Экспериментальное исследование трех вариантов реализации метода неравномерных покрытий для многоядерных систем с общей памятью

А.Ю. Горчаков, М.А. Посыпкин, Ю.В. Ямченко

***Аннотация*—В данной работе рассматриваются три эффективные параллельные реализации метода неравномерных покрытий, предназначенные для вычислительных систем с общей памятью. Метод неравномерных покрытий является одним из наиболее известных детерминированных методов решения задач глобальной оптимизации, основанных на схеме ветвей и границ. Учитывая вычислительную сложность метода и широкое распространение высокопроизводительных многоядерных систем с общей памятью, особую актуальность приобретает разработка параллельных реализаций данного метода. В статье предлагается несколько подходов к распараллеливанию метода неравномерных покрытий. В настоящее время существует несколько стандартов создания многопоточных приложений. В работе рассматривается два таких стандарта: OpenMP 4.0 и C++ 17. Для синхронизации между потоками используется несколько режимов. В работе приводится описание алгоритмов и их программных реализаций. Экспериментальное исследование проводилось на тестовой задаче, приближенной к реальной – поиск минимума энергии молекулярного кластера. В качестве вычислительных платформ для проведения экспериментов использовались современные высокопроизводительные системы. Исследование показало, что для данного типа задач, производительность методов (по количеству итераций) находится примерно на одном уровне. Кроме этого было показано экспериментально, что усложнение алгоритма не всегда приводит к увеличению его эффективности.**

***Ключевые слова*—метод неравномерных покрытий, многоядерные системы, параллельный алгоритм, балансировка нагрузки.**

*Работа выполнена при финансовой поддержке программы президиума РАН №27 «Фундаментальные проблемы решения сложных практических задач с помощью суперкомпьютеров» и проекта РФФИ № 16-07-00458

А.Ю. Горчаков – старший научный сотрудник Вычислительного центра им. А.А. Дородницина Федерального исследовательского центра «Информатика и управление» Российской академии наук. andrgor12@gmail.com

М.А. Посыпкин – главный научный сотрудник Вычислительного центра им. А.А. Дородницина Федерального исследовательского центра «Информатика и управление» Российской академии наук. mposypkina@gmail.com

Ю.В. Ямченко – аспирант кафедры "Системы автоматизированного проектирования" Московского Государственного Технического Университета имени Н.Э. Баумана. yamchenko.y.v@yandex.com

I. Введение

Задачи глобальной оптимизации, возникающие на практике, относятся к классу вычислительно сложных задач и их решение требует привлечения высокопроизводительной вычислительной техники. В связи со стремительным развитием многоядерных архитектур приобретает актуальность задача разработки параллельных алгоритмов, ориентированных на подобные системы. Типичные конфигурации вычислительных систем с общей памятью позволяют запускать на исполнение десятки и сотни одновременно работающих вычислительных потоков. В качестве примера можно привести архитектуру IBM Power9, допускающую до 24 ядер на процессор, с технологией SMT4 (4 потока на ядро).

В качестве метода решения задач глобальной оптимизации в данной работе рассматривается метод неравномерных покрытий [1-3], основанный на схеме ветвей и границ. Особенностью данного метода является то, что метод не только находит решение задачи, но и доказывает его оптимальность ($\varepsilon-$оптимальность). При этом его информационный граф имеет несбалансированную древовидную структуру, не известную до начала выполнения параллельной программы. Это приводит к необходимости разработки методов управления распределением вычислительной нагрузки в процессе расчетов.

При исследовании эффективности параллельных алгоритмов необходимо обращать особое внимание на технику поведения численного эксперимента. В отличие от последовательных реализаций метода неравномерных покрытий, информационный граф для параллельных реализаций может существенно отличаться от запуска к запуску. Поэтому показателями эффективности и надежности метода является как

математическое ожидание, так и дисперсия времени работы метода, количества вычислений функции и других показателей эффективности метода.

II. Метод ветвей и границ

Рассмотрим задачу оптимизации, в которой требуется найти минимум функции на заданном множестве $f(x) \rightarrow \min, x \in X$.

