ДЕКЛАРАТИВНЫЙ ПОДХОД К АВТОМАТИЗАЦИИ ВЫЯВЛЕНИЯ ПРИЧИН ПОТЕРИ ПРОИЗВОДИТЕЛЬНОСТИ MPI-ПРИЛОЖЕНИЙ

А.В. Дергунов

Нижегородский госуниверситет им. Н.И. Лобачевского

Для анализа MPI программ часто используют программные системы, которые осуществляют сбор и визуализацию трассы выполнения программы на кластере. Но при использовании таких инструментов пользователь сталкивается с проблемой анализа больших объемов информации. Другой проблемой при использовании средств визуализации является то, что часто встречающиеся ситуации, приводящие к потерям производительности MPI программ, явно не визуализируются. Поэтому возникает потребность в средствах для автоматизации анализа трассы, которые подсказали бы пользователю, как повысить производительность его программы. В работе описывается программная система, выполняющая эту задачу.

Схема работы системы

Схема работы системы представлена на рис. 1. Исходными данными является трасса, полученная с помощью трассировщика при запуске MPI-приложения на кластере. Она анализируется модулем автоматического анализа трассы. Результат анализа – список выявленных причин недостаточной производительности пользовательского приложения с указанием степени их влияния на общее время работы приложения.

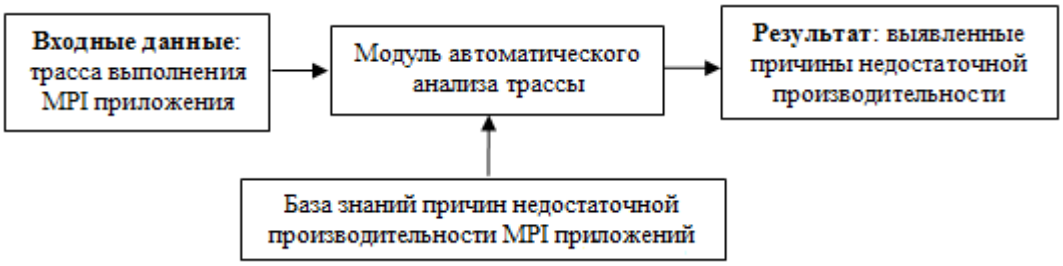

Рис. 1. Схема работы системы

Ключевым компонентом разработанной системы является база знаний причин недостаточной производительности MPI-приложений. База знаний состоит из правил, описанных с помощью декларативного языка анализа событий трассы, разработанного в рамках системы. Этот язык предназначен для:

- описания событий трассы, текущего состояния процесса обработки и результата анализа трассы;
- описания правил обработки потока событий трассы.

Описание событий трассы, текущего состояния процесса обработки и результата анализа трассы

Для представления данных в системе используются структуры, которые характеризуются типом и набором параметров. Каждый параметр имеет имя и тип данных (например, целочисленный, строковый и т.п.). Рабочая память системы хранит набор структур. Изначально память пуста. Она заполняется при считывании событий трассы или в результате выполнения правил.

Синтаксис описания типа структуры следующий:

```
defstruct <категория> <название> "<комментарий>"
  param <имя 1-го параметра> type <тип 1-го параметра>
  …
  param <имя n-го параметра> type <тип n-го параметра>;
```

Используются следующие категории структур:

- *События* (event) – простые события, собранные трассировщиком, или высокоуровневые события, сформированные во время анализа.
- *Состояния* (state) – описывают текущее состояние процесса обработки трассы.
- *Результат анализа* (observation) – выходная информация, полученная в результате обработки трассы.

Описание правил обработки потока событий трассы

Синтаксис описания правил для обработки потока событий трассы следующий:

```
defrule "<комментарий>"
  struct <имя 1-й переменной> type <тип структуры 1-й переменной>
  …
  struct <имя n-й переменной> type <тип структуры n-й переменной>
  where <логическое выражение>
  <действия>;
```

Для правила перечисляется, какие структуры и каких типов должны присутствовать в рабочей памяти. Каждой структуре присваивается имя переменной. Для срабатывания правила необходимо выполнение логического выражения. При выполнении правила запускаются указанные действия. Допустимыми действиями являются:

- *Создание новой структуры* (указанного типа) в рабочей памяти (assert). Это действие применимо для всех категорий структур.
- *Удаление структуры* из рабочей памяти (retract). Это действие применимо только для состояний. События автоматически удаляются из рабочей памяти, когда они не нужны. Управление результатами анализа осуществляется также автоматически системой.
- *Изменение структуры* в рабочей памяти (modify). Это действие также применимо только для состояний. События и результаты анализа неизменяемы.

