Распараллеливание реализаций сугубо последовательных алгоритмов

[1] *А.Б. Бугеря <shurabug@yandex.ru>*
[2] *Е.С. Ким <eugene.kim@ispras.ru>*
[2] *М.А. Соловьев <icee@ispras.ru>*
[1] *Институт прикладной математики им. М.В. Келдыша РАН, 125047, Россия, г. Москва, Миусская пл., д. 4*
[2] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*

Аннотация. Работа посвящена теме распараллеливания программ в особо сложных случаях – когда используемый алгоритм является сугубо последовательным, параллельных альтернатив используемому алгоритму нет, а время его выполнения неприемлемо велико. Рассматриваются различные методы распараллеливания программных реализаций таких алгоритмов и балансировки получающейся вычислительной нагрузки, позволяющие получить значительное ускорение выполнения прикладных программ, в которых используются сугубо последовательные алгоритмы. Приведенные методы иллюстрируются практикой их применения к двум алгоритмам, используемым в среде динамического анализа программ. Основная цель данной работы – показать, что использование в программной реализации сугубо последовательного алгоритма не означает неизбежность его последовательного выполнения. Предложенные методы распараллеливания реализаций таких алгоритмов и балансировки получающейся вычислительной нагрузки могут способствовать созданию эффективной параллельной программы, полностью использующей предоставленные ей аппаратные возможности современных вычислительных систем.

Ключевые слова: параллельное программирование; распараллеливание программ; балансировка вычислительной нагрузки

DOI: 10.15514/ISPRAS-2018-30(2)-2

Для цитирования: Бугеря А.Б., Ким Е.С., Соловьев М.А. Распараллеливание реализаций сугубо последовательных алгоритмов. Труды ИСП РАН, том 30, вып. 2, 2018 г., стр. 25-44. DOI: 10.15514/ISPRAS-2018-30(2)-2

1. Введение

В современном мире имеется огромное количество различных задач, решение которых требует наличия значительных вычислительных мощностей. Такие задачи имеются во всех областях науки и промышленности, в бизнесе и даже в сфере индивидуального применения. Типичными примерами таких

ресурсоемких задач могут служить решение задач математической физики численными методами (например, моделирование процессов, происходящих в ядерном реакторе), моделирование физических, химических и биологических процессов с огромным количеством взаимодействующих сущностей различной природы, анализ и преобразования графов, поиск и анализ информации в базах данных или в информационных потоках, статический и динамический анализ программ. Постоянно появляются новые задачи подобного рода.

Время выполнения программ, решающих такие задачи, может оказаться критически большим – зачастую таким, что это становится неприемлемым для использования такой программы. И даже просто весьма долгое выполнение каких-то задач, скажем, анализа программы при исследовании ее аналитиком в интерактивном режиме, существенно снизит производительность труда и увеличит время решения задачи. Поэтому не будет преувеличением сказать, что задача разработки эффективных программ имеет важное стратегическое значение. В данной работе мы не будем касаться вопроса скорости работы аппаратной части вычислительного комплекса или вопросов создания новых алгоритмов, быстрее решающих поставленную задачу. Наша цель – ускорить процесс решения программой поставленной задачи путем оптимизации самой программы. При этом особый интерес представляют программы, реализующие последовательный алгоритм решения задачи, который не имеет параллельного аналога.

2. Ускорение процесса выполнения программы

В процессе решения задачи оптимизации программы с целью ускорить время ее выполнения можно выделить следующие основные направления.

2.1 Выбор алгоритма

Выбор оптимального алгоритма, позволяющего решать поставленную задачу в заданном окружении (окружение – это в первую очередь целевой аппаратный вычислительный комплекс, а также предполагаемый набор входных данных). Забегая вперед, надо отметить, что если предполагается распараллеливание программы (см. разд. 3), то и выбор алгоритма должен производиться с учетом этого: некоторые алгоритмы могут быть эффективно распараллелены, а некоторые – нет. Зачастую бывает так, что для решения одной и той же задачи существует несколько алгоритмов. Одни из них более простые и эффективные в последовательном исполнении, другие же могут быть более сложными, менее эффективными в последовательном исполнении, но зато допускающими эффективное параллельное выполнение.

2.2 Реализация алгоритма

Оптимальная реализация выбранного алгоритма. Это немаловажный шаг в процессе построения эффективной программы, но в рамках данной работы мы

его подробно рассматривать не будем. Отметим только, что, прежде чем переходить к распараллеливанию программы и оптимизации ее параллельной версии, необходимо добиться того, чтобы последовательная версия работала оптимально: исключить повторные вычисления (кэшировать уже вычисленные данные для повторного использования, когда это возможно), исключить постоянное выделение/освобождение памяти (использовать пул предварительно выделенной памяти в таком случае), и провести прочие тому подобные оптимизации.

2.3 Распараллеливание реализации алгоритма

Распараллеливание реализации выбранного алгоритма. Когда все возможности увеличения скорости работы последовательной программы уже исчерпаны, а результат все же оставляет желать лучшего, помочь может только распараллеливание программы и последующее ее выполнение в окружении, эффективно поддерживающем выполнение параллельных программ. Параллельные вычислительные системы – это различные аппаратные решения, которые при поддержке соответствующих программных средств реализуют тем или иным способом одновременное (параллельное) выполнение различных команд, или одной и той же команды с различными наборами данных. Главная цель использования параллельных вычислений – повышение скорости вычислений за счет их параллельного выполнения. Главным критерием качества распараллеливания вычислений является сокращение общего времени решения задачи.

Идея распараллеливания вычислений базируется на том, что большинство задач может быть разделено на набор меньших независимых (или хотя бы мало зависимых) друг от друга подзадач, которые могут быть решены одновременно. Решая такой набор подзадач на параллельной вычислительной системе, можно добиться существенного уменьшения времени работы всей программы. Существует два основных типа распараллеливания (выделения подзадач): по управлению, когда в общей задаче можно выделить несколько независимых решаемых подзадач, каждая из которых выполняется со своим набором данных, и по данным, когда массив всех обрабатываемых данных делится на части, и с каждой такой частью данных решается одна и та же задача.

