Оглавление

Введение

Настоящая книга выходит в рамках серии «Суперкомпьютерное образование». Основной целью серии является представление подходов к созданию суперкомпьютерных технологий и опыта их эксплуатации для некоторых важных приложений. В одном из изданий серии [79] представлены глобальные модели атмосферы, океана, пограничного слоя атмосферы, разработанные в Институте вычислительной математики РАН, и кратко описывается единственная в России модель земной системы (атмосфера — океан — морской лед — процессы на поверхности суши — углеродный цикл — атмосферная химия). Там же рассматриваются особенности параллельных реализаций данных моделей, исследована параллельная эффективность программных комплексов моделей на сотнях и тысячах процессоров, представлены результаты численного моделирования. В другом издании [80] рассматривается использование суперкомпьютерных вычислений в двух прикладных задачах дискретной математики. Первая задача требует решения сверхбольших линейных систем над полем GF(2) и имеет непосредственное отношение к проблеме «взлома» криптографических кодов, а вторая задача, связанная с проблемой передачи данных через канал с шумом, требует построения и анализа кодов малой плотности проверки на четность (LDPC).

В настоящем учебном пособии представлен наш опыт создания программной платформы и графической среды для разработки параллельных численных моделей на сетках общего вида. Важность таких сеток в приложениях будет обсуждаться ниже. Основной целью книги является разбор

инструментария, используемого при разработке параллельных приложений. Ключевой особенностью авторского подхода является формирование инструментария для *суперкомпьютерного моделирования*. На основе этого инструментария разработчики прикладного программного обеспечения (например, Систем Инженерного Анализа, СИА) могут легко учитывать многопроцессорность компьютерной архитектуры, поскольку особенности межпроцессорных обменов — как локальных, так и глобальных, скрыты от разработчика, и вызовы коммуникаций имеют простейшую форму. Область применения излагаемого инструментария нам видится, прежде всего, в разработке и внедрении новых параллельных приложений в инженерных и научных расчетах. Целевую аудиторию книги составляют разработчики СИА, инженеры и математики-вычислители, деятельность которых связана с суперкомпьютерным моделированием, сюда входят те, кто непосредственно создает параллельные приложения или использует параллельные численные модели.

Главной задачей описываемой программной платформы и графической среды является обеспечение пользователя *всеми* необходимыми средствами для создания и исследования различных численных моделей. Сюда входит не только работа с распределенными по процессорам сеточными данными для сеток общего вида, но и удобный интерфейс для формирования систем линейных уравнений, а также дальнейшего их решения, например, с помощью общедоступных пакетов линейной алгебры (Trilinos, PETSc и др.) на имеющихся параллельных вычислительных системах. Более того, технологический комплекс включает в себя и инструментарий для создания графического пользовательского интерфейса, позволяющий контролировать все этапы технологической цепочки математического моделирования, задавать параметры и коэффициенты задачи, визуализировать и анализировать полученное сеточное решение. В дальнейшем для обозначения технологического комплекса, состоя-

щего из программной платформы и графической среды, мы будем использовать аббревиатуру INMOST (Integrated Numerical Modelling and Object-oriented Supercomputing Technologies).

Отметим, что существуют программные решения, частично приспособленные для тех же целей, что и комплекс INMOST. Настойчивые попытки авторов использовать эти решения не увенчались успехом по разным причинам. В некоторых библиотеках невозможно, либо трудно внедрить свои схемы дискретизации, многие оказались недостаточно надежными с точки зрения устойчивой работы, другие — малоэффективными, а некоторые еще до конца не реализованными. Обзор существующих решений с опорой на наш опыт представлен в первых разделах каждой главы данной книги. Пособие ни в коей мере не претендует на исчерпывающий обзор и теоретический анализ всех технологических решений для разработчиков параллельных приложений в математическом моделировании. Изучить эти вопросы читатель может по монографиям [56, 77, 78] и трудам специализированных отечественных [70, 71, 72] и международных конференций [68, 69], а также используя специализированные Интернет-ресурсы [73, 74, 75, 19].

Перечислим кратко основные достоинства разработанной нами программной платформы и графической среды комплекса INMOST.

Важнейшим преимуществом комплекса является максимально широкий класс используемых расчетных сеток: конформные сетки с многоугольными или многогранными ячейками. Конформность сетки подразумевает, что любые две ячейки либо не пересекаются, либо имеют ровно одну общую вершину, либо одно целое ребро, либо одну целую грань (в трехмерном случае). Многогранность ячеек допускает наличие ячеек разных типов, начиная с тетраэдра и кончая многогранником с большим числом граней. Фактически, даже неконформные сетки могут рассматриваться как конформ-

ные многогранные: например, сетки типа восьмеричное дерево (рис. 1) являются неконформными, если полагать, что их ячейки кубические, однако эти же сетки являются конформными, если их ячейки — многогранники, в которых некоторые грани могут лежать в одной плоскости.

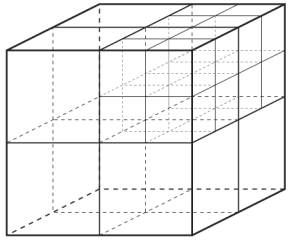


Рис. 1. Сетка типа восьмеричное дерево

Важность применимости платформы к сеткам общего вида диктуется также вопросами эффективности научного или инженерного расчета. С одной стороны, для заданного облака узлов (например, узлов кубической решетки), сеткой с минимальным числом ячеек является гексаэдральная сетка, в то время как тетраэдральная сетка содержит ячеек как минимум в 5 раз больше. Использование равномерных кубических сеток непрактично, поскольку исключает возможности локальной адаптации и аккуратного разрешения границ расчетной области. Существенно уменьшить количество рабочих ячеек позволяет использование сеток, допускающих адаптацию граней к границе моделируемого объекта. С другой стороны, в произвольных трехмерных областях только неструктурированные тетраэдральные сетки могут быть построены и адаптированы автоматически более или менее надежно [26, 56, 76], однако в анизотропных областях тетраэдральные ячейки могут иметь очень тупые двугранные углы. Для расчетных областей, характерных для различных приложений, разнятся и оптимальные типы

ячеек, которые можно построить надежно и в наименьшем количестве для заданного шага сетки. Иногда на практике удобно использовать гибридные сетки, в которых тип ячеек меняется в разных подобластях в зависимости от возможности применения той или иной технологии построения расчетной сетки. Например, в окрестности сложной границы, заданной в САПР системе, можно построить тетраэдральную сетку, которая вдали от сложной границы переходит в гексаэдральную сетку. Все перечисленные выше сетки относятся к классу конформных многогранных сеток. На рис. 2 приведены некоторые представители этого класса.

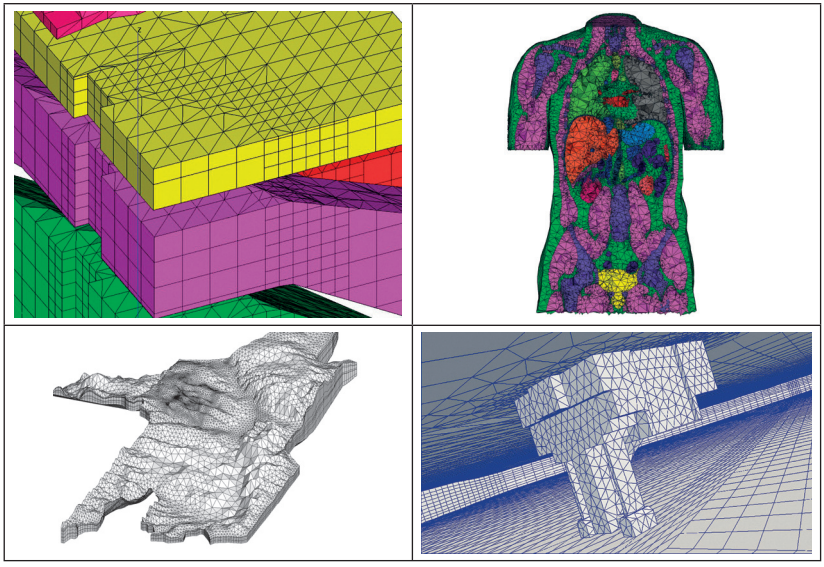


Рис. 2. Примеры конформных многогранных сеток

Отметим, что именно требования инженерных приложений привели нас к созданию новой программной платформы для разработки параллельных численных моделей на сетках *общего вида*.

Потребности современного моделирования таковы, что мощностей персональных однопроцессорных или даже многоядерных компьютеров оказывается недостаточно для получения требуемого результата, и моделирование на высокопроизводительных параллельных вычислительных комплексах не имеет альтернативы. При этом использование MPI (Message Passing Interface) для организации межпроцессорных обменов неизбежно. Второе важное достоинство программной платформы комплекса INMOST — ее параллельная составляющая, основанная на MPI-обменах для систем с распределенной памятью. При необходимости проведения расчетов на гибридных архитектурах возможно увеличение количества MPI-процессов до количества вычислительных ядер на каждом из вычислительных узлов с общей памятью.

Третье достоинство платформы — гибкость ее структуры сеточных данных, которая обеспечена функциональностью языка программирования C++. Гибкость структуры данных нужна для минимизации объема хранимой информации и линейности алгоритмов обработки данных.

Четвертое достоинство платформы — ее переносимость на основные операционные системы, Linux и Windows, используемые в научных вычислениях. Простота и удобство ее использования сочетается с гибкостью и универсальностью, а также надежностью и эффективностью.

Наконец, технологический комплекс INMOST предлагает удобный инструментарий для создания графической среды, в которой все этапы суперкомпьютерного моделирования могут контролироваться и управляться пользователем.

Перейдем к краткому обзору содержания книги.

В главе 1 представлена специальная сеточная база данных программной платформы комплекса INMOST, являющаяся основным ядром для работы с сетками, состоящими из многогранных ячеек. Здесь обсуждаются основные принципы работы сеточных баз данных в последовательном и параллельном режимах работы. Глава начинается с изложения принципов,

которые должны быть положены в основу современной программной платформы для разработки параллельных численных моделей на сетках общего вида. Затем мы приводим обзор и анализ современных подходов и существующих программных средств, которые могут быть использованы для этих целей. Из анализа следует необходимость разработки новой программной платформы, которая и описывается в последующих разделах главы. При этом вводится понятие базовых сеточных элементов: узел (или вершина), ребро, грань, ячейка. Рассматриваются как элементы общего вида, так и некоторые виды ячеек специального геометрического типа (тетраэдр, гексаэдр и т. д.). Описываются функции для работы с сеточными данными, включая ярлыки, атрибуты объектов и сами физические величины. Для удобства работы с множествами применяются различные виды итераторов. Приводятся наборы функций, необходимых для работы с сетками, а также обсуждаются алгоритмы их эффективной реализации. Приведенные описания могут быть использованы как руководство для создания сеточных базы данных. Далее в главе рассматривается механизм работы с распределенными по процессорам сеточными данными. Обсуждаются особенности работы с собственными и фиктивными ячейками. Приводится минимальный набор функций, достаточный для выполнения базовых операций, а также функции, обеспечивающие максимально широкий функционал возможностей для работы опытного пользователя. В конце главы рассматривается удобный программный интерфейс для формирования и решения систем линейных уравнений. Предлагается два подхода к решению систем на параллельных вычислительных системах: на основе общедоступных пакетов линейной алгебры и на основе собственной библиотеки параллельных решателей.

Глава 2 посвящена средствам разработки графической среды, которая может быть использована в численном моделировании на современных суперкомпьютерах. Формулировка требований, накладываемых на подобные средства разработ-

ки, дает возможность провести обзор существующих технологических решений и их анализ в свете этих требований. Описание набора программных инструментов для создания графической среды и графического интерфейса пользователя составляет основное ядро главы, которая заканчивается обсуждением особенностей реализации при разработке параллельных и массивно параллельных приложений.

В главе 3 представлен опыт применения технологического комплекса INMOST при разработке параллельной программы расчета геофильтрации и геомиграции радионуклидов GeRa, выполненной Институтом вычислительной математики РАН совместно с Институтом проблем безопасного развития атомной энергетики РАН в рамках проекта «Прорыв». Мы рассматриваем важные аспекты разработки всех этапов построения параллельной вычислительной технологии — от задания расчетных областей с помощью графического интерфейса до анализа полученного сеточного решения. Численные расчеты на основе программы GeRa проводились Кириллом Дмитриевичем Никитиным и Иваном Викторовичем Капыриным. Используемый в GeRa генератор расчетных треугольно-призматических сеток написан Андреем Валерьевичем Пленкиным, Вадимом Николаевичем Чугуновым и Александром Анатольевичем Даниловым. Задание границ пластов на основе геостатистических данных обеспечено Александром Владилиновичем Расторгуевым, Еленой Александровной Савельевой и Валерием Андриановичем Ивановым. Авторы глубоко признательны коллегам за предоставленные примеры и за ценный вклад в написание главы 3. Разработка платформы INMOST была поддержана проектом Росатома «Прорыв», Программой Президиума РАН «Алгоритмы и математическое обеспечение для вычислительных систем сверхвысокой производительности», грантами РФФИ 11-01-00971, 12-01-00283, 12-01-31275, 12-01-33084, ФЦП «Научные и научно-педагогические кадры инновационной России».

Глава 1

Программная платформа для разработки параллельных численных моделей

В этой главе рассматривается первая составляющая технологического комплекса INMOST — программная платформа, обеспечивающая работу с распределенными по процессорам сеточными данными. Сначала стоит сказать несколько слов об актуальности создания специализированной сеточной базы данных для работы с сетками, состоящими из многогранных ячеек.

Современные задачи математического моделирования требуют использования все более и более качественной аппроксимации. Использование прямоугольных декартовых сеток для необходимого качества аппроксимации решения потребовало бы столь большого количества ячеек, что мощностей современных компьютеров было бы явно недостаточно. Несколько улучшить ситуацию могло бы применение высококачественных схем дискретизации, однако это бы не решило проблему полностью. Существенно уменьшить количество рабочих ячеек позволяет использование сеток общего вида с адаптацией к границе моделируемого объекта.

С другой стороны, потребности современного моделирования таковы, что мощностей персональных однопроцессорных или даже многоядерных компьютеров оказывается недостаточно для получения требуемого результата. Необходимо использование высокопроизводительных параллельных вычислительных систем.

Создание комплексов программ, которые могут выполнять расчеты на таких параллельных компьютерах является достаточно трудоемкой задачей. При переходе от последовательных к параллельным программам требуется не только

добавлять в последовательную программу обмены данных между процессорами, но и значительно перестраивать всю структуру используемых данных. Для помощи в распараллеливании программ математического моделирования предлагается создание программной платформы, являющейся основой для всех этапов параллельного расчета: построения расчетных сеток, дискретизации краевой задачи на построенных сетках, а также для решения систем линейных уравнений, получающихся в результате этой дискретизации.

1.1. Требования к программной платформе

В качестве программной платформы может выступить библиотека протестированных вспомогательных программных средств, которая обеспечит необходимый функционал базовых операций. Одно из основных требований к платформе — обеспечение возможности работы как в последовательном, так и в параллельном режимах. Желательно, чтобы для разработчика переход от написания последовательных программ к параллельным происходил с минимальными усилиями.

Чтобы понять, из каких модулей будет состоять библиотека программных средств (программная платформа), рассмотрим последовательность действий разработчика, а также необходимые для моделирования базовые операции.

1.1.1. Стандартная технологическая цепочка

На примере стационарной краевой задачи рассмотрим стандартную технологическую цепочку и этапы моделирования, из которых она состоит:

- построение расчетной сетки;
- распределение сеточных данных по процессорам;
- построение дискретизации;
- формирование матрицы и правой части системы уравнений;

- решение линейной системы;
- визуализация результатов моделирования.

Процесс построения расчетной сетки напрямую зависит от геометрии области и в большой степени ею определяется. Распределение данных по процессорам может происходить как после построения сетки (например, когда расчетная сетка построена заранее и задана вне процесса моделирования), так и до него. При этом для успешного завершения построения расчетной сетки, необходимо, например, построить некоторую грубую начальную сетку, распределить ее по процессорам, а затем проводить уточнение и доразбиение сетки уже в параллельном режиме на всех имеющихся процессорах. Программная платформа должна обеспечить возможность распределенного хранения и использования сеточной информации.

Набор этапов, связанных с дискретизацией, включает в себя выбор метода дискретизации (метод конечных элементов или метод конечных объемов), выбор схемы дискретизации, формирование матрицы правой части системы уравнений, а также решение этой системы. Эти этапы будут повторяться на каждом шаге по времени (при решении нестационарной краевой задачи) или на каждом шаге метода Ньютона (при решении нелинейной задачи).

Метод визуализации полученных результатов зависит от предпочтений пользователя программной платформы, поэтому должна быть обеспечена возможность выбора различных средств визуализации посредством записи результатов расчета в файлы данных общепринятых форматов. При этом визуализация может проводиться как в процессе расчета, так и после сохранения результата расчета для проведения анализа результатов в дальнейшем.

Стоит заметить, что точность полученного решения напрямую зависит от построения сетки (в частности, от шага сетки вблизи границ или особенностей решения), а также от выбора метода дискретизации (зависящего от физической сущности моделируемых процессов).

1.1.2. Диаграмма зависимости модулей

Рассмотрим участвующие в моделировании объекты, а также программные модули:

- Сеточные данные (объект);
- Платформа (программный модуль);
- Линейный решатель (программный модуль);
- Визуализация (программный модуль).

Потоки данных между этими модулями при решении стационарной краевой задачи могут быть представлены следующей схемой:

Сеточные данные $\rightarrow$ Дискретизация
Сеточные данные $\rightarrow$ Платформа
Платформа $\leftrightarrow$ Дискретизация
Платформа $\leftrightarrow$ Линейный решатель
Платформа $\rightarrow$ Визуализация

Для нестационарной краевой задачи взаимосвязи в основном останутся теми же, только обратная связь

Платформа $\leftarrow$ Дискретизация

будет работать более активно.

Для динамических сеток появится новое направление взаимодействия

Сеточные данные $\leftarrow$ Платформа

причем для слабо меняющихся (редко перестраиваемых) сеток перестроение сеточных данных не обязано быть очень быстрым, а для сильно меняющихся сеток сами сеточные данные должны быть способны изменяться динамически (например, добавление или удаление одной ячейки, не должно занимать время порядка N, где N — количество ячеек сетки, а быть порядка 1 или, по крайней мере, не более $\log(N)$).

1.1.3. Формирование программной платформы

Формирование программной платформы должно происходить по результатам анализа существующих на данный момент программных средств, библиотек программ, реализующих рассмотренные модули. Основными критериями отбора являются:

- соответствие требуемому функционалу возможностей,
- эффективность,
- надежность работы,
- универсальность,
- простота использования,
- открытость исходного кода.

Задача поиска осложняется тем, что в настоящее время трудно найти комплексы программ, удовлетворяющие сразу всем требованиям на платформу. Поэтому естественным является выбор наиболее подходящих программных средств, объединение их в единую библиотеку и разработка недостающих компонентов платформы.

1.2. Анализ современных подходов и существующих программных средств

Для обоснования выбора программных средств необходимо провести их анализ вместе с анализом современных подходов. Для этого были рассмотрены как уже готовые свободно распространяемые программные решения, так и отдельные библиотеки программ.

Рассмотрим некоторые пакеты, обеспечивающие работу с распределенными сеточными данными.

1.2.1. Библиотека FMDB

Библиотека FMDB (Flexible distributed Mesh DataBase) [1] — сеточная библиотека, предназначенная для работы с распределенными динамическими сетками. FMDB под-

держивается программой SciDAC (Scientific Discovery through Advanced Computing) Департамента энергетики США, является частью проекта ITAPS (Interoperable Tools for Advanced Petascale Simulations) и поддерживает основную функциональность его сеточных операций. Для обмена сообщениями использует стандарт MPI.

К сожалению, сеточная библиотека FMDB не поддерживает работу с произвольными многогранниками. Кроме того, из-за использования стандартных библиотек, специфических для отдельных Linux-систем, компиляция осложнена даже на Unix-совместимых операционных системах. Проведенные эксперименты также показали, что функция обмена слоями фиктивных элементов, необходимая для работы с распределенными данными, выдает ошибку сегментации, а часть других функций не реализована.

1.2.2. Библиотека MOAB

Библиотека MOAB (A Mesh-Oriented datABase) [2] — сеточная библиотека с поддержкой всех необходимых базовых операций для распределенных сеточных данных. MOAB позволяет работать как со структурированными, так и с неструктурированными сетками, поддерживает все виды ячеек, включая произвольные многоугольники и многогранники.

Библиотека MOAB поддерживает ITAPS iMesh интерфейс, который является общим интерфейсом для нескольких различных пакетов, включая MOAB. Этим достигается возможность использования различных сеточных пакетов без изменения кода. Правда, за такую возможность часто приходится платить падением эффективности вычислений.

В основе библиотеки лежит идея сеточных баз данных, библиотека ориентирована на минимальное использование памяти компьютера, что негативно сказывается на эффективности таких базовых операций, как запрос смежных элементов или соседей данного элемента. Эксперимент

с распределенными сеточными данными показал, что существует множество сеточных топологий, которые не разрешаются данной библиотекой. Было также обнаружено, что упаковка и распаковка данных при обменах занимает неприемлемо длительное время.

1.2.3. Библиотека MSTK

MSTK (MeSh ToolKit) [3] — сеточная библиотека для работы с распределенными неструктурированными сетками. Разрабатывается небольшим коллективом авторов в Лос-Аламосской национальной лаборатории для Департамента энергетики США.

Сетка MSTK может содержать следующие элементы: узлы, ребра, грани, ячейки. Полностью поддерживается связность между этими элементами. К каждому элементу могут быть прикреплены данные, состоящие из целых или вещественных чисел, а также указатели. В настоящее время MSTK поддерживает все необходимые базовые операции, а также возможность работы с несколькими сетками одновременно, при этом каждый элемент может принадлежать только одной сетке.

Изменение сетки в процессе моделирования пока не допускается. Эту возможность планируется добавить в дальнейшем. К сожалению, в данный момент функции работы с распределенными сетками еще до конца не реализованы, а функция обмена фиктивными слоями элементов поддерживает только один слой фиктивных элементов, что является неприемлемым ограничением для построения качественных дискретизаций.

1.2.4. Библиотека STK

STK (Sierra ToolKit) [4] — сеточная библиотека, включенная в состав свободно распространяемого пакета Trilinos [12]. Библиотека STK поддерживает работу с распределенными,

неоднородными, а также динамически изменяющимися неструктурированными сетками.

Библиотека STK позволяет работать с произвольными многогранниками, поддерживает изменение топологии сетки, различные подмножества сетки. К элементам сетки могут быть приписаны коэффициенты и сеточные данные. Отдельно выделены функции, поддерживающие сборку матриц жесткости и векторов правых частей для дальнейшего решения полученных линейных систем в рамках пакета Trilinos.

К сожалению, библиотека STK, как и ничем не связанный с ней (кроме схожести названия) пакет MSTK, поддерживает только один слой фиктивных элементов. В настоящее время библиотека STK находится в разработке, достаточно слабо документирована.

1.2.5. Пакет Salome

Salome [5] — открытая интегрируемая платформа для численного моделирования. Представляет собой конечно-элементный пре- и постпроцессор.

Первоначально задуманная как связующее программное обеспечение систем инженерного анализа, Salome объединяет в себе различные модули, применяемые в приложениях численного моделирования. Средства САПР в Salome имеют достаточно тесную связь с платформой Open CASCADE Technology. Все программные модули Salome первоначально разрабатывались рядом французских компаний и институтов для проектирования атомных станций. Сейчас платформа Salome используется как база для проекта NURESIM (European Platform for NUclear REactor SIMulations), который предназначен для полномасштабного моделирования ядерных реакторов.

В основе Salome, прежде всего, лежит концепция объектно-ориентированного программирования. Платформа предоставляет собой набор удобных средств построения сеток и блочного построения схем. Salome также позволяет

разрабатывать собственные программные решения. Пакеты, составляющие Salome, обладают мощными средствами анализа результатов расчета. При этом некоторые расчетные модули Salome не являются открытыми. К сожалению, сеточный генератор содержит много ошибок, класс поддерживаемых сеток узок, в частности, он не позволяет работать с сетками большого размера. Кроме того, для поддержки распределенных вычислений используется интерфейс CORBA, который существенно замедляет обмены.

1.2.6. Пакет OpenFOAM

OpenFOAM (Open Source Field Operation And Manipulation CFD ToolBox) [6] — открытая интегрируемая платформа для численного моделирования задач механики сплошных сред. OpenFOAM является свободно распространяемым инструментарием вычислительной гидродинамики для операций со скалярными, векторными и тензорными полями.

Код OpenFOAM разработан в Великобритании в компании OpenCFD и используется многими промышленными предприятиями более 12 лет. Свое название и идеологию построения код берет от предшественника FOAM (Field Operation And Manipulation), который является закрытым и продолжает развиваться параллельно с OpenFOAM. Первоначально программа предназначалась для прочностных расчетов и в результате многолетнего академического и промышленного развития на сегодняшний момент позволяет решать задачи прочности, гидродинамики, теплопроводности и др.

В основе кода лежит набор библиотек, предоставляющих инструменты для приближенного решения систем дифференциальных уравнений в частных производных, как в пространстве, так и во времени. Рабочим языком кода является язык C++. Имеется псевдоязык записи компонентов дифференциальных уравнений, что позволяет пользователю добавлять новые компоненты уравнений. В пакет входит постпроцессор ParaView [33, 34], позволяющий просматривать

как сгенерированные сетки, так и результаты расчетов, а также строить графики, выполнять анимацию расчетов. Возможна интеграция с пакетом Salome.

Пакет представляет собой множество готовых моделей, содержит встроенный язык для построения систем обыкновенных дифференциальных уравнений, а также сеточные генераторы и возможность достаточно эффективно производить расчеты на распределенных вычислительных системах. Однако при изучении пакета OpenFOAM возникли существенные трудности при внедрении собственных схем дискретизации.

1.2.7. Другие сеточные пакеты

Существуют и другие сеточные пакеты, способные работать с распределенными неструктурированными сеточными данными. Среди таких пакетов можно упомянуть ParFUM, Triangle, ALPS. Некоторые пакеты выполняют параллельное уточнение распределенных сеток, среди них Racoon, AMR. Другие пакеты поддерживают некоторые функции для работы с распределенными сетками: PAOMD, PMDB.

Однако функциональность этих пакетов оказывается недостаточной для построения качественных дискретизаций.