Алгоритм обработки потока событий трассы

Система основана на последовательной обработке потока событий трассы, отсортированных по времени их возникновения. При этом трассы со всех процессов объединяются в один поток. Работа системы состоит из нескольких итераций, на каждой из которых выполняются следующие действия:

1) считать очередное событие из трассы (в порядке времени их возникновения) и поместить его в рабочую память;
2) поочередно выполнить все правила, условия которых соблюдены;
3) удалить из рабочей памяти события (как созданные на шаге 1), так и в процессе выполнения правил на шаге 2)). Предполагается, что все события были обработаны на шаге 2) и при необходимости дальнейшей обработки требуемая информация сохранена в структурах состояния;
4) если не достигли конца трассы, то перейти к шагу 1).

Пространственная и временная сложность предложенного алгоритма зависит от характера трассы. Если количество структур в рабочей памяти на каждом шаге можно

ограничить константой, то временная сложность алгоритма линейная, а требования по памяти ограничены константой.

Выявление поздней посылки данных с помощью декларативного языка анализа событий трассы

Одной из причин недостаточной производительности является плохая синхронизация приемов и передач данных в MPI программе. В результате процедура приема данных может простаивать, дожидаясь посылки данных (см. рис. 2).

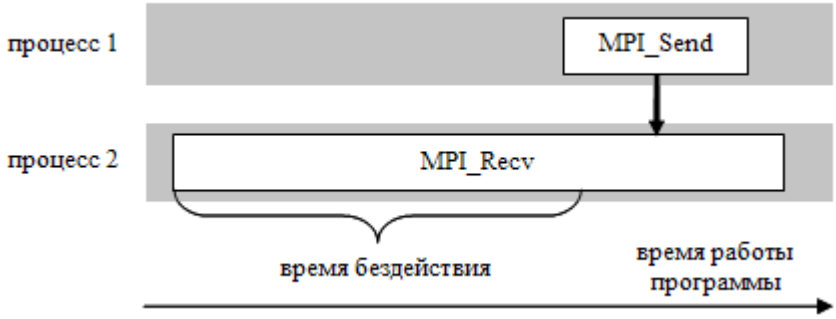

Рис. 2. Поздняя посылка данных

Приведенный внизу листинг на разработанном декларативном языке может быть использован для выявления этой ситуации.

```
 1: defstruct state recv_request "Запрос на прием данных"
 2:   param start_time, finish_time type int
 3:   param is_blocking type boolean
 4:   param source, dest, tag, comm type int;
 5:
 6: defstruct state send_request "Запрос на посылку данных"
 7:   param start_time, finish_time type int
 8:   param source, dest, tag, comm type int;
 9:
10: defstruct event point_to_point_operation "Двухточ. обмен"
11:   param is_recv_blocking type boolean
12:   param send_start_time, send_finish_time type int
13:   param recv_start_time, recv_finish_time type int;
14:
15: defrule "Поместить в рабочую память запрос на посылку"
16:   struct e type function_call_send_operation
17:   assert send_request(start_time = e.@start_time,
18:     finish_time = e.@finish_time, source = e.source,
19:     dest = e.dest, tag = e.tag, comm = e.comm);
20:
21: defrule "Поместить в рабочую память запрос на прием"
22:   struct e type function_call_receive_operation
23:   assert recv_request(start_time = e.@start_time,
24:     finish_time = e.@finish_time, is_blocking = true,
25:     source = e.source, dest = e.dest, tag = e.tag, comm = e.comm);
26:
27: defrule "Сформировать событие двухточечного обмена данными"
28:   struct send type send_request
29:   struct recv type recv_request
30:   where
31:     send.comm == recv.comm and send.dest == recv.dest and
32:     (send.source == recv.source or recv.source == MPI_ANY_SOURCE) and
33:     (send.tag == recv.tag or recv.tag == MPI_ANY_TAG)
34:   assert point_to_point_operation(is_recv_blocking = recv.is_blocking,
```

```
35:     send_start_time = send.start_time,
36:     send_finish_time = send.finish_time,
37:     recv_start_time = recv.start_time,
38:     recv_finish_time = recv.finish_time)
39:   retract send, recv;
40:
41: defstruct observation performance_observation "Причина недост. произв."
42:   param name, description, advice type string
43:   param impact_time type int;
44:
45: defrule "Выявление поздней посылки данных"
46:   struct op type point_to_point_operation
47:   where
48:     op.is_receive_blocking and op.send_start_time > op.recv_start_time
49:   assert performance_observation(name = "Поздняя посылка данных",
50:     description = "Операция посылки данных вызывается
51:       позднее операции приема. Из-за этого блокирующая
52:       операция приема вынуждена простаивать.",
53:     advice = "Улучшить синхронизацию приемов и передач
54:       данных, отправляя данные по возможности раньше, или
55:       использовать неблокирующие операции приема.",
56:     impact_time = op.send_start_time - op.recv_start_time);
```

