ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 279 за 2018 г.

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

Григорьев С.К.

Алгоритмы и модели данных для параллельной реализации алгоритма гарантированной генерации тетраэдральной сетки

Рекомендуемая форма библиографической ссылки: Григорьев С.К. Алгоритмы и модели данных для параллельной реализации алгоритма гарантированной генерации тетраэдральной сетки // Препринты ИПМ им. М.В.Келдыша. 2018. № 279. 16 с. doi:10.20948/prepr-2018-279
URL: http://library.keldysh.ru/preprint.asp?id=2018-279

О р д е н а Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В.Келдыша
Р о с с и й с к о й а к а д е м и и н а у к

Григорьев С.К.

Алгоритмы и модели данных для параллельной реализации алгоритма гарантированной генерации тетраэдральной сетки

Москва — 2018

Григорьев С.К.

Алгоритмы и модели данных для параллельной реализации алгоритма гарантированной генерации тетраэдральной сетки

Рассматриваются алгоритмы, структуры данных и особенности реализации алгоритма гарантированной генерации тетраэдральной сетки проекционным методом.

Ключевые слова: неструктурированные сетки, генератор расчетных сеток, MPI, рациональные числа

Sergej Konstantinovich Grigorjev

Algorithms and data models for parallel implementation of the algorithm of guaranteed generation of a tetrahedral mesh

Algorithms, data structures and implementation features of the projection-based tetrahedral mesh generation algorithm are considered.

Key words: unstructured meshes, mesh generation, MPI, rational numbers

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект 16-07-00519 а.

Оглавление

Введение

При численном моделировании физических процессов методами механики сплошной среды широко используются геометрические расчетные сетки. Процесс построения трёхмерной сетки в областях сложной формы является на сегодняшний день одним из наиболее алгоритмически сложных и затратных по времени.

Задача построения тетраэдральной сетки на сегодняшний день решается рядом прикладных программных пакетов, как коммерческих, так и с открытым исходным кодом.

Одним из пакетов с открытым исходным кодом, в котором имеется модуль построения тетраэдральных сеток, является пакет GMSH. В качестве алгоритма тетраэдризации в этом пакете используется алгоритм движущегося фронта (advancing front) [1]. Другим построителем сеток с открытым исходным кодом является Tgrid [2], позиционирующийся как построитель сеток в областях сложной формы.

Коммерческих пакетов, предоставляющих возможность тетраэдризации, заметно больше. Например, в пакете ANSYS представлен целый комплекс различных построителей сеток, позволяющих строить сетки различного качества для различных задач [3,4]. Другие построители, например Simcenter-3D [5], позиционируют себя как универсальные построители для произвольных областей. Существуют построители, например Poinwise [6], позволяющие строить гибридные сетки.

Несмотря на то что все эти и многие другие, не попавшие в обзор, пакеты используются для решения практических задач, в своей основе они используют алгоритмы, для которых не доказана возможность построения сетки за конечное время.

Алгоритм, описанный в [7], даёт гарантию получения результата при формировании сетки в произвольной области за конечное, полиномиально зависящее от количества треугольников поверхности время.

В силу большой затратности по времени процесса построения сетки алгоритмы генерации расчетных сеток целесообразно распараллеливать.

В работе рассматриваются структуры данных и алгоритмы, необходимые для реализации алгоритма [7], а также некоторые свойства параллельной реализации.

Особенности последовательного алгоритма

Алгоритм гарантированной генерации тетраэдральной сетки, подробно описанный в [7], хорошо распараллеливается с использованием методов геометрического параллелизма.

Упомянутый алгоритм состоит из 4 этапов: ортогонального проецирования и формирования призм, построения взаимных пересечений боковых граней призм (стыковка призм), триангуляции боковых стенок призм и формирования

тетраэдров на основании призм. Каждый из этих этапов распараллеливается отдельно, в силу того, что для начала каждого следующего этапа необходимо завершение предыдущего.

Для обеспечения точности вычислений и для исключения погрешности округления во время промежуточных вычислений используются арифметика рациональных чисел с произвольной разрядностью (далее – рациональная арифметика). Недостатком применения рациональной арифметики является снижение скорости вычисления элементарных операций и увеличение затрат на хранение данных в памяти.

Будем считать, что координаты узлов сетки, поступающей на вход алгоритма, помещаются в переменные типа double. Сохранение точности при преобразовании координат из вещественных чисел в рациональные осуществляется путём умножения на целочисленный коэффициент, соответствующий числу значащих десятичных цифр после запятой исходного числа. Таким образом, на момент начала работы алгоритма все знаменатели рациональных дробей равны единице.

В рамках работы рассматривается параллельная реализация алгоритма, написанная на языке C++ с использованием библиотеки для распределённых вычислений MPI. В качестве библиотеки для работы с числами произвольной разрядности используется библиотека cBigNumber [8].

Основные алгоритмы для межпроцессорного взаимодействия

Во время выполнения всех этапов алгоритма все координаты узлов хранятся в рациональной арифметике. Для работы на распределённой системе необходимо передавать эти координаты между MPI-процессами.

