УДК 004.051

ОБЗОР МЕТОДОВ АВТОМАТИЗИРОВАННОЙ ОЦЕНКИ ЭФФЕКТИВНОСТИ ВЫПОЛНЕНИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Н. Е. Андреев

Согласно отчету Межведомственной комиссии по развитию сверхмощных вычислений США, эффективность современных параллельных систем находится ниже отметки в 10 %. Для того, чтобы добиться приемлемой эффективности, разработчики пользуются инструментами поиска и устранения проблем производительности. Большая часть таких инструментов полагаются на пользователя при анализе и интерпретации данных, но есть ряд систем позволяющих в той или иной мере автоматизировать этот процесс и снизить нагрузку на пользователя. В статье приведен краткий обзор таких инструментов и методов.

According to the report of the USA High-end computing revitalization task force (HECRTF) sustained performance of contemporary parallel machines is less than 10 %. To gain acceptable parallel performance programmers often use performance analysis instruments. Most of them entirely rely on developer in finding parallel performance problems but some automate this process more or less. This paper is dedicated to such kind of instruments.

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

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]. Рынок сверхмощных вычислений просто не настолько велик, чтобы отвлечь внимание компьютерной индустрии от более крупных и более прибыльных секторов электронной коммерции и бизнес-расчетов. Продажи в сфере высокопроизводительных вычислений составляют примерно 1 миллиард долларов, тогда как на рынке серверов – более 50 миллиардов. Так как индустрия сосредоточена на доходном серверном рынке, 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. Рис. 1 иллюстрирует постоянно растущее несоответствие между теоретической пиковой производительностью суперкомпьютеров и производительностью на реальных приложениях. На рис. 1 представлены результаты теста, разработанного в Национальном исследовательском центре научных расчетов в области энергетики (NERSC), который расположен в Национальной лаборатории Лоуренса в Беркли, специально для оценки производительности приложений, использующихся в центре. Постоянные технологические улучшения микропроцессоров, подталкиваемые законом Мура, приводят к резкому росту верхней кривой 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 кривой производительности на реальных приложениях.

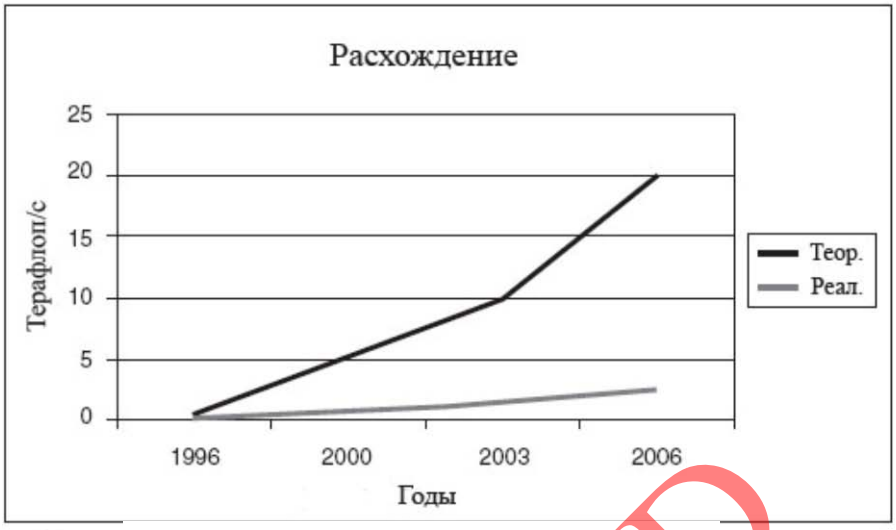
В условиях резкой нехватки производительности программистам необходимы инструменты анализа эффективности и оптимизации программ. Разработка и реализация таких инструментов для параллельных компьютеров очень сложна ввиду как архитектурной, так и эксплуатационной сложности таких систем. Общие требования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

– скрывать модель системы и брать на себя низкоуровневые операции по управлению системой, чтобы упростить процесс оптимизации программы;

– быть чувствительными к ограничениям, накладываемым задачей, на объем получаемой информации о системе;

– извлекать важные для анализа метрики системы;

– учитывать возможные возмущения системы, возникающие при оснащении (добавлении меток, снимающих метрики производительности) программы или искажении результатов моделирования, возникающие ввиду той или иной степени абстракции от физической системы.