1.2.8. Распределение и перераспределение данных

Для решения задачи распределения сеточных данных созданы пакеты, позволяющие проводить как разбиение связанного графа на подобласти, так и динамическое переразбиение графа в зависимости от заданных весов в узлах и ребрах графа. Среди таких пакетов можно отметить ParMETIS, PT-Scotch, Zoltan.

1.2.8.1. ParMETIS

Пакет ParMETIS [7] реализует множество алгоритмов по распределению неструктурированных графов, сеток, для поиска упорядочивания разреженных матриц, уменьша-

ющего заполнение множителей треугольного разложения. ParMETIS использует библиотеку обменов MPI и расширяет функциональность пакета METIS [8], применительно к выполнению на параллельных компьютерах.

Алгоритмы, реализованные в ParMETIS, основываются на параллельном многоуровневом разбиении графа, адаптивном переразбиении и параллельной многопараметрической схеме разбиения. ParMETIS обеспечивает следующие действия: распределение графа, распределение сетки, перераспределение графа, уточнение распределения, упорядочивание матрицы. Поддерживается как C, так и Fortran интерфейс вызова функций.

1.2.8.2. PT-Scotch

Пакет PT-Scotch [9,10] является параллельной версией разработанного в Laboratoire Bordelais de Recherche en Informatique (LaBRI), Université Bordeaux, пакета Scotch, предназначенного для распределения графов, сеток и переупорядочивания разреженных матриц. PT-Scotch использует MPI интерфейс, а также, по желанию, POSIX threads параллелизм.

Особенностью данного пакета является его возможность достаточно быстро строить качественные распределения данных. В частности, в 2010 году был превышен 32-битовый рубеж, когда на 2048 процессорах была распределена сетка, состоящая более чем из 2,4 млрд узлов и 7,3 млрд ребер. Так же, как и в пакете ParMETIS, поддерживается C и Fortran интерфейс вызова функций.

1.2.8.3. Zoltan

Пакет Zoltan [11] выполняет все необходимые функции по распределению данных по процессорам, включая распределение графов и матриц по процессорам, динамическую балансировку загрузки процессоров, включая средства миграции данных. Пакет Zoltan является частью пакета Trilinos. Последняя версия разработанного в Sandia National

Laboratories пакета Zoltan доступна вместе с пакетом Trilinos [12]. В пакете Zoltan имеется поддержка вызовов сторонних пакетов, включая PT-Scotch и ParMETIS.

1.2.9. Выводы

Анализ существующих пакетов для работы с распределенными сеточными данными показал, что ни один из существующих пакетов в полной мере не соответствует предъявляемым требованиям. Опробованные авторами библиотеки MOAB и FMDB на момент тестирования не продемонстрировали заявленный функционал возможностей, необходимых для создания программной платформы. С другой стороны, библиотеки STK, MSTK, Salome и OpenFOAM, как следует из их документации, вообще не предоставляют такого функционала.

Подытожим, по каким причинам существующие готовые решения чаще всего не подходят для решения наших конкретных задач.

1. Нет переносимости между различными платформами (Windows, Linux). Чаще всего конечные пользователи работают на персональных компьютерах в операционной системе Windows, а параллельные расчеты для наиболее сложных задач необходимо проводить на суперкомпьютерах под управлением операционной системы Linux.
2. Недостаточно надежные (сырые) реализации. Попытки самостоятельно доработать пакеты или расширить их функциональность приводят к тому, что с выходом очередной версии пакета эти же доработки приходится делать заново. Недостаточно отлаженные пакеты могут существенно замедлить разработку программной платформы и поставить под сомнение дальнейшее ее использование.
3. В некоторых пакетах невозможно, либо проблематично внедрить свои схемы дискретизации. Адекватная дис-

кретизация, обеспечивающая аппроксимацию достаточно высокого порядка точности, является залогом эффективного приближенного решения краевой задачи. Необходимо дать разработчику возможность полностью контролировать процесс построения дискретизации, а задача программной платформы состоит в том, чтобы сделать этот механизм наиболее наглядным.

Заметим, что имеются пакеты программ, которые решают некоторые из наших частичных задач и полностью устраивают нас по своей функциональности и надежности. К таким пакетам следует отнести пакет ParMETIS, который производит распределение и перераспределение данных по процессорам, а также пакет PETSc, применяемый для решения систем линейных уравнений, который будет подробнее рассмотрен в разделе 1.5. На основании этого было решено самостоятельно разрабатывать набор программных средств, обеспечивающий весь необходимый функционал возможностей, с подключением упомянутых пакетов.

Следует отметить, что решение самой общей задачи моделирования не предполагается. Можно сформулировать несколько ограничений, которые позволят нам существенно упростить разработку программного комплекса.

1. Можно считать, что расчетная сетка уже предварительно сгенерирована. На данном этапе мы не ставим себе задачи распределенного построения сеток, оставляя возможность поддержки таких операций на будущее.
2. Предположим также, что получающиеся в результате дискретизации системы линейных уравнений могут быть достаточно большими, но умеренно сложными. Размер линейных систем не имеет критического значения, т. к. в качестве основного режима расчета предполагается использование современных многопроцессорных систем, включая супер-ЭВМ.

Заметим, что сформулированные предположения проистекают из свойств общепринятых подходов к построению численных моделей, и в их рамках нисколько не ограничивают условий применимости разрабатываемой программной платформы.

Перейдем теперь к описанию базового ядра платформы для работы с распределенными сеточными данными.

1.3. Базовые операции с сеточными данными

Базовые операции с сетками должны обеспечивать необходимый функционал для таких широко используемых для решения краевых задач методов, как методы конечных разностей, конечных объемов, конечных элементов. Все эти методы требуют дискретизации области разбиением ее на многогранные ячейки, формирующие расчетную сетку. Поэтому представление сеточных данных и операций над ними играет важную роль в технологиях приближенного решения краевых задач.

1.3.1. Сеточные элементы

В первую очередь определим структуру данных, которая обеспечит наиболее общее, и, тем не менее, эффективное представление сеточных данных, затем рассмотрим набор функций для операций с сеточными элементами и данными и расширим функционал базового ядра для работы с распределенными сетками.

Перечислим базовые сеточные *элементы* (рис. 3):

- узел сетки, который содержит информацию о своем положении в пространстве;
- ребро, полностью определяемое двумя узлами;
- грань, в общем случае многоугольник, опирающийся на множество ребер;
- ячейка, в общем случае многогранник, опирающийся на множество граней.

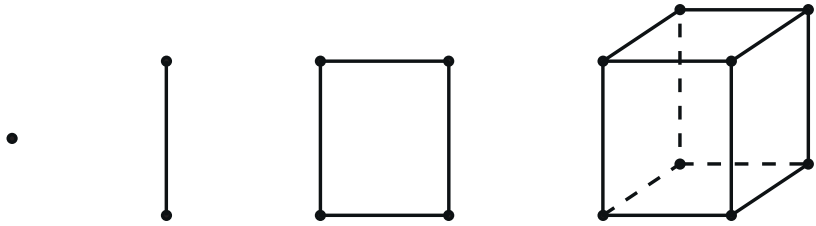


Рис. 3. Базовые сеточные элементы: узел, ребро, грань, ячейка

Если рассматривать частные случаи, например, кубические или тетраэдральные сетки, то можно ограничиться использованием только узлов и ячеек. Представление многогранных ячеек потребует добавления в структуру граней. Добавление ребер в структуру может быть, с одной стороны, оправдано в случае использования многогранных ячеек высокого порядка, другая причина использования ребер будет обсуждаться позднее. В итоге, наиболее полным можно считать набор элементов: узел, ребро, грань, ячейка.

Как в объектно-ориентированной программе, так и по тексту далее, будет удобно, если узел, ребро, грань и ячейка будут обозначаться одним понятием — элемент.

Для полного понимания геометрии тех или иных элементов требуется переход по иерархии вниз. То есть, для того, чтобы понять, какой является данная ячейка, необходимо узнать, из каких граней она состоит. Чтобы понять, какой является данная грань, необходимо узнать, из каких ребер она состоит, и т. д. В итоге возникает необходимость в установлении следующей диаграммы связей (рис. 4):

$$\text{Ячейка} \rightarrow \text{Грань} \rightarrow \text{Ребро} \rightarrow \text{Узел}.$$

В частных случаях, например, для кубов, тетраэдров и т. п. такое представление можно было бы сократить так (рис. 5):

$$\text{Ячейка} \rightarrow \text{Узел}.$$

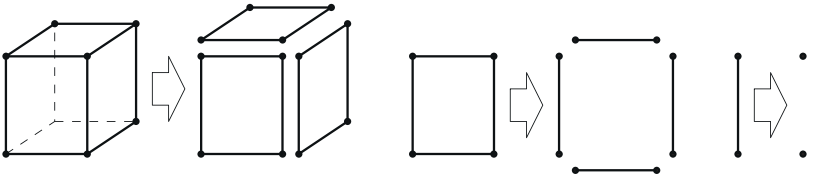


Рис. 4. Переход по иерархии вниз: ячейка, грань, ребро, узел

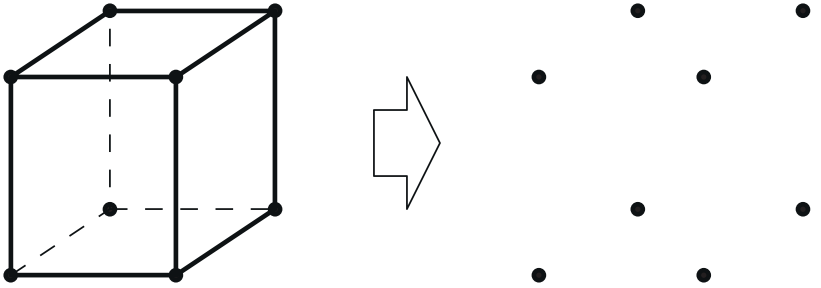


Рис. 5. Сокращенный переход по иерархии вниз: ячейка, узел

Это позволило бы сэкономить память компьютера, необходимую для хранения связей. Возможность поддерживать различные виды иерархии связей в зависимости от потребностей, а также сосуществование различных видов иерархий для различных элементов было бы большим плюсом, так как позволило бы использовать дополнительные элементы только там, где это необходимо и, тем самым, сэкономить память.

Мы могли бы также усложнить диаграмму связей, чтобы, к примеру, сразу можно было сказать, не только из каких граней, но и из каких ребер состоит ячейка, однако такого подхода следует избегать, так как это ведет к резкому увеличению требуемого объема памяти.

Для каждого элемента мы можем дополнительно задать некоторый геометрический тип и размерность:

- Вершина (размерность 0) — элемент, обладающий теми же свойствами, что и узел.

- Отрезок (размерность 1) — элемент, состоящий из двух узлов.
- Ломаная (размерность 2) — элемент, состоящий из нескольких ребер, которые не формируют замкнутое множество.
- Треугольник (размерность 2).
- Четырехугольник (размерность 2).
- Многоугольник (размерность 2).
- Полимногоугольник (размерность 3) — элемент, состоящий из нескольких граней, которые не формируют замкнутое множество.
- Тетраэдр (размерность 3).
- Куб (размерность 3).
- Призма (размерность 3).
- Пирамида (размерность 3).
- Многогранник (размерность 3).

Примеры элементов различных геометрических типов приведены на рис. 6.

Эти данные могут быть в дальнейшем использованы различными алгоритмами для оптимизации. Можно определить стандартный порядок узлов, ребер и граней для таких элементов, как треугольник, четырехугольник, тетраэдр, куб, призма, пирамида.

Выше речь шла о замкнутости элементов. Для граней можно определить замкнутость так, что при обходе всех ребер грани, каждый узел каждого ребра будет посещен дважды. Замкнутость для ячейки определим тем, что при обходе всех ее граней, каждое ребро каждой грани посещается дважды.

1.3.2. Связность сеточных элементов

При построении аппроксимации, например, конечно-объемным методом, требуется не только геометрическая информация о данном элементе, но также и информация о соседних элементах по отношению к данному. Если бы мы остановились на описанной выше иерархии, то запрос со-

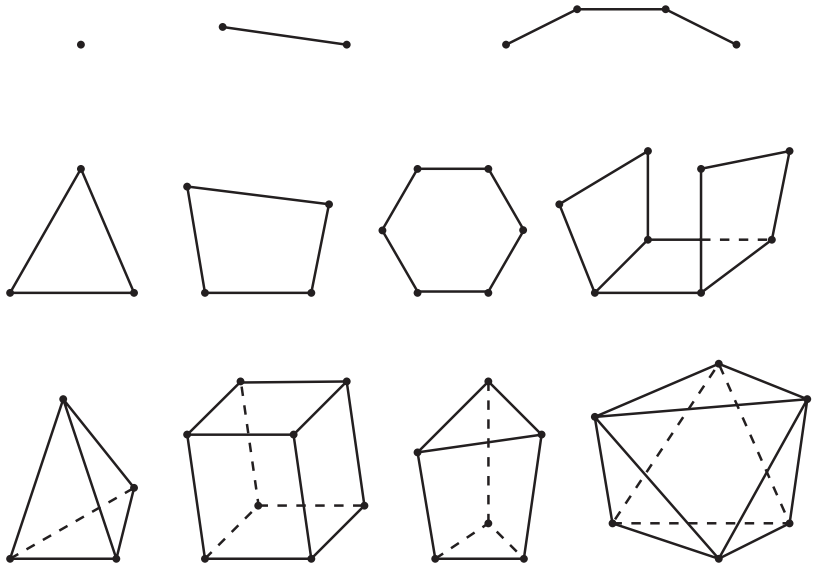


Рис. 6. Геометрические типы элементов: вершина, отрезок, ломаная; треугольник, четырехугольник, многоугольник, полимногоугольник; тетраэдр, куб, призма, многогранник

седних элементов по отношению к данному был бы алгоритмически сложной задачей. К примеру, чтобы узнать всех соседей данной ячейки через ее грани, пришлось бы пройти по всем ячейкам и проверить, содержат ли они среди своих граней хоть одну грань исходной ячейки. В итоге сложность алгоритма нахождения соседей для всех ячеек была бы $O(n^2)$, что является слишком накладным даже для относительно небольшой сетки в несколько десятков тысяч ячеек.

Для упрощения задачи мы можем расширить иерархию до следующей кольцевой диаграммы:

$$\text{Узел} \leftrightarrow \text{Ячейка} \leftrightarrow \text{Грань} \leftrightarrow \text{Ребро} \leftrightarrow \text{Узел}.$$

Это позволит быстро запрашивать соседние элементы по отношению к данному, проходя вверх по иерархии, и эле-

менты, из которых состоит данный элемент, проходя вниз по иерархии.

В итоге мы получим некоторый связный граф элементов. В некоторых случаях рассматривается направленность связей графа, например, исходя из порядка вершин в грани, выбирается направление нормали по отношению к этой грани и т. п. Сохранение направленности потребовало бы хранения элементов, которые различаются только порядком обхода своих элементов, что резко увеличило бы объем памяти. Мы не будем хранить направленность связей, однако, можно легко удостовериться в том, что подобную направленность можно восстановить исходя из иерархии элементов.

1.3.3. Упорядоченность сеточных элементов

В предыдущем пункте была затронута проблема, связанная с поиском дубликатов различных элементов сетки. Для этих целей можно ввести понятие упорядоченности всех элементов.

- Ввести порядок для узлов довольно просто. Пусть у нас есть два узла в двумерном пространстве с координатами (x_1, y_1) и (x_2, y_2). Если $x_1 < x_2$, то первый узел стоит раньше второго; если же $x_1 = x_2$, то если $y_1 < y_2$, аналогично первый узел стоит раньше, если же $y_1 = y_2$, то узлы являются дубликатами. Эту логику можно продолжить на n-мерное пространство.
- Далее, введем порядок для ломаных. Возьмем два ребра p_1 и p_2. Рассмотрим все узлы данных ребер $\{y_1\}$ и $\{y_2\}$, если количество узлов разное, то меньше то ребро, у которого узлов меньше. Иначе, пусть каждое ребро состоит из n узлов, тогда если первый узел $\{y_1\}_1$ стоит раньше последнего узла $\{y_1\}_n$, то будем рассматривать узлы в прямом порядке, иначе — в обратном. Обозначим порядок обхода как $\Pi(\{y_1\})$ и $\Pi(\{y_2\})$ для $\{y_1\}$ и $\{y_2\}$, соответственно. Далее сравним по порядку все узлы $\Pi(\{y_1\})$ и $\Pi(\{y_2\})$ в вы-

бранном порядке. Если $\Pi(\{y_1\})_i < \Pi(\{y_2\})_i$, то $p_1 < p_2$, если $\Pi(\{y_1\})_i = \Pi(\{y_2\})_i$, то переходим к следующему узлу. Если все узлы равны, то ребра p_1 и p_2 являются дубликатами. Таким образом, мы ввели упорядоченность для ребер.

- Введем порядок для граней. Возьмем две грани $г_1$ и $г_2$. Рассмотрим все ребра данных граней $\{p_1\}$ и $\{p_2\}$. Если количество ребер разное, то меньше та грань, у которой меньше ребер. Если количество одинаковое, то отсортируем по порядку ребра $\{p_1\}$ и $\{p_2\}$, используя описанную выше упорядоченность ребер. Далее на основе сравнения всех ребер $\{p_1\}$ и $\{p_2\}$ по порядку установим порядок граней $г_1$ и $г_2$.
- Аналогично можно ввести порядок для ячеек, сортируя грани данных ячеек и сравнивая их, основываясь на описанном ранее порядке для граней.
- Естественно также будет выстроить порядок для разных типов элементов следующим образом: узел должен быть меньше ребра, ребро меньше грани, а грань меньше ячейки.

Таким образом, мы ввели упорядоченность для ломаных и ребер. Для компьютера такая проверка порядка для сложных ячеек может оказаться довольно долгой, так как требует рекурсивной сортировки и проверки всех элементов ячейки вплоть до ее вершин. Поэтому для упрощения можно ввести понятие центроида элемента, как осреднение координат всех узлов, из которых он состоит. Центроидами для узлов являются их координаты. Тогда при предположении, что не может быть двух разных элементов с одинаковыми центроидами, можно установить порядок элементов, сравнивая их центроиды. Легко можно привести пример, когда это предположение не верно: если элемент *А* — куб со стороной *h*, а элемент *Б* — куб со стороной *H* с вырезанным в середине элементом *А*, то центроиды элементов *А* и *Б* будут совпадать. Поэтому в качестве основного упорядочивания можно ввести порядок по центроидам,

а далее, если центроиды элементов совпали, можно проверить порядок описанной выше процедурой сравнения. В том случае, когда нам известна глобальная нумерация элементов, можно использовать более экономную сортировку по глобальному номеру.

Если в дальнейшем будет необходимо ввести такие элементы, как параметрическая кривая, параметрическая поверхность, либо параметрическая фигура, которая зависит не от вершин, а от некоторого правила построения, то необходимо будет либо разбить такие элементы на узлы, ребра, грани, либо ввести некоторое новое правило сравнения для таких элементов.

Так как в рассмотренной выше задаче упорядочивания удобно работать с множествами самых различных элементов, то можно ввести в структуру базового ядра, помимо узла, ребра, грани, ячейки, понятие множества, которое может состоять из этих элементов. Множество может быть как упорядочено на основе установленного выше порядка, так и не упорядоченно. Будет логично, если определенное нами множество будет разрешать различные операции, характерные для множеств, такие как пересечение, объединение, разность элементов и т. п.

1.3.4. Сеточные данные

Одно из ключевых и еще не рассмотренных понятий, необходимых для полного описания структуры базового ядра, — *данные*. Данные — это та информация, которая может храниться в сеточных элементах. Например, для узла данными будут его координаты, для остальных элементов можно добавить в множество данных информацию о центроидах. Для методов решения дифференциальных уравнений математической физики данными будут температура, скорость, давление, концентрация, тензор напряжений и т. д. Для составления систем линейных уравнений будет необходим глобальный идентификатор элемента, который

будет определять его строку/столбец в матрице. Данные могут быть использованы для задания параметрических элементов в сетке.

Некоторые данные могут быть определены как для всех элементов, то есть узлов, ребер, граней, ячеек, так и для некоторых, например, только на гранях. Некоторые типы данных, например, пометка о типе граничных условий, могут стоять только на граничных гранях, поэтому нужен гибкий механизм из набора возможностей задания данных. Данные могут быть определены как на узлах, ребрах, гранях, ячейках, так и на множествах, а также на всей сетке. Таким образом, данные могут представлять из себя целое число, действительное число или массив действительных или целых чисел. Для общности можно ввести байтовые массивы, которые позволят представить произвольные данные.

Следует отметить, что, если длина массивов данных будет динамически изменяться в большом числе элементов, то выгоднее, чтобы под массивы выделялось новое пространство памяти, и данные копировались в новое пространство памяти, а старая память удалялась. Такая стратегия может занять немного больше времени, чем попытка расширения памяти в старом пространстве памяти, но позволит избежать сильной дефрагментации памяти компьютера.

1.3.5. Ярлыки

Для управления данными будет удобно ввести понятие ярлыка. Иногда в литературе для этого понятия используется также название «тег» (tag) или «атрибут» (attribute).

Каждый *ярлык* будет определяться своим именем, а также содержать информацию, где и как искать данные в элементе каждого типа, то есть в ячейке, грани, ребре, узле, либо во множестве, либо в сетке. Ярлык также будет обозначать, какому типу данных он соответствует: целочисленному, действительному или байтовому. В ярлыке также будет храниться информация, является ли он «плотным», т. е. создан

в каждом элементе данного типа, или «разреженным», т. е. принадлежит только некоторым элементам данного типа. Каждый ярлык впоследствии будет храниться в сетке и может быть запрошен по своему имени.

Таким образом, можно ввести следующий характер наследования между структурами:

Данные → Элемент, Множество, Сетка
Элемент → Узел, Ребро, Грань, Ячейка

и следующим образом определить структуру сетки:

Сетка → набор ярлыков, набор узлов, набор ребер, набор граней, набор ячеек, набор множеств.

Структуру связей между элементами мы уже определяли ранее круговой диаграммой:

Узлы ↔ Ячейки ↔ Грани ↔ Ребра ↔ Узлы.

Таким образом, мы ввели основную структуру базового ядра, теперь следует определить функции для операций с сеточными элементами и данными. Некоторые описанные далее функции будут использоваться для внутреннего управления сеточными данными и элементами и не должны быть видны пользователю, другие же функции будут доступны для пользователя.

Определим функционал для ярлыков, предназначенных для внутреннего управления.

- Указать позицию данных в сеточном элементе.
- Получить позицию данных в сеточном элементе.
- Проверить, определены ли данные на типе элементов.
- Конструктор ярлыка, принимающий в качестве параметров имя и тип данных. Конструктор доступен только для сетки.

Определим доступный пользователю функционал для ярлыков.

- Получить имя ярлыка.

- Получить тип данных, соответствующих ярлыку.
- Получить длину данных в байтах.
- Конструктор копирования.
- Оператор присваивания другого ярлыка.
- Оператор «меньше» другого ярлыка.
- Оператор равенства с другим ярлыком.
- Деструктор ярлыка.

Для данных определим следующие функции, доступные только для управления.

- Конструктор, принимающий в качестве параметра тип элемента.

Определим функции, доступные для пользователя.

- Получить по ярлыку действительное число.
- Получить по ярлыку массив действительных чисел.
- Получить по ярлыку целое число.
- Получить по ярлыку массив целых чисел.
- Узнать по ярлыку длину массива данных.
- Задать по ярлыку длину массива данных.
- Задать по ярлыку данные некоторой длины, начиная с некоторой позиции.
  - ♦ Для массивов целых и действительных чисел сдвиг и длина определяется в элементах этих массивов.
  - ♦ Для байтовых массивов сдвиг и длина определяется в байтах.
- Получить по ярлыку данные некоторой длины, начиная с некоторой позиции.
- Удалить данные, соответствующие ярлыку.
- Проверить, имеются ли данные, соответствующие ярлыку.
- Оператор присваивания других данных.
- Узнать, какому типу элемента соответствуют данные.
- Конструктор копирования.
- Деструктор.

Следует отметить, что в программе для удобства при запросе числа или массива чисел можно передавать не зна-

чения этих чисел, а ссылку на них, так, чтобы изменение соответствующих этим числам переменных отражалось на данных.

1.3.6. Функции для элементов

Далее определим функции для элементов. Недоступен для пользователя только конструктор элемента, получающий в качестве параметра тип элемента. Для пользователя доступны следующие функции.

- Получить количество соседних элементов заданных типов.
- Получить множество соседних элементов заданных типов.
- Получить геометрический тип элемента.
- Получить размерность элемента.
- Присвоить другой элемент.
- Убрать связи между данным и окружающими элементами.
- Деструктор.

Следует уточнить, что функция получения соседних элементов собирает элементы только по иерархии вниз или только по иерархии вверх. Это означает, что функция получения соседних ячеек для ячейки вернет также и ее саму. Поэтому для получения соседей второго порядка, то есть, к примеру, всех соседних ячеек через ребра, следует для начала запросить все соседние ребра данной ячейки, а затем для всех ребер запросить соседние ячейки и из полученного множества вычесть данную ячейку.

При отсоединении связей между данным элементом и окружающими его элементами можно предложить два варианта действий: либо элементы выше по иерархии, которые опирались на данный элемент, теряют замкнутость, либо элементы выше по иерархии удаляются. Конкретный выбор действий зависит от сеточного генератора, однако для определенности мы можем остановиться на втором варианте.

При удалении элемента, то есть вызове деструктора, связи между элементами убираются, чтобы никакие элементы не имели ссылок на уничтоженный элемент. Для удобства сделано так, чтобы элемент знал свою позицию в наборе элементов внутри сетки и мог освободить занимаемую позицию при удалении.

Для реализации быстрого и эффективного удаления и добавления элементов в сетке, помимо массива элементов, хранится массив пустых позиций в массиве элементов. При удалении элемента соответствующая ему позиция помечается как пустая в массиве пустых позиций. В дальнейшем при добавлении элемента проверяется массив пустых позиций, и если этот массив пустой, то элемент добавляется в конец массива элементов.

Элементы по правилу наследования получают все функции, определенные для данных.

1.3.7. Функции для множества элементов

Далее определим набор функций для множества элементов, доступный для пользователя.