На каждой итерации решения поставленной задачи методом ветвей и границ выполняется две основные операции: разбиение рассматриваемой задачи на несколько подзадач (ветвление) и отсечение неперспективных подзадач.

Схема метода ветвей и границ включает следующие шаги:

1. Инициализация параметров задачи;
2. Помещение элемента, включающего в себя все множество решений, в список подзадач, «кандидатов на ветвление»;
3. Процедура ветвления. Декомпозиция выбранных «подзадач-кандидатов» на более мелкие подзадачи по определенным правилам и помещение новых подзадач в список «кандидатов на ветвление» и удаление из этого списка подзадач, к которым была применена процедура ветвления;
4. Для каждой подзадачи из списка «кандидатов на ветвление» решаем оценочную задачу. По результатам решения этой задачи подзадача может быть исключена из рассмотрения, если было выполнено одно из следующих условий:

4.1. Подзадача не содержит допустимых решений;

4.2. При решении оценочной задачи было получено допустимое решение исходной задачи;

4.3. Подзадача может быть отсеяна по выбранному правилу отсева.

5. Если список «кандидатов на ветвление» пуст, то работа метода завершается. Иначе переходим к п.2.

Графически процесс решения задачи методом ветвей и границ можно представить в виде дерева ветвлений (рисунок 1). Вершины дерева – это подзадачи, которые получаются в результате операции ветвления, а дуги соединяют конкретную подзадачу с подзадачами, полученными из нее в результате применения к ней операции ветвления.

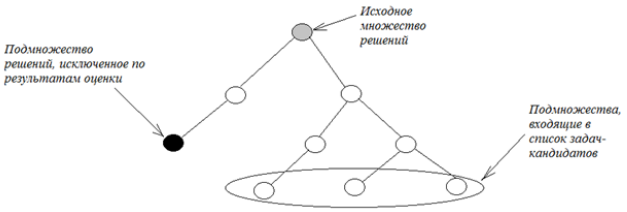

Рисунок 1. Процесс ветвления метода ветвей и границ

Алгоритмы процедур ветвления и вычисления оценок могут варьироваться в зависимости от специфики решаемой задачи.

Приведем наиболее известные стратегии ветвления.

1. Ветвление в глубину. Для ветвления выбирается подзадача из числа задач, сгенерированных на предыдущем шаге.
2. Для ветвления выбирается подзадача с наименьшим/наибольшим значением нижней/верхней оценки в задаче минимизации/максимизации.
3. Для ветвления выбирается задача, наименее удаленная от корня [4-6].

При распараллеливании метода ветвей и границ как на системах с общей памятью, так и на системах с распределенной памятью возникают две основные проблемы.

Первая проблема связана с тем фактом, что дерево ветвления не является сбалансированным. Вторая проблема заключается в том, что структура дерева ветвления формируется динамически и не известна до начала решения задачи.

В связи с наличием двух приведенных проблем возникает проблема распределения нагрузки между вычислительными блоками или проблема балансировки нагрузки.

III. Распараллеливание метода ветвей и границ на системах с общей памятью

Рассмотрим некоторые подходы к распараллеливанию метода ветвей и границ.

***Asynchronous Single Pool* (*ASP*)**

Данный подход предполагает наличие единых глобальных структур для каждого потока, из которых каждый поток будет брать материал для обработки, и в которую он будет записывать найденное значение. Обращение к таким структурам потоки вынуждены выполнять поочередно. Схематичное изображение подхода представлено на рисунке 2.

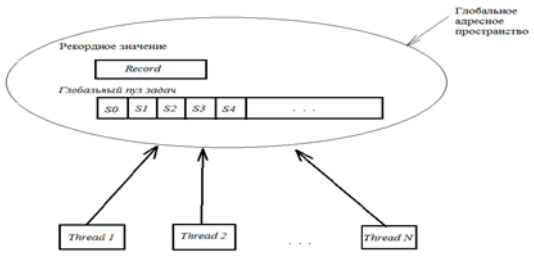

Рисунок 2. Asynchronous Single Pool (ASP)

Приведем основные принципы подхода ASP.