Теор.: теоретическая пиковая производительность
Реал.: производительность на реальных приложениях

Рис. 1. ***Расхождение теоретической и реальной производительности***

Полезность инструментального средства определяется точностью моделирования системы, на которой оно базируется. Ввиду сложности параллельных платформ, это знание достаточно сложно извлечь. Возможно, самый важный аспект инструмента – это способность к разрешению проблем производительности. Он может оказать неоценимую помощь в нахождении и устранении проблем параллельных программ, если он поддерживает возможность мониторинга состояний программы и системы, но такие инструменты могут сами влиять на систему в процессе измерений, необходимых для последующего анализа. В худшем случае возмущение поведения системы может достигать такого уровня, когда наблюдения будут ненадежны, а модели, в которых эти данные используются, приведут к неправильным выводам.

Инструменты анализа производительности

Анализ производительности – ключевой аспект, на который следует обращать внимание при разработке инструментов данного класса. То, как инструмент способен анализировать приложение, играет важную роль при его признании. В идеале такая программа должна не только показывать данные в интуитивно понятном виде, но и помогать пользователю детализировать интересующую его информацию так, чтобы возникшие узкие места и проблемы производительности могли быть легко выявлены.

Adam Leko в статье «Performance Analysis Strategies» [2] приводит эталонный процесс отладки производительности (рис. 2). Сначала пользователь добавляет в свою программу измерительный код (желательно без особых усилий), запускает его и далее подает полученные в процессе выполнения данные в программу анализа. Программа должна помочь пользователю найти и исправить проблемы производительности исходной программы. Этот процесс повторяется до тех пор, пока пользователь не добьется приемлемого уровня производительности. Измерительный код должен вносить низкий процент накладных расходов на выполнение пользовательской программы и собирать достаточный объем данных, чтобы выявить любую проблему производительности. Процесс анализа в идеале должен быть автоматизирован на столько, на сколько это возможно и выполнен полно и точно после того, как все необходимые данные были собраны в процессе выполнения программы. В реальности не существует инструментов, которые бы следовали этому процессу в таком виде. Зачастую они поддерживают его частично, балансируя между функциональностью, накладными расходами на выполнение и простотой использования. Современные системы настолько сложны, что собрать информацию, достаточную для выявления любой проблемы производительности, невозможно без серьезного влияния на скорость выполнения программы, что сводит на нет полезность такой информации. Даже когда удается собрать достаточно полную информацию без изменения характеристик выполнения программы, большой объем и различный тип информации создают проблемы объединения ее в единое целое, что затрудняет анализ данных.

Таким образом, реальные инструменты анализа производительности должны решить, какие данные они будут собирать и должны быть способны проводить анализ и фильтрацию даже недостаточно полной и точной информации. Фактически это означает, что фильтрация и анализ могут (а иногда должны) проводиться на разных этапах отладки производительности, вместо фиксированного, как показано на рис. 2.

Существующие методики можно разделить следующим образом:

1. Предварительный анализ, интерактивные методы, постобработка. Методы предварительного анализа выполняют оценку до непосредственного запуска программы, интерактивные методы выполняют анализ в процессе работы прикладной программы. Постобработка – это метод, который требует наличия полной информации, получаемой в процессе выполнения программы, перед тем, как анализ поведения этой программы может быть запущен.

2. Ручные и полуавтоматические методы. Некоторые стратегии анализа и поиска узких мест могут быть в той или иной степени автоматизированы, другие полностью полагаются на пользователя при анализе и интерпретации данных.

Необходимо заметить, что данные категории не взаимоисключающие. Некоторые стратегии могут основываться на комбинации вышеописанных методов. Таким образом, инструменты можно условно разделить на шесть категорий: постобработка, интерактивный метод и предварительный анализ в ручном и полуавтоматическом вариантах.

Еще одной важной особенностью, которую необходимо принимать во внимание при обсуждении методов анализа производительности, является масштабируемость. Методы, которые хорошо работают на небольших системах, но разваливаются на системах из нескольких десятков процессоров, не очень-то полезны, так как многие проблемы производительности не проявляются, пока размер системы не достигнет определенной отметки.

Также необходимо заметить, что в методе ручного анализа трудоемкий процесс поиска узких мест производительности выполняется пользователем. Инструменты, использующие данный метод, при поиске проблем полагаются полностью на предоставление пользователю соответствующих диаграмм, графиков и таблиц и методы манипулирования ими (масштабирование и поиск). Предоставление информации о производительности программы в графическом виде может быть мощным приемом, но без сложных методов фильтрации объем отображаемых данных может быть чрезвычайно большим.