Очевидно, что возможности распараллеливания по управлению достаточно сильно ограничены: обычно бывает сложно найти в решаемой задаче хотя бы несколько ресурсоемких различных подзадач, которые могут быть выполнены параллельно. А случаи, когда таких подзадач десятки и более – скорее исключения, подтверждающие правило. Поэтому при распараллеливании по управлению рассчитывать можно лишь на некоторое ускорение – не больше, чем в такое количество раз (в идеальном случае), сколько удалось выделить подзадач. На практике ускорение будет еще меньшим, определяться временем выполнения наиболее длительной подзадачи плюс накладные расходы по организации подзадач и их взаимодействия. Тем не менее, при отсутствии

возможности распараллеливания по данным, на вычислительных системах с небольшим количеством параллельных вычислителей (а сейчас все даже бытовые компьютеры и мобильные устройства являются такими благодаря использованию многоядерных процессоров) такой метод распараллеливания может быть вполне успешно применен.

Возможности распараллеливания по данным выглядят гораздо более привлекательными. Объемы обрабатываемых данных в ресурсоемких задачах обычно огромны (или, по крайней мере, значительны). Если все обрабатываемые данные можно разделить на множество наборов (для типичных задач в идеале – по числу имеющихся в целевой вычислительной системе вычислителей), которые могут быть обработаны хотя бы на одном шаге обработки независимо, то, выполняя обработку всех полученных наборов данных параллельно, можно получить ускорение, в идеальном случае приближенное к числу имеющихся в целевой вычислительной системе вычислителей. Конечно, не так часто встречаются реальные прикладные задачи, в которых обрабатываемые данные можно разделить на совсем независимые группы. Для разрешения этих зависимостей приходится прибегать к синхронизации, передаче данных между вычислителями и прочим дополнительным действиям, что, конечно, снижает эффективность распараллеливания. Но, тем не менее, во многих случаях распараллеливание по данным, даже при наличии зависимости между данными, достаточно эффективно для его успешного практического применения.

При наличии зависимостей между данными, не позволяющими добиться высокой эффективности при распараллеливании обработки данных на весь набор имеющихся вычислителей, но показывающих неплохую эффективность при использовании нескольких вычислителей, хорошим решением может оказаться комбинирование подходов распараллеливания по данным и по управлению.

3. Сугубо последовательные алгоритмы

К сожалению, в ресурсоемких прикладных задачах нередко встречается ситуация, когда применяемый алгоритм является сугубо последовательным, то есть не позволяющим произвести распараллеливание ни по управлению, ни по данным. Например, когда никаких сущностей, допускающих параллельное выполнение, в этом алгоритме нет, а процесс обработки данных на каждом шаге зависит от всех обработанных данных на всех предыдущих шагах. И альтернативы применяемому алгоритму тоже нет.

В данной статье в качестве примеров будут рассмотрены алгоритмы и их программные реализации, которые были использованы авторами при разработке среды динамического анализа программ по бинарным трассам [1]. Среда динамического анализа программ по бинарным трассам разрабатывается в Институте системного программирования им. В.П. Иванникова РАН. Целевая аппаратная платформа для выполнения среды динамического анализа – мощная

персональная рабочая станция с достаточно большим количеством процессорных ядер (12 и более) и большим объемом общей оперативной памяти. Анализируемые бинарные трассы могут быть очень большого объема – до 300 Гб в сжатом виде. Среда динамического анализа представляет в помощь аналитику широкий набор алгоритмов предварительного анализа бинарных трасс, повышающих уровень представления исследуемых программ и значительно облегчающих труд аналитика [2, 3]. На огромных объемах обрабатываемых данных алгоритмы предварительного анализа могут выполняться значительное время – вплоть до нескольких суток. Поэтому, несомненно, задача ускорения работы таких алгоритмов в среде динамического анализа программ является крайне актуальной. Некоторые используемые алгоритмы хорошо могут быть распараллелены по данным. Это, например, различные алгоритмы поиска, алгоритм разметки трассы на процессы, потоки выполнения и зоны. Некоторые же алгоритмы не могут быть распараллелены из-за зависимости по данным. Рассмотрим типичные примеры таких алгоритмов подробнее.

3.1 Алгоритм построения графа потока данных

С небольшим упрощением граф потока данных представляет собой несвязанный направленный граф, в котором вершины представляют собой ячейки памяти или регистры (в дальнейшем будем называть их элементами). Дуга от одной вершины к другой означает передачу информации от соответствующего первого элемента ко второму, и дуга имеет пометку – номер шага исследуемой трассы, на котором произошла передача информации. Например, если на шаге s1 в регистр EAX было прочитано 4 байта памяти по адресу 0x12345678, на шаге s2 к регистру EAX было прибавлено значение регистра EDX, и на шаге s3 содержимое регистра EAX было записано в память по адресу 0x87654321, то соответствующий граф выглядит следующим образом, рис.1:

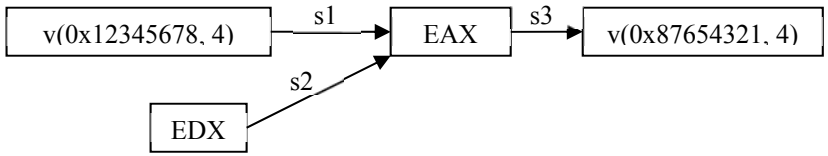

Рис. 1. Пример графа потока данных
Fig. 1. Data flow graph example

С помощью такого графа становится возможным отслеживать распространение помеченных данных, что используется в ряде других прикладных алгоритмов, таких как восстановление буфера, построение слайса и прочих.