1. Рекордное значение f_* хранится в глобальной переменной.
2. Все подзадачи хранятся в глобальной структуре данных L.
3. Количество потоков равно числу ядер в системе.
4. Каждый поток разбивает свою задачу на подзадачи и помещает их в структуру L.
5. При попытке взять задачу для разбиения из пустой структуры, поток блокируется.
6. Если все потоки заблокированы и $L \in \emptyset$, то потоки удаляются и алгоритм завершает работу.

Основное замедление работы параллельного алгоритма при реализации на основе ASP происходит

из-за блокировок, возникающих при обновлении каждым потоком данных в списке L.

Asynchronous Multiple Pool (AMP)

Подход *AMP* предполагает создание локального пула решаемых задач для каждого потока. В глобальной области собираются достигнутые рекорды и контролируется информация о созданных потоках. Схематичное изображение подхода представлено на рисунке 3.

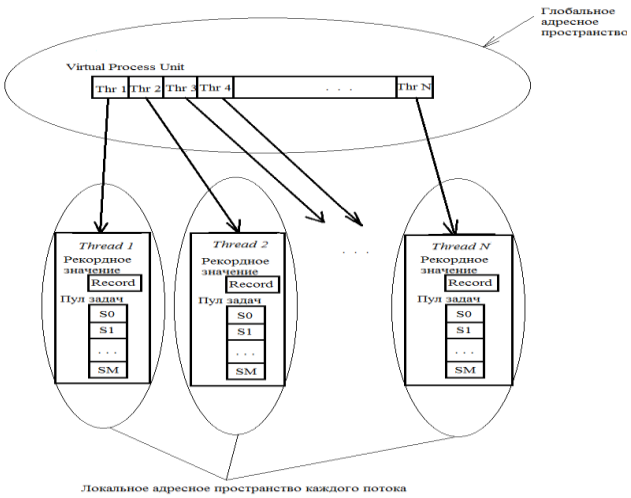

Рисунок 3. *Asynchronous Multiple Pool* (*AMP*)

Основные принципы подхода *AMP*:

1. Рекордное значение f_r хранится в глобальной переменной.

2. *Nth* – глобальная переменная, равная числу запущенных потоков. Алгоритм старается поддерживать *Nth* равной числу доступных ядер системы.

3. *VPU*(*Virtual Process Unit*) – глобальный вектор, в котором хранятся данные о всех потоках, достигнутые ими рекорды и списки задач, обрабатываемые каждым из потоков $L_i, i \in 1, \dots, Nth$.

4. Алгоритм начинает работу с создания единственного потока. Идентификатор потока помещается в элемент списка $VPU[1]$, при этом $L_1 = \{S_0\}, Nth = 1$, где S_0 - исходная задача.

5. Если в рабочем списке одного из потоков заканчиваются подзадачи, то этот поток освобождает соответствующую ему ячейку в *VPU,* уменьшает переменную $Nth = Nth - 1$ и завершает работу. Результат его работы сохраняется в глобальном списке.

6. На каждой итерации каждый поток проверяет условие *NTh* < *p*. Если условие выполняется и у текущего потока имеется более двух задач в рабочем списке, то он инициирует создание нового потока и отдает ему половину своих подзадач. При этом инкрементируется переменная $Nth = Nth + 1$.

7. Алгоритм завершает работу, если все потоки завершили свою работу, то есть при *NTh* = 0.

Подход AMP основан на идее сокращения конфликтов между потоками, посредством создания локальной рабочей области для каждого потока.

***Manager Pool* (*MP*)**

Часто при реализации параллельного варианта МВГ на системах с общей памятью один поток назначается потоком-менеджером, остальные потоки называют рабочими потоками. Поток-менеджер управляет всей логикой пересылки задач между потоками, синхронизацией данных, выделением памяти и созданием потоков [6]. Схематичное изображение подхода представлено на Рисунке 4.