Рис. 2. ***Цикл отладки производительности***

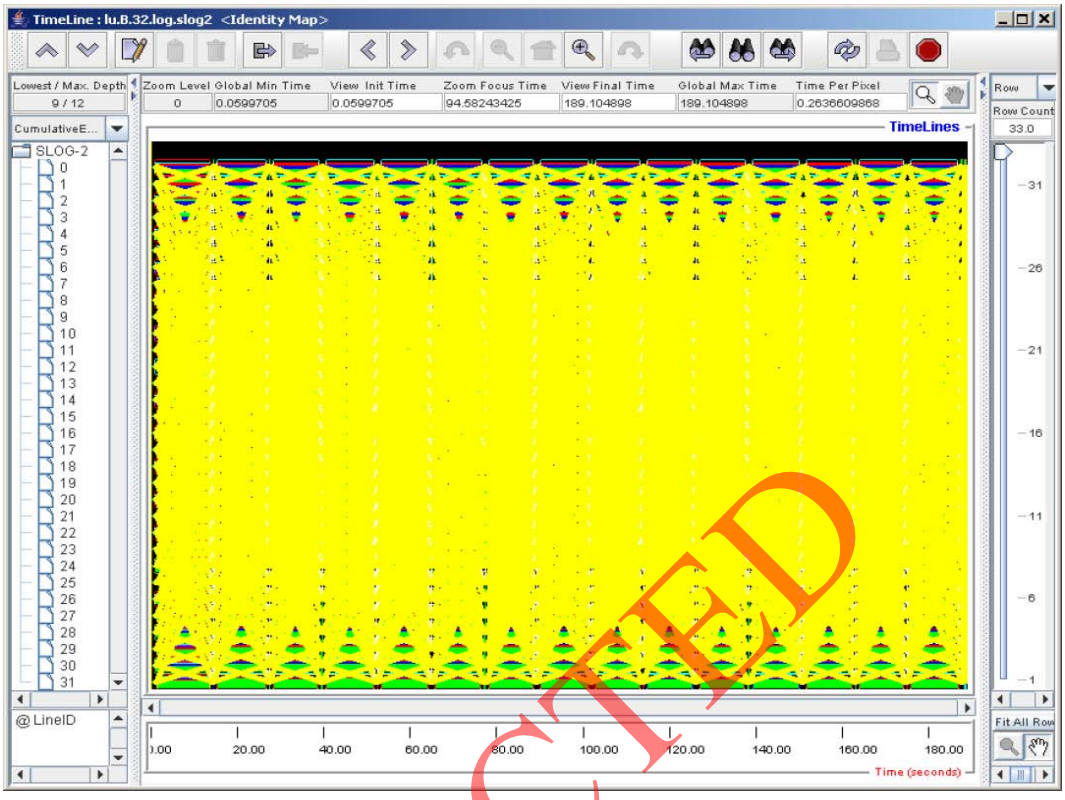


Рис. 3. ***Визуализации теста NAS LU в Jumpshot-4***

Чтобы привести пример того, как объем данных в трассах может стать чрезвычайно большим, посмотрим как выглядит визуализация бенчмарка NAS LU (нагрузка класса B) в Jumpshot-4 (рис. 3). В данном примере отсутствует какая-либо фильтрация данных. Размер трассировочного файла для трех минут выполнения составляет 325 МБ. Как видно из рисунка, очень сложно разглядеть, что же происходит в программе из-за большого количества коммуникаций, даже несмотря на то, что 32-процессорная система считается небольшой по сегодняшним меркам. Чтобы справиться с этой проблемой, инструменты, работающие с файлами трасс и предоставляющие лишь методы ручного анализа, зачастую позволяют осуществлять поиск, фильтрацию и масштабирование, чтобы ограничить объем отображаемых данных. Например, многие инструменты визуализации трасс MPI позволяют пользователю ограничить отображаемые передачи сообщений при помощи выбора тэга или источника и назначения. Тем не менее большинство инструментов, использующих ручной анализ при помощи трасс, не позволяют отобразить полученные данные обратно на исходный код, так как сложно увязать эту информацию с временной шкалой. За выбор фильтров в данном методе также ответственен пользователь.

Методы полуавтоматического анализа

Очевидно, что машины гораздо более приспособлены к обработке больших массивов информации. С учетом этого, некоторые разработчики создали прототипы систем, которые пытаются автоматизировать процесс поиска проблем производительности на столько, на сколько это возможно. Есть несколько реализованных прототипов, которые способны выполнять нечто похожее на автоматический анализ собранных в процессе выполнения программы данных. Большинство стратегий используют приемы и методы из области искусственного интеллекта, такие как экспертные системы на основе баз знаний и методы автоматической классификации. Далее в статье рассматриваются несколько инструментов, в основе которых лежат методы полуавтоматической обработки.