Алгоритм построения графа потока данных работает следующим образом. На каждом шаге есть множество всех известных на этот момент элементов. Изначально это множество пусто. Рассматривается очередной шаг исследуемой программы. Если в нем есть чтение и/или запись каких-то элементов, то проверяется для каждого такого элемента, есть ли его пересечение с элементами текущего множества. Если пересечения нет, то элемент добавляется в множество. Если же пересечение есть, то возможны два варианта. Первый – когда элемент в точности соответствует какому-то элементу из множества. В этом случае множество не меняется. Второй вариант – когда элемент частично пересекается с одним или несколькими элементами множества. Тогда, если это запись элемента, то элемент включается в множество и в граф, а затем начинают рассматриваться все такие пересекаемые элементы множества. Пересекаемый элемент удаляется из множества. Если пересечение составляло лишь часть элемента, то оставшиеся части образуют новые элементы, которые вновь включаются в множество и в граф, и создаются дуги от изначального элемента множества к вновь добавленным. Если же это было чтение элемента, то читаемый элемент разбивается на несколько элементов-пересечений, и каждый такой получившийся элемент рассматривается в операции чтения по отдельности. Он уже либо не пересекается с множеством, либо в точности соответствует одному элементу из множества. И, в завершение, создаются дуги между элементами, участвующими в операции текущей инструкции.

Именно постоянное «дробление» элементов на части и их обратная «склейка» при появлении элемента, объемлющего два или более других, являются основной причиной, почему алгоритм построения графа потока данных не может быть распараллелен. Ведь если попробовать распараллелить алгоритм по данным, то во всех обработчиках, кроме первого, получится, что трасса просматривается без предыстории, и нет информации, какие элементы должны сейчас находиться в текущем множестве и какие дуги должны быть созданы на текущем шаге.

3.2 Алгоритм построения стека вызовов

Стек вызовов предоставляет аналитику в каждой точке трассы информацию о находящихся выше вызовах текущего выполняемого потока. Для каждого вызова указана точка вызова, точка возврата и находящийся выше вызов. В дальнейшем, после распознавания модулей, построения функций и восстановления параметров вызовов, эта информация будет дополнена именем функции и фактическими параметрами вызова, и такой стек вызовов будет выглядеть так же, как будто бы исследуемая программа выполняется под отладчиком и остановлена в какой-то точке. Пример построенного стека вызовов приведен на рис. 2.

позиция в трассе

c1 CALL

```
c11      CALL
 r2      RET
c12      CALL
c121       CALL
r12      RET
 r1    RET
```

Рис. 2. Пример построенного стека вызовов
Fig. 2. Call stack example

С некоторыми упрощениями алгоритм построения стека вызовов можно описать следующим образом. Последовательно просматривается вся трасса, от первой до последней инструкции. При обнаружении на очередном шаге инструкции вызова подпрограммы, сохраняются ее параметры: шаг, идентификатор выполняющегося потока, адрес выполняющейся инструкции плюс ее размер (фактически, ожидаемый адрес инструкции после возврата из вызова). Ближайшая сохраненная ранее инструкция вызова подпрограммы с таким же идентификатором выполняющегося потока (если таковая имеется) полагается вышележащим вызовом, и эта информация также сохраняется.

Когда же на очередном шаге обнаруживается инструкция возврата из подпрограммы, предпринимается попытка сопоставить ее с одной из сохраненных ранее инструкцией вызова. Для этого определяется адрес инструкции, следующей за инструкцией возврата, и затем в обратном порядке просматриваются сохраненные ранее инструкции. Для каждой сохраненной инструкции сравнивается идентификатор выполняющегося потока с сохраненным, и, если они совпадают, проверяется, не совпадает ли адрес инструкции, следующей за инструкцией возврата, с сохраненным ожидаемым адресом инструкции после возврата. Если адреса совпадают, то инструкции вызова и возврата полагаются парными, т.е. соответствующими вызову подпрограммы и возврату из нее. Все инструкции вызовов с таким же идентификатором текущего потока, сохраненные после объявленной парной инструкции вызова (если таковые имеются), полагаются не имеющими своей парной инструкции возврата (такое часто встречается в коде ядер операционных систем) и образующими пару с той же «чужой» инструкцией возврата, для которой была только что найдена пара. Все такие пары сохраняются как результат и, в завершение, все инструкции вызова, которые на данном шаге образовали пару, удаляются из сохраненных ранее инструкций вызова. Дальнейший просмотр сохраненных ранее инструкций вызова для текущей инструкции возврата прекращается.

Если для текущей инструкции возврата парная инструкция вызова не находится (такое также часто встречается в коде ядер операционных систем), то такая инструкция возврата игнорируется.

Приведенный выше алгоритм позволяет построить хорошо согласованный стек вызовов для исследуемой программы даже и тогда, когда выполнение программы прерывается выполнением кода операционной системы.

Очевидно, что для правильного построения стека вызовов необходима история всех предыдущих вызовов, иначе, если начать обработку инструкций трассы не с самого начала, возможно не найти для текущей инструкции возврата соответствующей инструкции вызова, а также возможно не найти вышележащий вызов. Поэтому алгоритм построения стека вызовов является сугубо последовательным и распараллеливанию не подлежит.

4. Методы распараллеливания сугубо последовательных алгоритмов

Так что же делать, если другого алгоритма, решающего поставленную задачу, не существует, а имеющийся является сугубо последовательным, работает долго на значительных объемах данных, и крайне необходимо его ускорить? Казалось бы, выхода нет? Но это не всегда так, если попробовать взглянуть на поставленную задачу немного по-другому: распараллелить надо не алгоритм, а его реализацию. Пусть алгоритм так и останется последовательным, но если в его реализации в программе суметь найти возможности параллельного выполнения каких-то действий, можно попробовать добиться ускорения выполнения задачи, и, зачастую, весьма существенного.

4.1 Метод 1: все равно «разрезать», а потом «склеить»

Суть данного метода достаточно проста. Несмотря на то, что для корректного выполнения алгоритма в каждой рассматриваемой точке необходимо иметь какой-то набор данных, построенный по всем предыдущим точкам, все равно распределить обрабатываемые данные на равные непрерывные части по числу имеющихся вычислителей. Затем каждую часть данных обрабатывать параллельно так, как будто это первая часть и никакой предыстории нет. После того, как все части обработаны, надо осуществить «склейку» и коррекцию полученных результатов. Этот процесс выполняется последовательно: вначале первая часть со второй дают промежуточный результат, затем к нему также «склеивается» третья часть и так далее.

