СТРУКТУРЫ ДАННЫХ И ОРГАНИЗАЦИЯ ВЫЧИСЛЕНИЙ ДЛЯ ПОСТРОЕНИЯ АЖУРНЫХ СЕТОК КОНЕЧНЫХ ЭЛЕМЕНТОВ

В.Л. Тарасов, Д.Т. Чекмарев, П.Д. Чекмарев

Нижегородский госуниверситет им. Н.И. Лобачевского

Рассмотрены вопросы эффективной организации процесса прореживания сплошных сеток тетраэдральных конечных элементов в ажурные. Предложено использовать упорядоченный контейнер для хранения списка инцидентности конечных элементов их ребрам. Для распараллеливания вычислений исходная область разбивается на части, топологически эквивалентные сферическим слоям. Приведены оценки времени работы различных этапов алгоритма.

Введение

Методы конечных элементов предполагают, как правило, что расчетная область заполняется конечными элементами сплошь – без промежутков и наложения друг на друга. В [1, 2] предложены ажурные численные схемы МКЭ решения трехмерных задач механики сплошных сред, в которых конечные элементы заполняют расчетную область с промежутками. Это позволяет более эффективно использовать информацию в узлах сетки о напряженно-деформированнном состоянии сред и избегать лишних вычислений при описании процессов деформирования без ущерба для точности получаемых численных решений. Теоретические обоснования подобных схем даны в работах [3-5].

Существующие расчетные комплексы, использующие МКЭ, позволяют строить сплошные сетки конечных элементов. В работе [6] рассмотрено построение изначально ажурных сеток.

В настоящей работе предложен алгоритм, позволяющий преобразовывать сплошные сетки конечных элементов в ажурные, и его реализация. Для создания эффективных структур данных использована стандартная библиотека шаблонов (STL) языка C++ [7, 8]. Многопоточность реализована с помощью библиотеки Qt [9].

1. Постановка задачи, основные этапы алгоритма

Имеется тетраэдральная сетка, сгенерированная сторонним сеточным генератором, задаваемая координатами узлов и четверками номеров узлов, определяющих конечные элементы сетки.

Каждая пара узлов, относящихся к некоторому элементу, образует ребро. Конечный элемент в виде тетраэдра имеет шесть ребер, которые могут принадлежать сразу нескольким элементам. Будем называть число конечных элементов, которым принадлежит ребро, его весом.

Необходимо исключить максимальное число элементов так, чтобы выполнялись условия аппроксимации ажурной схемы. Достаточным условием этого является сохранение всех ребер сетки. Это значит, что если два узла исходной сетки принадлежат одному или нескольким конечным элементам, в ажурной сетке они будут принадлежать хотя бы одному конечному элементу.

В качестве входных данных выступает список элементов с указанием принадлежащих ему узлов:

List[1.. ElementNum]({Node1, Node2, Node3, Node4});

Предлагаемый алгоритм прореживания конечноэлементной сетки включает следующие этапы:

1. Составление списка взвешенных ребер и списков инцидентности ребер элементам.
2. Разбиение расчетной области на части для выполнения прореживания в параллельном режиме.
3. Прореживание сетки.

2. Составление списка взвешенных ребер

Предлагаемый процесс прореживания сетки заключается в последовательном переборе всех элементов с принятием решения об удалении конкретного элемента. Элемент можно удалить, если при этом не будет удалено ни одного ребра сетки. Таким образом, возникает необходимость в создании списка взвешенных ребер и списка инцидентности ребер элементам.

Общий вид класса ребер:

```
class Rib
{
 unsigned int node1; // Узел ребра с меньшим номером
 unsigned int node2; // Узел ребра с большим номером
 vector<unsigned int> BTcells; // Массив элементов, инцидентных ребру
 void AddCell(unsigned int cell); // Добавление элемента с номером cell
 bool operator==(const Rib& with)const;
 bool operator!=(const Rib& with)const;
 bool operator<(const Rib& with)const;
};
```

Необходимо уметь сравнивать ребра разных элементов, а значит нужно будет единственным образом их идентифицировать, поэтому класс ребра определен как класс с отношением эквивалентности (operator==() и operator!=()).

При решении вопроса, следует ли удалить элемент или оставить его в составе сетки необходимо знать веса всех его ребер, для чего необходимы списки инцидентности ребер элементам.