Входными данными являются события вызова блокирующих функций MPI приема и передачи сообщений, перехватываемые трассировщиком (соответствуют событиям типа function_call_receive_operation и function_call_send_operation). При чтении трассы эти события поочередно в порядке их возникновения поступают в рабочую память. В результате вызываются правила, определенные на строках 15-19 и 21-25, и в рабочую память попадают состояния запросов на прием/передачу данных. Правило на строках 27-39 формирует событие двухточечного обмена данными (с помощью действия assert на строках 34-38) при появлении соответствующих запросов на прием/передачу (условие соответствия указано в строках 31-33).

С помощью правила на строках 45-56 осуществляется обнаружение событий двухточечного обмена данными, функция приема данных в которых блокирующего типа (например, MPI_Recv) и время начала посылки данных позднее времени начала приема данных (строка 48). При выполнении перечисленных условий система делает вывод о неэффективной работе программы, связанной с поздней посылкой данных. В правиле также указывается формула для вычисления степени влияния на общее время работы программы (строка 56).

В работе [1] описан эксперимент по использованию разработанной системы для повышения производительности MPI программы, реализующей сеточные вычисления. В результате анализа трассы ее работы на кластере системой была выявлена поздняя посылка данных. После внесения небольших изменений программы общее время ее работы уменьшилось с 42,11 до 18,18 секунд (параметры программы: матрица 6400x6400, 1000 итераций, 64 процесса; параметры кластера: узлы с двумя двухъядерными процессорами Intel Xeon 2.66 GHz, 4 Gb оперативной памяти, сеть Gigabit Ethernet).

Заключение

Рассмотрена программная система, которая значительно облегчает задачу анализа производительности MPI-приложений. Предложен декларативный язык анализа событий трассы, использование которого позволяет автоматизировать анализ производительности MPI-программы и выявлять часто встречающиеся причины недостаточной производительности. Система особенно актуальна для больших суперкомпьютеров, ис-

пользующих большое количество процессоров, т.к. в этом случае задача анализа трассы особенно трудна.

Разработанная система основана на ином подходе к анализу производительности по сравнению системами для визуализации трассы работы MPI программы (например, Jumpshot [2]). Наиболее близким аналогом является система KOJAK [3]. Главным отличием рассмотренной системы является полностью декларативный подход к анализу производительности. В результате эта система удобнее в поддержке и развитии, но кроме того ее можно гораздо проще адаптировать для выполнения других задач, например, выявления ошибок в MPI программах, а также анализа приложений, использующих другие библиотеки параллельного программирования.

База знаний системы состоит из правил, осуществляющих выявление следующих причин недостаточной производительности MPI программ: поздняя посылка данных, поздний прием данных, поздняя посылка данных при операции «от одного ко многим» и т.д. Более подробно состав базы знаний описан в работе [1]. Поддерживаются все виды двухточечных, коллективных и односторонних обменов. На данный момент система состоит из 44 правил, которые в сумме составляют около 700 строк кода.

Литература

1. Дергунов А.В. Автоматизация выявления причин потери производительности MPI программ на экзафлопсных и других больших суперкомпьютерах // Научный сервис в сети Интернет: экзафлопсное будущее. Труды международной суперкомпьютерной конференции. – М.: Изд-во МГУ, 2011. С. 491-496.
2. Omer Zaki, Ewing Lusk, William Gropp, Deborah Swider. Toward Scalable Performance Visualization with Jumpshot // High-Performance Computing Applications, Vol. 13, number 2, P. 277-288, 1999.
3. Wolf F., Mohr B. Automatic performance analysis of hybrid MPI/OpenMP applications // Journal of Systems Architecture: the EUROMICRO Journal, Volume 49, Issue 10-11, 2003. P. 421-439.