- Конструктор множества (создает неупорядоченное множество).
- Конструктор множества с функцией сравнения в качестве параметра (создает упорядоченное множество).
- Деструктор множества.
- Вставка элемента во множество.
- Вставка набора элементов во множество.
- Удаление элемента из множества.
- Пересечение данного множества с другим.
- Объединение данного множества с другим.
- Вычитание из данного множества другого множества.
- Запрос количества элементов множества.
- Удаление всех элементов множества.
- Проверка, является ли множество пустым.
- Оператор присваивания другого множества.

- Получить итератор, указывающий на первый элемент множества.
- Получить итератор, указывающий на позицию за последним элементом множества.
- Получить реверсивный итератор, указывающий на последний элемент множества.
- Получить реверсивный итератор, указывающий на позицию перед первым элементом множества.
- Нахождение элемента во множестве, возвращающее итератор, указывающий на позицию элемента во множестве, либо на позицию за последним элементом множества, если такого элемента во множестве нет.

При определении функционала множества появились такие понятия, как итератор и реверсивный итератор. Фактически итераторы — это структуры, позволяющие перебрать элементы множества. Обычные итераторы перебирают элементы в прямом порядке, реверсивные — в обратном. Для итераторов определены операции получения элемента, на который они указывают, и перехода к итератору, указывающему на следующий элемент во множестве.

Следует отметить: так как структура множества наследует структуру данных, то все функции для данных также доступны для множества элементов.

1.3.8. Функции для сетки

Определим функции для сетки, доступные пользователю:

- Конструктор сетки.
- Конструктор копирования.
- Деструктор.
- Оператор присваивания другой сетки.
- Установить порог для функции сравнения элементов.
- Получить порог для функции сравнения элементов.
- Установить размерность пространства (определяет количество координат узлов).
- Получить размерность пространства.

- Загрузить сетку из файла.
- Сохранить сетку в файл.
- Создать ярлык с заданным именем, заданным типом данных, для заданных типов элементов, разреженный или плотный.
- Получить ярлык по имени.
- Получить имена всех ярлыков.
- Удалить ярлык и все соответствующие ему данные.
- Создать узел с заданными координатами.
- Создать ребро из набора узлов.
- Создать грань из набора ребер.
- Создать ячейку из набора граней.
- Создать множество.
- Объединить дубликаты соседних элементов заданных типов для заданной ячейки.
- Объединить дубликаты заданного элемента, лежащие ниже по иерархии.
- Объединить дубликаты для заданных типов элементов (сортирует все элементы, находит дубликаты и удаляет их).
- Заполнить пустые позиции в массивах элементов.

Определим также функции, необходимые для внутреннего управления работой с сеткой:

- Привязка элемента к сетке (реализует описанный выше алгоритм добавления элементов в массив с проверкой пустых позиций).
- Отсоединение элемента от сетки (реализует описанный выше алгоритм удаления элементов из массива с добавлением пустых позиций).
- Объединение двух элементов, второй элемент после объединения удаляется.
- Взятие нормали для набора узлов.
- Проверить планарность набора узлов.
- Проверить замкнутость набора ребер грани.
- Проверить замкнутость набора граней ячейки.

По аналогии с функциями для множеств, введем понятие итератора для сетки, чтобы можно было выполнять различные выборки элементов, например, запрашивать элементы по геометрическому типу, по размерности, по значению данных и т. п., а также пробегать по этой выборке.

Алгоритмическая реализация этих функций является достаточно очевидной.

1.4. Работа с распределенными сеточными данными

Таким образом, мы описали структуру сеточных элементов и данных, иерархию связей, функции для работы с сеточными данными, но не затрагивали функционал для распределенных сеточных данных. Будем считать, что этот функционал реализуется на основе интерфейса межпроцессорных коммуникаций MPI, хотя можно было бы использовать также любые другие механизмы передачи данных.

1.4.1. Свойства распределенных элементов

Сначала следует понять, какой именно функционал для распределенных элементов нам нужен. Поскольку мы решаем системы дифференциальных уравнений одним из методов: конечных элементов, конечных разностей или конечных объемов, ключевой операцией является получение соседних ячеек. Следовательно, для элементов, принадлежащих данному процессору, нам необходимо знать все соседние элементы. То есть необходимо создать приграничный слой фиктивных элементов вокруг принадлежащих данному процессору элементов. Для этого нам необходимо уметь вычислять множество граней, ребер и вершин, которые лежат между ячейками, принадлежащими двум разным процессорам, уметь упаковывать геометрические данные ячеек и пересылать их между процессорами. Необходимо также уметь синхронизировать данные между элементами.

И, наконец, необходимо уметь эффективно балансировать и перераспределять сеточные элементы.

Далее, введем несколько ярлыков данных, необходимых нам для дальнейшего расширения функционала, предназначенного для распределенных сеточных данных.

Состояние — ярлык, которому соответствует состояние элемента:

- собственный — элемент, которым владеет только данный процессор;
- общий — элемент, которым владеет данный процессор, но копия элемента имеется у других процессоров;
- фиктивный — копия элемента, которым данный процессор не владеет.

Владелец — ярлык, которому соответствует число, обозначающее идентификатор процессора-владельца элемента.

Процессоры — ярлык, которому соответствует отсортированный массив процессоров, указывающий, какие процессоры имеют копию данного элемента, помимо данного процессора. Такой же массив хранится в сетке для определения, с какими процессорами существуют связи в данной сетке.

1.4.2. Работа с распределенными элементами

Вполне вероятна ситуация, когда информация о состоянии, владельце и процессорах недоступна, и мы имеем только геометрическую сетку. Тогда такую информацию необходимо восстановить. Для этих целей можно использовать следующий алгоритм.

- Каждый процессор собирает свой массив, состоящий из координат узлов, локального номера узла в массиве и своего номера процессора.
- Каждый процессор поочередно обменивается со всеми остальными процессорами своим массивом посредством функции «MPI_Bcast».
- Свой массив и полученный массив сортируются по координатам, и в массиве ищутся продублированные узлы.

- Если узел продублирован на другом процессоре, то этот процессор добавляется в массив, соответствующий ярлыку «процессоры».
- Владельцем узла определяем процессор с наименьшим номером.
- Определим состояние узла.
  - ♦ Если массив процессоров пустой, то элемент собственный.
  - ♦ Если массив не пустой, но владеет элементом данный процессор, то элемент общий.
  - ♦ Иначе элемент фиктивный.
- Объединяя множества процессоров каждого элемента, мы получим множество всех процессоров, с которыми должен обмениваться данный процессор. С этого момента мы можем ограничить множество процессоров, с которыми должен общаться данный процессор.
- Для каждого ребра мы можем посчитать его массив процессоров, найдя пересечения множеств процессоров, которым принадлежат его узлы. Однако, если два узла ребра принадлежат определенному процессору, само ребро не обязательно принадлежит этому процессору. Для определения множества ребер, которые принадлежат другому процессору, мы выполняем следующие шаги.
  - ♦ Находим глобальную нумерацию для вершин, которая потребуется для однозначного нахождения ребра по его вершинам.
    - ✧ Каждый процессор считает число локальных элементов.
    - ✧ По полученному числу выполняется операция «MPI_Scan», в результате чего процессор узнает номер, с которого он должен пронумеровать свои элементы.
    - ✧ Все процессоры одновременно нумеруют свои элементы.
  - ♦ Составляем массив пар: глобальный номер, локальный номер узла и сортируем его.

- ♦ На каждом процессоре мы собираем в массив ребра, которые потенциально могут принадлежать соседнему процессору.
- ♦ Для каждого ребра собираем в числовой массив следующую информацию:
  - ✧ количество узлов;
  - ✧ глобальные номера узлов.
- ♦ Процессор посылает собранный массив всем соседям посредством асинхронной функции посылки «MPI_Isend».
- ♦ Составляем список всех соседей и ожидаем приема по очереди от каждого соседа посредством неблокирующей операции «MPI_Iprobe». В момент приема выполняем следующие действия.
  - ✧ Перебираем элементы принятого массива, находим узлы с помощью бинарного поиска в массиве пар, состоящих из глобального и локального номеров.
  - ✧ Находим посланное ребро по пересечению всех ребер данных узлов.
  - ✧ Если ребро найдено — добавляем его в массив найденных ребер, иначе переходим к следующей записи.
  - ✧ Когда вся принятая информация обработана, строим пересечение множеств потенциально общих ребер и найденных ребер. Пересечение этих множеств и есть искомое множество, общее для двух процессоров. Добавляем этим ребрам в массив процессоров соседний процессор.
  - ✧ Исключаем данный процессор из списка на ожидание приема и продолжаем прием.
- ♦ Заполнив список процессоров, отмечаем владельца ребра и состояние ребра.

- Описанный выше алгоритм обобщается для граней и ячеек.
- Если изначально в сетке были слои фиктивных ячеек, то все эти слои заберет себе процессор с меньшим номером, что не всегда бывает удобно. Поэтому для определенности перед выполнением описанной процедуры можно удалить все фиктивные ячейки.

Определим алгоритм удаления фиктивных ячеек.

- Заводим новый ярлык целочисленных разреженных данных произвольной длины для граней, с помощью которого мы будем помечать, удаляет ли данный процессор эти элементы.
  - Перебираем все фиктивные грани и запрашиваем число соседних ячеек для каждой грани. Если оно равно нулю, то помечаем, что процессор с данным номером удаляет эту грань.
  - Используем алгоритм аккумуляции данных, который будет описан ниже, чтобы получить на процессоре-владельце массив, который обозначает, какие процессоры удаляют эту грань.
  - Убираем из массива процессоров для грани те процессоры, которые ее удаляют, если массив оказался пустым, то помечаем грань на процессоре-владельце как собственную.
  - Все остальные процессоры, у которых данная грань является фиктивной, и количество соседних ячеек равно нулю, также удаляют эту грань.
- Применяем тот же самый алгоритм для ребер и узлов.

Заметим, что в начале алгоритма можно посчитать окаймляющие сетку кубы для каждого процессора и по пересечению этих кубов определить, какие процессоры должны обмениваться данными, сократив тем самым объем коммуникаций. Для добавления одного приграничного слоя фиктивных элементов требуется определить множество граней, ребер и вершин, которые находятся на разделе между ячейками данного процессора и процессора, для которого требуется создать приграничный слой фиктивных элементов. Ячейки, которые должны будут войти в приграничный слой фиктивных элементов для соседнего процессора, являются соседними по отношению к этому множеству.

Определим алгоритм, по которому можно найти это множество:

- Перебираем все фиктивные и общие грани, которые не принадлежат границе области, и рассматриваем соседние ячейки этих граней. Возможно четыре варианта:
  - Если фиктивные слои созданы с обеих сторон, то одна соседняя ячейка является фиктивной, а одна общей.
  - Если фиктивные слои не созданы, то должна быть только одна собственная ячейка.
  - Если фиктивный слой создан только данным процессором для соседнего, то должна быть только одна общая ячейка.
  - Если фиктивный слой создан соседним процессором для данного процессора, то должна быть одна собственная и одна фиктивная ячейка.
- В перечисленных выше четырех случаях будем добавлять грань во множество.
- Добавим во множество также все ребра и все вершины от полученного выше множества.

Найденное множество и будет искомым. Описанный выше алгоритм будет работать в предположении, что сетка конформная, и у каждой грани может быть как максимум две соседние ячейки. Алгоритм можно расширить на случай неконформных сеток, однако это потребует более сложного анализа. Определить, принадлежат ли грани границе области, можно с помощью геометрической модели.

Чтобы узнать, какие элементы данного множества следует использовать, чтобы создать слой фиктивных ячеек для другого процессора, каждый процессор помечает свои элементы своим номером, а затем выполняется редукция данных. В результате редукции каждый процессор узнает, какая часть элементов данного множества пересекается с соседними процессорами. Если геометрическая модель не доступна, то можно поместить в каждую грань глобальные номера соседних ячеек, а затем запустить аккумуляцию данных. Если полученный в процессе аккумуляции глобальный номер уже присутствует в массиве, то пропускаем его, иначе помещаем в массив. Если сетка конформная, то в результате все грани на границе области будут иметь по одной записи, а внутренние по две.

1.4.3. Упаковка, распаковка, синхронизация и аккумуляция данных

Перед описанием алгоритма обмена соседними ячейками опишем алгоритм упаковки, распаковки и синхронизации данных между ячейками, так как эти алгоритмы потребуются нам в дальнейшем.

Упаковку данных реализуем следующим образом, через функцию «MPI_Pack»:

- Параметры алгоритма:
  - ярлык данных, которые следует упаковать;
  - сортированное множество элементов, для которых следует упаковать данные;
  - промежуточный байтовый массив, который будет пересылаться, т. е. буфер.
- Создадим два массива: в одном хранятся длины массивов данных, соответствующие каждому из передаваемых элементов, во втором — сами массивы данных.
- Перебираем все элементы множества: если данные, соответствующие ярлыку, найдены на данном элементе, то мы добавляем в массив длин длину массива найденных данных, а в массив данных копируем сам массив. Если данные не найдены, то записываем в массив длин ноль.
- Упаковываем сначала массив длин, а затем массив данных посредством функции «MPI_Pack» записывается в конец буфера. При этом длина первого массива равна количеству элементов множества, а длина второго массива равна сумме длин из первого массива.

Следует отметить, что библиотека обменов MPI позволяет работать на процессорах с различной архитектурой и при передаче данных может конвертировать данные в промежуточный формат, чтобы данные могли быть верно приняты на процессоре с другой архитектурой. Для этого можно расширить функционал ярлыков, чтобы пользователь мог указать формат передаваемых данных для байтовых массивов, при этом формат данных для целых и действительных массивов чисел может выставляться автоматически.

Опишем алгоритм распаковки данных:

- Параметры алгоритма:
  - ♦ ярлык распаковываемых данных;
  - ♦ отсортированное множество элементов распаковываемых данных;
  - ♦ буфер, из которого следует распаковывать данные;
  - ♦ текущая позиция в буфере.
- Распаковываем из буфера по количеству элементов во множестве массив длин посредством функции «MPI_Unpack», сдвигаем текущую позицию в буфере.
- Суммируем полученные длины и распаковываем массив данных, сдвигаем текущую позицию.
- Перебираем все элементы множества и заменяем существующие данные на полученные.

Эти алгоритмы позволяют в один и тот же буфер поочередно упаковать и распаковать данные, соответствующие нескольким ярлыкам. Опишем теперь алгоритм синхронизации данных между процессорами. Предположим, что разметка по процессорам, владельцу и состоянию всех элементов не нарушена. Тогда, если взять два процессора $П_1$ и $П_2$, количество фиктивных элементов у процессора $П_2$, на которых стоит пометка, что владелец $П_1$, должно совпадать с количеством общих элементов у процессора $П_1$, у которых в массиве процессоров присутствует процессор $П_2$. Алгоритмы, обеспечивающие подобную слаженность, будут описаны в дальнейшем.

Алгоритм синхронизации данных.

- Параметры алгоритма:
  - ♦ ярлык, соответствующий передаваемым данным.
- Перебираем элементы массива процессоров данной сетки, пусть в данный момент рассматривается процессор П.
  - ♦ Проходим по всем элементам этого процессора, на которых определен данный ярлык, проверяем статус, владельца и массив процессоров данного элемента:
    - ✧ если элемент общий, и в его массиве процессоров присутствует процессор П, то помещаем его в один массив общих элементов;

- ✧ если элемент фиктивный и его владельцем является процессор П, то помещаем его в массив фиктивных элементов для процессора П (то есть массивов фиктивных элементов может быть несколько).
- ♦ Если массив общих элементов не пуст, то:
  - ✧ сортируем массив общих элементов, а также массив фиктивных элементов для процессора П;
  - ✧ запускаем функцию упаковки данных, соответствующих ярлыку и массиву общих элементов, и отправляем полученный буфер процессору П посредством асинхронной функции «MPI_Isend».
- ♦ Очищаем массив общих элементов и переходим к следующему процессору П в массиве процессоров.

• Создаем список соседних процессоров, перебираем элементы списка, пока он не окажется пустым, и ожидаем приема данных посредством функции «MPI_Iprobe». Пусть функция сигнализирует, что пришли данные от процессора П.
- ♦ Если множество фиктивных ячеек для данного процессора не пусто:
  - ✧ проверяем размер присылаемого сообщения от процессора П посредством функции «MPI_Get_count»;
  - ✧ принимаем буфер посредством функции «MPI_Recv»;
  - ✧ распаковываем данные из буфера в массив уже отсортированных фиктивных ячеек для процессора П.
- ♦ Удаляем процессор П из списка.

• Ожидаем завершения асинхронной пересылки посредством функции «MPI_Wait».

Так как множества фиктивных ячеек у принимающего процессора и общих ячеек у посылающего процессора сортированы одинаковым образом (и их количество одинаково), то данные всегда будут приходить по адресу. В качестве наиболее эффективной сортировки можно использовать сортировку по глобальной нумерации. Однако если синхронизируется именно глобальная нумерация, или глобальная нумерация не была присвоена элементам, то можно приме-

нить одну из ранее предложенных сортировок по центроидам или по иерархии элементов.

Алгоритм синхронизации данных переносит данные в одну сторону — из общих ячеек в фиктивные ячейки. В некоторых случаях может потребоваться обработать некоторым образом данные, помещенные как в общие, так и в фиктивные ячейки. Например, найти сумму или произведение значений, причем эта операция определяется заданной пользователем функцией. Назовем такой алгоритм *алгоритмом аккумуляции данных*. Алгоритм работает аналогично алгоритму синхронизации данных, только посылаются данные из фиктивных ячеек, а принимаются в общие. При приеме данных вызывается функция пользователя, которая должна обработать принятые данные. Одной собственной ячейке может соответствовать несколько фиктивных, тогда при приеме данных функция пользователя будет вызываться многократно. После того как данные были аккумулированы в общих ячейках, запускается алгоритм синхронизации, чтобы те же данные появились в фиктивных ячейках.

1.4.4. Упаковка и распаковка множества

Далее, для обеспечения обмена приграничными слоями фиктивных ячеек опишем три алгоритма: упаковки множества элементов с данными, распаковки множества элементов с данными и алгоритм сборщика помеченных на пересылку элементов, осуществляющего передачу элементов.

Чтобы описать алгоритм упаковки множества элементов, надо понять, что именно требуется для того, чтобы удаленный процессор мог восстановить в полном объеме геометрическую информацию о полученных элементах. Для этого необходимо передать удаленному процессору все элементы, лежащие в иерархии ниже по отношению к данному, а также связи по иерархии вниз. Связи по иерархии вверх удаленный процессор сможет восстановить сам. Таким же образом следует передать все данные, принадлежащие элементу. Некоторые данные, такие как номер элемента в массиве данной сетки или состояния

элемента на данном процессоре, передавать не следует, так как это может нарушить работу алгоритмов на удаленном процессоре. Для этого можно фильтровать данные по имени ярлыка, например, если имя ярлыка начинается с “PROTECTED”, то его передавать не следует. Считаем, что все упаковки и распаковки данных выполняются посредством функций «MPI_Pack» и «MPI_Unpack». Каждая соответствующая упаковка записывается в конец буфера, а каждая распаковка сдвигает позицию в буфере.

Алгоритм упаковки множества элементов.

- Параметры алгоритма:
  - ♦ множество упаковываемых элементов;
  - ♦ буфер, в который совершается упаковка;
  - ♦ номер процессора, для которого совершается упаковка;
  - ♦ список имен ярлыков, соответствующие данные которых следует передать.
- Перебираем множество упаковываемых элементов и собираем по отдельности массив вершин, ребер, граней и ячеек.
- Перебираем эти массивы и добавляем в соответствующие массивы элементы, лежащие по иерархии ниже:
  - ♦ для ячеек: грани, ребра, вершины;
  - ♦ для граней: ребра и вершины;
  - ♦ для ребер: вершины.
- Для вершин упаковываем их количество и координаты всех вершин.
- Для ребер (граней, ячеек) упаковываем их количество, затем массив длин, обозначающий, по сколько связей по иерархии вниз имеет каждое ребро (грань, ячейка), а затем массив номеров по порядку упакованных ранее вершин (ребер, граней, соответственно), которые для каждого элемента лежат ниже по иерархии.
- В процессе упаковки помечаем каждый собственный элемент как общий и добавляем в массив процессоров

элемента удаленный процессор. Кроме этого, каждый элемент, для которого владельцем не является удаленный процессор, запишем в отдельное множество.

- Отсортируем собранное множество элементов, для которых удаленный процессор не является владельцем; для этих элементов мы будем записывать в буфер данные следующим образом:
  - упакуем количество ярлыков передаваемых данных;
  - для каждого ярлыка упакуем длину имени и само имя;
  - воспользуемся ранее описанным алгоритмом для упаковки данных для каждого ярлыка и собранного множества элементов в текущий буфер.
- Записываем удаленный процессор в массив процессоров сетки.

Таким образом, мы описали алгоритм упаковки данных, опишем теперь алгоритм распаковки данных.

- Параметры алгоритма:
  - множество, в которое будут записаны распакованные данные;
  - буфер, из которого распаковываются данные;
  - номер процессора, приславшего данные.
- Отсортируем множество вершин — это поможет нам эффективно искать дубликаты среди полученных и имеющихся вершин за $O(\log(n))$ операций.
- Создадим множество распакованных вершин, ребер, граней и ячеек.
- Распакуем количество вершин, по координатам будем создавать каждую вершину и проверять, существует ли дубликат. Если дубликат найден, то в множество вершин добавляется найденная вершина, а созданная вершина удаляется, иначе в множество вершин добавляется созданная вершина.
- Распакуем количество ребер (граней, ячеек), распакуем массив длин и номера лежащих ниже по иерархии вершин (ребер, граней, соответственно). Создадим каждое

ребро (грань, ячейку) и запросим соседние ребра (грани, ячейки) для всех вершин (ребер, граней), лежащих ниже по иерархии. Среди соседних ребер (граней, ячеек) проверим, существует ли дубликат созданного ребра (грани, ячейки) и если дубликат найден, то удалим созданный элемент, а во множество ребер (граней, ячеек) добавим имеющийся, иначе добавим созданный.

- В процессе распаковки будем помечать каждый полученный несобственный элемент как фиктивный. Будем также добавлять каждый полученный несобственный элемент в отдельное множество, в которое будут записываться полученные данные.
- Отсортируем множество несобственных элементов и распакуем количество полученных ярлыков данных. Для каждого ярлыка:
  - Распакуем длину имени и имя ярлыка и запросим у сетки сам ярлык.
  - Воспользуемся описанным ранее алгоритмом распаковки данных в множество несобственных элементов.
- Добавим процессор, приславший данные в массив процессоров данной сетки.

1.4.5. Сборка распределенных элементов

Таким образом, мы описали алгоритмы распаковки и упаковки данных. Пусть у нас есть три процессора $П_1$, $П_2$, $П_3$, и пусть процессор $П_1$ передает процессору $П_2$ некоторые данные. При выполнении своей работы процессор $П_1$ мог упаковать часть элементов, исходно принадлежащих процессору $П_3$ и передать их процессору $П_2$. В итоге процессор $П_3$ не будет знать о том, что процессор $П_2$ получил его элементы, и при синхронизации данных мы придем к тому, что на процессоре $П_2$ фиктивных элементов от процессора $П_3$ будет больше, чем на процессоре $П_3$ общих элементов с процессором $П_2$. Чтобы не возникла такая ситуа-

ция, необходимо, чтобы $П_2$ проинформировал $П_3$ о том, что он получил эти элементы. Опишем алгоритм сборщика, который будет выполнять пересылки всех помеченных на пересылку элементов, а также выяснять, какие процессоры не были проинформированы о копиях элементов, и информировать их.

1. Пусть пользователь или программа пометили некоторые элементы сетки на пересылку некоторым процессорам.
2. Создадим массив связанных пар (*P*, *E*): процессор, множество посылаемых процессору элементов. Создадим массив целых чисел *R*, длина которого соответствует общему числу процессоров, и в котором будем фиксировать, что была совершена пересылка некоторому процессору.
3. Перебираем все элементы сетки, если на элементе присутствует пометка о том, что следует переслать данный элемент другому процессору, то добавляем этот элемент в соответствующую ему связную пару в массиве (*P*, *E*).
4. Перебираем массив связных пар (*P, E*), упаковываем соответствующий массив элементов *E* вместе с данными и перешлем его соответствующему процессору *P* посредством асинхронной операции посылки «MPI_ Isend». Всем посылкам будет присвоен индикатор $И_1$. Индикатор необходим для того, чтобы не принимать ненужное сообщение. Одновременно будем фиксировать в массиве *R*, что была совершена пересылка процессору *P*. Мы можем очистить массив связных пар и использовать его в дальнейшем.
5. Просуммируем между процессорами массив *R*, с помощью орации «MPI_Allreduce». В результате в массиве *R* в позиции, соответствующей номеру данного процессора, будет находиться число посылок данному процессору.

6. По этому числу посылок мы знаем, сколько нам необходимо совершить приемов данных. Будем совершать прием следующим образом:
   a) используем функцию «MPI_Probe» для проверки, прислал ли нам какой-либо процессор данные с индикатором $И_1$;
   b) узнаем длину присылаемого сообщения с помощью функции «MPI_Get_count»;
   c) создаем буфер и принимаем сообщения с помощью функции «MPI_Recv»;
   d) распаковываем элементы из полученного буфера и проверяем для полученных элементов, присутствует ли в их массиве процессоров данный процессор. Если данный процессор отсутствует, то процессор-владелец не проинформирован. Мы добавляем такие элементы в массив связных пар (P, E), где P — процессор-владелец.
7. Ожидаем окончания всех асинхронных посылок при помощи функции «MPI_Waitall».
8. Теперь мы можем очистить массив R и повторить шаги с 4 по 6, с тем отличием, что в 4-ом шаге мы не упаковываем данные ярлыков, в шагах 4 и 6 вместо индикатора $И_1$ используем индикатор $И_2$ и не делаем промежуточный шаг 6d. В результате каждый процессор-владелец элементов будет проинформирован обо всех копиях своих элементов.
9. Если все ярлыки данных были заранее синхронизированы, то в результате всех проделанных выше операций синхронизированы будут все ярлыки, кроме ярлыка, соответствующего массиву процессоров, так как данные этого ярлыка активно изменяются в процессе упаковки и распаковки, поэтому мы также выполняем синхронизацию этого поля.

В итоге мы описали механизм, который позволит пользователю или программе запрашивать создание фиктивных

элементов в любой области сетки. Хотя, используя подобный функционал, пользователь сам бы мог создать несколько приграничных слоев фиктивных элементов или выполнить балансировку или перераспределение данных, для удобства кратко опишем также и эти алгоритмы.