IPS-2

IPS [3] – это одна из ранних программ, первые публикации о которой датируются 1987 годом. Разрабатывалась Висконсин-Мэдисонским Университетом совместно с Bell Laboratories для систем, работающих на микропроцессорах VAX, DECStation и Sun 4 под управлением ОС 4.3BSD UNIX. В IPS используется несколько методов автоматизированного анализа программ: анализ критического пути и анализ фазового поведения.

Анализ критического пути (АКП) позволяет найти самый длинный по времени путь, через который параллельная программа проходит в процессе своего выполнения. Чтобы провести АКП, на основе трассировочной информации строится граф операций программы. Вершины графа – события, например, межпроцессные коммуникации или создание процессов, а дуги – время, затраченное на выполнение события. По ходу взаимодействия процессов друг с другом возникают различные пути, оптимизация самого длительного из которых, приведет к

сокращению общего времени выполнения программы. После того как граф операций построен, выполняется поиск самого длинного пути. Путь может состоять из тысяч вершин, поэтому на каждом из уровней иерархии пользователю информация представляется в агрегированном виде. Например, на уровне процессов, какой процент времени критического пути программа выполняла расчеты и какой межпроцессные коммуникации, для каждой пары процессов. IPS не позволяет рассчитывать пути, близкие к критическому. Если существуют пути близкие по времени и практически не пересекающиеся с критическим, то оптимизация не даст результатов. Для того чтобы определить, какое влияние на программу окажет оптимизация критического пути, IPS позволяет приравнять его к нулю и пересчитать оценку.

Анализ фазового поведения (АФП) [3] – это метод, способный выделить различные фазы выполнения программы. Например, в параллельной программе, работающей по схеме главный-подчиненный можно выделить следующие фазы:

1) главный процесс ставит задачу,
2) инициализируются подчиненные процессы,
3) главный распределяет части задачи каждому подчиненному,
4) подчиненный рассчитывает свою часть задачи,
5) главный объединяет частичные результаты.

Шаги 3-5 повторяются пока не получено решение. Каждая фаза имеет свои характеристики. Задача АФП автоматически выделить эти фазы, что позволит в дальнейшем выполнять оптимизацию, сосредоточившись на определенной фазе. АФП состоит из трех этапов: сглаживания, сегментации и комбинирования. На вход метода подаются кривые метрик производительности, которые представляют собой значение метрики в различные моменты времени выполнения программы. Задача сглаживания упростить последующий анализ за счет устранения пиков. Сглаживание выполняется по алгоритму скользящего окна. После сглаживания выполняется сегментация. Задача сегментации – построить кривую границ, которая бы для каждой метрики производительности m показывала вероятность возникновения перехода фаз для каждого момента времени t_i. Для этого сначала строится ступенчатая функция $h_{m,i}$, которая для момента времени t_i равна разнице между значениями предыдущего минимума (максимума) и следующего максимума (минимума) кривой метрики m. Далее строится кривая границ, которая имеет следующий вид: $B_{m,i} = abs(\frac{\Delta V_{m,i}}{\Delta t_i}) \times h_{m,i}$,

где – $\Delta V_{m,i}$ разница между значениями метрики m в моменты времени t_t и t_{i-1}.

Чем больше значение $B_{m,i}$, тем выше вероятность, что в данной точке кривой находится переход фаз. После того как кривые границ для каждой метрики посчитаны, они объединяются в итоговой функции, учитывающей все метрики производительности: $B_i = \sum_{m \in M} B_{m,i} = \sum_{m \in M} abs(\frac{\Delta V_{m,i}}{\Delta t_i}) \times h_{m,i}$. Считается, что в момент времени t_i происходит переход фаз, если первая производная от B_i равна нулю и B_i выше определенного порогового значения. Пороговое значение выставляется программистом вручную. Чем пороговое значение выше, тем меньше фаз и наоборот. После того, как фазы получены, программист получает возможность более гибко и оптимально выполнять оптимизацию программы. Например, он может применить АКП к определенной фазе.