Процесс построения списка взвешенных ребер и списка инцидентности ребер элементам состоит из трех шагов:

1. При переборе всех элементов его ребра либо добавляются в список ребер с указанием инцидентности текущему элементу, либо, если они уже в нем есть, помечается, что данное ребро также принадлежит еще одному элементу. Фактически создается список инцидентности элементов ребрам.
2. Перебирается список ребер. Сохраняются лишь количество элементов, которым принадлежит ребро. Заполняются списки инцидентности ребер элементам.
3. Удаляется список инцидентности элементов ребрам.

При создании списков инцидентности элементов ребрам необходимо хранить небольшие массивы, содержащие номера элементов, инцидентных конкретному ребру. Поскольку в конце процесса от списка ребер остаются только веса (что означает удаление всех объектов Rib), то эту временную информацию естественно разместить в классе Rib – этой цели служит vector<unsigned int> BTcells. Также в класс добавлен метод AddCell(), с помощью которого отмечается инцидентность ребра очередному элементу.

В первом варианте программы временный список инцидентности элементов ребрам был реализован с помощью контейнера vector<Rib>. При тестировании было замечено, что построение этого временного списка происходит нелинейно по времени. Это

является следствием того, что в неупорядоченном массиве сложность поиска равна O(N), где N – количество ребер. При этом количество поисков пропорционально N. Таким образом, сложность первого шага (построения взвешенного списка ребер) составляет $O(N^2)$.

Чтобы избавиться от излишней сложности, было принято решение заменить контейнер для хранения ребер на set<Rib>. Контейнер set описывает автоматически упорядочивающийся массив, реализованный внутри как красно-черное дерево с постоянной сложностью добавления и сложностью поиска O(ln(N)) ([7, 8]). Чтобы использовать контейнер set, пришлось ввести в запись Rib лексикографическое отношение порядка (operator<()). Данный подход обеспечивает сложность построения временного списка инцидентности элементов ребрам O(Nln(N)), однако замедляет его удаление.

Хотя в данной работе не рассматривался вопрос ускорения выполнения третьего шага, связанного с удалением списка инцидентности элементов ребрам, представляется, что соответствующее время можно минимизировать с помощью ручного управления памятью (перегружая new и delete), то есть можно распределять память под хранение Rib в специально создаваемой программой куче и удалять эту кучу целиком.

3. Прореживание. Однопоточный и параллельный алгоритмы

Когда создан список взвешенных ребер и для каждого элемента известно, какие ребра ему принадлежат (имеется в виду вновь полученная единая нумерация ребер), сам процесс прореживания выполняется путем перебора всех элементов и проверки весов принадлежащих ему ребер – если все они больше единицы, то элемент помечается, как удаленный, а веса его ребер уменьшаются на единицу. Этот процесс многократно повторяется до тех пор, пока в очередной итерации не будет удалено ни одного элемента.

Для процедуры удаления элементов реализовано распараллеливание путем разбиения области на несколько частей и запуска в каждой из них отдельного потока, выполняющего прореживание. При этом возникает проблема совместного доступа к элементам, находящимся на границе двух областей, так как к элементам границы только один поток в течении одного кванта времени должен иметь доступ. Поэтому элементы границы областей записываются отдельно, так как только для них используется синхронизация доступа при прореживании. В элементах внутренних частей областей доступ не требует межпоточного согласования. Для синхронизации используются кроссплатформенные мьютексы библиотеки Qt.

Если используется больше двух потоков, возникает вопрос о количестве простаивающих потоков при совместном доступе к границам. Было решено использовать такую конфигурацию областей, при которой захват мьютекса будет означать, что освобождения этого мьютекса может ожидать максимум еще один поток - топологически такая конфигурация напоминает сферические слои. Любая граница находится только между двумя областями, к каждой из которых приписано по потоку. При этом разумно каждой области приписать две границы – внешюю и внутреннюю (с точки зрения сферических слоев), и к каждой из этих границ приписать по отдельному мьютексу. Так минимизируются издержки на согласование потоков. Общий вид области таков:

```
struct CellPart
{
 vector<unsigned int> cells;              // Внутренняя область
 vector<unsigned int> bound1;  // Внутренняя граница
 vector<unsigned int> bound2;  // Внешняя граница
};
```

Такая разбивка на подобласти легко достигается с помощью фронтообразного перебора области (рис. 1): выбирается произвольный элемент, затем перебираются все его соседи, после чего перебираются их внешние соседи, затем их внешние соседи и так далее до исчерпания всей области.

Организация подобной последовательности доступа производится с помощью структуры данных «очередь» и меток, сигнализирующих, что данный элемент уже был посещен в процессе перебора. Подобный перебор выглядит как расширяющийся сферический фронт, откуда и возникло его название.