1.4.6. Обмен слоями фиктивных ячеек

При обмене приграничными слоями фиктивных элементов следует помнить, что легко может встретиться топология, в которой процессоры, соседние с данным процессором, не обладают необходимым количеством слоев элементов. Однако, чтобы алгоритмы пользователя корректно работали на многопроцессорной системе, программа должна гарантировать запрошенные слои независимо от топологии сетки и топологии разбиения сетки на процессоры. Чтобы предоставить эти слои, соседний процессор для начала должен получить слой от своего соседа. Таким образом, создание границ должно быть выполнено итерационно: сначала каждый из процессоров обменивается первым слоем, затем каждый процессор обменивается вторым слоем и т. д. Таким образом, алгоритм обмена слоями будет выглядеть следующим образом.

- Параметры алгоритма:
  - ♦ число слоев N;
  - ♦ тип элементов T, по которым находятся соседние ячейки: грани, ребра или вершины.
- Перебираем все элементы и образуем множество элементов S, которые находятся на разделе между ячейками, принадлежащими разным процессорам, по описанному ранее алгоритму.
- Перебираем массив процессоров данной сетки, для каждого процессора П из массива:
  - ♦ перебираем все элементы типа T из множества S, для которых известно, что они принадлежат соответствую-

щему множеству *S* на процессоре П, и находим все их соседние ячейки, владельцем которых не является П;
- в каждой найденной ячейке, если в ее массиве процессоров П не присутствует, помечаем, что она должна быть послана процессору П, и записываем в ярлык *A* число *N*, которое будет обозначать, что от данной ячейки надо создать *N* слоев.

- Запускаем цикл по *K* от *N* до 0:
  - вызываем алгоритм сборщика, чтобы он переслал (*N*-*K*)-ый слой элементов;
  - перебираем массив процессоров данной сетки, для каждого процессора П из масива:
    - перебираем все элементы сетки, которые были посланы процессору П, и в ярлыке *A* которых записано число *K*,
    - находим соседние элементы типа *T* данных ячеек и находим для этих элементов соседние ячейки, владельцем которых не является П,
    - помечаем найденные ячейки на пересылку процессору П, если в их массиве процессоров процессор П не присутствует, и записываем в ярлык *A* число *K*-1;
  - уменьшаем *K* и переходим к началу цикла.
- Стираем все использованные пометки.

В результате последовательно все процессоры гарантированно обменяются несколькими слоями фиктивных ячеек.

1.4.7. Перераспределение и балансировка данных

Опишем теперь алгоритм перераспределения и балансировки данных. Допустим, что при помощи внешнего пакета (ParMETIS, PT-Scotch, Zoltan) было рассчитано новое распределение процессоров по элементам. Помимо разметки элементов для сборщика, данный алгоритм, после работы алгоритма сборщика, должен разметить новых владельцев всех элементов согласно полученному распределению, из-

менить статусы, а также, если известно, что до перераспределения или балансировки мы имели некоторое количество слоев, — оставить то же количество слоев при новом распределении. Опишем алгоритм.

- Параметры алгоритма:
  - ♦ массив чисел, каждое число обозначает нового владельца ячейки;
  - ♦ количество слоев;
  - ♦ тип T элемента, по которому находятся соседние слои.
- Определим нового владельца для граней, ребер и вершин. Для этого рассмотрим новых владельцев для элементов выше по иерархии и выберем наименьший номер.
- Перебираем все собственные элементы сетки, если новый владелец элемента отличается от текущего, проверяем, есть ли новый владелец в массиве процессоров данного процессора, если нет, то помечаем элемент на посылку этому процессору.
- Находим множество граней S, для которых соседние ячейки имеют различных новых владельцев, либо у фиктивных или общих граней которых есть только одна соседняя ячейка, владелец которой поменялся.
- Пополняем множество S ребрами и вершинами, лежащими ниже по иерархии для всех граней.
- Для всех элементов из S находим соседние ячейки и помечаем их количеством слоев. Далее будем синхронизировать пометки между процессорами и помечать слой за слоем, как это было описано выше, не обмениваясь, однако, самими элементами.
- Запускаем сборщик, чтобы он переслал все элементы.
- Заменяем для всех элементов их номер владельца и изменяем их статус.
- Обновляем массив процессоров данной сетки.

Данный алгоритм может быть модифицирован с целью увеличения его эффективности.

1.4.8. Основные функции для работы с сеточными данными

В заключение перечислим функции для работы с сеточными данными, необходимые при создании программной платформы:

- упаковать данные;
- распаковать данные;
- упаковать элементы с данными;
- распаковать элементы с данными;
- синхронизировать данные;
- собрать помеченные элементы и обменяться этими элементами с другими процессорами;
- удалить слой фиктивных ячеек:
- разместить ярлыки состояния, владельца и массивов процессоров для всех элементов на основе геометрической информации;
- рассчитать слой граней, ребер и вершин, разделяющий ячейки двух процессоров;
- создать несколько слоев фиктивных ячеек;
- перераспределить или сбалансировать данные.

Таким образом, мы описали структуру сеточных элементов и данных, описали иерархию между сеточными элементами, характер наследования между структурами, основные функции для базового ядра и расширили этот функционал описанием алгоритмов, необходимых для работы с распределенными сеточными данными.

1.5. Формирование и решение линейных систем

Результатом дискретизации исходной задачи, как правило, является формирование системы уравнений, которую необходимо решить. Разрабатываемая программная платформа предназначена для обеспечения возможности задания матрицы системы и ее правой части.

При задании коэффициентов матрицы происходит взаимодействие следующих компонентов платформы:

- сеточных данных;
- распределенных данных пользователя;
- решателя линейных систем.

Первые два компонента были подробно рассмотрены в разделах 1.3 и 1.4, а последний компонент будет более подробно обсуждаться ниже. Сейчас же мы остановимся только на основном наборе действий, необходимых пользователю программной платформы для задания матрицы системы.

Программная платформа должна обеспечить возможность задания нового элемента матрицы, а также добавления элемента матрицы к уже существующему. Задание или добавление элемента может происходить поэлементно, построчно, а также в виде набора строк (или блочной строки).

Так как структура разреженности матрицы системы обычно не известна заранее, то необходимо предусмотреть эффективную процедуру выделения дополнительной памяти.

К сожалению, в некоторых библиотеках функции сбора матрицы могут работать очень не эффективно (по заверениям разработчиков возможно замедление в 50 раз) из-за избыточного количества вызовов выделения дополнительной памяти. В одной из промежуточных реализаций программной платформы на основе свободно распространяемых пакетов нами наблюдался случай замедления в 800 раз. Это явилось причиной решения создать для программной платформы свое собственное средство сбора матрицы системы, как промежуточного звена между пользователем и решателем линейных систем.

1.5.1. Общие требования к решателям линейных систем

Современные средства решения больших систем линейных уравнений являются очень востребованными при решении задач математического моделирования. Так же, как и рассмотренные выше библиотеки для работы с сеточными

данными, они широко представлены в виде свободно распространяемых библиотек и пакетов программ. Для уменьшения затрат разработчиков, желательно иметь возможность использования готовых достаточно эффективных программных средств. Одним из самых основных критериев оценки пакетов является возможность их эффективного использования на параллельных высокопроизводительных ЭВМ.

В этом параграфе мы рассмотрим наиболее часто используемые пакеты решателей линейных систем, а также сформулируем основные критерии их сравнения. Но сначала остановимся на общих требованиях, которые будут предъявлены к рассматриваемым пакетам для их использования в программной платформе.

От применяемого пакета решателей линейных систем мы ожидаем:

1) надежной работы;
2) эффективного решения не слишком жестких линейных систем;
3) хорошей масштабируемости на достаточно большом количестве процессоров;
4) наличия определенных типов переобуславливателей и итерационных схем;
5) открытости исходного кода пакета.

В той или иной мере все рассматриваемые ниже пакеты обладают первыми тремя свойствами. Надежность пакетов линейных решателей косвенно гарантируется тем, что их использует огромное количество исследователей во всех областях математического моделирования. Эффективность работы основных схем была проверена еще на последовательных компьютерах. С появлением параллельных компьютеров стали более понятны методы получения хорошей масштабируемости. Теперь необходимо определиться с наиболее важным четвертым пунктом, касающимся собственно методов решения линейных систем. Именно от этого пункта в большей степени зависит выполнение первых трех требований.

Во-первых, стоит отметить, что нас интересуют в основном итерационные методы решения. Иногда применяемые прямые методы способны находить решение только для модельных задач небольшого размера, т. к. требуемая для их работы память и количество операций с увеличением порядка матрицы растет слишком сильно (в некоторых случаях и квадратично). Для лучших же итерационных методов рост затрат обычно получается если не линейным, то достаточно близким к этому.

Во-вторых, выбор итерационной схемы зависит от особенностей решаемой задачи, например, для для систем с симметричными положительно определенными матрицами можно рекомендовать использование метода сопряженных градиентов, а для других систем — стабилизированный (CG, Conjugate Gradient method) [13], а для несимметричных задач стабилизированный метод би-сопряженных градиентов (BiCGStab) [14, 16] и обобщенный метод минимальных невязок (GMRES, Generalized Minimal RESidual method) [54, 15]. Отметим, что метод BiCGStab лучше применять при использовании достаточно сильного переобуславливателя.

В качестве базового переобуславливателя можно использовать неполное или приближенное треугольное разложение [16] (IC или ILU для симметричных или несимметричных матриц, соответственно). Для несложных задач может быть вполне достаточно применения разложения, сохраняющего исходную структуру разреженности [16] (IC0 или ILU0, соответственно).

Дополнительным положительным фактором при выборе пакета является наличие переобуславливателей на основе многосеточных и многоуровневых методов, которые при правильной настройке параметров могут обеспечить независимость количества итераций от размерности решаемой задачи.

Осталось пояснить только последний пункт про открытость исходного кода пакета, которая является совершенно необходимым свойством, особенно если может потребоваться возможность расширения решателей, например, путем введения своих собственных переобуславливателей или итерационных схем.

1.5.2. Библиотека PETSc

Библиотека PETSc (The Portable, Extensible Toolkit for Scientific Computation) [17] предназначена для численного решения систем дифференциальных уравнений в частных производных на многопроцессорных вычислительных системах с распределенной памятью. Она представляет собой набор структур данных и процедур, являющихся строительными блоками для реализации крупномасштабных параллельных приложений. Для осуществления взаимодействия параллельных процессов через обмен сообщениями PETSc использует стандарт MPI.

PETSc включает в себя расширяемый набор параллельных процедур решения линейных и нелинейных уравнений, а также интегрирования, которые могут использоваться в приложениях, написанных на алгоритмических языках Fortran, C и C++. PETSc предоставляет удобные автоматизированные механизмы, необходимые при разработке параллельных приложений, такие как процедуры параллельной матричной и векторной сборки. Библиотека организована иерархически, что позволяет пользователям применять уровень абстракции, наиболее подходящий для решения определенной задачи. За счет использования в PETSc объектно-ориентированного подхода достигается исключительная гибкость.

Основная часть пакета PETSc представляет собой набор процедур для решения систем линейных алгебраических уравнений, возникающих при дискретизации уравнений в частных производных. PETSc поддерживает параллельные вычисления на устройствах с распределенной памятью при помощи библиотеки MPI, параллельные вычисления на устройствах с общей памятью при помощи нитей pthreads [18], массивно параллельные вычисления на графических ускорителях NVIDIA GPGPUs [19], а также комбинированный MPI-pthreads и MPI-GPU параллелизм. PETSc представляет собой свободно распространяемый код, который может быть установлен на компьютерах с операционными системами Windows и Linux.

1.5.3. Библиотеки из пакета Trilinos

Trilinos [12] — это набор, разработанных в Sandia National Laboratories пакетов, распространяемых по лицензии GNU библиотек с открытым кодом, предназначенных для использования в качестве блоков при разработке приложений.

На данный момент среди библиотек Trilinos, относящихся к решению систем уравнений, существуют пакеты следующих назначений:

- работа с векторами, матрицами и графами;
- факторизация матриц;
- решение линейных систем прямыми и итерационными методами;
- решение нелинейных систем.

Начиная с самого низкого программного уровня, Trilinos был разработан с учетом поддержки распределенной памяти и параллельных архитектур. Многие пакеты содержат как последовательные, так и параллельные интерфейсы с полностью совпадающей семантикой.

Библиотека предназначена для использования в проектах, написанных на C/C++ и Fortran, эти же языки применялись для ее разработки. Отдельные компоненты библиотеки были написаны на языке Python.

На данный момент в Trilinos входит 50 пакетов, еще более 50 внешних пакетов могут быть подключены по дополнительному требованию. Перечислим сгруппированные по темам пакеты, которые могут быть использованы для решения линейных систем. Часть пакетов упоминается более чем в одном разделе.

Основные библиотеки линейной алгебры

- Epetra — основное ядро пакетов линейной алгебры, включает средства для построения и работы с распределенными и последовательными графами, плотными и разреженными матрицами, векторами и набором векторов.
- EpetraExt — расширение ядра пакетов линейной алгебры Epetra.

Переобуславливатели

- AztecOO — последовательные переобуславливатели типа неполных треугольных разложений ILU (см. также раздел «Решатели линейных систем»).
- IFPACK — распределенные переобуславливатели, включающие неполные разложения, сглаживающие переобуславливатели на основе разбиения области (domain decomposition), совместимые с пакетом AztecOO.
- ML — совместимые с пакетом AztecOO алгебраические многоуровневые и многосеточные переобуславливатели для распределенных линейных систем.

Решатели линейных систем

- Teuchos — обеспечивает интерфейс к библиотекам BLAS и LAPACK, предназначенным для прямого решения плотных и ленточных линейных систем.
- AztecOO — итерационные схемы крыловского типа, используются объекты из Epetra, совместимые с IFPACK, ML и Aztec.
- Amesos — совместимый с Epetra пакет для прямого решения линейных систем, поддерживает вызов многих внешних пакетов, включая DSCPACK, SuperLU, SuperLUDist и UMFPACK.

1.5.4. Прямое решение систем с плотными матрицами

PLAPACK (Parallel Linear Algebra Package) [20] — пакет параллельных процедур линейной алгебры, который включает параллельные версии процедур решения систем линейных уравнений с плотными матрицами при помощи LU и QR-разложений, а также разложения Холецкого. Пакет PLAPACK разработан в University of Texas at Austin, и проектировался как набор строительных блоков, которые могут быть использованы при разработке параллельных приложений и библиотек более высокого уровня. Такими строительными блоками стали параллельные подпрограммы для

решения основных задач линейной алгебры и систем линейных уравнений.

В пакете PLAPACK определены несколько типов объектов линейной алгебры, такие как матрицы, векторы, мультивекторы (совокупность векторов). Разработчиками пакета была предложена схема формирования матрицы на основе множества подматриц меньшего размера. Каждая подматрица создается на отдельном узле, причем данный процесс происходит параллельно на всех узлах. Детали распределения скрыты при помощи так называемых мнимых (шаблонных) матриц и векторов, которые описывают распределение реальных объектов между узлами многопроцессорной вычислительной системы.

Для осуществления межпроцессорных коммуникаций в PLAPACK использован интерфейс передачи сообщений MPI. При передаче сообщений в PLAPACK в основном используются коллективные операции, такие, как обобщенная передача данных от одного процесса всем процессам «MPI_Scatter», обобщенная передача данных от всех процессов одному процессу «MPI_Gather», широковещательная рассылка «MPI_Bcast» и другие.

PLAPACK включает интерфейсы для языков Fortran и C.

Стоит отметить, что этот пакет с успехом используется для решения на параллельных компьютерах систем линейных уравнений с плотными матрицами, возникающими, например, при дискретизации интегральных уравнений.

1.5.5. Прямое решение систем с разреженными матрицами

Прямое решение систем линейных уравнений с разреженными матрицами имеет достаточно узкое применение ввиду более строгих ограничений на размер решаемой задачи. Такие пакеты могут применяться для целей отладки приложений, повышения надежности решения линейных систем, а также при работе с задачами относительно малого размера на небольшом числе процессоров.

1.5.5.1. Библиотека SuperLU

SuperLU [21] — библиотека общего назначения, применяемая для прямого решения больших разреженных несимметричных систем линейных уравнений на высокопроизводительных компьютерах. Набор библиотек SuperLU был разработан в Lawrence Berkeley National Laboratory.

Библиотека SuperLU написана на языке C и содержит интерфейсы для языков C и Fortran. Библиотека распространяется свободно вместе с исходным кодом. Существует 3 различных варианта библиотеки:

1) для последовательных архитектур (библиотека SuperLU);
2) для параллельных архитектур с общей памятью (shared memory) (библиотека SuperLU_MT);
3) для параллельных архитектур с распределенной памятью (distributed memory) (библиотека SuperLU_DIST).

1.5.5.2. Библиотека WSMP

Библиотека WSMP (Watson Sparse Matrix Package) [22] составляет коллекцию алгоритмов для эффективного решения систем линейных уравнений с разреженными матрицами. Эти алгоритмы могут использоваться для систем с общей памятью, распределенной памятью, а также систем с гибридной архитектурой. В свободном доступе имеются версии библиотеки для операционных систем AIX, Linux, Mac OS, Blue Gene и Cray XE6.

1.5.5.3. Библиотека UMFPACK

Библиотека UMFPACK (Unsymmetric MultiFrontal sparse LU factorization PACKage) [24] выполняет решение линейных систем с разреженными матрицами с помощью несимметричного мультифронтального метода факторизации. Библиотека UMFPACK разработана в университете Флориды. Она написана на языке ANSI/ISO C и включает в себя C, Fortran и MATLAB интерфейсы вызова функций.

1.5.5.4. Библиотека MUMPS

Библиотека MUMPS [23] обеспечивает возможность параллельного решения систем линейных уравнений с разреженными матрицами, пожалуй, наиболее эффективно. Среди интересных особенностей стоит отметить возможность использования комплексной арифметики, итерационное уточнение построенного разложения, обратный анализ ошибок. MUMPS поддерживает: Fortran, C, Matlab и Scilab интерфейсы вызова функций. Возможно применение упорядочивания матриц с использованием внешних пакетов: AMD, AMF, PORD, METIS, ParMETIS, Scotch, PT-Scotch.

1.5.6. Другие средства решения систем с разреженными матрицами

Стоит отметить коммерческий пакет с закрытым кодом PARDISO [57], который также является достаточно эффективным для решения систем линейных уравнений на компьютерах с общей и распределенной памятью.

Среди общедоступных средств решения линейных систем можно отметить, например, пакет Ani3D [26], в котором используются переобуславливатели на основе треугольного разложения второго порядка точности [27]. В настоящее время в открытом доступе имеется только последовательный вариант такого решателя, однако, разработка его параллельной версии уже ведется [28, 29], что позволит включить его в программную платформу в ближайшем будущем.

1.5.7. Выводы

В качестве базового решателя в программной платформе комплекса INMOST решено использовать пакет PETSc. Основными причинами такого выбора явились высокая надежность кода с более чем десятилетней историей его разработки, поддержка всех доступных на настоящий момент уровней параллелизма, включая использование графических ускорителей. Разработчиками с успехом может быть

использован представительный широкий набор переобуславливателей, включая переобуславливатели на основе алгебраических многосеточных методов.

Использование этого пакета в программной платформе происходит в рамках класса Solver. Сборка матрицы системы была реализована независимо в рамках класса Matrix, ввиду упомянутой выше недостаточной эффективности ее реализации в пакете PETSc.

Применение пакета PETSc в программной платформе показало его достаточную надежность и эффективность при решении систем линейных уравнений. В следующем подразделе приводится пример его использования для решения краевой задачи.

Отметим, что помимо базового решателя в программную платформу будет включен параллельный решатель [28, 29], обладающий надежностью и высоким потенциалом параллелизма при решении линейных систем.

1.5.8. Пример использования пакета PETSc для решения краевой задачи

В качестве примера рассмотрим численные эксперименты с использованием выбранного и протестированного нами в многопроцессорном режиме работы (см., например, [58]) пакета PETSc. Для работы с сетками и построения дискретизации использовались инструменты программной платформы, изложенные в разделе 1.4.

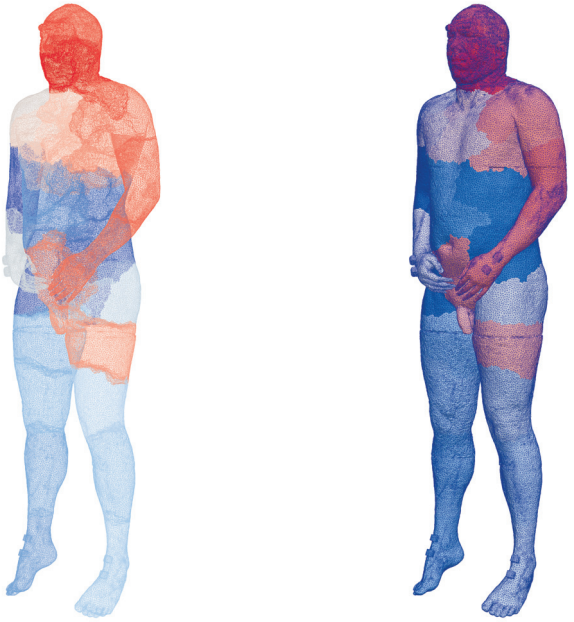
В качестве расчетной области выбрана подробная модель тела человека [55], представляющая собой геометрически сложную неструктурированную сетку с большим количеством тетраэдральных ячеек. Построенная сетка сгущается к границам 26 органов и тканей.

Матрица линейной системы получена из дискретизации уравнения Лапласа с граничными условиями Неймана методом конечных объемов со степенями свободы в центрах ячеек. Нормальный поток через грань ячейки вычислялся

как отношение разности значений в центрах ячеек, соседствующих через грань, к расстоянию между центрами ячеек.

Для решения линейной системы использовался пакет PETSc. Применялась итерационная схема GMRES с рестартами после каждых 30 итераций с использованием параллельного переобуславливателя по блочному методу Якоби, причем для каждого блока выполнялась неполная треугольная факторизация ILU0.

Для разбиения исходной сетки применялся метод PartGeomKway из пакета ParMETIS [7]. Разбиение для 36 процессоров продемонстрировано на рис. 7а в объеме и на рис. 7б на поверхности тела.



(а) *(б)*

Рис. 7. Разбиение сетки по процессорам:
объемное разбиение сетки (а); поверхностное разбиение (б)

Результаты параллельных расчетов представлены в таблице 1. Величина Nproc здесь обозначает количество используемых процессоров, T(ParMetis) — время разбиения исходной сетки по процессорам с помощью пакета ParMETIS, T(Assemble) — время сборки матрицы системы и T(Solve) — время решения сформированной линейной системы, необходимое для уменьшения начальной нормы невязки в 10^5 раз.

Таблица 1

Время, затраченное на этапах решения
для различного числа используемых процессоров

Nproc	T(ParMetis)	T(Assemble)	T(Solve)	Кол-во итераций
10	5,00602	11,8814	14,4237	190
20	5,03301	5,89182	6,43765	193
36	5,12716	3,59097	4,65554	193
40	5,11873	3,29811	4,89616	194
50	6,11521	2,61673	3,71723	194
55	6,05412	2,42676	3,16333	195

Общее количество неизвестных в исходной матрице линейной системы было равно 2 711 201, а общее количество ненулевых элементов матрицы составило 13 351 379. Столь большая размерность решаемой задачи была выбрана специально, чтобы показать реальную масштабируемость вычислений.

Анализ результатов численного эксперимента показывает, например, что общее время работы ParMETIS по распределению сетки слабо зависит от количества процессоров, а время сборки матрицы T(Assemble) хорошо масштабируется относительно количества процессоров. Заметим, что достаточно хорошо масштабируется также и решение линейной си-

стемы T(Solve) при слабом изменении количества итераций от числа процессоров. Последние замечания позволяют сделать вывод о надежности и эффективности работы выбранных пакетов ParMETIS и PETSc.

1.6. Реализация программной платформы

1.6.1. Использование дистрибутива программной платформы

Программная платформа имеет аббревиатуру MSPP (Meshing and Solving Parallel Platform). Она реализована на языке C++ и поставляется пользователю либо в виде исходных кодов, либо в виде готовой библиотеки программ для операционных систем Windows или Linux.

Процедура установки программной платформы может выглядеть следующим образом.

1. Распаковать архив дистрибутива MSPP:
   tar xvfz mspp.tar.gz
2. Создать библиотеку “mspp.a”:
   make
3. Рассмотреть несколько примеров использования платформы, которые находятся в директории “examples”:
   a) examples/Curve2D — пример работы с двумерной сеткой;
   b) examples/Curve3D — пример работы с трехмерной сеткой;
   c) examples/Octree — динамические сетки типа восьмеричного дерева;
   d) examples/DrawGrid — загрузка и отображение сетки;
   e) examples/Solve — чтение/запись файла vtk, решение модельной задачи (см. п. 1.5.8).

Для использования библиотеки платформы в программу необходимо включить заголовочный файл “mspp.h”, содержащий описание функций платформы. Краткое описание

каждой функции приводится в этом же файле в виде комментариев в стиле Doxygen [25], что, кроме выдачи описания в виде подсказок в Microsoft Visual Studio, также позволяет подготовить подробную документацию по программной платформе в формате PDF и/или html.