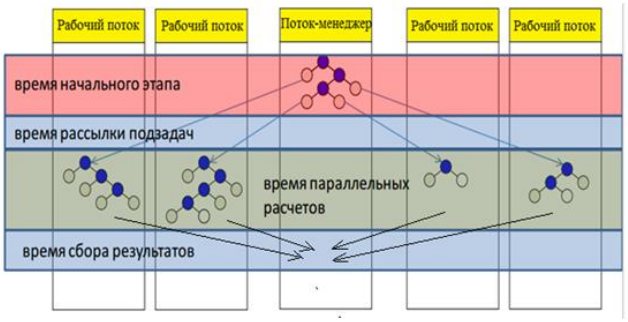

Рисунок 4. Manager Pool (MP)

Приведем типичную схему подхода MP.

Введем следующие вспомогательные обозначения:

1. W_b – пороговое число разбиений на рабочем потоке;

2. W_s – пороговое значение числа пересылаемых потоку-менеджеру задач;

3. U_m, L_m – максимальное и минимальное число подзадач, обрабатываемых потоком-менеджером;

С учетом введенных обозначений типичная схема подхода MP имеет следующий вид:

1. Каждый рабочий поток получает задачу на обработку и выполняет W_b разбиений. После этого он посылает не более W_s задач потоку-менеджеру;

2. Если у рабочего потока не остается ни одной задачи для обработки, то он запрашивает одну задачу у потока-менеджера;

3. Если в списке задач потока-менеджера накапливается более U_m подзадач, то он сигнализирует остальным потокам прекратить отправку ему задач;

4. Если в списке задач потока-менеджера находится менее L_m подзадач, то он сигнализирует остальным потокам возобновить отправку ему задач.

Основная трудность реализации данного подхода заключается в определении механизма оповещения рабочих потоков о наступлении определенного события. Кроме того, необходимо подобрать параметры W_b, W_s, U_m, L_m так, чтобы минимизировать потери на пересылки задач.

На основании обзора существующих подходов к распараллеливанию метода ветвей и границ, приведенного выше, было предложено три подхода к реализации параллельного алгоритма МВГ на системах с общей памятью: Simple BNB, BNBOmp и Manager-Slave BNB.

IV. Описание разработанных алгоритмов

Алгоритм *Simple BNB*

Идея алгоритма *Simple BNB* проста и заключается в том, чтобы создать максимально возможное число потоков, не превышающее значение *Thread_max*, и предоставить операционной системе возможность осуществлять дальнейшую балансировку загрузки системы.

Подход к реализации данного алгоритма схож с походом *AMP*: каждый поток имеет локальный пул подзадач, который он обрабатывает независимо от других потоков.

Работа алгоритма начинается с инициализации параметров: *Tread_max* - максимальное число потоков, которые могут быть запущены алгоритмом одновременно и *Step_max* - максимально возможное число итераций алгоритма; параметр *Steps_limit* означает минимальное число итераций, при котором имеет смысл разделение пула задач между несколькими потоками; структура *benchmark* содержит всю необходимую информацию о решаемой задаче, целевой функции и ограничениях; переменная *origin_sub* аккумулирует в себе информацию об исходной подзадаче (рассматриваемой области поиска); переменная *origin_state* хранит в себе информацию о состоянии процесса решения в начальный момент времени; *eps* – требуемая точность решения.

Затем вызывается функция *Solve*, в качестве параметров этой функции передается текущее состояние решаемой задачи (рекордное значение целевой функции, пул подзадач для решения, количество доступных итераций и потоков), *benchmark* и *eps* (в ходе решения конкретной задачи структура *benchmark* и значение *eps* остаются неизменными с момента инициализации). Функция *Solve* начинается с проверки условий эффективного распараллеливания: количество доступных потоков должно быть больше двух; количество доступных итераций не должно быть меньше некоторого наперед заданного значения *Steps_limit*. Далее алгоритм переходит на стадию *Presolving*, на которой выполняется несколько итераций алгоритма, при условии, что количество подзадач в пуле задач меньше двух до момента, когда это число станет равным двум. Отметим, что на этой стадии алгоритм может завершить выполнение при исчерпании подзадач в пуле.

