О МЕТОДАХ РАЗРАБОТКИ ПАРАЛЛЕЛЬНЫХ КОМБИНАТОРНЫХ АЛГОРИТМОВ

Н.Е. Тимошевская

Томский государственный университет

Введение

Под *комбинаторными задачами* здесь подразумеваются перечислительные и поисковые задачи на конечных *комбинаторных множествах*, элементами которых служат *комбинаторные объекты*, представляющие собой комбинации элементов других конечных (возможно, тоже комбинаторных) множеств – перестановки, размещения, разбиения, подмножества, и т.п. *Перечислительная задача* заключается в перечислении всех элементов некоторого подмножества конечного комбинаторного множества, а *поисковая* – в нахождении в последнем элемента с заданными свойствами. Алгоритмы решения комбинаторных задач называются *комбинаторными алгоритмами*.

То обстоятельство, что решение комбинаторной задачи подразумевает вычисления в пределах конечного множества, делает методы и алгоритмы непрерывной математики фактически не применимыми к ней. За редким исключением перебор и опробование элементов основного комбинаторного множества является единственным способом решения комбинаторной задачи. Различные методы построения комбинаторных алгоритмов нацелены на сокращение и ускорение перебора и отличаются друг от друга тем, как они это делают. Почти все комбинаторные алгоритмы имеют экспоненциальную сложность. Вместе с тем на практике необходимо уметь решать комбинаторные задачи достаточно больших размерностей.

Параллельные вычисления сегодня широко используются при решении различных задач, требующих больших вычислительных ресурсов [1, 2]. К таковым, несомненно, относятся и комбинаторные задачи. Один из способов уменьшения времени решения комбинаторной задачи состоит в том, чтобы разбить комбинаторное множество на классы и исследовать эти классы одновременно, т.е. параллельно, на нескольких вычислительных устройствах. В данной работе будут рассмотрены методы разработки параллельных алгоритмов трех классов – перечислительных, использующих дерево поиска и динамического программирования. Обзор составлен по результатам зарубежных публикаций и исследований автора. Среди алгоритмов, использующих дерево поиска, особое внимание уделяется способам организации параллельного обхода дерева в глубину, отмечаются особенности, возникающие при распараллеливании алгоритмов поиска опти-

мальных решений, и отдельно рассматриваются алгоритмы ветвей и границ.

1. Параллельные перечислительные алгоритмы

В алгоритмах перечисления комбинаторных объектов, как правило, объекты порождаются в некотором порядке путем преобразования текущего объекта в следующий за ним по определенным правилам [3, 4]. Для распараллеливания данной схемы множество перечисляемых объектов следует предварительно разбить на классы и перечислять объекты в различных классах независимо и параллельно. Однако когда речь идет о множестве комбинаторных объектов, в отличие, скажем, от числового множества, провести такое разбиение в явном виде не представляется возможным.

В работах [5, 6] для решения этой проблемы предлагается метод нумерации, суть которого заключается в сопоставлении перечисляемым комбинаторным объектам их номеров из натурального ряда в соответствии с установленным на множестве объектов линейным порядком. Последний, естественно, должен допускать восстановление объекта по его номеру, причем время, требуемое на восстановление, должно быть пренебрежимо малым по сравнению со временем, затрачиваемым на перечисление всех объектов одного класса.

Пусть, таким образом, на множестве перечисляемых комбинаторных объектов задан некоторый линейный порядок и имеются подходящие алгоритм восстановления объекта по его номеру и алгоритм перечисления в заданном порядке требуемого количества объектов, начиная с некоторого. Динамическую балансировку нагрузки процессоров можно обеспечить, разбив все множество объектов на большое число классов сравнительно малой мощности и выдавая полученные классы на обработку в ответ на запросы освободившихся процессоров. Этот подход может использоваться для организации распределенных вычислений в большой вычислительной сети.

