Методы балансировки вычислительной нагрузки при параллельной реализации алгоритмов на общей памяти

Т.П. Баранова[1], А.Б. Бугеря[1], Е.Н. Гладкова[1], К.Н. Ефимкин[1], М.А. Соловьев[2]

1 Институт прикладной математики им. М.В. Келдыша РАН
2 Институт системного программирования им. В.П. Иванникова РАН

Аннотация. В работе рассматриваются различные (как статические, так и динамические) методы балансировки вычислительной нагрузки при параллельной реализации алгоритмов на общей памяти. Также внимание уделено 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, использующих сугубо последовательные алгоритмы. Предложенные методы распараллеливания реализаций таких алгоритмов и балансировки вычислительной нагрузки на общей памяти могут поспособствовать созданию эффективных параллельных программ, полностью использующих аппаратные возможности современных вычислительных систем с общей памятью.

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

Computational load balancing methods for algorithms parallel implementations on shared memory.

T.P. Baranova[1], A.B. Bugerya[1], E.N. Gladkova[1], K.N. Efimkin[1], M.A. Solovev[2]

1 Keldysh Institute of Applied Mathematics of the Russian Academy of Sciences
2 Ivannikov Institute for System Programming of the Russian Academy of Sciences

Abstract. The work is dedicated to the topic of various (both static and dynamic) methods of balancing the computational load with parallel implementation of algorithms on shared memory. Attention is also paid 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 the 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 proposed methods of parallelizing implementations of such algorithms and balancing the resulting computational load on shared memory can help to develop efficient parallel program that fully utilize the hardware capabilities of modern computing systems with shared memory.

Keywords: parallel programming; program parallelization; computational load balancing

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 ресурсоёмких 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 её в интерактивном режиме аналитиком, существенно снизит производительность труда и увеличит время выполнения задания в целом. Поэтому не будет преувеличением сказать, что задача разработки эффективных программ имеет важное стратегическое значение. Эффективная программа решает поставленную задачу наиболее быстрым способом, полностью используя предоставленные ей аппаратные возможности вычислительного комплекса. Современные вычислительные системы, на использование которых ориентированы такие программы, безусловно, предусматривают возможность параллельных вычислений. Поэтому современная эффективная программа обязана быть параллельной.

Для своего эффективного выполнения параллельная программа должна обеспечить все имеющиеся в её распоряжении вычислители непрерывной загрузкой данными для вычислений. Но также, с другой стороны, она должна обеспечить синхронизацию вычислений, где необходимо, при обращении к общим данным, минимизировать простои вычислителей при синхронизации и при доступе к прочим ресурсам, как программным, так и аппаратным. Этот процесс называется балансировкой вычислительной нагрузки (computational

load balancing), и от его успешного выполнения в значительной степени зависит и эффективность выполнения программы в целом.

2. Распараллеливание выполнения программы

Когда все возможности увеличения скорости работы последовательной программы уже исчерпаны, а результат всё же оставляет желать лучшего, помочь может только распараллеливание программы и последующее её выполнение в окружении, эффективно поддерживающем выполнение параллельных программ. 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 ресурсоёмких 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 ещё 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 применён.

Возможности распараллеливания по данным выглядят гораздо более привлекательными. Объёмы обрабатываемых данных в ресурсоёмких 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.

К сожалению, в ресурсоёмких 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 будут приведены методы распараллеливания реализаций сугубо последовательных алгоритмов, а также методы балансировки вычислительной нагрузки при выполнении параллельных программ на вычислительной системе с общей памятью, 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 объёмом общей оперативной памяти. Анализируемые бинарные трассы могут быть очень большого объёма – 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 объёмах 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. Некоторые же алгоритмы не могут быть распараллелены из-за зависимости по данным. В работе [4] подробно рассмотрены примеры таких сугубо последовательных алгоритмов.

3. 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 объёмах 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.

Метод 1: всё 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, всё 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 «подклеивается» третья часть и так далее.

Для осуществления процесса «склейки» берётся набор данных, построенный по предыдущей части, и проверяется, как он повлиял на результат, построенный по «подклеиваемой» части. Возможно, при этом придётся 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 объёмом «подклеиваемой» части, и/или повторная обработка уже опирается на полученный результат и выполняется на порядок быстрее первичной, то всё 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 учётом 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.

В среде динамического анализа программ по бинарным трассам [1] реализация алгоритма построения графа потока данных [4] выполнена по этому методу.

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.