Для осуществления процесса «склейки» берется набор данных, построенный по предыдущей части, и проверяется, как он повлиял на результат, построенный по «подклеиваемой» части. Возможно, при этом придется заново обрабатывать начальную часть данных «подклеиваемой» части и корректировать полученный ранее результат. Но если эта заново обработанная начальная часть данных мала по сравнению со всем объемом «подклеиваемой» части, и/или повторная обработка уже опирается на полученный результат и выполняется на порядок быстрее первичной, то все равно, несмотря на повторную обработку каких-то данных, можно добиться существенного ускорения выполнения всего

алгоритма в целом. Последовательность процесса «склейки» также обусловлена тем, что помимо коррекции результата необходима корректировка текущего набора данных, полученного в конце «подклеиваемой» части, с учетом текущего набора данных, полученного в конце предыдущей части. Это необходимо для корректного «подклеивания» следующей части.

Очевидно, что формат представления частичного результата алгоритма, а также формат представления текущего набора данных должен позволять производить процесс их коррекции.

Вернемся к рассмотренному выше алгоритму построения графа потока данных. Разделим всю обрабатываемую трассу на равные части по числу имеющихся вычислителей, и начнем обработку каждой части с пустым множеством известных элементов. Но при этом для всех частей, кроме первой, будем дополнительно запоминать случаи, когда рассматриваемого элемента в текущем множестве элементов не было. Именно эти случаи, возможно, в дальнейшем потребуют коррекции.

Затем, после завершения обработки первой и второй части, можно начинать процесс их «склейки». Для этого начинаем по одному исследовать элементы текущего множества, образовавшегося в конце первой части. Возможны следующие три варианта:

1. Если такой элемент из множества конца первой части не входит в множество элементов, впервые появившихся во второй части, то такой элемент добавляется во множество текущих элементов конца второй части, так как, получается, что такой элемент во второй части не участвовал ни в одной операции.
2. Если такой элемент из множества конца первой части полностью совпадает с каким-то элементом из множества впервые появившихся элементов во второй части, данные дуг этих элементов объединяются, так как это один и тот же элемент.
3. И, гораздо более сложный случай, когда элемент из множества конца первой части пересекается с одним или более элементом из множества впервые появившихся элементов во второй части. Не будем приводить здесь полностью весь алгоритм работы, отметим лишь, что в этом случае требуется корректировка результата второй части, в котором присутствуют пересекаемые элементы, вплоть до того момента, когда эти пересекаемые элементы не были вновь объединены операцией записи объемлющего элемента, равного элементу из множества впервые появившихся элементов во второй части.

После завершения работы над «склейкой» второй части можно приступать к аналогичному процессу по отношению к полученному результату «склейки» второй части и модифицированному множеству текущих элементов конца второй части, с одной стороны, с результатом третьей части, с другой. И так далее, вплоть до «склейки» последней части. В итоге получается тот же

результат, что и был бы получен в случае последовательной обработки всего набора данных.

А вот распараллелить алгоритм построения стека вызовов методом разрезания и склеивания не представляется возможным. Все дело в том, что при завершении обработки первой части с большой вероятностью текущий стек незавершенных вызовов будет непустым. Более того, также с немалой вероятностью вызовы, находящиеся в нем, не будут полностью удалены из стека до конца обрабатываемой трассы. Типичный пример – вызов функции main(). Это приводит к тому, что вторую и последующие части придется фактически просматривать заново для корректировки результата. Очевидно, что никакого выигрыша от такого «распараллеливания» получено быть не может.

4.2 Метод 2: выделить независимые сущности в массиве обрабатываемых данных и обработать их независимо

Суть этого метода в том, что для параллельной обработки весь массив данных не разбивается на равные части из подряд идущих данных, а среди обрабатываемых данных выделяются части, возможно, разных размеров, каждая часть может состоять из множества разрозненных данных, но такие, что все эти части могут быть обработаны параллельно.

Вернемся к алгоритму построения стека вызовов. Очевидно, что стек вызовов строится независимо для каждого потока, присутствующего в исследуемой трассе. Поэтому можно для каждого потока параллельно получить множество инструкций, относящихся к этому потоку, обработать их, и получить частичный результат, относящийся к этому потоку. А затем, уже последовательно, объединить все полученные частичные результаты в итоговый, упорядочивая в нем все вызовы, полученные в частичных результатах, в порядке появления их в трассе.

Предложенный метод имеет существенный недостаток. Эффективность его применения сильно зависит от того, насколько выделенные независимые сущности в массиве обрабатываемых данных отличаются друг от друга по объему данных в каждой сущности. Например, если в исследуемой трассе имеется поток, занимающий большую часть трассы (или даже всю, если, например, исследуется однопотоковая встроенная система), то эффективность такого распараллеливания будет крайне низка.

Но в случаях, когда удается выделить хотя бы несколько сравнимых по объему существенных объемов данных, применение данного метода может дать весьма хороший результат.

4.3 Метод 3: распараллелить предварительную обработку данных

Попробуем взглянуть на задачу ускорения работы реализации какого-то сугубо последовательного алгоритма немного по-другому. Пусть алгоритм сугубо последователен, и его реализация тоже не может быть распараллелена предложенными выше методами. Но, обычно, в реализации каждого алгоритма, помимо действий по реализации собственно шагов алгоритма, есть какая-то предварительная работа. Например, в реализации алгоритма построения стека вызовов прежде, чем поместить инструкцию вызова в текущий стек вместе с ожидаемым адресом инструкции после возврата, нужно найти эту очередную выполненную инструкцию вызова и определить ее параметры. А это тоже занимает немалое время. Так почему бы не попробовать распараллелить эту работу, и предоставить последовательной части реализации алгоритма данные уже в предварительно обработанном виде, оптимизированном для их быстрой обработки этой последовательной частью?