Применение рациональных чисел произвольной разрядности для хранения координат создаёт сложности при передаче данных между процессорами из-за того, что для хранения числителя и знаменателя рациональной дроби используются динамические массивы, не поддерживаемые для регистрации в MPI.

Оценим количество памяти, требуемое для хранения одного рационального числа в сравнении со стандартными типами. Рациональное число в реализации представляет собой дробь, состоящую из двух целых чисел произвольной разрядности (экономия на знаке знаменателя рациональной дроби при использовании рациональной арифметики представляется нецелесообразной). Число с произвольной разрядностью содержит в себе, в рамках реализации в cBigNumber [8], динамический массив целых чисел, составляющих непосредственно число с произвольной разрядностью, и целое число, указывающее на количество «блоков» - длину динамического массива. Таким образом, хранение одного рационального числа требует не меньше чем 4 числа типа int, что в случае использования int32 даёт увеличение занимаемой

памяти в 2 раза, а в случае int64 – в 4 раза соответственно. В работе [9] было показано, что максимальная разрядность хранимых координат не может быть больше, чем в $36n_1 + 21$ разрядов, где n_1 – число разрядов исходных координат. Отсюда можно получить грубую оценку максимальной длины массива «блоков» – 6 элементов. Таким образом, в худшем случае для хранения одного рационального числа требуется в 7 раз больше памяти, чем для обычного числа (при использовании int32). Такой перерасход памяти приводит к чрезмерным затратам по памяти на буферы межпроцессорного обмена.

В силу того, что общий объём оперативной памяти, требуемой для хранения координат, достаточно велик, использование статического буфера, заведомо вмещающего в себя все передаваемые элементы, представляется неэффективным, т.к. такой буфер потребляет слишком много памяти.

Для уменьшения затрат по памяти можно использовать буфер обмена меньшего размера, чем необходимый для приёма и передачи всех необходимых узлов, при передаче разбивая весь массив передаваемых данных на несколько массивов меньшего размера, которые можно по мере формирования передавать. Таким образом, при передаче данных большого объёма функция MPI_Send вызывается несколько раз, вместо одного.

Другой проблемой является невозможность регистрации типов данных с динамическими массивами как типов MPI. Под типами данных с динамическими массивами понимаются такие структуры, длины полей двух различных объектов которых не совпадают. При помощи таких структур в том числе реализуются числа произвольной разрядности в [8].

Чтобы передать рациональное число с произвольной разрядностью, необходимо создание механизма преобразования из рационального числа к целочисленному типу, поддерживаемому MPI. Для реализации такого преобразования числитель и знаменатель каждого рационального числа записываются в буфер последовательно.

Для передачи целого числа произвольной разрядности реализуется алгоритм преобразования такого числа в несколько целых чисел (int) и обратно, что позволяет передавать числа произвольной разрядности с помощью MPI.

Таким образом, для приёма и передачи рациональных чисел используется целочисленный буфер, содержащий преобразованные в массивы целых числе числа произвольной разрядности.

Рассмотрим распараллеливание каждого из этапов в отдельности.

Распараллеливание основных этапов выполнения алгоритма

1. Алгоритмы для первого этапа

Первый этап состоит в построении множества призм при помощи ортогонального проецирования BOT-треугольников на TOP-треугольники.

Построение проекции под каждым рассматриваемым треугольником требует рассмотрения всех треугольников, попавших в область проецирования данного треугольника [7]. Так как построение проекции, триангуляция сформировавшегося в результате проецирования графа и формирование на основании результата триангуляции набора призм может выполняться для каждого TOP-треугольника независимо, то и каждый TOP-треугольник может быть обработан независимо.

Таким образом, можно использовать методы геометрического параллелизма для распараллеливания этого этапа алгоритма. Таким образом, сложность распараллеливания этого этапа алгоритма сводится к удачному распределению треугольников исходной поверхности между процессорами.

Балансировка нагрузки на этом этапе сопряжена с целым рядом трудностей.

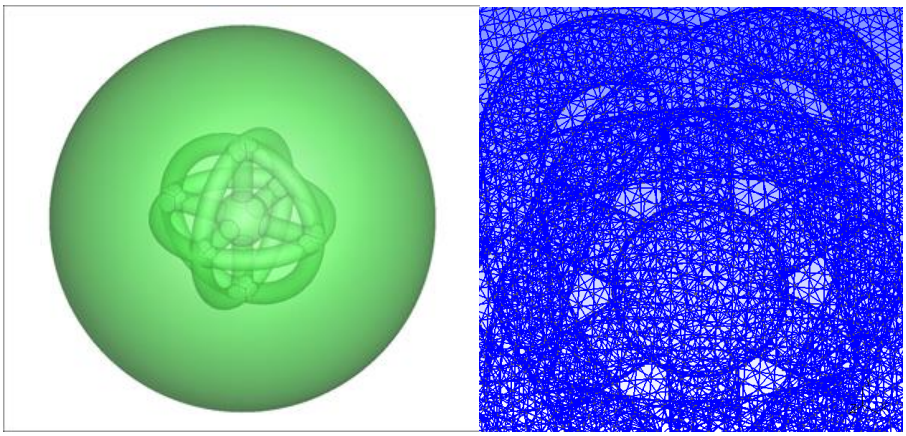

Рис. 1. Пример попадания большого количества треугольников под один из рассматриваемых треугольников.