С помощью данного метода может быть выполнена реализация алгоритма построения стека вызовов [4]. 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 нём 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 объёму 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 удаётся выделить хотя бы несколько сравнимых по объёму существенных объёмов данных, применение данного метода может дать весьма хороший результат.

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 [4] прежде, чем поместить инструкцию вызова в текущий стек вместе с ожидаемым адресом инструкции после возврата, нужно найти эту очередную выполненную инструкцию вызова и определить её 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 объём данных для предварительной обработки. Рабочий процесс второго типа берёт себе очередное задание из очереди, обрабатывает его, формирует набор предварительно обработанных данных, помечает задание как выполненное и берёт себе следующее свободное задание из очереди. Рабочий процесс первого типа ждёт готовности задания в голове очереди, по его готовности удаляет его из очереди и начинает его обработку. Схема работы очереди представлена на рис. 1.

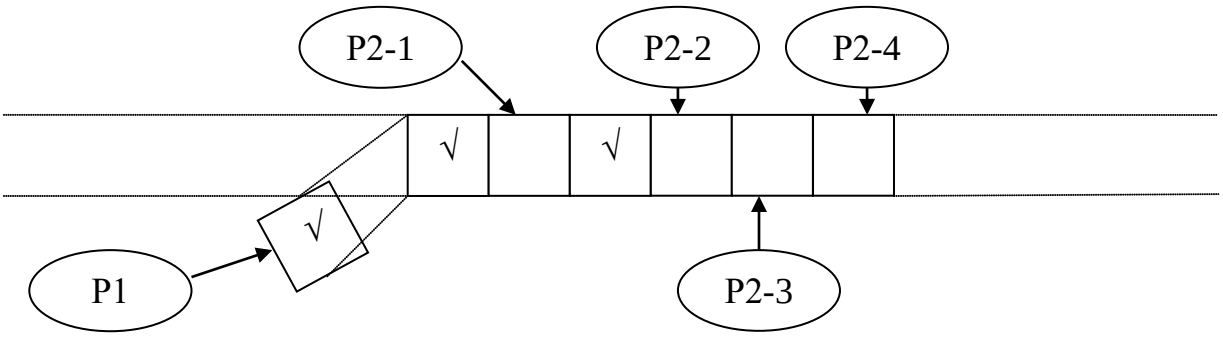

Рис. 1. Схема работы очереди заданий

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.

Комбинирование методов 2 и 3

Вернёмся к методу 2, когда в массиве обрабатываемых данных можно выделить параллельно обрабатываемые сущности, но одна из таких сущностей занимает существенный объём от общих данных. Например, в обрабатываемой трассе один из потоков занимает 60% от всего объёма 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 объём от общих данных. Вначале выясняем, по объёму 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.

4. Динамическая балансировка вычислительной нагрузки для множества однородных вычислительных потоков на общей памяти

В среде динамического анализа программ [1] существует большое количество реализаций разнообразных алгоритмов, осуществляющих обработку трассы, поиск различных данных, построение ресурсов и решающих ещё массу самых разнообразных задач. Большинство из них выполнены с помощью распараллеливания по данным. Другие, являясь реализациями сугубо последовательных алгоритмов, выполнены с использованием приведённых выше методов. При выполнении распараллеленных алгоритмов запускается несколько вычислительных рабочих потоков для параллельной обработки данных – обычно столько, сколько в системе имеется вычислительных ядер (по одному потоку на ядро). Теоретически, скорость выполнения такого алгоритма может быть увеличена во столько раз, сколько запущено параллельных вычислительных потоков.

Но в реальности чаще всего это не совсем так. Помимо вычислительного устройства, предоставляемого рабочему потоку в виде вычислительного ядра, практически всегда алгоритму требуются и другие ресурсы, такие как оперативная память, жёсткий диск и прочее. Они-то и могут стать «бутылочным горлышком», ограничивающим скорость выполнения алгоритма в целом. Более того, так как разные рабочие потоки начинают конкурировать между собой в борьбе за дефицитный ресурс, то могут возникать различные коллизии, на разрешение которых также тратится время, и скорость выполнения алгоритма в целом снижается ещё более. Например, при одновременной интенсивной работе с жёстким диском из нескольких параллельно выполняющихся вычислительных потоков жёсткий диск вынужден постоянно перемещать магнитные головки, удовлетворяя запросы из различных потоков – очевидно, относящихся к разным физическим местам

жёсткого диска – вместо того, чтобы осуществлять линейное чтение/запись. Всё это в итоге может приводить к тому, что увеличение количества вычислительных потоков алгоритма, несмотря на имеющиеся свободные вычислительные ядра, даёт в результате не ожидаемый и желанный эффект – увеличение скорости работы алгоритма в целом – а наоборот, ведёт к её деградации.