Для этого в реализации алгоритма выделяется два типа рабочих процессов. Первый тип, существующий в единственном экземпляре, выполняет исключительно последовательную часть алгоритма. Рабочие процессы второго типа организуются в количестве имеющихся в системе аппаратных вычислителей. Для взаимодействия между процессами организуется очередь заданий. Каждое задание представляет собой некий объем данных для предварительной обработки. Рабочий процесс второго типа берет себе очередное задание из очереди, обрабатывает его, формирует набор предварительно обработанных данных, помечает задание как выполненное и берет себе следующее свободное задание из очереди. Рабочий процесс первого типа ждет готовности задания в голове очереди, по его готовности удаляет его из очереди и начинает его обработку. Схема работы очереди представлена на рис. 3.

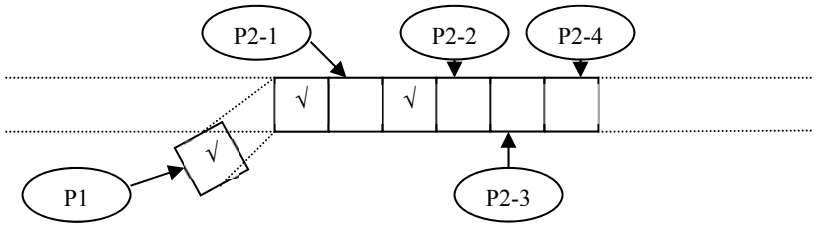

Рис. 3. Схема работы очереди заданий
Fig. 3. Job queue working diagram

Очевидно, что общая производительность такой реализации алгоритма не будет выше производительности того единственного рабочего процесса, реализующего последовательную часть на уже подготовленных данных. И,

если он не будет успевать обрабатывать все подготовленные данные, очередь будет сильно разрастаться. Подробнее организация очереди, балансировка нагрузки и распределение вычислительных ресурсов между рабочими процессами первого и второго типа будут рассмотрены ниже.

4.4 Комбинирование методов 2 и 3

Вернемся к методу 2, когда в массиве обрабатываемых данных можно выделить параллельно обрабатываемые сущности, но одна из таких сущностей занимает существенный объем от общих данных. Например, в обрабатываемой трассе один из потоков занимает 60% от всего объема трассы. Тогда, если распараллеливать обработку такой трассы по методу 2, иметь более чем два параллельно выполняющихся обработчиков не нужно – пока один из них будет обрабатывать самый большой поток, второй обработчик должен обработать все остальные. Таким образом, максимальное ускорение, которое возможно получить в данной ситуации – это менее чем в два раза.

Если же распараллелить предварительную обработку данных по методу 3, то единственный рабочий процесс первого типа, осуществляющий последовательную часть работы алгоритма, может не справляться с потоком предварительно обработанных данных, производимых множеством рабочих потоков второго типа. Но ничто не мешает применить одновременно оба метода. Вначале, следуя методу 2, выделяются независимые сущности в массиве данных, и пусть среди них будут такие, которые занимают существенный объем от общих данных. Вначале выясняем, по объему данных, сколько таких сущностей имеет смысл обрабатывать параллельно (подробнее этот процесс будет описан ниже, в главе про балансировку вычислительной нагрузки). И, если таких сущностей оказывается более одной, но существенно менее имеющихся в вычислительной системе вычислителей, к каждой такой параллельно обрабатываемой сущности далее применяем метод 3. То есть организуем несколько (по числу сущностей) очередей, в каждой из которых есть один рабочий поток первого типа и один или более рабочих потоков второго типа. При этом общее число рабочих потоков второго типа должно не превышать число имеющихся вычислителей. Когда какая-то очередь заканчивает свою работу, другие очереди могут увеличить число своих рабочих потоков второго типа.

Вернемся к примеру, когда один из потоков занимает 60% от всего объема трассы. Допустим, что выполнение происходит на 12-ядерной рабочей станции. Тогда, применяя метод 2, мы получим только 2 занятых ядра (а в конце работы и вовсе будет только одно). 10 ядер будут простаивать. Применение метода 3 даст следующую картину: 1 рабочий процесс первого типа и 11 рабочих процессов второго типа. Справится ли один процесс с последовательной обработкой данных, предоставленных одиннадцатью процессами предварительной обработки? Как показывает практика, в алгоритме построения стека – нет. И значительная часть ядер опять будет простаивать. А вот

применение комбинированного подхода дает уже гораздо более сбалансированную картину: 2 потока первого типа, у каждого – по 5 потоков второго типа. И (опять же в нашем примере – в случае алгоритма построения стека) загрузка всей системы становится близка к 100%, все имеющиеся вычислительные ресурсы используются эффективно, общее выполнение алгоритма ускоряется в несколько раз.

5. Балансировка вычислительной нагрузки

В отличие от простого распараллеливания по данным, когда весь объем данных делится на равные части и распределяется по имеющимся вычислителям, при использовании описанных выше методов распараллеливания сугубо последовательных алгоритмов, гораздо более сложных, с различными типами взаимодействующих вычислительных процессов, задача правильной балансировки вычислительной нагрузки между вычислительными процессами становится крайне актуальной и совсем не тривиально решаемой. От правильной балансировки вычислительной нагрузки зависит успешное решение задачи в целом – ускорение выполнения реализации алгоритма. Несбалансированная вычислительная нагрузка может свести на нет все усилия по распараллеливанию.

Балансировка вычислительной нагрузки может производиться как статически, путем первоначального распределения данных, так и динамически – во время выполнения реализации алгоритма, путем перераспределения части данных и/или путем временного приостанавливания отдельных вычислительных процессов. Предпочтительнее, по возможности, использовать оба подхода.

Рассмотрим различные методы балансировки вычислительной нагрузки, которые могут быть применены в зависимости от используемого метода распараллеливания сугубо последовательных алгоритмов.