1.6.2. Иерархия классов программной платформы

В соответствии с описанной выше структурой программной платформы приведем иерархию составляющих ее классов, с кратким их описанием.

```
namespace MSPP — Пространство имен платформы MSPP
{
  class Tag — Ярлык (тег), предназначенный для хранения
              некоторого признака и работы с ним;
  class TagMemory — Класс для размещения информации
                    о ярлыке;
  class TagManager — Класс для работы с ярлыками;
  class Storage — Класс памяти;
  class ElementSet — Класс множества элементов
  {
    class iterator — Класс итератора по множеству
                     элементов;
    class reverse_iterator — Класс обратного итератора
                             по множеству элементов;
  };
  class ElementList — Класс списка элементов
  {
    class iterator — Класс итератора по списку элементов;
    class reverse_iterator — Класс обратного итератора
                             по списку элементов;
  };
  class adjacent — Класс связности элементов
  {
    class iterator — Класс итератора по связности
                     элементов;
    class reverse_iterator —  Класс обратного итератора
                              по связности элементов;
  };
```

```
class Element —  Класс элемент (определяемый через
                 класс Storage)
{
  enum GeometricType — Перечисление геометрических
                       типов
  {
    Unset,  — тип элемента не определен;
    Vertex, —  нольмерные: вершина (с 2 или 3
               координатами, в зависимости
               от размерности пространства);
    Line,MultiLine, — одномерные: отрезок, ломаная;
    Tri,Quad,Polygon,MultiPolygon, — двумерные:
       треугольник, четырехугольник, многоугольник,
       полимногоугольник (незамкнутый);
    Tet,Hex,Prism,Pyramid,Polyhedron — трехмерные:
  тетраэдр, гексаэдр, призма, пирамида, многогранник;
  };
  enum Status — Статус элемента
  {
    Owned = 1 — собственный элемент процессора;
    Shared = 2 — разделяемый элемент;
    Ghost = 4 — фиктивный элемент;
    Any = 7 — статус элемента не определен;
  };
};
class Node — Класс узел (определяемый через класс
             Element);
class Edge — Класс ребро (определяемый через класс
             Element);
class Face — Класс грань (определяемый через класс
             Element);
class Cell — Класс ячейка (определяемый через класс
             Element);
class Mesh — Класс сетка (определяемый через классы
             TagManager, Storage)
{
  class abstract_query — Класс абстрактного запроса;
  class query —  Класс запроса (определяемый через
                 класс abstract_query);
  class tag_query —  Класс ярлыка запроса
          (определяемый через класс abstract_query);
```

```
    class geometric_query — Класс геометрического
      запроса (определяемый через класс abstract_query);
    class base_iterator — Класс базового итератора;
    class iteratorElement — Класс итератора
                        по элементам (определяемый через
                        класс base_iterator);
    class iteratorSet — Класс итератора по множеству
              (определяемый через класс base_iterator);
    class iteratorCell — Класс итератора по ячейкам
            (определяемый через класс base_iterator);
    class iteratorFace — Класс итератора по граням
            (определяемый через класс base_iterator);
    class iteratorEdge — Класс итератора по ребрам
            (определяемый через класс base_iterator);
    class iteratorNode — Класс итератора по узлам
            (определяемый через класс base_iterator);
};
class Geometry — Класс геометрия (содержащий
    реализации вспомогательных геометрических функций);
class Partitioner — класс распределения сетки;
class Solver — Класс решатель (содержащий матрицу
    Matrix и два вектора Vector: вектор правой
    части и вектор решения)
{
class Row — Класс строка матрицы;
class Matrix — Класс матрица (содержащий набор строк
               Row данной матрицы);
class Vector — Класс вектор;
};
```

Глава 2

Графическая среда для разработки параллельных численных моделей

В этой главе будет описана вторая часть технологического комплекса INMOST, охватывающая графический интерфейс пользователя, систему научной визуализации и анализа данных.

Под термином *научная визуализация* мы понимаем ту область компьютерной науки, которая охватывает интерфейс пользователя, представление данных и алгоритмы обработки, а также графическое изображение этих данных. Появившись более двадцати лет назад, научная визуализация первоначально выполняла лишь две функции — обеспечение контроля и объективной трактовки численных экспериментов, а также иллюстративную функцию. С течением времени ситуация изменилась, и научная визуализация из иллюстративного инструмента превратилась в инструмент аналитический. Прежде всего, это связано с возросшим объемом данных, производимых в численном эксперименте, а также с широким применением суперкомпьютеров. Например, осмыслить 200 Гб данных, полученных в результате численного моделирования, очень трудно без системы научной визуализации. Можно сказать, что системы научной визуализации постепенно превращаются из вспомогательного средства в полноправный инструмент численного моделирования [63].

При этом важно, чтобы система научной визуализации была интерактивной и позволяла исследователю не только наблюдать конечный результат расчетов, но и вмешиваться в процесс моделирования, изменяя параметры модели в процессе проведения численных расчетов. Другим немаловаж-

ным требованием является возможность распараллеливания обработки данных в процессе визуализации, что опять же связано с огромным объемом данных и ресурсоемкостью процесса обработки.

При выборе технологической платформы для создания графической среды в задачах численного моделирования мы исходили из следующих требований:

- платформа должна быть надежной и опробованной;
- кроссплатформенной;
- основываться на пакетах с открытым кодом;
- платформа должна позволять интеграцию с различными языкми программирования, такими как Fortran, C, C++, Python.

Эти требования относятся к обоим компонентам платформы — и к графическому интерфейсу, и к системе визуализации. Однако каждый из компонентов имеет и специфические требования. Графический интерфейс дополнительно должен:

- иметь развитый набор стандартных элементов графического интерфейса: кнопки, меню, полосы прокрутки, менеджеры размещения и т. д.;
- давать возможность создавать свои элементы управления и диалоги.

Современная система визуализации должна:

- иметь богатый набор структур данных для научной визуализации: структурированные и неструктурированные сетки, различные типы дву- и трехмерных ячеек, многоугольники, ломаные и т. д.;
- уметь визуализировать скалярные, векторные и тензорные поля;
- иметь широкий выбор алгоритмов визуализации, таких как интерполяция, отсечение, построение изолиний и т. д.

Следующий раздел посвящен анализу различных программных продуктов, как возможных кандидатов для графической среды в комплексе INMOST.

2.1. Обзор современных подходов и существующих программных средств с открытым кодом

Для формулировки функциональных и технических требований к создаваемой графической среде были протестированы различные библиотеки и пакеты для научной визуализации: IBM OpenDX [30], Visualization Library [31], Visualization Toolkit (VTK) [32, 38, 39], ParaView [33, 34] и VisIt [35].

2.1.1. IBM OpenDX

IBM OpenDX является библиотекой для научной визуализации, написанной главным образом на C, и графическим интерфейсом, основанным на Motif. IBM OpenDX

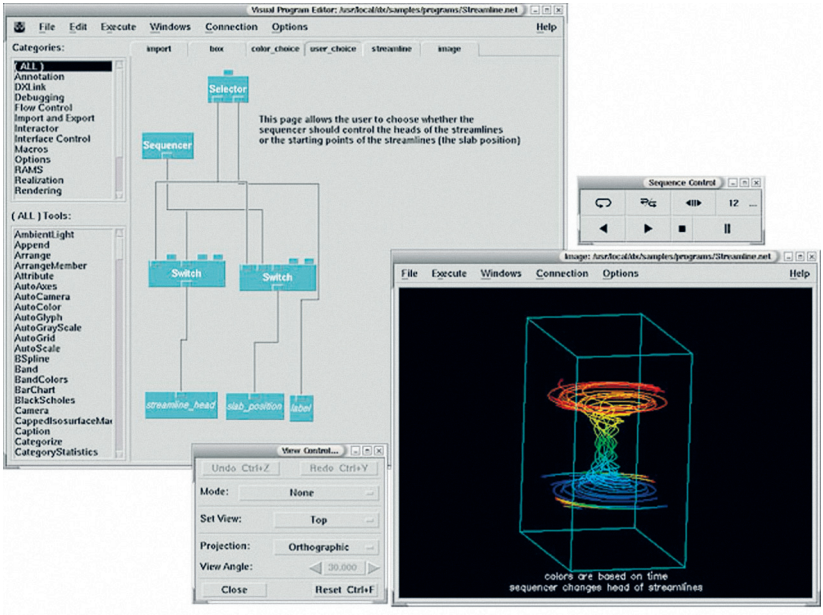


Рис. 8. Визуализация с использованием IBM OpenDX

был запущен в 1991 году как коммерческий продукт, а затем продолжил существование как проект с открытым программным кодом. Библиотека OpenDX поддерживает скалярные, векторные и тензорные данные. Из возможностей 3D-визуализации поддерживает закраску цветом, отрисовку линий потоков и лент, вращение.

2.1.2. Visualization Library (VL)

Visualization Library (VL) — кроссплатформенная библиотека с открытым программным кодом, написанная на C++. Поддерживает все версии OpenGL, интегрирована со

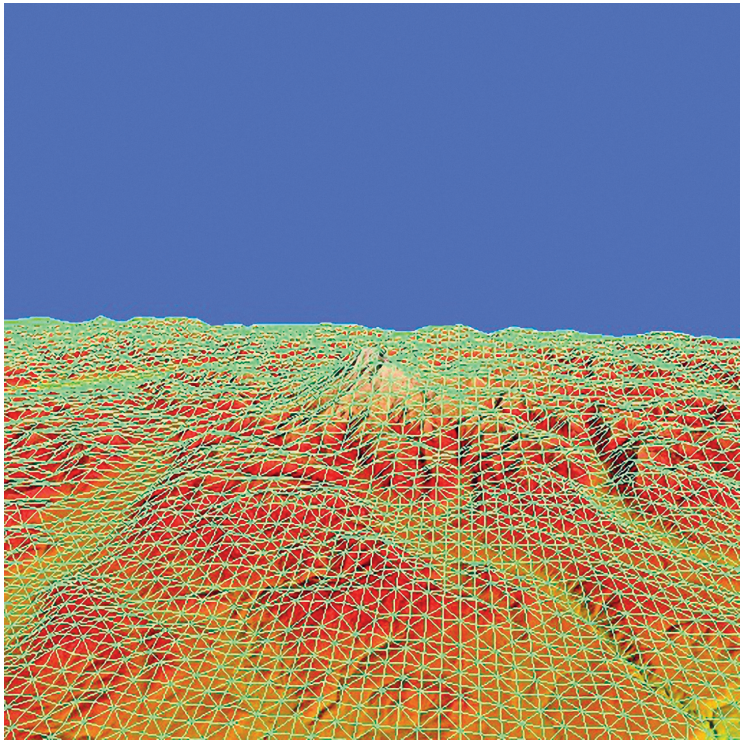


Рис. 9. Визуализация с помощью Visualization Libary

многими графическими интерфейсами, такими как Win32, MFC, Qt4, WxWidgets, GLUT. Поддерживает импорт достаточно большого количества 3D-форматов данных и изображений. Имеет встроенные алгоритмы объемной визуализации и геометрической оптимизации. Поддерживает работу с 1/2/3D-текстурами. Активно развивается в настоящее время.

2.1.3. Visualization ToolKit (VTK)

Visualization ToolKit (VTK) — кроссплатформенная библиотека с открытым кодом для научной визуализации, разработанная компанией Kitware. Написана на C++ и интегрирована с языками программирования Python, Tkl/Tc. Имеет развитую систему наборов данных, включая структурированные и неструктурированные сетки, а также много реализованных алгоритмов визуализации, включая скалярные, векторные и тензорные алгоритмы. Широко используется во многих программах для научной визуализации данных. Более подробно VTK рассмотрена в подразделе 2.2.2.

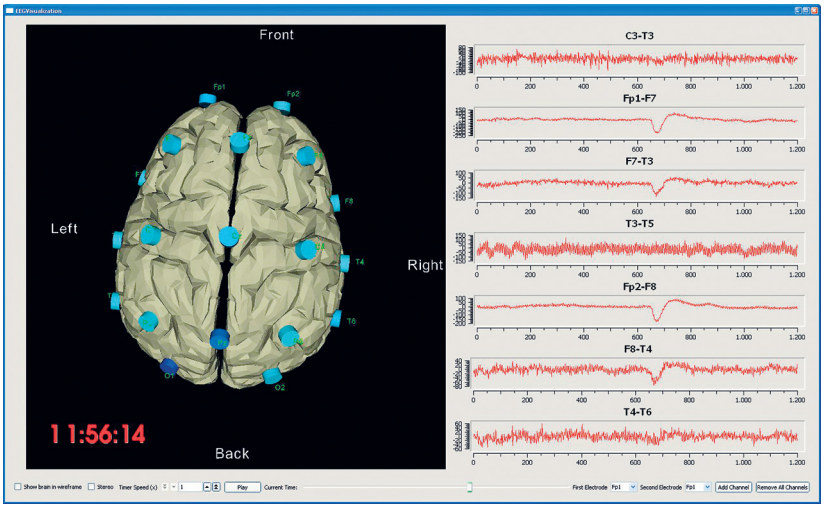


Рис. 10. Визуализация с помощью VTK

2.1.4. ParaView

ParaView — кроссплатформенное приложение с открытым кодом, предназначенное для интерактивной научной визуализации. ParaView построена на базе VTK и может выполняться как на отдельном компьютере с общей памятью, так и на суперкомпьютере с распределенной памятью. ParaView был создан для визуализации огромных массивов данных, и поэтому в нем предусмотрено распараллеливание обработки и использование механизма подстройки уровня детализации к частоте кадра.

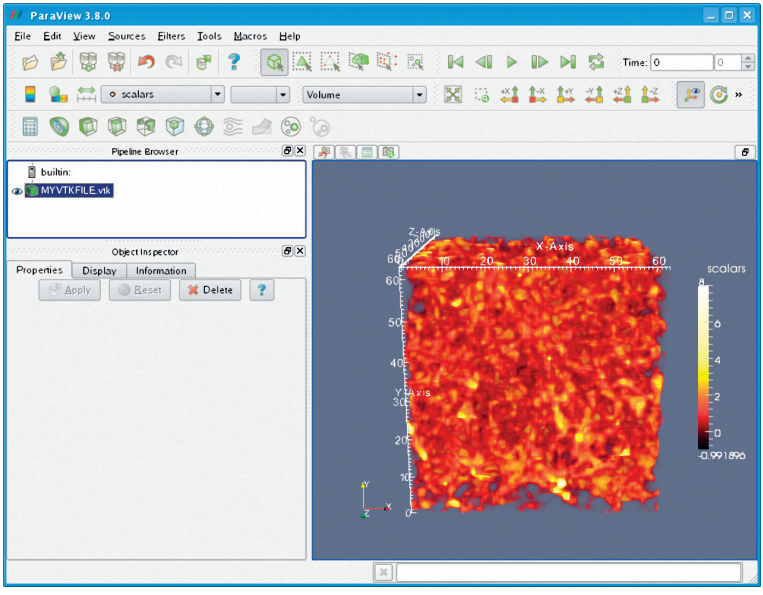


Рис. 11. Визуализация с помощью ParaView

2.1.5. VisIt

VisIt — кроссплатформенное приложение с открытым кодом, предназначенное для интерактивной параллельной визуализации научных данных очень большого размера. VisIt

базируется на библиотеке VTK и имеет средства для визуализации скалярных, векторных и тензорных данных. Поддерживает многочисленные типы сеток. Имеет интеграцию с языками Java, Python и C++, а также возможность расширения через динамически подгружаемые плагины. VisIt был разработан по заказу министерства энергетики США и активно развивается в настоящее время.

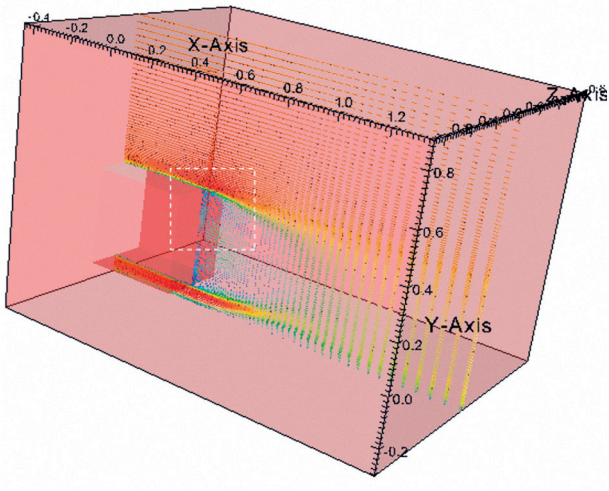


Рис. 12. Визуализация с помощью VisIt

2.1.6. Другие пакеты научной визуализации

Помимо библиотек с открытым кодом существуют и коммерческие пакеты, например, GMV [59] (рис. 13), Techplot [60] (рис. 14), Avizo [61] (рис. 15), а также разработанный российской компанией GDT Software Group пакет Scientific VR [62] (рис. 16). Визуализация с помощью этих пакетов представлена на соответствующих рисунках. Однако все эти пакеты не рассматривались более подробно в силу того, что они не являются пакетами с открытым программным кодом.

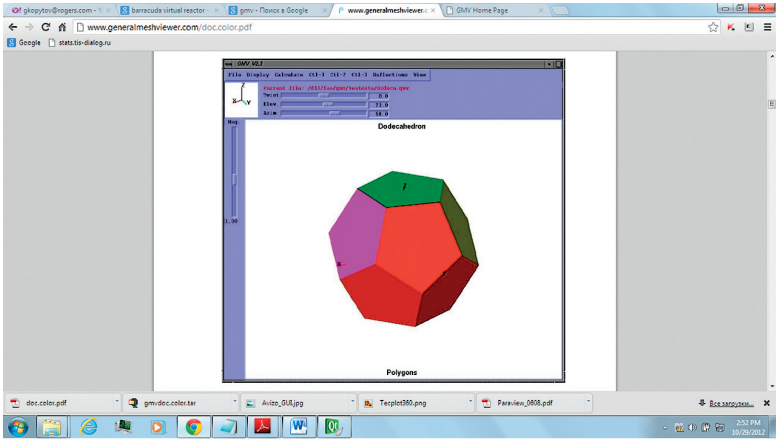


Рис. 13. Визуализация с помощью GMV

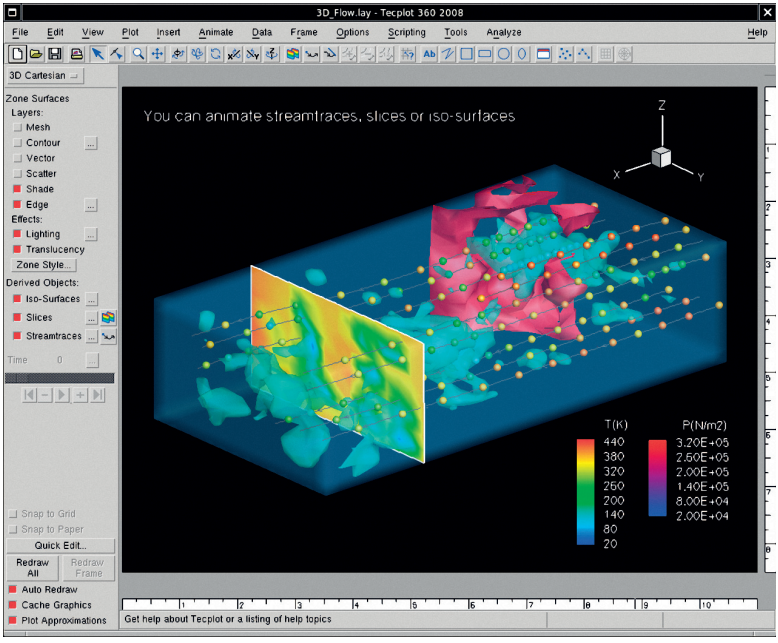


Рис. 14. Визуализация с помощью Techplot

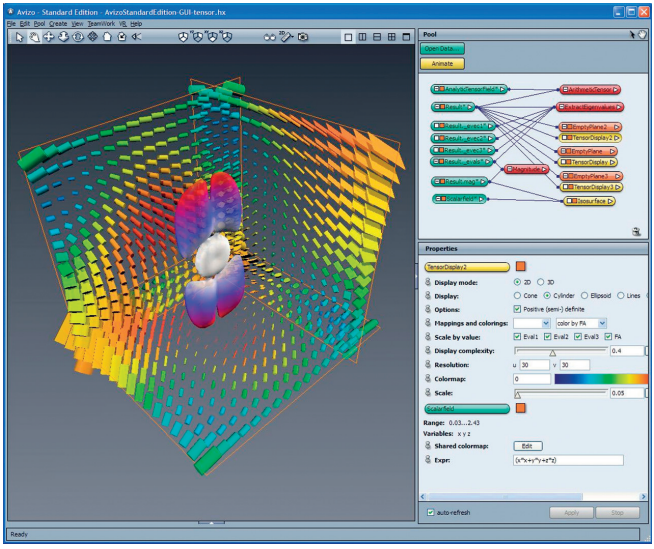


Рис. 15. Визуализация с помощью Aviz

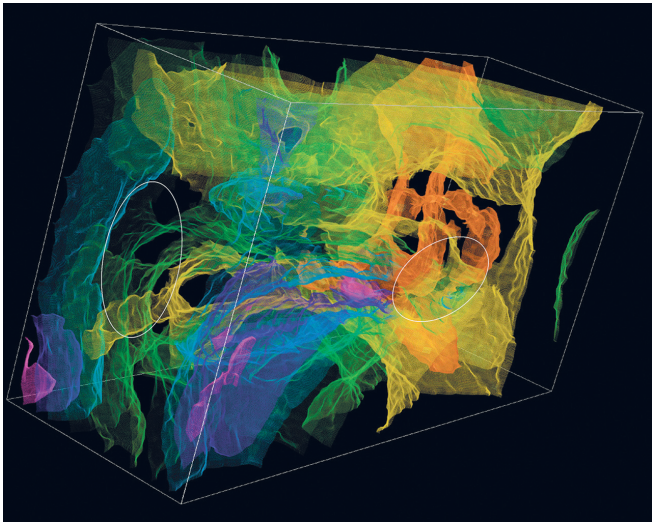


Рис. 16. Визуализация с помощью пакета Scientific VR

Также не являются открытыми отечественные системы научной визуализации: SharpEye [64, 65], разработанная в ИММ УрО РАН, и Пре-пост процессор пакета Логос [66, 67], разработанный в РФЯЦ-ВНИИЭФ. SharpEye — изящная графическая среда, написанная для платформы .NET, которая позволяет создать трехмерную визуализацию из довольно широкого набора входных данных. Пакет Логос предназначен для расчета широкого класса задач математической физики на многопроцессорных ЭВМ и его Пре-пост процессор обладает широким функционалом: взаимодействие с САПР системами, исправление геометрии и расчетных сеток, параллельная визуализация.

2.2. Выбор технологической платформы для создания графической среды численного моделирования

Наилучшим выбором, на наш взгляд, стала платформа Qt [36, 37] в сочетании с пакетом для научной визуализации VTK [32, 38, 39]. Обе удовлетворяют требованиям, перечисленным в начале главы 2, активно развиваются и были использованы в разработке как коммерческих программ, так и программ с открытым кодом. Библиотека OpenDX была исключена потому, что она, по сравнению с VTK, реализует меньшую функциональность как по структурам данных, так и по алгоритмам визуализации, а также перестала активно развиваться в последнее время. Пакеты ParaView и VisIt были отклонены в силу того, что это программное обеспечение для визуализации и анализа уже готовых решений, но они не могут поддержать всю технологическую цепочку от задания модели и параметров расчета, до запуска решателя и визуализации результатов расчетов.

Пакет Visualization Library удовлетворяет всем общим и специальным требованиям, имеет богатые функциональные воз-

можности и активно развивается. Однако предпочтение было отдано пакету VTK, поскольку при прочих равных условиях, он имеет более длительную «историю успеха», а также реализует большее количество алгоритмов визуализации.

2.2.1. Платформа Qt

Qt является кроссплатформенным инструментарием для разработки программного обеспечения на языке C++. Qt разрабатывался с 1996 года сначала компанией TrollTech, а затем компанией Nokia. Отличительной особенностью этого пакета является совместимость на уровне исходных кодов с большинством современных операционных систем, что позволяет запускать написанное на Qt приложение путем простой перекомпиляции программы для каждой ОС. Qt поддерживает ОС Windows, Linux, а также Symbian, которая используется на мобильных устройствах. Qt также интегрирован с другими языками программирования, такими как Python и Java. Этот пакет основывается на языке C++, что позволяет легко подключать библиотеки, написанные на языке Fortran, а это может быть важно для задач численного моделирования. Qt был использован как графическая платформа при создании таких известных приложений, как Google Earth, KDE, Autodesk Maya и других.

Qt имеет удобный инструментарий QtDesigner, который поддерживает технологию быстрой разработки приложений (RAD — Rapid Application Development), называемую иногда визуальным программированием. Это дает все преимущества, связанные с RAD-технологией: значительно ускоряет разработку графических приложений и дает возможность разрабатывать программное обеспечение в условиях нечетко поставленных требований.

Qt не является средой разработки на чистом C++. В отличие от чистого C++, где иерархия объектов является «лесом» (то есть не имеет общего корня), объектная модель Qt является «деревом», в котором есть родовой класс QObject,

а все остальные классы являются его наследниками. Класс QObject обеспечивает поддержку:

- сигналов и слотов;
- механизма объединения объектов в иерархии;
- событий и механизма их фильтрации;
- метаобъектной информации и приведения типов.

Введенный в Qt механизм *слотов* и *сигналов* значительно упростил разработку графических приложений, однако потребовал некоторого расширения синтаксиса языка C++ и введения специального препроцессора MOC (Meta Object Compiler). Слоты и сигналы — это средства, позволяющие эффективно производить обмен информацией о событиях, вырабатываемых объектами, и тем самым соединять не связанные друг с другом объекты. Механизм сигналов и слотов полностью замещает старую модель функций обратного вызова, которая широко использовалась при разработке графических интерфейсов.

Классы, которые используют механизм слотов и сигналов, должны быть предварительно обработаны компилятором MOC, который по описанию класса создает промежуточный код на C++. Затем этот код может быть откомпилирован любым компилятором C++. В состав Qt входит утилита qmake — специальная версия GNU make, которая автоматизирует процесс предварительной компиляции.

Использование механизма слотов и сигналов дает программисту следующие преимущества.

- Каждый класс может иметь любое количество сигналов и слотов;
- Сообщения, посылаемые посредством сигналов, могут иметь множество аргументов различного типа;
- Сигнал может соединяться с любым количеством слотов. Посылаемый сигнал при этом поступает ко всем подсоединенным слотам;
- Слот может принимать сообщения от многих сигналов, посылаемых разными объектами;

- Соединение сигналов и слотов можно производить в любой точке приложения;
- При уничтожении объекта происходит автоматическое разъединение соответствующих сигнально-слотовых связей.

Qt имеет большое количество графических элементов отображения, включая окна с полосой прокрутки, древовидные представления, метки, кнопки и прочее, что избавляет программиста от рутинной работы по написанию стандартных элементов графического интерфейса.

Qt также имеет средства работы с графикой, основанные на стандарте OpenGL. Эти средства включают в себя работу с перьями и кистями, градиенты, отсечения и т. д., что в принципе позволяет использовать Qt для 2D- и 3D-визуализации. Однако эти средства являются базовыми и не в полной мере удовлетворяют тем требованиям к системе визуализации, которые были перечислены выше. Поэтому в дополнение к Qt было решено использовать библиотеку для научной визуализации VTK.