Отдельного упоминания в данном аспекте заслуживает тема виртуальных вычислительных ядер – гипертрединг. При его использовании одно физическое вычислительное ядро на аппаратном уровне представляется двумя виртуальными. За счёт различных оптимизаций в архитектуре процессора такая виртуализация на некоторых задачах даёт выигрыш до 10%. Но на некоторых задачах никакого выигрыша нет. И, при этом, возвращаясь к теме конкуренции рабочих потоков алгоритма за другие ресурсы, отметим, что количество таких рабочих потоков, если придерживаться распределения по одному потоку на ядро, возрастает в 2 раза. Соответственно, возрастает и конкуренция за дефицитный ресурс.

Так как же максимально эффективно задействовать имеющиеся аппаратные ресурсы вычислительными рабочими потоками алгоритма, чтобы общее время его выполнения было минимальным? Для автоматического решения этой задачи предлагается следующий механизм динамической балансировки вычислительной нагрузки путём изменения количества выполняемых параллельных потоков алгоритма.

Изначально вычислительных рабочих потоков создаётся столько, сколько в системе имеется виртуальных вычислительных ядер. Очевидно, что создавать вычислительных потоков больше, чем имеющееся количество виртуальных вычислительных ядер, нет никакого смысла. Алгоритм может уменьшить количество создаваемых потоков, если он понимает, что в силу специфики решаемой задачи большее число рабочих потоков ему просто не нужно. Но ни сам алгоритм, ни среда динамического анализа программ не могут делать предположения о том, при каком количестве рабочих потоков какой-либо ресурс может стать «бутылочным горлышком» и привести к конкуренции рабочих потоков и борьбе за него. Это сильно зависит как от аппаратных характеристик вычислительной системы, на которой выполняется программа, так и от специфики обрабатываемых данных. Выход один – динамически ограничивать число рабочих потоков алгоритма при возникновении конкуренции рабочих потоков и борьбе за какой-либо ресурс.

Самый первый вопрос, который возникает при создании механизма динамической балансировки количества параллельных потоков алгоритма, – это как определить, что началась конкуренция и борьба за ресурс? Предлагается использовать следующие утверждения для решения данной задачи. Каждый рабочий поток, выполняемый на выделенном ему виртуальном вычислительном ядре, должен потреблять процессорное время. В идеале – 100%, что означало бы, что рабочий поток полностью занят вычислениями и

максимально полно использует предоставленное ему вычислительное ядро. Если же потребление процессорного времени рабочим потоком менее 100%, то это означает, что оставшуюся часть времени поток проводит в ожидании каких-то событий. Это может быть синхронизация с другими потоками или ожидание готовности каких-то ресурсов, получение данных от них. Для типичного параллельного алгоритма такие накладные расходы чаще всего неизбежны, поэтому их наличие само по себе не является признаком того, что что-то идёт не так. Но если оказывается, что рабочий поток начинает проводить в состоянии ожидания времени больше или сравнимо со временем активной работы – это и есть верный признак того, что началась конкуренция в борьбе за какой-то ресурс.

Именно мониторинг использования рабочими потоками процессорного времени и лег в основу механизма динамической балансировки количества параллельных потоков алгоритма. Этот механизм встроен в инфраструктуру среды динамического анализа программ и, через небольшое время после запуска нескольких рабочих потоков алгоритма, начинает собирать и периодически анализировать статистику использования процессорного времени наблюдаемым набором рабочих потоков. В случае обнаружения конкуренции в борьбе за какой-то ресурс, определяемой по среднему значению использования процессорного времени каждым потоком, среда приостанавливает выполнение одного из потоков и продолжает наблюдение. Если при следующем анализе статистики выясняется, что конкуренция всё равно присутствует, то приостанавливается ещё один поток. И так далее, пока не будет устранена конкуренция или пока не останется всего лишь 2 потока. Если же при очередном анализе статистики выясняется, что среднее потребление процессорного времени существенно выросло, то среда возобновляет выполнение одного из приостановленных потоков.