5.1 Статическая балансировка с весами

При использовании метода 1 (все равно «разрезать», а потом «склеить») статическое распределение данных между вычислителями должно быть, казалось бы, простое – всем поровну, как при простом распараллеливании. Но это не так, если учесть, что после процесса параллельной обработки всех данных предстоит еще процесс «склейки», а он будет производиться последовательно от первой части к последней. Для того чтобы начать процесс «склейки», необходима готовность первой и второй части, а третья и последующие части могут продолжать при этом обрабатываться. Только после того, как будет завершен процесс «склейки» первой и второй части, нужна будет готовность третьей.

Таким образом, идеальным распределением нагрузки будет такое, когда каждая последующая часть будет готова как раз к тому моменту, когда будет завешен процесс «склейки» предыдущей части. Для достижения этого каждой части

нужно назначить вес обрабатываемых данных. Первая и вторая части получают наименьший вес, далее – равномерное нарастание веса вплоть до максимального у последней части. Как правильно выбрать соотношение минимального и максимального веса – зависит от алгоритма и от типичного времени для процесса «склейки» и, видимо, должно подбираться экспериментально. Для алгоритма построения графа потока данных эти значения оказались следующими (при наличии достаточного количества вычислительных ядер): от 0.7 для первой и второй части до 1 для последней.

5.2 Статическая балансировка упорядочиванием обрабатываемых данных

При использовании метода 2 балансировка вычислительной нагрузки также производится статически. Для начала надо оценить объем обрабатываемых данных в каждой независимой сущности и упорядочить сущности по уменьшению этого объема. Именно в этом порядке сущности будут выдаваться вычислителям для обработки: сначала те, которые имеют больший объем данных, затем – меньший. Вычислительный процесс берет очередную сущность с головы очереди, обрабатывает ее, и берет следующую. Сущности с наибольшими объемами данных таким образом будут обработаны в первую очередь, а конце работы останется набор сущностей с наименьшими объемами данных, что даст и в конце работы алгоритма равномерную загрузку вычислительных процессов.

Кроме того, перед началом работы следует оценить необходимое количество вычислительных процессов. При более-менее равномерном распределении объема данных между сущностями, и когда таких сущностей достаточно много, вычислительных процессов должно быть столько, сколько в вычислительной системе имеется аппаратных вычислителей. Но если имеется сущность (или несколько сущностей, но меньше, чем число аппаратных вычислителей), объем данных которой (которых) имеет в общем объеме данных превалирующее значение, то необходимо ограничить число вычислительных процессов. Так как все равно при таком неравномерном распределении данных общее время работы алгоритма будет определяться временем работы вычислительного процесса с сущностью с наибольшим объемом данных, а другие вычислительные процессы, обработав остальные сущности, затем будут простаивать. В то время как изначальное снижение числа вычислительных процессов снизит их конкуренцию за общие ресурсы (например, доступ к памяти или файлам на жестком диске) и, таким образом, увеличит скорость работы вычислительного процесса с сущностью с наибольшим объемом данных.

Для оценки количества необходимых вычислительных процессов предлагается использовать следующую формулу. Пусть имеется n сущностей, $n > 1$ и $p_i, i = 1..n$ – объем обрабатываемых данных i-той сущности, такой, что

$p_1 \geq p_2 \geq \ldots \geq p_n$. Тогда минимальное значение k, при котором будет выполняться условие $\sum_{i=1}^{k-1} p_i \geq \sum_{i=k}^{n} p_i$ и есть число необходимых вычислительных процессов для параллельной обработки всех сущностей.

5.3 Динамическая балансировка очереди заданий

При использовании метода 3 балансировка вычислительной нагрузки производится динамически, путем управления очередью заданий и количеством активных рабочих процессов второго типа. Все данные для предварительной обработки делятся на сравнительно небольшие подряд идущие куски. Рабочих процессов второго типа организуется столько, сколько в вычислительной системе имеется аппаратных вычислителей. Рабочий процесс второго типа в начале своей работы, или закончив свое предыдущее задание, обращается к очереди за очередным заданием. Ему оформляется очередной, пока еще не выданный на обработку кусок данных в виде задания, это задание помещается в конец очереди и выдается на обработку рабочему процессу. Рабочий процесс, выполнив задание, помечает его в очереди как выполненное и обращается за новым.

Рассмотрим теперь, что будет происходить, когда рабочий процесс первого типа, собственно осуществляющий обработку предварительно подготовленных данных согласно алгоритму, не будет успевать обрабатывать подготовленные ему данные и будет выбирать с головы очереди выполненные задания медленнее, чем очередь будет пополняться с хвоста рабочими процессами второго типа. Очевидно, что при таком развитии событий очередь будет разрастаться, занимая ресурсы оперативной памяти. А когда все данные будут предварительно подготовлены и помещены в очередь, рабочие процессы второго типа прекратят свою работу, и останется выполняться только один рабочий процесс первого типа, который будет долго еще разбирать подготовленные ему данные. Помимо излишнего потребления оперативной памяти это также приведет к тому, что, пока все рабочие процессы второго типа работали, они конкурировали, в том числе и с рабочим процессом первого типа, за системные ресурсы. И, таким образом, рабочий процесс первого типа, от которого собственно зависит время выполнения всего алгоритма, в условиях конкуренции работал гораздо медленнее, чем мог бы работать при меньшей конкуренции.

Для решения этой проблемы предлагается достаточно простое решение – ограничить размер очереди по сумме выданных и выполненных заданий. Когда рабочий процесс второго типа обращается к очереди за следующим заданием, проверяется, сколько сформированных заданий (в сумме как выполненных, так и тех, над которыми еще ведется работа другими процессами второго типа) находится в очереди. Если их меньше установленного максимального размера

очереди, очередное задание выдается обратившемуся процессу. Если же максимальный размер очереди достигнут, то обратившийся рабочий процесс второго типа приостанавливается.