Подход Vetter'а

Jeffrey Vetter из Национальной Ливерморской лаборатории им. Лоурэнса разработал метод автоматической классификации неэффективных коммуникаций для программ, написанных на MPI [4]. При разработке подхода Vetter преследовал три цели: сокращение объема данных, портативность и точность. Основа метода – подход, позаимствованный из области машинного обучения. Как и в IPS-2, используется постобработка трассировочной информации. Новизна подхода в использовании классификации на базе дерева решений. Vetter в своей классификации уделяет внимание четырем операциям: блокирующим и не блокирующим операциям отправки и приема сообщений. Исходя из возможных вариантов неэффективного использования данных операций, он выделяет 7 классов неэффективных коммуникаций: нормальный – операция выполнена нормально, поздняя отправка – операция MPI_Send запущена значительно позже MPI_Recv, поздний прием – операция MPI_Recv запущена значительно позже MPI_Send, поздний старт отправки – то же что и поздняя отправка, но для MPI_Isend, позднее завершение отправки – операция MPI_Wait для соответствующей операции MPI_Isend запущена поздно, программа может получить доступ к буферу отправки раньше, поздний старт приема – то же что и поздний прием, но для MPI_Irecv, позднее завершение приема – операция MPI_Wait для соответствующей операции MPI_Irecv запущена поздно, программа может получить доступ к буферу приема раньше. На первом этапе работы программы пользователь должен вручную обучить дерево решений на заранее подготовленных тестах эффективного и неэффективного поведения MPI-программ. Это делается потому, что основной параметр, которым оперирует дерево решений при классификации, – это время выполнения операции, а на разных программно-аппаратных конфигурациях времена могут сильно разниться. После того, как дерево обучено, пользователь может запустить задачу и подать результаты трассировки модулю классификации. В качестве результата пользователь получает таблицу, каждая строка которой – это пара отправитель-получатель и количество попаданий в каждую из 7 категорий [4]. Отправитель и получатель в таблице указаны функциями программы, из которых выполнялись операции, и смещением относительно начала

функции, чтобы программист мог точно определить проблемную пару операций. При проверке дерева решений на тех же данных, на которых оно было обучено, автор получил процент ошибки от 3 на тесте поздняя отправка, до 42 на тесте позднее завершение приема. Средний процент ошибки по совокупности тестов – 13 %. Автор подтверждает, что его подход менее точен, чем, например, нейронные сети, но более понятен для разработчика, который может посмотреть, почему пара отправка-прием была классифицирована по той или иной ветви. В прототипе поддерживается 7 категорий неэффективного поведения и 4 MPI операции, но дерево решений можно достаточно легко расширить. Для этого необходимо разработать тест для соответствующей проблемной ситуации.

SCALEA

Иной подход используется в пакете SCALEA [5, 6] – инструменте анализа производительности для программ, написанных на OpenMP, MPI, HPF и смешанных параллельных/распределенных программ. На первом этапе препроцессор SCALEA строит абстрактное синтаксическое дерево (АСД) программы, что позволяет пользователю выбрать интересующие его блоки кода, будь то обычные циклы, процедуры, циклы HPF INDEPENDENT, циклы OpenMP PARALLEL, секции OpenMP, операции MPI Send/Recv или что-либо другое. Если необходимо выбрать произвольную область кода, можно воспользоваться специальными директивами, которые вручную помещаются в исходный код программы. Для каждого анализируемого блока кода формируется файл описания инструментовки, который содержит идентификатор блока, информацию о его местоположении в исходном коде и ссылку на собранную информацию о производительности. Это позволяет уменьшить объем трассировочной информации за счет того, что служебная информация в каждой строчке трассы сокращается до единственного идентификатора, а также четко отображать данные анализа на исходный код программы. После того как инструментовка закончена, программа запускается, и в процессе ее выполнения собирается трассировочная информация, а также показатели аппаратных счетчиков. Обработка собранной информации выполняется в два этапа. В начале данные проходят первичную фильтрацию, и строится динамический граф вызовов блоков кода (ДГБ). ДГБ для программы Q задается направленным графом потоков $G = (R, E, s)$, где R – множество узлов, а E – множество дуг. Узел $r \in R$ представляет собой блок кода, который хотя бы единожды выполняется в процессе работы программы Q. Дуга $(r_1, r_2) \in E$ указывает, что при выполнении программы Q блок кода r_2 вызывается внутри r_1, r_2 в таком случае - динамический подблок r_1. Блок кода, с которого начинается выполнение Q, задается s. ДГБ позволяет структурировать данные о производительности программы и точно рассчитать издержки на те или иные операции по каждому блоку на основе информации из подблоков там, где это необходимо. После того, как ДГБ построен и трассы распределены по соответствующим блокам, начинается непосредственно анализ производительности. SCALEA выполняет анализ в форме поиска издержек выполнения программы. С точки зрения программиста, коммуникации MPI Send/Recv можно считать издержками на перемещения данных, которых в последовательной программе нет, а циклы процессора, которые тратятся на создание и уничтожение нитей в OpenMP при входе в параллельную секцию, – издержками на управление параллелизмом. Согласно закону Амдала [7], теоретически лучший последовательный алгоритм требует времени T_s для того, чтобы выполнить программу, T_p – время, необходимое для выполнения параллельной версии на p процессорах. Тогда временные издержки могут быть заданы как $T_o = T_p - T_s / p$ и отражают разницу между достигнутой и оптимальной производительностью T_o, в свою очередь, можно разделить на T_i и T_u таким образом, что $T_o = T_i + T_u$, где T_i – это издержки, которые можно установить, а T_u – это часть издержек, которые не удается детально проанализировать. В SCALEA выделяется 6 групп издержек:

– *Перемещения данных* – любые передачи данных внутри адресного пространства процесса (доступ к локальной памяти) или между процессами (доступ к удаленной памяти).