Границы значений потребления процессорного времени, при преодолении которых среда делает заключение о конкуренции в борьбе за какой-то ресурс и об улучшении обстановки, подбирались экспериментально. В результате загрузку процессора в среднем каждым работающим потоком менее 40% было принято решение считать за признак конкуренции, а загрузку процессора более 50% – признаком её отсутствия. Именно эти значения стали использоваться по умолчанию, но, так как эти пороги могут различаться для различных аппаратных систем, пользователю предоставлена возможность изменять эти значения.

Отдельно стоит коснуться темы выбора рабочего потока для приостановки его выполнения в случае обнаружении конкуренции потоков и для возобновления работы в случае улучшения обстановки. Очевидно, что если алгоритм разделил весь объём обрабатываемых данных на равные части между рабочими потоками и небольшая часть этих рабочих потоков будет приостановлена для устранения возникшей конкуренции, а основная часть продолжит работу, то в результате эта приостановленная часть потоков начнёт

выполняться только после окончания работы основной части. Как следствие, во время работы этой изначально приостановленной части потоков система будет нагружена не полностью, а время работы всего алгоритма будет приближаться к удвоенному времени работы одного потока. А нужно было всего лишь немного снизить нагрузку на систему для достижения приемлемого уровня конкуренции потоков.

Для решения этой проблемы был разработан метод распределения рабочего состояния между рабочими потоками. Были организованы две очереди рабочих потоков – одна на приостановку, другая на запуск. Изначально все потоки помещаются в очередь на приостановку, а очередь на запуск пуста. При обнаружении конкуренции потоков среда выбирает для приостановки первый поток из очереди на приостановку, приостанавливает его и помещает в конец очереди на запуск. И симметрично, при обнаружении снижения конкуренции, выбирается первый поток из очереди на запуск, запускается и помещается в конец очереди на приостановку. А в случае, если после анализа состояния принимается решение ничего не менять и имеются приостановленные потоки, то выполняется и приостановка одного потока, и запуск другого по описанному выше алгоритму. Таким образом, осуществляется ротация рабочих потоков и обеспечивается равномерность их выполнения.

Но, как оказалось, не всем алгоритмам нужна такая ротация. Это происходит, например, в случае реализации сугубо последовательного алгоритма методом 2. Если у такого алгоритма есть какая-то обрабатываемая сущность (или несколько, но их меньше, чем рабочих потоков) с превалирующим объёмом данных, то алгоритм в первую очередь заинтересован в выполнении тех рабочих потоков, которые обрабатывают такие сущности. Поэтому ротация рабочих потоков для таких алгоритмов не нужна – приостанавливать надо в первую очередь более легковесные рабочие потоки, а запускать – более тяжеловесные.

Поэтому для таких алгоритмов был создан второй метод распределения рабочего состояния между рабочими потоками. Потоки ставятся в очередь на приостановку в порядке уменьшения их веса. И в случае, когда один из них необходимо приостановить, он выбирается не с начала очереди, а с конца и ставится в конец очереди на запуск. Запускаются потоки также не с начала очереди на запуск, а с её конца. Фактически, очереди начинают работать не как очереди, а как стеки. Никакой ротации потоков не производится.

Выбор использования того или иного метода распределения рабочего состояния между рабочими потоками (ротация или стек) осуществляется алгоритмом перед созданием рабочих потоков.

5. Методы балансировки вычислительной нагрузки при распараллеливании реализаций сугубо последовательных алгоритмов

В отличие от простого распараллеливания по данным, когда весь объём данных делится на равные части и распределяется по имеющимся вычислителям, при использовании 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. Балансировка вычислительной нагрузки может производиться как статически, путём первоначального распределения данных, так и динамически – во время выполнения реализации алгоритма, путём перераспределения части данных и/или путём 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.

Статическая балансировка с весами

При использовании метода 1 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 ещё 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 завершён 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 завешён 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.

Помимо изначальной статической балансировки вычислительной нагрузки к реализациям, выполненных по методу 1, также применяется описанная в предыдущей главе динамическая балансировка данных в режиме ротации рабочих потоков.

Статическая балансировка упорядочиванием обрабатываемых данных

При использовании метода 2 балансировка вычислительной нагрузки также производится статически. Для начала надо оценить объём обрабатываемых данных в каждой независимой сущности и упорядочить сущности по уменьшению этого объёма. Именно в этом порядке сущности будут выдаваться вычислителям для обработки: сначала те, которые имеют больший объём данных, затем – меньший. Вычислительный процесс берёт очередную сущность с головы очереди, обрабатывает её, и берёт следующую. Сущности с наибольшими объёмами данных таким образом будут обработаны в первую очередь, а конце работы останется набор сущностей с наименьшими объёмами 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 объёма 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), объём данных которой (которых) имеет в общем объёме данных превалирующее значение, то необходимо ограничить число вычислительных процессов. Так как всё равно при таком неравномерном распределении данных общее время работы алгоритма будет определяться временем работы вычислительного процесса с сущностью с наибольшим объёмом 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 и, таким образом, увеличит скорость работы вычислительного процесса с сущностью с наибольшим объёмом данных.