Если же используемая вычислительная система не изменяется в динамике, то можно использовать статическую балансировку загрузки процессоров, разбив все множество объектов на классы, отвечающие по мощности производительности процессоров в системе, например, следующим образом. Пусть имеется система из s процессоров с числовыми характеристиками их быстродействия $w_1 \geq w_2 \geq \ldots \geq w_s$ соответственно. Если z − число всех перечисляемых объектов и $w = \sum_{j=1}^{s} w_j$, то число z_j объектов, перечисляемых j-м процессором, и номер I_j первого из них вычисляются как $z_j = \left[\frac{w_j}{w}\right] \cdot z$ для $j = 1, \ldots, s-1$, $z_s = z - \sum_{j=1}^{s-1} z_j$ и $I_j = \sum_{k=1}^{j-1} z_k + 1$ для всех $j = 1, \ldots, s$.

Возможны различные пути организации параллельной работы процессоров. Первый вариант следующий. Для каждого $j = 1, \ldots, s$ один из процессоров по заданным $w_1, w_2, \ldots, w_s, z$ вычисляет значения I_j, z_j и посылает их j-му процессору. Затем на j-м процессоре с помощью алгоритма восстановления строится объект с номером I_j и запускается алгоритм, перечисляющий z_j следующих за ним объектов. Во втором варианте каждому процессору передается уже восстановленный объект, и на процессоре работает только алгоритм перечисления. В третьем – значения I_j и z_j не передаются, но вычисляются j-м процессором по известным всем числам $w_1, w_2, \ldots, w_s, z$.

В работе [6] представлены параллельные алгоритмы перечисления перестановок и сочетаний методом нумерации. Установленная в эксперименте эффективность этих алгоритмов в среднем равна 0,94 и 0,87 соответственно. Алгоритм параллельного перечисления разбиений множества методом нумерации также имеет высокую эффективность (порядка 0,96).

Следует отметить, что для достижения высокой эффективности при использовании статической балансировки необходимо, чтобы в течение всего времени счета параллельной программы установленное соотношение производительностей процессоров сохранялось неизменным, что на практике не всегда имеет место (в частности, когда система обслуживает многих пользователей). Заметим также, что применение данного метода ограничено кругом задач, допускающих возможность нумерации требуемых в задаче объектов так, что по номеру можно легко восстановить сопоставленный ему объект.

2. Параллельные алгоритмы, использующие обход дерева поиска

Благодаря комбинационной структуре объектов, во многих комбинаторных задачах в рассмотрение вводится *дерево поиска*, которое используется в качестве модели пространства возможных решений, а его обход – как модель алгоритма поиска решения. Дерево поиска может иметь настолько большие размеры, что его невозможно представить в компьютере целиком. По этой причине дерево обрабатывают поэлементно. На однопроцессорном компьютере обход дерева выполняется последовательно – узлы дерева воспроизводятся один за другим по одному. В зависимости от порядка воспроизведения узлов различают обходы в глубину, в ширину и в порядке «наилучшего выбора».

2.1. Методы параллельного обхода дерева в глубину

При обходе в глубину в каждом поддереве дерева сначала проходится корень, а затем его поддеревья.

Идея распараллеливания процедуры обхода дерева, прослеживаемая почти во всех работах, состоит в том, что дерево поиска разбивается на

множества поддеревьев и их обход поручается разным процессорам. Проблема заключается в том, как разбить (статически или динамически) дерево так, чтобы обеспечить высокую эффективность распараллеливания, на снижение которой влияют многие факторы, в том числе неравномерная загрузка процессоров системы, накладные расходы на коммуникации, конкуренция процессоров за доступ к разделяемым данным, избыточность просматриваемой области поиска.

Статический подход к распределению поддеревьев по процессорам эффективен только для деревьев с предсказуемой структурой. В работе [8] этот подход используется для обхода в глубину дерева поиска разбиений. Поддеревья распределяются по процессорам в соответствии с их номерами. Экспериментально показано, что при правильном выборе яруса, содержащего корневые вершины распределяемых поддеревьев, удается достигнуть максимального ускорения.

В большинстве же комбинаторных задач дерево поиска носит нерегулярный, непредсказуемый характер, и для обеспечения хорошей эффективности распараллеливания его обхода требуется выполнение динамических процедур выделения и распределения поддеревьев между процессорами. Если какой-либо из процессов, а каждый процесс исполняется на отдельном процессоре, завершает обход назначенных ему поддеревьев раньше других, то он становится *свободным* и должен получить часть работы какого-либо занятого процесса, называемого в данном случае *источником*.