– *Синхронизация* (например барьеры и блокировки) – это операции, координирующие работу процессов и нитей при доступе к данным для поддержания целостности данных и вычислений.

– *Управление параллелизмом* (например, fork/join – операции и диспетчеризация циклов) используется для регулирования и управления параллелизмом программы и может осуществляться пользователем, компилятором или библиотекой времени выполнения.

– *Дополнительные вычисления* отражают любые изменения исходной последовательной программы, включая изменения алгоритма и трансформации компилятора, с целью увеличения параллелизма (например устранение зависимостей по данным) или локальности данных (например при помощи изменения шаблонов доступа к данным).

– *Потеря параллелизма* ввиду неполного распараллеливания программы, которую можно дальше классифицировать как нераспараллеливаемый код (выполняемый только одним процессором), дублированный код (выполняемый всеми процессорами) и частично распараллеливаемый код (выполняемый более чем одним процессором, но не всеми).

– *Неидентифицированный параллелизм* относится к издержкам неохватываемым вышеописанными категориями.

Для того чтобы подсчитать издержки, SCALEA строит ДГБ для последовательной и параллельной

версий программы и сравнивает соответствующие регионы кода двух версий. Обозначим ДГБ параллельной и последовательной версий – $DRG_s(V_s, E_s, s_s)$ и $DRG_p(V_p, E_p, s_p)$ соответственно. Для региона кода R_i^p в DRG_p найдем соответствующий R_i^p – регион в DRG_s. Предположим, что это регион R_i^s. Положим, что $T_s(R_i^s)$, $T_p(R_i^p)$ и $T_o(R_i)$ – это время выполнения последовательной версии, время выполнения параллельной версии и полученные издержки. Тогда накладные расходы рассчитываются следующим образом:
$T_o(R_i) = T_p(R_i^p) - T_s(R_i^s) / p$.

EXPERT

EXPERT [8, 9] – это один из популярных инструментов, который также использует подход баз знаний для поиска узких мест параллельных программ. EXPERT в своем подходе позволяет отделить сам процесс анализа от описания шаблонов неэффективного поведения, имеет модульную архитектуру и расширяем. EXPERT – это модуль анализа, который входит в состав набора инструментов KOJAK, который автоматически выполняет оценку производительности программ, написанных на C/C++ или Fortran с использованием MPI, OpenMP или смешанного подхода. В качестве источника данных используются файлы трасс. Инструментовка программы может быть выполнена автоматически при помощи профилировочного интерфейса компилятора, с помощью TAU [10] или DPCL [11]. Если необходимо, пользователь может оснастить произвольные блоки кода при помощи POMP-директив, которые потом обрабатываются OPARI [12]. OPARI используется также для автоматического оснащения OpenMP-программ. Инструментовка MPI-директив выполняется автоматически библиотекой-обверткой. После того как данные о работе программы собраны, они передаются в модуль анализа EXPERT.

В процессе анализа EXPERT ищет в трассировочных файлах шаблоны неэффективного поведения. То, как EXPERT представляет эти шаблоны внутри себя, позволяет ему фиксировать очень сложные ситуации, не охватываемые профилировочными инструментами и визуализаторами трасс. EXPERT описывает шаблоны в виде составных событий. Составное событие – это набор найденных в трассировочном файле событий, которые удовлетворяют условиям возникновения ситуации, описываемой шаблоном. Так как составные события обычно включают в себя сложные межсобытийные связи, необходимы высокоуровневые структуры данных, которые способны отслеживать и предоставлять в нужный момент такую информацию. Поэтому EXPERT построен на базе языка EARL [13] – языка доступа к трассировочным файлам. Основная задача EARL – упростить описание шаблонов неэффективного поведения, что позволяет легко расширять и корректировать набор шаблонов, используемых в процессе анализа. В отличие от необработанного файла трасс, который позволяет считывать записи в последовательном виде, EARL обеспечивает произвольный доступ к различным событиям и два типа абстракций: состояния и указатели. Состояния отражают различные аспекты общего состояния выполнения программы. Каждое состояние – это множество событий. Появление события приводит к добавлению, либо к удалению элемента из множества. Например, для каждой пары процессов EARL поддерживает очередь сообщений. Если возникает событие отправки сообщения, оно добавляется в очередь, если возникает событие приема, то соответствующее ему событие отправки удаляется из очереди. EARL также поддерживает состояния для коллективных коммуникаций MPI, операций OpenMP parallel, синхронизаций на основе блокировок, стеков блоков кода и дерева вызовов. Указатели – это атрибуты событий, ссылающиеся на другое соответствующее событие. Например, событие приема содержит атрибут sendptr, который указывает на соответствующее ему событие отправки. Наличие указателей, дает возможность воспользоваться состояниями. Другие указатели связывают соответствующие события входа и выхода, операции над одной и той же переменной синхронизации и информацию о порядке вызовов. Состояния и указатели формально описаны в [14], а исчерпывающая документация по EARL API может быть найдена в [13].