Когда рабочий процесс первого типа вынимает из головы очереди выполненное задание с предварительно подготовленными данными, тем самым уменьшая размер очереди на единицу, проверяется, нет ли приостановленных рабочих процессов второго типа. И, если есть, то любой один из них запускается, и ему выдается очередное задание. Таким способом достигается, что в каждый момент времени выполняется ровно столько рабочих процессов второго типа, чтобы создать непрерывную загрузку рабочему процессу первого типа, и не более того. За счет этого уменьшается конкуренция процессов за ресурсы, и рабочий процесс первого типа, а вместе с ним и весь алгоритм, выполняются максимально быстро.

Максимальный размер очереди должен быть таким, чтобы, с одной стороны, обеспечить непрерывную загрузку подготовленными данными рабочего процесса первого типа, а с другой стороны – не потреблять неоправданно системные ресурсы. Как показала практика, установка максимального размера очереди как трехкратное количество рабочих процессов второго типа полностью соответствует этим критериям.

Обратная ситуация, когда рабочий процесс первого типа успевает работать быстрее, чем рабочие процессы второго типа подготавливают ему предварительно обработанные данные, не требует никакой балансировки. Притом, что рабочих процессов второго типа должно быть столько, сколько в вычислительной системе имеется аппаратных вычислителей, следует, что и загрузка системы в таком случае будет близка к 100%.

5.4 Комбинированная балансировка при использовании комбинированных методов

Очевидно, что при использовании комбинирования методов распараллеливания 2 и 3, балансировка вычислительной нагрузки также должна производиться комбинировано. Вначале необходимо статически, точно так же, как для метода 2, отсортировать сущности по объему данных и определить необходимое количество независимых вычислительных процессов. Затем, в каждом таком процессе, создать очередь заданий и динамически управлять ей, как описано выше. Каждая такая очередь должна управляться независимо от других, с одним исключением: если в одной из очередей появляются приостановленные рабочие процессы второго типа, а в другой – нет, возможно организовать передачу приостановленного процесса в другую очередь, но с обязательным возвращением его обратно в случае, если потом, возможно, ситуация изменится, и размер первой очереди начнет уменьшаться.

6. Результаты применения методов распараллеливания сугубо последовательных алгоритмов

В данной главе приведены временные характеристики выполнения алгоритма построения графа потока данных и алгоритма построения стека вызовов на различных обрабатываемых данных и при различном количестве предоставляемых программе аппаратных процессорных ядер. Все измерения производились на персональной рабочей станции следующей конфигурации: 2 процессора Intel Xeon E5-2690 v2, по 10 вычислительных ядер в каждом, поддержка гипертрединга включена, размер оперативной памяти – 192 Гб.

График времени выполнения алгоритма построения графа потока данных приведен на рис. 4 для одной трассы архитектуры x86, размера 2.6 Гб (370 398 536 шагов).

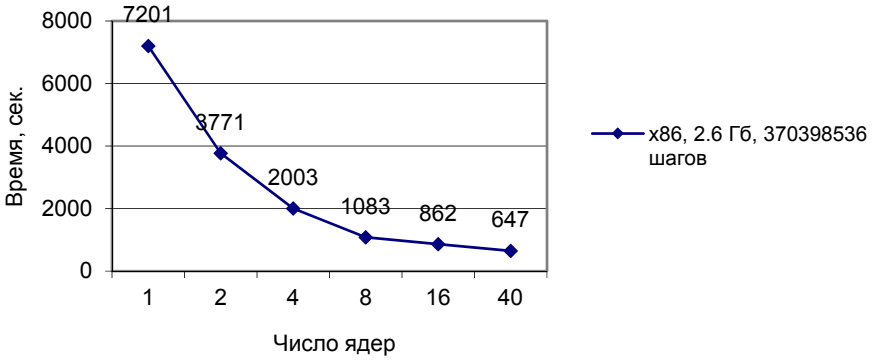

Рис. 4. Время выполнения алгоритма построения графа потока данных
Fig. 4. Dependency graph building time

На трассах для других архитектур и объемов график времени выполнения алгоритма ведет себя подобным образом. Параллельная реализация алгоритма построения графа потока данных, выполненная по методу 1, демонстрирует хорошую масштабируемость, давая в итоге при задействии всех 20 аппаратных процессорных ядер (40 – это с гипертредингом) ускорение более чем в 10 раз по сравнению с последовательным выполнением.

Далее приведены графики времени выполнения алгоритма построения стека вызовов для различных трасс. При распараллеливании реализации этого алгоритма вначале производится оценка имеющихся потоков в исследуемой трассе по методу 2. Затем выбирается метод распараллеливания: 2, 3 или их комбинация. Для каждого метода в качестве примера приводится по одной трассе.

На рис. 5 приведены нормированные (время выполнения последовательной версии алгоритма взято за 100) графики времени выполнения алгоритма построения стека вызовов для следующих трасс:

- Та же трасса, что и на рис. 4, архитектуры x86, размера 2.6 Гб (370 398 536 шагов). Здесь и далее также приведем еще один параметр – в трассе имеется 192 потока выполнения, для обработки которых можно задействовать 61 параллельный вычислитель. Таким образом, эта трасса имеет хорошие возможности для параллельной обработки, и при распараллеливании реализации алгоритма применялся метод 2.
- Трасса архитектуры MIPS, размера 3.3 Гб (1 933 615 005 шагов), в трассе имеется единственный поток выполнения, поэтому применение к ней метода 2 невозможно, при распараллеливании реализации алгоритма применялся метод 3.
- Трасса архитектуры ARM, размера 1.9 Гб (641 679 867 шагов), в трассе имеется 96 потоков выполнения, но для обработки их можно задействовать только 2 параллельных вычислителей, так как один из потоков занимает более половины трассы. Поэтому при распараллеливании реализации алгоритма применялась комбинация методов 2 и 3.