На рисунке 1 показано, что даже если в конечном счете под одним из треугольников окажется относительно мало призм, то при построении проекции придётся рассмотреть существенно большее число треугольников, располагающихся под рассматриваемым. При этом балансировать по процессам необходимо именно TOP-треугольники, так как именно они инкапсулируют в себе действия на этом этапе работы алгоритма. Основная проблема заключается в том, что предсказать, какой из BOT-треугольников сформирует призму под рассматриваемым треугольником, а какой нет, без построения проекции невозможно. Остаётся использование приближенного критерия оценки времени обработки каждого TOP-треугольника, в зависимости от числа треугольников, попавших в те же блоки кэша, что и рассматриваемый треугольник. По этой же причине достаточно сложно распределять BOT-треугольники между процессорами. Для небольших сеток можно просто передавать все имеющиеся BOT-треугольники между процессорами.

2. Алгоритмы для второго этапа

Второй этап выполнения алгоритма подразумевает стыковку построенных на первом этапе призм между собой, а также добавление всех WALL-треугольников.

Стыковка призм подразумевает под собой построение всех рёбер и точек пересечения на боковых гранях для всех призм. Существует несколько возможных вариантов соприкосновения призм (Рис. 2 а-д.):

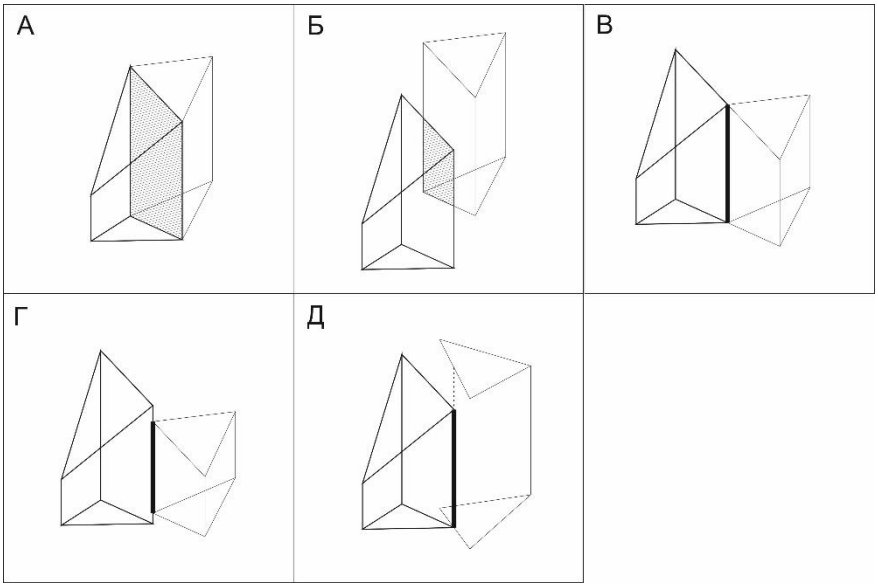

Рис. 2. Рассматриваемые варианты касания призм [7]

Эти варианты касания призм приводят к необходимости решения задачи поиска двух соприкасающихся призм [7].

Существует несколько последовательных алгоритмов решения данной задачи. Первый заключается в добавлении рёбер каждой отдельно взятой призмы в макроструктуру, объединяющую в себя все вершины и рёбра призм. Второй состоит в раздельном добавлении вершин и частей рёбер в топологию каждой отдельной призмы.

Рассмотрим достоинства и недостатки описанных методов с точки зрения реализации параллельных версий этих алгоритмов. Первый метод позволяет существенно экономить память на хранении узлов и рёбер призм, так как при поддержании макроструктуры все эти элементы хранятся в одном экземпляре. Недостатками являются необходимость корректного поддержания пограничных слоёв узлов и рёбер, а также сложности с добавлением новых узлов и рёбер в макроструктуру. Второй подход позволяет существенно сократить время работы за счет рассмотрения и добавления новых элементов только к элементам, принадлежащим отдельной призме, ценой существенного увеличения объёма хранимых данных. Так, большая часть узлов в конечном итоге оказывается продублирована несколько раз. Второй метод предпочтительнее с точки зрения реализации параллельного алгоритма, так как

позволяет не производить обмены данными между процессорами в течение работы всего этапа. В рамках работы рассматривается параллельная реализация второго метода.

Возможность независимо обрабатывать каждую из призм позволяет применить метод геометрического параллелизма для распараллеливания этого этапа. Необходимость уточнения взаимного расположения призм между собой приводит к тому, что на каждом процессоре необходимо хранить не только непосредственно обрабатываемые призмы, но и все призмы, расположенные рядом в кэше [7]. В худшем случае на каждом процессоре будут храниться все призмы (Рис. 3), что вместе с необходимостью хранить все промежуточные результаты в рациональной арифметике приводит к неприемлемым затратам по памяти.