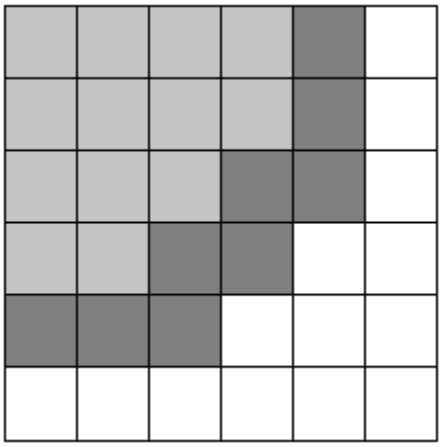

Рис. 1. Сбор элементов в подобласть

Сама разбивка производится следующим образом: во фронтообразном переборе в очередную подобласть вносится элементы. Как только сумма числа выбранных в подобласть и имеющихся в очереди элементов становится равной требуемому числу элементов в подобласти, мы помечаем уже выбранные элементы как внутренние для подобласти, а оставшиеся в очереди - как внешнюю границу. Граница выбирается исчерпанием очереди с одновременным наполнением новой очереди, состоящей из новых соседей-элементов. Эта новая очередь станет внутренней границей новой подобласти. Структура подобластей и границ схематично изображена на рис. 2.

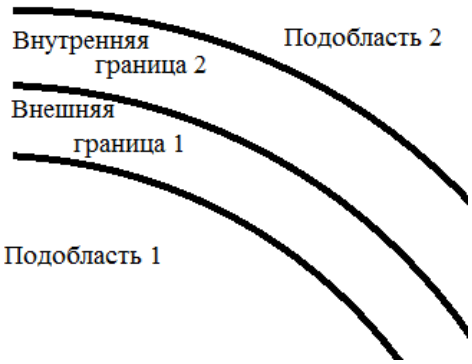

Рис. 2. Структура подобластей и границ

4. О тестировании

Программа тестировалась на нескольких сетках. При числе элементов сетки порядка десятков тысяч время работы программы составляет порядка минуты. Для сеток, содержащих порядка миллиона конечных элементов, время работы составляет десятки минут. Замечено, что самыми затратными по времени этапами были чтение/запись и процесс удаления списка инцидентности элементов ребрам. Этот этап занял около половины времени работы программы, что мотивирует на дальнейшее совершенствование кода.

Количество удаляемых элементов показывает, что ажурная схема способна уменьшать количество вычислений примерно втрое.

В то же время, тесты на параллельном алгоритме не оправдали использования параллелизма, потому что этап прореживания и без того оказался практически моментальным, а вследствие того, что его сложность по времени – линейная, очевидно, что он будет выполнятся быстро на любых сетках. С другой стороны, рассмотренный подход организации совместного доступа может быть использован для эффективного распараллеливания изначально медленных задач, например, расчетных программ МКЭ.

Литература

1. Чекмарев Д.Т. Ажурные схемы метода конечного элемента // Прикладные проблемы прочности и пластичности. Нижний Новгород: ННГУ, 1997. Вып. 55. С. 157-159.
2. Чекмарев Д.Т. Численные схемы метода конечного элемента на «ажурных» сетках // Вопросы атомной науки и техники. Сер. Математическое моделирование физических процессов. 2009. Вып. 2. С. 49-54.
3. Баженов В.Г., Чекмарев Д.Т. Об индексной коммутативности численного дифференцирования // Ж. вычисл. математики и мат. физики. 1989. Т. 29. № 5. С. 662-674.
4. Баженов В.Г., Чекмарев Д.Т. Вариационно-разностные схемы в нестационарных волновых задачах динамики пластин и оболочек. Н.Новгород: Изд-во Нижегород. ун-та, 1992. 159 с.
5. Баженов В.Г., Чекмарев Д.Т. Решение задач нестационарной динамики пластин и оболочек вариационно-разностным методом. Нижний Новгород: ННГУ, 2000. 118 с.
6. Чекмарев Д.Т., Кастальская К.А. О построении трехмерных ажурных сеток // Труды XII Межд. семинара «Супервычисления и математическое моделирование». Саров. 2011. С.374-381.
7. Джосьютис Н. C++ Стандартная библиотека. Для профессионалов. СПб.: Питер, 2004. – 730 с.
8. The C++ Resources Network. – http://www.cplusplus.com/.
9. Земсков Ю.В. Qt 4 на примерах. СПб.: БХВ-Петербург, 2008. – 608 с.