Если число итераций, оставшихся после стадии *Presolving* больше значения *Steps_limit*, алгоритм делит текущее состояние *state* на два состояния $state_1, state_2$ с количеством подзадач равным $state.sub\_bag.size / 2$. На следующем шаге алгоритм создает два потока выполнения $thread_1, thread_2$, каждый из которых, выполняет функцию *Solve* с новым параметром состояния. Текущий поток выполнения при этом переходит в состояния ожидания завершения выполнения потоков $thread_1, thread_2$. После этого результаты решения подзадач из потоков $thread_1, thread_2$, которые хранятся в переменных состояния $state_1, state_2$ соответственно, объединяются в переменной *state* и возвращаются в вызывающую функцию.

Если число итераций после стадии *Presolving* оказалось меньше или равно значению *Steps_limit,* то из текущего состояния *state* запускается последовательный вариант алгоритма.

Главным достоинством данного алгоритма является то, что он не содержит разделяемых переменных, что исключает задержки, связанные с блокировкой потоков при одновременном обращении к общим ячейкам памяти.

К недостаткам данного алгоритма можно отнести отсутствие контроля над загрузкой процессоров и, как следствие, невозможность применения алгоритмов балансировки загрузки. Поэтому для полноценной загрузки системы предлагается выставлять значение параметра *Thread_max* заведомо большим, чем максимально возможного количества независимо выполняющихся потоков в системе. Однако при таком подходе увеличивается нагрузка на планировщик операционной системы.

Алгоритм *Manager-Slave BNB*

Алгоритм *Manager-Slave BNB* основан на идее подхода *MP*. Алгоритм предполагает наличие одного управляющего потока (*manager*) и нескольких потоков выполнения (*slaves*). Управляющий поток обеспечивает своевременное создание потоков выполнения, балансировку нагрузки между потоками выполнения, аккумуляцией результатов решения подзадач и распределение ресурсов между потоками выполнения. Единственная задача потоков выполнения заключается в обработке предоставленных им подзадач в условиях ограниченного числа итераций.

Работа алгоритма начинается аналогично алгоритму Simple BNB с инициализации следующих параметров: параметры *Tread_max* и *Step_max* были описаны выше; параметры *Steps_limit* и *Subs_limit* означают минимальное число итераций и подзадач в пуле соответственно, при котором имеет смысл производить балансировку нагрузки; структура *benchmark* содержит всю необходимую информацию о решаемой задаче, целевой функции и ограничениях; переменная *origin_sub* аккумулирует в себе информацию об исходной области поиска; переменная *origin_state* хранит в себе информацию о состоянии процесса решения в начальный момент времени; *eps* – требуемая точность решения.

На следующем этапе вызывается функция решения задачи *Solve*. Управляющий поток в начальный момент времени находится в состоянии *origin_state*, то есть пул задач процесса содержит исходную задачу *origin_sub*, значение рекорда равно некоторому начальному значению, максимальное доступное число итераций равно значению переменной *Step_max.* Далее управляющий поток создает первый поток выполнения и назначает ему на обработку задачу *origin_sub*. Также потоку выполнения передаются доступные ресурсы по числу итераций. Следующим шагом управляющий

поток добавляет поток выполнения в список потоков и инициирует его выполнение. Поток выполнения начинает решать предоставленную ему задачу последовательным алгоритмом МВГ. Управляющий поток в свою очередь переходит в режим балансировки загрузки системы.

Перед тем, как обсудить процесс балансировки загрузки, заметим, что процессы выполнения могут находиться в трех различных состояниях: *Ready, Processing* и *Finished*. Состояние *Ready* означает, что процесс готов к обработке новой задачи. Поток переходит в состояние *Processing* при запуске задачи на выполнение и находится в этом состоянии на протяжении всего времени решения. Состояние *Finished* означает, что поток завершил обработку подзадач и готов сообщить результаты управляющему процессу.

Контроль загрузки системы осуществляется следующим образом. Управляющий поток просматривает список созданных потоков в попытке найти поток в одном из состояний *Finished*, *Ready* или *Processing*. Если он находит поток в состоянии *Finished*, то обновляет рекордное значение и забирает оставшиеся после обработки подзадач ресурсы (итерации или подзадачи, в зависимости от того, по какой причине поток завершил выполнение) и переводит поток в состояние *Ready*. Затем управляющий поток пытается сразу загрузить освободившийся поток, исходя из имеющихся у потока-менеджера ресурсов.