2.2.2. Пакет VTK

Другим базовым элементом графической платформы для численных расчетов является VTK (Visualization Toolkit), который появился в 1993 году и сейчас активно поддерживается и развивается компанией Kitware. VTK представляет собой библиотеку классов C++ с открытым исходным кодом, реализующую многочисленные алгоритмы визуализации, включая скалярные, векторные, тензорные и волюметрические, а также работу с текстурами. VTK предоставляет интерфейсы к таким языкам программирования, как Java, Python и Tcl/Tk. VTK поддерживает различные методы визуализации, такие как явное моделирование, сглаживание сетки, отсечение, обработка контуров и триангуляцию Делоне. Эта библиотека также включает в себя различные 3D-виджеты для информационной визуализации, легко инте-

грируется с Qt, а также является кроссплатформенной. На базе VTK построены различные программы для научной визуализации, такие как 3DSlicer и, упоминавшиеся выше, ParaView, VisIt, а также некоторые другие.

В VTK реализована как абстрактная модель трехмерной графики, так и визуализационная модель, которая представляет собой управляемый потоком данных процесс визуализации (конвейер визуализации).

2.3. Графическая среда и интерфейс пользователя

В этом разделе будет описан набор средств, предоставляемый библиотекой VTK 5.10, которые позволяют нам сделать графическую среду для моделирования физических процессов в трехмерной сцене. Эта среда визуализации использует с Qt 4.8.

2.3.1. Модель 3D-графики

Модель трехмерной графики основана на абстрактной модели, используемой во многих пакетах и библиотеках визуализации, которая также реализована в VTK. Она включает семь основных объектов, которые используются для отрисовки 3D-сцены: окно рисования (класс vtkRenderWindow), рендерер (класс vtkRenderer), освещение (класс vtkLight), камера (класс vtkCamera), 3D-объекты (класс vtkActor), свойства и геометрия объектов (класс vtkMapper).

Окно рисования управляет окном на графическом дисплее и скрывает от пользователя машинно-зависимые особенности реализации графики в той или иной системе. Например, в операционной системе Windows экземпляром этого класса будет окно Windows (класс vtkWin32OpenGLRenderWindow), а на Linux — окно X Window (класс vtkXOpenGLRenderWindow). Окно рисования хранит специфи-

ческие характеристики графических устройств, такие как размер, позиция, глубина окна (количество бит на пиксел), и управляет двойной буферизацией для устранения эффекта мерцания.

Рендерер отвечает за координацию источников света, камеры и объектов 3D-сцены в процессе визуализации. Он должен быть связан с окном рисования (viewport), а внутри него — с прямоугольной областью, в которой происходит процесс отрисовки. По умолчанию рендерер рисует во всем окне рисования. Окно рисования может иметь несколько ассоциированных с ним рендереров. Рендерер поддерживает список 3D-объектов, которые он должен нарисовать, и этот список должен быть не пустым. Визуализация 3D-сцены также требует наличия источников света и камеры, однако, если пользователь не создал их явно, то рендерер создает их автоматически таким образом, что 3D-объекты расположены в центре экрана, а камера направлена вниз по оси *Z*.

Источники света могут быть либо позиционными, то есть располагающиеся в определенной точке пространства и имеющие угол освещения и коэффициент затухания, либо бесконечно удаленными, которые распространяют лучи света параллельно друг другу без затухания. Источник света может находиться во включенном или выключенном состоянии, и рендерер следит за тем, чтобы сцена имела хотя бы один включенный источник света.

3D-сцена должна иметь хотя бы одну активную *камеру*, ссылку на которую можно получить, вызвав метод GetActiveCamera() класса vtkRenderer. Камера характеризуется положением (метод GetPosition() класса vtkCamera), точкой фокусировки (метод GetFocalPoint()), расположением передней и задней плоскостей отсечения (метод GetClippingRange()). Управление камерой осуществляется по трем независимым углам — азимуту (метод Azimuth()), возвышению над горизонтом (метод Elevation()) и углу вращения

(метод Roll()). Кроме того, она предоставляет возможность управления перспективой (метод Zoom()).

3D-объекты (класс vtkActor) сцены имеют геометрию, задаваемую через класс vtkMapper и свойства, задаваемые объектом класса vtkProperty в методе SetProperty(), а также ориентацию в глобальном координатном пространстве, которая задается матрицей трансформаций. Пользователю обычно нет необходимости заботиться о матрице трансформаций — ее объект поддерживает сам, однако задание геометрии объекта и его свойства — это обязанность пользователя. Одним из основных свойств объекта является цвет. Свойства объекта позволяют задать три фактора, которые будут влиять на результирующий цвет объекта — коэффициент диффузного отражения по закону Ламберта (метод SetDiffuse()), если поверхность объекта матовая, коэффициент отражения от глянцевых поверхностей (метод SetSpecular()) и коэффициент рассеянного отражения, который будет учитывать вклад рассеянного света от окружающих объектов (метод SetAmbient()). Кроме того для объекта можно задать его прозрачность методом SetOpacity() в диапазоне от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный).

2.3.2. Модель данных

Модель данных, которая используется для визуализации, основывается на модели данных VTK. В общем виде, модель данных представляется в виде совокупности геометрических данных и топологии. Данные визуализации могут быть регулярными или нерегулярными (другими словами структурированными или неструктурированными). В структурированных данных, например, в сетке с постоянным шагом, геометрия и топология объекта определяются неявно, исходя из природы самих данных. В неструктурированных объектах геометрия и топология задаются явно. Регулярные данные можно хранить более компактно, и их обработка бо-

лее эффективна. Однако нерегулярные структуры данных имеют то преимущество, что мы можем сгущать и разрежать расчетную сетку там, где это необходимо, тем самым увеличивая точность расчетов и экономя вычислительные ресурсы.

Базовый объект модели — набор данных (класс vtkDataSet), который является абстрактным классом и имеет две части — топологию и геометрию. *Топология* — это набор свойств объекта, независимый от геометрических преобразований, то есть вращения, перемещения в пространстве и нелинейного масштабирования. *Геометрия* — это конкретизация топологии в определенных точках трехмерного пространства. Например, «треугольник» — это топология, а треугольник с заданными вершинами — это геометрия. Топология в наборе данных задается через ячейки, а геометрия — через точки.

Типы ячеек. Ячейка (класс vtkCell) задается своим типом и упорядоченным списком точек, который также называется списком связности. Списки связности задают грани ячейки и ее ребра. Ячейка характеризуется типом (всего их более 60), охватывающей границей, а также предоставляет методы для определения, принадлежит ли данная точка ячейке, и позволяет получить список соседствующих ячеек, то есть имеющих общие ребра, вершины или грани с данной ячейкой (метод GetCellNeighbours()). Класс vtkCell является абстрактным классом, который наследует конкретные реализации различных типов 2D- и 3D-ячеек. Важнейшие типы ячеек: линия (vtkLine), ломаная (vtkPolyLine), треугольник (vtkTriangle), четырехугольник (vtkQuad), многоугольник (vtkPolygon), тетраэдр (vtkTetra), призма (треугольная vtkWedge, с пятиугольным основанием vtkPentagonalPrism и с шестиугольным основанием vtkHexagonalPrism), пирамида (vtkPyramid) и шестигранник (vtkHexahedron), который представляет из себя параллелепипед. Последние четыре относятся к 3D-ячейкам,

первые пять — к 2D-ячейкам. Кроме того, поддерживаются типы ячеек, в которых стороны не являются прямыми, а описываются уравнениями второй степени. Все они наследуют класс vtkNonLinearCell.

Атрибуты данных. Данные могут иметь атрибуты (в платформе MSPP атрибут представляется «ярлыком»), например, температура в точке, давление в ячейке или поток тепла, проходящий через грань ячейки. Как правило, атрибуты в наборе данных имеют точки и ячейки, но это также могут быть грани и ребра ячеек или группы точек или ячеек. Атрибуты могут быть скалярами, векторами, нормалями, тензорами или *n*-мерным индексом в карте текстур. *Скаляр* является простейшим атрибутом, приписанным к элементу набора данных и состоящим из одного числа. *Вектор* — это атрибут, который имеет направление и длину и представляется триплетом (*u*, *v*, *w*). Примеры атрибутов-векторов: скорость в точке, функция градиента и т. д. *Нормаль* — это вектор длины 1. Нормали часто используются для контроля затенения объектов, а также в некоторых алгоритмах по управлению ориентацией объекта. Текстурные атрибуты используются для отображения точки в декартовой системе координат в одно-, дву-, или трехмерное пространство текстур, которое иногда называют картой текстур. *Тензор* является довольно сложным атрибутом и представляет собой обобщение многомерного массива. Например, тензор размерности 0 — это скаляр, размерности 1 — это вектор, размерности два — это матрица и т. д. Как атрибут данных, тензор, например, может представлять тензор напряжения в определенной точке твердого тела. В качестве атрибута данных могут выступать только тензоры второго ранга, представимые действительными симметричными матрицами размера 3×3.

Типы наборов данных. Набор данных является абстрактным классом, определяющим общие свойства конкретных наборов данных. Наибольший интерес представляет сетка

с постоянным шагом (класс vtkUniformGrid), сетка с переменным шагом (vtkRectilinearGrid), структурированная сетка (vtkStructuredGrid), набор многоугольников (vtkPolyData) и неструктурированная сетка (vtkUnstructuredGrid), которая является наиболее общим конкретным представлением абстрактного набора данных.

2.3.3. Алгоритмы визуализации

Алгоритмы визуализации преобразовывают наборы данных. Эти преобразования могут затрагивать как топологию, так и геометрию объекта, и могут быть разбиты на категории в зависимости от структуры и типа преобразования. К структурным преобразованиям могут быть отнесены: геометрические преобразования, топологические преобразования, изменения атрибутов и комплексные преобразования. Геометрические преобразования изменяют геометрию объекта, но не трогают топологию. К примерам таких преобразований можно отнести вращение объекта, перемещение в пространстве, масштабирование. Топологические преобразования, напротив, изменяют топологию объекта, но не трогают его геометрию. Преобразования атрибутов затрагивают только изменение или создание новых атрибутов данных, оставляя общую структуру объекта (т. е. топологию и геометрию) неизменной. Комплексные преобразования затрагивают и структуру, и атрибуты данных.

По типу алгоритмы визуализации разделяются на скалярные, векторные, тензорные и алгоритмы моделирования. Основными *скалярными алгоритмами* является раскраска цветом (color mapping) и построение изолиний (рис. 17). Для раскраски цветом необходимо использовать метод ColorByArrayComponent() класса vtkMapper, в котором указывается номер атрибута, который будет определять цвет. Однако цветовая палитра должна задаваться пользователем самостоятельно. Цветовая палитра — объект класса vtkLookupTable, который позволяет задать коли-

чество цветов в палитре и их диапазон, а также диапазон изображаемого атрибута. Палитра может быть задана либо вручную, либо быть сгенерирована автоматически в виде линейной или логарифмической палитры. Цвет палитры поддерживает альфа канал, то есть может быть сделан полупрозрачным.

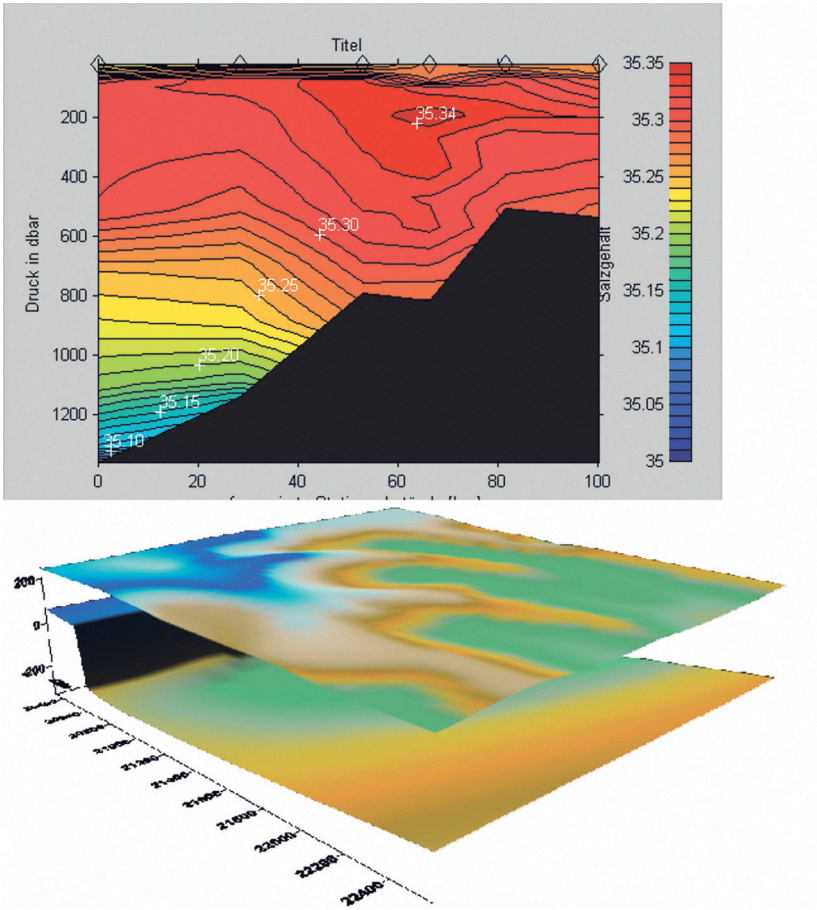


Рис. 17. Изолинии и раскраска цветом

При построении изолиний или изоповерхностей используется алгоритм марширующих квадратов в 2D-случае или алгоритм марширующих кубов в 3D-случае. Алгоритм марширующих кубов был предложен в 1987 году в работе [40] и был запатентован, однако срок патента истек в 2005 году. Для генерации изолиний или изоповерхностей используется класс vtkContourFilter, который применяется к исходному набору данных.

Обобщением перечисленных выше двух скалярных алгоритмов является метод, в котором скалярная величина генерируется из нескалярных с использованием явного алгоритма. Например, вычисляется скалярное произведение вектора скорости в точке на вектор нормали к поверхности в этой же точке. Затем полученная скалярная величина используется для раскраски цветом или отрисовки изолиний. В этом случае метод вычисления скалярной величины существенно зависит от физики задачи и от того, какие именно характеристики объекта исследователь хочет визуализировать.

К более сложным скалярным алгоритмам относятся алгоритм разделяющих кубов [41], ковровые графики (carpet plots) и отсечение с помощью скалярных величин.

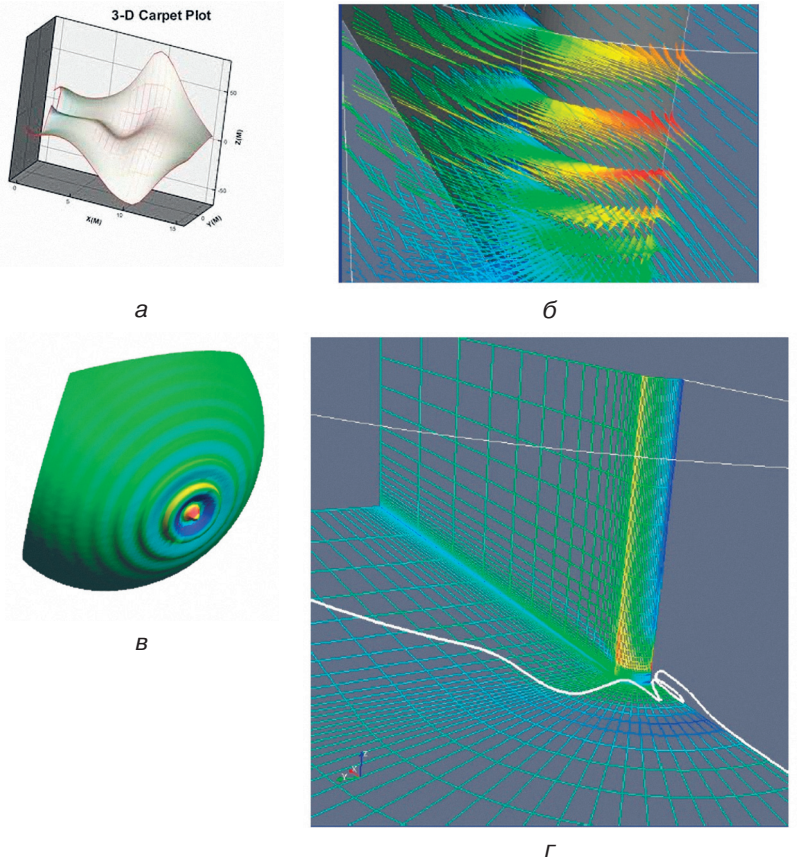
Алгоритм разделяющих кубов похож на алгоритм марширующих кубов, с той лишь разницей, что когда объемный элемент (воксель) пересекается изоповерхностью, то поверхность пересечения проецируется на плоскость. Если проекция меньше или равна одному пикселю, то она визуализируется как точка, иначе к объему применяется алгоритм марширующих кубов, которые разделяют его на более мелкие объемы. Алгоритм разделяющих кубов был первоначально разработан для визуализации объемных данных (компьютерная томография, магнитно-резонансная томография), однако он может применяться к другим наборам данных путем разделения их по параметрическим координатам. Алгоритм разделяющих кубов реализуется классом vtkDividingCubes и требует задания значения изоповерхности и окрестности различения точек. Чем более плотные об-

лака точек на изоповерхности необходимо сгенерировать, тем меньшую окрестность различения точек надо задать.

Ковровые графики (carpet plots, рис. 18а) применяются для визуализации двумерных данных, связанных со скалярной величиной. Для их построения используется искривление двумерной поверхности в направлении нормали к поверхности, причем величина искривления зависит от скалярной величины. Обычно ковровые графики применяются к изображениям, однако они также могут быть применены к двумерным структурированным или неструктурированным сеткам. Алгоритм создания ковровых графиков реализуется классом vtkWarpScalar. Он позволяет выгнуть двумерные данные в направлении нормали к поверхности с указанным коэффициентом масштабирования. Если нормали не присутствуют в данных, то класс vtkWarpScalar позволяет задать одну нормаль для всех точек в наборе данных, в направлении которой будет выполняться искривление поверхности.

Механизм отсечения широко используется в компьютерной графике. Однако он применяется, в основном, для геометрических отсечений. Отсечение с использованием скалярных величин позволяет производить более сложные виды отсечений. Например, можно производить отсечение на основе скалярных атрибутов, ассоциированных с набором данных, тем самым выявляя внутренние части общей структуры, объединенные некоторым признаком. Например, можно выделить из общей структуры все точки с заданной температурой. В силу того, что система визуализации не делает никаких предположений о физической природе скалярных атрибутов в наборе данных, метод отсечения с использованием скалярных величин может быть довольно мощным инструментом анализа и интерпретации данных. Алгоритм отсечения реализуется классом vtkClipPolyData. Он позволяет производить как геометрическое, так и скалярное отсечение, причем скалярная величина может быть как атрибутом данных, так и вычисляться заданной пользователем функцией.

Векторные алгоритмы включают в себя изображение в виде векторного поля (класс vtkGlyph3D, рис. 18б), алгоритм искривления, в котором поверхность искривляется в соответствии с векторным полем (класс vtkWarpVector, рис. 18в), графики смещения, когда изображается величина смеще-



а *б* *в* *г*

Рис. 18. Примеры работы алгоритмов визуализации:
а) ковровый график; б) векторное поле;
в) искривление поверхности; г) траектории частиц

ния точки в направлении перпендикулярном к поверхности (класс vtkVectorDot), а также алгоритмы вычисления траекторий частиц (класс vtkStreamTracer, рис. 18г). Алгоритмы вычисления и визуализации траекторий частиц весьма наглядны, однако они не показывают двух важных характеристик векторного поля — завихренности и дивергенции. Для этой цели разработаны «ленточные» алгоритмы, в которых несколько близко лежащих траекторий объединяются в ленту с плоской поверхностью. При этом степень закрученности ленты в пространстве характеризует вихрь векторного поля, а ширина ленты пропорциональна дивергенции поля. Ленточный алгоритм реализован в классе vtkRibbonFilter.

Однако вихрь и дивергенция не являются единственными характеристиками векторного поля. Техника «потока многоугольников» (stream polygon), разработанная Шредером с соавторами в 1991 году [42], позволяет визуализировать такие локальные характеристики векторного поля, как напряжение, смещение и вращение. Если опустить математическую подоплеку, то суть этого метода заключается в том, чтобы поместить n-угольник в заданную точку векторного поля и затем проследить его движение и деформацию под влиянием локальных характеристик поля. Класс vtkStreamTube в сочетании с vtkStreamLine реализует этот алгоритм.

Особая техника визуализации разработана для векторных полей, имеющих критические точки [43, 44]. В критической точке величина вектора равна нулю и, следовательно, его направление не определено. Поведение векторного поля в окрестности критической точки характеризуется собственными числами якобиана, составленного из частных производных компонентов вектора скорости по x, y, z в этой точке. В общем случае собственное значение якобиана является комплексным числом, в котором мнимая часть описывает вращение векторного поля вокруг критической точки, а действительная часть определяет притяжение или отталкивание поля от критической точки. Визуализация вектор-

ного поля с критическими точками включает несколько шагов. Во-первых, определяются сами критические точки в поле методом деления отрезка пополам, и в них вычисляются собственные числа якобиана для определения характера поведения поля в окрестности критической точки. Затем все векторное поле делится на области, в которых поле ведет себя топологически эквивалентно стационарному течению. Критические точки являются точками отрыва или притяжения поля и соединяются линиями потоков (streamlines). Техника выделения критических точек и определение областей квази-стационарного поведения поля является важным аналитическим инструментом для понимания общей структуры векторного поля. Аналогичная техника может быть использована для анализа взаимодействия поля и твердых поверхностей. В этом случае точками отрыва и притяжения уже являются не критические точки поля, а те точки на поверхности, в которых касательная вектора поля равна нулю и, следовательно, поток движется перпендикулярно поверхности.

Тензорные алгоритмы. Тензорные атрибуты данных часто возникают в задачах, в которых используются тензоры деформаций или напряжений. Поскольку в качестве атрибута данных допускается только действительная симметричная матрица размером 3×3 (тензор ранга 2), то ее собственные значения всегда действительны, а собственные векторы ортогональны друг другу, и из них можно составить ортонормальный базис. Следовательно, в каждой точке, имеющей тензорный атрибут, можно построить эллипсоид с центром в этой точке и ориентированный по направлениям собственных векторов тензора. Этот факт и используется в алгоритмах визуализации тензоров. Один из них описан в работе [45]. Он строит тензорный эллипсоид в заданной точке (x, y, z), а затем проводит линию потока, двигаясь в направлении одного из собственных векторов эллипсоида. Линия потока представляет в этом случае трубку с перемен-

ным сечением, которая не только характеризуется своим положением в трехмерном пространстве, но также обладает разной толщиной, что является дополнительной характеристикой линии потока. По этой причине подобная техника визуализации получила название гиперлиний потока. Реализация гиперлиний потока сделана с помощью классов vtkHyperStreamline и vtkTensorGlyph.

Моделирующие алгоритмы. Моделирующие алгоритмы включают в себя такие общепризнанные алгоритмы визуализации данных, такие как как

(а) выделение данных по геометрическому признаку;
(б) взятие образцов (sampling), то есть отбор каждой *n*-й вершины или грани;
(в) выделение данных по пороговому значению атрибута;
(г) зондирование (probing), то есть преобразование входного набора данных по вершинам другого набора;
(д) генерация полос треугольников (triangle strip);
(е) выделение поднабора данных по признаку связности (например, все точки одной изолинии);
(ж) генерация нормалей.

Кроме того, в число моделирующих алгоритмов включены:

- алгоритм децимации, предложенный Шредером [46] и предназначенный для уменьшения общего количества треугольников в сетке;
- алгоритм сглаживания сетки, предназначенный для улучшения качества сетки, и использующий сглаживание Лапласа.

Кроме того, поддерживается алгоритм триангуляции Делоне как для 2D-, так и для 3D-случая.

2.3.4. Взаимодействие с пользователем

Важной частью любой системы визуализации является взаимодействие с пользователем. Он должен иметь возможность перемещать, вращать, увеличивать или уменьшать изображение на экране, менять позицию, с которой рассмат-

ривается изображение, точку фокусировки и т. д. В VTK это реализовано с помощью так называемых стилей взаимодействия (класс vtkInteractorStyle). По умолчанию принят стиль взаимодействия (класс vtkInteractorStyleTrackballCamera), который выполняет наиболее распространенные действия, такие как вращение, перемещение и масштабирование объектов с помощью мыши, и более редкие операции, например, сброс камеры в начальное положение с помощью нажатия клавиш. При этом возможно переключение режимов, при которых мышь управляет движением объекта или камеры, а также определяется, как интерпретировать нажатия клавиш и движения мыши. Кроме того, стиль взаимодействия, принятый по умолчанию, реализует несколько других полезных функций, например, включение или отключение синхронизации фокуса камеры и расположения источника освещения или переключение режима отображения объектов с каркасного на поверхностный. Существуют и более специализированные режимы взаимодействия и визуализации, которыми также можно управлять с помощью стандартного стиля взаимодействия. К примеру, если поверхность объекта содержит большое количество многоугольников, например, сотни тысяч или миллионы, то его визуализация неизбежно становится медленной. При такой ситуации возможно включение режима, который будет балансировать точность отрисовки деталей с частотой обновления кадра, и установить желаемую частоту обновления кадра.

Механизм взаимодействия с пользователем должен давать возможность его перенастройки. Например, если в трехмерном окне важна возможность вращать объект, то в двумерном, напротив, эта опция должна быть отключена. При этом перемещение объекта по экрану и масштабирование остается желательным. Перенастройка механизма взаимодействия осуществляется несколькими способами. Во-первых, можно написать подкласс стандартного сти-

ля взаимодействия и переопределить необходимые (или все) функции. Часто в этом нет необходимости, поскольку и так существует довольно большое число переопределенных стилей, например, стиль взаимодействия в 2D-окне (класс vtkInteractorStyleRubberBand2D), описанный выше, можно установить в окне отрисовки. Во-вторых, можно использовать механизм обратных вызовов и определить пользовательские функции, которые будут вызываться главным циклом обработки событий при возникновении системных (например, нажатие левой клавиши мыши) или пользовательских (например, выделение элемента трехмерного изображения) событий. Механизм обратных вызовов дает неограниченную возможность перенастройки способа взаимодействия с пользователем, хотя он нужен лишь в особо сложных случаях, которые не покрываются возможностью выбора или переопределения стиля взаимодействия.