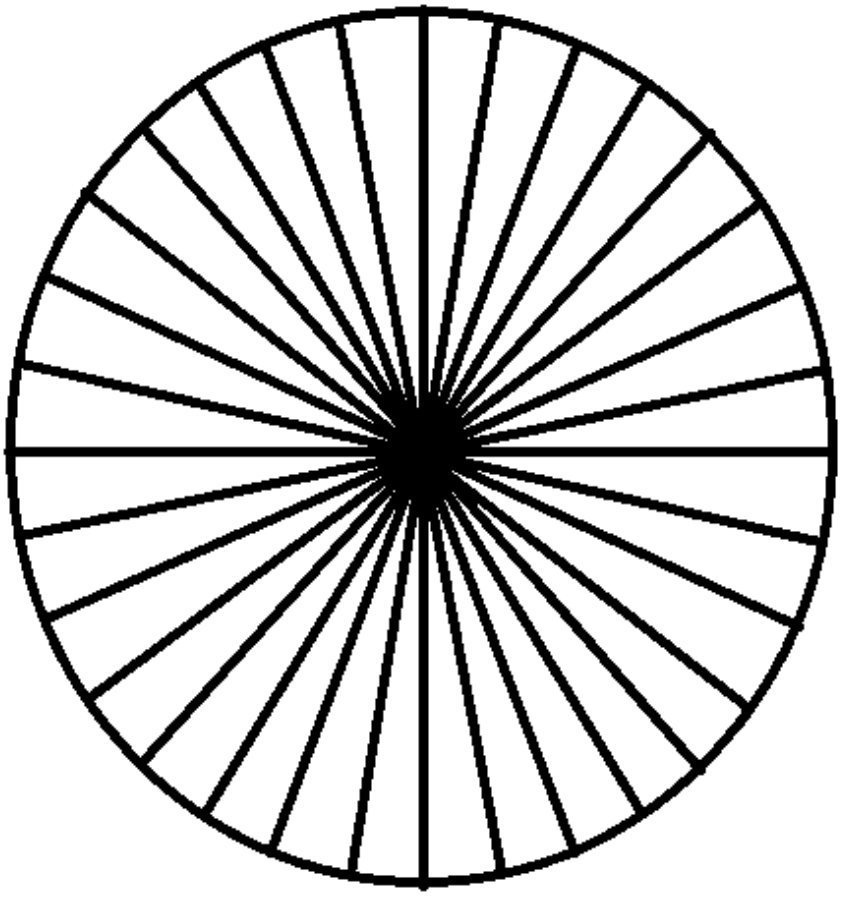

Рис. 3. Наихудший случай для второго этапа алгоритма [7]

Случай, показанный на рисунке 5, является вырожденным, и в общем случае можно существенно сократить объём хранимых на одном вычислительном узле призм при помощи декомпозиции графа приближенного соседства. При этом задачами декомпозиции являются максимально возможная

экономия памяти и балансировка нагрузки, причем приоритет отдаётся экономии памяти.

На этом этапе алгоритма неизвестно точное отношение соседства между всеми призмами, можно использовать геометрический кэш для формирования графа приближенного соседства. В зависимости от конкретной рассматриваемой области можно использовать как двумерный, так и трёхмерный кэш.

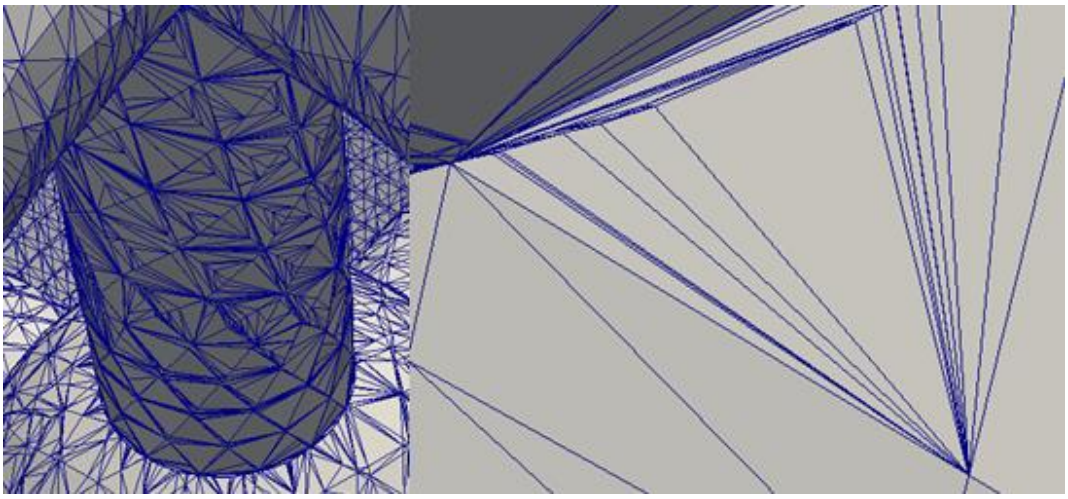

Рис. 4. Пример большого числа призм, находящихся крайне близко друг к другу [9].