В [7] дается классификация методов обхода по следующим правилам распределения поддеревьев: 1) разбиение и передача инициируются источником [9, 10]; 2) разбиение инициируется источником, а передача выполняется по запросу свободного процесса [11, 12]; 3) разбиение и передача выполняются по запросу свободного процесса [13–17].

В первой группе методов процессы, выступающие источниками, генерируют и рассылают корни поддеревьев независимо от того, в каком состоянии находятся другие процессы. Если обход дерева сопровождается выполнением в вершине большого объема вычислений, то, наверное, оправданным будет подход, описанный в работе [10], где корни поддеревьев источник рассылает случайно выбранным процессам при каждом ветвлении, что в некоторой степени обеспечивает равномерную загрузку процессоров. Однако в такой схеме коммуникации сильно перегружают сеть и число вершин на каждом процессоре может расти неограниченно, что приводит к ограничениям по памяти.

В работе [9] представлен вариант этой схемы для вычислительных систем с архитектурой butterfly или гиперкуба, в котором порождаемые вершины бинарного дерева распределяются по соседним процессорам. Авторами метода показывается, что с высокой степенью вероятности время,

требуемое для параллельного обхода дерева с M вершинами на P процессорах, равно $O(M/P + \log P)$. Эта схема локализует все пересылки и поэтому оказывается более эффективной по сравнению с представленной в [10]. Введение ограничения, запрещающего передавать вершины, расположенные ниже заданного *граничного* яруса, позволяет повысить эффективность описанных методов [18]. Все вершины, расположенные ниже граничного яруса, проходятся породившим их процессом, что приводит к снижению объема коммуникаций.

Во второй группе методов корневые вершины поддеревьев (поддеревья) генерируются источниками независимо, но пересылаются свободным процессам, только по запросу от них. В работе [12] представлен метод *назначаемых поддеревьев*, в котором генерирование корневых вершин возложено на отдельно выделенный процесс, называемый *управляющим*, а все остальные процессы называются *рабочими*. Управляющий процесс вначале генерирует большое число корневых вершин поддеревьев, затем назначает каждому процессу по одному поддереву, после обхода которого рабочий процесс получает следующую корневую вершину и т.д. В работе [11] для генерирования корневых вершин используется несколько процессов-источников. Каждый рабочий процесс приписан к одному из них. Если у источника заканчиваются поддеревья, то его рабочие процессы приписываются к другому источнику. Такой подход позволяет избежать перегрузки управляющего процесса. В [19] предлагается метод *fixed-packet DFS*, в котором на первом этапе все процессы выполняют обход дерева в ширину до некоторого яруса. Множество вершин последнего яруса разбивается на классы по числу процессов. На втором этапе каждый процесс начинает обход в глубину поддеревьев, корневые вершины которых лежат в его классе. Когда процесс освобождается, он направляет кому-нибудь запрос. В ответ источник посылает освободившемуся процессу часть корневых вершин из своего класса. Этот метод успешно применяется в итеративном поиске.

К третьей (самой распространенной) группе относятся методы, в которых и разбиение, и передача поддеревьев выполняются по запросам от свободных процессов. Эти методы представлены в работах [12–14, 18, 20, 28, 29]. Опишем их общую схему. Каждый из процессов выполняет обход в глубину назначенных ему одного или нескольких поддеревьев, при этом периодически проверяет наличие запросов от других процессов. Когда процесс завершает обход назначенных ему поддеревьев, он становится свободным и выбирает для себя процесс, которому посылает запрос на работу. Процесс, получивший запрос, становится источником и либо посылает отказ, либо выделяет и посылает часть оставшихся у него поддеревьев свободному процессу.

Методы последней группы различаются способами выбора процесса-источника, выделения и начального распределения поддеревьев и условиями посылки отказа.

Описано несколько способов (стратегий) выбора процесса-источника, т.е. того, к кому направляется запрос свободного процесса. В стратегии *случайного голосования* (random polling) [14, 17, 21, 29] выбор осуществляется случайным образом. Если в ответ от выбранного процесса приходит отказ, то случайный выбор повторяется. Минусом этой стратегии является большое число отказов. В стратегии *глобальной кольцевой очереди* [22] имеется глобальный указатель на процесс, к которому надо направлять запрос. После обработки запроса указатель – переменная, содержащая номер процесса, переходит к следующему процессу (увеличивается по модулю числа процессов). В [22] показано, что использование глобального указателя позволяет значительно уменьшить число запросов для большого числа видов деревьев, однако при большом числе процессов возникает конкуренция за доступ к глобальному указателю. В стратегии *оптимального выбора* [12] запрос направляется к наиболее загруженному процессу.