Если управляющий поток находит поток в состоянии *Ready*, то старается загрузить его подзадачами, если располагает достаточным количеством ресурсов.

Если встречается поток в состоянии *Processing*, количество активных потоков меньше значения *Thread_max* и, при этом, число оставшихся итераций и подзадач превышает *Steps_limit* и *Subs_limit* соответственно, то производится попытка разгрузить этот поток. Разгрузка может происходить путем создания нового потока либо поиском потока в состоянии *Ready* и передачи ему части ресурсов.

Работа алгоритма продолжается до тех пор, пока количество потоков в состоянии *Processing* $ActiveThreadCount > 0$ или поток-менеджер располагает ресурсами для обработки.

Оценим достоинства и недостатки алгоритма *Manager-Slave BNB*. Одним из главных преимуществ алгоритма является возможность контролировать загруженность потоков и осуществлять балансировку загрузки системы. Однако данное преимущество одновременно является и главным недостатком, так как для осуществления балансировки загрузки управляющему потоку необходимо получить доступ к состоянию рабочих потоков, что сопровождается приостановкой их работы. Каждый рабочий поток, имеет свой мьютекс (*mutex*), что позволяет выполнять остановку только тех потоков, которые участвуют в процессе балансировки загрузки.

Еще одним недостатком данной реализации являются расходы на контроль над работой потоков выполнения со стороны управляющего потока. Чтобы снизить расходы данного типа предлагается приостанавливать работу управляющего потока и возобновлять ее только по мере необходимости.

Алгоритм BNBOmp

Данный алгоритм является самым простым в реализации. Распараллеливание производится с помощью директив OpenMP. Определяется значение *thread_num* – количество потоков, которое можно запустить одновременно. Для большинства архитектур, за редким исключением, оно берется равным количеству логических вычислительных ядер в системе (вызывается функция omp_get_thread_num()). Создается два массива пулов *pool* и *pool_new*, длина массивов *thread_num*. Чтение задач из массива пулов *pool*, распараллеливается с помощью директив #pragma *omp parallel* и *#pragma omp for nowait schedule(dynamic)*, новые задачи каждый поток записывает в свой элемент массива *pool_new*. После исчерпания задач в массиве пулов *pool* и *pool_new* меняются местами (меняются ссылки на массивы). Синхронизация обновления рекордного значения производится с помощью механизма атомарных операций.

V. Численный эксперимент

Для проведения вычислительного эксперимента по сравнению эффективности разработанных алгоритмов был использован гибридный высокопроизводительный вычислительный комплекс (ВК) ФИЦ ИУ РАН [7]. Указанный комплекс обладает следующими характеристиками:

- 2-х процессорный сервер: 2 x IBM Power9 3.8 Гц, 1024 Гб RAM;
- каждый процессор содержит 20 ядер с поддержкой технологии SMT4;
- общее количество доступных логических ядер: 160.

В качестве вспомогательной системы была использована виртуальная машина:

- количество независимых потоков выполнения: 4;
- процессор: *Intel Core i7-2600k CPU @ 3.40 GHz 3.40 GHz*;
- ОЗУ: 8 *Gb*.

В качестве тестовой функции была выбрана функция *Cluster2D2*, используемая при решении «задачи о молекулярном кластере». Функция *Cluster2D2* представляет собой сумму попарных взаимодействий пар совокупности n атомов

$$f(x) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} v(i,j),$$

где $v(i,j) = v_{LJ}(||x_i - x_j||)$ – потенциал парного взаимодействия Леннарда-Джонса, вычисляемый по формуле $v_{LJ}(r) = r^{-12} - 2r^{-6}$. В данном случае $||x_i - x_j||$ – расстояние между атомами i и j, а x_i, x_j – двумерные векторы координат соответствующих атомов, $n = 2$.