На рисунке 4 показан один из плохих сценариев, когда большое количество призм находится крайне близко друг к другу в пространстве, хотя на самом деле друг друга не касаются. На левой стороне рисунка на каждый треугольник опирается отдельная призма. На правой стороне рисунка представлено приближенное изображение описываемой проблемы. Этот пример приводит к решению: если мы не можем использовать в данном случае двумерный геометрический кэш, так как он приводит к количеству операций $O(N^2)$, где N – число призм, то предлагается формировать трёхмерный геометрический кэш, чтобы несколько уменьшить вычислительную сложность на данном этапе.

Принцип работы трёхмерного программного геометрического кэша схож с принципом работы двумерного кэша, с той разницей, что в трёхмерном случае блоком кэша будет параллелепипед (далее будем называть «куб кэша»). Этот подход позволяет в некоторых случаях существенно уменьшить число призм, попавших в куб кэша, что уменьшает число рёбер в графе приближенного соседства и позволяет более точно определить приближенный вес призмы. Однако этот подход далёк от идеала. Например, на рисунке 7 показаны некоторые топологические артефакты, т.н. «звёздчатые структуры». Обработка таких структур не может быть ускорена при помощи геометрического кэширования.

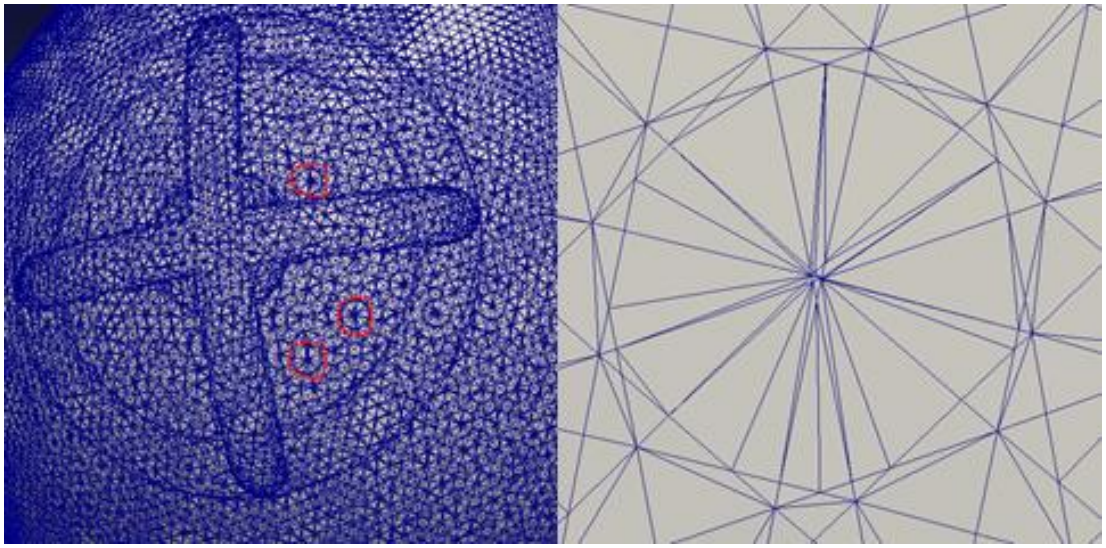

Рис. 5. Пример низкой эффективности трёхмерного кэша.

Ещё одним способом решения проблемы с областями, показанными на рис. 4, является поворот исходной области вокруг центра масс, таким образом уменьшается число призм, расположенных друг над другом в непосредственной близости или касаясь друг друга. У этого подхода также есть проблемы, связанные с определением подходящего угла поворота, уменьшающего, а не увеличивающего число касающихся друг друга призм. Кроме того, этот подход не решает проблемы звёздчатых структур, однако, как будет показано ниже, этот подход приводит в некоторых случаях к существенному снижению времени работы всего алгоритма.

Ещё одной проблемой, вытекающей из проблемы звёздчатых структур, является необходимость хранения всех элементов подобных структур на одном процессоре, что вкупе с использованием рациональной арифметики произвольной разрядности приводит существенному росту используемой алгоритмом памяти.

Другая проблема состоит в формировании распределённого графа на большом количестве доменов, с минимальной разницей в суммарном весе доменов и минимальным количеством связей между доменами для минимизации количества призм, хранящихся на одном процессоре. Для декомпозиции такого графа используется пакет ParMETIS [10].

Основной проблемой такого подхода с точки зрения балансировки нагрузки является исключение из рассмотрения времени добавления узлов и рёбер в призмы, а также добавления всех WALL-треугольников исходной сетки, не рассмотренных на первом этапе алгоритма.

Кроме того, звёздчатые структуры создают серьёзные проблемы для балансировки нагрузки, так как разделить случаи, показанные на рис. 3 и рис. 5, априорно не представляется возможным.