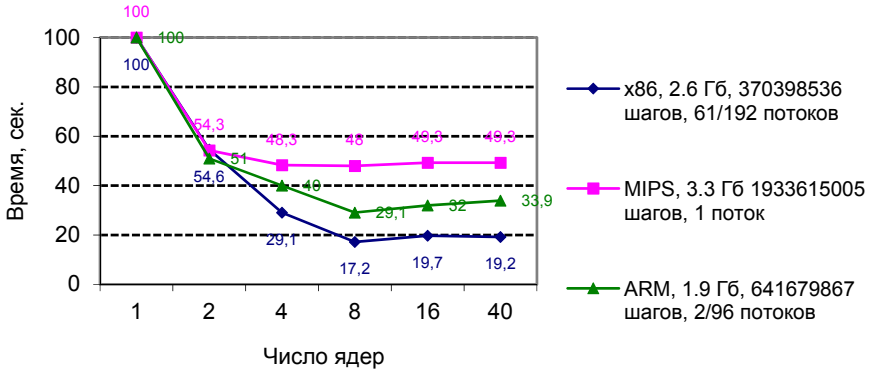

Рис. 5. Время выполнения алгоритма построения стека вызовов
Fig. 5. Call stack building time

Трасса x86, имеющая множество равнообъемных потоков выполнения, показывает хорошие результаты распараллеливания, уменьшая время своего выполнения вплоть до использования 8 процессорных ядер. Дальнейшее увеличение количества процессорных ядер никакого увеличения скорости работы уже не приносит. По-видимому, дело здесь уже в производительности жесткого диска, с которого и читается сама трасса, и пишется частичный результат для каждого потока. Тем не менее, увеличение скорости работы более

чем в 5 раз по сравнению с последовательным вариантом для выполнения такого алгоритма – мы полагаем, очень хороший результат.

Среди приведенных примеров тяжелее всего с распараллеливанием обработки трассы MIPS, имеющей только один поток. Применяется метод 3, и на единственный рабочий процесс первого типа при увеличении числа процессорных ядер пропорционально увеличивается число рабочих процессов второго типа, осуществляющих предварительную подготовку данных. Как видно по графику, использование их в количестве более четырех уже не приводит к росту производительности – единственный рабочий процесс первого типа уже не справляется с получающейся нагрузкой. Тем не менее, увеличение скорости работы более чем в 2 раза по сравнению с последовательным вариантом достигнуто даже в таком, совсем плохо подходящем для распараллеливания случае.

А вот если в трассе появляется возможность использования еще хотя бы одного рабочего процесса первого типа (трасса ARM), и распараллеливание выполняется комбинированным методом, то и результат получается гораздо лучше – увеличение скорости работы вплоть до использования 8 процессорных ядер и общее ускорение почти в 4 раза по сравнению с последовательным вариантом.

7. Заключение

Основная цель данной работы – показать, что использование в программной реализации сугубо последовательного алгоритма не означает неизбежность его последовательного выполнения. Предложенные в статье методы распараллеливания реализаций таких алгоритмов и балансировки получающейся вычислительной нагрузки могут поспособствовать созданию эффективной параллельной программы, полностью использующей предоставленные ей аппаратные возможности современных вычислительных систем.

Список литературы

[1]. В.А. Падарян, А.И. Гетьман, М.А. Соловьев, М.Г. Бакулин, А.И. Борзилов, В.В. Каушан, И.Н. Ледовских, Ю.В. Маркин, С.С. Панасенко. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8.

[2]. В.А. Падарян. О представлении результатов обратной инженерии бинарного кода. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 31-42. DOI: 10.15514/ISPRAS-2017-29(3)-3.

[3]. Alexander Getman, Vartan Padaryan, Mikhail Solovyev. Combined approach to solving problems in binary code analysis. Proceedings of the 9th International Conference on Computer Science and Information Technologies (CSIT), 2013, pp. 295-297.

Parallelization of implementations of purely sequential algorithms

[1] *A.B. Bugerya <shurabug@yandex.ru>*
[2] *E.S. Kim <eugene.kim@ispras.ru>*
[2] *M.A. Solovev <icee@ispras.ru>*
[1] *Keldysh Institute of Applied Mathematics of the Russian Academy of Sciences, Miusskaya sq., 4, Moscow, 125047, Russia*
[2] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*

Abstract. The work is dedicated to the topic of parallelizing programs in especially difficult cases - when the used algorithm is purely sequential, there are no parallel alternatives to the algorithm used, and its execution time is unacceptably high. Various parallelization methods for software implementations of such algorithms and resulting computational load balancing are considered, allowing to obtain significant performance acceleration for application programs using purely sequential algorithms. The above methods are illustrated by the practice of their application to two algorithms used in a dynamic binary code analysis toolset. The main goal of this paper is to show that the use of a purely sequential algorithm in a software implementation does not necessarily imply inevitability of its sequential execution. The proposed methods of parallelizing implementations of such algorithms and balancing the resulting computational load can help to develop efficient parallel program that fully utilize the hardware capabilities of modern computing systems.

Keywords: parallel programming; program parallelization; computational load balancing.

DOI: 10.15514/ISPRAS-2018-30(2)-2

For citation: Bugerya A.B., Kim E.S., Solovev M.A. Parallelization of implementations of purely sequential algorithms. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 2, 2018, pp. 25-44 (in Russian). DOI: 10.15514/ISPRAS-2018-30(2)-2

References

[1]. V.A. Padaryan, A.I. Getman, M.A. Solovyev, M.G. Bakulin, A.I. Borzilov, V.V. Kaushan, I.N. Ledovskich, U.V. Markin, S.S. Panasenko. Methods and software tools for combined binary code analysis. Trudy ISP RAN/Proc. ISP RAS, 2014, vol. 26, issue 1, pp. 251-276 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-8.

[2]. V.A. Padaryan. On representation used in the binary code reverse engineering. Trudy ISP RAN/Proc. ISP RAS, 2017, vol. 29, issue 3, pp. 31-42 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-3.

[3]. Alexander Getman, Vartan Padaryan, Mikhail Solovyev. Combined approach to solving problems in binary code analysis. Proceedings of the 9th International Conference on Computer Science and Information Technologies (CSIT), 2013, pp. 295-297.