Другой важной чертой механизма взаимодействия с пользователем является выбор объекта или детали объекта на 2D- или 3D-сцене. Выбор объекта на 2D-сцене обычно не вызывает больших затруднений, хотя даже в этом случае есть нюансы. Например, если пользователь кликнул мышью на сегмент ломаной линии, что он имел ввиду — выбрать лишь этот сегмент или всю ломаную линию? В случае трехмерной сцены задача еще более усложняется, потому что трехмерная сцена изображена на экране в двумерном виде, и у пользователя нет возможности указать глубину выбора в третьем измерении. Для решения этой задачи реализован общий механизм взаимодействия рендерера и компонента, который отвечает за выбор объекта. Для всех компонентов выбора определен абстрактный класс vtkAbstractPicker, который отвечает за интерфейс взаимодействия этих компонентов с рендерером. Каждый рендерер имеет ассоциированный компонент выбора, который реализует данный интерфейс. Главным методом этого интерфейса является метод Pick(), который вызывается при позиционировании на экране

и нажатии пользователем левой кнопки мыши. В методе Pick() компонент взаимодействия с пользователем сообщает компоненту выбора координаты точки *x*, *y*, *z*, причем *z* передается как 0.0 и дает ссылку на текущий рендерер. Ссылка на рендерер необходима, чтобы компонент выбора мог просмотреть список объектов, связанных с рендерером, и по координатам *x*, *y*, *z* решить, какой объект выбран. Существует конкретная реализация абстрактного компонента выбора — класс vtkPicker. Он пускает луч из камеры в точку выбора и вычисляет точки пересечения луча с охватывающими параллелепипедами для всех объектов сцены. После этого выбирает ближайший к камере объект. Эта реализация в свою очередь имеет два подкласса vtkPointPicker и vtkCellPicker, один из которых выбирает ближайшую точку в выбранном объекте и сообщает ее идентификатор, а другой выбирает наиболее подходящую грань и тоже сообщает ее идентификатор. При необходимости делать выбор объекта на основе других критериев, а это может зависеть от конкретной сцены, режима работы программы и т. д., программист должен написать собственный подкласс, реализующий интерфейс абстрактного компонента выбора, и установить его в качестве рабочего в компоненте, отвечающей за взаимодействие с пользователем в текущем окне рисования.

2.4. Особенности визуализации при расчетах на суперкомпьютерах

Проведение научного эксперимента на современных суперкомпьютерах имеет две отличительные особенности. Во-первых, расчет требует длительного времени, и в связи с этим возникает необходимость в промежуточной визуализации данных. Во-вторых, объем данных, производимых в расчете, как правило, очень большой. Поэтому реализация классической схемы, в которой конечный пользователь

готовит модель на своей машине, затем передает данные на суперкомпьютер для расчета, а потом принимает результаты вычислений на свою машину и там начинает визуализацию, в некоторых случаях может оказаться неэффективной. Если суперкомпьютер является удаленным, и конечный пользователь связан с ним через интернет каналом с пропускной способностью не выше 100 Мб/с, то ситуация усугубляется.

В этом случае более логичной является схема, в которой на стороне суперкомпьютера находится интерфейсная машина, связанная с суперкомпьютером высокоскоростным каналом, которая, с одной стороны, осуществляет связь с графическим интерфейсом пользователя, а с другой, – распределяет нагрузку на узлы суперкомпьютера при проведении научного эксперимента, и собирает данные расчетов в промежуточном хранилище. Поскольку одной из схем организации параллельных вычислений является схема «начальник—подчиненный», то интерфейсная машина естественным образом берет на себя функции «начальника». Это означает, что она является не выделенной управляющей машиной, стоящей сбоку от суперкомпьютера, а одним из узлов суперкомпьютера, связанного с ансамблем вычислительных процессов в рамках одного MPI коммуникатора. Это, во-первых, позволяет получать промежуточные результаты вычислений от процессов-вычислителей, и, во-вторых, дает возможность пользователю интерактивно вмешиваться в процесс вычислений, например, приостанавливать вычисления или перезапускать с новыми значениями граничных параметров. Третьей функцией управляющей машины становится организация параллельной обработки данных для визуализации и анализа на конечной машине пользователя. Тем самым решается проблема передачи больших объемов информации по медленным каналам между суперкомпьютером и машиной пользователя. Кроме того, поскольку алгоритмы визуализации могут быть достаточно ресурсоемкими, то их распараллеливание на супер-

компьютере дает дополнительные выгоды для уменьшения времени обработки (тем самым повышая интерактивность) и использования больших объемов распределенной оперативной памяти суперкомпьютера, что затруднительно сделать на машине пользователя.

Рассмотрим конкретный пример такой ресурсоемкой визуализации, проводимой на суперкомпьютере. Далее будет описана параллельная реализация алгоритма трассировки лучей для получения изображения высокой четкости на основе пакета Povray 3.6.1 и дана оценка эффективности и ускорения параллельной обработки изображения течений со свободной поверхностью [47, 48]. Метод трассировки лучей является вычислительно трудоемким, поэтому процесс визуализации может занимать значительную часть времени расчета. Для ускорения работы пакета Povray была предложена и реализована эффективная параллелизация данного пакета на основе технологии MPI, что позволило получить ускорение, почти пропорциональное числу используемых процессоров.

Параллельная реализация пакета Povray 3.6.1 основана на модели «начальник—подчиненный» Процессор с нулевым рангом помечается как «начальник» и выполняет задачу по распределению и приему обработанных частей изображения.

На исходную область изображения размеров $N \times M$ пикселей накладываются квадратные подобласти размером $K \times K$ пикселей. Если N или M не кратно K, то крайние подобласти обрезаются. Таким образом, исходная задача делится на $[N/K] \times [M/K]$ подзадач. В данной работе параметр K равен 16.

Процессор-«начальник» распределяет среди «подчиненных» по L подзадач и ожидает выполненную работу при помощи синхронной операции приема «MPI_Recv».

После выполнения приема результат записывается во временный массив и, если процессор-«подчиненный» закончил свои L задач, «начальник» отправляет новые L задач при помощи асинхронной команды «MPI_Isend». В опи-

сываемом исследовании параметр L был равен 15. Подзадачи можно передавать как в порядке возрастания их номеров, так и в произвольном порядке. Для определенности, в данной работе подзадачи передавались по возрастанию их номеров.

Затем данные из временного массива записываются в конечное изображение.

Для выполнения асинхронных пересылок сделана многократная буферизация данных, то есть перед очередной пересылкой процессор проверяет, свободен ли очередной буфер отправки при помощи команды «MPI_Wait», затем записывает данные в этот буфер и вызывает команду на асинхронную отправку «MPI_Isend». Для отправки новых задач процессор-«начальник» хранит массив из $B_M = 256$ буферов. Аналогичный метод многократной буферизации в сочетании с асинхронными пересылками используется для отправки выполненных данных процессором-«подчиненным», массив состоит из $B_S = 16$ буферов.

В итоге процессор-«начальник» выполняет следующие шаги:

1. синхронная рассылка первоначальных номеров подзадач всем узлам;
2. при получении выполненной подзадачи:
   (a) прием подзадачи во временный массив;
   (b) асинхронная отправка номеров новых подзадач (или -1, если новых подзадач нет) с текущего буфера, если «подчиненный» выполнил все подзадачи;
   (c) ожидание следующего буфера отправки номеров подзадач;
   (d) переключение на следующий буфер отправки номеров подзадач;
   (e) запись пикселей из временного массива в конечную картинку;
   (f) при наличии графической оболочки — вывод полученного массива пикселей на экран;

3. если все подзадачи выполнены, то производится запись картинки в файл и выход, иначе — переход к пункту 1.

Процессоры-«подчиненные» одновременно выполняют следующие шаги:

1) прием номеров подзадач;
2) выполнение текущей подзадачи, если она имеет номер −1, то ожидание отправки всех буферов и выход;
3) асинхронная отправка выполненной подзадачи с текущего буфера;
4) ожидание следующего буфера хранения выполненной подзадачи;
5) переключение на следующий буфер хранения выполненной подзадачи;
6) если все подзадачи выполнены, то ожидание новых номеров подзадач, переход к пункту 2.

Такой комплексный подход с использованием

1) модели «начальник—подчиненный»,
2) асинхронных пересылок,
3) многократных буферизаций,
4) выполнения «подчиненным» по несколько подзадач

направлен как на то, чтобы получить адаптивную балансировку загрузки узлов, так и на то, чтобы сгладить задержку, вызванную латентностью сети. Балансировка загрузки обеспечивается благодаря тому, что процессор, загруженный областью, требующей большего времени расчета, не будет получать новых подзадач, пока не завершит выполнение своих. Таким образом, подзадачи будут распределяться между менее загруженными процессорами. Латентность сети снижается благодаря множественной буферизации и тому, что асинхронные пересылки, где возможно, производятся одновременно с расчетом.

Параллельная версия тестировалась на кластере, состоящем из 23 узлов, с двумя 4-ядерными процессорами Intel

Xeon 2.6 Гц в каждом. Тест проводился на сцене, результат расчета которой изображен на рис. 19. Целью теста было проверить эффективность получения изображения размера 4096×3072 пикселей в зависимости от количества использованных процессоров. Наравне с обычными поверхностями, сцена имеет большое число геометрически сложных отражающих поверхностей, преломляющих областей, а также областей с рефракцией. Тем самым, эта задача является хорошим тестом на сбалансированность работы процессоров. Следует отметить, что часть времени уходит на загрузку самой сцены.

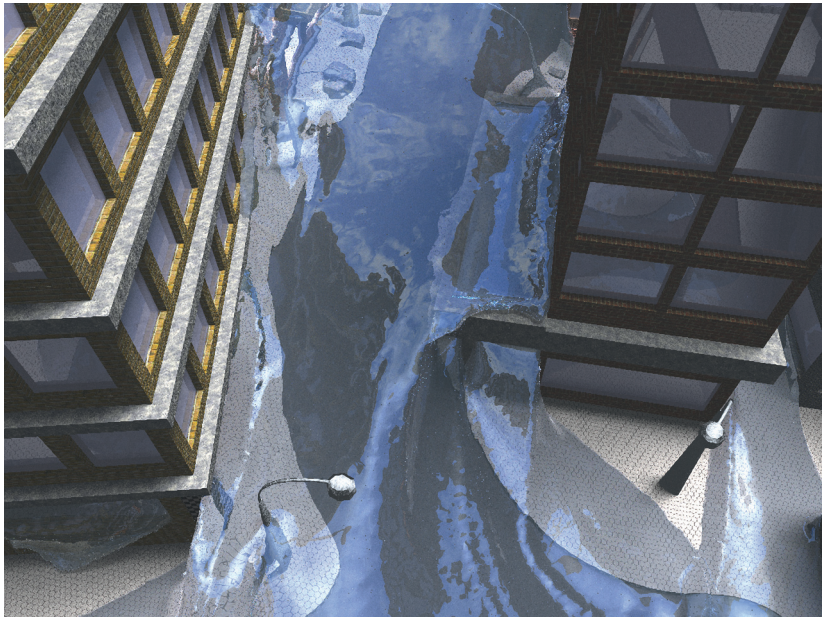


Рис. 19. Задача о затоплении города

Ускорение, полученное при визуализации с разрешением 4096×3072 для сцены, изображенной на рис. 19, представлено в таблице 2. Приведенная таблица демонстрирует

хорошую эффективность выполненной параллельной реализации пакета Povray 3.6.1 при визуализации сложных сцен.

Таблица 2

Время визуализации для одного кадра 4096×3072 (в секундах)

Число вычислительных ядер	15	30	60	120
Время в секундах	367	195	107	62

Глава 3

Применение технологического комплекса INMOST для решения задач геофильтрации

В данной главе мы опишем применение технологий INMOST в разработке программного комплекса GeRa (геомиграция радионуклидов) для решения задач геофильтрации, необходимого при моделировании геомиграции радионуклидов из захоронений в глубоких геологических формациях. Программная платформа и графическая среда комплекса INMOST положены в основу технологической цепочки, реализующей расчет геофильтрационных потоков в слоистых средах.

3.1. Постановка задачи геофильтрации

Рассмотрим задачу геофильтрации в слоистой среде с характеристиками, аналогичными характеристикам одного из полигонов захоронения радиоактивных отходов. На рис. 20 показана структура расчетной области: выделены 11 геологических слоев, обладающих различными коэффициентами фильтрации (таблица 3). В области существует геологический разлом, вдоль которого имеется смещение геологических слоев. Слои могут вырождаться, формируя так называемые выклинивания слоев. Процесс фильтрации в насыщенных средах описывается уравнением диффузии с тензорным коэффициентом диффузии (фильтрации) в диффузионном операторе div K grad. Неизвестной величиной является функция напора, диффузионный поток от которого равен вектору скорости фильтрации с обратным знаком.

Таблица 3

Свойства геологических слоев

№	Слой	Горизонтальный коэффициент фильтрации K_{xy}, м/сут	Вертикальный коэффициент фильтрации K_z, м/сут
1	A	0,00001	0,0001
2	I	0,1	0,1
3	B	0,001	0,0001
4	F	0,001	0,0001
5	V	0,001	0,0001
6	II	1,15	1,15
7—8	G	0,001	0,0005
9	III	0,26	0,26
10	D	0,001	0,0005
11	Q	2	1

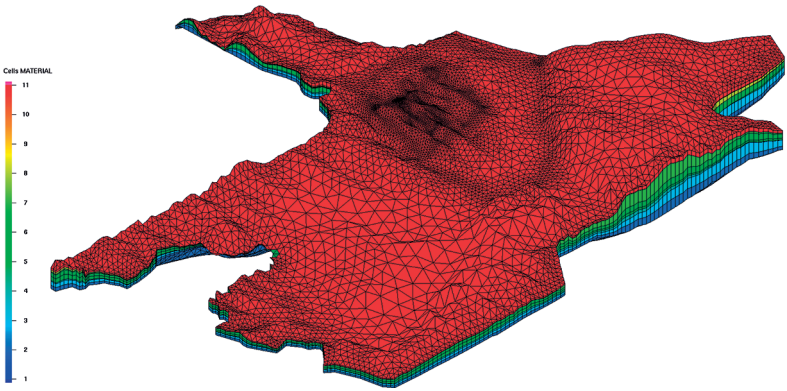
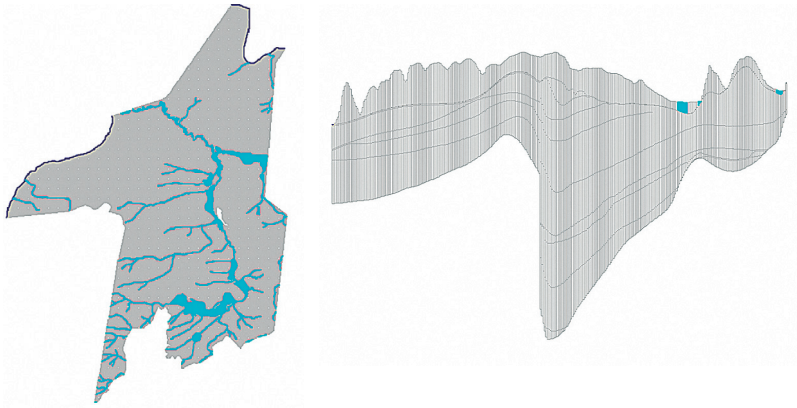


Рис. 20. Структура расчетной области.
Цветом отмечены различные геологические слои

Граничные условия для диффузионного уравнения диктуются физическими соображениями. На частях границы, совпадающих с реками, задан фиксированный напор

(рис. 21а, реки отмечены синими линиями). На остальной границе заданы диффузионные потоки. Через верхнюю кровлю задана постоянная инфильтрация, включающая осадки, испарения и отток с учетом поверхностных рек. Боковые и нижняя границы расчетной области (рис. 21б) считаются непроницаемыми.



(а) *(б)*

Рис. 21. Схема граничных условий для расчетной области: через верхнюю кровлю задан фиксированный поток, боковые и нижняя границы области непроницаемы, отток осуществляется в реки с заданными напорами

Проиллюстрируем на примере этой задачи все этапы расчета геофильтрационных потоков в рамках программного комплекса GeRa.

3.2. Технологическая цепочка приближенного решения задачи геофильтрации

Как было сказано выше, простейшее уравнение геофильтрации представляет собой стационарное уравнение диффузии с неоднородным тензорным коэффициентом диффу-

зии и смешанными краевыми условиями. Технологическая цепочка приближенного решения этого уравнения состоит из следующей последовательности действий:

- задание расчетной области и задание в ней параметров задачи;
- построение расчетной сетки;
- построение дискретизации краевой задачи и формирование системы сеточных уравнений;
- решение системы сеточных уравнений;
- визуализация и анализ сеточного решения.

В рамках программного комплекса GeRa на первом этапе создается геологическая модель, которая представляет собой трехмерную пространственную модель слоистой среды и пространственное распределение ряда параметров задачи, таких как коэффициент фильтрации и коэффициенты краевых условий. Трехмерная пространственная модель определяется геологическими слоями, ограниченными кровлями и подошвами, разломами и плановыми (т. е. заданными в горизонтальной плоскости) границами (рис. 22).

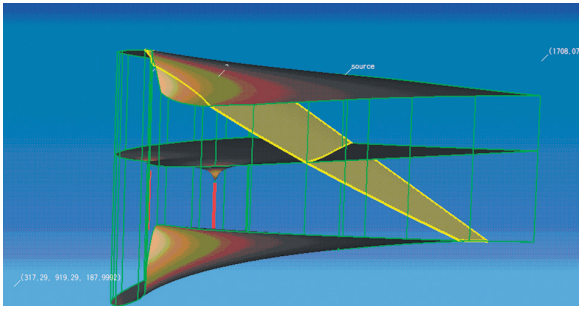


Рис. 22. Кровли и подошвы геологических слоев, разлом и плановые границы

На втором этапе строится расчетная треугольно-призматическая сетка. У границы области и в окрестности выклинивания слоев допускается вырождение призм в пирамиды

и тетраэдры. На рис. 23 представлен пример треугольно-призматической сетки для одиннадцати геологических слоев в расчетной области. Для удобства отображения расчетная область растянута в 15 раз по вертикали. Построенная сетка содержит 117 829 ячеек.

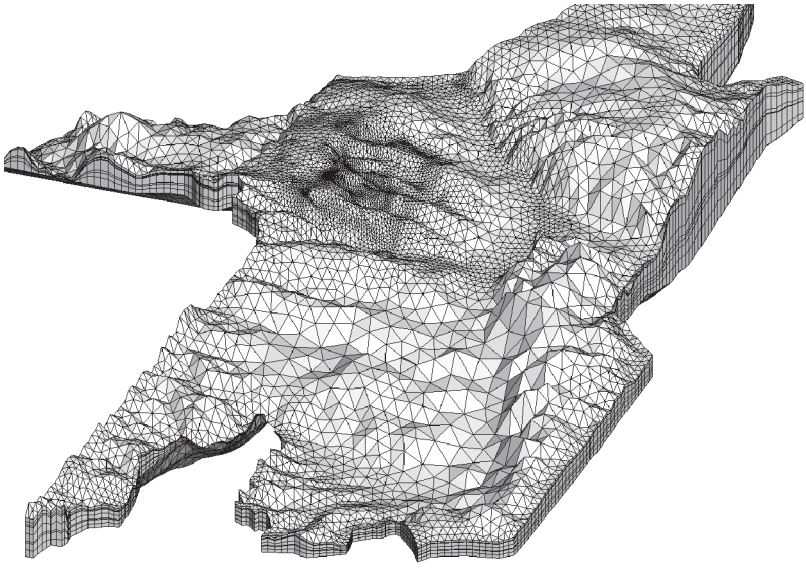


Рис. 23. Треугольно-призматическая сетка в расчетной области, содержащая 117 829 ячеек

Локально адаптированная триангуляция, лежащая в основании трехмерной сетки, представлена на рис. 24.

На третьем этапе применяется консервативный метод дискретизации уравнения диффузии на построенной сетке. В качестве дискретизации используется монотонный нелинейный метод конечных объемов [50, 51, 52, 53]. Метод обладает рядом важных свойств. Во-первых, метод имеет свойство монотонности в слабом смысле, т. е. сохраняет неотрицательность дискретного решения, что особенно важно

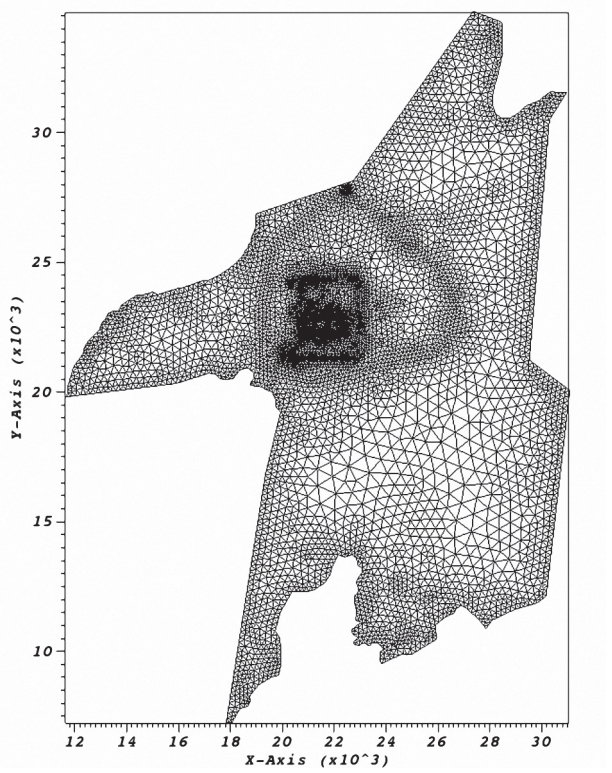


Рис. 24. Неструктурированная треугольная сетка в основании треугольно-призматической сетки

при решении задач геомиграции. Отметим, что дальнейшее развитие метода [49] обеспечивает монотонность в сильном смысле, т. е. гарантирует выполнение дискретного принципа максимума. Во-вторых, шаблон дискретизации диффузионного потока двухточечный $\mathbf{q}_f^h \cdot \mathbf{n}_f = M_f^+ C_{r_+} - M_f^- C_{r_-}$, что позволяет получать предельно компактный шаблон сеточного диффузионного оператора. Коэффициенты шаблона зависят от взаиморасположения ячеек вокруг грани f и зна-

чений напора в этих ячейках, поэтому метод является нелинейным. В-третьих, метод обеспечивает сходимость второго порядка на гладком решении при любых тензорах диффузии и расчетных сетках с многогранными ячейками произвольной формы. Таким образом, нелинейная схема позволяет получать качественное решение с высокой точностью для широкого класса задач, сохраняя при этом максимально компактный шаблон.

На четвертом этапе происходит решение сеточных систем. Поскольку дискретизации потоков через грани являются нелинейными, итоговая система уравнений также будет нелинейной: $M(C)C = G(C)$. Для решения нелинейной системы используется двухуровневый итерационный метод: на внешнем уровне применяется метод Пикара $M(C^k)C^{k+1} = G(C^k)$, на внутреннем уровне решаются линейные системы с линеаризованным сеточным оператором диффузии.

На заключительном этапе происходит визуализация и анализ результатов расчета.

3.3. Использование функционала программной платформы при создании программного комплекса GeRa

Функционал программной платформы задействуется при построении дискретизации краевой задачи, а также при формировании и решении сеточных уравнений.

Рассмотрим вариант, когда задача решается в последовательном режиме. Сначала платформа загружает построенную сетку вместе с поставленными граничными условиями и коэффициентами фильтрации. Граничные условия и коэффициенты записываются сеточным генератором согласно заданной геологической модели и сохраняются в элементах сетки посредством механизма ярлыков (класс Tag).

Затем мы можем подготовить сетку для вычисления нелинейных потоков, заранее рассчитав шаблон и коэффициенты, необходимые для получения двухточечного шаблона. Для этого используем итератор по всем граням, запрашиваем соседние ячейки и находим коэффициенты. Полученные коэффициенты и указатели на ячейки, которым эти коэффициенты соответствуют, также запоминаем в элементах посредством механизма ярлыков.

Далее на каждой нелинейной итерации нам требуется заполнить матрицу $M(C^k)$ и вектор правой части $G(C^k)$. Заполнение производится посредством предлагаемых платформой классов Solver::Matrix и Solver::Vector. Для этих целей сначала пронумеруем все ячейки, чтобы однозначно определить, какая строка матрицы и какая позиция в векторе соответствует каждой ячейке. Так как в конечно-объемной дискретизации рассматриваются потоки через грани, то запустим итерацию по всем граням:

- во внутренних гранях вычислим коэффициенты M_f^+ и M_f^- двухточечного шаблона, подставляя записанные в ярлыках на текущей итерации данные C^k в заранее рассчитанный нами шаблон;
- на граничных гранях воспользуемся заданными краевыми условиями.

Коэффициенты двухточечного шаблона добавляются в соответствующие строки матрицы.

Затем сформированная таким образом линейная система решается посредством предлагаемого платформой класса Solver. После решения линейной системы, данные C^k в элементах заменяются на данные C^{k+1}, полученные в процессе решения.

Завершая процесс решения, мы можем сохранить данные посредством вызова функции Mesh::Save в один из предлагаемых платформой форматов (vtk, pvtk, gmv, pmf), чтобы либо впоследствии возобновить расчет, либо проанализировать полученные данные сторонними средствами визуализации.

Теперь перейдем к параллельному решению задачи.

В параллельном режиме после загрузки сетки может потребоваться разбить сетку или для повышения эффективности сбалансировать ее для заданного количества процессоров, что можно выполнить средствами платформы и сторонних пакетов ParMETIS, PT-Scotch или Zoltan. Для двухточечного шаблона потребуется один слой фиктивных ячеек, т. е. чтобы у каждой грани расчетной области данного процессора был сосед. Для вычисления нелинейных двухточечных шаблонов вокруг некоторых ячеек могут потребоваться дополнительные фиктивные ячейки, лежащие во втором и третьем слое фиктивных ячеек. Доступ к данным в таких дополнительных ячейках может быть обеспечен двумя способами: либо формированием двух-трех слоев фиктивных ячеек вдоль всей границы, либо формированием расширения одного слоя за счет присоединения дополнительных ячеек. Оба способа доступа к данным реализованы в программной платформе.

При заполнении матрицы системы будем перебирать только те грани, у которых есть хотя бы одна собственная или общая ячейка, и помещаем результат в вектор или матрицу в те позиции, которые соответствуют общим и собственным граням. Затем решаем систему линейных уравнений, помещаем полученные данные в элементы и даем запрос программной платформе на синхронизацию данных.

Далее итерации Пикара продолжаются до достижения указанного пользователем критерия остановки.