Для оценки количества необходимых вычислительных процессов предлагается использовать следующую формулу. Пусть имеется n сущностей, $n > 1$, и $p_i, i = 1..n$, – объём 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.

Помимо изначальной статической балансировки вычислительной нагрузки к реализациям, выполненных по методу 2, также применяется описанная в предыдущей главе динамическая балансировка данных в режиме стека рабочих потоков.

Динамическая балансировка очереди заданий

При использовании метода 3 балансировка вычислительной нагрузки производится динамически, путём 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 своё предыдущее задание, обращается к очереди за очередным заданием. Ему оформляется очередной, пока ещё не выданный на обработку кусок данных в виде задания, это задание помещается в конец очереди и выдаётся 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 ещё разбирать подготовленные ему данные. Помимо излишнего потребления оперативной памяти это также приведёт к тому, что, пока все рабочие процессы второго типа работали, они конкурировали, и 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 ещё ведётся работа другими процессами второго типа) находится в очереди. Если их меньше установленного максимального размера очереди, очередное задание выдаётся 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 выдаётся 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 счёт 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 трёхкратное 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%.

Описанная в предыдущей главе автоматическая динамическая балансировка данных в этом случае применяться не должна.

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 объёму 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 начнёт уменьшаться.

6. Заключение

Использование в программной реализации сугубо последовательного алгоритма не означает неизбежность его последовательного выполнения. Предложенные в статье методы распараллеливания реализаций таких алгоритмов и балансировки вычислительной нагрузки на общей памяти могут поспособствовать созданию эффективных параллельных программ, полностью использующих аппаратные возможности современных вычислительных систем с общей памятью.

Работа выполнена при финансовой поддержке гранта РФФИ № 18-01-00131-а.

Литература

1. Падарян В.А., Гетьман А.И., Соловьев М.А., Бакулин М.Г., Борзилов А.И., Каушан В.В., Ледовских И.Н., Маркин Ю.В., Панасенко С.С. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода // Труды ИСП РАН. — М.: ИСП РАН, 2014. — Т 26. — вып. 1. — С. 251-276. — ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). — doi: 10.15514/ISPRAS-2014-26(1)-8.
2. Падарян В.А. О представлении результатов обратной инженерии бинарного кода // Труды ИСП РАН. — М.: ИСП РАН, 2017. — Т 29. — вып. 3. — С. 31-42. — ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). — 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.
4. Бугеря А.Б., Ким Е.С., Соловьев М.А. Распараллеливание реализаций сугубо последовательных алгоритмов // Труды ИСП РАН. — М.: ИСП РАН, 2018. — Т 30. — вып. 2. — С. 25-44. — ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). — doi: 10.15514/ISPRAS-2018-30(2)-2. — URL: http://www.ispras.ru/proceedings/isp_30_2018_2.

References

1. Padarian V.A., Getman A.I., Solovev M.A., Bakulin M.G., Borzilov A.I., Kaushan V.V., Ledovskikh I.N., Markin Iu.V., Panasenko S.S. Metody i programmnye sredstva, podderzhivaiushchie kombinirovannyi analiz binarnogo koda // Trudy ISP RAN. — M.: ISP RAN, 2014. —vyp. 1. — S. 251-276. — ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). — doi: 10.15514/ISPRAS-2014-26(1)-8.
2. Padarian V.A. O predstavlenii rezultatov obratnoi inzhenerii binarnogo koda // Trudy ISP RAN. — M.: ISP RAN, 2017. — T 29. — vyp. 3. — S. 31-42. — ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). — 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.
4. Bugerya A.B., Kim E.S., Solovev M.A. Rasparallelivanie realizatsii sugubo posledovatelnykh algoritmov // Trudy ISP RAN. — M.: ISP RAN, 2018. — T 30. — vyp. 2. — S. 25-44. — ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). — doi: 10.15514/ISPRAS-2018-30(2)-2. — URL: http://www.ispras.ru/proceedings/isp_30_2018_2.