Решение задачи оптимизации параметров указанной функции требует значительных вычислительных затрат, а количество подзадач, обработанных в ходе процесса решения, слабо зависит от порядка их обработки. Значение глобального минимума функции -1.0. Поиск

производится на параллелепипеде {0,0; 0,3}{0,0;0,2}{0,7;1,0}{0,8;1,0}. Параметр наибольшего возможного числа итераций был принят равным 10^9. Требуемая точность решения $eps = 0{,}1$. Параметры, зависящие от метода, были выбраны следующими - для виртуальной машины:

Simple BNB *Tread_max=64, Steps_limit=1000*
Manager-Slave BNB *Tread_max=4, Steps_limit=1000, Subs_limit=10*
BNBOmp *thread_num=4*
для Power9:
Simple BNB *Tread_max=1600, Steps_limit=2000*
Manager-Slave BNB *Tread_max=160, Steps_limit=2000, Subs_limit=10*
BNBOmp *thread_num=160*

Эффективность работы алгоритмов будем оценивать среднему времени, затраченному на решение задачи, и критерию avg_*tps* (*average time per subproblem*) = *T*/*N*, где *T* и *N* – время и число итераций, затраченные на решение задачи, соответственно.

Также в результатах экспериментов будут приведены следующие характеристики, отражающие устойчивость разработанных алгоритмов:

- Avg_time_dev – стандартное отклонение времени решения задачи;
- Avg_iter_dev – стандартное отклонение числа итераций, затраченных на решение задачи.

Так как значения параметров числа итераций *N* и времени решения *T* могут различаться при двух независимых запусках решения поставленной задачи, для сравнения алгоритмов используются усредненные значения параметров, полученные при многократном запуске рассматриваемого алгоритма.

Таблица 1. Средние значения количества итераций, времени решения задачи и времени решения подзадач для виртуальной машины.

	avg_iter	avg_time (сек)	avg_tps (мкс)
Simple BNB	23457268	81.88	3.49
BNBOmp	23457223	136.86	5.83
Manager-Slave BNB	23457261	92.20	3.93

Как видно из таблицы 1 среднее количество итераций, затраченных на решение задачи, для всех алгоритмов различается несущественно. Среднее время решения задачи и критерий avg_*tps* изменяется в пределах 30 процентов.

Таблица 2. Средние значения количества итераций, времени решения задачи и времени решения подзадач для Power9.

	avg_iter	avg_time (сек)	avg_tps (мкс)
Simple BNB	23457241	10.55	0.45
BNBOmp	23457223	7.9	0.34
Manager-Slave BNB	23457236	19.51	0.83

Как видно из таблицы 2, среднее количество итераций, затраченных на решение задачи, для всех алгоритмов так же различается несущественно. При сравнении по среднему времени решения задачи и критерию avg_*tps* наилучший результат показывают алгоритмы ***Simple BNB*** и **BNBOmp**. Алгоритм ***Manager-Slave BNB*** показывает результат в два раза хуже.

Таблица 3. Стандартное отклонение количества итераций и времени решения задачи для виртуальной машины

	avg_iter_dev	avg_time_dev (сек)
Simple BNB	93.88	4.84
BNBOmp	0	4.80
Manager-Slave BNB	13.81	5.51

Таблица 4. Стандартное отклонение количества итераций и времени решения задачи для Power9.

	avg_iter_dev	avg_time_dev (сек)
Simple BNB	26.78	0.92
BNBOmp	0	0.24
Manager-Slave BNB	28.86	1.90

Из таблиц 3 и 4 видно, что для алгоритма **BNBOmp** стандартное отклонение количества итераций равно нулю. Так же у этого алгоритма существенно лучшее стандартное отклонение времени решения задачи. Для алгоритма ***Simple BNB*** при увеличении параметра *Tread_max* среднее отклонение количества итераций уменьшается.