При выполнении обхода в глубину каждым процессом используется стек, в который помещаются и затем по мере обхода извлекаются корневые вершины еще не пройденных поддеревьев. Таким образом, можно сказать, что в каждый момент времени процесс имеет набор поддеревьев, корневые вершины которых находятся в стеке. В ответ на запрос освободившегося процесса процесс-источник должен выделить часть своих поддеревьев, чтобы передать их свободному процессу. В стратегии *node splitting* выделяется одно поддерево – путем исключения одной вершины из стека [12, 28, 29]. В стратегии *stack splitting* выделяется несколько поддеревьев – путем расщепления стека на два [18, 20, 22, 29]. Согласно второй стратегии свободный процесс получает стек вершин. Стек процесса-источника расщепляется так, чтобы в стеке свободного процесса и в стеке источника оказалось примерно одинаковое число вершин и чтобы в каждом стеке были вершины разных ярусов. Преимущество расщепления стека авторы этого метода видят в том, что оно должно приводить к равномерному распределению вычислений между свободным процессом и источником.

Начальное распределение поддеревьев между процессами можно организовать по-разному. В некоторых методах изначально все дерево назначается одному процессу, а остальные являются свободными. Затем дерево постепенно динамически распределяется между процессами [14, 17, 18]. Этот подход характеризуется большим числом коммуникаций уже на начальном этапе. В методе *выделяемых поддеревьев*, описанном в [12], один процесс вначале генерирует столько корневых вершин, сколько есть процессов, выполняя обход дерева в ширину. Полученные в результате обхода в ширину корневые вершины располагаются максимально близко к корню.

Условия посылки отказа определяются скорее не методом параллельного исполнения поиска, а задачей и алгоритмом, к которому он применяется. Одно из самых простых условий накладывается на число уже полно-

стью пройденных ярусов в дереве процесса-источника, а именно: если предположительное число оставшихся ярусов очень мало (насколько мало зависит от размерностей задачи и характеристик используемой системы), то высылается отказ.

С помощью описанных выше методов разными авторами построены параллельные алгоритмы решения таких задач, как задача коммивояжера, тур коня, игра в пятнадцать, проверка тавтологии, автоматическое генерирование тестов, задача о рюкзаке и др. Алгоритмы исследовались на многопроцессорных системах различных типов, и почти для всех задач тем или иным методом достигалось линейное ускорение.

2.2. Параллельный поиск оптимального решения

В задачах поиска оптимального (для определенности будем говорить минимального) решения используется некоторая целевая функция – характеристика решения. На ее основе, как правило, строится функция нижней оценки стоимости частичного решения. В каждой вершине дерева поиска проводится анализ, связанный с полученным на данный момент лучшим решением, по результатам которого продолжается ветвление дерева из рассматриваемой вершины либо выполняется возврат на предыдущий уровень.

При поиске оптимального решения параллельным обходом дерева в глубину необходимо информировать все процессы о находимых решениях. Методы поиска отличаются друг от друга тем, как они это делают [7, 13]. В системах с общей памятью можно хранить глобальное лучшее решение среди разделяемых данных, к которым все процессы имеют доступ. В ряде методов, ориентированных на системы, не имеющие общей памяти, текущее решение хранится в локальной памяти каждого процесса. Когда какой-либо из процессов находит решение лучше хранимого у него, он обновляет его у себя и передает всем остальным процессам, которые, если надо, обновляют свои решения. Заметим, что если текущее решение, хранящееся у процесса, хуже глобального текущего решения, то это лишь снижает эффективность поиска, но не влияет на корректность найденного в итоге решения. Для ряда задач и архитектур известны алгоритмы с линейным ускорением. Например, в работе [23] дается изложение параллельных алгоритмов и результатов исследований для задачи оптимальной компоновки схем.