Балансный вес каждой из призм получается из геометрического кэша путём получения приближенного числа соседей рассматриваемой призмы, из-за

чего итоговая балансировка нагрузки может оказаться крайне неравномерной. Например, в случае, показанном на рисунке 6, реальное количество соседей для каждой из призм будет отличаться от балансного веса более чем в 10 раз. С другой стороны, такой подход позволяет точно определить все призмы, которые необходимо рассмотреть при стыковке с текущим элементом, что позволяет получить достаточно неплохую балансировку и по используемой памяти, и по нагрузке [9].

3. Алгоритмы для третьего и четвёртого этапов

Третий этап работы алгоритма подразумевает построение согласованной триангуляции боковых граней призм. Существует два подхода к реализации этого алгоритма: триангуляция каждой грани в отдельности или триангуляция боковых граней в рамках призмы, что подразумевает создание детерминированного алгоритма двумерной триангуляции.

Основная сложность первого варианта состоит в обработке случаев соприкосновения призм частями граней (см. рис. 2(б)). Также после триангуляции отдельных граней (частей граней) необходимо восстановить призмы для следующего этапа, что существенно увеличит объём передаваемой между процессами информации.

Рассмотрим второй вариант, как требующий меньшего объёма передаваемой информации. В таком случае каждая призма может быть обработана отдельно, вне зависимости от остальных призм. Это, однако, приводит к удвоению выполняемой работы.

Балансировка нагрузки может быть осуществлена при помощи распределения призм между процессорами, в качестве критерия балансировки в таком случае будет использоваться количество узлов, принадлежащее призме. Кроме того, можно использовать балансировку второго этапа, так как используемый на втором этапе способ определения балансного веса даёт приближенную оценку, в том числе количеству вершин и рёбер, которые окажутся у той или иной призмы. Второй способ даёт худшее итоговое распределение нагрузки [9], но позволяет сэкономить время на передаче призм с их топологиями между процессорами.

Четвёртый этап, добавление точки в центр масс призмы, осуществляется полностью независимо для каждой призмы, таким образом, распараллеливание этапа осуществляется путём равномерного распределения призм между вычислительными узлами.

Параллельная реализация алгоритма приводит к появлению дополнительного этапа, а именно восстановления полученного результата из распределённых по процессорам наборов призм. Для этого необходимо получить для каждого узла его уникальный номер в общей конфигурации.

Эта задача решается путём упорядочения всех вершин по координатам алгоритмом на основе сети сортировки Бэтчера [11].

Кроме того, формирования дву- и трёхмерного кэша также должно осуществляться параллельно. Это является узким местом алгоритма, так как сформированный кэш в полном размере должен оказаться на всех вычислительных узлах, для дальнейшего построения графа соседства.

Алгоритм параллельного формирования двумерного кэша состоит из двух этапов. На первом этапе каждый отдельный процесс проходит по доставшимся ему элементам, и строит на их основе локальный ограничивающий прямоугольник. После этого, по всем процессорам создаётся общий ограничивающий прямоугольник, размеры которого могут превосходить по размерам построенный локально на каждом процессе. В завершении первого этапа формирования кэша происходит поиск минимальных и максимальных значений координат, образующих общий ограничивающий прямоугольник. Эти величины передаются на все процессоры.

Второй этап состоит в последовательном заполнении кэша локально на каждом процессоре. После этого производиться слияние всех ячеек кэша по всем процессорам, таким образом, на всех процессорах оказывается один и тот же кэш.

В рамках работы считается, что размер двумерного кэша составляет 100×100 ячеек, объём трёхмерного кеша составляет $100 \times 100 \times 4$ ячеек. Объём кэша представлен в виде произведения $n_x \times n_y \times n_z$, где n_x, n_y, n_z – количество ячеек кэша вдоль соответствующего направления.

Параллельный алгоритм формирования трёхмерного кэша работает аналогично, с превращением ограничивающего прямоугольника в ограничивающий параллелепипед.

Вычислительная сложность заполнения подобного кэша составляет $O(N)$ в последовательном случае.

Следует отметить, что подобный кэш необходим только в том случае, если исходная сетка не разделена на поддомены другими способами.

В рамках первых трёх этапов обработки возможно распараллеливание на общей памяти с использованием библиотеки OpenMP.

В силу описанных особенностей реализации каждый из этапов алгоритма выполняется без использования операций обмена данными между процессорами, таким образом, теоретическое ускорение алгоритма линейно зависит от числа процессоров p. Однако на общее ускорение будут влиять все промежуточные действия, такие как балансировка нагрузки, параллельное формирование кэша и формирование ответа.

Наихудшим в смысле ускорения будет этап формирования конечного результата, в силу возможного существенного дисбаланса по количеству вершин между процессорами.

Особенности реализации

Рассмотрим реализацию описанного выше алгоритма. Все дальнейшие эксперименты проводились на вычислительном кластере К100.

