгоритмов, оптимизированные авторами, вошли в исходный пакет — UPC NPB.

Заключение

Инструменты анализа производительности оказывают помощь разработчикам параллельных приложений в оптимизации параллельного кода. Большинство существующих инструментов основаны на методах ручного анализа. В работе был представлен набор шаблонов, построенных на основе метода поиска шаблонов неэффективного поведения. Данный набор уже содержит в себе описание распространенных ситуаций, которые могут привести к снижению производительности параллельной программы. При возникновении такой ситуации программист сразу получает ответ на вопрос, что является причиной медленной работы программы. Это означает, что нет необходимости вручную просматривать длинные временные шкалы (timeline), как, например, в TAU, что особенно актуально в условиях постоянного роста количество ядер в современных кластерах и увеличения сложности параллельных комплексов. Следует также отметить, что шаблоны разработаны для языка Unified Parallel C, который является одним из представителей программной модели Partitioned Global Address Space — нового витка в развитии языков параллельного программирования.

Список литературы

[1] High-End Computing Revitalization Task Force. Federal Plan for High-End Computing // `http://www.nitrd.gov/pubs/2004_hecrtf/20040702_hecrtf.pdf`, 2004.

[2] DARPA, et al. High Productivity Computing Systems // `http://www.highproductivity.org/`, 2004.

[3] Bell C., Bonachea D., Nishtala R., Yelick K. Optimizing bandwidth limited problems using one-sided communication and overlap // 20th Intern. Parallel & Distributed Proc. Symp. Rhodes Island, 2006.

[4] El-Ghazawi T., Carlson W., Draper J. UPC Language Specifications // `http://www.gwu.edu/~upc/documentation.html`, 2003.

[5] Корж А.А. Результаты масштабирования бенчмарка NPB UA на тысячи ядер суперкомпьютера Blue Gene/P с помощью PGAS-расширения OpenMP // Вычисл. методы и программирование. 2010. Т. 11. С. 31–41.

[6] Корж А.А. Распараллеливание задач с нерегулярным доступом к памяти с помощью расширенной библиотеки SHMEM+ на суперкомпьютерах Blue Gene/P и "Ломоносов" // Там же. 2010. Т 11. С. 123–129.

[7] Андрюшин Д.В., Семенов А.С. Исследование реализации алгоритма Survey Propagation для решения задачи выполнимости функций булевых переменных (SAT-задача) на языке UPC // Труды Междунар. суперкомпьютерной конф. "Научный сервис в сети Интернет: Суперкомпьютерные центры и задачи". Абрау-Дюрсо, 2010. С. 133–135.

[8] Johnson A. Unified Parallel C within computational fluid dynamics applications on the Cray X1 // Proc. of the Cray User Group Conf. Albuquerque, 2005.

[9] Beech-Brandt J. Applications of UPC // `http://www.nesc.ac.uk/talks/892/applicationsofupc.pdf`, 2008.

[10] Gordon B., Nguyen N. Overview and Analysis of UPC as a Tool in Cryptanalysis. Tech. Rep. FL 32611. Univ. Florida, 2003.

[11] LBNL, UC Berkeley. Berkeley UPC User's Guide // `http://upc.lbl.gov/docs/user/index.shtml`, 2010.

[12] Su H.H., Billingsley M., George A. Parallel performance Wizard: A performance analysis tool for Partitioned Global-Address-Space programming // 9th IEEE Intern. Workshop on Parallel & Distributed Scientific and Engineering Computing. Miami, 2006.

[13] Андреев Н.Е. Методы автоматизированного анализа производительности параллельных программ // Вестник Новосибирского гос. ун-та. 2009. Т. 7, № 1. С. 16–25.

[14] Андреев Н.Е. Обзор методов автоматизированной оценки эффективности выполнения параллельных программ // ВИНИТИ, 2009.

[15] Miller B.P., Clark M., Hollingsworth J. et al. IPS-2: The second generation of a parallel program measurement system // IEEE Transactions on Parallel and Distributed Systems. Washington, D.C., 1990. P. 206–217.

[16] Vetter J. Performance analysis of distributed applications using automatic classification of communication inefficiencies // Proc. of Conf. on Supercomputing. Dallas, 2000. P. 245–254.

[17] Truong H.L., Fahringer T., Madsen G. et al. On using SCALEA for performance analysis of distributed and parallel programs // Proc. of Conf. on Supercomputing. Denver, 2001.

[18] Helm B.R., Malony A.D., Fickas S. Capturing and automating performance diagnosis: the poirot approach // Intern. Parallel Proc. Symp. Santa Barbara, 1995. P. 606–613.

[19] Андреев Н.Е. Архитектура системы автоматизированного анализа UPC-программ // XVII Всероссийская научно-метод. конф. "Телематика'2010". СПб., 2010. С. 164–166.

[20] Wolf F., Mohr B. Specifying performance properties of parallel applications using compound events // Parallel and Distributed Comp. Practices. 2001. Vol. 4, No. 3. P. 91–110.

[21] Knowledge Specification for Automatic Performance Analysis — APART Technical Report. Tech. Rep. FZJ-ZAM-IB-9918. Research Centre Julich, 1999.

Поступила в редакцию 23 августа 2010 г.,
с доработки — 8 февраля 2011 г.