VI. Заключение

Сравнительный анализ алгоритмов показал, что для решения задачи о молекулярном кластере на маломощной вычислительной системе (виртуальная машина, 4 независимых потока исполнения) алгоритмы балансировки нагрузки и механизмы синхронизации рекордного значения не играют существенной роли. При росте вычислительной мощности (увеличении независимых потоков исполнения) растет доля времени, затрачиваемая на синхронизацию и/или управлению потоками исполнения. Наилучшие результаты показывают алгоритмы ***Simple BNB*** и **BNBOmp**. Так же эти алгоритмы имеют меньшее количество параметров по сравнению с ***Manager-Slave BNB***. В заключение можно сказать, что более простой алгоритм **BNBOmp** показывал примерно равные (для маломощной системы)

или немного лучшие (примерно в два раза по сравнению с ***Manager-Slave BNB***) результаты. Учитывая простоту реализации и отсутствие параметров метода (параметр *thread_num* задается равным количеству логических ядер в системе), авторы рекомендуют использовать этот метод, при наличии достаточного количества оперативной памяти и когда количество подзадач, обработанных в ходе процесса решения, слабо зависит от порядка их обработки.

Библиография

[1] Evtushenko, Y. G., & Posypkin, M. A. (2011). An application of the nonuniform covering method to global optimization of mixed integer nonlinear problems. Computational Mathematics and Mathematical Physics, 51(8), 1286-1298. doi:10.1134/S0965542511080082

[2] Евтушенко Ю. Г., Посыпкин М. А., Рыбак Л. А., Туркин А. В. Отыскание множеств решений систем нелинейных неравенств //Журнал вычислительной математики и математической физики. – 2017. – Т. 57. – №. 8. – С. 1248-1254. doi:10.7868/S0044466917080075

[3] Горчаков А. Ю. Применение метода неравномерных покрытий для решения задачи поиска максимума информативности предиката //International Journal of Open Information Technologies. – 2017. – Т. 5. – №. 2. – С.29-33.

[4] М.А. Посыпкин, Н.П. Храпов, В.Н. Филиппов «Метод ветвей и границ на грид-системах персональных компьютеров», Программные системы: теория и приложения №2(16), 2013, –С. 43-69.

[5] М.А. Посыпкин «Архитектура и программная организация библиотеки для решения задач дискретной оптимизации методом ветвей и границ на многопроцессорных вычислительных комплексах» // Труды ИСА РАН, 2006, Т. 25, –С. 18-25.

[6] Bader, D.A.; Hart, W.E.; Phillips, C.A., Parallel Algorithm Design for Branch and Bound. In Tutorials on Emerging Methodologies and Applications in Operations Research: Presented at INFORMS 2004, Denver, CO, International Series in Operations Research & Management Science, Kluwer Academic Publishers: Dordrecht, The Netherlands, 2004, –pp. 5-44.

[7] Федеральный исследовательский центр Информатика и управление РАН [Электронный ресурс]: сайт. – Москва: ФИЦ ИУ РАН. – URL: http://hhpcc.frccsc.ru (дата обращения: 12.09.2018)"

Experimental study of three options for implementing of the nonuniform covering method for multicore systems with shared memory

A.Y. Gorchakov, M.A. Posypkin, Y.V. Yamchenko

***Abstract*— In this paper, we consider three effective parallel implementations of the nonuniform method covering intended for computing systems with shared memory. The nonuniform covering method is one of the most well-known deterministic methods for solving global optimization problems based on the branch and bound scheme. Given the computational complexity of the method and the widespread use of high-performance multi-core systems with shared memory, the development of parallel implementations of this method is of particular relevance. The article proposes several approaches to parallelizing the method of nonuniform covering. Currently, there are several standards for creating multi-threaded applications. The paper discusses two such standards: OpenMP 4.0 and C ++ 17. Several modes are used for synchronization between threads. The paper provides a description of the algorithms and their software implementations. An experimental study was carried out on a test problem close to real — the search for the minimum energy of a molecular cluster. As a computational platform for conducting experiments, modern high-performance systems were used. The study showed that for this type of tasks, the performance of the methods (by the number of iterations) is approximately at the same level. In addition, it was shown experimentally that the complexity of the algorithm does not always lead to an increase in its efficiency.**

***Keywords*— method non-uniform covering, multicore systems, parallel algorithm, load balancing.**