В процессе анализа поведения приложения EXPERT последовательно перебирает файлы трасс и пытается найти шаблоны, описанные ранее в виде составных событий. Для того чтобы проиллюстрировать, как описываются и обнаруживаются составные события, обратимся к рис. 4, на котором в виде временной шкалы изображено составное событие *поздняя отправка*. Процесс A ожидает сообщения от процесса B, которое отправлено гораздо позже начала операции приема. Таким образом, большая часть времени, потраченная операцией приема, фактически время простоя, которое может быть использовано с пользой. EXPERT распознает этот шаблон, ожидая операции приема в потоке событий. Как только такое событие поймано, EXPERT при помощи указателей (пунктирные линии на рисунке), подсчитанных EARL, находит моменты входа обоих коммуникационных операций и определяет смещение (idle time на рисунке 4). Из рисунка также видно, что шаблона поздняя отправка можно избежать, поменяв порядок приема сообщений. Так как сообщение от процесса C отправлено раньше B, оно в любом случае достигнет процесса A раньше. Поэтому, вместо того, чтобы ждать сообщения от процесса B, A может провести это время с пользой. В данном контексте шаблон поздняя отправка называется *поздняя отправка / неправильный порядок*. EXPERT распознает эту ситуацию, просмотрев состояние выполнения, подсчитанное EARL, на момент приема процессом A сообщения от процесса B. Проверка очереди сообщений, отправленных A, показывает, что есть более старые сообщения, чем только что принятое.

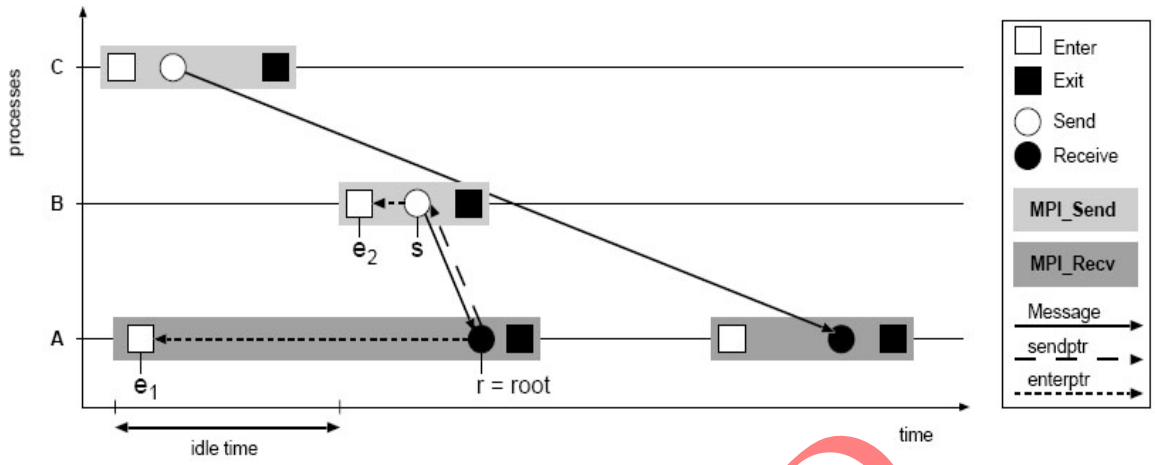


Рис. 4. ***Составное событие поздняя отправка / неправильный порядок***

EXPERT позволяет: вычислить стоимость коммуникаций, синхронизации и стоимость операций ввода\вывода, выявить преобладающую и самую частую операции коммуникации, большие сообщения, наличие неравномерного распределения и разбалансировку на барьерной операции. Также в поле зрения метода попадают такие сложные события, как поздняя отправка, поздний прием, неправильный порядок сообщений и несколько событий из парадигмы главный – подчиненные.