2.3. Аномалии ускорения в параллельных алгоритмах обхода дерева поиска

Если решается задача поиска и достаточно найти хотя бы одно решение, то во всех методах параллельный поиск прекращается, как только какой-нибудь из процессов находит решение. В задачах такого типа наиболее

заметны эффекты несовпадения просматриваемых областей поиска для параллельного и последовательного алгоритмов и, как следствие, выполнения ими работы разного объема. В параллельном случае он может оказаться как меньше, так и больше. *Фактор избыточности поиска* R определяется как отношение объема работы (числа просмотренных вершин) последовательного алгоритма к объему работы параллельного. Тогда верхняя граница ускорения для p процессов есть pR. В некоторых случаях фактор избыточности может оказаться меньше единицы, что приводит к суперлинейному ускорению. Если фактор избыточности поиска в среднем меньше единицы, это означает, что решения в дереве поиска распределены неравномерно или что выбранный последовательный алгоритм не является лучшим для этой задачи.

В работе [16] проведен глубокий анализ этой проблемы. Предлагаются абстрактные модели пространства поиска, просматриваемого алгоритмами параллельного поиска в глубину двух типов: простого и с упорядочиванием порождаемых вершин. Анализируется и сравнивается среднее число узлов, просматриваемых последовательным и параллельным алгоритмами. Для простого поиска доказывается, что если распределение решений по областям поиска разных процессоров одинаково, то области поиска последовательного и параллельного алгоритмов примерно одинаковы, и если распределения решений по областям поиска разных процессоров различаются, то область поиска последовательного алгоритма больше области поиска параллельного. Для упорядоченного поиска доказывается, что число узлов, просматриваемых параллельным алгоритмом, не больше числа узлов, просматриваемых последовательным.

В работах [24, 25] представлен метод, в котором ветвление в вершинах выполняется по приоритетам так, что параллельный алгоритм посещает узлы дерева в том же порядке, что и последовательный. Такой подход позволяет значительно повысить ускорение при использовании большого числа процессов при решении достаточно больших задач. Однако необходимость слежения за приоритетами делает эту схему слабо масштабируемой.

2.4. Итеративный поиск в глубину (IDA)*

Итеративный последовательный поиск в глубину чаще применяется для решения задач оптимизации и бывает в двух вариантах [26]. В первом – с ограничением по глубине – на каждой итерации задается номер яруса, до которого выполняется поиск, в надежде, что длина пути, представляющего решение, меньше этого яруса. Этот вариант применяется тогда, когда оптимальное решение задается кратчайшим путем в дереве. От итерации к итерации граничный ярус увеличивается. Во втором варианте на каждой итерации увеличивается значение границы, т.е. если оценка

стоимости решения в рассматриваемой вершине превышает заданную границу, то ветвления из нее не производится.

Параллельное исполнение итеративного поиска встречается в литературе в двух версиях. В первой – различные процессы независимо работают на разных итерациях. Однако при решении задач оптимизации быть уверенным в оптимальности решения, найденного на некоторой итерации, можно только после того, как завершится поиск на всех итерациях. Во второй – на каждой итерации выполняется параллельный поиск в глубину с указанными ограничениями. После завершения итерации, если решение еще не найдено, то специально выделенный процесс задает новое ограничение и запускается следующая итерация [19].

2.5. Методы параллельного выполнения алгоритмов ветвей и границ

В алгоритмах ветвей и границ на каждом шаге для ветвления в дереве поиска выбирается наиболее перспективная из еще не обработанных вершин. Одним из подходов к распараллеливанию этих алгоритмов является хранение глобального списка вершин и выдача из него самой перспективной вершины запросившему процессу. После выполнения ветвления процесс оценивает полученные вершины и те из них, которые удовлетворяют граничным условиям, помещает в глобальный список. Методы такого типа относятся к централизованным [7, 13, 21, 27]. Они отличаются друг от друга только используемыми для хранения списка структурами данных. Для систем, не имеющих общей памяти, такой подход требует огромных расходов на передачи вершин после каждого ветвления.

Альтернативный подход заключается в разбиении дерева на поддеревья, которые независимо исследуются разными процессами. Каждый из процессов хранит собственный список необработанных вершин в его поддереве. Этот подход не требует больших коммуникационных расходов, но может привести к избыточности просматриваемой области поиска и несбалансированности загрузки процессов [35].