3.4. Использование функционала графической среды при создании пользовательского интерфейса программного комплекса GeRa

Общий вид пользовательского интерфейса показан на рис. 25. Интерфейс состоит из трех окон — окна для изображения трехмерных объектов, двумерного окна для задания

контуров и окна объектов модели, которое позволяет управлять видимостью объектов. При необходимости программа GeRa может скрывать 2D- или 3D-окно, если в нем нет объектов, тем самым увеличивая рабочее пространство для другого окна. Окно объектов модели присутствует в интерфейсе всегда.

Иерархически все три окна являются потомками центрального окна (класс QWidget), в котором установлен менеджер размещения QBoxLayout с горизонтальным размещением элементов. Менеджер размещения автоматически вычисляет размеры окон и выравнивает их в зависимости от имеющегося рабочего пространства. Окна для работы с 2D- и 3D-объектами являются экземплярами класса QVTKWidget — специального графического контейнера, предоставляемого библиотекой VTK. С одной стороны, QVTKWidget наследует все свойства QWidget и тем самым позволяет его включать в иерархию графических компонентов Qt. С другой стороны, он является окном рисования VTK, то есть позволяет использовать все возможности визуализации, имеющиеся в библиотеке VTK. Окно трехмерной визуализации имеет в левой нижней части экрана 3D-навигатор, который показывает ориентацию в пространстве геологической модели.

Дерево объектов в левой части экрана содержит все объекты, изображенные в 2D- и 3D-окне, и предназначено для управления видимостью объектов. Пользователь по желанию может отключить изображение тех объектов, с которыми он сейчас не работает.

Геологическая модель в программном комплексе GeRa задается с помощью дискретных поверхностей кровель и подошв, разломов и границ модели. Программа поддерживает как создание геологической модели с нуля, при котором определяются внешние границы модели и шаги геологической сетки (рис. 26 и 27), так и импорт геологической модели в различных форматах. Поверхности задаются

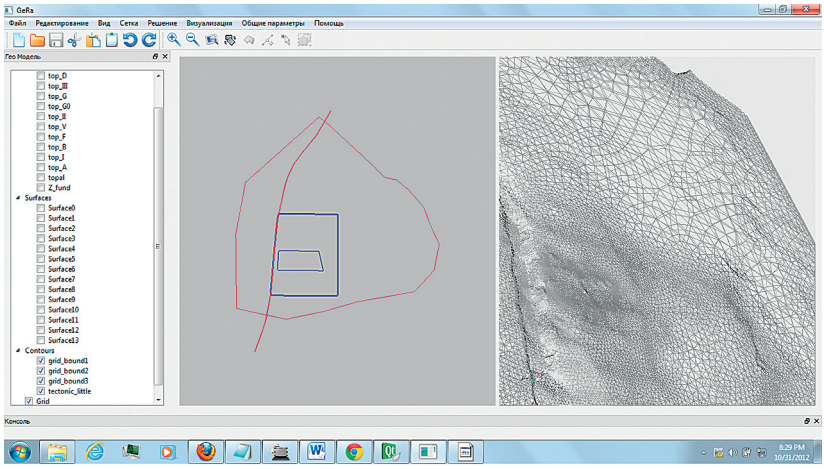


Рис. 25. Общий вид пользовательского интерфейса программного комплекса GeRa

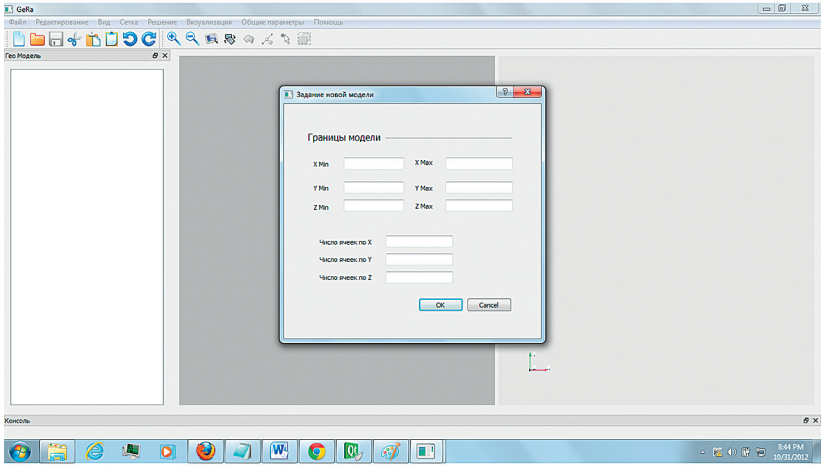


Рис. 26. Задание внешних границ модели и шага сетки

поверхностными триангуляциями. Если импортируемая модель содержит четырехугольную сетку, то внутренний алгоритм преобразует ее к триангуляции.

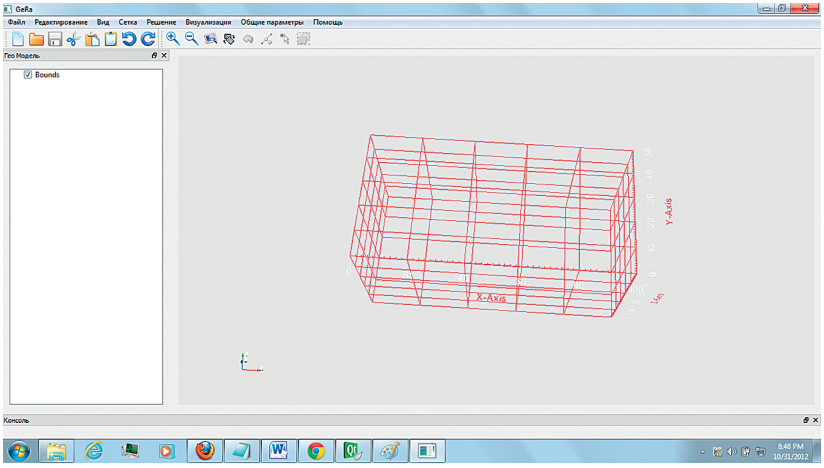


Рис. 27. Результат создания геологической сетки

Граничные условия можно задавать на боковых границах и на кровле и подошве расчетной области или на граничных гранях расчетной сетки. Типы граничных условий зависят от типа границы.

Геологическая модель в программе GeRa реализована в виде экземпляра класса grGeoModel, который является синглтоном (то есть существует в программе в единственном экземпляре). Класс grGeoModel предоставляет расчетным модулям данные об охватывающем параллелепипеде (метод GetBounds()), представляющем внешние границы модели, данные о геометрии слоев и поверхностей (методы GetLayer() и GetSurface()), а также данные о контурах областей в плане (на горизонтальной плоскости *xy*), в которых надо сгущать расчетную сетку.

Геометрические данные поверхностей кровель и подошв геологических слоев хранятся в модели в виде структур данных vtkPolyData. Данные генератора сетки и решателя хранятся в виде неструктурированных сеток vtkUnstructuredGrid, ссылки на которые также находятся в модели после работы соответствующих расчетных модулей.

Импорт поверхностей осуществляется в формате файлов *.xyz через пункт меню **Файл/Импортировать модель**. Границы модели в этом случае вычисляются автоматически и при необходимости могут быть изменены вручную (рис. 28).

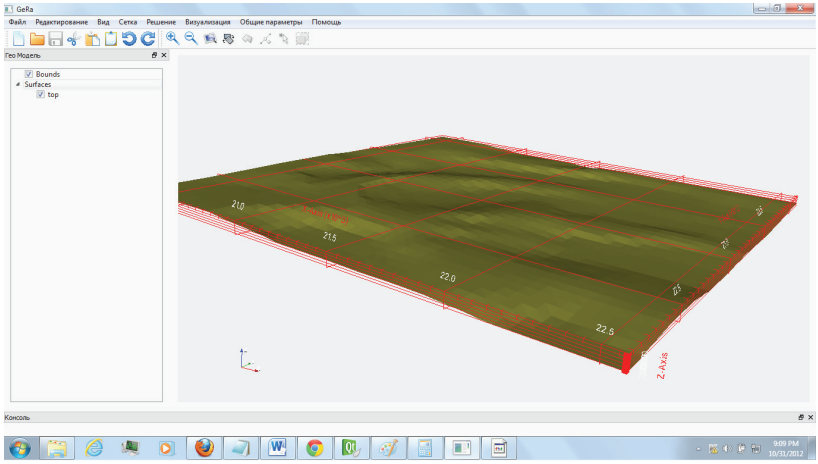


Рис. 28. Импорт поверхностей

Импорт контуров показан на рис. 29 и предназначен для задания границ расчетной области, определения линий разломов и определения областей сгущения расчетной сетки в плане (т. е. на горизонтальной плоскости *xy*). Контуры импортируются также через пункт меню **Файл/Импортировать модель.** Файлы импортируются в формате *.mif — MapInfo Interchange Format.

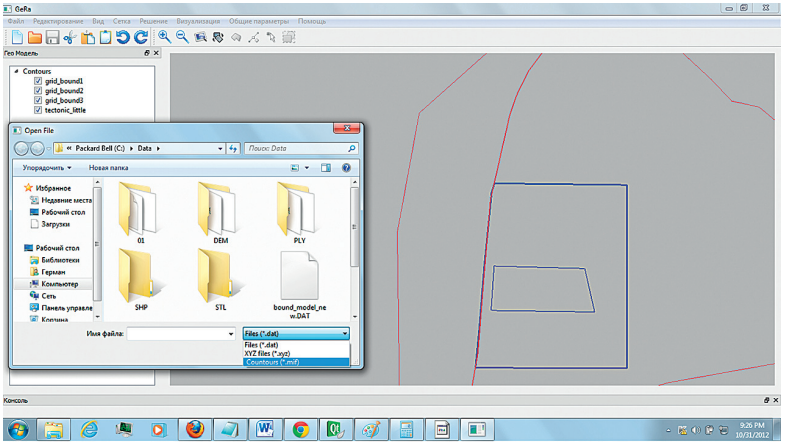


Рис. 29. Импорт контуров

Геологическая модель содержит описание геометрии поверхностей слоев и загружается через пункт меню **Файл/Импортировать модель** в виде *.dat файла. Результат работы импорта геологической модели показан на рис. 30.

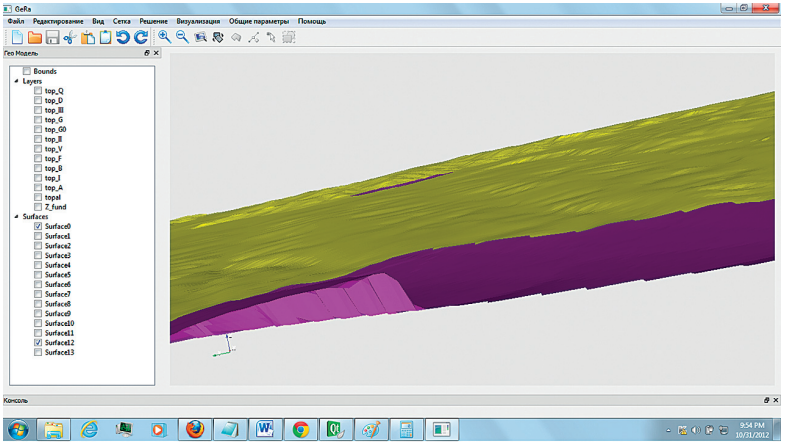


Рис. 30. Импорт геологической модели

Граничные условия на поверхностях задаются в виде условий Дирихле 1-го рода, условий Неймана 2-го рода, а также граничных условий 3-го рода и условий пользовательского типа. Для задания граничных условий надо выделить контур или сегмент контура в 3D-окне, либо ячейку граничной поверхности в 3D-окне и в появившемся диалоге задать соответствующие коэффициенты (рис. 31). Граничные грани можно задать выделением ячеек, которые удаляются из сетки, и их грани становятся граничными. Это удобно при моделировании скважин.

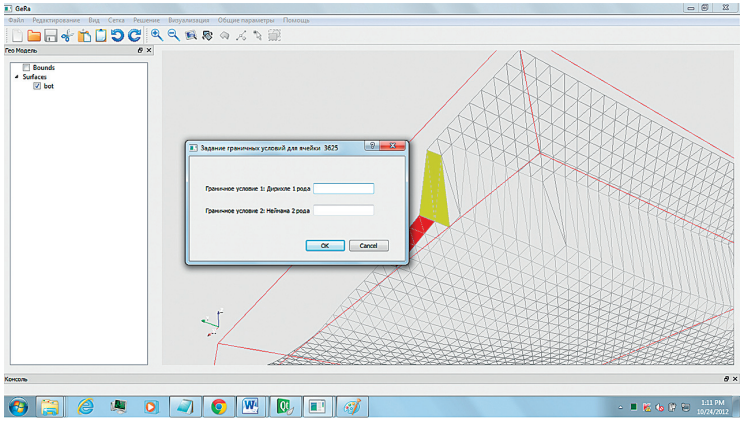


Рис. 31. Задание граничных условий на контурах и поверхностях

После создания геологической модели становится доступным пункт меню **Сетка/Запустить генератор сетки**. Сеточный генератор строит треугольно-призматическую сетку.

Запуск решателя осуществляется при выборе пункта меню **Решение/Дискретизация и решение**. На этом этапе происходит дискретизация задачи, формирование матрицы системы и ее правой части, а также решение полученной системы.

На этапе визуализации и анализа результатов можно изображать и изучать рассчитанные поля, например, скалярное

поле напора или векторное поле фильтрационных потоков. Активизируется эта функция в пункте меню **Визуализация/Показать решение**. При этом в панели инструментов появляется выпадающий список названий величин, которые были рассчитаны в модели, и по которым можно проводить анализ. Пункт меню **Визуализация/Анализ результатов** позволяет запустить внешнюю программу анализа решения (рис. 32, 33).

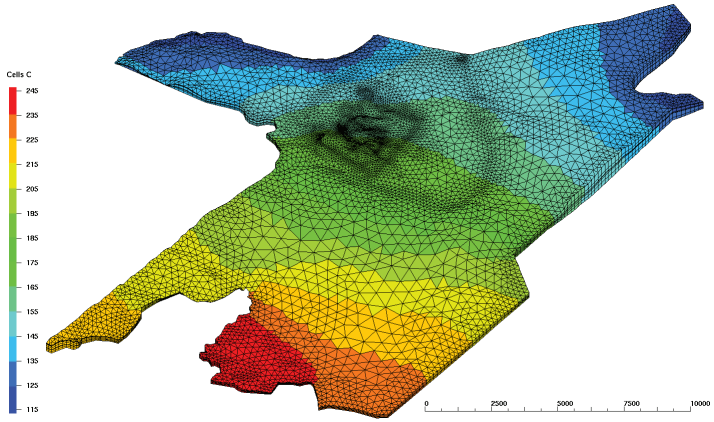


Рис. 32. Пример визуализации скалярного поля: напор в слое III

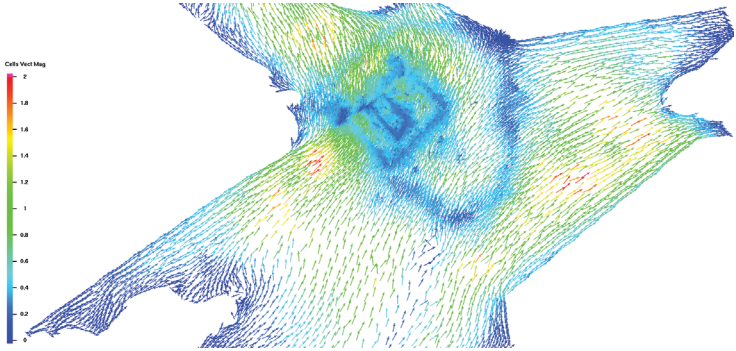


Рис. 33. Пример визуализации векторного поля: фильтрационные потоки в слое I

Список ссылок на источники

1. FMDB. URL: http://www.scorec.rpi.edu/FMDB/
2. MOAB. URL: http://trac.mcs.anl.gov/projects/ITAPS/wiki/MOAB
3. MSTK. URL: https://software.lanl.gov/MeshTools/trac
4. STK. URL: http://trilinos.sandia.gov/packages/stk/
5. Salome. URL: http://www.salome-platform.org/
6. OpenFOAM. URL: http://www.openfoam.com/
7. ParMETIS. URL: http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview
8. METIS. URL: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
9. Scotch & PT-Scotch. URL: http://www.labri.fr/perso/pelegrin/scotch/
10. PT-Scotch 5.1 User's guide. Technical report, LaBRI, 2008. F. Pellegrini. URL: https://gforge.inria.fr/docman/view.php/248/7103/ptscotch_user5.1.pdf
11. Zoltan. URL: http://www.cs.sandia.gov/Zoltan/Zoltan.html
12. Trilinos. URL: http://trilinos.sandia.gov/
13. *Hestenes, Magnus R.; Stiefel, Eduard.* Methods of Conjugate Gradients for Solving Linear Systems // J. Research National Bureau Standards. 1952. V. 49. N. 6. P. 409–436.
14. *Van der Vorst*, H.A. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear System // SIAM J. Sci. Stat. Comput. 1992. V. 13. N. 2. P. 631–644.
15. *Saad Y., Schultz M.H.*, GMRES: A Generalized Minimal Residual Method for Solving nonsymmetric Linear Systems // SIAM J. Sci. Stat. Comput. 1986. V. 7. N. 3. P. 856–869.

16. *Saad Y.* Iterative Methods for Sparse Linear Systems. 2nd edition. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2003.
17. PETSc. URL: http://www.mcs.anl.gov/petsc/
18. *Barney B.* POSIX Threads Programming. Lawrence Livermore National Laboratory. URL: https://computing.llnl.gov/tutorials/pthreads/
19. NVIDIA. URL: http://www.nvidia.com/page/home.html
20. PLAPACK. URL: http://www.cs.utexas.edu/users/plapack/
21. SuperLU. URL: http://crd.lbl.gov/~xiaoye/SuperLU/
22. WSMP. URL: http://researcher.watson.ibm.com/researcher/view_project.php?id=1426
23. MUMPS. URL: http://mumps.enseeiht.fr/
24. UMFPACK. URL: http://www.cise.ufl.edu/research/sparse/umfpack/
25. Doxygen. URL: http://www.stack.nl/~dimitri/doxygen/
26. Ani3D. URL: http://sourceforge.net/projects/ani3d/
27. *Kaporin I.E.* High quality preconditionings of a general symmetric positive definite matrix based on its $U^TU + U^TR + R^TU$-decomposition // Numer. Linear Algebra Applic. 1998. V. 5. P. 483–509.
28. *Капорин И.Е., Коньшин И.Н.* Параллельное решение симметричных положительно-определенных систем на основе перекрывающегося разбиения на блоки // Ж. Вычисл. Матем. и Матем. Физ. 2001. Т. 41. № 4. С. 515–528.
29. *Капорин И.Е., Коньшин И.Н.* Постфильтрация множителей IC2 разложения для балансировки параллельного предобусловливания // Ж. Вычисл. Матем. и Матем. Физ. 2009. Т. 49. № 6. С. 940–957.
30. OpenDX. URL: http://www.opendx.org/
31. Visualization Library. URL: http://www.visualizationlibrary.org
32. VTK. URL: http://www.vtk.org/
33. ParaView. URL: http://www.paraview.org/

34. ParaView User's Guide v. 3.14, 2012.
35. VisIt. URL: https://wci.llnl.gov/codes/visit/
36. Qt. URL: http://qt.digia.com/
37. *Шлее М.* Qt4.5. Профессиональное программирование на C++. СП.: БХВ-Петербург, 2010. 896 с.
38. *Schroeder W., Martin K., Lorensen B.* The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, 3rd edition. 2002. Kitware, Inc.
39. The Visualization Toolkit User's Guide 4.0. 2001 Kitware, Inc.
40. *Lorensen W.E., Cline H.E.* Marching Cubes: A high resolution 3D surface construction algorithm // Comput. Graphics. 1987. V. 21. N. 4.
41. *Globus A., Levit C., Lasinski T.* A tool for visualizing the topology of three-dimensional vector fields // Proc. of Visualization'91. IEEE Comput. Society Press, Los Alamos, CA, 1991. P. 33–40.
42. *Schroeder W., Volpe C., and Lorensen W.* The stream polygon: A technique for 3D vector field visualization // Proc. of Visualization'91. IEEE Comput. Society Press, Los Alamos, CA, 1991. P. 126–132.
43. *Tricoche X., Garth C.* Topological methods for visualizing vortical flows // Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration, Ed. by Möller T., Hamann B., Russell R., Springer-Verlag, 2009. P. 89–108.
44. *Hellman J., Hesselink L.* Visualization of vector field topology in fluid flows // IEEE Comp. Graph. Appl. 1991. V. 11. N. 3. P. 36–46.
45. *Delmarcelle T., Hesselink L.* Visualizing second-order tensor fields with hyperstreamlines // IEEE Comp. Graph. Appl. 1993. V. 13. N. 4. P. 25–33.
46. *Schroeder W., Volpe C., Lorensen W.* Decimation of triangle meshes // Comp. Graphics (SIGGRAPH'92). 1992. V. 26. N. 2. P. 65–70.

47. *Nikitin K., Olshanskii M., Terekhov K., Vassilevski Yu.* A numerical method for the simulation of free surface flows of viscoplastic fluid in 3D // J. Comp. Math. 2011. V. 29. N. 6. P. 605–622.
48. *Vassilevski Yu.V., Nikitin K.D., Olshanskii M.A., Terekhov K.M.* CFD technology for 3D simulation of large-scale hydrodynamic events and disasters // Russ. J. Numer. Anal. Math. Modelling. 2012. V. 27. N. 4. P. 399–412.
49. *Lipnikov K., Svyatskiy D., Vassilevski Yu.* Minimal stencil finite volume scheme with the discrete maximum principle // Russ. J. Numer. Anal. Math. Modelling. 2012. V. 27. N. 4, P. 369–385.
50. *Danilov A., Vassilevski Yu.* A monotone nonlinear finite volume method for diffusion equations on conformal polyhedral meshes // Russ. J. Numer. Anal. Math. Modelling. 2009. V. 24. N. 3. P. 207–227.
51. *Nikitin K., Vassilevski Yu.* A monotone nonlinear finite volume method for advection–diffusion equations on unstructured polyhedral meshes in 3D // Russ. J. Numer. Anal. Math. Modelling. 2010. V. 25. N. 4. P. 335–358.
52. *Lipnikov K., Svyatskiy D., Vassilevski Yu.* A monotone finite volume method for advection-diffusion equations on unstructured polygonal meshes // J. Comp. Phys. 2010. V. 229. P. 4017–4032.
53. *Lipnikov K., Svyatskiy D., Vassilevski Yu.* Interpolation-free monotone finite volume method for diffusion equations on polygonal meshes // J. Comp. Phys. 2009. V. 228. N. 3. P. 703–716.
54. *Марчук Г.И., Кузнецов Ю.А.* К вопросу об оптимальных итерационных процессах // ДАН СССР. 1968. Т. 181. № 6. С. 1331–1334.
55. *Василевский Ю В., Данилов А.А., Николаев Д.В., Руднев С.Г., Саламатова В.Ю., Смирнов А.В.* Конечно элементный анализ задач биоимпедансной диагностики //

Ж. Вычисл. Матем. и Матем. Физ. 2012. Т. 52. № 4. С. 733–745.
56. *Василевский Ю.В., Данилов А.А., Липников К.Н., Чугунов В.Н.* Автоматизированные технологии построения неструктурированных расчетных сеток. – М.: Физматлит, 2013.
57. PARDISO. URL: http://www.pardiso-project.org/
58. *Terekhov K.M., Volodin E.M., and Gusev A.V.* Methods and efficiency estimation of parallel implementation of the s-model of general ocean circulation // Russ. J. Numer. Anal. Math. Modelling. 2011. V. 26. N. 2. P. 189–208.
59. GMV. URL: http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html
60. Techplot. URL: http://www.tecplot.com/
61. Avizo. URL: http://www.vsg3d.com
62. Scientific VR. URL: http://www.cfd.ru/russian/products/Mirage/mir_frm_main_inf.htm
63. Научная визуализация. Электронное издание. URL: http://sv-journal.com/index.php?lang=ru
64. *Васев П.А., Кумков С.С., Шмаков Е.Ю.* Среда-конструктор систем научной визуализации // XIV Международная конференция: Супервычисления и математическое моделирование. Саров, 2012. РФЯЦ ВНИИЭФ. 2012. С. 51–52.
65. SharpEye. URL: http://www.sharpeye.lact.ru
66. *Потехин А.Л., Никитин В.А., Логинов И.В. и др.* Пакет программ Логос. Новые возможности графической постобработки результатов моделирования инженерных задач в параллельной системе ScientificView // XIV Международная конференция: Супервычисления и математическое моделирование. Саров, 2012. РФЯЦ ВНИИЭФ. 2012. С. 134–135.
67. *Потехин А.Л., Логинов И.В., Тарасов В.И. и др.* ScientificView — параллельная система постобработки результатов, полученных при численном моделировании

физических процессов // Вопросы Атомной Науки и Техники. 2008. Вып. 4. С. 37–45.
68. SIAM Conferences. URL: http://siam.org/meetings/archives.php#PP
69. OpenFOAM Conferences. URL: http://www.extend-project.de/openfoam-workshop
70. Международная научная конференция: «Научный сервис в сети Интернет». Абрау-Дюрсо, 1999–2012.
71. Международная научная конференция: «Параллельные вычислительные технологии» (ПаВТ). 2007–2012.
72. Международная научная конференция: «Супервычисления и математическое моделирование». Саров, РФЯЦ ВНИИЭФ, 1999–2012.
73. URL: http://www.parallel.ru
74. MPI. URL: http://www.mpi-forum.org/
75. OpenMP. URL: http://www.open-mpi.org/
76. *Frey P.J., George P.L.* Mesh generation: application to finite elements. — Oxford, Hermes Sci. Publishing, 2000.
77. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. — СПб: БХВ-Петербург. 2002. 602 с.
78. *Foster I.* Designing and Building Parallel Programs. Concept and Tools for Parallel Software Engineering. Addisson-Wesley, NewYork – Amsterdam – Paris. 1995, 381 p. URL: http://www.mcs.anl.gov/~itf/dbpp/
79. *Толстых М.А., Ибраев Р.А., Володин Е.М. и др.* Модели глобальной атмосферы и Мирового океана: алгоритмы и суперкомпьютерные вычислительные технологии. — М.: Издательство Московского университета, 2012. 144 с.
80. *Замарашкин Н.Л.* Алгоритмы для разреженных систем линейных уравнений над GF(2). — М.: Издательство Московского университета, 2012. 136 с.

Предметный указатель