Рассмотрим несколько примеров. Начнем с сетки, показанной на рисунке 6. Эта сетка содержит 1268 треугольников, 636 узлов. Как видно из графика, предложенного на рис. 7, время работы на двенадцати процессах в 2.64 раза меньше времени выполнения на 4 процессах. Время работы на 8 процессах оказалось в 1.74 раза меньше, чем время работы на 4 процессах. Нелинейность объясняется низким качеством балансировки нагрузки первого этапа алгоритма: время обработки на одном из процессоров оказывается стабильно на 40% больше, чем на остальных. Связанно это с тем, что при построении призм на границе вогнутой области (на рисунке слева), хотя количество треугольников в кубе кэша оказывается невелико, количество результирующих формируемых призм оказывается достаточно большим, что и обуславливает задержку.

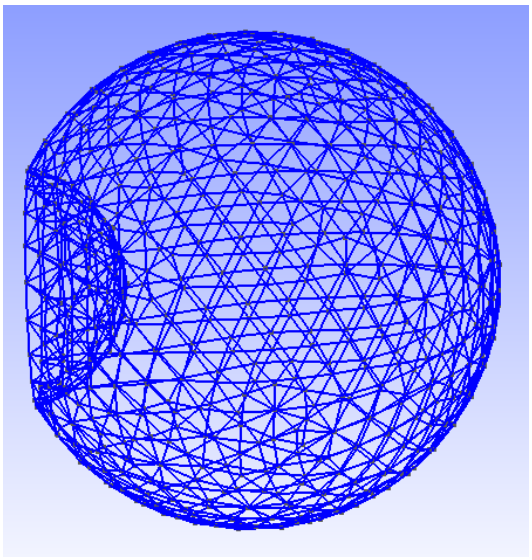

Рис. 6. Пример области.

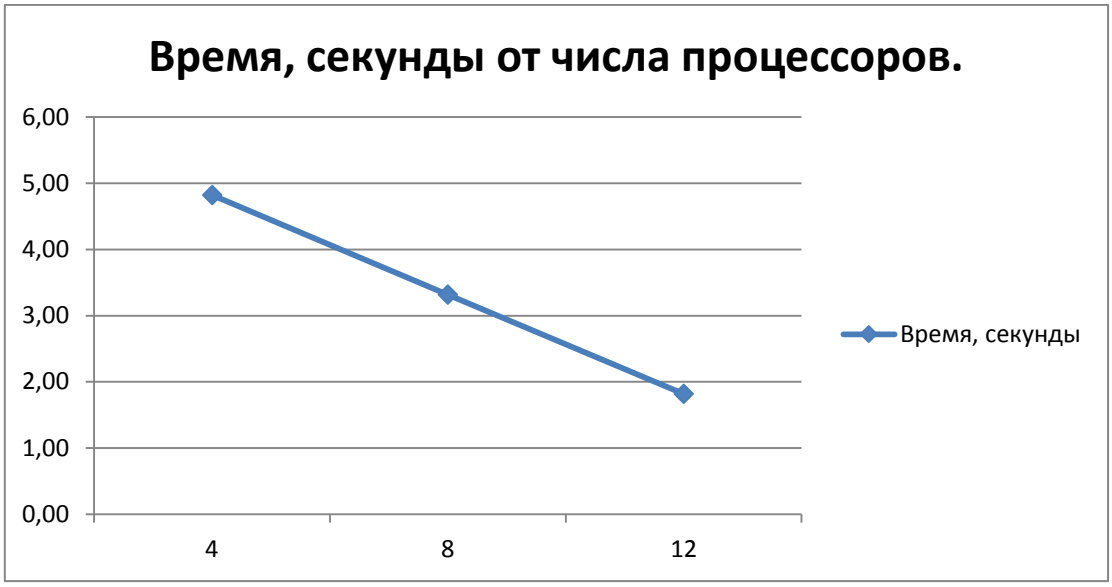

Рис. 7. График зависимости времени выполнения от числа процессоров, пример 1.

Рассмотрим второй пример, представляющий собой два вложенных куба (рис. 8). Сетка имеет характерную особенность, заключающуюся в большом количестве соприкасающихся друг с другом различным образом призм, большом количестве WALL-треугольников, что делает эту область сложной для рассматриваемого алгоритма, несмотря на маленькие размеры – 467 треугольников поверхности. Время построения сетки в такой области составляет 1.4 минуты, на восьми MPI процессах. При этом время построения проекции колеблется от 1.5 до 5 секунд, а время стыковки призм – от 36 до 41 секунды. На этом примере хорошо видны особенности рассматриваемых алгоритмов балансировки. Простое наложение нескольких слоёв треугольников друг на друга приводит в худшем случае к серьёзному дисбалансу нагрузки на первом этапе. При увеличении и усложнении способов соприкосновения призм между собой возникают проблемы и с балансировкой нагрузки второго этапа.

Разработанные алгоритмы и библиотеки интегрированы в пакет MARPLE3D [12].