Встречаются методы, использующие композицию этих двух подходов. С некоторой периодичностью процессы составляют глобальный список вершин из своих локальных списков и выбирают наиболее перспективные вершины в качестве корневых для дальнейшего исследования. Чем чаще они обмениваются данными, тем меньше потерь, связанных с балансировкой и избыточностью области поиска, но больше накладных расходов на коммуникации, и наоборот. В работе [30] показано, что для некоторых алгоритмов ветвей и границ существует такое k, что одновременное ветвление более чем в k вершинах не дает большого преимущества. Такие значения экспериментальным путем были установлены для задачи о рюкзаке.

При исполнении параллельных алгоритмов ветвей и границ также наблюдаются аномалии ускорения [30, 31]. Одной из причин их возникновения является то, что многие вершины в дереве могут иметь одинаковую оценку, хотя большинство из них не лежит на пути решения. Следовательно, если параллельный алгоритм раньше начнет исследовать часть дерева, содержащую правильный путь, то будет получено суперлинейное ускорение, в противном случае – наоборот.

3. Параллельные алгоритмы динамического программирования

Метод динамического программирования состоит в сведении исходной задачи к решению некоторых ее подзадач меньшего размера и использовании табличной техники для сохранения уже найденных ответов. Решение подзадач при этом происходит в порядке возрастания их размеров — от меньших к большим до тех пор, пока не будет решена исходная задача. Некоторый обзор методов распараллеливания алгоритмов динамического программирования можно найти в работе [7].

Пусть последовательный алгоритм разбивает задачу на подзадачи нескольких уровней так, что решение подзадач одного уровня зависит только от решений подзадач предыдущих уровней. Тогда в самом простом параллельном исполнении на каждом уровне задачи распределяются по процессорам, которые решают их и рассылают остальным процессорам полученные решения для использования в решении задач следующего уровня [21, 33]. В результате каждому процессу известны решения всех подзадач текущего уровня. Если уровень содержит N подзадач, то каждому из p процессоров назначается по $[N/p]$ подзадач. В этом методе не может быть задействовано больше процессоров, чем максимальное число подзадач одного уровня.

Использование нескольких процессоров для вычисления стоимости каждой подзадачи иногда оказывается более эффективным [32]. В другой технике распараллеливания решение подзадач на разных уровнях ведется по конвейерному принципу. Для решения некоторой подзадачи не требуются решения всех подзадач предыдущих уровней, вместо этого ожидается решение только тех подзадач, от которых непосредственно зависит решение рассматриваемой. Эта техника используется для решения задачи о расстановке скобок в произведении матриц [34]. Последовательный алгоритм решения этой задачи имеет сложность $O(n^3)$. При использовании $O(n^2)$ процессоров вычисления выполняются за время $O(n)$.

Литература

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002. – 608 с.

2. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных машин. – Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2003. – 176 с.
3. Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы теория и практика. – М.: Мир, 1980. – 476 с.
4. Липский В. Комбинаторика для программистов. – М.: Мир, 1988. – 200 с.
5. Тимошевская Н.Е. Параллельная генерация сочетаний и перестановок // Вторая Сибирская школа-семинар по параллельным вычислениям. – Томск: Изд-во Том. ун-та. – 2004. – С. 55–59.
6. Тимошевская Н.Е. О нумерации перестановок и сочетаний для организации параллельных вычислений в задачах проектирования управляющих систем // Известия Томского политехнического университета – Томск: Изд-во Томского политехнического университета, 2004.
7. Grama A., Kumar V. A survey of parallel search algorithms for discrete optimization problems // ORSA Journal of Computing. − 1995. – Vol. 7(4). − P. 365–385.
8. Семенов В.В. Алгоритмы параллельного перечисления разбиений множества // Тезисы Третьей Сибирской школы-семинара по параллельным вычислениям. – Томск: Изд-во Том. ун-та, 2005. – С. 29–30.
9. Ranade A.G.. Optimal speedup for backtrack search on a butterfly network // Proceedings of the Third ACM Symposium on Parallel Algorithms and Architectures, 1991.
10. Wei Shu, Kale L.V. A dynamic scheduling strategy for the share-kernel system // Proceedings of SuperComputing Conference. − 1989. – P. 389–398.
11. Furuichi M., Taki K. and Ichiyoshi N. A multi-level load balancing scheme for OR-parallel exhaustive search programs on the Multi-PSI // Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. − 1990. – P. 50–59.
12. Тимошевская Н.Е. Параллельные методы обхода дерева // Математическое моделирование. − 2004. – Т. 16, № 1. – С. 105–114.
13. Grama A., Kumar V. Parallel processing for discrete optimization problem // Encyclopedia of microcomputers. − 1992. – P. 129–157.
14. Sanders P. Parallelizing NP-complete problems using tree shaped computations // Max-Planck-Institut fur Informatik, Im Stadtwald, Saarbrucken, Germany, 1999.
15. Gao Y., Marsland T.A. Multithreaded Pruned Tree Search in Distributed Systems // Computing Science Department University of Alberta Edmonton, Canada T6G 2H1.