Заключение

В данной статье представлен ряд методов, реализованных в инструментах автоматизированного анализа и отладки производительности параллельных приложений. Подавляющее большинство современных инструментов идут по пути наименьшего сопротивления, давая возможность пользователю вручную фильтровать и просматривать полученные в процессе сбора данные. Инструменты, рассмотренные в статье, выпадают из этого списка. Они реализуют в себе сложные подходы, которые позволяют значительно снизить нагрузку на пользователя в процессе анализа приложения и повысить результативность данного процесса. Полагаясь только на временные шкалы и профилировку, программист рискует пропустить действительные причины, приведшие к низкой эффективности его приложения. Большинство утилит анализа базируются на постобработке трасс. Причиной тому – легкость реализации и удобство для пользователя. Также большинство разработчиков признают крайнюю важность вовлечения пользователя в процесс анализа. Хотя некоторые пытаются (со временем) полностью избавить его от утомительного анализа и оптимизации, проблема оптимизации изучена не настолько глубоко. Поэтому любой метод, выходящий за рамки простого анализа масштабируемости, предполагает вмешательство пользователя, и раз так, инструмент, реализующий данный подход, должен обеспечить пользователя интерфейсом должного уровня.

Литература

1. Federal Plan for High-End Computing: Report of the High-End Computing Revitalization Task Force (HECRTF) [Электронный ресурс], 2004. URL: http://www.nitrd.gov/pubs/2004_hecrtf/20040702_hecrtf.pdf (дата обращения: 2.03.09).

2. Leko, A. Performance Analysis Strategies [Электронный ресурс]. URL: http://www.hcs.ufl.edu/prj/upcgroup/upcperf/documents/20050302AnalysisDraft.pdf (дата обращения: 2.03.09).

3. Miller, B. PIPS-2: The Second Generation of a Parallel Program Measurement System. IEEE Transactions on Parallel and Distributed Systems, IEEE Computer Society / B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. S. Lim, T. Torzewski. – 1990. – С. 206 – 217.

4. Vetter, J. Performance Analysis of Distributed Applications Using Automatic Classification of Communication Inefficiencies / J. Vetter // Труды XIV International Conference on Supercomputing, ACM Press. – 2000. – С. 245 – 254.

5. Truong, H.-L.. On using SCALEA for Performance Analysis of Distributed and Parallel Programs / H.-L. Truong, T. Fahringer, G. Madsen, A. D. Malony, H. Moritsch, S. Shende // Труды конференции SC2001, ACM Press, 2001. – С. 37.

6. Truong, H.-L. Scalea - a Performance Analysis System for Distributed and Parallel Programs / H.-L. Truong, T. Fahringer // Труды VIII International Euro-Par Conference, Springer, 2002. – С. 75 – 85.

7. Amdahl, D. Validity of the Single-processor Approach to Achieving Large-scale Computer Capabilities / D. Amdahl // Труды конференции AFIPS, AFIPS Press, 1967. – С. 483 – 485.

8. Wolf, F. Automatic Search for Patterns of Inefficient Behavior in Parallel Applications / F. Wolf, B. Mohr, J. Dongarra, S. Moore. [Электронный ресурс]. URL: http://www.netlib.org/netlib/utk/people/JackDongarra/PAPERS/auto-apart-2005.pdf (дата обращения: 2.03.09).

9. Wolf, F. Automatic Performance Analysis of MPI Applications Based on Event Traces / F. Wolf, B. Mohr.

// Труды VI International Euro-Par Conference, Springer. – 2000. – С. 123. – 132.

10. Shende, S. S.. The Role of Instrumentation and Mapping in Performance Measurement: докторская диссертация, University of Oregon, 2001.

11. DeRose, L. The Dynamic Probe Class Library – An Infrastructure for Developing Instrumentation for Performance Tools. / L. DeRose, T. Hoover, J. Hollingsworth // Труды XV International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2001.

12. Mohr, B. Design and Prototype of a Performance Tool Interface for OpenMP / B. Mohr, A. Malony, S. Shende, F. Wolf // Journal of Supercomputing, Kluwer Academic Publishers, 2002. – С. 105 – 128.

13. F. Wolf. EARL - API Documentation. [Электронный ресурс]. URL: http://www.fzjuelich.de/jsc/datapool/Kojak/earl-2.0.pdf (дата обращения: 2.03.09).

14. F. Wolf. Automatic Performance Analysis on Parallel Computers with SMP Nodes: докторская диссертация, RWTH Aachen University, 2003.

Рецензент – В. П. Потапов, директор института угля и углехимии СО РАН.