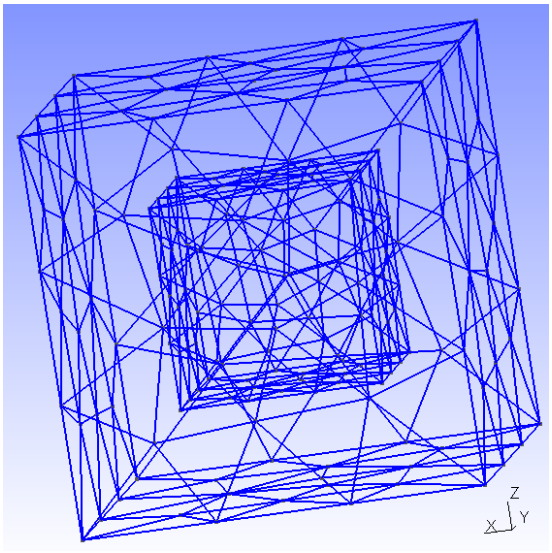

Рис. 8. Область с большим количеством нетривиально касающихся призм.

Заключение

Предложены методы для распараллеливания четырёх этапов рассматриваемого алгоритма гарантированной генерации тетраэдральной сетки проекционным методом: построение проекции и формирования призм, стыковки призм, триангуляции боковых стенок призм, заполнение сформированных призм тетраэдрами. Для второго этапа рассмотрено несколько возможных последовательных алгоритмов выполнения, указаны особенности их распараллеливания. Для распараллеливания всех этапов используются методы геометрического параллелизма. Для реализации выбраны алгоритмы с наименьшим числом межпроцессорных взаимодействий во время выполнения этапов.

Разработанные алгоритмы реализованы на языке программирования C++ с использованием библиотеки MPI. Показан график зависимости времени выполнения от числа процессоров, рассмотрены особенности предложенных алгоритмов. На тестовом примере время работы на восьми процессах было в 1.74 раза меньше, чем время обработки на четырёх процессах.

Предложены алгоритмы для обеспечения параллельной работы с рациональными числами. Проведена оценка количества памяти, требуемой для буфера приема и передачи данных. Размер буфера при использовании рациональной арифметики оказывается в 2-7 раз больше, чем при использовании стандартных типов. Разработан алгоритм передачи буфера частями для экономии используемой памяти.

Литература

1. J.-F. Remacle, F. Henrotte, T. Carrier-Baudouin, E. Béchet, E. Marchandise, C. Geuzaine and T. Mouton. A frontal Delaunay quad mesh generator using the

L∞ norm. International Journal for Numerical Methods in Engineering, 94(5), pp. 494-512, 2013.

2. Qi, D., and W. Lin (2006), TGrid: A new grid environment, paper presented at First International Multi-Symposiums on Computer and Computational Sciences Conference, Int. Multi-Symp. on Comput. and Comput.Sci., Hangzhou, China, 20–24 June.
3. Overset Mesh [Электронный ресурс]. – Режим доступа: https://www.ansys.com/products/fluids/ansys-fluent/overset-mesh (дата обращения 13.12.2018).
4. ANSYS ICEM CFD [Электронный ресурс]. – Режим доступа: https://cae-expert.ru/product/ansys-icem-cfd (дата обращения 13.12.2018).
5. Simcenter-3D [Электронный ресурс]. – Режим доступа: https://www.plm.automation.siemens.com/global/ru/products/simcenter/simcenter-3d.html (дата обращения 13.12.2018).
6. Pointwise [Электронный ресурс]. – Режим доступа: https://www.pointwise.com/ (дата обращения 13.12.2018).
7. М.В. Якобовский, С.К. Григорьев Алгоритм гарантированной генерации тетраэдральной сетки проекционным методом // Препринты ИПМ им. М.В.Келдыша. 2018. No 109. 18 с. doi:10.20948/prepr-2018-109 URL: http://library.keldysh.ru/preprint.asp?id=2018-109
8. Р. Н. Шакиров Применение методов обнаружения и компенсации ошибок в целочисленном классе cBigNumber // Программирование. – 2010. No 1. С. 50-65 (R.N.Shakirov C++ class for integers of unlimited range URL: http://www.imach.uran.ru/cbignum) [Электронный ресурс]. – Режим доступа: http://www.imach.uran.ru/cbignum
9. Grigorjev, Sergej & V. Yakobovskiy, Mikhail. (2018). Static Balancing Methods in Projection-Based Mesh Generation Algorithm: 12th International Conference, PCT 2018, Rostov-on-Don, Russia, April 2–6, 2018, Revised Selected Papers. 135-146. 10.1007/978-3-319-99673-8_10.
10. Karypis G. (2011) METIS and ParMETIS. In: Padua D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA
11. Д.Э. Кнут Искусство программирования, т.3. Сортировка и поиск 2-е изд.: Пер. с английского – М.: Издательский дом «Вильямс», 2001.
12. Пакет прикладных программ MARPLE3D для моделирования на высокопроизводительных ЭВМ импульсной магнитоускоренной плазмы / В.А.Гасилов [и др.] // Препринты ИПМ им. М.В.Келдыша. 2011. № 20. 36 с. URL: http://library.keldysh.ru/preprint.asp?id=2011-20