16. Rao N.A., Kumar V. On the Efficiency of Parallel Backtracking // IEEE Trans. Parallel Distrib. Syst. − 1993. – Vol. 4(4). − P. 427–437.
17. Sanders P. Better Algorithms for Parallel Backtracking // Workshop on Algorithms for Irregularly Structured Problems, number 980 in LNCS. – 1995. – P. 333–347.
18. Rao V.N., Kumar V. Parallel depth-first search. Part I: Implementation // International Journal of Parallel Programming. − 1987. – Vol. 16(6). − P. 479–499.
19. Reinefeld A., Schnecke A. Work-Load Balancing in Highly Parallel Depth-First Search // Proceedings Scalable High Performance Computations Conference SHPCC'94. – 1994. – P. 773–780.
20. Kumar V., Rao V.N. Parallel depth-first search. Part II: Analysis // International Journal of Parallel Programming. − 1987. – Vol. 16(6). − P. 501–519.
21. Kryukova S. A. Parallel programming archetypes in combinatorics and optimization // Partial fulfillment of the Requirements for the Degree of master of Science. California Institute of Technology Computer Science department. – Pasadena, 1995.
22. Kumar V., Grama A., Rao V.N. Scalable load balancing techniques for parallel computers // Journal of Parallel and Distributed Computing. − 1994. – Vol. 22(1). − P. 60–79.
23. Arvindam S., Kumar V., Rao V.N. Floorplan optimization on multiprocessors // Proceedings of the 1989 International Conference on Computer Design. – 1989.
24. Saletore V., Kale L.V. Consistent linear speedup to a first solution in parallel state-space search // Proceedings of the 1990 National Conference on Artificial Intelligence. − 1990. – P. 227–233.
25. Sinha A.B., Kale L.V. A load balancing strategy for prioritized execution of tasks // Department of Computer Science University of Illinois Urbana, IL 61801. – 1992.
26. Korf R.E. Depth-first iterative-deepening: An optimal admissible tree search // Artificial Intelligence. − 1985. – Vol. 27. − P. 97–109.
27. Quinnand M.J., Deo N. An upper bound for the speedup of parallel branch-and-bound algorithms // BIT. − 1986. – Vol. 26, № 1.
28. Ferguson Ch., Korf R. Distributed tree search and its application to alpha-beta pruning // Proceedings of the 1988 National Conference on Artificial Intelligence, – 1988.
29. Finkel R.A., Manber U. DIB – a distributed implementation of backtracking // ACM Transactions of Programming Languages and Systems. − 1987. – Vol. 9, № 2. − P. 235–256.
30. Lai T.H., Sahni S. Anomalies in parallel branch and bound algorithms // Communications of the ACM. – 1984.

31. Li G.-J., Wah B.W.. Coping with anomalies in parallel branch-and-bound algorithms // IEEE Transactions on Computers. – 1986.
32. De Mello J.D., Calvet J.L., Garcia J.M. Vectorization and multitasking of dynamic programming in control: experiments on a CRAY-2 // Parallel Computing. − 1990. – Vol. 13. − P. 261–269.
33. Lee J., Shragowitz E., Sahni S. A hypercube algorithm for the 0/1 knapsack problem // Journal of Parallel and Distributed Computing. – 1988.
34. Ibarra O.H., Pong T.C., Sohn S.M. Parallel recognition and parsing on the hypercube // IEEE Transactions on Computers. − 1991. – Vol. 40(6). − P. 764–770.
35. Забиняко Г.И. Параллельный алгоритм целочисленной линейной оптимизации // Автометрия. − 2001. − № 6. − С. 107–114.